Милош Раденковић • Маријана Деспотовић-Зракић • Зорица Богдановић
Душан Бараћ • Александра Лабус • Милица Симић

РАЗВОЈ СОФТВЕРА ОРИЈЕНТИСАНОГ НА ПРОЦЕСЕ

ОДАБРАНА ПОГЛАВЉА
DEVOPS МИКРОСЕРВИСИ КОНТЕЈНЕРИ

II ИЗМЕЊЕНО И ДОПУЊЕНО ИЗДАЊЕ

Београд 2024

Рачунарски факултет
Факултет организационих наука

Милош Раденковић, Маријана Деспотовић-Зракић, Зорица Богдановић

Душан Барић, Александра Лабус, Милица Симић

РАЗВОЈ СОФТВЕРА ОРИЈЕНТИСАНОГ НА ПРОЦЕСЕ

ОДАБРАНА ПОГЛАВЉА

DEVOPS МИКРОСЕРВИСИ КОНТЕЈНЕРИ

II ИЗМЕЊЕНО И ДОПУЊЕНО ИЗДАЊЕ

Београд 2024

РАЗВОЈ СОФТВЕРА ОРИЈЕНТИСАНОГ НА ПРОЦЕСЕ

ОДАБРАНА ПОГЛАВЉА

DEVOPS МИКРОСЕРВИСИ КОНТЕЈНЕРИ

Милош Раденковић, Маријана Деспотовић-Зракић, Зорица Богдановић,

Душан Бараћ, Александра Лабус, Милица Симић

Рецензенти:
проф. др Саша Лазаревић

проф. др Бојана Димић Сурла

Издавачи:
Рачунарски факултет, Београд, Кнеза Михаила 6/VI

Факултет организационих наука, Београд, Јове Илића 154

Лектор:
Лаура Барна

Корице:
Марко Савићевић

Техничка обрада:
Милица Симић

Штампа:
Newpress, Смедерево

Тираж: 30

CIP - Каталогизација у публикацији Народна библиотека Србије, Београд

004.42:004.9(075.8)(0.034.2)

РАЗВОЈ софтвера оријентисаног на процесе [Електронски извор] :
одабрана поглавља : DevOps, микросервиси, контејнери / Милош Раденковић
... [и др.]. - Београд : Рачунарски факултет : Факултет организационих наука,
2024 (Смедерево : Newpress). - 1 електронски оптички диск (CD-ROM) ; 12 cm

Системски захтеви: Нису наведени. - Насл. са насловне стране документа. -
Тираж 30. - Библиографија уз свако поглавље.

ISBN 978-86-7680-458-0 (ФОН)

1. Раденковић, Милош Б., 1991- [autor]

а) Апликативни софтвер -- Пројектовање

COBISS.SR-ID 146275081

Ова књига је одобрена за употребу у настави као основни уџбеник на Факултету организационих наука Универзитета у Београду и на Рачунарском факултету Унион универзитета.

Аутори задржавају сва права. Репродукција појединих делова или целине ове публикације није дозвољена.

Електронска верзија овог уџбеника лиценцирана је под лиценцом *Creative Commons* Ауторство - Некомерцијално-Без прерада - *Attribution-NonCommercial-NoDerivs* 4.0.

Захваљујемо проф.др Божидару Раденковићу
на визији и креативности које су обликовале садржај ове књиге,
на знању и искуству које је несебично поделио са нама.

Аутори

САДРЖАЈ

I УВОД 1

1. Софтверско инжењерство 1

2. Парадигме развоја софтвера 4

3. Парадигма развоја процедуралног софтвера 6

4. Парадигма развоја објектно-оријентисаног софтвера 8

5. Парадигма развоја софтвера оријентисаног на процесе 12

5.1 Историја и еволуција метода развоја софтвера оријентисаног на процесе 15

5.1.1 Развој софтвера вођен подацима 16

5.1.2 Развој софтвера вођен моделом 20

5.1.3 Фабрике софтвера 23

II УПРАВЉАЊЕ СОФТВЕРСКИМ ПРОЦЕСОМ 27

6. Модел водопада 28

7. Спирални модел 29

8. Итеративни модел 31

9. Агилни модели и Скрам 32

10. DevOps 35

10.1 Планирање (Plan) 39

10.2 Кодирање (Code) 40

10.3 Конструкција софтвера (Build) 41

10.4 Тестирање софтвера (Test) 41

10.5 Издавање нове верзије софтвера (Release) 41

10.6 Постављање софтвера у продукционо окружење (Deploy) 42

10.7 Експлоатација и оперативни рад софтвера (Operate) 43

10.8 Надгледање и праћење (Monitor) 43

11. Евалуација софтверског процеса 44

12. Отворене иновације и софтвер отвореног кôда у DevOps-у 44

III АНАЛИЗА И ПРОЈЕКТОВАЊЕ 46

13. Прикупљање захтева и анализа пословног система 47

13.1 Преглед традиционалних техника за анализу захтева 48

13.2 Анализа захтева помоћу UML-а 49

13.3 Моделовање пословних процеса и BPMN 51

13.4 Извршна спецификација пословних процеса 56

14. Пројектовање софтвера 57

14.1 Принципи пројектовања софтвера 59

14.2 Апстракција и модели у пројектовању софтвера 61

14.3 Процес пројектовања софтвера 65

15. Модуларност 67

15.1 Организација модула 70

15.2 Повезивање и комуникација модула ... 71
15.2.1 Спрега између модула ... 72
15.2.2 Кохезија ... 74
15.2.3 Раздвајање надлежности ... 75
15.2.4 Инверзија зависности ... 76
15.2.5 Уметање зависности ... 77
16. Интеграција ... 79
16.1 Front-end интеграција ... 80
16.2 Интеграција на апликационом слоју ... 80
16.3 Интеграција података ... 81
17. Архитектура софтвера ... 82
17.1 Монолитна архитектура ... 84
17.2 Сервисно-оријентисана архитектура ... 85
17.3 Микросервисна архитектура ... 87
17.4 Serverless архитектура ... 88
17.5 Архитектура софтвера заснованог на процесима ... 88
17.6 Концепти дефинисања архитектуре софтвера ... 89
18. Пројектовање архитектуре софтвера ... 95
19. Патерни архитектуре софтвера ... 98
19.1 Антипатерни ... 100
19.2 Архитектурни стилови у пројектовању софтвера ... 101
19.3 Таксономија патерна и архитектурних стилова ... 101
19.4 Основни архитектурни патерни ... 104
19.4.1 Патерн клијент-сервер архитектуре ... 104
19.4.2 Патерни вишеслојне архитектуре ... 105
19.4.3 Патерн master-worker ... 106
19.4.4 Патерн peer-to-peer архитектуре ... 107
19.4.5 Патерн посредника – брокер архитектура ... 108
19.4.6 Патерн виртуелизованог средњег слоја ... 109
19.4.7 Патерн филтрирања тока података ... 110
19.4.8 Патерн архитектуре у форми микрокернела ... 111
19.5 Патерни пројектовања корисничког интерфејса ... 112
19.5.1 Model View Controller ... 113
19.5.2 Model View Presenter ... 114
19.5.3 Model View View Model ... 115
20. Патерни архитектуре оријентисане на процесе ... 116
20.1 Доменом вођено пројектовање софтвера ... 117
20.1.1 Домен ... 117
20.1.2 Речник термина у оквиру домена ... 118
20.1.3 Повезан контекст ... 119
20.1.4 Доменски модел ... 119
20.1.5 Употреба доменског модела ... 121
20.2 Концепти моделовања софтвера оријентисаног на процесе ... 122
20.2.1 Агрегације у моделима софтвера оријентисаног на процесе ... 122

20.2.2 Догађај ... *123*
20.2.3 Сервис ... *125*
20.2.4 Процес ... *126*
20.2.5 Интеракција процеса – апликација ... *126*
20.2.6 Информациони систем ... *127*
20.2.7 B2B мреже ... *127*
20.3 Информациони модел софтвера оријентисаног на процесе ... *127*
20.3.1 Метаподаци и подаци догађаја ... *128*
20.3.2 Метаподаци и подаци процеса ... *128*
20.4 Патерни складиштења догађаја ... *129*
20.4.1 Event store ... *132*
20.4.2 Информационо језеро ... *133*
20.5 Патерни обраде догађаја ... *135*
20.5.1 Патерн основне обраде догађаја ... *137*
20.5.2 Комуникација и синхронизација процеса – Pub/Sub патерн ... *139*
20.5.3 Обрада токова догађаја ... *142*
20.5.4 Обрада комплексних догађаја ... *144*
20.5.5 Lambda и Карра архитектуре ... *145*
21. Микросервиси ... 147
21.1 Комуникација микросервиса са окружењем ... *148*
21.2 Апликациона логика микросервиса ... *149*
21.3 Типови микросервиса ... *150*
21.4 Примена микросервиса у развоју софтвера оријентисаног на процесе ... *153*
21.5 Кореографија и оркестрација микросервиса ... *154*
21.6 Управљање подацима у микросервисним архитектурама ... *157*
21.6.1 CAP теорема ... *158*
21.7 Управљање трансакцијама у микросервисним архитектурама ... *160*
21.7.1 Двофазни протокол извршавања трансакција – 2PC ... *160*
21.7.2 Сага патерн ... *161*
21.8 Патерни постављања микросервиса у продукционо окружење ... *163*

IV ИМПЛЕМЕНТАЦИЈА ... 167
22. Апликациони програмски интерфејси ... 168
22.1 Пројектовање API ... *169*
22.1.1 Патерни архитектуре API ... *170*
22.1.2 API архитектурни стилови и протоколи ... *173*
22.1.3 SOAP и REST у сервисно-оријентисаној архитектури ... *175*
22.2 REST API ... *177*
22.2.1 Препоруке и ограничења за REST API архитектуру ... *178*
22.2.2 Елементи REST API архитектуре ... *180*
22.3 Сигурност API ... *181*
22.4 Имплементација API ... *182*
22.4.1 Code first и API first приступи имплементацији ... *185*
22.5 Тестирање API ... *185*

22.6 Модели зрелости API архитектура *188*

23. Патерни интеграције **190**

23.1 Агрегатор *192*

23.2 Bulkhead *192*

23.3 Sidecar *193*

23.4 Прокси и реверзни прокси *194*

23.5 API прокси *195*

23.6 API гејтвеј *197*

23.7 Откривање микросервиса *199*

23.8 Обезбеђење паралелизације скалабилности и поузданости *200*

24. Софтверски оквири, платформе и екосистеми **203**

24.1 Софтверски оквири *203*

24.1.1 Врсте оквира *205*

24.2 Софтверске платформе *206*

24.2.1 Врсте софтверских платформи *207*

24.2.2 Low-code софтверске платформе *209*

24.3 Софтверски екосистеми *210*

24.3.1 Примери софтверских екосистема *212*

25. Континуална интеграција, испорука и постављање софтвера у продукционо окружење **213**

25.1 Увод и дефиниције *213*

25.2 Континуална интеграција *215*

25.3 Интеграциона платформа као сервис *218*

25.4 Патерни постављања софтвера у продукционо окружење *218*

V ИНФРАСТРУКТУРА **220**

26. Виртуелизација **222**

26.1 Еволуција и историја развоја виртуелизације *222*

26.2 Типови виртуелизације *225*

26.2.1 Потпуна виртуелизација *226*

26.2.2 Виртуелизација подржана кернелом *226*

26.2.3 Виртуелизација подржана хардвером *227*

26.2.4 Паравиртуелизација *227*

26.3 Виртуелизација рачунарских мрежа *227*

26.4 Виртуелизација складиштења података *229*

26.5 Избор хипервизора и типа виртуелизације *230*

27. Контејнери **230**

27.1 Концепти Linux кернела у контејнерској виртуелизацији *231*

27.1.1 Linux процеси *232*

27.1.2 Креирање процеса *232*

27.1.3 Организација процеса *233*

27.1.4 Међупроцесна комуникација *233*

27.1.5 Контрола и ограничавање ресурса – контролне групе (cgroups) *234*

27.1.6 Изолација ресурса – области именовања (namespaces) *235*

27.2 Екосистем контејнера ... *239*
27.3 Животни циклус контејнера ... *241*
27.4 Умрежавање контејнера ... *243*
27.5 Фајл системи контејнера ... *244*
27.6 Управљање волуменима у контејнерском окружењу ... *245*
27.7 Регистар контејнера ... *246*
27.8 Стандардизација контејнерског екосистема ... *248*
27.8.1 OCI Image-spec ... *248*
27.8.2 OCI Runtime-spec ... *250*
27.9 Container runtime и container engine ... *251*
27.10 Cloud native приступ развоју софтвера ... *252*
27.11 Оркестрација контејнера ... *253*
27.11.1 Single host оркестрација – Docker Compose ... *254*
27.11.2 Multi-node оркестрација ... *254*
28. Kubernetes ... 256
28.1 Kubernetes архитектура ... *256*
28.1.1 API сервер ... *257*
28.1.2 Etcd складиште ... *258*
28.1.3 Scheduler ... *258*
28.1.4 Controller Manager ... *258*
28.1.5 Kubelet ... *258*
28.1.6 Kube-proxy ... *259*
28.1.7 Container runtime ... *259*
28.1.8 Интеракција и синхронизација компоненти Kubernetes-а ... *259*
28.2 Пôд ... *261*
28.3 Контролери ... *262*
28.3.1 ReplicaSet ... *262*
28.3.2 Deployment Controller ... *263*
28.3.3 StatefulSet ... *263*
28.3.4 Перзистентно складиштење података у Kubernetes-у ... *263*
28.4 Умрежавање у Kubernetes-у ... *265*
28.4.1 Kubernetes мрежни сервиси апликационог нивоа ... *266*
28.4.2 Kubernetes Ingress ресурси и мрежни сервиси ... *267*
28.5 Функционалности Kubernetes кластера ... *269*
28.5.1 Генерисање и распоређивање пôдова ... *270*
28.5.2 Балансирање оптерећења у оквиру кластера ... *271*
28.5.3 Скалирање перформанси ... *271*
28.5.4 Обрада непланираних прекида и опоравак ... *272*
28.5.5 Мониторинг и логови ... *272*
28.5.6 Аутоматско скалирање ... *273*
28.5.7 Service Mesh у Kubernetes-у ... *274*
28.5.8 Безбедност ... *275*
28.5.9 CI/CD ... *277*
28.5.10 Helm Charts ... *278*

ПРЕДГОВОР

Предмет анализе и излагања градива у овом уџбенику је развој софтвера оријентисаног на процесе. Градиво укључује теоријске основе, методе које се користе у пројектовању софтвера, имплементациона окружења, као и алате и методе за управљање развојем софтвера. Циљ је да се у оквиру књиге представе најзначајнији процеси софтверског инжењерства и њихова примена, почев од корисничких захтева који се данас постављају пред информационе системе великих предузећа: дистрибуираност, линеарна скалабилност и поузданост. Затим, преглед теоријских концепата који то омогућују: паралелни процеси, сервиси, догађаји и методе за њихово управљање, као и методе за управљање софтверским процесом, где је посебна пажња посвећена *DevOps*-у. Теоријске основе парадигме развоја софтвера оријентисаног на процесе, каква је у употреби данас, у литератури су дате у фрагментима и с главним референцама које су објављиване у протеклих неколико деценија, тако да данас није могуће у једној референци наћи комплетан и конзистентан приказ ове парадигме. У теоријским радовима, а нарочито у методама и алатима, мешају се концепти објектно и процесно оријентисане парадигме, истим концептима у различитим методама и софтверским алатима дају се другачија имена, с настојањима да се из разлике у синтакси дефинише нова семантика, иако се ради о синонимима. Овај уџбеник је покушај да се учини макар мали допринос уклањању нејасноћа, обједињавањем различитих теоријских приступа са јасном дистинкцијом између различитих парадигми. Наравно, различите парадигме развоја софтвера могу се комбиновати у истом пројекту или софтверском алату. Знање о предностима и недостацима сваке од парадигми може допринети квалитетнијем софтверском решењу.

С обзиром на то да је софтверско инжењерство у највећој мери окренуто пракси и конкретној примени, део ове књиге посвећен је имплементационим окружењима и алатима за управљање животним циклусом контејнера као и њиховом оркестрацијом.

Уџбеник и практикум лабораторијских вежби, који је у припреми, чине целину наставних материјала за савлађивање изложеног градива.

Линкови дати у фуснотама су били валидни на дан 20. маја 2024.

Аутори

I УВОД

1. СОФТВЕРСКО ИНЖЕЊЕРСТВО

Софтверско инжењерство (*Software Engineering*) као засебна инжењерска дисциплина настало је почетком шездесетих година прошлог века, издвајањем из Рачунарских наука (*Computer Science – CS*) и Информационих технологија (*Information technology – IT*). Настало је у тежњи да се на систематичан начин, уважавајући и користећи достигнућа, искуства и методе вишевековне инжењерске праксе у другим инжењерским дисциплинама[1], нађе прави одговор на дугогодишњи период „прве софтверске кризе“[2].

Као први траг у коришћењу термина „софтверско инжењерство“, у литератури се наводи Маргарет Хамилтон, која је у периоду 1963–1964. као водећи програмер на *Skylab* и *Apollo* програмима користила тај термин да би у стручним дискусијама нагласила разлику између природе свог посла и хардверског инжењерства[3]. Први оглас за посао „софтверског инжењера“ објављен је јуна 1965. у часопису *Computers and Automation*[4]. Термин „софтверско инжењерство“ у стручној литератури први пут је употребио председник *ACM*, 1966. године[5].

IEEE (*The Institute of Electrical and Electronics Engineers*) дефинише софтверско инжењерство као „систематску примену квантитативног приступа структурама, системима, уређајима, производима и процесима у развоју, раду и одржавању софтвера“[6].

---

[1] Wirth, N. (2008). A brief history of software engineering. *IEEE Annals of the History of Computing*, 30(3), 32–39.
[2] Valdez, M. E. P. (1989). *A gift from Pandora's box: The software crisis* (докторска дисертација), University of Edinburgh (United Kingdom).
[3] Hamilton, M. H. & Hackler, W. R. (2008). Universal systems language: lessons learned from Apollo. *Computer*, 41(12), 34–43.
[4] *Computers and Automation*. New York, New York: Edmund Berkeley and Associates, June 1965.
[5] Oettinger, A. G. (1966). President's Letter to the ACM Membership. *Communications of the ACM*, 9(12), 838–839.
[6] IEEE Standard Glossary of Software Engineering Terminology, in: *IEEE Std 610.12-1990*, vol., no., pp.1–84, 31 Dec. 1990, doi: 10.1109/IEEESTD.1990.101064.

Од почетка развоја дисциплине, па све до деведесетих година прошлог века, у стручној литератури представљан је модел софтверског инжењерства хијерархијском структуром принципа, метода, техника и алата који се користе у животном циклусу развоја и експлоатације софтвера (Слика 1).

Слика 1. Модел софтверског инжењерства – на почетку развоја дисциплине[7]

Дефиниције слојева овог модела су:

Принципи: Принципи у софтверском инжењерству примењују се на софтверске производе, послове и поступке у развоју и експлоатацији софтвера. Принципи одређују начин практичне и конкретне примене изабраних метода и техника у развоју и експлоатацији софтвера. Поред принципа који су одређени корисничким захтевима и специфичностима изабране парадигме развоја софтвера, скуп уобичајених принципа који се примењују у софтверском инжењерству су: ригорозност и формализам, модуларност, хоризонтална и вертикална декомпозиција сложених система, апстракција, генерализација, антиципација промена и инкрементaлност. У складу са усвојеним принципима бирају се методи, технике и алати. У овом слоју се дефинише и метод за управљање развојем софтвера који у највећој мери одређује послове и начин реализације управљања животним циклусом развоја софтвера (*Software development lifecycle – SDLC*)[8].

Методи: Метод представља општеприхваћен или стандардизован, ригорозан, систематски и дисциплинован приступ за ефикасно извршење задате активности у софтверском инжењерству. Методи софтверског инжењерства представљају техничка упутства за реализацију софтвера. Методи се састоје од препорука или стандарда за спецификацију корисничких захтева, анализу, пројектовање, конструкцију софтвера, тестирање и одржавање.

Технике: Технике имају више имплементационих и техничких детаља у односу

[7] Прилагођено према: Ghezzi, C., Jazayeri, M. & Mandrioli, D. *Fundamentals of Software Engineering* (2nd ed. 2003).
[8] Murch, R. (2012). *The Software Development Lifecycle-A Complete Guide.* Richard Murch.

на методе. Обично имају мању могућност примене. Технике и методи се могу међусобно мењати при извршавању одређене активности. Скуп метода и техника за решавање неког проблема формира методологију.

Методологија: Предлагање или стандардизација скупа изабраних метода и техника за решавање неког проблема.

Почетком осамдесетих година прошлог века, појавом глобализације и глобалне конкуренције, пред предузећа су постављани строжији услови за повећање ефикасности и продуктивности. Одговор на нове захтеве је, пре свега, тражен у промени модела организације предузећа. Са дотадашње функционалне организације прешло се на процесну. Процесни приступ организацији предузећа је ефикаснији и с временом је постао глобалан и доминантан[9]. То је довело и до схватања нужности редефинисања модела софтверског инжењерства. При редефинисању софтверског инжењерства, процеси су стављени у први план. Широко, а понекад и недовољно јасно дефинисани „принципи“ софтверског инжењерства (Слика 1) замењени су Софтверским процесом[10,11]. Софтверски процес је подељен у две групе процеса: *IT* операције и Развој софтвера[12] (Слика 2).

Слика 2. Редефинисан модел софтверског инжењерства

Као синоним за софтверски процес, у литератури се раније користио термин *Software Development Lifecycle – SDLC*. У новијим референцама, почев од спиралног

[9] Magretta, J. (2002). Why business models matter. *Harvard Business Review.* Harvard Business School Publishing Corporation.
[10] ISO. Standard for Information Technology – Software Life Cycle Processes. ISO/IEC 12207, 1995.
[11] Boehm, B. (2006, May). A view of 20th and 21st century software engineering. In: *Proceedings of the 28th international conference on Software engineering* (pp. 12–29).
[12] Erich, F., Amrit, C. & Daneva, M. (2014). Cooperation between software development and operations: a literature review. In: *The 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.* ACM, Torino.

модела (1986)[13], у складу са редефинисаним моделом, у већој мери се користи термин „софтверски процес“ (*Software Process*). За подпроцесе софтверског процеса, у редефинисаном моделу су уведени индикатори перформанси и индикатори квалитета. Управљање квалитетом је постављено као кључан процес у софтверском инжењерству[14].

Коректна и корисна систематизација знања и преглед метода у софтверском инжењерству може се наћи у *Software Engineering Body of Knowledge* (*SWEBOK*)[15].

Развој софтверског инжењерства очекиван је у редефинисању софтверског процеса ка масовности производње квалитетног софтвера, већој примени апстракција, машинског учења, вештачке интелигенције и аутоматског генерисања кôда[16]. У многим областима пословања постоји јака потреба за интеграцијом и усклађивањем *IT* операција и пословања предузећа (енгл. *IT-Business Alignment*)[17]. Будући развој софтверског инжењерства свакако треба и у овој области да понуди адекватне одговоре.

2. ПАРАДИГМЕ РАЗВОЈА СОФТВЕРА

Софтверски процес обухвата цео животни циклус развоја софтвера[18]. Избор парадигме развоја софтвера у највећој мери одређује подпроцесе, активности и карактеристике софтверског процеса[19].

Парадигма је врста апстракције и састоји се од скупа претпоставки, концепата, интелектуалних вредности које стручна и академска заједница прихвата, међусобно дели и примењује, ради једноставнијег и успешнијег решавања конкретних проблема у развоју софтвера. Различите парадигме развоја софтвера имају фундаментално различите концепте помоћу којих описују и апстрахују поглед на реални свет и решавање проблема у процесу развоја софтвера, коришћењем метода, техника и алата софтверског инжењерства. Најзначајнији концепти помоћу којих се описују парадигме развоја софтвера су: архитектура софтвера, процеси развоја софтвера, потребна инфраструктура и начин постављања у продукционо окружење. Историјски преглед еволуције парадигми развоја софтвера дат је на Слици 3.

---

[13] Boehm, B. (1986). A spiral model of software development and enhancement. *ACM SIGSOFT Software engineering notes*, 11(4), 14–24.
[14] Dlamini, G., Ergasheva, S., Kholmatova, Z., Kruglov, A., Sadovykh, A., Succi, G., ... & Zouev, E. (2022). Metrics for software process quality assessment in the late phases of SDLC. In: *Science and Information Conference* (pp. 639–655). Springer, Cham.
[15] https://www.computer.org/education/bodies-of-knowledge/software-engineering
[16] Laato, S., Mäntymäki, M., Islam, A. K. M., Hyrynsalmi, S. & Birkstedt, T. (2022). Trends and Trajectories in the Software Industry: implications for the future of work. *Information Systems Frontiers*, 1–16.
[17] Njanka, S. Q., Sandula, G. & Colomo-Palacios, R. (2021). IT-Business Alignment: A Systematic Literature Review. *Procedia Computer Science*, 181, 333–340.
[18] Huff, K. (1996). Software process modeling. *Software Process*, 5, 1–24. John Wiley & Sons.
[19] Kaisler, S. H. (2005). *Software paradigms*. John Wiley & Sons.

Слика 3. Еволуција концепата за опис парадигми развоја софтвера[20]

Парадигме софтверског инжењерства се према обухвату могу класификовати на:

- парадигме развоја софтвера, које обухватају концепте целокупног животног циклуса развоја софтвера;
- парадигме пројектовања софтвера које обухватају само концепте процеса пројектовања софтвера; и
- парадигме програмирања, које се односе на технике програмирања.

За анализу у овој књизи су од интереса првенствено парадигме развоја софтвера које обухватају цео софтверски процес и у већој мери укључују концепте из парадигми пројектовања софтвера и програмирања.

Софтверски инжењери су с мање или више успеха експериментисали с многим парадигмама у којима су различите концепте стављали у први план[21], али су у пракси најбоље прихваћене и у највећој мери усвојене и коришћене три парадигме развоја софтвера:

[20] Прилагођено према: Bogdanović, Z., Despotović-Zrakić, M., Barać, D., Labus, A. & Radenković, M. (2023). The Role of DevOps in Sustainable Enterprise Development. In: García Márquez, F. P., Lev, B. (eds.) *Sustainability. International Series in Operations Research & Management Science,* vol 333. Springer, Cham. https://doi.org/10.1007/978-3-031-16620-4_12

[21] Vranic, V. (2000). Multiple software development paradigms and multi-paradigm software development. In: *Proc. of the Information Systems Modelling* (pp. 191–196). http://www2.fiit.stuba.sk/~vranic/pub/MultiParadigmSwDev.pdf

- парадигма развоја процедуралног софтвера,
- парадигма развоја објектно-оријентисаног софтвера, и
- парадигма развоја процесно-оријентисаног софтвера.

3. ПАРАДИГМА РАЗВОЈА ПРОЦЕДУРАЛНОГ СОФТВЕРА

Парадигма развоја процедуралног софтвера представља функционални поглед на реални свет и пословање. Настала је почетком шездесетих година двадесетог века, под утицајем функционалне организације предузећа, која је тада била доминантна[22]. Развијала се упоредо с развојем рачунарства, почев од дефиниције алгоритма у деветом веку[23], радова Алана Тјуринга[24] из 1936. године, па до тражења одговора на изазове софтверске кризе[25] у радовима Дајкстре, Кнута, Хора и других, шездесетих и седамдесетих година прошлог века. Најзначајнији концепти који су генерисани током развоја ове парадигме су: примена апстракције у пројектовању софтвера с нагласком на *top-down* приступ, увођење процедуралне апстракције и подпрограма[26]. У методе програмирања уведени су структурно програмирање[27] и модули[28], представљани као црна кутија с минималним интерфејсима, претеча данашњих модула у објектном приступу и нити и сервиса у процесном приступу[29]. У току развоја ове парадигме, поред развоја мултипрограмских оперативних система[30,31], појавили су се и концепти конкурентног програмирања за сихронизацију процеса и контролу приступа критичном региону[32], који су веома успешно интегрисани у инжењерску праксу. Уведен је и појам процеса који може бити комплетан рачунарски програм или његов део који се може паралелно извршавати[33]. Публиковани су и први радови о конкурентним дистрибуираним

[22] Chandler Jr, A. D. (1969). *Strategy and structure: Chapters in the history of the American industrial enterprise* (Vol. 120). MIT press.
[23] Nabirahni, D. M., Evans, B. R. & Persaud, A. (2019). Al-Khwarizmi (algorithm) and the development of algebra. *Mathematics Teaching Research Journal*, 11(1–2), 13–17. https://ioinformatics.org/journal/v11si_2017_71_74.pdf
[24] Turing, A.M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, Series 2, 42 (1936–37): 230–265 https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf
[25] Ensmenger, N. (1998). *Building Castles in the Air: The Software Crisis and the Art of Programming 1945–1968*. Society for the History of Technology Conference.
[26] Wilkes, M. V., Wheeler, D. J. & Gill, S. (1951). *The Preparation of Programs for an Electronic Digital Computer: With special reference to the EDSAC and the Use of a Library of Subroutines.* Addison-Wesley Press.
[27] Dahl, O.J., Dijkstra, E.W. & Hoare, C.A.R. (eds.) (1972). *Structured programming*. Academic Press Ltd.
[28] Dennis, J. B. (1975). Modularity. *Software Engineering*, 128–182.
[29] Shaw, M. (1984) Abstraction Technique in Modern Programming Languages, *IEEE Software*, 10–26.
[30] Hansen, P. B. (1970). The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4), 238–241.
[31] Hansen, P. B. (1973). *Operating system principles*. Prentice-Hall, Inc. Englewood Cliffs, N.J.
[32] Dijkstra, E. W. (1968). Cooperating sequential processes. In: *The origin of concurrent programming* (pp. 65–138). Springer, New York, NY.
[33] Dennis, J. B. (1975). Concurrency in software systems. *Software Engineering*, 111–127.

програмима[34]. Упоредо с развојем апстакција у програмирању, развијане су апстракције над подацима. Кодов релациони модел[35] и *SQL* базиран на релационој алгебри, постали су део стандардног информационог модела за апстракцију података у овој парадигми. Нарочиту помоћ програмерима донело је увођење појма трансакције[36] и њене контроле на нивоу софтвера за управљање базама података. За управљање развојем софтвера користила се и унапређивала метода водопада.

Највећи недостатак ове парадигме је низак ниво апстракције при представљању концепата реалног света у форми рачунарског кôда[37], који је захтевао од програмера стручна знања, интелектуални напор и висок ниво креативности при писању програма. Креативних програмера у то време није било довољно. Уместо тражења погоднијих апстракција и ефикаснијих метода пројектовања софтвера, главни промотери ове парадигме су писали о програмирању као о уметности[38]. Без квалитативно нових концепата, који би решили низак ниво апстракције и дали одговор на нове и озбиљније захтеве привреде, ова парадигма је запала у кризу и почетком деведесетих година двадесетог века је напуштена у програмерској пракси.

Без обзира на недостатке, парадигма развоја процедуралног софтвера поставила је добре основе за објектно-оријентисани приступ развоју софтвера, а касније, када се схватило да ни објектни приступ не може у потпуности да испуни захтеве модерних информационих система, који у први план стављају дистрибуираност, скалабилност и поузданост, показала се као солидна основа за нову парадигму која те захтеве испуњава: развој процесно-оријентисаног софтвера.

При дефинисању парадигме развоја процесно-оријентисаног софтвера, анализом научних радова од пре више деценија дошло се до закључка да захтеве дистрибуираности, скалабилности и поузданости може да испуни софтвер представљен у форми међусобно повезаних црних кутија, како је на почетку рачунарства предлагано[39], с данашњим микросервисима као дистрибуираном процедуралном апстракцијом и редукованим дистрибуираним једнодимензионим информационим моделом, који се састоји од само једне глобалне бинарне релације у форми *NoSQL* базе података. Тиме је постављена основа за нову парадигму која је данас доминантна: развој процесно-оријентисаног софтвера. Имајући у виду

[34] Hansen, P. B. (1978). Distributed processes: A concurrent programming concept. *Communications of the ACM*, 21(11), 934–941.
[35] Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), 377–387.
[36] Codd, E. F. (1979). Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems (TODS)*, 4(4), 397–434.
[37] Fitzgerald, B. (1996). Formalized systems development methodologies: a critical perspective. *Information systems journal*, 6(1), 3–23.
[38] Dijkstra, E. W. (1971). *A short introduction to the art of programming* (Vol. 4). Eindhoven: Technische Hogeschool Eindhoven.
[39] Haigh, T. (2002). Software in the 1960s as Concept, Service, and Product. *IEEE Annals of the History of Computing*, 24(1), 5–13.

искуства с методом водопада, која је највише коришћена у управљању развојем софтвера применом парадигме развоја процедуралног софтвера и новим читањем научених лекција из протеклих деценија, метода водопада је редизајнирана у складу с модерним захтевима, тако да буде циклична, никад се не завршава, подржава перманентну израду и испоруку софтвера, а уместо на стриктну методологију и строги формализам, акценат је стављен на менаџмент, алате и кључне индикаторе перформанси који прате фазе развоја софтвера. Тако је из научених лекција из прошлих времена настао *DevOps*, као доминантан метод за управљање процесом развоја софтвера данас.

Имајући наведено у виду, решења многих проблема с којима се суочава софтверско инжењерство, поред модерне литературе, треба тражити и у научним радовима из протеклих деценија. Проблеми с којима су се у то време суочавали софтверски инжењери, слични су данашњим.

4. ПАРАДИГМА РАЗВОЈА ОБЈЕКТНО-ОРИЈЕНТИСАНОГ СОФТВЕРА

Парадигма развоја објектно-оријентисаног софтвера формирала се упоредо с парадигмом развоја процедуралног софтвера, најпре из потребе да се у симулационом софтверу за војне сврхе, спецификација модела реалног света представи јасним и препознатљивим концептима из реалног света – објектима и њиховом интеракцијом.

Објектно-оријентисани приступ развоју софтвера оријентисан је ка објектима као концептима апстракције структуре реалног света. Интеракције између објеката се реализују разменом порука (*message*). Информациони модел је рудиментаран и односи се само на атрибуте (променљиве) објеката. Динамика промена атрибута се спецификује имплицитно, извршавањем метода. Имплицитна спецификација динамике промене атрибута највећи је недостатак парадигме развоја објектно-оријентисаног софтвера. За предузећа је важно да се све промене експлицитно спецификују и да се о њима благовремено извештава.

Концепт класе и објекта као инстанце класе први пут је употребљен 1966. године[40]. Касније су уведени и концепти за сакривања информација[41], под контроверзним називом *dataless programming*, који су претеча апстрактних типова података[42], с препорукама да се подаци, њихова интерна организација и процедуре за претраживање и ажурирање држе сакривени, или да се преко апстрактних интерфејса потпуно изместе изван модула.

[40] Hoare, C. A. R. (1966). Further thoughts on record handling AB21. 3.6. *ALGOL Bulletin*, (23), 5–11.
[41] Balzer, R. M. (1967, November). Dataless programming. In: *Proceedings of the November 14–16, 1967, fall joint computer conference* (pp. 535–544).
[42] Parnas, D. L. (1972). A technique for software module specification with examples. *Communications of the ACM*, 15(5), 330–336.

Објектно-оријентисано програмирање спада у класу конкурентног програмирања. Инстанце објеката заједно с подацима егзистирају истовремено. Такође, и методи се извршавају паралелно у реалном времену. Паралелна егзистенција објеката и извршења метода довели су до повећења скалабилности, али су постали и узрок неугодних проблема везаних за контролу приступа критичном региону при манипулацији подацима објеката. Методи који се паралелно извршавају, могу истовремено приступати истим подацима објекта.

Да би се ублажили проблеми конкурентног програмирања и програмери ослободили од размишљања о критичном региону и сигнализацији, за пренос података између метода уведен је концепт поруке у форми структурираних података и позива извршења операција, а као начин за њихов пренос уведен је нов концепт, канал или ред порука (*Message Queue*), реализован у форми коначног бафера, који интерно решава проблеме приступа критичном региону и сигнализацији, па програмер о томе не мора да брине. *Message Queue* је заснован на догађајима и не припада објектно-оријентисаном приступу развоја софтвера.

У сложенијим апликацијама, када методи паралелно приступају подацима објеката, поруке и њихови редови нису довољан концепт за заштиту од проблема конкурентног програмирања који су се у таквим случајевима јављали. Нужно је било у објектни модел увести нове концепте за сигнализацију и координацију између објеката, који ће сакрити коришћена средства оперативног система, тако да се способност пријемпције[43] може везати за објекат или неки његов метод. Да би се овај проблем решио, накнадно је у објектно-оријентисано програмирање уведен концепт Монитора[44] као посебне класе у оквиру које се као засебне варијабле могу декларисати синхронизациона средства оперативног система. Тиме се отворила могућност да програмери имплементирају пријемпцију унутар свог програма и примењују је на комплексне случајеве координације и синхронизације. За лакше програмирање унапред су дефинисани стандардни типови монитора, који су на располагању програмерима у форми библиотека: коначни бафер за размену порука између објеката, ресурсни монитор за приступ заједничком ресурсу, синхронизациони објекат у коме се, у једном тренутку времена, може извршити само један метод, као и разне друге врсте монитора.

Информациони модел реалног система и његова динамика су код парадигме развоја објектно-оријентисаног софтвера у другом плану, што је велики недостатак ове парадигме. У реалном свету постоје разноврсне везе између објеката које се реализују кроз податке, а које објектни модел игнорише. Чак је и *COBOL* са уграђеним индекс-секвенцијалним базама података имао бољи начин управљања подацима од објектно-оријентисаног приступа развоју софтвера[45].

---

[43] Пријемпција се у рачунарству дефинише као могућност привременог прекида извршења програма, како би се извршио програм вишег приоритета.
[44] Hoare, C. A. R. (1974). Monitors: An operating system structuring concept. In: *The origin of concurrent programming* (pp. 272–294). Springer, New York, NY.
[45] Schricker, D. (2000). Cobol for the next millennium. *IEEE Software*, 17(2), 48–52.

Релациони модел база података је у време настанка објектно-оријентисаног модела био познат. У релационом моделу, проблем категоризације атрибута за стационарне структуре података једним делом је био добро решен (спољни кључ, композитни кључ, агрегација). Чак је направљен и искорак ка објектном приступу у форми *entity-relationship model*-а (*ER model*)[46,47]. Нажалост, искуства из релационог модела нису укључена у спецификацију објектног модела[48]. То је био велики недостатак због којег се парадигма објектно-оријентисаног развоја софтвера никада није опоравила.

Релационим базама података могуће је било приступити преко конектора, али то више личи на процедуралну парадигму него на објектни приступ[49]. Везе између података су изван објектног модела, коришћење таквих веза представља имплицитно нарушавање принципа сакривања информација на коме се инсистирало[50]. Објектно-релационе базе података, које су се појавиле, решавају само проблем перзистентног складиштења локалних података објекта[51], а не решавају квалитетно проблем инкорпорирања веза између података различитих објеката, као и веза између вредности података које се могу динамички појавити. Објектно-релациони мапери, као специфичан *middleware*[52] само су делимично решили овај проблем[53].

Дистрибуираност је један од највећих изазова за парадигму објектно-оријентисаног развоја софтвера. Објектно-оријентисан модел је монолитан. За дистрибуирање објеката је потребно уградити *middleware* који ће администрирати дистрибуиране објекте и обезбедити илузију да су сви објекти локално доступни. Један од озбиљнијих покушаја да се овај проблем реши је *CORBA*[54]. *CORBA* је после

[46] Chen, P. P. S. (1976). The entity-relationship model – toward a unified view of data. *ACM transactions on database systems (TODS)*, 1(1), 9–36. https://dl.acm.org/doi/pdf/10.1145/320434.320440

[47] Chen, P. (2002). Entity-relationship modeling: historical events, future trends, and lessons learned. *Software pioneers: contributions to software engineering*, 296–310. https://nsu.ru/xmlui/bitstream/handle/nsu/8875/Chen_Pioneers.pdf

[48] Smith, K. E. & Zdonik, S. B. (1987, December). Intermedia: A case study of the differences between relational and object-oriented database systems. In: *Conference proceedings on Object-oriented programming systems, languages and applications* (pp. 452–465).

[49] Balzer, S., Gross, T. R. & Eugster, P. (2007, July). A relational model of object collaborations and its use in reasoning about relationships. In: *European Conference on Object-Oriented Programming* (pp. 323–346). Springer, Berlin, Heidelberg.

[50] Ireland, C., Bowers, D., Newton, M. & Waugh, K. (2009, March). A classification of object-relational impedance mismatch. In: *2009 First International Conference on Advances in Databases, Knowledge, and Data Applications* (pp. 36–43). IEEE.

[51] Bertino, E. & Martino, L. (1991). Object-oriented database management systems: concepts and issues. *Computer*, 24(4), 33–47.

[52] O'Neil, E. J. (2008, June). Object/relational mapping 2008: hibernate and the entity data model (edm). In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (pp. 1351–1356).

[53] Torres, A., Galante, R., Pimenta, M. S. & Martins, A. J. B. (2017). Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information and software technology*, 82, 1–18.

[54] Vinoski, S. (1997). CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2), 46–55.

дуже употребе напуштена због комплексног одржавања и високих трошкова[55]. Дистрибуирање обраде је могуће позивима удаљених процедура (*Remote Procedure Call – RPC*)[56]. У пракси је примена била компликована због посебне клијентске и серверске инфраструктуре за удаљене позиве процедура[57]. Поред тога, *RPC* припада процедуралној парадигми. Делимично успешан покушај да се направи потпуно објектни *RPC*, реализован је у форми библиотека за позив удаљеног метода (*Remote Method Invocation – RMI*) који преко *RPC* преноси референце на објекат[58].

Ефикасна замена за *RPC* и *RMI* је веб сервис (*Web Service*[59]), који за позив и пренос аргумената употребљава широко коришћен и лако доступан *http* протокол. Скалабилност се може повећати паралелизацијом извршења више инстанци истог веб сервиса[60], а поузданост редундантношћу веб сервиса. Веб сервис је био значајно проширење објектног приступа развоју софтвера. Међутим, веб сервис је концепт парадигме развоја процесно-оријентисаног софтвера. Решење проблема дистрибуираности нађено је изван објектно-оријентисаног приступа.

За парадигму развоја објектно-оријентисаног софтвера нема ни једноставног, ни ефикасног начина за обезбеђивање дистрибуираности, скалабилности и поузданости.

Значајан недостатак објектне парадигме је тај што се динамика и промене у информационом моделу не спецификују експлицитно, већ имплицитно преко метода, што у великој мери ограничава ефикасно моделовање и праћење динамике реалног система.

Најзначајнији недостатак објектно-оријентисане парадигме развоја софтвера је то што не садржи концепте којима се могу имплементирати нефункционални захтеви, архитектура софтвера, модули и интеграција. Све допуне програмских језика за објектно-оријентисано програмирање: нити за извршење програма, пакети за модуле, сервиси за дистрибуираност и скалабилност, представљају концепте развоја софтвера оријентисаног на процесе.

Многи софтверски инжењери за мање пројекте и монолитне апликације још

[55] Henning, M. (2006). The Rise and Fall of CORBA: There's a lot we can learn from CORBA's mistakes. *Queue*, 4(5), 28–34.
[56] Tay, B. H. & Ananda, A. L. (1990). A survey of remote procedure calls. *ACM SIGOPS Operating Systems Review*, 24(3), 68–79.
[57] Tanenbaum, A. S. & Van Renesse, R. (1988). A critique of the remote procedure call paradigm. In: *Proc. Euteco'88* IR 123 (pp. 1–11).
[58] Waldo, J. (1998). Remote procedure calls and java remote method invocation. *IEEE concurrency*, 6(3), 5–7.
[59] Newcomer, E. (2002). *Understanding web services: XML, WSDL, SOAP, and UDDI*. Addison-Wesley Professional.
[60] Bravetti, M., Gilmore, S., Guidi, C. & Tribastone, M. (2007, November). Replicating web services for scalability. In: *International Symposium on Trustworthy Global Computing* (pp. 204–221). Springer, Berlin, Heidelberg.

увек користе парадигму развоја објектно-оријентисаног софтвера. Међутим, за информационе системе великих предузећа, скромно дефинисан информациони модел[61] и ограничене могућности за дистрибуираност, скалабилност и поузданост су разлог да се ова парадигма развоја софтвера напусти.

5. ПАРАДИГМА РАЗВОЈА СОФТВЕРА ОРИЈЕНТИСАНОГ НА ПРОЦЕСЕ

Парадигма развоја софтвера оријентисаног на процесе[62] настала је као покушај тражења одговора на изазове друге по реду софтверске кризе (*Software crisis 2.0*)[63], која је у великој мери изазвана ограничењима парадигме развоја објектно-оријентисаног софтвера[64], а која су постала видљива услед брзог ширења интернет технологија. Решења су тражена у повратку на логичне и експлицитне концепте за спецификацију динамике система, који су били познати и на почетку рачунарства.

Централни концепт парадигме развоја процесно-оријентисаног софтвера је процес. Основна градивна јединица процеса је догађај који представља било коју промену у софтверу или његовом окружењу. Изведени, апстрактни концепт је сервис који представља колекцију догађаја који мењају стање одређеног ресурса. Сходно томе, процес се може посматрати и као хронолошки уређена секвенца догађаја и сервиса који су везани за неку појаву и њен животни циклус. Процеси, сервиси и догађаји егзистирају паралелно. Комуникација и синхронизација између процеса врши се преко података, догађаја и сервиса.

Највећа предност парадигме развоја процесно-оријентисаног софтвера у односу на друге парадигме је то да се процеси, сервиси и догађаји могу једноставно дистрибуирати, паралелизовати и скалирати. Такође, софтвер оријентисан на процесе на најприроднији начин подржава функционисање пословног окружења које је оријентисано на процесе. Значајна промена коју је у развој софтвера унела парадигма развоја софтвера оријентисаног на процесе је већи удео интеграције, интеграционих сервиса, софтверских оквира, платформи и екосистема, у односу на удео пројектовања и кодирања појединачних микросервиса. То је могуће јер су сервиси, процеси и апликације, агрегације догађаја, који се могу интегрисати у апликације применом архитектура обраде догађаја организованих у софтверске платформе и екосистеме. Тиме се већи део активности, у развоју софтвера, измешта из фазе кодирања у фазу конфигурисања платформе.

---

[61] Ireland, C. & Bowers, D. (2015, May). Exposing the myth: object-relational impedance mismatch is a wicked problem. In: *DBKDA 2015, The Seventh International Conference on Advances in Databases, Knowledge, and Data Applications* (pp. 21–26). IARIA XPS Press.

[62] Gruhn, V. (2002). Process-centered software engineering environments, a brief history and future challenges. *Annals of Software Engineering*, 14(1), 363–382.

[63] Fitzgerald, B. (2012). Software Crisis 2.0. *Computer*, 45(4), 89–91.

[64] Mike Hinchey, ”Software Crisis 2.0“;in: *Software Technology: 10 Years of Innovation in IEEE Computer*, IEEE, 2018, pp.1–16, doi: 10.1002/9781119174240.ch1.

У теорији менаџмента предузећа, након напуштања функционалног модела који је предузеће посматрао као скуп пословних функција, почетком осамдесетих година прошлог века, фокус се помера ка управљању процесима. Као даља надградња управљања процесима кренуло се ка управљању променама, а нешто касније и ка управљању производима.

Догађај је есенцијални манифестациони концепт којим се описује промена, тако да су при управљању променама у предузећу, догађај и реакција на догађај стављени у први план. Да би софтвер могао на нивоу концепата пратити промене у начину организације и управљања у предузећима, нужно је било да се у моделовање софтвера уведу концепт догађаја и реакције на догађај.

Функционална организација *IT* сектора у предузећима, подела на одељења развоја софтвера и *IT* операције, представљали су проблем за имплементацију процесног приступа. Процеси су обухватали оба одељења. Било је нужно редефинисати организацију и направити искорак ка процесној организацији која обједињује процесе у оба одељења.

За управљање софтвером као производом, нужно је било интегрисати све фазе софтверског процеса у јединствену методологију, тј. повезати развој софтвера са одржавањем и експлоатацијом. Док је парадигма развоја процедуралног и делом објектно-оријентисаног софтвера конципирана у академским круговима, парадигму данашњег развоја процесно-оријентисаног софтвера у највећој мери су профилисала велика софтверска предузећа, на почетку због својих потреба, а касније и да би се задовољили захтеви клијената. Производи највећих софтверских предузећа, која предводе *Гугл*, *Амазон*, *Мета* (некадашњи *Фејсбук*), *Мајкрософт*, *Оракл* и други, у потпуности су дигитални. Испорука софтвера се третира као испорука производа, који је део ланца вредности (*Value Stream*). Менаџмент у центар пажње ставља управљање током вредности софтвера (*Software Value Flow*)[65]. Пројектни приступ у развоју софтвера у овим предузећима је напуштен. Уместо управљања пројектима, у овим предузећима се **управља производима**. С обзиром на то да се ради о најзначајнијим предузећима која одређују трендове, њихова модерна организација, инфраструктура, менаџмент и иновативни производи су допринели иреверзибилним променама у актуелној парадигми развоја софтвера[66]. Сасвим је извесно да ће софтверска предузећа доминирати светском економијом, почев од блиске будућности[67], па све до краја ове ере[68] и да ће у значајној мери

[65] Jeong, B. K., Ji, S. Y. & Jeong, D. H. (2022). Lean IT With Value Stream Mapping Analysis: A Case Study in Software Development Life Cycle Process. *Information Resources Management Journal* (IRMJ), 35(1), 1–18.

[66] Laato, S., Mäntymäki, M., Islam, A. K. M., Hyrynsalmi, S. & Birkstedt, T. (2022). Trends and Trajectories in the Software Industry: implications for the future of work. *Information Systems Frontiers*, 1–16.

[67] Chourasia, S., Tyagi, A., Pandey, S. M., Walia, R. S. & Murtaza, Q. (2022). Sustainability of Industry 6.0 in *Global Perspective: Benefits and Challenges*. MAPAN, 1–10.

[68] Duggal, A. S., Malik, P. K., Gehlot, A., Singh, R., Gaba, G. S., Masud, M. & Al-Amri, J. F. (2022). A sequential roadmap to Industry 6.0: Exploring future manufacturing trends. *IET Communications*, 16(5), 521–531.

утицати на развој софтверског инжењерства.

У парадигми развоја процесно-оријентисаног софтвера, процеси, сервиси и догађаји су у првом плану, и то:

1. Софтверско инжењерство је редефинисано и у први план је стављен софтверски процес[69].
2. Софтвер се посматра као производ[70], а подпроцеси животног циклуса развоја софтвера се организују као неки од облика фабрике софтвера[71].
3. Основни концепт моделовања у информационом систему предузећа је пословни процес (*business process*)[72]. Ентитети у предузећу се описују подацима, а промене података пословним процесима, сервисима и догађајима. Шеме података су прилично лабаво дефинисане. Не инсистира се на унапред дефинисаним структурним везама између података. Везе између података су динамичке, у реалном времену се откривају статистичким и методама вештачке интелигенције. За ту намену је развијена нова научна област: Наука о подацима (*Data Science*)[73].
4. На најнижем нивоу кодирања и имплементације користе се концепти за извршење рачунарских програма и његових делова као што су: процеси, нити, сервиси и догађаји, који су већим делом уграђени у кернел мултипрограмског система, имају способност пријемпције и омогућавају ефикасну координацију и синхронизацију процеса у рачунарском програму[74]. Тиме је успостављена квалитетна и готово једнозначна веза концепата, са истим називима и функционалностима, између пословне логике – апстрактног логичког модела процеса информационог система и физичког, извршног модела у форми рачунарског кôда. То је у великој мери допринело поједностављењу процеса генерисања кôда, али и квалитету и могућностима аутоматског генерисања софтвера.

Развој савремених предузећа је динамичан. Број клијената и обим информација могу се у кратком периоду знатно повећати. Основни пројектни захтев у развоју софтвера је да буде направљен тако да без значајнијих измена у кôду подржи брзе промене на тржишту и промене у предузећу[75]. Животни циклус софтвера треба да буде континуалан и да прати животни циклус организације.

Изузимајући корисничке захтеве који се односе на домен пословања и који су за

[69] Fuggetta, A. & Di Nitto, E. (2014). Software process. In: *Future of Software Engineering Proceedings*, 1–12.
[70] Deelstra, S., Sinnema, M. & Bosch, J. (2004, August). Experiences in software product families: Problems and issues during product derivation. In: *International Conference on Software Product Lines* (pp. 165–182). Springer, Berlin, Heidelberg.
[71] Cusumano, M. A. (1989). The software factory: a historical interpretation. *IEEE Software*, 6(2), 23.
[72] Aguilar-Saven, R. S. (2004). Business process modelling: Review and framework. *International Journal of production economics*, 90(2), 129–149.
[73] Dhar, V. (2013). Data science and prediction. *Communications of the ACM*, 56(12), 64–73.
[74] Banga, G. & Mogul, J. C. (1998, June). Scalable Kernel Performance for Internet Servers Under Realistic Loads. In: *USENIX Annual Technical Conference* (p. 12).
[75] Kim, B., Kim, H. & Jeon, Y. (2018). Critical success factors of a design startup business. *Sustainability*, 10(9), 2981.

сваку организацију различити, скуп захтева који су заједнички за већину великих предузећа је:

1. **Дистрибуираност**. Делови мултинационалног предузећа и клијенти могу бити у различитим државама. Подаци треба да се чувају и обрађују на месту где су настали.
2. **Линеарна скалабилност**. Информациони систем треба да има могућност једноставног повећања снаге обраде и усклађивања потребних рачунарских ресурса, искључиво оперативним и административним поступцима додавања и искључивања инфраструктурних капацитета, без додатног кодирања или промена у софтверу.
3. **Поузданост**. Информациони систем треба да настави да ради с пуном функционалношћу и у случају отказивања неког његовог дела.

У протекле две деценије учињен је огроман напор у изградњи инфраструктуре рачунарства у облаку, која је окосница виртуелне инфраструктуре за овај приступ: изградњи софтверски дефинисаних мрежа које представљају глобални модел умрежавања, *NoSQL* база података и глобалних фајл система, који обезбеђују глобални информациони модел, развоју екосистема и оквира базираних на догађајима, сервисима и процесима, као алата у софтверским фабрикама, и контејнера који су постали стандардан концепт за испоруку софтвера.

Иако се објектно-оријентисан приступ и даље користи, парадигма развоја софтвера оријентисаног на процесе постала је доминантна у пракси софтверског инжењерства. Методе развоја софтвера, развојна и продукциона окружења већ дужи низ година прилагођавају се само овој парадигми.

5.1 Историја и еволуција метода развоја софтвера оријентисаног на процесе

Методе развоја софтвера су се дужи низ година упоредо развијале и користиле истовремено за парадигме развоја објектно и процесно-оријентисаног софтвера. Многи објектно-оријентисани програмски језици добили су процесне екстензије, чиме су им проширене могућности и продужен век примене. У складу с тим, модификоване су методе и алати за развој софтвера. Развојем и све већом применом парадигме развоја Развојем и све већом применом парадигме развоја процесно-оријентисаног софтвера, у методама и алатима за развој софтвера напуштају се ограничења и стриктне методе објектне парадигме[76]. Акценат се ставља на моделовање динамике података и прилагођавање променама структуре информационог система. Тиме су отворена врата за појаву нових метода у развоју софтвера које су експлицитно усмерене на моделовање процеса и токова

[76] Ksiksi, A., Lazzez, O., Ltifi, H., Hamdani, T. M. & Alimi, A. M. (2023). FARUL3S: a New Framework for Agile Regulated Ultra-Large-Scale Software Systems. Institute of Electrical and Electronics Engineers (IEEE).

података. Уместо на моделовање података, код пројектовања је акценат стављен на моделовање знања, метаподатака и података којима се описују процеси у апликацији и информационом систему.

У оквиру парадигме развоја процесно-оријентисаног софтвера, данас се најчешће користе методе које представљају еволутивни искорак потекао из две методе развоја објектно-оријентисаног софтвера, које су током времена и стицања искуства из праксе модификоване, проширене и прилагођене конкретним проблемима и експлоатационим условима, а то су: развој софтвера вођен подацима и развој софтвера вођен моделом. Из њих је проистекло доменом вођено пројектовање софтвера. С њима се комбинују и искуства из примене сервисно-оријентисаних архитектура и микросервиса.

Развој софтвера вођен подацима усмерен је на методе управљања подацима и токовима догађаја, како би се постигле задовољавајуће перформансе софтвера. Развој софтвера вођен моделом усмерен је на изградњу или унапређење модела и алгоритама софтверског система, ради постизања захтеваних перформанси. Доменом вођено пројектовање софтвера настало је као рационализација моделом вођеног развоја софтвера, која се односи само на процес пројектовања пословне логике у сложеним пословним системима.

Развој софтвера вођен моделом и развој софтвера вођен подацима представљају методе које се примењују у свим слојевима модела: презентационом, апликационом, доменском и имплементационом слоју. Доменом вођено пројектовање софтвера односи се искључиво на пројектовање доменског слоја пословне логике која је основа за пројектовање апликационе логике.

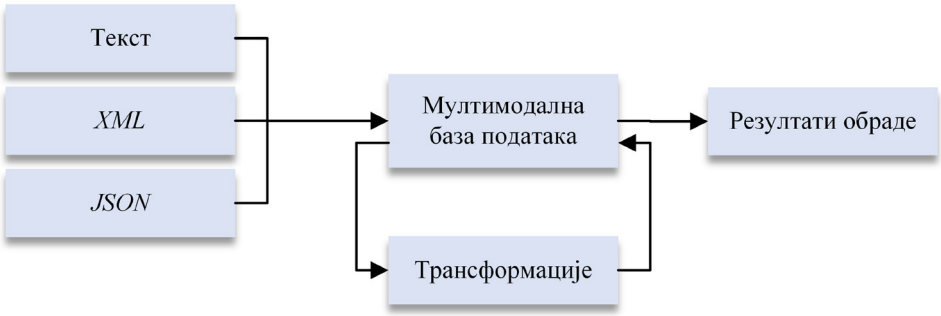
5.1.1 Развој софтвера вођен подацима

Развој софтвера вођен подацима (*Data Driven Software Development*) потекао је од *Data Flow* метода који је под разним називима у литератури познат још из периода 1920–1950. године[77]. Карактерише га оријентација на спецификацију процеса преко секвенци појединачних догађаја, с високом способношћу за паралелизацију[78]. *Data Flow* и *FlowChart* први пут су представљени у раду о структурном пројектовању софтвера из 1974. године[79]. *Data Flow* као метод за развој софтвера у наредним годинама мало се користио и био је готово заборављен. Појавом *IoT* (*Internet of Things*) уређаја и сензора с великом количином генерисаних података, као и развојем друштвених медија на којима се генеришу велике количине садржаја, *Data Flow* је поново откривен, редефинисан у складу с новим захтевима, и дато

[77] Koepke, D. J. (1985). *The evolution of software design ideas* (Master's thesis, California State University, Northridge).
[78] Dennis, J. B. (1974). First version of a data flow procedure language. In: *Programming Symposium* (pp. 362–376). Springer, Berlin, Heidelberg.
[79] Stevens, W. P., Myers, G. J. & Constantine, L. L. (1974). Structured design, *IBM Systems Journal, 13*(2), 115–139.

му је ново име – Развој софтвера вођен подацима, и веома брзо је прихваћен у програмерској заједници[80].

Развој софтвера вођен подацима заснива се на идеји да се основни концепти развоја процесно-оријентисаног софтвера, подаци и њихове промене (догађаји), морају експлицитно дефинисати. Подаци су најчешће мултимодални: текст, слика, звук, видео снимак итд. и могу имати разноврсне изворе. Приступ мултимодалним подацима, најчешће преко глобалних *NoSQL* база података, програмерима отвара могућност да имплементирају трансформације података, коришћењем разних савремених метода за претраживање, откривање знања и генерисање пословних извештаја (Слика 4).



Слика 4. Развој софтвера вођен подацима[81]

Једна од главних претпоставки у развоју софтвера вођеног подацима је да су везе између података динамичке, већином скривене у подацима и њиховим вредностима, ограниченог су трајања, потребно је открити их у реалном времену и укључити их у извештаје или акције. Тиме је за процесе откривања знања у подацима и извештавање на велика врата отворен простор за примену статистичких метода, машинског учења и вештачке интелигенције[82]. Појавом и применом *Big Data* и нерелационих база података, са својим специфичностима и предностима, овај приступ развоју софтвера трансформише се у Развој софтвера вођен великим подацима (*Big Data driven Development*)[83]. За извештавање се развија и профилише посебна научна област под називом *Big data* аналитика (*Big Data Analytics*). Акценат се с података пребацује на знање, па је назив методе у неким референцама модификован у Развој система за управљање знањем вођен великим подацима

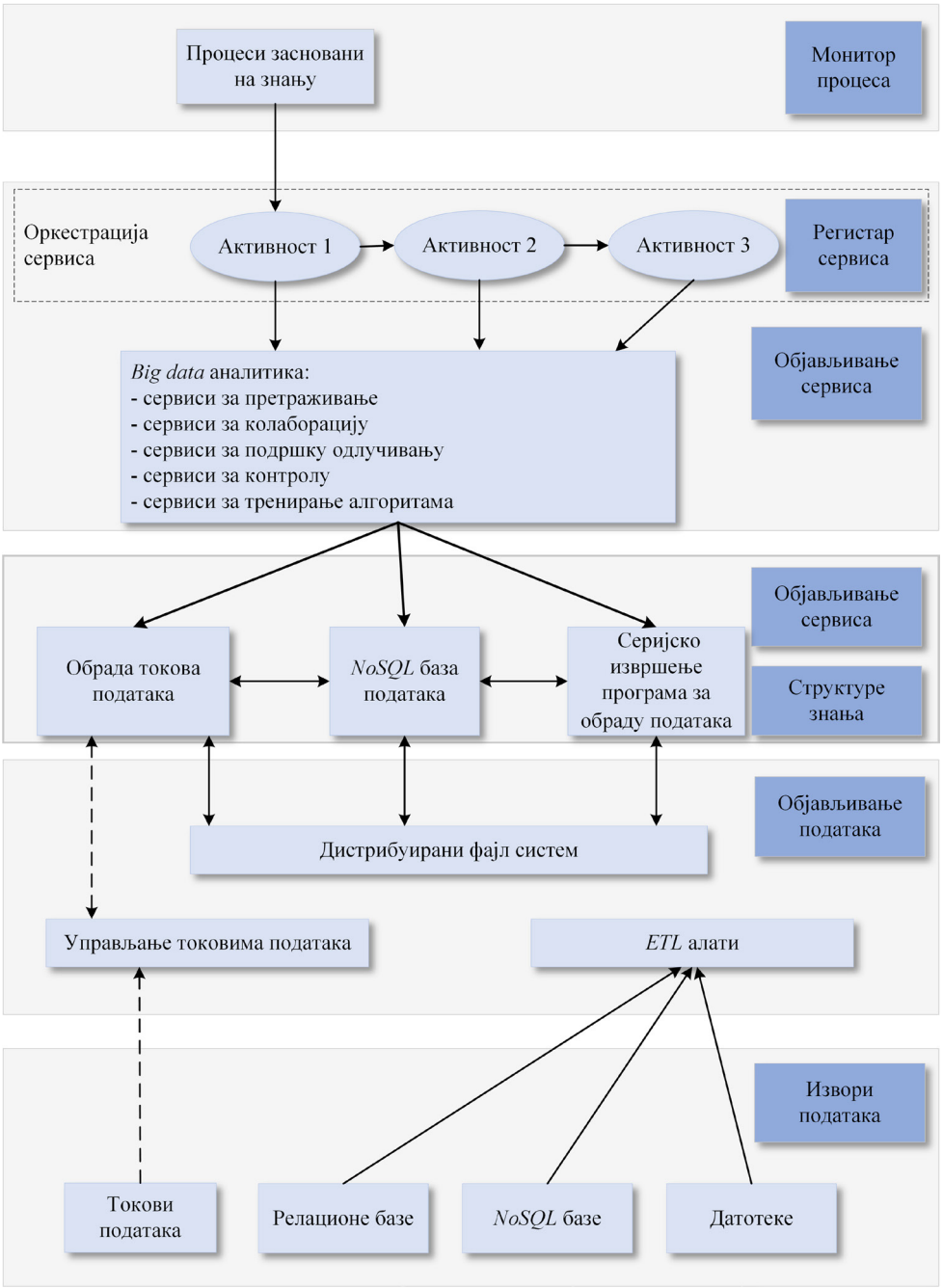
[80] Simmhan, Y., Ravindra, P., Chaturvedi, S., Hegde, M. & Ballamajalu, R. (2018). Towards a data-driven IoT software architecture for smart city utilities, *Software: Practice and Experience, 48*(7), 1390–1416.
[81] Прилагођено према: Laquer, E. Data-driven software development, O'Reilly Media, 2017.
[82] Hesenius, M., Schwenzfeier, N., Meyer, O., Koop, W. & Gruhn, V. (2019, May). Towards a software engineering process for developing data-driven applications. In: *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)* (pp. 35–41). IEEE.
[83] Ghandour, A. (2015). Big Data driven e-commerce architecture, *International Journal of Economics, Commerce and Management*, *3*(5), 940–947.

(*Big Data Driven Knowledge Management Systems*)[84]. Архитектура таквог система приказана је на Слици 5. Приказана архитектура представља једну од могућих варијанти популарне *Lambda* архитектуре[85].

Карактеристика приказане архитектуре је да се подаци масовно генеришу у токовима података (*data stream*), у тзв. *edge* делу инфраструктуре, где их треба и обрадити, како би се смањила латенција и количина података који се преносе преко мреже. Инфраструктура се састоји од великог броја хетерогених извора токова података, који се разликују по локацији, капацитету, могућностима и сервисима који се током времена могу мењати[86]. Традиционални системи за управљање релационим базама података немају добру подршку за управљање токовима података и у њима је тешко имплементирати ефективну паралелну обраду великих количина података[87]. За ту сврху су погодније глобалне дистрибуиране *NoSQL* базе података и глобални фајл системи. За смештање података веома су популарна и информациона језера (*Data Lake*)[88] која се једноставно имплементирају.

Посебан проблем односи се на то како динамички пронаћи и агрегирати потребне дистрибуиране рачунарске ресурсе и користити их за откривање знања и подршку при доношењу одлука. Основа нове инфраструктуре за откривање знања у токовима података заснива се на променама у начину меморисања и претраживања великих количина неструктурираних података. *Map-reduce*[89] алгоритам је предложен као погодан метод који омогућује паралелно претраживање и обраду података који су дистрибуирани између неколико стотина или хиљада сервера. Применом *map-reduce* алгоритма, апстрахује се комплексност обраде великих количина података, сакривају се детаљи паралелизације, расподеле оптерећења и заштите од отказа, чиме се програмерима омогућује да коришћењем одговарајућих библиотека развијају поуздане, скалабилне и дистрибуиране системе засноване на великим подацима.

[84] Le Dinh, T., Phan, T. C. & Bui, T. (2016). Towards an architecture for big data-driven knowledge management systems. *Twenty-second Americas Conference on Information Systems*, San Diego.
[85] Kiran, M., Murphy, P., Monga, I., Dugan, J. & Baveja, S. S. (2015, October). Lambda architecture for cost-effective batch and speed big data processing. In: *2015 IEEE international conference on big data (big data)* (pp. 2785–2792). IEEE.
[86] Renart, E. G., Diaz-Montes, J. & Parashar, M. (2017, May). Data-driven stream processing at the edge. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)* (pp. 31–40). IEEE.
[87] Madden, S. (2012). From databases to big data. IEEE Internet Computing, 16(3), 4–6.
[88] Singh, A. & Ahmad, S. (2019). Architecture of data lake, *International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 5*(2), 4–4.
[89] Dean, J. & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM,* 51(1), 107–113.



Слика 5. Архитектура система за управљање знањем заснованог на великим подацима[90]

[90] Прилагођено према: Le Dinh, T., Phan, T. C. & Bui, T. (2016). Towards an architecture for big data-driven knowledge management systems. *Twenty-second Americas Conference on Information Systems*, San Diego.

За решавање изазова при откривању знања у великим количинама података, потребна је промена традиционалног приступа статистичкој анализи података и развој нових рачунарских алгоритама који могу ефективно обрађивати велике количине података. Од значаја су приступи за редукцију димензија[91] и одабир релевантних варијабли[92] у великим скуповима података[93].

Имплементација савремених система заснованих на знању реализује се комбинацијом технологија као што су: нерелационе базе података (*MongoDB*, *Redis*, *Dynamo*, *BigTable* итд.), дистрибуирани фајл системи (*gfs*, *hdfs*), системи за обраду токова података (*Kafka*, *Spark*, *Flink*), или коришћењем *cloud* инфраструктура и сервиса које нуде *Гугл*, *Амазон*, *Мајкрософт* и други.

5.1.2 Развој софтвера вођен моделом

Развој софтвера вођен моделом (*Model-Driven Development – MDD*, у литератури познат и као: *Model-Driven Engineering-MDE*, *Model-Driven Architecture – MDA*, *Model-Driven Software development – MDSD*), фокусиран је на изградњу хијерархијског модела софтверског система. Основни принципи који се користе код изградње модела су: апстракција при генерисању највиших нивоа хијерархије и аутоматизација у генерисању нижих нивоа хијерархије. Омогућава брзу имплементацију и одржавање, уз смањене трошкове.

Развој софтвера вођен моделом значајан је за парадигму развоја софтвера оријентисаног на процесе, јер је поставио основе за хијерархијску декомпозицију у процесу пројектовања софтвера[94] и утицао на развој фабрика софтвера које су у значајној мери дефинисале развојна окружења, методе управљања знањем, технике и алате за *DevOps* методологију управљања софтверским процесом. Њиме је инспирисано и из њега потекло пројектовање софтвера вођено доменом, које је поставило основе за савремене методе пројектовања софтвера оријентисаног на микросервисе[95].

Развој софтвера вођен моделом настао је из потребе да се развој и одржавање великих и комплексних софтверских система, с више врста различитих екосистема, платформи, оквира, база података, са огромним бројем модула, класа и сервиса, унапреди, побољша и формализује. У пракси је увођење нове или измена

---

[91] Reddy, G. T., Reddy, M. P. K., Lakshmanna, K., Kaluri, R., Rajput, D. S., Srivastava, G. & Baker, T. (2020). Analysis of dimensionality reduction techniques on big data, *IEEE Access*, 8, 54776-54788.

[92] Li, J. & Liu, H. (2017). Challenges of feature selection for big data analytics, *IEEE Intelligent Systems*, 32(2), 9–15.

[93] Fan, J., Han, F. & Liu, H. (2014). Challenges of big data analysis. *National science review*, 1(2), 293–314.

[94] Atkinson, C., Kühne, T. (2005). A Generalized Notion of Platforms for Model-Driven Development. In: Beydeda, S., Book, M., Gruhn, V. (eds): Model-Driven Software Development. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-28554-7_6

[95] Bühler, C. (2021). Microservices in a DevOps Context. https://eprints.ost.ch/id/eprint/936/1/Microservices-in-a-DevOps-Context.pdf

постојеће функционалности код таквих система веома компликовано. Велике проблеме ствара и отежано усклађивање софтвера с новим верзијама библиотека, напуштање застареле развојне платформе и прелазак на нову платформу. Током развоја софтверског инжењерства било је више покушаја да се ови проблеми реше или бар ублаже, коришћењем апстрактних модела и метода у развоју софтвера[96], увођењем *CASE* алата[97] и *IDE* окружења за развој софтвера[98]. Успех ових покушаја је био ограничен.

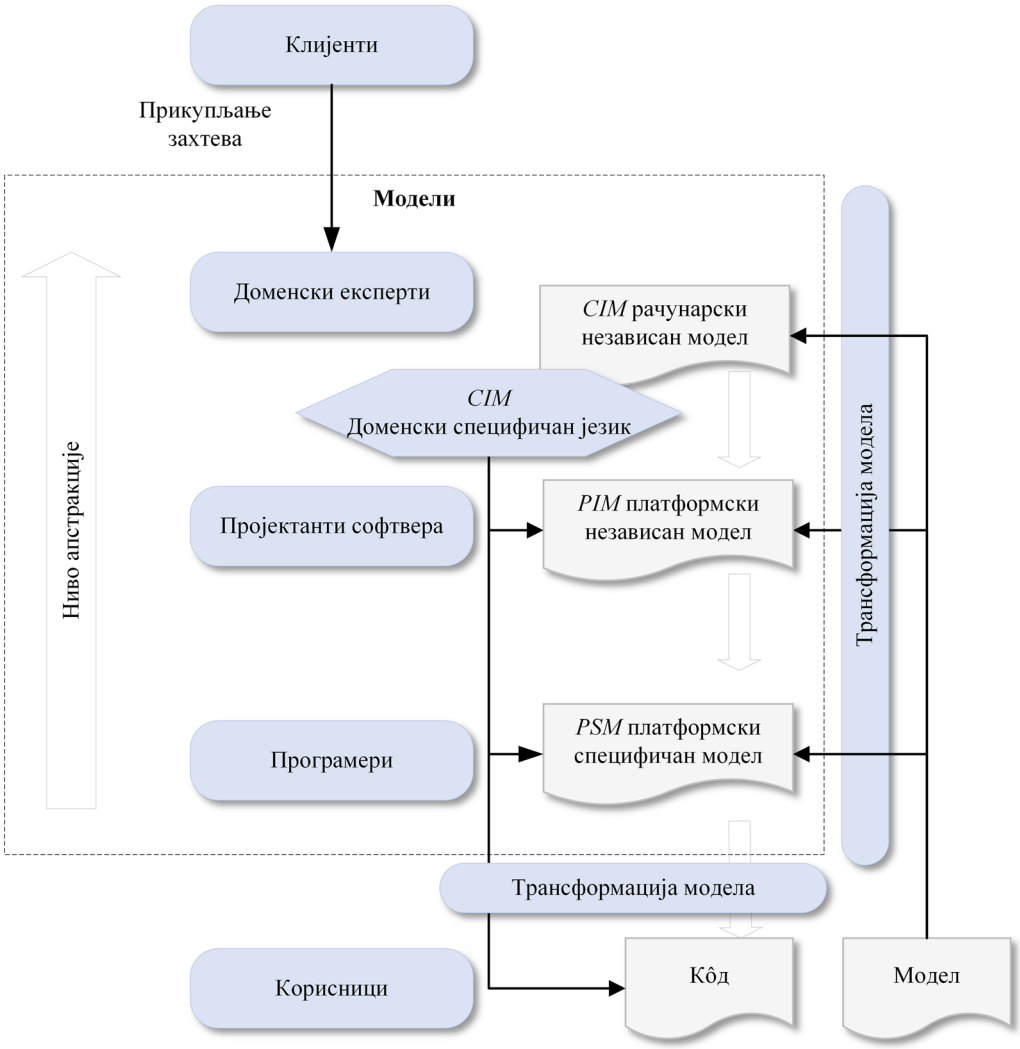
Највећи изазов је начин како дефинисати границе у моделу између дела модела који се односи на функционалности реалног система који се не мења често, и дела модела који зависи од библиотека, апликационих програмских интерфејса и имплементационог окружења који се чешће ажурирају. Значајније резултате у решавању овог проблема дао је предлог Моделом вођеног приступа развоју софтвера *OMG* групе[99].

MDA архитектура чине три хијерархијски повезана модела: први модел се састоји од спецификације реалног система, података и функционалности, на начин независан од имплементације на рачунару (*Computational independent Model – CIM*). *CIM* је специфичан за област примене и обично се реализује коришћењем неког доменски специфичног језика (*Domain Specific Language – DSL*); други модел садржи детаље који се односе на конкретну имплементацију и који су описани независно од имплементационе платформе (*Platform Independent Model – PIM*); трећи модел је платформски специфичан и односи се на имплементацију претходних модела на конкретној технолошкој платформи (*Platform Specific Model – PSM*). Процес развоја софтвера коришћењем ове архитектуре са актерима и њиховим улогама дат је на Слици 6.

Иако је *MDA* архитектура показала добре резултате, овај приступ развоју софтвера није ни брзо ни широко прихваћен, највише због комплексности, али и због инсистирања *OMG* да се за моделовање користи *UML* (*Unified Modeling Language*), а за спецификацију модела *MOF* (*Meta Object Facility*). Ни *UML*, ни *MOF* не садрже концепте за пуну подршку свим активностима софтверског процеса.

---

[96] Jacobson, I., Booch, G. & Rumbaugh, J. (1999). The unified process, *IEEE Software*, 16(3), 96.
[97] Kurbel, K. & Schnieder, T. (1994). *Integration issues of information engineering based I-CASE tools* (No. 33). Arbeitsberichte des Instituts für Wirtschaftsinformatik.
[98] Lopes, D., Hammoudi, S., Bézivin, J. & Jouault, F. (2006). Mapping specification in MDA: from theory to practice. In: *Interoperability of enterprise software and applications* (pp. 253–264). Springer, London.
[99] OMG, Model Driven Architecture, 2003. https://www.omg.org/mda/



Слика 6. MDA архитектура, слојеви и трансформације[100]

Потпуну и једноставну реализацију *MDA* архитектуре омогућила је парадигма развоја процесно-оријентисаног софтвера и сервисно-оријентисане архитектуре (*Service oriented architecture – SOA*), применом специфичних доменски оријентисаних језика као замене за *UML*. На пример, *CIM* модел се реализује помоћу *BPML* спецификације (*Business Process Modeling Language*), *PIM* модел се добија трансформацијом *BPML* модела у *BPEL* (*Business Process Execution Language*) спецификацију, *PSM* модел се добија извршавањем *BPEL* на конкретној платформи која подржава пословне процесе, као што су *Oracle BPM Suite*, *Microsoft BizTalk server*, *IBM Business Process Manager* и друге.

[100]Прилагођено према: OMG, Model Driven Architecture, 2003. https://www.omg.org/mda/

Недостатак *MDA* је и то што није универзална. Спецификација процеса у *CIM* моделу је специфична за привредне гране и конкретна предузећа, па је могућност поновног коришћења већ изграђених модела ограничена.

Без обзира на наведене недостатке, велика софтверска предузећа су основне постулате и искуства из *MDA* прихватила, изменила и прилагодила својој пракси. Хијерархијска декомпозиција и вишеслојна архитектура је задржана. На врху хијерархије је увек модел пословног процеса, најчешће у графичкој форми. На средњем нивоу су модел апликационе и пословне логике и архитектура софтвера. На имплементационом нивоу се користе апликациони програмски интерфејси, оквири, платформе и екосистеми.

Из тих измена и прилагођења настао је модел фабрике софтвера (*Software Factory*)[101] који је значајно унапредио софтверски процес ефикаснијим организовањем окружења за развој софтвера. Фабрике софтвера су даљим развојем постале основа и преовлађујући модел за организовање, управљање и коришћење знања, примену најбољих пракси, метода техника и алата у *DevOps* методологији за управљање софтверским процесом[102].

5.1.3 Фабрике софтвера

Идеја фабрике софтвера (*Software Factory*) настала је још у време парадигме развоја процедуралног софтвера, седамдесетих година прошлог века[103], као покушај да се проблем пројектовања и имплементације комплексних софтверских система реши применом метода хијерархијске и структурне декомпозиције, као и применом добрих искустава из других инжењерских области: масовне производње, производних линија, модуларности и кастомизације производа[104], као и да се смање трошкови и ризик при изради и одржавању софтвера[105].

Основни концепти фабрике софтвера: хијерархијска декомпозиција и модуларизација на сваком нивоу хијерархије, у софтверском инжењерству су познати још од шездесетих година двадесетог века[106]. Теорија модуларног пројектовања користи се у различитим областима менаџмента и инжењерским дисциплинама које се баве развојем архитектуре комплексних система, као што су: организациони системи, технички системи итд[107]. Искуства из ових дисциплина

---

[101] Greenfield, J. & Short, K. (2003, October). Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 16–27).
[102] https://techbeacon.com/devops/devops-scale-how-build-your-software-factory
[103] Bratman, H. & Court, T. (1975). The software factory, *Computer*, 8(5), 28–37.
[104] Griss, M. L. (1993). Software reuse: From library to factory, *IBM systems journal*, 32(4), 548–566.
[105] Cusumano, M. A. (1989). The software factory: a historical interpretation, *IEEE Software*, 6(2), 23.
[106] Parnas, D. L.(1972). On the criteria to be used in decomposing system into modules, *Communications of the ACM*, Vol. 15, No. 12, December 1972, pp. 1053–1058.
[107] Baldwin, C. Y. & Clark, K. B., Design Rules: The Power of Modularity, *MIT Press*, 2000.

уграђена су у архитектуру фабрике софтвера. Комплексни систем, па и софтвер, гради се комбиновањем модула по одговарајућим шемама – патернима, при чему се применом одговарајућих патерна постижу поузданост, скалабилност и дистрибуираност. Поједине особине основних концепата развоја процесно-оријентисаног софтвера: догађаји, сервиси, микросервиси и процеси, уграђене су у неке од најчешће коришћених софтверских патерна.

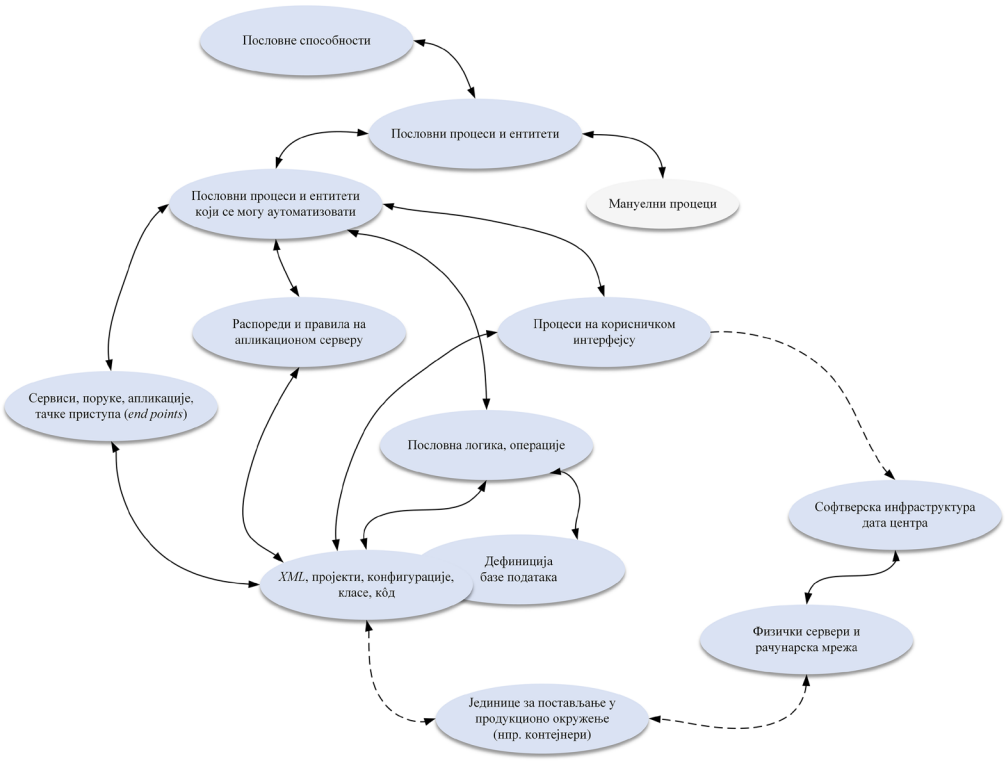
Хијерархијска декомпозиција је основна идеја у моделом вођеном развоју софтвера. Међутим, у време када је модел предложен, ограничења формалних метода и *CASE* (*Computer Aided Software Engineering*) алата условила су стагнацију и недовољну реализацију ове идеје. *UML*, који је касније предложен и на коме се инсистирало, није имао подршку за све фазе софтверског процеса. *UML* не подржава фазе тестирања, постављања у продукционо окружење и експлоатације. Такође, не постоји ни одговарајућа подршка за моделовање података. Током времена, стечено искуство са *MDD* показало је да је принцип поделе на доменски-оријентисане моделе и језике донео конкретне резултате у пракси.

MDD је основни метод развоја софтвера у фабрици софтвера. За потпуну реализацију фабрике софтвера потребно је дефинисати развојно окружење које покрива и у мери која је могућа повезује и аутоматизује све активности софтверског процеса, користећи при томе за сваки конкретан проблем најпримеренije доменски-специфичне концепте и алате. Пословно окружење је препознало значај идеје развоја доменско-специфичних језика, али је уместо развоја формалних метода и језика, предност дата развоју појединачних алата, метода и техника који су специфични за домен примене. Фабрике софтвера добијају структуру сличну данашњој почетком 21. века[108,109]. У фабрикама софтвера преовладава примена парадигме развоја софтвера оријентисаног на процесе[110], коришћењем доменски специфичних модела за спецификацију процеса.

Фабрике софтвера су најчешће везане за конкретна софтверска предузећа, домен примене, доминантну технологију, екосистем или платформу која се користи[111]. Углавном се реализују проширивањем популарних интегрисаних развојних окружења (*Integrated Development Environment – IDE*) као што су *Eclipse*[112] или

[108] Greenfield, J. (2004). The case for software factories, *Microsoft Architects Journal* (3).
[109] Greenfield, J. & Short, K. (2003, October). Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 16–27).
[110] Siy, H. P., Herbsleb, J. D., Mockus, A., Krishnan, M. & Tucker, G. T. (2001, April). Making the software factory work: Lessons from a decade of experience. In: *Proceedings Seventh International Software Metrics Symposium* (pp. 317–326). IEEE.
[111] Evans, M. W. (1989). The software factory: a fourth generation software engineering environment. *John Wiley & Sons, Inc.*
[112] Duchamp, A., Hassam, S. F., Mevel, S., Parigot, D. & Sophia, E. P. N. Software Factory on top of Eclipse: SmartTools. http://www.inria.fr/smartool

Visual Studio[113], увођењем подршке за изабране методе, технике, алате и процесе за доменски специфичне језике и моделе, компајлирање, конструкцију, конфигурацију, тестирање, отклањање грешака, постављање софтвера у продукционо окружење и праћење процеса експлоатације софтвера, са одговарајућим базама знања које се односе на најбоље праксе, искуство и начине решавања проблема. Шема и међусобни однос компоненти за типичну фабрику софтвера приказани су на Слици 7.



Слика 7. Типична шема фабрике софтвера[114]

Процеси, методи и алати фабрике софтвера углавном прате методу за управљање софтверским процесом коју примењује предузеће. У парадигми развоја софтвера оријентисаног на процесе, у највећој мери се користи *DevOps*[115].

DevOps није у потпуности стандардизован, па су процеси, методи, технике и алати фабрике софтвера најчешће изабрани на основу преовладавајуће праксе. *DevOps* приступ организацији фабрике софтвера заснива се на конвергенцији *MDD*

[113] Lenz, G., Wienands, C., Greenfield, J. & Kozaczynski, W. (2006). Practical software factories in. NET (p. 213). New York: Apress.
[114] Прилагођено према: Greenfield, J., Short, K., Cook, S. & Kent, S. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, Wiley.
[115] Pedone, C., Hartley, D., Darbin, R. & Clements, P. (2022, July). Practical Experience Applying Feature-based Product Line Engineering in a DevOps Environment: Achieving the Best of Both Worlds. In: *INCOSE International Symposium* (Vol. 32, No. 1, pp. 579–593).

и доменски-оријентисаних апликативних програмских интерфејса, софтверских оквира и екосистема. Развој апликативних програмских интерфејса постао је кључна активност за брзу имплементацију пословних идеја, па се они сматрају основним градивним блоковима дигиталне трансформације и често се третирају као посебни софтверски производи[116]. *API* омогућава повезивање различитих пословних и технолошких ресурса у дигиталне екосистеме скалабилних архитектура, који аутоматски могу решити комплексне проблеме[117]. Развој апликативних програмских интерфејса отвореног кôда у различитим областима развоја софтвера, на пример у развоју мобилних апликација, отворили су могућност примене концепата *crowdsourcing*-а[118]. Овим се организација помера од традиционалних фабрика софтвера ка отвореним иновацијама[119], где велики број заинтересованих (корисници, независни програмери, мала предузећа, стартапови и др.) могу да учествују у пројектовању и развоју софтвера.

Реално је очекивати конвергенцију, прилагођавање, а највероватније и утапање фабрике софтвера у *DevOps* и његове алате. Будући развој фабрика софтвера и *DevOps*-а биће усмерен ка континуалним иновацијама[120], коришћењу софтвера отвореног кôда, већој примени вештачке интелигенције и већој повезаности с пословним процесима и менаџментом предузећа[121].

[116] Ravichandran, A., Taylor, K. & Waterhouse, P. (2016). *DevOps for digital leaders: Reignite business with a modern devops-enabled software factory* (p. 173). Springer Nature.
[117] Li, W., Badr, Y. & Biennier, F. (2012, October). Digital ecosystems: challenges and prospects. In *Proceedings of the International Conference on Management of Emergent Digital EcoSystems* (pp. 117–122).
[118] Li, W., Tsai, WT. & Wu, W. (2015). Crowdsourcing for Large-Scale Software Development. In: Li, W., Huhns, M., Tsai, WT., Wu, W. (eds.) *Crowdsourcing. Progress in IS*. Springer, Berlin, Heidelberg. (pp. 3–23).
[119] Munir, H., Wnuk, K. & Runeson, P. (2016). Open innovation in software engineering: a systematic mapping study, *Empirical Software Engineering*, 21, 684–723.
[120] Alt, R., Auth, G. & Kögler, C. (2021). DevOps for Continuous Innovation. In: *Continuous Innovation with DevOps* (pp. 17–36). Springer, Cham
[121] Bhanot, V. (2021). Factory of the future: How to vitalize digital transformation: See five attributes of the factory of the future. IT and OT organizations, hardware and software synergy should be part of the roadmap. *Control Engineering*, 68(9), 36–38.

II УПРАВЉАЊЕ СОФТВЕРСКИМ ПРОЦЕСОМ

Софтверски процес представља комплексан скуп поступака у спецификацији, пројектовању, имплементацији, евалуацији, тестирању и еволуцији софтверских система. Састоји се од више повезаних активности и подпроцеса који се могу паралелно, циклично или секвенцијално извршавати.

Софтверски процес се у циљу анализе и ефикасног управљања представља одговарајућим моделима. Модел софтверског процеса је апстракција софтверског процеса прилагођена захтевима и ограничењима стејкхолдера који учествују у софтверском процесу, особинама реалног система, парадигми развоја софтвера и имплементационом окружењу[122]. Најчешће се представља у графичкој форми, као низ корака и редоследа извршења активности у развоју софтвера. У оквиру модела се дефинишу задаци, учесници, улоге, ток и секвенце за сваки задатак, улаз, излаз, услови и ограничења, као и индикатори перформанси за праћење успешности и квалитета[123].

Модели софтверског процеса настали су у тежњи да се превазиђе мануфактурна израда софтвера и на формалан начин приступи процесу његовог развоја. Почетком шездесетих година двадесетог века, примењиване су методе и технике које су се формирале на основу искуства из индустријске производње[124,125]. Модели софтверског процеса су се развијали током више деценија, упоредо с парадигмама развоја софтвера. Током развоја софтверског инжењерства, експериментисало се с многим моделима софтверског процеса, почевши од модела водопада, итеративних, еволутивних, агилних, па до *DevOps*-а.

Модели софтверског процеса се могу класификовати на основне и изведене. Основни модели софтверског процеса су: секвенцијални модел (водопад);

---

[122] Fuggetta, A. (2000, May). Software process: a roadmap. In: *Proceedings of the Conference on the Future of Software Engineering* (pp. 25–34).
[123] Humphrey, W.S. (1989). *Managing the software process*. Addison-Wesley Longman Publishing Co., Inc.
[124] Томашевић П. (2007). Управљање развојем софтвера (дипломски рад), ФОН.
[125] Kneuper, R. (2017). Sixty years of software development life cycle models. *IEEE Annals of the History of Computing*, 39(3), 41–54.

спирални модел; и итеративни модел. Најважнији изведени модели софтверског процеса, добијени комбинацијом основних модела, јесу: *Rapid Application Development*[126], *Extreme programming* (*XP*)[127], *Microsoft Solutions Framework* (*MSF*)[128] и више врста агилних модела[129]. Поред основних и изведених модела за управљање софтверским процесом, развијен је и *DevOps*[130] у форми комплексне методологије, која се састоји од скупа најбољих пракси и више модела, метода, техника и алата.

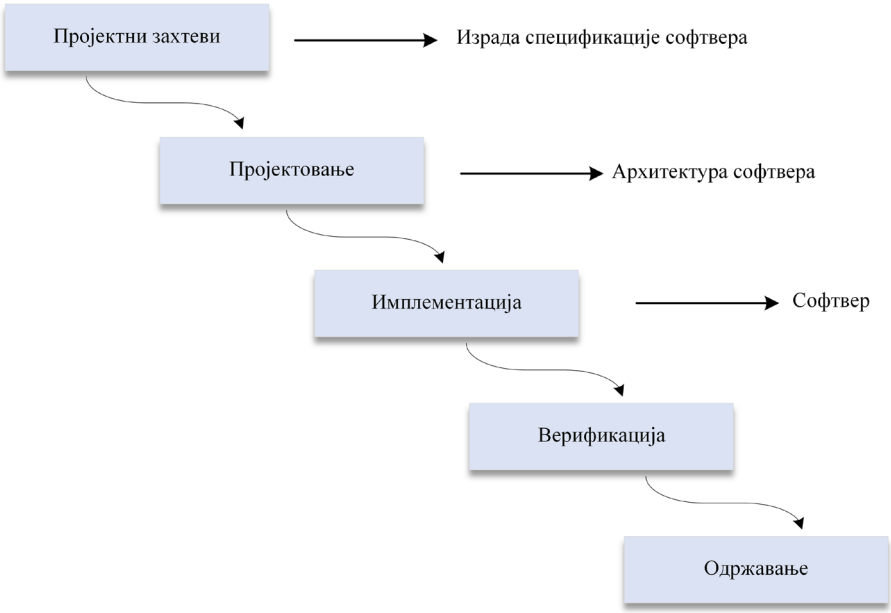
6. МОДЕЛ ВОДОПАДА

Модел водопада (*Waterfall Model*)[131] се представља скупом секвенцијалних корака у развоју софтвера, и то (Слика 8): спецификација пројектних захтева, пројектовање, имплементација, интеграција и верификација, инсталација, одржавање[132]. Ток развоја софтвера строго је секвенцијалан, без итерација и повратних спрега.

Модел водопада је показао добре резултате код огромних софтверских пројеката са стационарном структуром реалног система, много учесника и више нивоа

[126] Martin, J. (1991). *Rapid application development*. Macmillan Publishing Co., Inc.
[127] Succi, G. & Marchesi, M. (eds.). (2001). *Extreme programming examined*. Addison-Wesley Longman Publishing Co., Inc.
[128] Turner, M. (2006). *Microsoft® solutions framework essentials: building successful technology solutions*. Microsoft Press.
[129] Anwer, F., Aftab, S., Waheed, U. & Muhammad, S. S. (2017). Agile software development models TDD, FDD, DSDM, and Crystal methods: A survey. *International journal of multidisciplinary sciences and engineering*, 8(2), 1–10.
[130] Ebert, C., Gallardo, G., Hernantes, J. & Serrano, N. (2016). DevOps. *IEEE Software*, 33(3), 94–100.
[131] Royce, Winston W. (1970, August). Managing the Development of Large Software Systems, In *Proceedings of the IEEE WESCON* (pp. 1–9). Orginaly published by TRW. https://blog.jbrains.ca/assets/articles/royce1970.pdf
Royce, који се сматра аутором модела водопада, у овом раду је предлагао приступ развоју софтвера који је више итеративан него секвенцијалан. Модел водопада му је послужио као пример за ризичан и гломазан метод од којег се у анализи полази. Да би га побољшао, у модел је укључио неколико битних концепата: повратну везу између фаза и развој прототипа у току пројектовања и програмирања. Та побољшања, без обзира на анализу, примере и препоруке дате у овом раду, наредних година потпуно су занемарена и заборављена. Назив: „модел водопада“, *Royce* у овом раду није употребио, писао је о корацима секвенцијалног модела. Метод који је *Royce* предлагао кудикамо је ближи спиралном моделу. Нажалост, остао је упамћен као аутор модела водопада, о којем се у литератури писало искључиво негативно. Радови из времена када је модел водопада настао, користе га више као основу за критичка уводна разматрања, него као усвојен, примењен и анализиран метод за управљање софтверским пројектима. Иако никада није хваљен, задржао се до данашњих дана на великим пројектима где је проблем комуникације и координације доминантан, због великог броја учесника. Фазе модела описане у овом раду, са унапређењима која су уследила, послужиле су као основа за развој *DevOps* методологије, који се ефикасно може применити и за управљање великим софтверским пројектима са изразитом динамиком у пројектним захтевима.
[132] Royce, W. W. (1987, March). Managing the development of large software systems: concepts and techniques. In: *Proceedings of the 9th international conference on Software Engineering* (pp. 328–338).

комуникације и одлучивања[133]. Код информационих система код којих се пројектни захтеви континуално мењају, најчешће због динамичног развоја предузећа која мењају величину, структуру и производне програме у реалном времену, показао се изразито неуспешним. Из тог разлога је напуштен, а решења су се тражила у увођењу повратних спрега, итерација и цикличности између појединих фаза, како би се што раније, у првим фазама развоја софтвера смањио ризик и нашао прави одговор на измењене пројектне захтеве[134].



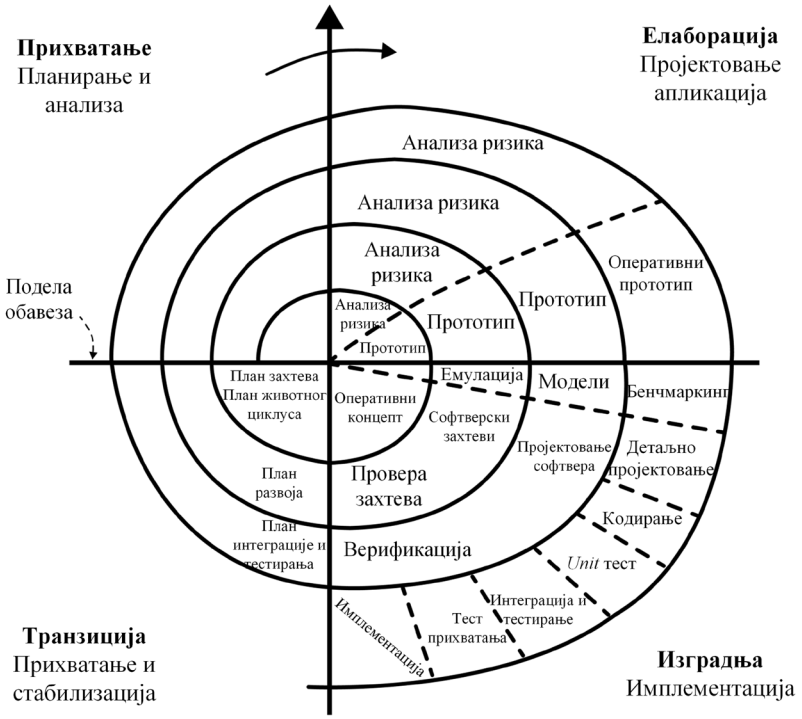
Слика 8. Модел водопада[135]

7. СПИРАЛНИ МОДЕЛ

Велики број неуспешних софтверских пројеката реализованих коришћењем методе водопада разлог је за побољшање модела и увођење формалних поступака за управљање ризиком. Спирални модел је настао у покушају да се ризик смањи и да се њиме управља, тако што се поступак развоја софтвера раздваја на мањи број секвенцијалних подпроцеса који се могу у спиралном, инкрементaлном,

[133] Petersen, K., Wohlin, C. & Baca, D. (2009). The waterfall model in large-scale development. In: *Product-Focused Software Process Improvement: 10th International Conference*, PROFES 2009, Oulu, Finland, June 15–17, 2009. Proceedings 10 (pp. 386–400). Springer Berlin Heidelberg.
[134] Ruparelia, N. B. (2010). Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, 35(3), 8–13.
[135] Прилагођено према: https://www.loginworks.com/blogs/role-waterfall-model-sdlc/

еволутивном поступку понављати и побољшавати[136]. У Спиралном моделу, развој софтвера почиње од прототипа који се у концентричним, инкременталним, спиралним циклусима унапређује и побољшава, ка коначној верзији софтвера (Слика 9).



Слика 9. Спирални модел[137]

Унапређење прототипа врши се анализом постигнутог и усклађивањем с пројектним захтевима, који се у току пројекта у сваком циклусу могу мењати, применом *top-down* методе, и анализом постигнутих резултата прототипа и његовог прилагођавања новим верзијама пројектних захтева применом *bottom-up* методе. На тај начин се грешке могу открити и отклонити на основу резултата анализе првих прототипова, у првим спиралним циклусима развоја софтвера, чиме се смањују и ризик и трошкови.

Више детаља о фазама и поступцима у Спиралном моделу могу се наћи у раду[138]. Најзначајнији резултат и новина коју је овај модел донео јесте инкрементални развој софтвера у коме је могуће управљати ризиком. Недостатак је тај што је модел комплексан, скуп за реализацију и није погодан за пројекте с ниским ризиком.

[136] Boehm, B. (1986). A spiral model of software development and enhancement. *ACM SIGSOFT Software engineering notes*, 11(4), 14–24. https://dl.acm.org/doi/pdf/10.1145/12944.12948
[137] Прилагођено према: Boehm, B. & Hansen, W.J. http://www.sei.cmu.edu/reports/00sr008.pdf
[138] Boehm, B. & Hansen, W. J. (2000). *Spiral development: Experience, principles, and refinements.* Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst. https://apps.dtic.mil/sti/pdfs/ADA382590.pdf

Такође, без прецизно дефинисаних критеријума за завршетак циклуса, спирала се може дуго понављати, шта повећава трошкове пројекта.

Значај Спиралног модела је у томе што је дефинисао еволутивну природу развоја софтвера, која је касније у различитим формама уграђивана у све наредне изведене моделе софтверског процеса.

8. ИТЕРАТИВНИ МОДЕЛ

Итеративни модел софтверског процеса настао је комбинацијом модела водопада и инкременталног, спиралног модела[139]. Итеративни модел се састоји од четири фазе:

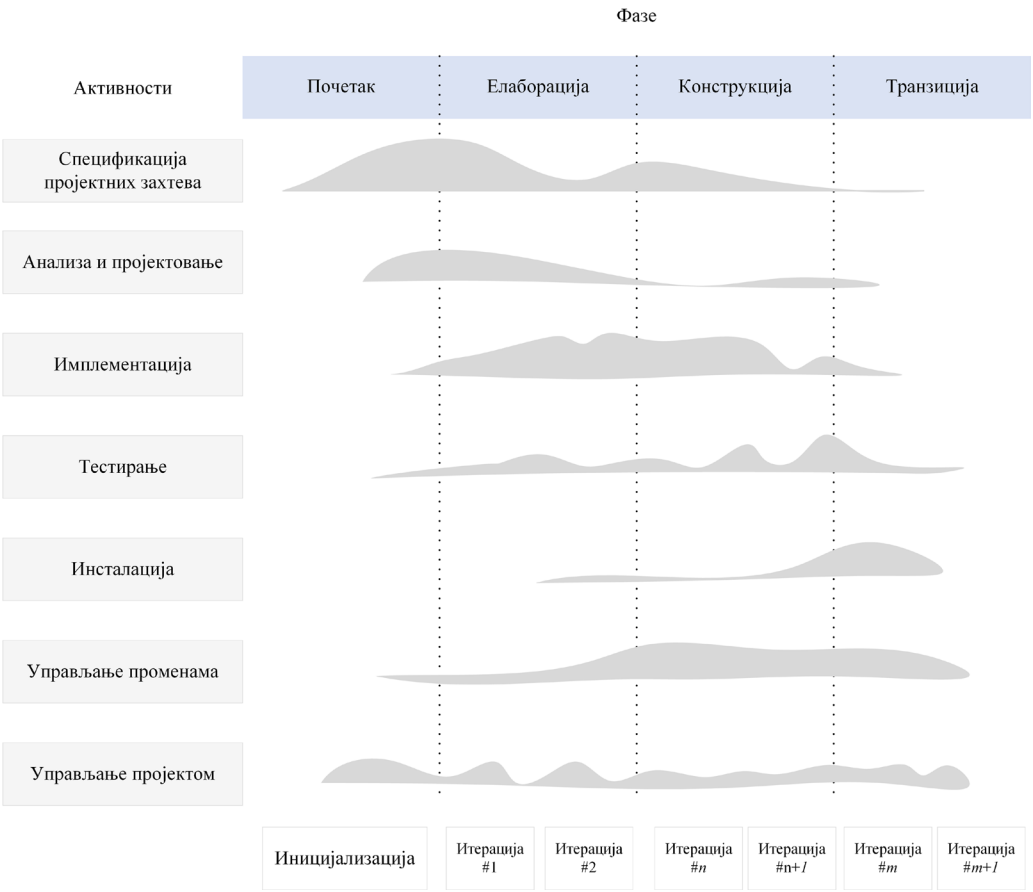
- **Почетак**. Фокус је на разумевању пословних потреба, обима проблема и изводљивости. Циљ је дефинисање визије пројекта и првих процена потребних ресурса, буџета и ризика.
- **Елаборација**. У овој фази, када су пројектни захтеви дефинисани и верификовани, пројектује се архитектура софтвера и обавља развој прототипа у циљу провере изводљивости пројекта.
- **Конструкција**. Фокус је на имплементацији. У овој фази се врши кодирање и имплементира пројектована архитектура софтвера.
- **Транзиција**. Фокус је на тестирању софтвера и обуци корисника. Када се тестирање и обука заврше, софтвер се пушта у рад.

Свака фаза се одвија у више итерација. Наредна итерација се ослања на резултате претходне. Итерације се понављају све док се софтвер не развије до нивоа који у потпуности задовољава пројектне захтеве. У свакој од фаза, у обиму који је за ту фазу потребан, итеративно се реализују следеће активности: спецификација пројектних захтева, анализа и пројектовање, имплеметација, тестирање, инсталација, управљање променама, управљање пројектом (Слика 10).

Недостаци овог модела су: комплексност и неопходност квалификованих и стручних извршилаца, пројектаната и менаџера за вођење пројекта. Из тих разлога је мање коришћен у пракси. Његов заначај је у томе што је један од основних модела и што су његови концепти уграђени у друге, новије и више коришћене моделе софтверског процеса[140].

[139] Larman, C. & Basili, V. R. (2003). Iterative and incremental developments. a brief history. Computer, 36(6), 47–56. https://www.it.uu.se/edu/course/homepage/acsd/vt08/SE1.pdf

[140] Verma, S. (2014). Analysis of strengths and weakness of SDLC models. *International Journal of Advance Research in Computer Science and Management Studies*, 2(3).



Слика 10. Удео активности у фазама Итеративног модела софтверског процеса

9. АГИЛНИ МОДЕЛИ И СКРАМ

Модел водопада у управљању софтверским пројектима по својим особинама припада линеарном менаџменту, који се заснива на способности организације да се предвиде и контролишу стања и излази. Имајући у виду промене захтева клијената и перманентне промене у пословном окружењу, од касних осамдесетих година прошлог века развијају се агилне методе засноване на нелинеарном менаџменту, које омогућавају уређивање пословног система заснованог на самоорганизацији, еволуцији и прилагођавању променљивим условима. Главна карактеристика агилних модела је да се софтвер развија итеративно-инкрементално, у кратким итерацијама, како би се што пре дошло до функционалног прототипа и верзије која се може пустити у продукцију.

Током деведесетих година развија се већи број различитих агилних модела, као што су Екстремно програмирање, Скрам (*Scrum*), *Crystal Clear* и друге *Crystal*

методе, Метод динамичког развоја система (*DSDM*), и мноштво других. Године 2001, група од седамнаест искусних програмера и стручњака за различите агилне методе донела је Агилни манифест[141], документ који је у пракси прихваћен као дефиниција агилног управљања развојем софтвера. У манифесту су дефинисане вредности и принципи који указују на разлике између традиционалних и агилних метода:

- стејкхолдери и њихове интеракције су важније од процеса и алата,
- софтвер који ради је важнији од свеобухватне документације,
- сарадња с корисником је важнија од уговора и преговора,
- реакције на промену су важније од праћења плана.

Током година које су уследиле, агилни модели су преиспитивани и редефинисани, а у текстовима и предавањима потписника *Агилног манифеста* Дејва Томаса критиковани и проглашвани мртвим – *Agile is dead*[142,143].

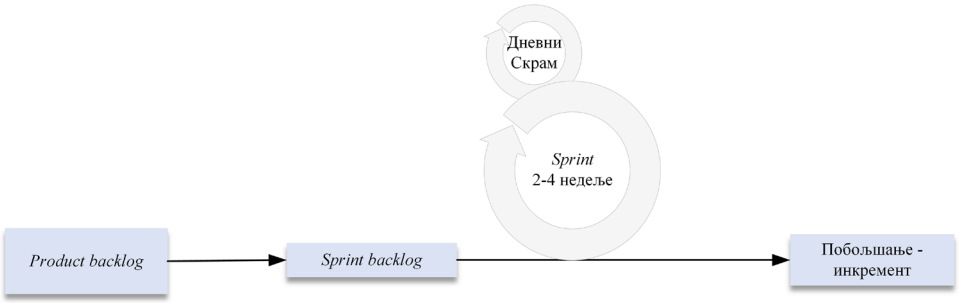
Један од покушаја за унапређење агилних метода је идеја развоја агилне канцеларије (*Agile Transformation Office*) коју предлаже Мекинси[144], као механизам за трансформацију од управљања пројектима ка управљању производима, тј. ка *DevOps*-у.

Иако се током претходних декада јављао већи број различитих агилних приступа управљању софтверским пројектима, једини који је опстао до данас је Скрам (Слика 11). Појам „скрам“ потиче из рагбија и односи се на уређену формацију играча током започињања или наставка игре. У контексту развоја нових производа овај термин су први пут употребили професори Такеучи и Нонака, 1986. године, у чланку у часопису *Harvard Business Review*[145], а као агилни метод развоја софтвера профилисали су га Садерленд и Швабер[146].

Скрам карактерише развој у кратким итерацијама – спринтовима, у малим тимовима, уз могућност примене већег броја техника и метода. У наставку текста описане су Скрам улоге, процес и артифакти[147].

---

[141] https://agilemanifesto.org/
[142] https://pragdave.me/blog/2014/03/04/time-to-kill-agile.html
[143] https://www.youtube.com/watch?v=a-BOSpxYJ9M
[144] https://www.mckinsey.com/capabilities/people-and-organizational-performance/our-insights/why-an-agile-transformation-office-is-your-ticket-to-real-and-lasting-impact
[145] Takeuchi, H. & Nonaka, I. (1986). The new new product development game. *Harvard business review*, 64(1), 137–146.
[146] Schwaber, K. & Sutherland, J. (2011). *The scrum guide*. Scrum Alliance, 21(1), 1–38.
[147] Westland, C. (2022) *Scrum Methodology: Roles, Events & Artifacts*, https://www.projectmanager.com/blog/scrum-methodology



Слика 11. Скрам

Основне улоге у Скрам тиму су власник софтверског производа (*Product Owner*), скрам мастер и развојни тим. Власник софтверског производа је једина особа која је одговорна за управљавање задацима развоја производа (*Product Backlog*). Управљање се реализује дефинисањем потребних функционалности, планирањем редоследа и приоритета развоја функционалности и, на крају, прихватањем или одбацивањем резултата рада. Скрам мастер је особа која управља пројектом, чија је главна улога да обезбеди услове да развојни тим буде функционалан и продуктиван, да се комуникација између свих одвија неометано и да нема спољних ометања. Развојни тим у Скраму обично чини 5–9 чланова различитих специјалности: програмери, тестери, пројектанти итд. Чланови тима би требало да буду ангажовани пуно радно време и да имају могућност самоорганизације. Састав тима се током пројекта може мењати, али само између спринтова.

Главни елементи управљања у Скрам процесу су[148]: спринт, планирање спринта, дневни састанак, преглед спринта и ретроспектива спринта. Скрам пројекти остварују напредак кроз низ спринтова. За време спринта, софтверски производ се пројектује, кодира и тестира. Промене у пројектним захтевима током спринта нису дозвољене. Спринт се најчешће планира тако да траје 2–4 недеље, тј. онолико колико се реално може радити без евидентиране промене у корисничким захтевима. Планирање спринта је састанак на коме цео тим анализира и процењује *product backlog*, утврђује циљ и начин реализације спринта, а постављају се и конкретни задаци (*Sprint Backlog*), као и процена потребног времена за њихову израду. Дневни Скрам састанци намењени су решавању конкретних текућих проблема. Састанци трају кратко, обично петнаестак минута. Циљ дневних састанака је да сваки члан тима информише остале о томе шта је урађено јуче, шта ће бити урађено данас, и да ли има неких препрека у раду. Преглед спринта је неформални састанак на коме се презентује оно што је урађено за време спринта, у форми демонстрације нових функционалности и пројектоване архитектуре. Ретроспектива спринта је састанак на коме цео тим, заједно са скрам мастером и власником софтверског производа, анализира завршени спринт. Типично траје 15–30 минута.

[148] У литератури на енглеском језику уобичајен је израз *scrum events* иако нису у питању догађаји. Планирање спринта, дневни састанак, преглед спринта и ретроспектива спринта су састанци, а током спринта се ради на извршењу задатака.

Артефакти карактеристични за Скрам су *product backlog*, *sprint backlog*, побољшање, тј. инкремент и *burndown* дијаграми. *Product backlog* је уређена листа у том тренутку свега познатог, што је потребно реализовати у софтверском производу. То је једини извор захтева за све промене које ће бити уграђиване у софтверски производ. Власник софтверског производа је одговоран за *product backlog*, његову садржину, доступност и редослед послова. *Product backlog* се у току рада на пројекту динамички мења, како би се у ходу обезбедило да софтверски производ буде одговарајући, конкурентан и користан. Из *product backlog*-а се издваја један део који треба да се имплементира током спринта, и тај део представља *sprint backlog*. *Sprint backlog* је реална слика посла који тим за развој планира да оствари у току спринта. *Sprint backlog* дефинише задатке развојног тима. Чланови тима се сами опредељују за задатке и свакодневно процењују преостали посао. Побољшање (*increment*) је збир свих *product backlog* ставки које су завршене током једног спринта. На крају спринта, ново побољшање се означава као завршено (*done*), што значи да софтвер треба да буде у употребљивом стању, без обзира на то да ли ће власник производа одлучити да га прихвати или не прихвати као нову верзију. *Burndown* дијаграми су графички приказ преосталог дела посла у спринту.

За велика предузећа развијене су посебне варијанте скрама, као што су *SAFe*, *Scrum-at-Scale*, и друге[149], код којих се фокус помера ка портфолију производа, усаглашавању пословних и *IT* циљева (*business-IT alignment*), бољој координацији тимова на нивоу целог предузећа и унапређењу комплетног пословања[150]. Ово представља значајан искорак од агилних метода ка *DevOps*-у.

10. DEVOPS

Још од педесетих година двадесетог века, информационе технологије и софтвер су развијани и примењивани реактивно, покушавајући да у организацији софтверског процеса прате захтеве и промене у теорији менаџмента и организације. Иако се у науци о менаџменту, функционална организација као доминантан облик организовања предузећа још од седамдесетих година полако напушта, а фокус помера ка управљању процесима, управљању променама и нешто касније ка управљању производима, у најцитиранијим радовима о управљању софтверским пројектима из деведесетих година, у односу на тај тренд доста се каснило, па се бизнис-*IT* усклађивање на функционалном нивоу тада још увек помињало као препоручени начин усклађивања[151]. Као ретроградни отклон од новог процесног начина

[149] Conboy, K. & Carroll, N. (2019). Implementing large-scale agile frameworks: challenges and recommendations. *IEEE software*, 36(2), 44–50.
[150] Laanti, M. (2014). Characteristics and principles of scaled agile. In *Agile Methods. Large-Scale Development, Refactoring, Testing, and Estimation: XP 2014 International Workshops*, Rome, Italy, May 26–30, 2014, Revised Selected Papers 15 (pp. 9-20). Springer International Publishing.
[151] Henderson, J. C. & Venkatraman, H. (1999). Strategic alignment: Leveraging information technology for transforming organizations. *IBM systems journal*, 38(2.3), 472–484.

размишљања, тада доминантни објектно-оријентисани приступ развоју софтвера, додатно је повећао јаз између бизниса и софтверске подршке. Ригидне објектно-оријентисане *IT* инфраструктуре и нефлексибилна хијерархијска организација предузећа допринеле су томе да многа предузећа у том периоду нису могла да постигну агилност и флексибилност коју наступајућа дигитална трансформација захтевала, ни да издрже тржишну утакмицу. Један број предузећа је покушавао да овај проблем реши развијајући „дупли *IT*“[152], где су спори и комплексни наслеђени системи задржани као основна инфраструктура, а паралелно су развијани и често аутсорсовани нови, модернији *IT* системи који се баве корисницима. То је довело тек до незнатних помака.

У трасформацији софтверског процеса ка потпуно процесном приступу, из наведених разлога се много каснило. Тек у новијим научним радовима, као антипатерн, експлицитно се препознаје функционална зависност између бизниса и софтвера. Као начин да се превазиђу проблеми који проистичу из примене антипатерна, препоручује се процесни приступ[153]. Софтверски процес обухвата две функционалне целине предузећа: *IT* сектор (или одељење) и сектор развоја софтвера. У трансформацији *IT* сектора ка процесној организацији, скуп пословних процеса је подељен на *IT* операције (*IT operations – Ops*): управљање хардверском, софтверском и мрежном инфраструктуром, склабилношћу, сигурности, експлатацијом софтвера и корисничком подршком. Сектор развоја софтвера (*Software Development – Dev*) пословне процесе реализује агилним или неким другим моделом, који се односи на: планирање, кодирање, тестирање и испоруку софтвера. Повезивањем пословних процеса ова два сектора у једну целину омогућила би се реализација континуалног развоја и испоруке софтвера кроз јединствен софтверски процес и задовољио један од најважнијих захтева у дигиталној трансформацији. У ту сврху, предложена је методологија која је настала као скуп тада најбољих пракси, модела, техника и алата, једноставно названа *DevOps*[154]. Избор конкретних модела, метода, техника и алата за реализацију процеса у *DevOps* методологији није фиксно одређен и у пракси се може прилагођавати потребама предузећа. *DevOps*, од првих фрагментарних примена 2009. године[155], с временом се трансформисао ка усклађивању менаџмент процеса целокупног предузећа и процеса у *IT* сектору. Трансформација се односи на промене у организацији и организационој култури, које подразумевају креирање крос-функционалних тимова у којима запослени из различитих пословних функција раде заједно у циљу континуалног пружања

[152] Horlach, B., Drews, P. & Schirmer, I. (2016). Bimodal IT: Business-IT alignment in the age of digital transformation. *Multikonferenz Wirtschaftsinformatik* (MKWI), 3, 1417–1428.
[153] Gouigoux, J. P. & Tamzalit, D. (2022). Business-IT alignment anti-patterns: a thought from an empirical point of view. In: *Advances in Information Systems Development: Crossing Boundaries Between Development and Operations in Information Systems* (pp. 67–90). Cham: Springer International Publishing.
[154] Wettinger, J., Breitenbücher, U. & Leymann, F. (2014). Devopslang–bridging the gap between development and operations. In: *Service-Oriented and Cloud Computing: Third European Conference*, ESOCC 2014, Manchester, UK, September 2–4, 2014. Proceedings 3 (pp. 108–122). Springer Berlin Heidelberg.
[155] DevOps Days 2009. https://legacy.devopsdays.org/events/2009-ghent/

нових вредности клијентима[156]. За постизање постављених циљева: континуална интеграција, континуална испорука и континуално постављање софтвера у продукционо окружење (*Continuous Integration*, *Continuous Delivery and Continuous Deployment*)[157], неопходна је ефикасна комуникација и сарадња свих учесника у софтверском процесу.

Прелазак на процесну организацију у развоју софтвера није био једноставан. За ефикаснији прелазак са функционалне на процесну организацију, Британска централна државна телекомуникациона и рачунарска агенција (*Central Computer and Telecommunications Agency – CCTA*) издала је 1989. године скуп препорука и најбољих пракси са циљем да стандардизује скуп процеса у области *IT* операција и усклади их с потребама предузећа, под називом: *Information Technology Infrastructure Library* (*ITIL*)[158] и препорукама за управљање *IT* сервисима (*IT service management – ITSM*). Примена ових препорука у софтверским предузећима имала је велики успех у претходним деценијама. *ITIL* препоруке су усклађиване с трендовима и имале су више издања. У 2019. години је издата четврта верзија *ITIL* препорука која укључује и препоруке за имплементацију *DevOps* процеса и примену рачунарства у облаку[159].

Поред процесног приступа, усвајање *DevOps*-а за управљање софтверским процесом захтева промене и у начину пројектовања софтвера, архитектури софтвера и начину одржавања[160]. Потребно је моделирати структуру и промене у софтверском систему, идентификовати потребе за паралелизацијом, као и места оптерећења.

Најзначајније промене које су настале под утицајем *DevOps*-а су[161]:

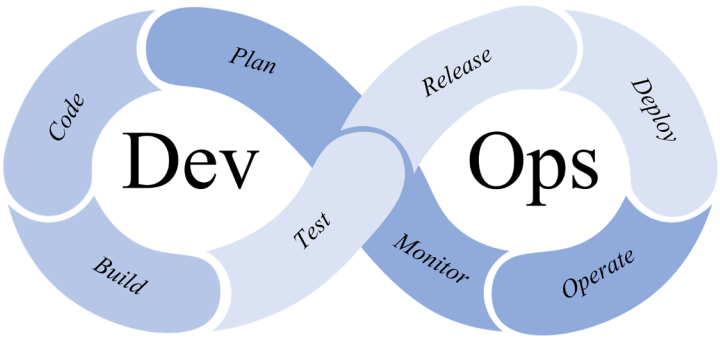
- **Микросервисна архитектура**[162]. Како би се омогућио ефективан *CI/CD* ток, неопходно је организовати софтвер у мале, слабо спрегнуте сервисе без дељених компоненти, који се могу независно развијати, тестирати, постављати у продукционо окружење и одржавати.
- **Инфраструктура и сервиси рачунарства у облаку.** За дефинисање виртуелне инфраструктуре уведен је нов концепт: инфраструктура као кôд (*Infrastruc-*

[156] Ebert, C., Gallardo, G., Hernantes, J. & Serrano, N. (2016). DevOps. *IEEE Software*, 33(3), 94–100.
[157] Bass, L., Weber, I. & Zhu L. (2015) *DevOps: A Software Architect's Perspective*, Addison-Wesley Professional.
[158] https://www.itlibrary.org/
[159] https://www.atlassian.com/itsm/itil
[160] Shahin, M. & Babar, M. A. (2020, June). On the role of software architecture in DevOps transformation: An industrial case study. In: *Proceedings of the International Conference on Software and System Processes*, 175–184.
[161] Leite, L., Rocha, C., Kon, F., Milojicic, D. & Meirelles, P. (2019). A survey of DevOps concepts and challenges. *ACM Computing Surveys* (CSUR), 52(6), 1–35.
[162] Balalaie, A., Heydarnoori, A. & Jamshidi, P. (2016). Microservices architecture enables DevOps: Migration to a cloud-native architecture. *IEEE Software*, 33(3), 42–52.

ture as Code)[163]. Скалабилни хардверски и софтверски ресурси рачунарства у облаку омогућују *DevOps* тиму брз развој, тестирање и постављање апликације у продукцију. Из бројних примера *DevOps* најбоље праксе, показало се да примена виртуелизоване инфраструктуре доприноси успеху *XaaS* апликација[164].

- **Побољшање стабилности и сигурности софтвера.** У традиционалним приступима за стабилност и сигурност софтвера задужен је тим за *IT* операције. Сада се ови проблеми могу решавати и у фазама пројектовања и кодирања за које је задужен развојни тим. Последица тога је да за примену *DevOps*-а треба обезбедити компетентне и образоване софтверске инжењере за све фазе софтверског процеса. Позитиван резултат је знатно побољшање стабилности и сигурности софтвера.

Графички приказ *DevOps*-а са свим фазама софтверског процеса: анализе, пројектовања, кодирања, интеграције, тестирања, испоруке, експлоатације и праћења[165], дат је на Слици 12.



Plan – планирање
Code – кодирање
Build – конструкција софтвера
Test – тестирање
Release – издавање нове верзије софтвера
Monitor – надгледање
Operate – експлоатација
Deploy – постављање у продукционо окружење

Слика 12. DevOps фазе софтверског процеса[166]

Алати за *DevOps* нису формално прописани и зависе од праксе и искуства у организацији, као и од избора софтверске платформе или екосистема. За све фазе софтверског процеса, поред комерцијалних, постоји и велики број алата отвореног кôда. Преглед најчешће коришћених алата за *DevOps* дат је на Слици 13.

[163] Artac, M., Borovssak, T., Di Nitto, E., Guerriero, M. & Tamburri, D. A. (2017, May). DevOps: introducing infrastructure-as-code. In: *2017 IEEE/ACM 39th international conference on software engineering companion* (ICSE-C), 497–498. IEEE.
[164] Battina, D. S. (2020). DevOps, A New Approach to Cloud Development & Testing. *International Journal of Emerging Technologies and Innovative Research* (www. jetir.org), 7(8), 982–985.
[165] Leite, L., Rocha, C., Kon, F., Milojicic, D. & Meirelles, P. (2019). A survey of DevOps concepts and challenges. *ACM Computing Surveys* (CSUR), 52(6), 1–35.
[166] Прилагођено према: Storbakken, M. (2020) *DevOps: Principles and Outcomes*, https://blogs.vmware.com/cloud/2020/10/05/devops-principles-outcomes/

Слика 13. Преглед често коришћених алата за DevOps[167]

У наставку су укратко описане главне фазе у *DevOps*-у и наведени алати који се у свакој од фаза најчешће користе[168].

10.1 Планирање (Plan)

Имајући у виду да је *DevOps* настао као резултат примене и надградње агилних метода развоја софтвера, за фазу планирања преузети су поступци карактеристични за агилне методе који су се најбоље показали у пракси. Фаза планирања укључује припрему за развој, прикупљање информација од стејкхолдера, припрему мапе развоја (*Roadmap*), анализу и дефинисање списка захтева (*Product Backlog*), планирање ресурса, постављање приоритета и дефинисање метрика за праћење тока софтверског процеса. Уобичајено је да метрике укључују: учесталост испоруке софтвера, време потребно за измене у софтверу, учесталост отказа, време опоравка сервиса, поузданост и оцену доприноса организационим циљевима[169]. Кључне улоге у овој фази имају менаџер пројекта или менаџер производа.

Планирање у *DevOps*-у је перманентна активност која се одвија у итеративно инкрементaлним циклусима, заједно са фазом кодирања (*code*).

Спецификација захтева стејкхолдера, међу којима су и корисници, дефинише се у форми документа (*backlog*) са јасно одређеним јединицама (*unit*), којима су

[167] Прилагођено према: https://devopsartisan.ro.digital/blog/how-to-use-ansible-for-infrastructure-provisioning-and-configuration

[168] Bogdanović, Z., Despotović-Zrakić, M., Barać, D., Labus, A. & Radenković, M. (2023). The Role of DevOps in Sustainable Enterprise Development. In *Sustainability: Cases and Studies in Using Operations Research and Management Science Methods*, 217–237, Cham: Springer International Publishing.

[169] Accelerate State of DevOps Report (2022). https://services.google.com/fh/files/misc/2022_state_of_devops_report.pdf

додељени приоритети. Додела приоритета јединицама *backlog*-а је најосетљивији поступак и критична активност при дефинисању захтева стејкхолдера. Јединице *backlog*-а највишег приоритета се прво пројектују, имплементирају и у итерацијама унапређују, док се јединице нижег приоритета развијају касније, када се на основу тока целог пројекта доноси коначна одлука о њиховом увођењу.

Јединице *backlog*-а се састоје од корисничких захтева, ограничења везаних за архитектуру софтвера, имплементациону платформу, дистрибуираност, скалабилност, поузданост, од метрика за оцену квалитета софтвера и уочених грешака у току развоја софтвера. Најнижа по хијерархији, али најзначајнија градивна компонента *backlog*-а, јесте „дискусија корисника“ (*user story*). Дискусије корисника описују пословне процесе из угла крајњег корисника. За сваку дискусију корисника дефинише се посебан задатак или део задатка (*task* или *subtask*) који је основа за даљи поступак пројектовања софтвера. На основу приоритета, који се додељује дискусији корисника, дефинишу се и приоритети задатака. *Backlog* се може лако мењати и ажурирати. Довољно је на почетку уврстити дискусије корисника које су најзначајније и које имају највиши приоритет, и кренути у итеративно инкрементaлан процес развоја софтвера.

Документ у коме се представља очекивана динамика реализације задатака из *backlog*-а назива се „мапа пута“ (*Roadmap*). Мапа пута се припрема, одржава и ажурира упоредо са верзијама *backlog*-а. *Backlog* и *roadmap* одржава власник софтверског пројекта или производа. У њиховом дефинисању и ажурирању учествују сви стејкхолдери са својим очекивањима, ограничењима и приоритетима.

На основу задатака из *backlog*-а, приступа се даљим фазама пројектовања: моделирању података, моделирању архитектуре, моделирању корисничких интерфејса[170], спецификацији компоненти, модула и њихове интеракције. Резултат је апстрактни модел софтверског система, на основу којег се софтвер може имплементирати.

У овој фази се највише користе алати за управљање пројектима, за комуникацију и сарадњу, као што су *Jira*, *Slack*, *Mattermost*, *OpenProject* или *Asana*.

10.2 Кодирање (Code)

Ова фаза се односи на пројектовање софтвера и писање програма. Често се фаза планирања и фаза кодирања преплићу са скупом послова који се односе на анализу корисничких захтева. У фази кодирања дефинише се архитектура софтвера која одговара захтевима корисника и приступа се имплементацији. Све промене се прате применом система за управљање верзијама кôда, интегрисаног с радним окружењем. Системи за управљање верзијама кôда омогућују праћење историје

[170] Cockton, G. & Gram, C. (eds.). (1996). *Design principles for interactive software*. Springer Science & Business Media.

свих промена у кôду и конфигурацији, лакше утврђивање узрока настанка грешака, бекап и интеграцију делова кôда које пишу различити програмери. Када програмер заврши једну фазу рада, он чува кôд у заједничком репозиторијуму. У циљу брзе испоруке, *DevOps* пракса је да програмери једном дневно (некада и чешће) чувају кôд у систему за управљање верзијама кôда. За програмирање се обично користе интегрисана развојна окружења (*IDE*), као што су *Eclipse* или *MS Visual Studio*. Често коришћени системи за управљање верзијама кôда су *Git*, *SVN* и други.

10.3 Конструкција софтвера (Build)

Конструкција софтвера је фаза у којој се врши интеграција кôда који пишу различити програмери и тимови, анализа и решавање зависности (*dependencies*), инсталирање и подешавање библиотека и компајлирање. У овој фази се обавља и верификација кôда, проверава се да ли је кôд заснован на усвојеним најбољим праксама, проверавају се сигурносни елементи и сл. Резултат ове фазе је софтверски артефакт (програм) у форми пакета спремног за продукцију[171]. По завршетку ове фазе, почиње тестирање. Континуална изградња софтвера омогућује да се потенцијални проблеми интеграције новог кôда детектују рано, и грешке исправе. Познати *build* алати су *Gradle*, *Maven*, *Ant* и други.

10.4 Тестирање софтвера (Test)

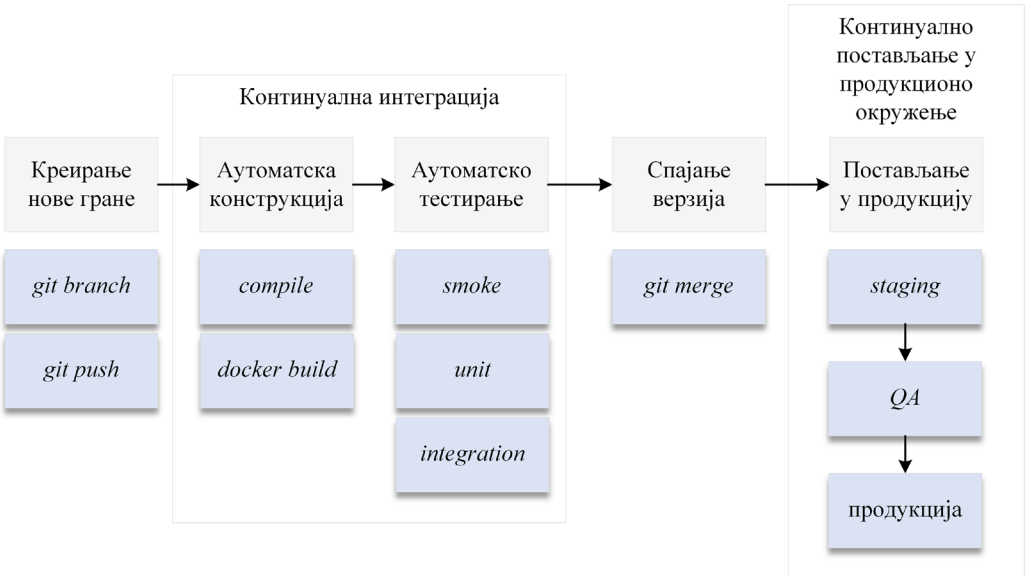
Уколико је конструкција софтвера била успешна, детаљно тестирање се покреће аутоматски. Окружење за тестирање се најчешће обезбеђује као део процеса имплементације, коришћењем инфраструктуре у виду услуге (*Infrastructure as a Service – IaaS*) и контејнерске архитектуре. Тестирање се изводи путем аутоматизованих тестова, а када је то потребно, и мануелним тестирањем. У овој фази се врше и тестови који нису ограничени на функционалност, укључујући тестове *UX/UI*, скалабилности, сигурности, компатибилности, и других аспеката употребе софтвера.

10.5 Издавање нове верзије софтвера (Release)

Ова фаза представља прекретницу у *DevOps* процесу, у којој је софтвер успешно прошао тестирање и постао спреман за испоруку и постављање у продукционо окружење. Фазе кодирања, конструкције, тестирања и издавања нове верзије се у *DevOps* пракси често интегришу у аутоматизоване *CI/CD* (*Continuous Integration / Continuous Delivery pipeline*) токове[172] (Слика 14). *CI/CD* интеграција омогућава

[171] Лазаревић, С., Марковић, Д. и Кундовић, И. (2011). *Конструкција софтвера рефакторисањем програмског кôда до нивоа узора*, стр. 162, ФОН.
[172] Pulkkinen, V. (2013, August). Continuous deployment of software. In *Proc. of the Seminar* (Vol. 58312107, p. 46).

бржу испоруку апликација тако што се поједине фазе животног циклуса софтвера аутоматизују. *Jenkins* је један од најчешће коришћених алата за аутоматизацију *CI/CD* токова.



Слика 14. Пример CI/CD тока

Код аутоматизованих *CI/CD* токова, након издавања нове верзије софтвера, потребно је да се софтвер, а у неким случајевима и његово продукционо окружење, преведе из текућег, старог стања, у ново стање или нову верзију. Прелаз са старе на нову верзију софтвера назива се *forward transition*, а синоними су *upgrade* или *roll-forward*. Уколико дође до проблема у раду с новом верзијом софтвера, потребно је обезбедити повратак на претходну верзију, *backward transition*, синоними су *downgrade* и *rollback*. При преласку са старе на нову верзију, потребно је планирати и детаљно тестирати све опције, тако да *roll-forward* и по потреби *rollback* буду без грешака и застоја у раду софтвера. Нова верзија софтвера, која је компатибилна с претходном и коју је могуће једноставно вратити на претходну верзију, назива се *backwards compatible*. Добра је пракса да се приликом издавања нове верзије софтвера обезбеди да софтвер буде *backwards compatible*.

10.6 Постављање софтвера у продукционо окружење (Deploy)

У овој фази је завршено тестирање и нова верзија софтвера је спремна за примену у продукционом окружењу. Ова фаза се састоји од[173]:

1. Обезбеђења инфраструктуре (*Provision*). Инфраструктурни ресурси се обезбеђују

[173] Coupaye, T. & Estublier, J. (2000, March). Foundations of enterprise software deployment. In: *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering* (pp. 65–73). IEEE.

постављањем или изнајмљивањем физичких сервера, виртуелних машина, система за управљање контејнерима, мрежних и ресурса за складиштење података.

2. Конфигурисања (*Configure*). Конфигуришу се параметри инфраструктуре и продукционих окружења, апликациони сервери, сигурност, расположивост, перформансе, мрежни и други ресурси.
3. Постављања софтвера у продукционо окружење (*Deploy*). Представља инсталацију или надоградњу софтвера на већ припремљене и конфигурисане ресурсе.

Постављање софтвера у продукционо окружење може се аутоматизовати коришћењем *Jenkins* алата. Могу се користити и алати за аутоматско управљање конфигурацијама као што су *Ansible*, *Chef* и *Puppet*, или алати провајдера рачунарства у облаку (*Cloud provider*), од кога ће се изнајмити виртуелни ресурси за рад софтвера[174].

10.7 Експлоатација и оперативни рад софтвера (Operate)

У овој фази ново издање софтвера је оперативно, постављено је у продукционо окружење и користи се. Подешавање ангажованих ресурса врши се у реалном времену, у складу са оптерећењем и постављеним критеријумима перформанси и трошкова. Управљање перформансама може бити: проактивно, планирањем, или реактивно, праћењем оптерећења ресурса и реаговањем на догађаје. Окружење се може аутоматски скалирати повећањем или смањивањем ангажованих ресурса, коришћењем *Kubernetes* или других алата. На основу прикупљања повратних информација од корисника, може се утицати на одржавање и будући развој производа[175].

10.8 Надгледање и праћење (Monitor)

У последњу фазу *DevOps*-а спадају надгледање и праћење мреже, сервера и апликација. У овој фази се користе информације које се односе на кориснике и њихово понашање, коришћене ресурсе, остварене перформансе, могуће грешке, али уочавају се и уска грла у раду софтвера. Детектоване грешке се, ако је могуће, аутоматски исправљају, или се обавештавају запослени задужени за њихове исправке. За континуално надгледање се користе алати као што су: *Nagios*, *Prometheus*, *New Relic* и др. У овој фази се подешавају и скалирају потребни ресурси за рад информационог система.

---

[174] Singh, C., Gaba, N. S., Kaur, M. & Kaur, B. (2019, January). Comparison of different CI/CD tools integrated with cloud platform. In: *2019 9th International Conference on Cloud Computing*, Data Science & Engineering (Confluence), 7–12. IEEE.

[175] Bogdanović, Z., Despotović-Zrakić, M., Barać, D., Labus, A. & Radenković, M. (2023). The Role of DevOps in Sustainable Enterprise Development. In *Sustainability: Cases and Studies in Using Operations Research and Management Science Methods*, 217–237, Cham: Springer International Publishing.

11. ЕВАЛУАЦИЈА СОФТВЕРСКОГ ПРОЦЕСА

Евалуација софтверског процеса у пракси се често реализује применом *CMMI* модела (*Capability Maturity Model Integration*), чији је циљ да кроз оцену нивоа зрелости идентификује снаге и слабости постојеће организације и укаже на могућност унапређења[176]. Зрелост предузећа се по овом моделу оцењује на петостепеној скали, при чему највећи ниво подразумева дефинисан и потпуно оптимизован софтверски процес, који се прати индикаторима перформанси. *CMMI* је усаглашен с другим стандардима и моделима квалитета софтвера, као што су *CMM*, *ISO* 12207 и *ISO* 15504.

Усвајање *DevOps* приступа не може се реализовати као један пројекат, већ је неопходно да предузећа координисано и планирано имплементирају низ процеса и алата, и унапреде организациону културу. Поједине консултантске куће предлажу *DevOps* моделе зрелости[177] као оквир за евалуацију увођења *DevOps* приступа, праћењем више димензија које се односе на људске ресурсе, процесе и алате. *CMMI* се због цене ангажовања консултаната, потребних организационих промена и потребног времена за увођење, показао као непогодан за мала софтверска предузећа, а многа средња и велика софтверска предузећа су одустала због недостатка мотивације и процене недовољног бенефита у односу на цену увођења[178].

12. ОТВОРЕНЕ ИНОВАЦИЈЕ И СОФТВЕР ОТВОРЕНОГ КÔДА У DEVOPS-У

Традиционална организација предузећа ослања се на затворени начин иновирања, који подразумева да се коришћењем интерних ресурса, далеко од очију конкуренције, генеришу и тестирају идеје и развијају производи. Растући трошкови технолошког развоја и краћи век трајања нових производа, праћени променама у изворима инвестиција и доступности експерата, учинили су модел затворених иновација изузетно неефикасним и неисплативим.

Као алтернативно решење, развијају се модели отворених иновација. Отворене иновације дефинишу се као управљање токовима знања изван граница предузећа путем дистрибуираног иновационог процеса[179], што укључује преузимање екстерних знања и коришћење туђих открића и њихово укључивање у интерне иновационе

[176] Silva, F. S., Soares, F. S. F., Peres, A. L., De Azevedo, I. M., Vasconcelos, A. P. L., Kamei, F. K. & de Lemos Meira, S. R. (2015). Using CMMI together with agile software development: A systematic review. *Information and Software Technology*, 58, 20–43.
[177] Menzel, G., Macaulay, A., Ajith, N. C. & Dhanesh, A. (2015). *DevOps-The Future of Application Life-cycle Automation*. Capgemini. Com, 24.
[178] Staples, M., Niazi, M., Jeffery, R., Abrahams, A., Byatt, P. & Murphy, R. (2007). An exploratory study of why organizations do not adopt CMMI. *Journal of systems and software*, 80(6), 883–895.
[179] Chesbrough, H. & Bogers, M. (2014). Explicating open innovation: Clarifying an emerging paradigm for understanding innovation. *New Frontiers in Open Innovation*. Oxford: Oxford University Press, 3–28.

процесе, као и пласирање интерно генерисаних знања ка другим предузећима[180].

У циљу подстицања синергије и интегративних процеса у иновацијама, Европска комисија је предложила концепт Отворених иновација 2.0[181], који се темељи на сарадњи актера из шире друштвене заједнице у заједничком креирању вредности и креацији дигиталних иновационих екосистема. У иновационом процесу учествују предузећа, јавна управа, академска заједница и цивилни сектор, који балансирају коришћење интерних и екстерних ресурса, унапређују организациону културу, развијају иновативне пословне моделе базиране на информационим технологијама, афирмишу и развијају људске ресурсе[182].

Као главни изазови у отвореним иновацијама јављају се: проналажење креативних начина за експлоатацију интерних иновација, укључивање екстерних иновација у интерни развој и мотивисање екстерних извора иновација да учествују у иновационом процесу[183]. Поставља се и питање како и зашто предузећа треба да улажу новац у истраживања, ако ће резултате слободно делити с конкуренцијом[184]. Ово је чест случај у развоју софтвера отвореног кôда.

Софтвер отвореног кôда (*Open Source Software*) настао је као одговор на модел власничког комерцијалног софтвера, са циљем да се смањи зависност купаца од произвођача софтвера, да се омогући заједнички развој, као и да се обезбеди право на стално коришћење софтвера. Развојем интернета, омогућена је лакша дистрибуција софтвера отвореног кôда, али и учешће у развоју софтвера великог броја програмера из целога света.

Популаризација концепта отворених иновација утицала је да све више решења софтвера отвореног кôда долази из софтверских предузећа која су имала затворене начине иновирања, као што су *Гугл* и *Мајкрософт*. Учешћем у развоју софтвера отвореног кôда, ова предузећа покушавају да изграде широко доступне софтверске екосистеме у којима се приходи генеришу, не кроз продају софтвера, већ кроз развој екосистема у коме се имплементирају нови и иновативни пословни модели и генерише нова вредност. Будући развој *DevOps*-а свакако ће бити усмерен ка отвореним иновацијама и интеграцији платформи за отворене иновације у *DevOps*[185].

---

[180] Сарић Ж. (2022). Модел управљања иновационим пројектима у дигиталној трансформацији, Докторска дисертација, http://t.eteze.bg.ac.rs/application/showtheses?thesesId=8841

[181] Curley, M. & Salmelin, B. (2013). Open innovation 2.0: a new paradigm. OISPG White Paper, 1–12.

[182] Sarić, Ž., Obradović, V., Bogdanović, Z., Labus, A. & Mitrović, S. (2022). Crowd-based open innovation in telco operators: readiness assessment for smart city service development. *Serbian Journal of Management*, 17(1), 179–196.

[183] West, J. & Gallagher, S. (2006). Challenges of open innovation: the paradox of firm investment in open-source software. *R&D Management*, 36(3), 319–331.

[184] West, J. & Bogers, M. (2014). Leveraging external sources of innovation: a review of research on open innovation. *Journal of product innovation management*, 31(4), 814–831.

[185] Sarić, Ž., Obradović, V., Bogdanović, Z., Labus, A. & Mitrović, S. (2022). Crowd-based open innovation in telco operators: readiness assessment for smart city service development. *Serbian Journal of Management*, 17(1), 179–196.

III АНАЛИЗА И ПРОЈЕКТОВАЊЕ

Анализа (*analysis*) и пројектовање (*design*) су фазе у развоју производа које је софтверско инжењерство преузело из других инжењерских дисциплина. Анализа се односи на домен проблема који треба решити. Излаз из фазе анализе је документ са спецификацијом захтева и скупом ограничења. Пројектовање је процес у коме се из домена проблема прелази у домен решења. Излаз из фазе пројектовања је пројекат, документ који садржи упутство за имплементацију софтвера који испуњава постављене захтеве и ограничења[186].

Функционални захтеви из анализе пословног система односе се на понашање софтвера. Нефункционални захтеви се односе на дистрибуираност, паралелизацију, скалабилност, поузданост, сигурност и квалитет софтвера.

Пројектовање обухвата пројектовање архитектуре која примарно решава нефункционалне захтеве, и пројектовање софтвера које обухвата модуле и сервисе, у којима се примарно решавају функционални захтеви. Граница између ова два поступка пројектовања није јасно одређена. За функционалности модула и сервиса је потребно дефинисати архитектуру, а за интеграцију и друге нефункционалне захтеве из архитектуре софтвера потребно је пројектовати модуле и интерфејсе за њихову имплементацију[187].

У софтверу оријентисаном на процесе, архитектура интеграције која решава нефункционалне захтеве истовремено је и архитектура функционалности, па је и процес пројектовања архитектуре и функционалности обједињен. С обзиром на то да је ова књига намењена развоју софтвера оријентисаног на процесе, надаље ће се под термином „Пројектовање софтвера“ подразумевати обједињено пројектовање архитектуре и функционалности.

Анализа и пројектовање су поступци засновани на апстракцији, па не постоји ни оптималан, ни једнозначан метод за анализу система, спецификацију пројектних

---

[186] McPhee, K. (1996). Design theory and software design.

[187] Jaiswal, M. (2019). Software architecture and software design, *International Research Journal of Engineering and Technology (IRJET) e-ISSN*, 2395–0056.

захтева и пројектовање софтвера[188]. Методе анализе система и пројектовања софтвера развијале су се упоредо с развојем софтверског инжењерства. Током протеклих деценија, у програмерској пракси су се користиле многе методе за анализу и пројектовања процедуралног или објектно-оријентисаног софтвера. О њима је много писано и на српском језику[189,190,191]. Литература о пројектовању софтвера у *DevOps*-у је скромнијег обима[192,193].

Анализа система и пројектовање софтвера у *DevOps*-у, највећим делом се реализују у фазама: планирања, кодирања и конструкције софтвера, али су повратним везама и итерацијама повезане и са осталим фазама *DevOps*-а. Повратне везе и итерације између фаза су саставни део методолошког поступка у континуалном развоју и постављању софтвера у продукционо окружење, што чини процес пројектовања софтвера компликованијим и захтевнијим, у односу на некадашње агилне и друге методе развоја софтвера. Свакој итерацији пројектовања софтвера и кодирања у *DevOps*-у претходи фаза прикупљања захтева и анализе пословног система.

13. ПРИКУПЉАЊЕ ЗАХТЕВА И АНАЛИЗА ПОСЛОВНОГ СИСТЕМА

Анализа пословања (*business analysis*) део је софтверског процеса који се односи на прикупљање пословних захтева. Одвија се кроз фазе припреме, спровођења, документовања и потврде резултата. Главни резултат је документ с пословним захтевима (*Business Requirements Document*, *BRD*) који садржи пословне захтеве усклађене с правилима и ограничењима пословања[194]. Анализу пословања врши пословни аналитичар (*business analyst*). У анализу пословања укључене су и друге стране:

- Клијент – организациони систем или појединац, интерни или екстерни, за кога пословни аналитичар дефинише, прикупља и документује захтеве.
- Стејкхолдер – свако ко на неки начин има везе с пројектом, ко може утицати на исход пројекта или на кога исход пројекта може утицати.
- Крајњи корисник – појединац или група који директно користе развијено решење.

[188] Budgen, D. (2003). *Software design*. Pearson Education.
[189] Перишић, Б. (2007). *Пројектовање софтвера* (електронско издање), ФТН, Нови Сад.
[190] Влајић, С. (2021). *Пројектовање софтвера* (скрипта), ФОН, Београд.
[191] Тартаља, И. (2011). *Пројектовање софтвера* (скрипта), ЕТФ, Београд.
[192] Bogdanović, Z., Despotović-Zrakić, M., Barać, D., Labus, A. & Radenković, M. (2023). The Role of DevOps in Sustainable Enterprise Development. In: *Sustainability: Cases and Studies in Using Operations Research and Management Science Methods* (pp. 217–237). Cham: Springer International Publishing.
[193] Radenković, M., Popović, S. & Mitrović, S. (2022, June). Project based learning for DevOps Project based learning for DevOps: School of Computing experiences. In: *E-business technologies conference proceedings* (Vol. 2, No. 1, pp. 127–131).
[194] Marshall, C. (2000). *Enterprise modeling with UML: designing successful software through business analysis*. Addison-Wesley Professional.

- Спонзор – појединац или група који доносе важне одлуке у вези с пројектом, а које се тичу циљева, буџета и других ресурса.
- Доменски експерт – особа која до најситнијих детаља познаје процесе који су предмет пословне анализе.

Прва фаза анализе пословања је прикупљање захтева, и она се реализује техникама као што су: брејнсторминг, анализа постојеће документације, анкете, интервјуи, фокус групе, посматрање, радионице итд. На основу резултата прве фазе, прелази се на анализу захтева (*requirements analysis*). Анализа захтева се реализује графичким моделовањем, коришћењем одговарајућих техника. Користе се многобројне технике моделовања, како би модели били разумљиви различитим групама стејкхолдера. Неке од техника моделовања укључују структурну систем-анализу, дијаграме тока, дијаграме случајева коришћења, моделе објекти-везе, моделе пословних процеса и бројне друге моделе. На основу анализе захтева, прелази се на њихову спецификацију (*requirement specification*), где се врши детаљнија категоризација, описивање, приоритетизација и валидација захтева. На крају, неопходно је да *Business Requirements Document* буде одобрен од стране одговарајућих стејкхолдера.

Најчешће коришћене технике за прикупљање и анализу захтева, као и за моделовање пословних процеса су различити типови дијаграма тока, *IDEF*, *UML* и *BPMN*[195].

13.1 Преглед традиционалних техника за анализу захтева

Током претходних деценија, развијено је више техника и приступа за прикупљање и анализу захтева као и моделовање пословних процеса[196]. Неке од њих описане су у тексту који следи.

Дијаграми тока (*flow charts*) користили су се за опис логике или тока извршења програма у оквиру једног процеса. Скуп симбола који се користе за графичко моделовање релативно је мали и више прилагођен програмерима него бизнис аналитичарима.

Дијаграми тока података (*data flow charts*) користили су се за моделовање корака обраде података у пословном систему, где се у сваком кораку реализује једна акција над подацима. Карактеристични су за приступ **структурне систем-анализе** у којој се спецификација обраде података врши коришћењем четири основна графичка елемента: процеси, складишта, токови и интерфејси[197]. Оригинална идеја је била да се при моделовању апстрахују детаљи о физичким локацијама података, базама

[195] Giaglis, G. M. (2001). A taxonomy of business process modeling and information systems modeling techniques. *International Journal of Flexible Manufacturing Systems*, 13(2), 209–228.
[196] Popović, S. (2007). *Modeliranje poslovnih procesa u razvoju aplikacija elektronskog poslovanja* (magistarska teza), Fakultet organizacionih nauka, Univerzitet u Beogradu.
[197] DeMarco, T. (1979) Structure analysis and system specification. In *Pioneers and Their Contributions to Software Engineering: SD&M Conference on Software Pioneers*, Bonn, June 28/29, 2001, Original Historic Contributions, pp. 255–288. Berlin, Heidelberg: Springer Berlin Heidelberg.

података, датотекама, времену извршења, платформама и извршиоцима, па нису погодни за детаљну спецификацију пословних процеса, нити спецификацију синхронизације и координације процеса.

Дијаграми тока контроле (*control flow diagrams*) представљају проширење дијаграма тока података. Погоднији су за спецификацију апликација заснованих на догађајима, где је мањи фокус на подацима. Примењују се у различитим инжењерским областима, а у контексту софтверског инжењерства погодни су за моделовања елемената *IoT* система.

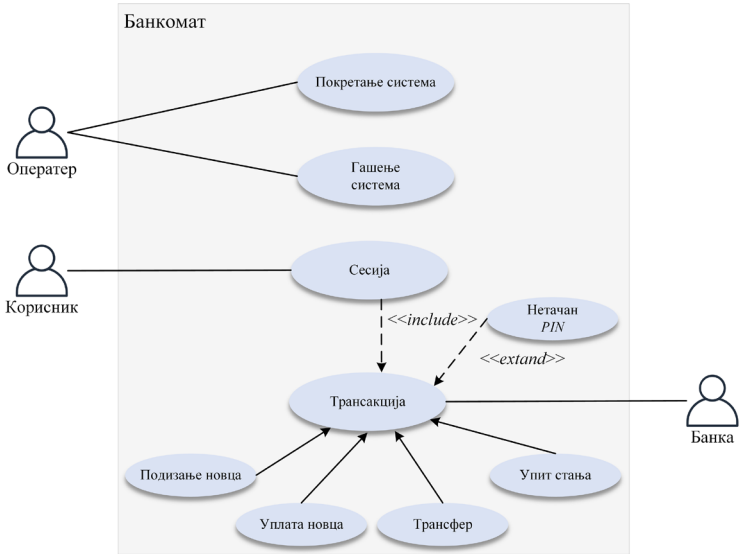
IDEF0 и касније ***IDEF5***[198] техника моделовања карактеристична је за период 1970–1990. Представља први графички модел који је могао истовремено да подржи послове које извршавају људи и које извршава систем. Главна предност је комплементарност са моделом објекти-везе. *IDEF0* модел активности користио се за моделовање пословних процеса унутар предузећа и спецификацију улаза, излаза и контроле задатака.

13.2 Анализа захтева помоћу UML-а

Развојем објектно-оријентисаног приступа, доминантан језик за спецификацију, визуелизацију, конструкцију и документовање производа софтверског система, као и за моделовање пословних процеса и других несофтверских система постаје *UML*[199]. Захтеви скалабилности, поузданости и дистрибуираности су у време настанка *UML*-а већ били идентификовани, али се решења траже у оквирима објектно-оријентисаног приступа.

Случајеви коришћења (*use cases*) представљају често коришћени начин за представљање захтева понашања које систем треба да испуни[200]. Њима се идентификују корисници и њихова интеракција са софтверским системом. У објектно-оријентисаном моделовању, случајеви коришћења представљају полазну основу за моделовање динамике система. Пример дијаграма случајева коришћења приказан је на Слици 15.

[198] https://www.idef.com/idefo-function_modeling_method/
[199] Eriksson, H. E. & Penker, M. (2000). *Business modeling with UML*. New York, 12.
[200] Rosenberg, D. & Scott, K. (1999). *Use case driven object modeling with UML* (pp. 1–4). Reading: Addison-Wesley Professional.

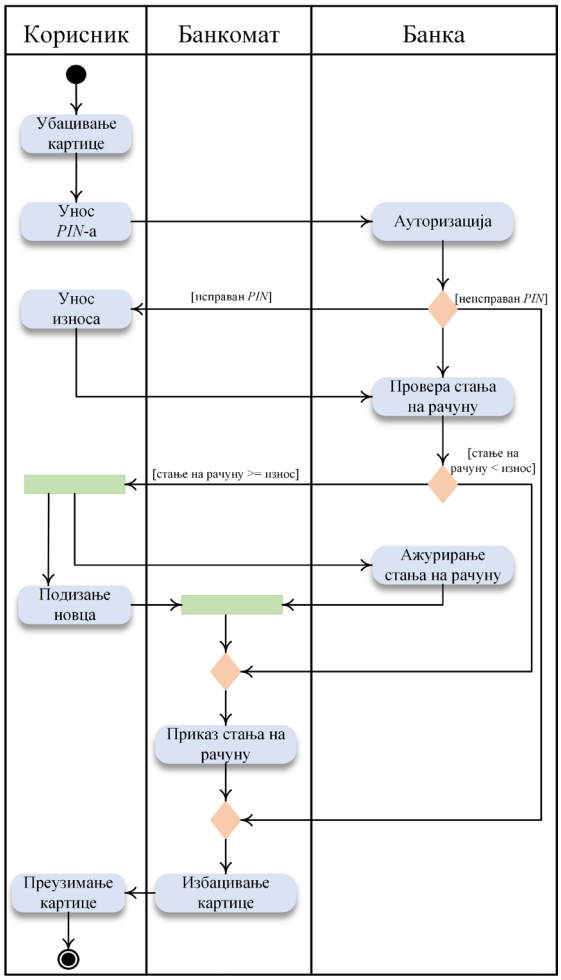


Слика 15. Пример дијаграма случајева коришћења

Динамика се у објектно-оријентисаном приступу моделује помоћу дијаграма понашања: дијаграма секвенци и дијаграма активности. Дијаграми активности приказују учеснике у пословном процесу, стања у којима се ентитети у процесу налазе, гранања и спајања. Коришћењем дијаграма активности могу се моделовати различити патерни паралелизације и синхронизације између процеса[201]. Пример дијаграма активности приказан је на Слици 16.

Главни и неочекиван недостатак *UML*-а је то што не садржи одговарајуће дијаграме за представљање нити, које су главно средство за извршење програма и паралелизацију у објектно-оријентисаном програмирању. Такође, *UML* не дефинише директна мапирања у спецификације процесних језика, нити се у дијаграмима активности може спецификовати потпуна семантика пословних процеса. Након моделовања применом *UML*-а, неопходно је даље пројектовати доменски модел, моделирати архитектуру система, апликационе програмске интерфејсе, и на крају имплементирати софтвер.

[201] Dumas, M. & Ter Hofstede, A. H. (2001). UML activity diagrams as a workflow specification language. In: *International conference on the unified modeling language* (pp. 76–90). Berlin, Heidelberg: Springer Berlin Heidelberg.



Слика 16. Пример UML дијаграма активности

Имајући у виду недостатке *UML*-а и његову неприлагођеност, било је потребно изабрати други језик за моделовање пословних процеса, па се трага за бољим приступима моделовању, који треба да омогуће спецификацију процеса која се аутоматски може превести у архитектуру софтвера и имплементацију пословне логике. За ове потребе развијен је *BPMN* (*Business Process Model and Notation*)[202].

13.3 Моделовање пословних процеса и BPMN

Управљања пословним процесима је област менаџмента која се односи на унапређење организационих перформанси кроз анализу, надгледање и унапређење ефикасности и ефективности пословних процеса. Развој ове области везује се

[202] https://www.bpmn.org/

за ране фазе развоја менаџмента, па се у текстовима Адама Смита из 18. века говори о подели посла као методи за паралелизацију, којом се процес производње може убрзати[203]. Прва документација и анализа пословних процеса везује се за *Time and Motion* студије Фредерика Тејлора и Френка Гилбрета из 1911, у циљу смањења потребног времена за извршење посла, смањење броја потребних акција у производњи или повећање ефективности радника. Френк Гилбрет је у свом раду *Process charts – First Steps to Finding the One Best Way* из 1921. године представио први начин за графичко приказивање пословних процеса[204]. Концепт *workflow*-а јавља се седамдесетих година двадесетог века, као начин моделовања послова базираних на документима, унутар једног дела предузећа, које обављају само људи. Од 1980. почиње тзв. ера квалитета (*Total Quality management*), где се фокус ставља на континуално унапређење квалитета производа[205]. Најбољи пример овог приступа је 6-*sigma* правило[206].

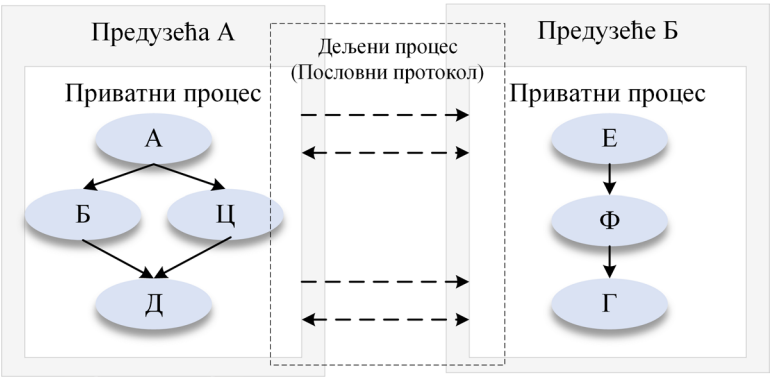
Значајне промене настају деведесетих година, у интернет ери, када доминантан приступ постаје реинжењеринг пословних процеса и дигитална трансформација, где је циљ драстична промена у организацији предузећа и извршавању процеса, у циљу унапређења производње и ефикасности[207]. Промене у пословним процесима се заснивају на дигитализацији, унапређењу *IT* инфраструктуре и електронској размени података. С обзиром на то да у овом периоду доминира објектно-оријентисан приступ програмирању, *UML* се користи за графичко моделовање и пресликавање пословних процеса у функције информационог система[208]. Од 2000. настаје тзв. *big data* ера, коју карактерише презасићење подацима и усмеравање софтвера ка захтевима линеарне скалабилности, дистрибуираности и поузданости рада. Класично објектно програмирање и објектно-оријентисана парадигма развоја софтвера не подржавају ниједан од ових захтева, па се фокус помера ка процесно-оријентисаном приступу и моделовању пословних процеса.

BPM (*business process modelling*) је приступ управљању променама у пословању предузећа, који подржава прелаз с функционалне на процесну организацију. Циљ је унапређење пословних процеса, при чему су промене свеобухватне и обухватају цео животни циклус процеса, полазећи од анализе и пројектовања, па до имплементације, аутоматизације и извршавања процеса. *BPM* посматра организацију као скуп процеса који се могу дефинисати, којима се може управљати

[203] Buzacott, J. A. (1996). Commonalities in reengineered business processes: models and issues. *Management Science*, 42(5), 768–782.
[204] https://www.thegilbreths.com/resources/Gilbreth-Process-Charts-1921.pdf
[205] Becker, J., Rosemann, M. & Von Uthmann, C. (2002). Guidelines of business process modeling. In: *Business Process Management: Models, Techniques, and Empirical Studies* (pp. 30–49). Berlin, Heidelberg: Springer Berlin Heidelberg.
[206] Montgomery, D. C. & Woodall, W. H. (2008). An overview of six sigma. *International Statistical Review/Revue Internationale de Statistique*, 329–346.
[207] Davenport, T. H. & Short, J. E. (1990). The new industrial engineering: information technology and business process redesign. *MIT Sloan Management Review*. MIT.
[208] Booch, G., Rumbaugh, J. & Jacobsen, I. (1997). *UML: unified modeling language*. Versão.

и који се могу оптимизовати[209]. Моделовање пословних процеса се може сматрати предусловом за дигиталну трансформацију, постизање интеграције, а такође може помоћи при решавању проблема интероперабилности[210]. Фазе *BPM*-а су: идентификација, моделовање, анализа, побољшање, имплементација, извршавање, надгледање и контрола процеса.

Према намени унутар организације, пословни процеси се могу поделити на: стратешке, оперативне и процесе подршке. Зависно од дистрибуираности, пословни процеси се деле на: приватне пословне процесе који се налазе унутар граница једне организације и пословне протоколе, тј. дељене пословне процесе, који се користе за дефинисање пословног процеса дистрибуираног између више организација (Слика 17).



Слика 17. Приватни и дељени пословни процеси[211]

BPMN нотација је данас најкоришћеније средство за моделовање пословних процеса. Развијана је дуго година и преузела је многе добре особине других метода и техника које су се користиле у ту сврху. Прва званична верзија представљена је у мају 2004. године. Последња доступна верзија је 2.02 из 2014. *BPMN* дефинише графичко окружење и језик за опис пословних процеса. Примарни циљ је успостављање стандардне нотације која је разумљива за све учеснике, укључујући: кориснике, менаџере, аналитичаре, пројектанте, програмере и друге. *BPMN* подржава концепте моделовања који се односе на пословне процесе, укључујући податке и метаподатке. Састоји се из две нотације: графичке нотације и извршне спецификације засноване на *XML*-у. У првим верзијама извршна спецификација

[209] Recker, J., Rosemann, M., Indulska, M. & Green, P. (2009). Business process modeling – a comparative analysis. *Journal of the association for information systems*, 10(4), 1.
[210] Dumas, M., Van der Aalst, W. M. & Ter Hofstede, A. H. (2005). *Process-aware information systems: bridging people and software through process technology*. John Wiley & Sons.
[211] Прилагођено према: Radenković, B., Despotović-Zrakić, M., Bogdanović, Z., Barać, D., Labus, A. (2015). *Elektronsko poslovanje*, Fakultet organizacionih nauka, ISBN:978-86-7680-304-0.

је била ограничена на *BPEL*[212], а од верзије 2.0 се може применити и у другим програмским језицима и окружењима. *BPMN* је стандардизован као *ISO/IEC* 19510, еквивалентно *BPMN* 2.0.1. верзији коју је развио *OMG*[213]. Главни елементи *BPMN*-а су:

- *Pools* и *lanes*, користе се за означавање процеса и учесника.
- Елементи тока, укључују активности, догађаје, тачке одлучивања.
- Везе, укључују секвенце догађаја и активности, различите типове веза између других ентитета у моделу, као и поруке које размењују.
- Подаци, спецификација података и информација потребних за обављање појединих активности.

У Табели 1 су приказани типични *BPMN* елементи.

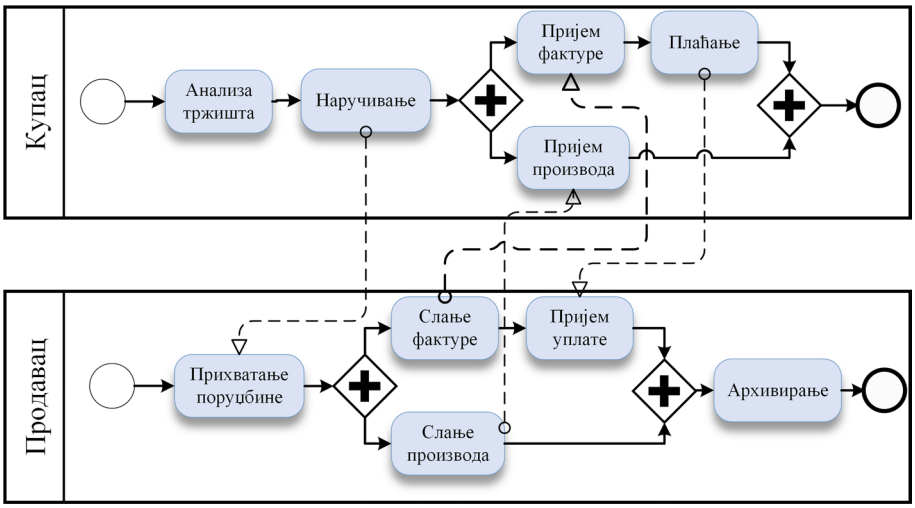
Табела 1. Типични BPMN елементи

Pool		*Pool* елемент представља учесника у процесу. Може бити пословни ентитет, улога, или сл.
Lane		*Lane* елемент омогућује поделу унутар *pool* елемента у циљу категоризације.
Активност		*Task* је елемент за означавање активности. Постоје различити типови задатака: позив сервиса, слање поруке, пријем поруке, активност коју обавља корисник, активност која се ради ручно, пословно правило, извршење скрипта.
Догађаји		Спецификација догађаја у току извршења пословног процеса. Могу се односити на почетак, трајање или завршетак процеса. *BPMN*2.0[214] садржи већи број симбола за догађаје који се односе на поруке, услове, грешке, линкове итд.
Тачке одлучивања		*Gateways* представљају гранања у процесу. Могу означавати ексклузивна гранања, инклузивна гранања, паралелизацију, гранања на основу типа догађаја.
Ток		Ток представљен пуном линијом и стрелицом приказује редослед извршавања активности у процесу.

[212] Ouyang, C., Dumas, M., Ter Hofstede, A. H. & Van der Aalst, W. M. (2006, September). From BPMN process models to BPEL web services. In: *2006 IEEE International Conference on Web Services* (ICWS'06) (pp. 285–292). IEEE.
[213] https://www.iso.org/standard/62652.html
[214] https://www.omg.org/spec/BPMN/2.0/PDF

Порука		Испрекидана линија с непопуњеном стрелицом приказује ток порука у процесу.
Веза		Тачкаста линија са отвореном стрелицом означава везу података (артефаката) са објектима тока.
Податак	Name (state)	Елемент за приказ података који су улаз или излаз из активности.
Анотација	Служе за упис додатних информација	Обезбеђује додатне текстуалне информације на дијаграму.

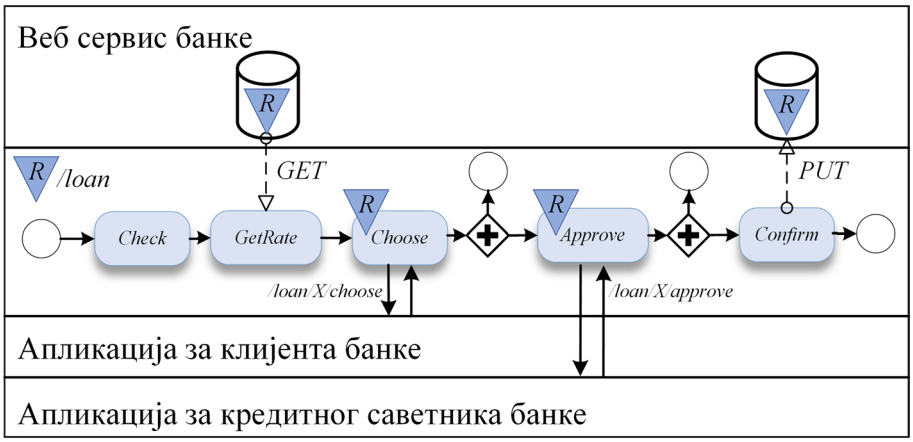
Пример *BPMN* дијаграма с високим нивоом апстракције за пословни процес сарадње између два предузећа приказан је на Слици 18.



Слика 18. Пример колаборативног пословног процеса у BPMN[215]

Поред тога што је *BPMN* нотација развијена за моделовање комплексних пословних процеса и њихове интеракције, увођењем кореографије у верзију 2.0, и развојем микросервисиних архитектура отворила су се нове могућности за имплементацију софтвера за подршку пословним процесима. Једна од могућности је примена *BPMN*-а за моделовање кореографије и оркестрације *RESTful* веб сервиса и микросервиса[216]. Пример је приказан на Слици 19.

[215] Прилагођено према: Weske, M. (2007). *Business process management architectures* (pp. 305–343). Springer Berlin Heidelberg.
[216] Pautasso, C. (2011, November). BPMN for Rest. In: *International Workshop on Business Process Modeling Notation* (pp. 74–87). Berlin, Heidelberg: Springer Berlin Heidelberg.



Слика 19. Пример моделовања пословног процеса са спецификацијом REST позива

13.4 Извршна спецификација пословних процеса

Почевши од верзије 2.0, *BPMN* подржава извршну спецификацију за пословне процесе, која омогућава аутоматизацију генерисања рачунарског кôда, која није ограничена само на *BPEL*, већ се може реализовати и с другим интерпретерима[217], другим окружењима[218] и програмским језицима, нпр. *Java*[219].

Позиција извршне спецификације у структури *BPMN* приказана је на Слици 20.

Извршна спецификација је серијализована у *XML* формату. У оквиру извршне спецификације дефинисане су две *XML* шеме: *Diagram interchange*, која садржи све релевантне информације за трансфер модела из *BPMN* у неки други алат или програм, и *Execution semantics*, која описује техничке детаље процеса[220].

[217] https://www.activiti.org/
[218] Mutarraf, U., Barkaoui, K., Li, Z., Wu, N. & Qu, T. (2018). Transformation of Business Process Model and Notation models onto Petri nets and their analysis. *Advances in Mechanical Engineering*, 10(12), 1687814018808170.
[219] https://camunda.com/
[220] Börger, E. & Sörensen, O. (2011). BPMN core modeling concepts: Inheritance-based execution semantics. In: *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges* (pp. 287–332). Berlin, Heidelberg: Springer Berlin Heidelberg.

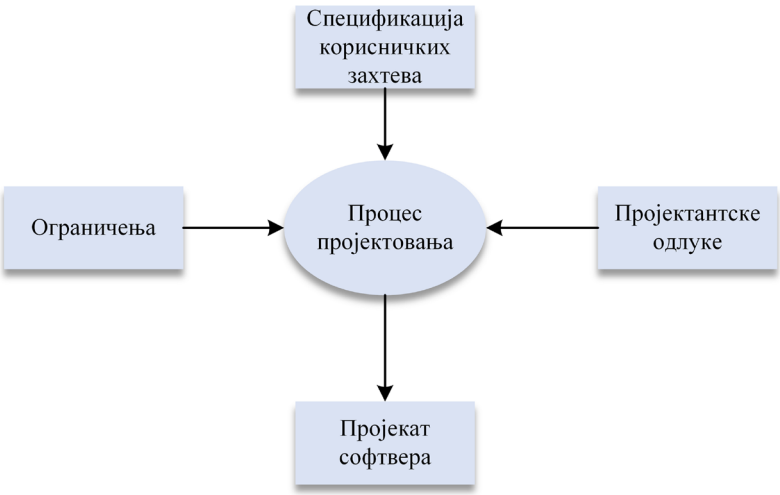
Слика 20. Извршна спецификација пословних процеса у BPMN[221]

На основу извршне спецификације, користећи архитектурне патерне и *API*-је платформи које су оријентисане на процесе, могуће је генерисати рачунарски кôд за извршење пословног процеса. Извршна спецификација је посебно погодна за *low code* платформе[222] које омогућавају флексибилан избор програмског језика и развојног окружења (*Java*, *Python*, *.NET*), као и већу ефикасност и прецизност у извршењу пословних процеса[223].

14. ПРОЈЕКТОВАЊЕ СОФТВЕРА

Пројектовање софтвера (енгл. *Software Design*) прва је фаза у животном циклусу развоја софтвера у којој се с домена проблема прелази у домен решења[224]. У овој фази се преводе кориснички захтеви у архитектуру софтвера, идентификују се компоненте и модули, интерфејси преко којих они комуницирају, начини интеграције, и спецификују детаљи потребни за имплементацију[225]. Општи модел пројектовања софтвера приказан је на Слици 21.

[221] Прилагођено према: https://www.omg.org/oceb-2/documents/real-life-bpmn-CamundaOCEB-2-DownloadSection.pdf
[222] https://camunda.com/
[223] Funk, D. (2022). Creating a Low-Code Business Process Execution Platform With Python, BPMN, and DMN. *IEEE Software*, 40(1), 9–17.
[224] Budgen, D. (2003). *Software design*. Pearson Education.
[225] ISO/IEC/IEEE 24765:2010 Systems and Software Engineering—Vocabulary, ISO/ IEC/IEEE, 2010



Слика 21. Општи модел пројектовања софтвера[226]

Одлике добро пројектованог софтверског система су: испуњава функционалне и нефункционалне захтеве, целовит је, укључује све потребне компоненте, податке и интерфејсе, поуздан је, може се лако одржавати и мењати према захтевима клијената, заснован је на ограничењима која се тичу ресурса, времена, и трошкова развоја и експлоатације. С друге стране, одлике лоше пројектованог софтвера су недостатак модуларности, јака спрега између модула, дуплицирање кôда, испреплетане зависности и изостанак могућности за рефакторисања кôда, што доводи до тешкоћа у одржавању и нагомилавање тзв. техничког дуга (*technical debt*), тј. трошкова који ће настати у будућности, а који су последица лоших одлука приликом пројектовања[227].

Један од приступа који се користи приликом пројектовања софтвера познат је под енглеским називом *design thinking*[228]. Овај приступ промовише итеративан и нелинеаран процес пројектовања који се заснива на сарадњи пројектанта софтвера и пословних корисника. Кључни кораци обухватају јасно дефинисање и добро разумевање захтева, развој концептуалног решења, анализирање могућности за имплементацију концепта, брз развој прототипа и итеративно унапређење прототипа на основу повратних информација.

[226] Прилагођено према: Budgen, D. (2003). *Software design*. Pearson Education
[227] Kruchten, P., Nord, R. L. & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice, *IEEE Software*, 29(6), 18–21.
[228] Brown, T. (2008). Design thinking, *Harvard business review*, 86(6), 84. https://readings.design/PDF/Tim%20Brown,%20Design%20Thinking.pdf

14.1 Принципи пројектовања софтвера

Приликом пројектовања софтвера, тежи се ка следећим циљевима:

- Тачност (енгл. *correctness*). Добро пројектовање софтвера резултује здравим и стабилним системом који испуњава функционалне и нефункционалне захтеве, и на одговарајући начин интегрише компоненте и модуле.
- Целовитост. Добро пројектован софтвер садржи све потребне компоненте и модуле, податке, интерфејсе и др. Све зависности су усклађене.
- Оптимално коришћење ресурса. Системски ресурси се користе рационално и ефикасно.
- Скалабилност. Дугорочна скалабилност се често постиже применом модуларизацације и организације модула по слојевима.
- Једноставност одржавања. Софтвер треба пројектовати тако да се измене могу брзо и лако имплементирати.

За постизање наведених циљева, потребно је примењивати принципе пројектовања софтвера. Основни принципи и концепти који се примењују у пројектовању софтвера су[229,230]:

- **Апстракција.** Приликом пројектовања софтвера неопходно је идентификовати суштину проблема, а занемарити мање битне детаље. Апстракција омогућује пројектовање на концептуалном нивоу, без улажења у детаље имплементације, чиме се убрзава процес развоја и унапређује квалитет развијеног решења.
- **Раздвајање надлежности.** Декомпозиција система на мање делове који се могу лакше разумети и анализирати и у којима су надлежности јасно раздвојене на основу карактеристика проблема који се решава, омогућује да се квалитетно дефинише модуларна архитектура софтвера. Овај принцип је кључан за пројектовање софтвера код којег су кључни захтеви везани за атрибуте квалитета.
- **Декомпозиција и модуларизација.** Организација система у конкретне модуле са јасно дефинисаним интерфејсима представља конкретизацију раздвајања надлежности. Модули се могу независно развијати, и користити у различитим контекстима. Логички повезани модули могу се организовати у компоненте. Модули се могу паралелно извршавати, а тиме се омогућава постизање веће скалабилности. Излаз из модуларизације је архитектура модула.
- **Интеграција.** Модуларизација је једна од фаза у (*Top-Down*) пројектовању софтвера. Пројектоване модуле је потребно интеграцијом (*Bottom-Up*) организовати у апликације. Модуларизација и интеграција се у пројектовању софтвера агилним методама примењују у итеративно инкрементaлним корацима. Излаз из фазе интеграције је архитектура апликације.
- **Сакривање података – енкапсулација.** Подаци унутар модула су сакривени, и не може им се приступити из других модула. Размена података између модула мора бити експлицитно пројектована.

[229] https://www.computer.org/resources/importance-of-software-design-is-important
[230] https://www.scaler.com/topics/software-engineering/what-is-software-design/

- **Побољшавање (унапређење, усавршавање) пројекта** (енгл. *Refinement*). Провером пројекта софтвера потребно је идентификовати и отклонити све структурне недоследности. Ово се постиже декомпозицијом и анализом на нижим нивоима апстракције, у итеративно-инкрементaлном процесу провере и евалуације предложеног решења.
- **Интерфејс.** Интерфејс је дељена граница између модула или компоненти, преко које се одвија комуникација. Једини начин на који модули и компоненте могу размењивати податке је преко интерфејса.
- **Архитектура софтвера.** Архитектура софтвера односи се на низ пројектантских одлука у вези са општом структуром и понашањем софтверског система. Архитектура је апстракција која омогућује стејкхолдерима да разумеју и анализирају како ће систем реализовати кључне нефункционалне захтеве, као што су доступност, поузданост, сигурност и сл. Приликом пројектовања архитектуре, фокус је на идентификацији модула и компоненти, и њихових интеракција.
- **Патерни.** Патерни у пројектовању софтвера се користе као метамодели, шаблони, добре праксе или општа решења за проблеме који се често јављају. Користе се патерни архитектуре софтвера, патерни пројектовања, антипатерни и др.
- **Рефакторисање.** Рефакторисање кôда је процес континуланог унапређења и реорганизације кôда без мењања оригиналне функционалности. Рефакторисањем се побољшава кôд, у циљу унапређења архитектуре, структуре и имплементације система.

У контексту агилног објектно-оријентисаног развоја често се помињу *SOLID* принципи, чији је задатак да унапреде квалитет софтвера. Међутим, ови принципи се могу аналогно применити и на процесно-оријентисани развој софтвера, где се принципи уместо на објекте примењују на сервисе или микросервисе[231]:

- **Принцип јединствене одговорности** (енгл. *single responsibility principle*). Класа или сервис имају само један посао, и један разлог за промену. Уколико класа или сервис имају више од једне одговорности, добра праксе је разбити их на више мањих.
- **Принцип *open/closed*.** Након што су класа или сервис креирани и тестирани, уколико дође до промене корисничких захтева, не треба дозволити модификације, већ само проширења. Овај принцип је уграђен у више архитектурних и дизајн патерна.
- ***Liskov Substitution* принцип.** Објекти суперкласе могу се заменити објектима подкласе, без утицаја на рад програма. Верификација овог принципа је уграђена у компајлере појединих програмских језика (нпр. *Java*), док се код сервисних и микросервисних архитектура мора проверити кроз тестирање и верификацију *API* шеме.
- **Принцип раздвајања интерфејса.** Свака класа треба да има посебан интерфејс, а зависности између класа треба базирати на најмањем могућем интерфејсу. У контексту микросервисне архитектуре, ово значи да не треба стављати велики

[231] https://david-vancouvering.medium.com/applying-solid-principles-to-services-e56ef2382a26

број функционалности у један *API*, већ пројектовати већи број *API*-ја са мањим бројем функционалности.

- **Принцип инверзије зависности.** Модули вишег нивоа не треба да буду зависни од модула нижег нивоа, већ оба треба да зависе од апстракција (апстрактних класа и интерфејса). У софтверу оријентисаном на процесе овај принцип се имплементира преко апстрактних *API*-ја. Апстрактни *API*, и окружење које од њега зависи, дефинише произвођач софтвера. Имплементација апстракних *API*-ја препушта се кориснику, како би могао да прилагоди сервис у складу с његовим специфичним потребама.

14.2 Апстракција и модели у пројектовању софтвера

Апстракција је упрошћавање и поједностављивање погледа на реални свет, који је валидан и употребљив за одређене услове, ситуације и ограничења. Апстракција олакшава процес пројектовања софтвера и смањује време потребно за његову имплементацију[232].

Апстракција је основно средство за изградњу модела софтвера. Процес изградње модела назива се моделовање. Резултат моделовања је модел, који представља апстрактну представу реалног система. Модел је валидан за одређене услове и ограничења. Услови и ограничења су саставни део модела и дефинишу се корисничким захтевима, границом пословног система, парадигмом развоја софтвера, изабраном технолошком платформом и реалним условима пословања.

Пројектовање софтвера засновано је на апстракцији и декомпозицији. Резултат пројектовања софтвера, применом више итерација апстракције и декомпозиције, јесте модел софтверског система који је валидан за доменска ограничења, корисничке захтеве и услове експлоатације софтвера.

У процесу моделовања софтверског система, апстракцијом се одбацују непотребни детаљи, а декомпоновањем се прецизније дефинише његова преостала структура.

Постоји више врста апстракције. У процесу пројектовања софтвера највише се користе следеће апстракције:

- **Класификација**, представља поступак у коме се концепти моделовања групишу према заједничким особинама. Груписани концепти представљају класу концепата. Инстанцирање је поступак супротан класификацији, и представља издвајање једног концепта из класе.
- **Генерализација** је поступак идентификације заједничких особина групе концепата и генерисање концепта вишег хијерархијског нивоа који поседује идентификоване заједничке особине. Генерализација представља основни метод за спецификацију хијерархије при изградњи модела. Супротан поступак

[232] Jacobsen, E. E., Kristensen, B. B. & Nowack, P. (1998, September). Models, Domains and Abstraction in Software Development. In: *Proceedings Technology of Object-Oriented Languages*. TOOLS 27 (Cat. No. 98EX224) (pp. 28–42). IEEE.

је специјализација, генерисање концепта нижег хијерархијског нивоа, који садржи све особине концепта вишег хијерархијског нивоа и поред њих неке своје специфичне особине.

- **Асоцијација** је апстракција у којој концепт вишег хијерархијског нивоа успоставља везу с концептом нижег хијерархијског нивоа.
- **Агрегација** је апстракција у којој се више концепата са свим својим особинама групишу у нови агрегирани концепт. Агрегирани концепт и концепти који чине агрегацију егзистирају независно.
- **Редукција** је апстракција при којој се одбацују детаљи који су небитни за ограничења и корисничке захтеве.

Софтверски инжењери су прилагодили терминологију и семантику апстракција, које користе у свакодневном раду, жаргону који је уобичајен за пројектовања софтвера. Најчешће коришћени термини за поступке апстракције у пројектовању софтвера и њихову семантику су:

1. **Енкапсулација**, учаурење, апстракција редукције и асоцијација 1:1, представља ограничење видљивости неког дела програма или структуре података на само најнеопходније делове који су потребни за интеракцију са окружењем. Омогућава једноструко наслеђивање, без могућности модификације структуре енкапсуларних података и програма. Типичан пример за енкапсулацију у развоју процесно оријентисаног софтвера је сервис или микросервис који енкапсулира податке и догађаје. Енкапсулацијом се може обухватити и модуларизација, издвајање делова софтвера у модуле са интерфејсима преко којих модули комуницирају са окружењем. Контејнер је у модерним архитектурама преферирано средство за енкапсулацију микросервиса.
2. **Композиција**, облик апстракције у коме се концепти моделовања агрегирају на основу критеријума, који не мора бити повезан с њиховим особинама. При композицији се дефинише домен, тј. граница композиције у односу на окружење. Композиција обухвата агрегацију и додатну детаљизацију, односно уношење више детаља у апстракцију. Она омогућава да се унутар једног концепта дефинише други концепт (нпр. догађај, сервис или подпроцес се могу дефинисати у оквиру процеса). Композиција је поступак који се примењује при интеграцији микросервиса и модула у апликацију. Типична примена композиције у развоју софтвера оријентисаног на процесе је кореографија или оркестрација сервиса, у којима се рачунарска апликација, која представља скуп процеса, склапа из скупа догађаја и сервиса. Композиција се користи и у конструкцији софтвера, при склапању модула у рачунарски програм.
3. **Декомпозиција**, апстракција супротна од композиције у којој се целина раздваја у више делова. Представља један од најважнијих корака у пројектовању софтвера. У пројектовању софтвера, након процеса апстракције када се добије језгро модела у коме су одбачене све непотребне особине, језгро модела се даље профињује тако што се декомпозицијом разложи у више компоненти, а свака од компоненти се у низу даљих итерација апстракције и декомпозиције усклађује с пројектним захтевима и другим ограничењима.

4. **Модуларност**, облик декомпозиције у коме се програм дели у више мањих делова, модула, који могу аутономно функционисати. При декомпозицији софтвера дефинишу се интерфејси и начин комуникације између модула. Модули се користе при дистрибуирању и паралелизацији рада софтвера. При дистрибуирању, критеријум за дефинисање модула, поред аутономности рада, представља и географска локација послова и података у оквиру модула. Процесно-оријентисан софтвер има веће могућности за модуларност од објектно-оријентисаног софтвера. У објектно-оријентисаном софтверу, најмања јединица модуларности је једна нит с једним објектом, његовим структурама података и методама. У процесно-оријентисаном софтверу, најмања јединица модуларности је контејнер који садржи један микросервис.
5. **Интеграција** је облик апстракције композиције и агрегације, у којој се модули агрегирају у апликацију. Развој софтвера се састоји од низа итеративно-инкрементaлних корака модуларизације, у којој се врши декомпозиција система у складу с функционалним и нефункционалним захтевима, и интеграције, у којој се модули поново интегришу у апликације. У софтверу оријентисаном на процесе, основни градивни елемент је догађај. Сви остали концепти, сервиси, процеси и апликације, информациони системи и *B2B* мреже су агрегације догађаја, па се за њихову спецификацију користи апстракција интеграције, која се имплементира архитектурама обраде догађаја, у форми софтверских оквира, платформи и екосистема. На тај начин се удео кодирања у развоју софтвера оријентисаног на процесе, у односу на објектно-оријентисани софтвер, значајно смањује.
6. **Наслеђивање**, врста специјализације у којој компонента на нижем нивоу хијерархије наслеђује особине компоненте на вишем нивоу хијерархије. Поред особина компоненте вишег нивоа, компонента нижег нивоа хијерархије може имати и своје додатне особине. У објектно-оријентисаном софтверу, наслеђивање је ограничено на објекте. У процесно-оријентисаном софтверу, наслеђивање је најпре оријентисано на патерне архитектуре софтвера. Применом патерна архитектуре, имплементирана архитектура има све особине патерна, али и своје посебне особине које су карактеристичне за област примене и конкретан случај коришћења. У имплементацији се наслеђују особине процеса и ограничења ресурса (*cgroups*, *namespaces*) код контејнера.
7. **Полиморфизам**, специфична врста специјализације и декомпозиције, облик наслеђивања у коме компонента на нижем нивоу хијерархије може у реалном времену мењати наслеђене особине[233]. У објектно-оријентисаном приступу, полиморфизам је ограничен на објекте. Полиморфизам се у сервисно-оријентисаној архитектури може реализовати коришћењем *REST* архитектурног стила, увођењем динамичке промене садржаја документа на који показује идентификатор ресурса. У софтверу оријентисаном на процесе, као и код наслеђивања, полиморфизам је усмерен на архитектурне патерне. Типичан пример за полиморфизам је патерн посредника (*proxy*, инверзни *proxy*, *API proxy*), који у реалном времену може мењати сервисе опслуживања, као и њихове дефиниције,

[233] Cardelli, L. & Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys* (CSUR), 17(4), 471–523. https://inria.hal.science/inria-00498407/file/Duca09c-TSE-SOAArchitectureExtraction.pdf

нпр. замена извора догађаја (*event source*) или прихватача догађаја (*event handler*) у реалном времену у *Pub/Sub* патерну интеракције процеса. Код контејнера, *name-space* нижег хијерархијског нивоа може мењати наслеђене лимите и изолацију ресурса на свом нивоу хијерархије. Измене се даље пропагирају кроз ниже нивое хијерархије.

8. **Инстанцирање**, апстракција супротна класификацији, врста асоцијације у којој се компонента модела вишег нивоа назива концепт моделовања (у објектном приступу класа), а компонента нижег нивоа његова инстанца. Један концепт моделовања може имати више инстанци. Концепт моделовања садржи опис или дефиницију, док инстанца представља извршну имплементацију те дефиниције. Догађаји, сервиси, процеси и апликације имају своје дефиниције које се налазе у бази знања, а у време извршења могу имати више инстанци које се паралелно извршавају. У развоју софтвера оријентисаног на процесе, уобичајено је да се исти термин користи за дефинисање концепта и његове инстанце. На пример, под терминима: догађај, сервис, процес и апликација, у пројектовању софтвера се подразумева да се одређени термин односи на концепт моделовања и његову дефиницију, док се у време извршења програма под тим термином подразумева његова извршна инстанца. У литератури се могу наћи и примери да се концепт назива његовим именом, нпр. „Процес *XY*“, док се његова инстанца назива „Инстанца процеса *XY*“. Инстанца се при извршењу генерише у реалном времену, аутоматски, на основу дефиниције. Свака инстанца процеса је изолована, има своје приватне структуре података и извршава се у посебној програмској нити. Инстанце процеса се извршавају паралелно. С обзиром на паралелно извршавање инстанци, треба посветити посебну пажњу проблемима конкурентног програмирања: приступу заједничком ресурсу, критичним регионима и сигнализацији. Проблеми конкурентног програмирања најчешће се решавају, или у највећој мери избегавају, применом архитектурних стилова и патерна који те проблеме заобилазе, нпр. у развоју софтвера оријентисаног на процесе користи се микросервисна архитектура, која не дозвољава дељење података који се налазе унутар микросервиса, а интеракција и размена података с другим микросервисима се реализује применом *Pub/Sub* патерна. Контејнери се такође инстанцирају. На основу слике контејнера (*image*), креира се његова извршна инстанца.

Поред наведених, у софтверском инжењерству користе се и многи други термини који представљају синониме уобичајених назива за апстракције, са семантиком која је прилагођена њиховој примени у софтверском инжењерству[234].

Треба напоменути да је апстракција у објектно-оријентисаном програмирању усмерена на програмске језике, структуре података и објекте, док је апстракција у пројектовању софтвера оријентисаног на процесе усмерена на концепте моделовања, архитектуру софтвера, архитектурне стилове, патерне[235], микросервисе и контејнере.

[234] Jackson, D. (2012). *Software Abstractions: logic, language, and analysis*. MIT press.
[235] Zdun, U., Hentrich, C. & Dustdar, S. (2007). Modeling process-driven and service-oriented architectures using patterns and pattern primitives. *ACM Transactions on the Web* (TWEB), 1(3), 14-es.

14.3 Процес пројектовања софтвера

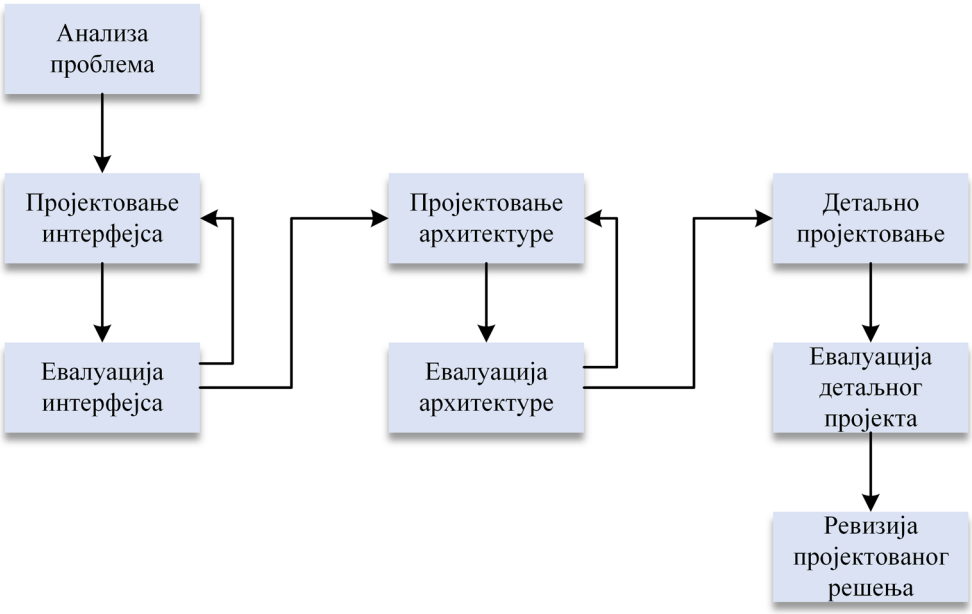
Процес пројектовања софтвера обухвата низ активности којима се преводе захтеви корисника у форму на основу које се софтвер може имплементирати. Током процеса пројектовања врши се спецификација архитектуре, модула, компоненти, интерфејса и података, на различитим нивоима апстракције.

Процес пројектовања софтвера реализује се у три подпроцеса[236]:

- **Пројектовање интерфејса.** Пројектовање интерфејса односи се на спецификацију интеракције система са окружењем. На овом нивоу се занемарује интерна структура софтвера, а систем се третира као црна кутија. Фокус приликом пројектовања је на комуникацији система са корисницима и другим апликацијама или уређајима. Пројекат интерфејса треба да садржи следеће детаље: прецизан опис догађаја или порука из окружења на које систем треба да одговори, прецизан опис догађаја и порука које систем треба да креира, спецификацију података и формата података који улазе или излазе из система, спецификацију динамике интеракција.
- **Пројектовање архитектуре софтвера.** Архитектура софтвера може се дефинисати као општа структура система која обезбеђује интегритет система на концептуалном нивоу. Архитектуром се идентификују главне компоненте и модули система, њихове надлежности, својства, интерфејси и интеракције, као и начин интеграције у целовиту апликацију. Пројектовање архитектуре се реализује на високом нивоу апстракције, па су интерни детаљи о компонентама и модулима сакривени. У итеративно-инкрементaлном поступку пројектовања архитектуре софтвера користе се два комплементарна приступа:
  - ***Top-down.*** Током пројектовања архитектуре софтвера неопходно је декомпоновати систем на главне компоненте, тј. модуле, имајући у виду нефукнционалне захтеве, тј. атрибуте квалитета, затим алоцирати функционалне захтеве по компонентама и пројектовати интерфејсе и комуникацију компоненти.
  - ***Bottom-up.*** Током пројектовања архитектуре софтвера неопходно је обезбедити механизме за ефективну интеграцију појединачних компоненти у функционалну целину.
- **Детаљно пројектовање.** На основу архитектуре, врши се пројектовање софтвера на нижем нивоу апстракције. Резултат је детаљна спецификација за сваки од модула, која обухвата и логичку структуру унутар модула, и физичку структуру на основу које се врши имплементација. Ово обухвата моделовање података, структуре података, алгоритме, управљање стањима, и друге имплементационе детаље.

Процес пројектовања софтвера приказан је на Слици 22.

[236] https://www.scaler.com/topics/software-engineering/what-is-software-design/



Слика 22. Процес пројектовања софтвера[237]

Наведени подпроцеси, у пројектовању софтвера оријентисаног на процесе, садрже специфичне активности, које се међусобно преплићу, и то:

- Прикупљање захтева и анализа пословног система.
- Пројектовање:
  - пројектовање презентационог слоја и корисничких интерфејса,
  - пројектовање информационог модела, пословне логике и доменског модела,
  - пројектовање архитектуре,
  - пројектовање апликационих програмских интерфејса.
- Имплементација и тестирање.

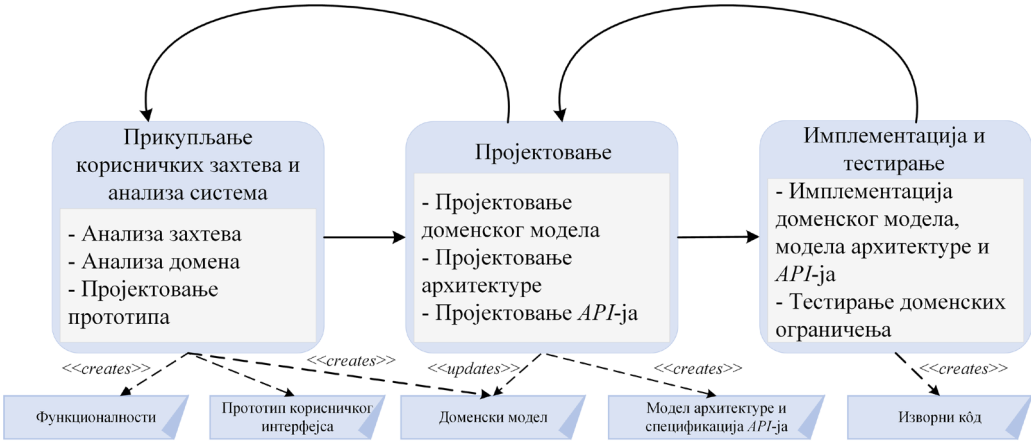
У току изградње модела пословног система, анализира се пословни систем и документују се пословни процеси у складу с корисничким захтевима. На основу модела пословног система, пројектује се архитектура софтвера и приступа се његовој имплементацији и тестирању.

Модел пословног система описује се моделом структуре и моделом динамике. У процесном приступу акценат је на моделовању динамике. Моделовање структуре је у другом плану. Када се направи модел динамике у форми модела процеса, у њему је имплицитно садржан и модел структуре система[238].

[237] Прилагођено према: https://www.geeksforgeeks.org/software-engineering-software-design-process/
[238] Winter, M. & Pryss, R. (2023). The effects of modular process models on gaze patterns-A follow-up investigation about modularization in process model literacy, *Expert Systems with Applications*, 121176.

Прелаз са објектног на процесни приступ донео је велико олакшање пројектантима пословног софтвера, јер се у свим фазама пројектовања софтвера, од анализе реалног система, дефинисања архитектуре, па до конкретне имплементације, на свим нивоима апстракције барата концептима истог типа са истим називима: догађајима, сервисима, процесима, као и њиховим интеракцијама.

Типичан редослед активности при пројектовању процесно-оријентисаног софтвера базираног на микросервисима, коришћењем доменом вођеног пројектовања софтвера, дат је на Слици 23.



Слика 23. Активности у пројектовању софтвера оријентисаног на процесе[239]

15. МОДУЛАРНОСТ

Историја развоја концепата софтверског инжењерства, од процедуралних апстракција и компоненти, објектно-оријентисаног програмирања и пакета, па до микросервиса и контејнера, може се посматрати као историја трагања за најпогоднијим концептима модуларности и интеграције, са циљем да се задовоље све сложенији кориснички захтеви[240].

Модули су, у свим својим појавним облицима и изазовима које су решавали, креирали путању на којој се софтверско инжењерство налази данас. Без њиховог утицаја и континуалне адаптације на нове технолошке изазове, иновације којима смо сведочили у последњим деценијама можда не би биле могуће[241]. Кроз симбиозу

[239] Прилагођено према: Hippchen, B., Giessler, P., Steinegger, R., Schneider, M. & Abeck, S. (2017). Designing microservice-based applications by using a domain-driven design approach, *International Journal on Advances in Software*, 10(3&4), 432–445.
[240] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules, *Communications of the ACM,* 15(12), 1053–1058. https://dl.acm.org/doi/pdf/10.1145/361598.361623
[241] Wirth, N. (2008). A brief history of software engineering, *IEEE Annals of the History of Computing*, 30(3), 32–39.

с другим технологијама[242], концепти модуларности и интеграције свакако ће имати централну улогу и у будућем развоју софтверског инжењерства.

Модуларност је способност разлагања сложеног софтверског система у скуп међусобно повезаних софтверских подсистема који се називају компоненте. Компоненте се даље могу разлагати на модуле. Модул је апстракција софтверског подсистема, заснована на енкапсулацији и повезивању. Мера разложености софтвера на модуле се назива грануларност[243].

Модул поседује структуру и функционалност, а садржи све што је неопходно да се реализује један аспект понашања или жељене функционалности. Карактеристике модула су: самосталност, заменљивост и способност интеграције његове функције са окружењем[244].

Модуларизација (*modularization*), као облик апстракције, примењује се у свим фазама софтверског процеса и представља снажан интелектуални апарат којим се пројектанти софтвера користе при суочавању са сложеношћу софтверских система. Циљ модуларизације је развој и имплементација софтвера који се може лако адаптирати, проширити и одржавати. Резултат модуларизације је пројекат софтверског система који се састоји од скупа независних и измењивих модула повезаних интерфејсима[245].

Модули се дефинишу и имплементирају у више слојева архитектуре. Функционалности модула дефинишу се у највишим слојевима, а дистрибуираност, скалабилност и поузданост, на нижим слојевима, ближим инфраструктури.

Модули се могу користити у имплементацији, како монолитног, тако и дистрибуираног софтвера. Свођење модуларности на један слој, само на објекте и класе, у монолитном (објектно-оријентисаном) софтверу, омогућава само ефикасније одржавање софтвера, без додатних предности модуларизације које се односе на дистрибуираност, скалабилност и поузданост које се имплементирају на нижим слојевима модуларности.

Архитектура софтвера оријентисаног на процесе у основи је дистрибуирана, вишеслојна и погодна за модуларизацију. Модули се организују у форми апликација, процеса, сервиса и микросервиса. Спецификују се коришћењем *REST API*, који омогућава транспарентно, једноставно и ефикасно обезбеђивање дистрибуираности, скалабилности и поузданости.

---

[242] Rellermeyer, J. S., Duller, M. & Alonso, G. (2009, May). Engineering the cloud from software modules. In: *2009 ICSE workshop on software engineering challenges of cloud computing* (pp. 32–37). IEEE.
[243] Kästner, C., Apel, S. & Kuhlemann, M. (2008, May). Granularity in software product lines. In: *Proceedings of the 30th international conference on Software engineering* (pp. 311–320).
[244] Sullivan, K. J., Griswold, W. G., Cai, Y. & Hallen, B. (2001). The structure and value of modularity in software design, *ACM SIGSOFT Software Engineering Notes*, 26(5), 99–108.
[245] Parnas, D. L., Clements, P. C. & Weiss, D. M. (1985). The modular structure of complex systems. *IEEE Transactions on software Engineering*, (3), 259–266.

Модули се имплементирају у програмским језицима високог нивоа. Програмски језици *Fortran*, *Cobol*, *Algol*, *Pascal*, *C* и *C++* у првим верзијама нису имали подршку за модуле. Подршка за модуле је додата у каснијим верзијама тих програмских језика. *Ada*, *Modula* и *Modula* 2 су први језици који су у својој основној спецификацији садржали подршку за модуле.

Објектно-оријентисано програмирање, инсистирањем да објекат и класа буду ексклузивни начини енкапсулације, ограничило је могућности модуларизације и значајно је успорило развој програмских језика. Напуштањем стриктног објектно-оријентисаног приступа у првој деценији 21. века, створени су услови за формализацију и ширу примену концепата модуларности и интеграције.

Програмски језик *Java* садржао је рудиментарну подршку за модуле преко организовања објеката, класа и њихових интерфејса, коришћењем Пакета (*Package*, *Java Namespaces*)[246], а од верзије 9. у 2017. години додата је потпуна подршка за модуле преко *Java platform Module System*[247]. Програмски језик *C#* је на сличан начин формализовао организацију модула и њихових интерфејса преко *C# Namespaces*[248]. Савремени програмски језици, као што су *JavaScript*[249], *Python*[250] и *Perl*[251], у актуелним верзијама, или преко екстензија, такође подржавају концепт модуларности.

Најзначајнији искорак ка широј и ефикаснијој примени концепта модуларности настао је развојем инфраструктуре и сервиса рачунарства у облаку[252,253]. У *Kubernetes*-у, модули се могу представити пôдовима[254], који се састоје из више контејнера, у којима су садржани сервиси који се могу динамички дистрибуирати, и којима се може аутоматски додељивати неопходна скалабилност[255]. Области именовања, *Namespaces* у *Kubernetes*-у, доприносе јасном раздвајању модула, како при спецификацији, тако и при њиховом извршењу.

---

[246] https://docs.oracle.com/javase/8/docs/api/index.html
[247] https://openjdk.org/projects/jigsaw/spec/
[248] https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/namespaces
[249] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules
[250] https://docs.python.org/3/tutorial/modules.html
[251] http://www.cpan.org/modules/index.html
[252] Rellermeyer, J. S., Duller, M. & Alonso, G. (2009, May). Engineering the cloud from software modules. In: *2009 ICSE workshop on software engineering challenges of cloud computing* (pp. 32–37). IEEE.
[253] Guo, R., Tafti, A. & Subramanyam, R. (2023). Internal IT modularity, firm size, and adoption of cloud computing, *Electronic Commerce Research*, 1–30. https://link.springer.com/article/10.1007/s10660-023-09691-8
[254] https://kubernetes.io/docs/concepts/workloads/pods/
[255] https://medium.com/kokster/modular-kubernetes-programming-69f1f13db996

15.1 Организација модула

Начин груписања функционалности и организација модула у највећој мери зависи од пројектоване архитектуре софтвера[256]. Модули се најчешће организују и групишу по слојевима архитектуре, и то у[257]:

Модуле за управљање подацима (*Data Modules*). Садрже базе података, репозиторијуме, разноврсне изворе података и њихове апстракције, као и функционалности за приступ подацима и њихово одржавање. У вишеслојним архитектурама користе се за имплементацију инфраструктурног слоја, слоја података и доменског слоја. Примарни задаци ових модула су:

- Енкапсулација података и пословне логике за одређени домен.
- Излагање података енкапсулираних у модул према окружењу, преко јавног апликационог програмског интерфејса (*Public API*). Подаци у аргументима *API* представљају се као апстрактни репозиторијум који је независан од конкретног извора података.
- Сакривање имплементационих детаља који су везани за извор података. Из окружења је изворима података могуће приступити само преко *API*, и скупа трансформација који апстрактни репозиторијум, дефинисан у *API*, пресликавају у конкретан извор података.

Модуле функционалности (*Feature modules*). Енкапсулирају функционалности и пословну логику, које могу, али не морају бити укључене у основну апликацију. Специфичне функционалности се могу укључити у апликацију условно или преузети на захтев[258]. Модули функционалности се у вишеслојним архитектурама користе за имплементацију доменског слоја и слоја пословне логике. У свом раду користе модуле за управљање подацима.

Апликационе модуле. Апликациони модули (*App modules*) имплементирају апликациону логику. Користе модуле функционалности за спецификацију апликационе логике. Обезбеђују улаз на почетну страницу апликације.

Заједничке модуле. Заједнички модули (*Common modules*, *Core modules*) садрже кôд који други модули често користе. Смањују редундансу и доприносе лакшем одржавању софтвера. Не представљају ниједан специфичан слој архитектуре и могу се наћи у свим архитектурним слојевима. Примери заједничких модула су: модули корисничког интерфеја (*UI modules*), модули аналитике (*Analytics modules*), мрежни модули (*Network modules*) и услужни модули (*Utility modules*).

---

[256] Sethi, K., Cai, Y., Wong, S., Garcia, A. & Sant'Anna, C. (2009, September). From retrospect to prospect: Assessing modularity and stability from software architecture. In: *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture* (pp. 269–272). IEEE.

[257] https://developer.android.com/topic/modularization/patterns

[258] https://developer.android.com/guide/playcore/feature-delivery

15.2 Повезивање и комуникација модула

Модули у ретким случајевима егзистирају у потпуној издвојености. Модули, по правилу, а нарочито у вишеслојним архитектурама, зависе од других модула и с њима комуницирају. У комуникацији, модули размењују структуриране и неструктуриране податке који су дефинисани апликационим програмским интерфејсима. У неким случајевима, модули могу размењивати податке преко заједничке базе података.

Пренос података између модула врши се уз помоћ комуникационог медијума. Комуникациони медијум може бити рачунарска мрежа, *Unix Socket*, или систем за размену порука. Модули могу комуницирати директно, или посредно путем трећег модула који преузима улогу посредника. Комуникација преко посредника се користи у случајевима када није могућа директна комуникација између модула или пак није пожељна због архитектурних или имплементационих ограничења. Модул посредник се користи нарочито у случајевима када је потребно ослабити спрегу између модула, обезбедити дистрибуираност, скалабилност и поузданост.

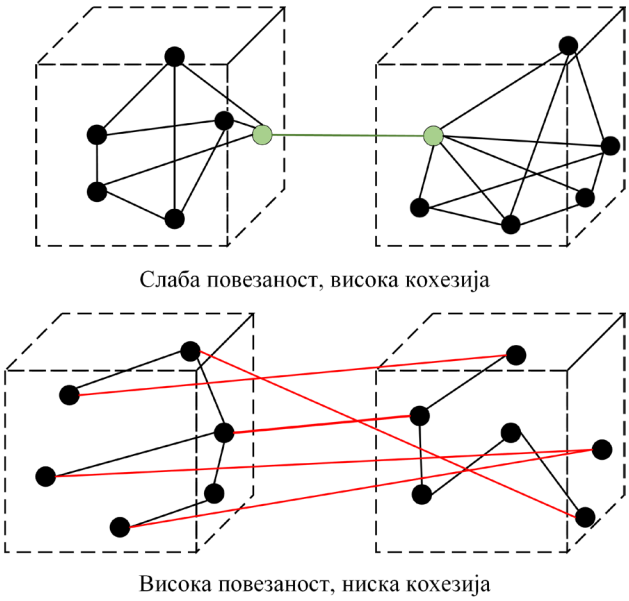
Индикатори помоћу којих се вреднује квалитет и успешност повезивања модула су: спрега између модула (*coupling*), која се односи на меру у којој је један модул зависан од других модула, и кохезија унутар модула (*cohesion*), која показује у којој мери су задаци или одговорности унутар једног модула међусобно повезани[259]. Међусобни однос модула, спреге између њих и кохезије унутар модула приказани су на Слици 24. На горњем делу слике приказан је пример слабе повезаности и високе кохезије, док доњи део слике приказује пример веће повезаности а ниже кохезије.

У добро пројектованом софтверу, спрега између модула треба да буде што мања, а кохезија унутар модула што већа[260]. Избалансирана спрега и кохезија су кључни у ефикасном пројектовању софтвера[261].

[259] Allen, E. B. & Khoshgoftaar, T. M. (1999, November). Measuring coupling and cohesion: An information-theory approach. In: *Proceedings sixth international software metrics symposium (cat. no. pr00403)* (pp. 119–127). IEEE.

[260] Candela, I., Bavota, G., Russo, B. & Oliveto, R. (2016). Using cohesion and coupling for software remodularization: Is it enough?, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(3), 1–28.

[261] https://medium.com/@caitlinjeespn/modularization-in-software-engineering-1af52807ceed



Слика 24. Спреге између модула и кохезија[262]

15.2.1 Спрега између модула

Спрега представља меру зависности између модула. Мања спрега значи да промене у једном модулу имају мањи утицај на друге модуле, што олакшава одржавање и тестирање. Постоје различите врсте спреге између модула:

- **Спрега преко података** (*data coupling*) постоји када модули деле податке искључиво преко параметара или аргумената спецификованих у интерфејсу модула. Сваки модул одржава сопствене податке, не приступа и не модификује податке других модула. Овај тип спреге промовише енкапсулацију и међузависност модула.
- **Спрега путем маркирања** (*stamp coupling*) слабији је облик повезивања од спреге преко података. Модули деле сложену структуру података, а да при томе појединачни модули приликом повезивања не користе све елементе дељене структуре података. У овом облику повезивања, подаци у дељеној структури података треба да буду организовани и постављени тако да се нежељени или некоришћени подаци не деле, нити прослеђују између модула.
- **Спрега преко глобалних података или ресурса** (*common coupling*) јавља се у случајевима када модули деле глобалне структуре података или ресурсе. Ова врста спреге доводи до проблема у вези са конкурентним програмирањем и приступом критичном региону, па ову врсту спреге између модула треба избегавати, кад год је то могуће и оправдано.

[262] Прилагођено према: https://medium.com/wix-engineering/what-exactly-does-low-coupling-high-cohesion-mean-9259e8225372

- **Спрега путем управљања** (*control coupling*) појављује се у случајевима када један модул контролише понашање другог модула. Овај тип спреге подразумева да контролни модул има знање о интерној структури и функционисању контролисаног модула, што у већини случајева отежава одржавање софтвера.
- **Спрега преко окружења** (*external coupling*) представља меру утицаја промена у структури окружења на испуњење функционалности модула. У случају ниске повезаности са окружењем, структурне промене у окружењу имају мали утицај на унутрашњу имплементацију модула и начин функционисања модула. При средњем степену повезаности, промене у структури окружења захтевају извесне модификације унутар модула и прилагођавања новим интерфејсима из окружења. Висок ниво спреге са окружењем најмање је пожељан облик спреге, јер промене у структури окружења имају значајан утицај на имплементацију модула, што захтева опсежне модификације унутар модула.
- **Спрега преко садржаја модула** (*content coupling*), код ове врсте спреге један модул користи или модификује рачунарски кôд другог модула. Ово је најлошији вид спреге, јер промене у једном модулу могу бити узрок непредвиђених последица у другим модулима.

Предности слабе спреге између модула

При слабој спрези између модула смањен је утицај промена у једном модулу на структуру и функционисање других модула. Тиме се побољшава одржавање, поједностављују се измене у кôду или замена комплетних модула. Модули се могу развијати и тестирати у изолацији, побољшавајући модуларност и поновно коришћење рачунарског кôда. Слаба спрега омогућава једноставно додавање нових модула или повлачење постојећих, поједностављујући скалирање перформанси у реалном времену.

Недостаци јаке спреге између модула

Јака спрега између модула резултује повећањем зависности између модула, чинећи софтверски систем комплекснијим и тежим за разумевање, пројектовање и одржавање. Смањена је флексибилност софтвера, што чини ажурирање, промену или замену модула без утицаја на функционисање комплетног система комплексним и тешким, а некада и немогућим. Јака спрега између модула, развој и тестирање модула у изолацији, систем чини комплекснијим, смањујући ниво модуларности и могућност поновног коришћења рачунарског кôда. Јаку спрегу између модула треба избегавати, али када се појави, треба је у процесу пројектовања софтвера ослабити применом архитектурних стилова и патерна који то омогућавају.

15.2.2 Кохезија

Кохезија се односи на степен у којем су функције или задаци унутар модула повезани. Висока кохезија значи да модул обавља добро дефинисан задатак. То олакшава тестирање, отклањање грешака и читљивост кôда. Врсте кохезије су:

- **Функционална кохезија** (*functional cohesion*), када су сви елементи потребни за извршење функционалности модула садржани у модулу и распоређени у циљу обављања његове функције. При томе су елементи модула међусобно слабо повезани. То је идеалан и пожељан облик кохезије унутар модула. Карактеристичан је за микросервисне архитектуре.
- **Секвенцијална кохезија** (*sequential cohesion*) се јавља у случају серијске обраде, када излази из једног елемента представљају улазе у други елемент модула. Овај облик кохезије је својствен функционалним програмским језицима, а често се среће и у патернима обраде токова догађаја. Секвенцијална кохезија спада у пожељне особине модула.
- **Комуникациона кохезија** (*communicational cohesion*) се јавља при дефинисаном протоколу обраде, када елементи модула раде над истим подацима, по одређеном редоследу и дефинисаним начинима интеракције. Представља пожељан облик кохезије.
- **Процедурална кохезија** (*procedural cohesion*) се јавља када су елементи модула груписани по основу редоследа извршења. Процедурална кохезија се може наћи у структурираним програмским језицима. Представља мање пожељан облик кохезије.
- **Временска кохезија** (*temporal cohesion*), код које су елементи модула повезани временом и редоследом њиховог извршења. Користи се код модула за иницијализацију софтверских система, а среће се и код патерна за обраду догађаја.
- **Логичка кохезија** (*logical cohesion*), код које су елементи модула логички повезани, иако се односе на различите функционалности.
- **Информациона кохезија** (*informational cohesion*) се јавља када су елементи модула груписани на основу њиховог односа према некој структури података или објекту. Најчешће се појављује у објектно-оријентисаном програмирању.
- **Кохезија слојева** (*layer cohesion*) постоји када су елементи модула груписани на основу нивоа апстракције. Кохезија слојева се обично користи у великим софтверским системима за организовање кôда у слојеве, којима се може лакше управљати[263].
- **Случајна кохезија** (*coincidental cohesion*), када елементи модула нису повезани у некој смисленој форми, осим локације у изворном кôду модула. То је најнепожељнији облик кохезије.

[263] https://www.geeksforgeeks.org/software-engineering-coupling-and-cohesion/

Предности јаке кохезије

Јака кохезија унутар модула је резултат јасно дефинисаног и фокусираног модула на једну добро дефинисану сврху. Тиме се омогућава једноставније пројектовање, разумевање кôда и одржавање. Јака кохезија смањује вероватноћу утицаја промене у једном делу модула на друге делове модула, омогућавајући једноставнију изолацију и исправку грешака. Јака кохезија чини да су модули мање подложни грешкама и функционишу конзистентније.

Недостаци слабе кохезије

Слаба кохезија резултује модулима који немају јасну сврху и садрже елементе који не припадају једној целини. На тај начин се смањује њихова функционалност и отежава одржавање. Слаба кохезија може отежати пројектантима разумевање сврхе и понашање модула, што доводи до грешака у процесу пројектовања и недостатка јасноће у функционисању модула. Ниска кохезија доводи и до понављања кôда, јер елементи модула могу бити расути и понављати се у више различитих модула[264].

Избалансирана спрега између модула и кохезија унутар модула омогућавају пројектовање софтвера у којем је сваки модул самосталан, са јасно дефинисаним интерфејсом и одговорностима. Ово олакшава интеграцију, одржавање и проширење софтвера.

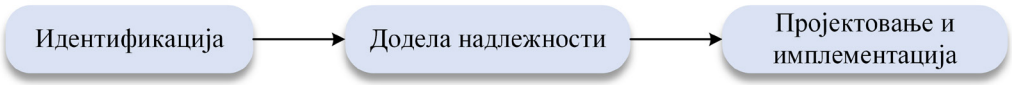
15.2.3 Раздвајање надлежности

Раздвајање надлежности (*Separation of Concerns*[265]) је методолошки поступак у пројектовању софтвера који обезбеђује да различите надлежности не коегзистирају унутар архитектуре или кôда једног модула. . Настао је из „Принципа јединствене одговорности“ (*Single Responsibility Principle*), дефинисаног у скупу принципа пројектовања софтвера под називом *SOLID*[266]. Раздвајање надлежности у већини случајева доприноси постизању слабе спреге између модула, уз очување високог степена кохезије[267].

Поступак раздвајања надлежности у највећој мери зависи од корисничких захтева и састоји се од: идентификације различитих надлежности дефинисаних корисничким захтевима; доделе једне надлежности једном модулу; пројектовања и имплементације модула (Слика 25).

---

[264] Малков, С. (2021). *Пројектовање софтвера* (књига у припреми), 2021.
[265] Dijkstra, E. W. (1982). On the role of scientific thought. Selected writings on computing: a personal perspective, 60–66.
[266] Arora, G. K. (2017). SOLID Principles Succinctly. https://www.freedownloads247.com/Uploaded-Files/8-2017/4223/solidprinciplessuccinctly.pdf
[267] Mili, H., Elkharraz, A. & Mcheick, H. (2004). Understanding separation of concerns. *Early aspects: aspect-oriented requirements engineering and architecture design*, 75–84.



Слика 25. Поступак раздвајања надлежности између модула[268]

Предности раздвајања надлежности су инхерентна подршка високој кохезији модула, подршка слабој спрези између модула, повећање модуларности, могућности одржавања и поновне употребе кôда.

Раздвајање надлежности се може вршити по различитим критеријумима, као што су: подаци, пословна правила, функционалности софтвера и слично. Раздвајање надлежности представља проблем вишекритеријумског одлучивања. Ако постоји доминантан критеријум за раздвајање надлежности, тада се други критеријуми не могу адекватно применити. Ово се у литератури често назива „тиранија доминантне декомпозиције“ (*tyranny of the dominant decomposition*[269]), јер одабрани критеријум декомпозиције програма диктира структуру софтвера и отежава раздвајање надлежности по другим потребним критеријумима.

Раздвајање по подацима који су дефинисани у објектима, доминантан је критеријум за раздвајање надлежности, који није могуће избећи у објектно-оријентисаном програмирању. На почетку су се решења за проблем доминантне декомпозиције тражила у приступима заснованим на мултидимензионом раздвајању надлежности[270], а касније је овај приступ, заједно са објектно-оријентисаним приступом, напуштен и прешло се на примену других парадигми и одговарајућих архитектурних стилова у развоју софтвера, који овај проблем немају, умањују га или избегавају.

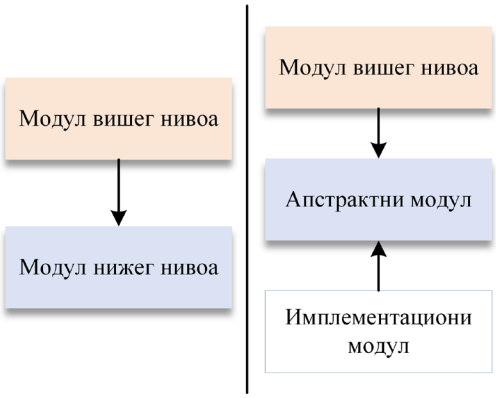
15.2.4 Инверзија зависности

Јаку спрегу између модула, како је претходно објашњено, потребно је у процесу пројектовања софтвера ослабити. Поступци који се у ту сврху често користе су: Инверзија зависности (*Dependency inversion*) и Уметање зависности (*Dependency injection*).

Инверзија зависности је пети принцип из скупа *SOLID* принципа[271]. Представља поступак у коме се јака спрега имеђу модула, у којој модул вишег нивоа зависи од модула нижег нивоа, лабави раздвајањем модула нижег нивоа на два нова модула:

[268] Прилагођено према: https://medium.com/@onuryanar/separation-of-concerns-eca786e75f5a
[269] Tarr, P., Ossher, H., Harrison, W. & Sutton Jr, S. M. (1999, May). N degrees of separation: Multi-dimensional separation of concerns. In: *Proceedings of the 21st international conference on Software engineering* (pp. 107–119).
[270] Ossher, H. & Tarr, P. (2002). Multi-dimensional separation of concerns and the hyperspace approach. *Software Architectures and Component Technology*, 293–323.
[271] https://krishnakishorev.medium.com/the-solid-principles-dependency-inversion-principle-dip-b67194592ebd

апстрактни модул који само спецификује атрибуте и параметре енкапсулације, најчешће у форми *API*, и модул за његову имплементацију (Слика 26). Тиме се постиже да модул вишег нивоа и имплементациони модул буду у јакој спрези према новоуведеном апстрактном модулу који садржи само *API*[272].



Слика 26. Инверзија зависности увођењем апстракног и имплементационог модула[273]

Најзначајнији резултат примене инверзије зависности је **дефинисање неопходне архитектуре за софтверске оквире и платформе**, код којих се у потпуности имплементира архитектура модула вишег нивоа, дефинишу се апстрактни *API* за модуле нижег нивоа, а корисницима се оставља да конкретне имплементације модула нижег нивоа прилагоде сопственим потребама.

15.2.5 Уметање зависности

При инверзији зависности претпоставља се да је имплементациони модул статичан. У пракси, због нових верзија софтвера, увођења нових функционалности, или другачијег начина обраде, намеће се потреба да имплементациони модул има понашање и структуру која се мења динамички. То значи да је потребно обезбедити полиморфизам имплементационог модула у реалном времену. Поступак којим се обезбеђује полиморфизам имплементационог модула у реалном времену, назива се: Уметање зависности (*Dependency injection*) и први пут га је описао *Martin Fowler* 2004. године[274].

Шири појам од уметања зависности је инверзија контроле. Инверзија контроле (*Inversion of Control*) је принцип пројектовања објектно-оријентисаног софтвера који прописује да уместо статичког конфигурисања зависности, класу треба да конфигурише друга спољна класа. Уметање зависности се у објектно-оријентисаном

[272] https://stackify.com/dependency-inversion-principle/
[273] Прилагођено према: https://developer.android.com/topic/modularization/patterns
[274] Fowler, M. (2004). Inversion of control containers and the dependency injection pattern. https://martinfowler.com/articles/injection.html

програмирању имплементира тако што се објекти апстрактног модула прослеђују имплементационом модулу као параметар, уместо да се креирају и њима приступа унутар имплементационог модула. Тиме се контрола креирања и конфигурација зависности врши екстерно и избегава се њихова уградња у имплементациони модул. Поступак креирања и конфигурације зависности у реалном времену је потребно уградити у апликациону логику.

Постоји неколико поступака који се користе за уметање зависности у објектно-оријентисаном програмирању. Најчешће коришћени су[275]:

- *Constructor Injection* – односи се на праксу уметања зависности путем конструктора класе. Ова метода омогућава да се неопходни објекти проследе приликом инстанцирања класе, обезбеђујући тако неопходне ресурсе.
- *Setter Injection* – подразумева уметање зависности путем сетера класе. Уместо кроз конструктор, вредности се постављају путем сетер метода. Овај приступ омогућава динамичке промене зависности током извршавања, пружајући флексибилност.
- *Interface Injection* – подразумева да класе имплементирају интерфејс са методама за уметање зависности. Класе које желе користити ове зависности имплементирају тај интерфејс, обезбеђујући потребне методе за убацивање. Овај приступ пружа флексибилност, али може повећати комплексност јер сви корисници интерфејса морају имплементирати потребне методе.

Уметање зависности није ограничено само на објекте у објектно-оријентисаном софтверу. Може се применити и на модуле и компоненте у процесном приступу развоја софтвера. Коришћење апликационих корисничких интерфејса заснованих на догађајима, уместо оних које контролише процедурални кôд, може се сматрати једним од примера инверзије контроле који није ограничен само на објекте и објектно-оријентисане програме.

Идеја која стоји иза уметања зависности јесте да компоненте или модули не треба да буду свесни конкретне имплементације коју користе у свом раду, већ само интерфејса који дефинише њихову функционалност. Тиме се омогућава замена или ажурирање делова апликације, једноставном заменом показивача на конкретну имплементацију модула, која се може извршити у реалном времену и без утицаја на рад остатка система.

Уметање зависности као методолошки поступак игра кључну улогу у унапређењу модуларности, одрживости и тестирања софтверских система. Тестирање, одржавање и вишеструка употреба програма постају ефикаснији јер корисници нису оптерећени познавањем детаља о имплементацији зависности. Уметање зависности олакшава тестирање, с обзиром на то да се конфигурације могу изместити у конфигурационе датотеке, дозвољавајући конфигурацију система без

[275] Lundberg, A. (2015). Dependency Injection frameworks: an improvement to testability? Umeå University, Sweden.

потребе за поновним компајлирањем. Тиме се ствара погодно окружење за агилније и одрживије софтверске системе. Овакав приступ је посебно користан у већим пројектима где је важно одржавати модуларност и раздвојеност компонената[276].

Уметање зависности у софтверу оријентисаном на процесе врши се тако што се јака веза између модула лабави увођењем *Pub/Sub* посредника, који у реалном времену управља уметањем зависности, путем претплате на догађаје и објављивањем догађаја.

У сервисно-оријентисаним архитектурама које користе *REST API*, уметање зависности своди се на измену документа који садржи репрезентацију ресурса у коме се само промене линкови тако да указују на нове ресурсе. Конкретно уметање зависности у реалном времену врше инфраструктурне компоненте, као што су *API* прокси и *API* гејтвеј.

Уметање зависности представља и специфичан случај примене инверзије контроле у управљању контејнерима, где контејнер пружа конфигурабилни контекст апликацији[277].

У процесу пројектовања софтвера оријентисаног на процесе, поред поштовања свих принципа и препорука за поступак модуларизације, где год је то могуће, за спрегу између модула потребно је користити софтверске стилове и патерне који смањују спрегу између модула, нпр. *REST API*, *API* прокси и *Pub/Sub*. За имплементацију модула препоручује се коришћење микросервиса, који инхерентно имају јаку кохезију, и контејнера као њиховог извршног окружења. Тиме се у већини случајева постиже грануларност модуларизације на нивоу једног микросервиса, слаба спрега и јака кохезија унутар модула, са свим предностима које из тога произлазе[278].

16. ИНТЕГРАЦИЈА

Интеграција представља процес повезивања модула и компоненти у функционалне апликације. За потребе интеграције пројектује се посебна архитектура софтвера. У оквиру архитектуре софтвера, слој интеграције обезбеђује механизме за ефикасну размену података и функционалности између апликација, система и сервиса.

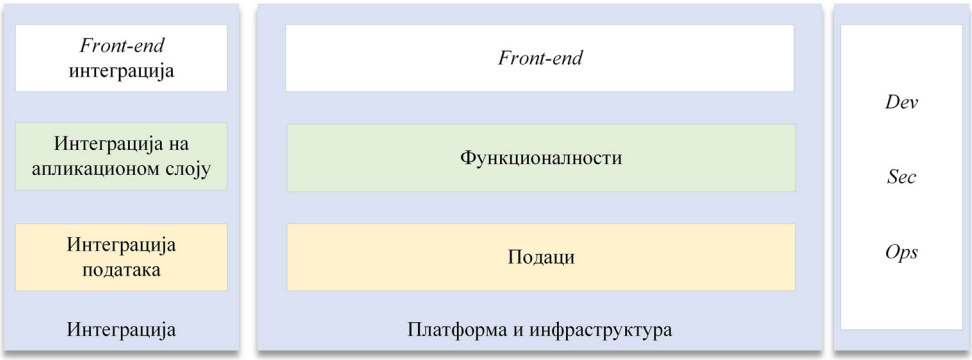
Интеграција модула и компоненти у сложен софтверски систем није једноставан а ни једнозначан задатак. Предмет интеграције су најчешће вишеструке инстанце апликација, које се покрећу на различитим платформама, и на различитим локацијама. Да би се тај сложени посао олакшао, добављачи софтвера нуде софтверске пакете за интеграцију пословних апликација преко платформи и

[276] https://www.techtarget.com/searchapparchitecture/definition/dependency-injection
[277] https://www.techtarget.com/searchapparchitecture/definition/dependency-injection
[278] Winter, M. & Pryss, R. (2023). The effects of modular process models on gaze patterns-A follow-up investigation about modularization in process model literacy, *Expert Systems with Applications*, 121176.

софтверских оквира. Међутим, ова техничка инфраструктура обрађује само мали део сложености послова интеграције. Стварни изазови интеграције обухватају широк спектар пословних и техничких проблема. У пројектовању софтвера, интеграцији се мора посветити готово једнака пажња колико се посвећује и модуларности.

Послови интеграције у вишеслојној архитектури софтвера (Слика 27) грубо се могу поделити на:

1. *Front-end* интеграцију,
2. Интеграцију на апликационом слоју,
3. Интеграцију података.



Слика 27. Типови интеграција[279]

16.1 Front-end интеграција

Front-end интеграција односи се на процес повезивања корисничког интерфејса са *back-end* системима и сервисима. Има за циљ да кориснику пружи утисак да различите функционалности или сервиси функционишу као интегрисана целина. Примери укључују интеграцију различитих апликација у оквиру веб портала или коришћење *API*-ја за интеграцију спољних сервиса у кориснички интерфејс. Овакав тип интеграције се категоризује као вертикална интеграција.

16.2 Интеграција на апликационом слоју

Интеграција на апликационом слоју фокусира се на дефинисање архитектуре која повезује апликацију с неопходним функционалностима, које се налазе у различитим софтверским системима у оквиру информационог система предузећа. Тиме се побољшава ефикасност софтвера и способност брзог одговора на захтеве клијената. Овакав тип интеграције се категоризује као хоризонтална интеграција[280].

[279] Прилагођено према: https://www.itarch.info/2020/05/technology-architecture-integration.html
[280] Hasselbring, W. (2000). Information system integration. *Communications of the ACM*, 43(6), 32–38.

Изазов интеграције на апликационом слоју је осигурати да сваки део информационог система може на ефикасан и поуздан начин да комуницира с другим деловима. Ово захтева пажљиво пројектовање архитектуре и имплементацију стандардизованих протокола и интерфејса.

Постоје различити приступи интеграцији на апликационом слоју. Већа предузећа, у којима се још увек користи сервисно-оријентисана архитектура, обично имају магистралу сервиса информационог система предузећа (*Enterprise Service Bus*, *ESB*) која олакшава интеграцију, а најчешћи протоколи и стандарди који се користе за интеграцију су *SOAP* и *REST*.

ESB, као концепт интеграције, због лоших резултата у пракси полако се напушта. Решења за ефикасну интеграцију траже се у оквиру патерна архитектура оријентисаних на процесе, као што је *API* гејтвеј и *Service Mesh*, којима је агрегација инхерентна, а интеграција уграђена у архитектуру. Микросервисна архитектура омогућава раздвајање апликација на мање, самосталне компоненте које се могу лакше интегрисати и скалирати, па је погодна за интеграцију. Кључни нефункционални захтеви интеграције на апликационом слоју укључују сигурност, скалабилност и перформансе. Имплементација безбедносних механизама као што су аутентификација и ауторизација кључни су за заштиту интегритета података и сервиса.

16.3 Интеграција података

Интеграција података је интеграција на инфраструктурном слоју. Усредсређује се на коришћење и трансформацију података из различитих извора како би се омогућила њихова ефикасна анализа и употреба. Овај тип интеграције обезбеђује униформан приступ подацима, без обзира на то где се ти подаци налазе и у каквом су облику. За разлику од интеграције апликационог слоја и *front-end* интеграције, уопште се не бави логиком апликације.

Ова врста интеграције обухвата процесе као што су трансформација, синхронизација и конверзија података из различитих извора и формата. Примери укључују интеграцију података из различитих база података у један виртуелни централни складишни систем. Такође, интеграција комплексних и великих података може укључивати и коришћење *ETL* (*Extract*, *Transform*, *Load*) платформи за пренос и трансформацију података између различитих система и формата, чиме се олакшава рад с различитим типовима података.

Ови типови интеграције углавном се користе за интеграцију и пренос података ка складиштима података (енгл. *data warehouse*) или информационим језерима (енгл. *data lakes*).

17. АРХИТЕКТУРА СОФТВЕРА

Архитектура софтвера је метафора настала из потребе да се добра искуства и концепти, на основу аналогија из архитектуре и грађевинарства: модуларност, интеграција модула у комплексну целину, патерни и њихово поновно коришћење, примене и на пројектовање софтвера.

Архитектура софтвера (енгл. *software architecture*) је апстрактан модел софтвера, чијим дефинисањем започиње процес пројектовања софтвера. Примарни циљ архитектуре је да идентификује нефункционалне и функционалне пројектне захтеве који утичу на структуру софтвера и да генерише апстрактан, модуларан модел софтвера који је с тим захтевима усклађен. Осим дефинисања модула, архитектуром се дефинише и начин њихове интеграције. Модули и интеграција се пројектују тако да архитектура буде у могућности да задовољи пројектне захтеве у реалним условима рада.

Архитектура обухвата приказ структуре софтвера без имплементационих детаља. Састоји се из докумената који садрже: листе атрибута квалитета, функционални опис софтвера, јасан, комплетан, конзистентан и достижан скуп функционалних захтева, архитектуру софтвера у најмање два слоја декомпозиције, оквир за временско планирање и документ или опис процеса који омогућује да се функционална декомпозиција прати.

Архитектура софтвера се пројектује у хијерархијски уређеним слојевима. Примарни циљ при пројектовању првог слоја је дефинисање архитектуре која првенствено задовољава нефункционалне пројектне захтеве, који се односе на дистрибуираност, скалабилност, поузданост, сигурност, квалитет, одржавање и друго. Функционални захтеви се у првом слоју само идентификују с модулима. Детаљна разрада архитектуре модула и њихових функционалности врши се на нижим слојевима, или у процесу имплементације модула.

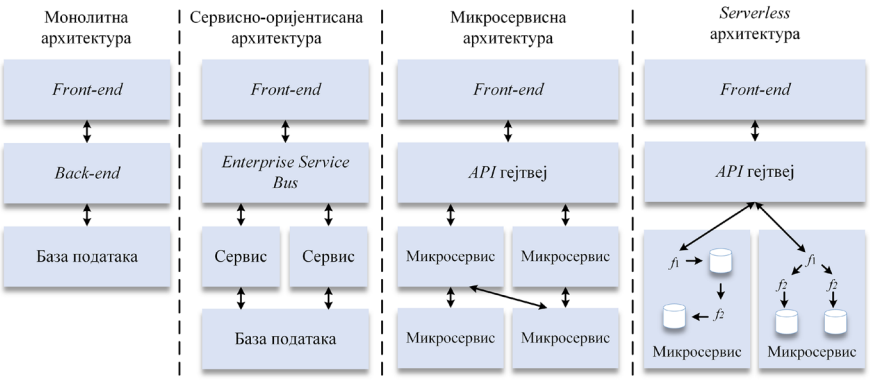
Први рад у коме су изложени концепти архитектуре софтвера објављен је 1962. године[281]. Појам архитектуре софтвера датира из 1969. године[282]. До осамдесетих година двадесетог века користио се углавном у контексту физичке архитектуре рачунарских система. Као засебна дисциплина у оквиру софтверског инжењерства, архитектура софтвера се активно развија почев од деведесетих година, пре свега у циљу отклањања недостатака који су произашли из монолитне архитектуре објектно-оријентисаног приступа развоју софтвера, где су сви елементи (кориснички интерфејс, пословна логика, подаци) садржани у једној апликацији, без модула.

Пројектовањем вишеслојне архитектуре, покушавају се превазићи проблеми који се односе на скалабилност, дистрибуираност, паралелизацију и сигурност софтвера,

[281]Evans, M. K. & Hague, L. R. (1962). Master plan for information systems, *Harvard Business Review*, 40(1), 92–103.
[282]http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF

који се не могу довољно добро решити применом *UML*-а, тада доминантног језика за моделовање објектно-оријентисаних програма. Рудиментарни модел једнослојне архитектуре софтвера у *UML*-у, назива се концептуални модел. У пракси, ни концептуални модел, а ни комплетан *UML*, са свим наредним екстензијама, нису били адекватни за представљање архитектуре савременог софтвера. Вишеслојне апстракције наслеђивања и полиморфизма у примени модула и софтверских патерна није било могуће представити концептуалним моделом, па је из тих разлога *UML* за моделовање архитектуре софтвера напуштен.

Повећана комплексност софтвера довела је до потребе за раздвајањем монолитне архитектуре на модуле, па се тако јављају клијент-сервер архитектуре, а касније и сервисно-оријентисане архитектуре, које омогућују да се поједини делови софтвера независно развијају, скалирају и одржавају. Разбијањем софтвера на још мање делове долази се до микросервисних архитектура, где се сваки микросервис може независно развијати, постављати у продукцију и одржавати. Микросервис је одговоран за појединачну функцију, и комуницира са осталим компонентама преко *API*-ја. Развијају се и *serverless* архитектуре у којима су микросервиси заправо функције које се апликацијама испоручују као *cloud* сервис. Еволуција софтверских архитектура приказана је на Слици 28.



Слика 28. Еволуција софтверских архитектура[283]

Почетком деведесетих година двадесетог века, па надаље, развија се већи број језика за спецификацију архитектуре софтвера (нпр. *ADL*), методе за анализу (нпр. *SAAM*), стандарда за спецификацију архитектуре (нпр. *IEEE*/*ISO*/*IEC* 42010-2022), већи број архитектурних стилова, патерна и платформи с предефинисаним оквирима архитектуре (нпр. *.NET*, *J2EE*). Добар историјски преглед развоја софтверских архитектура и релевантних референци до почетка двехиљадите године може се наћи у тексту[284].

[283] Прилагођено према: https://www.linkedin.com/pulse/evolution-application-software-architecture-jayant-kumar/

[284] Kruchten, P., Obbink, H. & Stafford, J. (2006). The past, present, and future for software architecture, *IEEE software*, 23(2), 22–30. https://www.inf.ed.ac.uk/teaching/courses/seoc/2006_2007/resources/Arc_PPF.pdf

17.1 Монолитна архитектура

Монолитна архитектура је била основни модел архитектуре софтвера за многе организације током дугог временског периода. Апликације су пројектоване као велика целина, са свим компонентама и функционалностима интегрисаним у једну монолитну структуру[285].

У монолитној архитектури, софтвер се пројектује да буде самосталан и независан од других апликација. Софтвер се састоји од компоненти које су чврсто повезане и комбиноване у једну апликацију. Свака компонента мора бити у фукцији, да би се успешно извршило компајлирање, повезивање и софтвер пустио у рад.

Монолитне апликације је тешко одржавати због њихове комплексности. Због некомпатибилности верзија, приликом ажурирања модула, рачунарски програм може доћи у неконзистентно стање (*dependency hell*)[286]. Било каква промена у модулима захтева поновно покретање апликације. Монолитне апликације имају ограничену скалабилност и могућност паралелизације. Паралелизација је могућа на нивоу целе апликације, а не само модула који захтева повећане ресурсе. Коришћење ресурса није оптимално, с обзиром на то да један рачунар треба да пружи ресурсе модулима који имају различите захтеве.

Програмски језици, као што су *Java*, *C*/*C*++ и *Python*, називају се монолитним јер су намењени креирању једног монолитног извршног артефакта, апликације или рачунарског програма. У њима се проблем комплексности структуре решава апстракцијом, поделом рачунарског програма на модуле и спецификацијом њихових веза. Апстракција модуларизације заснива се на подели ресурса у оквиру истог рачунара (меморија, база података, датотеке). Будући да модули зависе од заједничких ресурса, једноставно повећање скалабилности није могуће.

Пре појаве сервисно-оријентисане архитектуре и микросервиса, монолитне апликације су биле стандардни приступ у развоју софтверских решења. Упркос предностима као што су једноставност у развоју и управљању, и мања сложеност у почетној фази развоја, монолитне апликације су се сусретале са изазовима, а најизраженији међу њима су скалабилност и отежан процес интеграције[287].

У оквиру монолитних апликација, процес интеграције често је представљао комплексан и захтеван задатак. Монолитне апликације комуницирају с другим

---

[285] Prasandy, T., Murad, D. F. & Darwis, T. (2020, August). Migrating application from monolith to microservices. In: 2*020 International Conference on Information Management and Technology (ICIMTech)* (pp. 726–731). IEEE.

[286] Fan, G., Wang, C., Wu, R., Xiao, X., Shi, Q. & Zhang, C. (2020, July). Escaping dependency hell: finding build dependency errors with the unified dependency graph. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 463–474).

[287] Abgaz, Y., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., ... & Clarke, P. (2023). *Decomposition of monolith applications into microservices architectures: A systematic review.* IEEE Transactions on Software Engineering.

системима путем директних веза (*point to point*), веб сервиса, *REST API*-ја[288] или путем повезивања са системима за интеграцију апликација (енгл. *Enterprise Application Integration – EAI*)[289].

Интеграција између монолитних апликација и других система често резултира сложеним кôдом, што може отежати одржавање и скалирање апликација. То се донекле може поправити интеграцијом, путем посредничких система за интеграцију, који омогућавају да се монолитна апликација повеже с другим софтверским системима на посебном, посредничком слоју, што може олакшати управљање комуникацијом и омогућити већу флексибилност у случају промена у другим системима. Ипак, и овај приступ може бити комплексан, посебно ако се користе комплексни *EAI* алати који захтевају сложену конфигурацију и одржавање.

Монолитне архитектуре се још увек користе у раним фазама развоја софтвера, због једноставности пројектовања, кодирања и постављања у продукционо окружење. Одржавање монолитног софтвера је отежано, јер измена једног дела кôда уједно значи измену комплетног софтвера. Уколико је потребна већа скалабилност софтвера, биће неопходна и промена архитектуре.

17.2 Сервисно-оријентисана архитектура

Сервисно-оријентисана архитектура (енгл. *Service Oriented Architecture - SOA*) је превазишла проблеме монолитних апликација увођењем концепта дистрибуираног сервиса. У сервисно-оријентисаној архитектури, апликације се организују као колекција независних сервиса који комуницирају или међусобно координирају активности. Основне карактеристике сервиса су: разоткривост, динамичко повезивање, мрежно-адресибилан интерфејс, самосадрживост, модуларност, слаба спрега и способност самоизлечења. Сервисно-оријентисана архитектура испуњава захтеве који се односе на дистрибуираност, скалабилност и поузданост. Комуникација сервиса се типично одвија преко http протокола, па се сервиси могу налазити било где на интернету. Сервис се може поставити на више различитих сервера, чиме се постижу и скалабилност и поузданост.

Као последица дистрибуиране модуларности сервиса, у сервисно-оријентисаној архитектури све више постаје изражен проблем интеграције. Решавање овог проблема је покушано увођењем магистрале сервиса (енгл. *enterprise servise bus*, *ESB*) као патерна архитектуре софтвера који подржава размену података и порука између сервиса[290]. *ESB* представља интеграциони слој који је задужен за операције попут трансформације података, конверзије протокола, рутирање порука,

[288] Al-Debagy, O. & Martinek, P. (2018, November). A comparative review of microservices and monolithic architectures. In: *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)* (pp. 000149–000154). IEEE.
[289] Linthicum, D. S. (2000). *Enterprise application integration*. Addison-Wesley Professional.
[290] Schmidt, M. T., Hutchison, B., Lambros, P. & Phippen, R. (2005). The enterprise service bus: making service-oriented architecture real, *IBM Systems Journal*, 44(4), 781–797.

оркестрација сервиса и друге[291]. Такође, *ESB* пружа могућности за имплементацију различитих патерна интеграције информационог система предузећа (енгл. *Enterprise Integration Patterns*).

ESB се разликује од традиционалних инфраструктура интеграције пружајући функционалности као што су[292]:

1. Динамичко рутирање и конфигурација: *ESB* пружа могућности динамичког рутирања које омогућава креирање сложених токова порука између сервиса. Унутар *ESB*, руте се могу прилагодити на основу специфичних конфигурационих порука, пружајући флексибилност у процесу рутирања порука.
2. Посредовање: *ESB* пружа посредничке услуге за трансформацију или транслирање протокола, формата порука и садржаја. Ово је кључно за интеграцију, јер апликације често не користе исти формат података. На пример, *ESB* користи алате попут *XSL* и *XPath* за обраду *XML* порука.
3. Адаптери за интеграцију апликација: *ESB* решења долазе са уграђеним адаптерима за апликације као што су *ERP* (енгл. *Enterprise resource planning*), *SCM* (енгл. *Supply chain management*) и *CRM* (енгл. *Customer relationship management*). Ови адаптери успостављају везе са „нативним“ трансакционим интерфејсима, *API*-јима и структурама података унутар пословних апликација, нудећи стандардизовани интерфејс који олакшава поновну употребу пословних функционалности и података.
4. Безбедност: *ESB* обезбеђује сигурну размену порука кроз енкрипцију и декрипцију садржаја порука, механизме аутентификације и контроле приступа за крајње тачке размене порука. На овај начин се чува интегритет и поверљивост података.
5. Управљање и надгледање процеса интеграције: *ESB* пружа алате за надгледање инфраструктуре и различитих сценарија интеграције путем ревизије и вођења логова. Такође, нуди централизоване механизме за конфигурацију и администрацију активности што омогућава ефикасну контролу и надзор над процесима интеграције.

Недостатак овог приступа је то што послови интеграције и пословне логике нису стриктно раздвојени, чиме су отежани измена, тестирање и одржавање. Значајан део пословне логике налази се у *ESB* слоју, развија се, примењује и скалира у време извршавања, у монолитној архитектури[293]. Ово је уједно и највећи недостатак, који је довео до постепеног напуштања овог приступа.

У наредним фазама развоја сервисних архитектура, строго се раздвајају елементи архитектуре за функционалне захтеве и архитектура за нефункционалне захтеве.

---

[291] https://aws.amazon.com/what-is/enterprise-service-bus/

[292] Menge, F. (2007, August). Enterprise service bus. In: *Free and open source software conference* (Vol. 2, pp. 1–6).

[293] Aziz, O., Farooq, M. S., Abid, A., Saher, R. & Aslam, N. (2020). Research trends in enterprise service bus (ESB) applications: A systematic mapping study, *IEEE access*, 8, 31180-31197.

Тиме се избегава мешање пословне логике са интеграцијом, па се пословна логика и интеграција могу независно развијати, тестирати и постављати у продукционо окружење. Интеграције се уграђују прво у архитектуру а затим и у инфраструктуру, па се сервиси за интеграцију нуде као уграђене функционалности софтверских платформи и екосистема.

17.3 Микросервисна архитектура

Микросервисна архитектура је варијанта сервисно-оријентисане архитектуре која подразумева већу независност између сервиса који немају заједничке ресурсе, већ комуницирају искључиво преко порука. Намена микросервиса је извршавање једне функционалности. Софтвер се у микросервисној архитектури развија као колекција малих, независних сервиса који функционишу у сопственим процесима, с могућношћу независног развоја и имплементације[294]. Сваки микросервис у микросервисној архитектури садржи сопствене функције за управљање корисницима, пословну логику и функције *back-end*-а. Такође, могуће је да микросервис има и своју базу података. Могуће је и дељење базе података између више микросервиса, у зависности од потреба и захтева апликације. Главни принципи који леже у основи архитектуре микросервиса подразумевају да сваки сервис има једну одговорност (први *SOLID* принцип). Микросервиси су конципирани као аутономне јединице, самосталне и независне, потпуно одговорне за извршавање одређене функционалности. Због те аутономије, они садрже све своје зависности, укључујући библиотеке, извршна окружења попут веб сервера, контејнера или виртуелних машина[295]. Оваква архитектура погодна је за паралелизацију и управљање скалабилношћу.

Микросервисна архитектура се појавила као парадигма за превазилажење недостатака *ESB*/*SOA* приступа, издвајањем по једне функционалности у један микросервис и строгог раздвајања микросервисне архитектуре за имплементацију функционалних захтева, од микросервисне архитектуре за нефункционалне захтеве и интеграцију.

Задаци које *ESB* обавља у сервисно-оријентисаној архитектури не нестају преласком на архитектуру микросервиса. Уместо тога, функционалности централизованог *ESB* се раздвајају и распоређују на појединачне микросервисе. Уклањањем *ESB* слоја у архитектури микросервиса поставља се питање где имплементирати функционалности интеграције које су обрађиване од стране *ESB*-а, као што су: рутирање порука, трансформације протокола и формата порука, и други патерни интеграције информационог система предузећа. Најсложенији изазов у имплементацији микросервисне архитектуре није пројектовање и имлементација

[294] Lewis, J. & Fowler, M. (2014). *Microservices: a definition of this new architectural term* (2014). URL: http://martinfowler.com/articles/microservices.html

[295] Nadareishvili, I., Mitra, R., McLarty, M. & Amundsen, M. (2016). *Microservice architecture: aligning principles, practices, and culture*. O'Reilly Media, Inc.

микросервиса, већ њихова међусобна комуникација. Решења се траже у примени специјализованих алата за интеграцију микросервиса, примени архитектура заснованих на догађајима, примени различитих интеграционих патерна, као што су *API* гејтвеји, *Service Mash* и др.

17.4 Serverless архитектура

Serverless архитектура карактеристична је за извршење софтвера у *cloud* окружењу. Омогућује развој комплексних, дистрибуираних апликација коришћењем релативно једноставних функција, без потребе да се програмер бави управљањем серверским ресурсима (нпр. скалирањем) или операцијама (нпр. аутентификацијом)[296]. Серверски део инфраструктуре и даље постоји, али о њему води рачуна провајдер. Проблеми везани за конфигурацију, управљање, капацитет и одржавање апстрахују се коришћењем алата провајдера. Управљање инфраструктуром је у великој мери аутоматизовано, коришћењем алата као што је *Kubernetes*.

17.5 Архитектура софтвера заснованог на процесима

Развој архитектура пратио је еволуцију парадигми развоја софтвера. Развој архитектура непосредно је био повезан и у великој мери одређен и ограничен развојем концепата модуларности и интеграције.

Основни принцип процесно-оријентисане архитектуре, паралелно извршење рачунарских програма, појавом концепата конкурентног програмирања, кернела оперативног система, пријемпције и средстава за координацију и синхронизацију процеса, почео је још шездесетих година двадесетог века интензивно да се користи у пракси. Оперативни системи су прва област софтверског инжењерства која је у потпуности имплементирала концепте процесно-оријентисане архитектуре софтвера, још почетком седамдесетих. Рачунарски програми у оквиру оперативног система посматрају се као паралелни процеси. Процеси (рачунарски програми) могу имати и подпроцесе, који се назвају „нити“ (*threads*), који се паралелно извршавају и деле заједнички адресни простор с надређеним процесом. Меморијски адресни простор процеса је заштићен од приступа других процеса и припада искључиво процесу, заједно с придруженим нитима. Комуникација и синхронизација различитих процеса обавља се преко средстава кернела оперативног система: региона, семафора, догађаја и услова. За размену порука, између процеса користи се структурни механизам оперативног система коначни бафер (*bounded buffer*), често називан и „ток података“ (*pipe*). Поред коначног бафера, процеси могу размењивати податке и преко специјалног и за ту сврху одвојеног сегмента меморије, коју контролише кернел, под називом дељена меморија (*shared mem-*

[296] Van Eyk, E., Iosup, A., Seif, S. & Thömmes, M. (2017, December). The SPEC cloud group's research vision on FaaS and serverless architectures. In: *Proceedings of the 2nd International Workshop on Serverless Computing* (pp. 1–4).

ory). Динамика извршења процеса манифестује се у форми догађаја као што су: извршење инструкције програма, промена садржаја меморије или промена стања улазно-излазног уређаја. Од појаве првих мултипрограмских оперативних система, па до модерних оперативних система (*UNIX*, *LINUX*, *MS-WINDOWS*), у процесно-оријентисаној архитектури софтвера, на нивоу концепата се ништа није променило. Основи концепт ове архитектуре је догађај, а процес чини секвенца догађаја. Од тог времена до данас, свака платформа или окружење намењено за извршење софтвера оријентисаног на процесе, па и оно најмодерније, мора садржати макар основне елементе кернела мултипрограмског система и концепте конкурентног програмирања за координацију и синхронизацију процеса.

На основу ових концепата, развијала се архитектура софтвера оријентисаног на процесе, најпре дефинисањем различитих нивоа апстракције за процесе и податке, затим и развојем архитектурних модела и патерна за обезбеђивање дистрибуираности, линеарне скалабилности и поузданости, како за програме, тако и за податке. Развијени су и нови методолошки поступци за развој софтвера који ће користити процесно-оријентисане архитектуре, као и нови оквири, платформе и алати за њихову подршку.

Савремене архитектуре софтвера оријентисаног на процесе укључују: архитектуре оријентисане на догађаје, микросервисне архитектуре, контејнере као концепте модуларности, системе за њихову оркестрацију, платформе као сервис рачунарства у облаку које нуде комплетна *serverless* развојна и извршна окружења, *NoSQL* базе података и *Big Data* архитектуре.

17.6 Концепти дефинисања архитектуре софтвера

Основни кораци у пројектовању архитектуре софтвера су: дефинисање контекста и улога, идентификација и планирање атрибута квалитета, пројектовање архитектуре софтвера, израда документације, анализа и вредновање архитектуре софтвера[297].

Пројектни захтеви који имају мерљиви утицај на архитектуру и квалитет софтвера називају се захтевима значајним за архитектуру софтвера. Примери ових захтева су: захтеви у директној вези с пословним вредностима или ризицима, захтеви који су битни стејкхолдерима, захтеви који се тичу атрибута квалитета, које постојећа архитектура не може да испуни, захтеви везани за екстерне компоненте које су непоуздане и не могу се контролисати, захтеви који утичу на више компоненти система и њихову интеракцију, нпр. сигурносни захтеви и захтеви који су изазвали критичне ситуације у прошлости[298].

У циљу испуњења пројектних захтева, доносе се одлуке о архитектури (*Architectural Decision – AD*)[299], као рационалан избор из скупа могућности које испуњавају

[297] https://magdamiu.com/2021/02/22/software-architecture-methodology/
[298] https://www.ozimmer.ch/practices/2020/09/24/ASRTestECSADecisions.html
[299] https://adr.github.io/

постављене захтеве. Ове одлуке, као и разлози за њихово доношење, представљају стандардни део документације архитектуре софтвера. За документацију архитектуре софтвера користе се различити стандарди, перспективе и оквири.

Стандарди за дефинисање архитектуре софтвера

Међународни стандард 42010-2022 – *IEEE/ISO/IEC*[300] прописује да елементи спецификације архитектуре које језик или оквир треба да подржи укључују: стејкхолдере и њихове захтеве, различите перспективе (погледе) на архитектуру ентитета, и различите врсте модела којима се архитектура документује. Стандард специфицира и начин описа ентитета из којих се састоји модел архитектуре. Ентитетима се сматрају софтвер, системи, производи, предузећа, пословни домени, и др. Архитектуре, процеси, методе и алати нису предмет стандарда.

Перспективе (Architecture viewpoints)

Архитектура софтвера се може представити погледима из различитих перспектива (*viewpoints*), аналогно спецификацији архитектуре зграда. Перспектива спецификације архитектуре представља колекцију патерна, темплејта и конвенција за пројектовање једног погледа на софтверски систем[301]. У оквиру једне архитектуре софтвера, може бити више перспектива. Перспективе се могу организовати по категоријама пројектних захтева, или се могу односити на пројектне захтеве групе стејкхолдера.

Категорије по којима се елементи архитектуре групишу у перспективе су[302,303]:

- **контекст** – опис веза, међузависности и интеракције система и окружења.
- **функција** – опис функционалних елемената, њихових одговорности, интерфејса и интеракција.
- **подаци** – опис начина чувања, обраде и дистрибуције информација, укључујући одговоре на питања која се тичу садржаја, структуре, власништва, латенције и сл.
- **конкурентност** – мапирање функционалних елемената у делове система који се могу извршавати паралелно, укључујући механизме координације и синхронизације.
- **развој** – опис архтектуре са аспекта процеса развоја софтвера.
- **деплојмент** – опис окружења у коме ће систем функционисати, укључујући релевантне хардверске компоненте и мапирање софтверских елемената у извршна окружења.

---

[300] https://ieeexplore.ieee.org/document/9938446
[301] Rozanski, N. & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.
[302] https://www.viewpoints-and-perspectives.info/home/viewpoints/
[303] https://pubs.opengroup.org/architecture/archimate31-doc/apdxc.html#_Toc10045495

- **операције** – начин на који ће систем функционисати, како ће се администрирати и како ће се њиме управљати током експлоатације.

Језици за дефинисање архитектуре софтвера

Архитектура софтвера може се спецификовати посебним језицима, генерички названим *architecture description language* (*ADL*). Први језици за дефинисање архитектуре, као што су *AADL*, *Wright*, потичу из академског окружења и омогућују спецификацију архитектуре углавном из аспекта пројектовања компоненти и њиховог повезивања. Новији језици, као што је *ArchiMate*[304], подржавају пројектовање архитектуре софтвера из различитих перспектива. Делимично, за пројектовање архитектуре софтвера могу се користити и *UML* и *BPMN*.

Оквири за дефинисање архитектуре софтвера

Поступак пројектовања архитектуре софтвера може се унапредити, а при томе, у већини случајева и значајно поједноставити применом одабраног оквира пројектовања архитектуре софтвера. Оквир обухвата конвенције, принципе и препоручене праксе којима се документује архитектура.

Оквир се може имплементирати применом више перспектива сагледавања архитектуре и коришћењем различитих језика и нотација за опис архитектуре. Постоји више оквира за дефинисање архитектуре софтвера, као што су: *Arc*42[305], *C*4 модел[306], 4+1[307], *RM-ODP*[308], *TOGAF*[309] и други.

У *Arc*42 модел интегрисан је *Software Architecture Canvas*[310], техника за пројектовање архитектуре софтвера, предложена по угледу на познати *Business Canvas* модел. Елементи који се односе на *business case*, контекст, функционалне захтеве, атрибуте квалитета, организациона и техничка ограничења описују шта софтвер треба да ради. Хипотезама се претпоставља како се то може постићи, техничким изазовима и ризицима се врши евалуација одлуке о избору архитектуре. *Software Architecture Canvas* је приказан у Табели 2.

---

[304] https://www.archimatetool.com/
[305] https://arc42.org/
[306] https://c4model.com/
[307] Kruchten, P. B. (1995). The 4+1 view model of architecture, *IEEE software*, 12(6), 42–50.
[308] https://rm-odp-new.lcc.uma.es/
[309] Прилагођено према: https://www.opengroup.org/togaf
[310] https://www.workingsoftware.dev/software-architecture-canvas/

Табела 2. Software Architecture Canvas

<table>
<tr><td>Software Architecture Canvas</td><td>Софтверски систем:</td><td>Пројектантски тим:</td><td>Датум:</td><td>Итерација:</td></tr>
<tr><td colspan="5">Пословни проблем (Business Case)
Кратак опис пословног проблема</td></tr>
<tr><td colspan="2">Преглед функционалних захтева
Списак најважнијих функционалних захтева које софтвер треба да испуни. Функционалне захтеве треба описати на високом нивоу апстракције.</td><td rowspan="2">Контекст
Одвојити систем који се развија од екстерних елементата (корисника, система, уређаја) са којима систем комуницира.</td><td colspan="2">Организациона ограничења
Организациони захтеви и ограничења која архитекта софтвера треба да узме у обзир приликом пројектовања архитектуре.</td></tr>
<tr><td colspan="2">Атрибути квалитета
Најзначајнији нефункционални захтеви, атрибути квалитета и њихове вредности, за најважније стејкхолдере. Препорука је одабрати три најзначајнија.</td><td colspan="2">Техничка ограничења
Техничка ограничења која утичу на одлуке приликом пројектовања архитектуре софтвера.</td></tr>
<tr><td colspan="3">Хипотезе архитетуре софтвера
На основу анализе осталих елемената, формулишу се хипотезе везане за пројектовану архитектуру софтвера. Важно је укључити хипотезе које се односе на пројектантске одлуке које могу довести до већих трошкова или ризика, као и оправдање.</td><td colspan="2">Технички изазови и ризици
Листа тренутно познатих изазова и техничких ризика.</td></tr>
</table>

Атрибути квалитета

Квалитет софтвера представља меру изврсности софтвера која резултује стањем софтвера у пуној функционалности, без недостатака или дефеката. Квалитет софтвера у значајној мери зависи од архитектуре софтвера. Атрибути квалитета представљају особине софтвера које су различите од функционалних захтева, и могу се класификовати на:

- **статичке атрибуте квалитета**, који рефлектују структуру система. У директној вези су са архитектуром, пројектовањем и изворним кôдом. Нпр. модуларност, одржавање итд. Крајњи корисник статичке атрибуте квалитета не види, али су они у директној вези с трошковима, тако да види њихове последице.
- **динамичке атрибуте квалитета**, који се односе на понашање софтвера током извршења. У вези су са архитектуром, пројектовањем, изворним кôдом, конфигурацијом, деплојментом, окружењем, платформом. Видљиви су за крајњег корисника. Нпр. скалабилност, проток, брзина…

Сценаријима квалитета спецификују се активности којима се управља атрибутима квалитета и спречава отказ софтвера. Примери атрибута квалитета софтвера приказани су у Табели 3[311,312].

[311] https://www.tutorialspoint.com/software_architecture_design/introduction.htm

[312] Silva, S., Tuyishime, A., Santilli, T., Pelliccione, P. & Iovino, L. (2023, March). Quality Metrics in Software Architecture. In: *2023 IEEE 20th International Conference on Software Architecture (ICSA)* (pp. 58–69). IEEE.

Табела 3. Атрибути квалитета софтвера

Категорија	**Атрибут**	**Опис**
Квалитет пројектовања	Концептуални интегритет	Мера конзистентности целокупоне архитектуре.
	Одржавање	Једноставност спровођења промена.
	Поновно коришћење	Способност да се компоненте или подсистеми користе и у другим апликацијама.
Квалитет извршења	Интероперабилност	Способност система да ради кроз комуникацију и размену информација с другим системима.
	Управљивост	Једноставност одржавања система са аспекта администратора.
	Поузданост	Способност система да исправно ради током времена.
	Скалабилност	Способност система да подржи повећан обим посла без утицаја на перформансе.
	Сигурност	Способност система да спречи нападе.
	Перформансе	Мера способности система да изврши тражену акцију у задатом интервалу времена.
	Доступност	Проценат времена у коме систем ради. Може се мерити и у проценту времена у коме систем не ради.
Квалитет система	Подршка	Способност система да пружи информације корисне за решавање проблема током коришћења.
	Тестирање	Мера једноставности креирања тестова за софтверски систем или његове појединачне компоненте.
Употребљивост	Употребљивост	Мера једноставности коришћења система, са аспекта крајњег корисника.
Квалитет архитектуре	Тачност	Степен задовољења свих постављених функционалних и нефункционалних захтева.
Non-runtime Quality	Преносивост	Способност система да ради у различитим рачунарским окружењима.
	Интеграција	Способност да се појединачне компоненте интегришу у целовит систем који исправно ради.
	Модификација	Једноставност одржавања и измена током експлоатације.
Пословна вредност	Трошкови	Трошкови система у односу на тржиште, животни циклус софтвера и употребну вредност.
	Тржишна вредност	Употреба система у односу на конкуренте.

За идентификацију и управљање атрибутима квалитета софтвера може се користити *RACI* матрица. *RACI* матрицом се за сваки атрибут квалитета дефинише који стејкхолдер је:

- *Responsible* – одговоран да уради посао или донесе одлуку,
- *Accountable* – одговоран за тачност и темељност урађеног задатка,
- *Consulted* – одговоран да пружи потребне информације,
- *Informed* – информисан о току извршења задатка.

Приликом идентификације и анализе атрибута квалитета потребно је узети у обзир планиране промене у систему, очекивани раст, и ванредна оптерећења система.

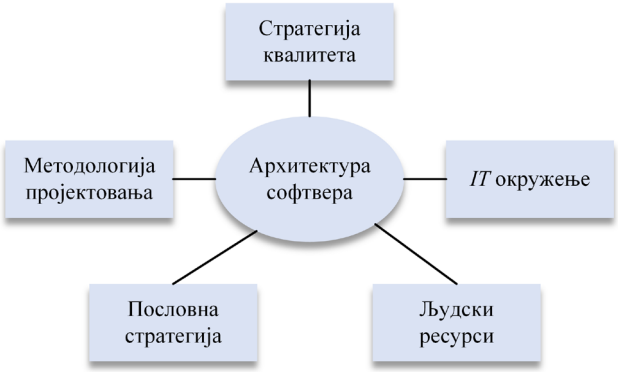
Контекст и улоге

Софвер који се пројектује најчешће је део ширег контекста информационог система предузећа. Приликом пројектовања потребно је јасно дефинисати границе софтверског система, стејкхолдере и њихове улоге. Типичи стејкхолдери и улоге су:

- *Acquirers* – надгледају набавку софтвера.
- *Assessors* – задужени за усаглашавање софтвера са стандардима и правним оквиром.
- *Communicators* – достављају релевантне информације другим стејкхолдерима (документација, обуке).
- *Architect* – архитекта софтвера – пројектује архитектуру у складу с пословним циљевима, креира и одржава план архитектуре софтвера.
- *Developers* – програмери – задужени за програмирање и постављање софтвера у продукционо окружење.
- *Testers* – задужени за тестирање система.
- *System Administrators* – задужени за управљање и одржавање софтвера у продукцији.
- *Suppliers* – добављачи хардвера, софтвера и инфраструктуре на којој се развијени софтвер извршава.
- *Support Staff* – корисничка подршка за крајње кориснике.
- *Users* – корисници дефинишу захтеве и користе софтвер.

18. ПРОЈЕКТОВАЊЕ АРХИТЕКТУРЕ СОФТВЕРА

Архитектура софтвера је апстракција којом се структура софтвера прилагођава конкретним захтевима свих стејкхолдера и условима у којима софтвер треба да ради. На архитектуру софтвера утиче више фактора: пословна стратегија, стратегија квалитета, људски ресурси, методологија пројектовања и *IT* окружење (Слика 29)[313].



Слика 29. Фактори који утичу на архитектуру софтвера[314]

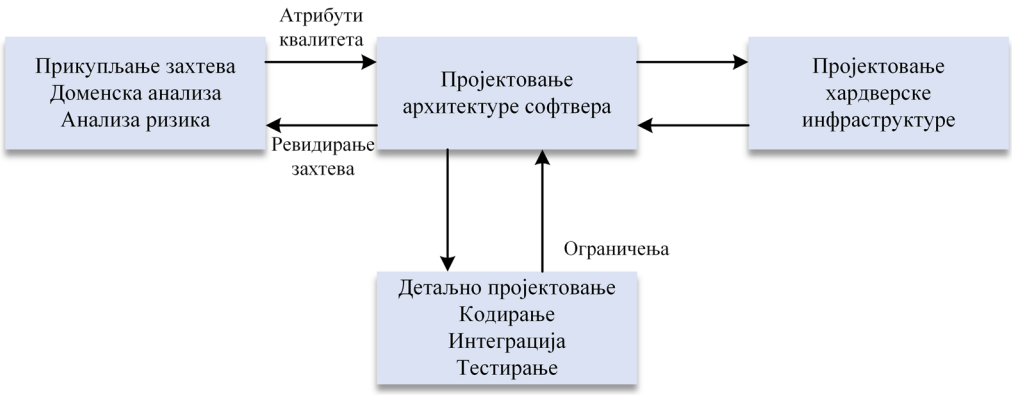
За пројектовање архитектуре софтвера потребно је разумевање организације предузећа и његовог пословног окружења. У пројектовање архитектуре софтвера укључен је већи број стејкхолдера и одлуке се доносе на нивоу целе организације. Добра архитектура софтвера смањује пословне ризике везане за техничко решење и премошћава разлике између пословних и техничких захтева.

У пројектовању архитектуре, фокус је на нефункционалним захтевима и дефинисању апстрактне структуре софтвера која те захтеве задовољава, врши спецификацију најважнијих компоненти, дефинише њихове везе и начине њихове интеракције. У пројектовању софтвера (*software design*), фокус је превасходно на функционалним захтевима и модулима, који се пројектују с више имплементационих детаља.

Резултат пројектовања архитектуре софтвера је модел који описује компоненте софтверског система и начин њиховог повезивања у јединствену целину. Обухвата спецификацију модула са одговарајућим апликационих програмским интерфејсима (*API*) преко којих модули међусобно комуницирају, и преко којих софтвер комуницира са окружењем.

Пројектовању архитектуре софтвера приступа се након иницијалног прикупљања захтева, доменске анализе и анализе ризика, а пре детаљног пројектовања функционалних захтева, кодирања, интеграције и тестирања (Слика 30).

[313] Wilt., J. (2012). File Pillars of IT architecture, https://iasaglobal.org/Public/Public/TOPICS/Iasa_Five_Pillars_Presentation.aspx
[314] Прилагођено према: https://www.tutorialspoint.com/software_architecture_design/introduction.htm



Слика 30. Пројектовање софтвера[315]

При пројектовању архитектуре, идентификују се пословни и технички захтеви стејкхолдера, особине софтвера, као и други нефункционални захтеви, попут дистрибуираности, скалабилности, поузданости, сигурности, флексибилности и квалитета. На основу њих се пројектује структура и понашање софтвера који те захтеве задовољавају.

Пројектовање архитектуре софтвера треба да обухвати: дефинисање функционалних захтева, где се дају одговори на питање шта софтвер треба да ради; дефинисање нефункционалних захтева, који обухватају дистрибуираност, скалабилност, поузданост, сигурност и атрибуте квалитета; дефинисање ограничења, попут анализе правног оквира, стандарда, трошкова и сл.

Архитектура се може пројектовати на различитим нивоима апстракције:

- **Архитектура информационог система предузећа (*Enterprise level*)**

  Односи се на пројектовање *IT* инфраструктуре предузећа која дугорочно треба да обезбеди оптималне перформансе и подршку за велики број апликација у оквиру предузећа. Пројектује се на основу анализе пословних потреба, тако да обезбеди да *IT* инфраструктура подржава пословне циљеве предузећа. Пројектовање се реализује у комуникацији са учесницима из целога предузећа.
- **Архитектура софтверских решења (*Solution level*)**

  Односи се на технолошка решења која могу решити конкретан пословни захтев. Фокус је на једној или неколико конкретних апликација. Ово је архитектура на нижем нивоу апстракције, која се пројектује у комуникацији више тимова унутар предузећа.
- **Архитектура апликације (*Application Level*)**

  То је архитектура на најнижем нивоу апстракције која се односи на једну конкретну апликацију. Детаљна је, и најчешће се пројектује у оквиру једног развојног тима.

[315] Прилагођено према: https://www.tutorialspoint.com/software_architecture_design/introduction.htm

Архитектура се може пројектовати и по различитим доменима:

- **Архитектура података (*Data level*)**

  Односи се на дефинисање процедура, политика, алата и модела које развојни тимови користе за креирање, организацију, складиштење и проналажење података у предузећу, као и за управљање миграцијама података из постојећих система.

- **Архитектура пословних процеса (*business process architecture*)**

  Односи се на колекцију пословних процеса и њихових веза, приказаних на високом нивоу апстракције. Користи се као основа за детаљано пројектовање пословних процеса. Неки од патерна архитектуре пословних процеса су: архитектура оријентисана на циљеве (*goal-based*), оријентисана на акције (*action-based*), оријентисана на пословне функције (*funcion-based*)[316] и друге.

- ***Cloud* архитектура (*cloud architecture*)**

  Односи се на пројектовање, управљање и функционисање *cloud* инфраструктуре предузећа.

Основни проблеми приликом пројектовања архитектуре односе се на недостатак адекватних алата и стандардизованих начина за приказ архитектуре, недостатак адекватних метода за анализу и предвиђање адекватности одабране архитектуре, лошу комуникацију између стејкхолдера и недовољно разумевање процеса пројектовања.

За пројектовање архитектуре софтвера задужен је архитекта софтвера. Ова улога захтева искуство у различитим областима пројектовања софтвера, укључујући: објектни и процесни приступ, вођење развојних тимова и координацију активности; доменско знање; технолошку експертизу, избор програмских језика, оквира, платформи, база података; методолошку експертизу, избор одговарајуће методологије развоја софтвера; вештине комунукације, управљање поверењем између тимова, дељење знања и заштита тима од ометајућих фактора.

У пројектовању архитектуре софтвера, полази се од пројектних захтева значајних за архитектуру софтвера (*Architecturally Significant Requirements – ASR*) који представљају комбинацију: главних функционалних захтева, главних захтева везаних за атрибуте квалитета, захтева везаних за животни циклус софтвера, и ограничења.

Поједностављени процес пројектовања архитектуре софтвера може се представити следећим корацима:

1. Анализирати захтеве и проверити да ли је прикупљено довољено информација о захтевима.
2. Одабрати делове система за декомпозицију.
3. Приоритизовати захтеве и идентификовати оне који су значајни за пројектовање архитектуре.

[316] Dijkman, R., Vanderfeesten, I. & Reijers, H. A. (2016). Business process architectures: overview, comparison and framework, *Enterprise Information Systems*, 10(2), 129–158.

4. Одабрати одговарајући архитектурни патерн, стил и приступ.
5. Дефинисати елементе архитектуре и алоцирати функционалности.
6. Дефинисати интерфејсе (*API*-је) за елементе архитектуре.
7. Анализирати, вредновати и верификовати постављену архитектуру кроз анализу почетних захтева и дефинисати ограничења за елементе архитектуре.
8. Понављати кораке 2–7 за следећи део система чија архитектура се пројектује.

За анализу и вредновање архитектуре софтвера могуће је користити различите технике: засноване на постављању питања (упитници, интервјуи, чек-листе итд.), засноване на мерењу параметара као што су кохезија, дубина наслеђивања и других, као и технике засноване на поређењу оригинално пројектованог и реализованог решења.

Једна од често коришћених техника за вредновање архитектуре софтвера позната је под скраћеницом *ATAM* (*Architecture Tradeoff Analysis Method*)[317], којом се процењују последице појединих одлука током пројектовања архитектуре, у контексту захтева везаних за атрибуте квалитета и пословне циљеве. Потенцијално проблематична одлука сматра се ризиком, а елементи архитектуре који значајно утичу на поједине атрибуте квалитета називају се тачкама осетљивости. *Tradeoff* је елемент који представља тачку осетљивости за више атрибута квалитета, који су често противречни. Лоше одлуке у вези с пројектовањем архитектуре доприносе повећању тзв. техничког дуга (*technical debt*), тј. трошкова који ће настати у будућности да би се исправиле грешке које су последица садашњих одлука[318].

19. ПАТЕРНИ АРХИТЕКТУРЕ СОФТВЕРА

Патерни се могу посматрати као метамодели архитектуре софтвера. Архитектура софтвера се гради комбинацијом патерна. Основне идеје патерна су модуларност и наслеђивање особина патерна, његовом применом.

Идеје патерна, као концепта модуларности и основног градивног елемента у дефинисању архитектуре софтвера, везују се за радове Кристофера Александера[319]. Иако се Александер бавио архитектуром и патернима у грађевинарству, његови радови су често цитирани као основна литература за патерне у софтверском инжењерству. Многи термини, као што су силе (*forces*), патерни (*patterns*), језици патерна (*pattern languages*), и други, преузети су управо од њега. Александер дефинише патерн као модел који успоставља релације између: проблема, решења и контекста.

[317] Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H. & Carriere, J. (1998, August). The architecture tradeoff analysis method. In: *Proceedings. fourth IEEE international conference on engineering of complex computer systems* (cat. no. 98ex193) (pp. 68–78). IEEE.
[318] Kruchten, P., Nord, R. L. & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice, *IEEE software*, 29(6), 18–21.
[319] Alexander, C. (1977). *A pattern language: towns, buildings, construction*. Oxford university press.

Архитектура софтверског система је хијерархијска. За опис на највишем нивоу користе се патерни архитектуре, док су се на нижим нивоима апстракције користили патерни пројектовања софтвера (*design patterns*)[320]. Патерни пројектовања софтвера повезују се са објектно-оријентисаном парадигмом развоја софтвера, првенствено из ралога што се као главна литература дуго користила књига *Design Patterns*[321] четворице аутора познатих као *Gang of four – GoF*. Патерни пројектовања у овој књизи обухватили су 23 софтверска патерна, груписана у: креационе, структурне и патерне понашања. Упркос великој популарности, бројне критике односиле су се на чињеницу да су предложени патерни највећим делом решавали недостатке објектно-оријентисаног језика *C++*[322]. У реалности, критике су се односиле на недостатке комплетног објектно-оријентисаног приступа развоју софтвера који није имао уграђене концепте за квалитетну имплементацију нефункционалних захтева. За квалитетну имплементацију нефункционалних захтева био је потребан искорак ка патернима архитектуре софтвера. Објектно-оријентисани приступ, а ни *UML* нису имали концепте који би тај искорак подржали.

Као модел решавања проблема у дефинисању архитектуре софтвера, патерни би требало да буду универзални. Имајућу у виду Александерову дефиницију, као и приступе других аутора[323,324], софтверски патерни који се данас користе превазилазе оквире појединачних парадигми развоја софтвера, а често подразумевају и мешање концепата различитих парадигми, на пример, објектне и процесне: објекти, нити, сервиси и догађаји.

У развоју савременог софтвера оријентисаног на процесе, граница између патерна архитектуре и патерна пројектовања софтвера је избрисана. Пројектовање софтвера обухвата и почиње пројектовањем архитектуре.

Кроз праксу, током година искуства, издвојио се мањи скуп најчешће коришћених патерна архитектуре софтвера за које су развијени софтверски оквири, платформе и алати, тако да избор одговарајуће архитектуре софтвера у великој мери одређује и избор платформе и алата који ће се користити. И обратно, уколико је у корисничким захтевима унапред дефинисана софтверска платформа, она ће у великој мери одредити архитектуру будућег софтвера.

---

[320] Влајић, С. (2014). *Софтверски патерни*, Златни пресек, Београд.

[321] Gamma, E., Helm, R., Johnson, R., Vlissides, J. & Patterns, D. (1995). *Design Patterns Elements of Reusable Object-Oriented Software*. Addison Wesly.

[322] Norvig, P. (1996). *Design Patterns in Dynamic Programming*, http://www.norvig.com/design-patterns/design-patterns.pdf

[323] Vlajić, S., Stanojević, V., Savić, D., Milić, M., Antović, I. & Lazarević S. (2016). The General Form of GoF Design Patterns, *The World of Computer Science and Information Technology Journal (WSCIT)*, Volume 6, Issue 2. pp. 12–20, https://www.researchgate.net/publication/305608959_The_General_Form_of_GoF_Design_Patterns

[324] Beck, K., Crocker, R., Meszaros, G., Coplien, J. O., Dominick, L., Paulisch, F. & Vlissides, J. (1996, March). Industrial experience with design patterns. In: *Proceedings of IEEE 18th International Conference on Software Engineering* (pp. 103–114). IEEE. https://web.eecs.umich.edu/~weimerw/481/readings/design.pdf

Код предузећа која су почела развој софтвера без формалног пројектовања архитектуре, дешавали су се проблеми који су резултовали честим прекидима у раду софтвера[325]. Проблеми се тичу недостатка као што су: проблеми проширења функционалности, скалабилности, сигурности, јасно дефинисаних улога, одговорности, и веза између њих. Пројектовање архитектуре софтвера, засновано на архитектурним патернима, показало се као добар путоказ у избегавању проблема и развоју квалитетног софтвера.

Архитектурни патерни представљају концепте модуларности и интеграције, који су настали на примерима добре праксе и систематизацији метода избегавања негативног искуства из проблематичних пројеката. Планирање и одабир одговарајуће архитектуре, коришћењем патерна, омогућује проактивно управљање ризиком, боље управљање перформансама и смањење утицаја лоше пројектованог софтвера на пословне процесе.

Патерни архитектуре представљају оквир који пројектантима омогућује да креирају основну шему структуре софтверског система на проверен и сигуран начин. Користе се када се пројектује пословна логика. Њима се дефинишу компоненте софтвера, улоге стејкхолдера и одговорности.

Конкретан архитектурни патерн је груба слика архитектуре, која се у току пројектовања прилагођава и разрађује за конкретан пројекат. Примена патерна омогућава дефинисање потребних карактеристика софтвера, укључујући снаге и слабости које сваки патерн има, квалитет, ефикасност, већу продуктивност, агилност у развоју и лакше решавање проблема у експлоатацији софтвера, на начин проверен у примерима најбоље праксе.

Архитектура реалног софтвера се у већини случајева састоји од комбинације више патерна. У наредним поглављима описани су поједини патерни софтверских архитектура који имају широку примену и нису ограничени парадигмом развоја софтвера[326]. Посебна пажња и главни део ће бити посвећен архитектурама које су специфичне за процесно-оријентисани софтвер.

19.1 Антипатерни

Значајан фактор који утиче на избор архитектуре софтвера су захтеване или очекиване перформансе софтверског система у реалним условима рада. Стуктура софтверског система зависности између компоненти и специфичности њиховог постављања у продукционо окружење, од којих у највећој мери зависе перформансе, најчешће нису познате, или нису познате у потпуности у време развоја софтвера. Да би се проблеми с перформансама система ублажили или

[325] Dhaduk, H. (2020). 10 Software Architecture Patterns You Must Know About, https://www.simform.com/blog/software-architecture-patterns/

[326] Richards, Mark. *Software architecture patterns*. Vol. 4. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Incorporated, 2015.

избегли, нужно је о њима макар и начелно размишљати и узети их у обзир при дефинисању архитектуре софтвера. Из тих разлога су у примерима најбоље праксе идентификоване и систематизоване честе грешке које чине пројектанти при дефинисању архитектуре софтвера, а које у одговарајућим околностима могу негативно утицати на перформансе софтвера[327]. Те грешке су назване антипатерни. Антипатерни су специфични патерни који имају своју намену и своје добре особине, али својим особинама у неким реалним радним условима утичу на лошу расподелу оптерећења, појаву уских грла и непотребног посла, повећање времена обраде захтева током времена, трешинг итд.[328] Код спецификације антипатерна обавезно се наводе услови у којима примена патерна даје лоше резултате. При пројектовању архитектуре софтвера, нужно је елементе пројектоване архитектуре анализирати на евентуално постојање антипатерна за очекиване услове експлоатације[329].

19.2 Архитектурни стилови у пројектовању софтвера

Архитектурни стил представља скуп препорука и ограничења при пројектовању софтвера, који има за циљ да дефинише архитектуру софтвера с карактеристикама типичним за решавање одређене класе проблема. Примери архитектурних стилова су *REST*, Микросервиси и Доменом вођено пројектовање софтвера.

И архитектурни стил и софтверски патерн дефинишу архитектуру софтвера која је погодна за решавање одређене класе проблема, с тежњом и циљем да постану широкоприхваћен или стандардизован начин њиховог решавања. Разлика је у томе што архитектурни стил дефинише архитектуру имплицитно, коришћењем препорука и ограничења, док софтверски патерн дефинише архитектуру експлицитно, спецификацијом компоненти архитектуре и њихових међусобних веза. У оба случаја је излаз исти: конкретан модел архитектуре. Из тих разлога се у литератури оба ова појма мешају. Неки аутори и пројектанти (нпр. *Мајкрософт* у неким упутствима за коришћење софтвера) користе искључиво темин „архитектурни“ стил, док други користе искључиво термин „софтверски патерн“. У оквиру ове књиге јасно ће се нагласити случајеви када је архитектура дефинисана архитектурним стилом, а када патерном.

19.3 Таксономија патерна и архитектурних стилова

Патерни архитектуре софтвера и архитектурни стилови намењени су решавању специфичних проблема при пројектовању архитектуре софтвера. По областима и

[327] Brunnert, A., van Hoorn, A., Willnecker, F., Danciu, A., Hasselbring, W., Heger, C., ... & Wert, A. (2015). Performance-oriented DevOps: A research agenda. arXiv preprint arXiv:1508.04752. https://arxiv.org/pdf/1508.04752.pdf

[328] Smith, C. U. & Williams, L. G. (2002, December). New software performance antipatterns: More ways to shoot yourself in the foot. *Int. CMG Conference* (pp. 667–674).

[329] Palomba, F., De Lucia, A., Bavota, G. & Oliveto, R. (2014). Anti-pattern detection: Methods, challenges, and open issues. *Advances in Computers* (Vol. 95, pp. 201–238). Elsevier.

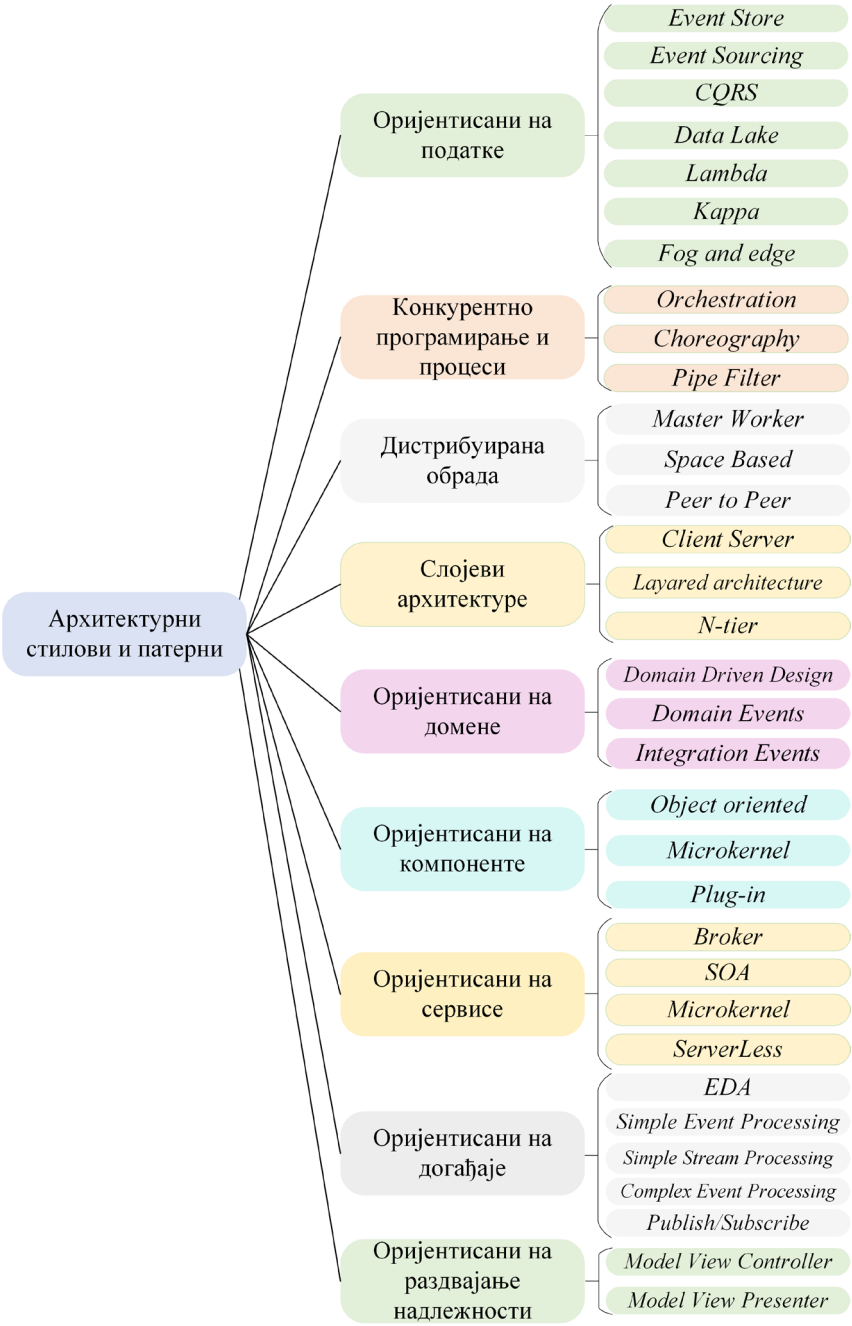
класама проблема које решавају, патерни и архитектурни стилови могу се поделити на (Слика 31):

1. Оријентисане на податке (*Data Centric*);
   Event Store; *Event Sourcing*; *CQRS*; *Data Lake*; *Lambda*; *Kappa*; *Fog and edge*.
2. Конкурентно програмирање и процесе (*Concurrent Programing*);
   Orchestration; *Choreography*; *Pipe Filter*.
3. Дистрибуирану обраду (*Distributed Systems*);
   Master Worker; *Space Based*; *Peer to Peer*.
4. Слојеве архитектуре (*Layered*);
   Client Server; *Layered architecture*, *n-tier*.
5. Оријентисане на домене (*Domain Driven*);
   Domain Driven Design, *Domain Events*, *Integration Events*.
6. Оријентисане на компоненте (*Component Based*);
   Object oriented; *Microkernel*; *Plug-in*.
7. Оријентисане на сервисе (*Service Based*);
   Broker; *SOA*; *Microkernel*; *Serverless*.
8. Оријентисане на догађаје (*Event Based*);
   EDA; *Simple Event Processing*; *Event Stream Processing*; *Complex Event processing*; *Publish/Subscribe*.
9. Оријентисане на раздвајање надлежности (*Separation of Concern*);
   Model View Controller MVC; *Model View Presenter MVP*.

Поред поделе по областима примене, за потребе излагања у овој књизи, патерне архитектуре и архитектурне стилове који су дати на Слици 31 можемо сврстати у четири групе:

1. Основни архитектурни патерни који служе за имплементацију елементарних функционалних и нефункционалних захтева. Могу бити саставни део сложенијих патерна.
2. Патерни корисничких интерфејса.
3. Патерни интеграције.
4. Патерни архитектуре и архитектурни стилови који су специфични само за пројектовање софтвера оријентисаног на процесе који обухватају специфичне патерне и архитектурне стилове за складиштење и обраду догађаја. Патерни ове врсте спадају и у категорију патерна интеграције, јер користе апстракцију агрегације догађаја. Често се имплеметирају у форми софтверских платформи и екосистема.

При пројектовању софтвера оријентисаног на процесе, пројектовани модел архитектуре софтвера може обухватити међусобно повезане патерне из све четири наведене групе.



Слика 31. Графички приказ таксономије архитектурних патерна и стилова[330]

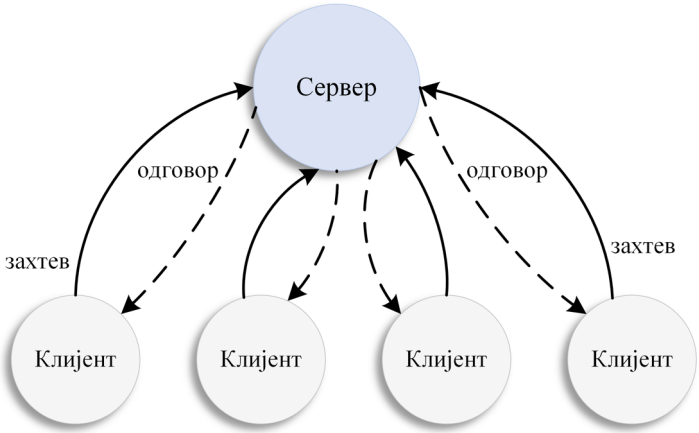
[330] Прерађено према: Sharma, L. (2023) The Architect's Blueprint: Understanding Software Styles and Patterns with Cheatsheet, https://medium.com/bytebytego-system-design-alliance/the-architects-blueprint-understanding-software-styles-and-patterns-with-cheatsheet-5c1f5fd55bbd

19.4 Основни архитектурни патерни

19.4.1 Патерн клијент-сервер архитектуре

Клијент-сервер архитектура[331] примењује се у развоју дистрибуираних апликација које захтевају повећану скалабилност. Састоји се од две компоненте: клијента и сервера, повезаних у мрежу. Комуникација се одвија тако што клијент захтева ресурс са сервера, сервер потом проналази тражени ресурс и шаље одговор клијенту (Слика 32). Основна карактеристика сервера је способност пружања одређене услуге (сервиса) или функционалности клијентима. Начин рада сервера је сакривен од клијената. Један сервер може опслуживати већи број клијената, а клијент може користити сервисе различитих сервера.

Клијент-сервер архитектура је настала из потребе да се повећа скалабилност обраде тако што ће се апликација раздвојити на клијентски и серверски део. Скалабилност обраде на клијентском делу апликације се повећава с бројем клијената, јер сваки клијент користи своје ресурсе који не утичу на друге клијенте. Скалабилност серверског дела апликације се повећава с бројем сервера којима клијент приступа. У овом моделу архитектуре, иако је скалабилност обраде повећана, нарочито на клијентској страни, нема ефикасног начина за постизање линеарне скалабилности серверског дела апликације.



Слика 32. Патерн клијент-сервер архитектуре

Типичне употребе овог патерна су апликације као што су имејл, сервиси електронског банкарства, веб, сервиси за дељене фајлова, онлајн игре и слично. Главни недостатак је што у случају отказа сервера цео систем престаје да ради (*single point of failure*). Сервер може представљати уско грло, може бити сложен за одржавање, а свака промена архитектуре је комплексна.

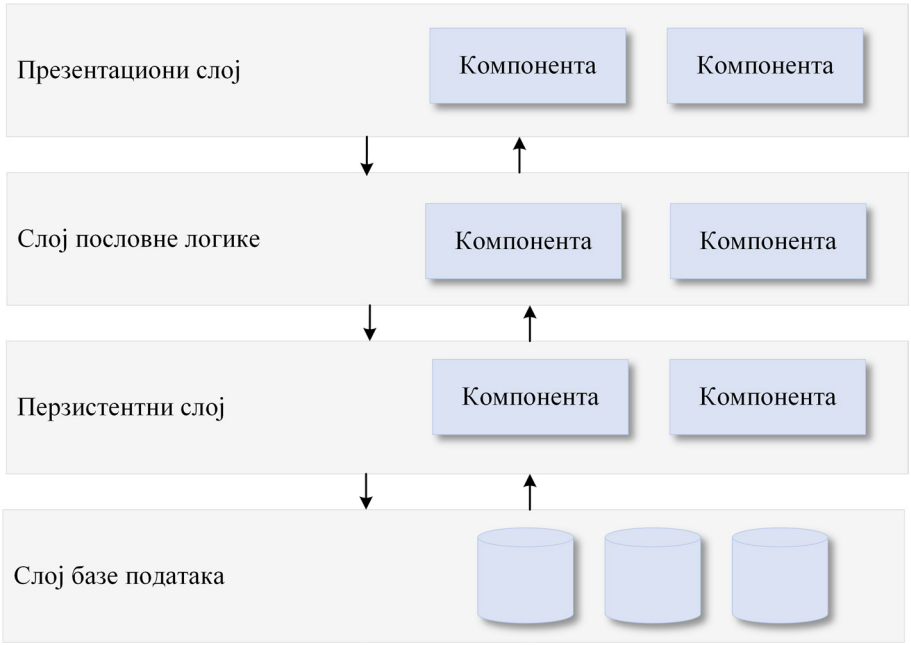
[331] Berson, A. (1996). *Client/server architecture*. McGraw-Hill, Inc.

19.4.2 Патерни вишеслојне архитектуре

Патерн вишеслојне (*n-tier*) архитектуре је широко прихваћен међу архитектама и пројектантима софтвера. Патерн не дефинише стриктно број и назив слојева, па се у пракси јавља у више варијанти. Најпознатија је трослојна архитектура, која обухвата слојеве презентације, пословне логике и слој података. Сваки слој има јасно дефинисану улогу и може комуницирати само са суседним слојевима (Слика 33). Слојеви су изоловани, па се компоненте унутар сваког слоја могу независно мењати. Користи се код апликација које треба брзо развити, а које захтевају стриктно поштовање стандарда за одржавање и тестирање. Погодан је за апликације у предузећима с традиционалном организацијом *IT* сектора, као и у тимовима мање искусних програмера.

Овај патерн доприноси умереном повећању скалабилности апликације. Уколико се презентациони слој имплементира на клијентима, програми презентационог слоја с повећањем броја клијената имаће знатно повећање скалабилности. Додатно повећање скалабилности доноси имплементација сваког слоја на посебном серверу.

Главни недостатак овог патерна је што чак и мале промене у кôду могу захтевати поновно постављање комплетне апликације у продукционо окружење. Популарни софтверски оквири, као што је *Java EE*, користе овај патерн софтверске архитектуре.

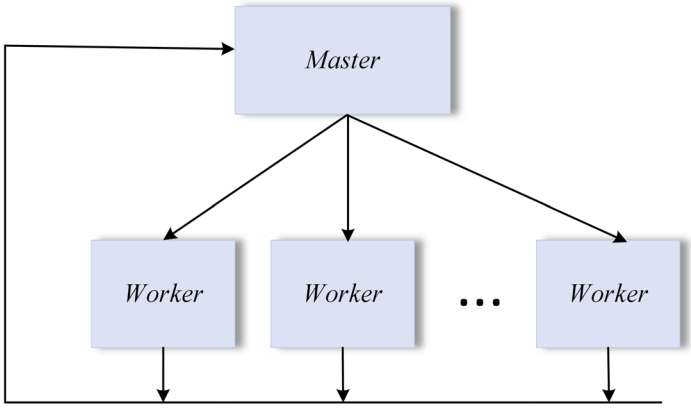


Слика 33. Вишеслојна архитектура

19.4.3 Патерн master-worker

Овај архитектурни патерн у литератури је познат и као *Master-Slave*. Из разлога политичке коректности[332], последњих година се за овај патерн користи назив *Leader-Follower*. Настао је као покушај да се креира архитектура која повећава скалабилност обраде серверског сегмента у клијент-сервер архитектури.

Када једна компонента клијент-сервер архитектуре прима велики број захтева – на пример, база података – може се догодити да није у могућности да их све обради, што може довести до успорења или застоја у раду система. Могуће решење за повећање скалабилности је примена патерна *master-worker* архитектуре, која омогућује да мастер компонента система распореди захтеве на више *worker* компоненти (Слика 34). Када *worker* компонента обради захтев, резултат се враћа до мастера, интегрише се с резултатима обраде на осталим *worker* компонентама и формира се одговор за клијента. Улога мастер сервера је да контролише рад *worker* компоненти, управља комуникацијом и приоритетима. Специфично за овај патерн је што *worker* компоненте истовремено обрађују захтеве, потом резултате обраде шаљу мастеру, па је захтев обрађен тек када све *worker* компоненте, којима је мастер упутио захтев, заврше обраду и пошаљу резултат.



Слика 34. Патерн master-worker архитектуре

Овај патерн погодан је за апликације код којих се задаци могу поделити у мање сегменте који се паралелно могу извршити. Примери коришћења су оперативни системи с мултипроцесорским архитектурама, базе података, апликације које обрађују сирове податке у дистрибуираним системима, и други. Користи се и у другим архитектурним патернима, када је потребно повећати скалабилност пословне логике. Једна од најзначајнијих примена овог патерна је у *map-reduce* алгоритму за обраду великих података[333].

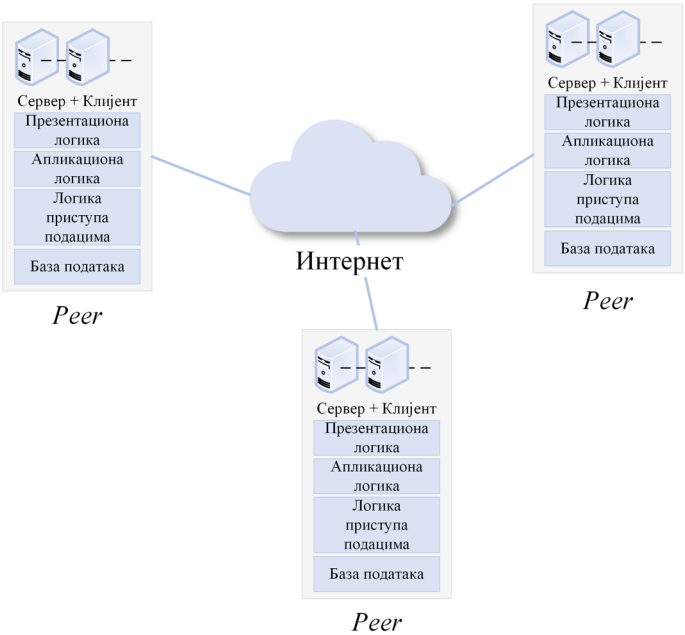
[332] https://www.bbc.com/news/technology-53050955
[333] Dean, J. (2006). *Experiences with MapReduce, an abstraction for large-scale computation*. Google Inc. https://storage.googleapis.com/pub-tools-public-publication-data/pdf/32721.pdf

Главни недостатак је што грешка на мастеру може довести до губитка података. Такође, трошкови имплементације могу бити високи, због већег броја *worker* компоненти које се морају конфигурисати за вршно оптерећење.

19.4.4 Патерн peer-to-peer архитектуре

У *peer-to-peer* архитектури, свака компонента архитектуре (*peer*) равноправна је и може се понашати као клијент, сервер, као оба истовремено или може мењати улогу у реалном времену (Слика 35). Као клијент, *peer* може тражити сервис од другог *peer*-а који тада преузима улогу сервера. Као сервис, *peer* ће нудити сервисе другим *peer*-овима, прихватати и обрађивати њихове захтеве. Главна разлика у односу на традиционалну клијент-сервер архитектуру је што нема централизованог сервера који прима захтеве клијената, већ сваки умрежени рачунар може бити сервер. Поједини аутори чак описују *peer-to-peer* архитектуру као супротност клијент-сервер архитектури[334].

Примери *peer-to-peer* архитектуре су мрежне апликације за дељење фајлова (*BitTorrent*), блокчејн мреже, поједине чет апликације, и други. Главни недостатак је немогућност гарантовања квалитета сервиса, перформансе зависе од броја доступних *peer*-ова у мрежи, нема могућности бекапа.



Слика 35. Патерн peer-to-peer архитектуре

[334] Schollmeier, R. (2001, August). A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In: *Proceedings first international conference on peer-to-peer computing* (pp. 101–102). IEEE.

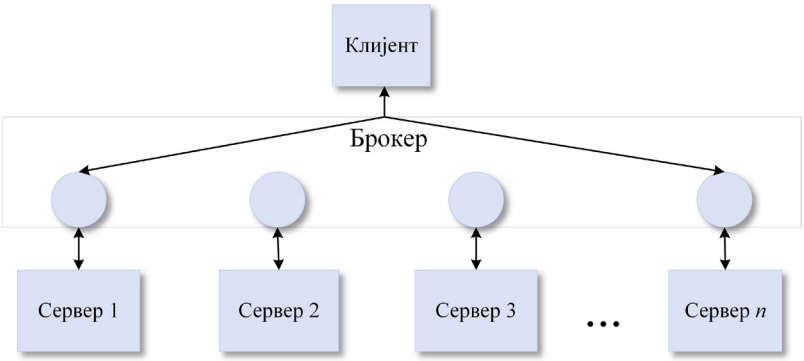
19.4.5 Патерн посредника – брокер архитектура

Брокер архитектура представља проширење клијент-сервер архитектуре. Брокер је посредник између клијента и сервера (Слика 36). Главна му је намена одвајање клијентске од серверске стране и апстракција архитектуре серверске стране. Користи се за структурирање дистрибуираних система са слабо повезаним компонентама, код којих клијент у моменту када од брокера затражи услугу сервера, нема податке о конкретном серверу који ће извршити тражену услугу. С друге стране, када сервер од брокера добије захтев, нема податке за којег клијента врши обраду.

Брокер патерн садржи три компоненте: клијент, сервер и брокер. Брокер је задужен за интеракцију и координацију с компонентама на серверу. Када клијент захтева сервис од брокера, брокер преусмерава захтев до одговарајуће компоненте на серверу. Комуникација брокера са серверским компонентама је независна од клијената и одвојена је од пословне логике, што доприноси већој флексибилности у деплојменту.

Увођењем брокера као проширења клијент-сервер архитектуре постиже се већа флексибилност система, повећава се склабилност и смањује међузависност клијента и сервера. Често се примењује и као проширење микросервисне архитектуре. Применом ове архитектуре могуће је једноставно и динамичко управљање основним операцијама над разним и хетерогеним изворима података (креирање, ажурирање, брисање, претрага).

Брокер патерн се често користи у софтверским решењима за управљање токовима података, као и у дистрибуираним системима најразличитијих намена. Неки од недостатака укључују лошу отпорност на отказе, мање перформансе због додатних слојева, већу латенцију.



Слика 36. Патерн брокер архитектуре[335]

[335] Прилагођено према: Márquez, G. & Astudillo, H. (2018, December). Actual use of architectural patterns in microservices-based open source projects. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)* (pp. 31–40). IEEE.

19.4.6 Патерн виртуелизованог средњег слоја

Патерн виртуелизованог средњег слоја (*Virtualized middleware*) обезбеђује скалабилну и поуздану дистрибуирану обраду података. Алтернативна имена овог патерна су: *Cloud-Based Architecture* и *Space-based Architecture*[336]. Назив *Space-based Architecture* је настао по *tuple space*, једном од познатих метода за имплементацију дистрибуираних система. *Tuple space* апстракција је погодна за представљање енкапсулације у форми модула који садржи функционалности са јасно дефинисаним интерфејсима, а која се у овом архтектурном патерну користи за поделу апликације на модуле и њихову расподелу по процесним јединицама.

Архитектура система се састоји од скупа повезаних дистрибуираних компоненти за обраду података, који се називају процесним јединицама, и виртуелизованог средњег слоја (*Virtualized Middleware*) који, с једне стране комуницира с клијентима, а с друге с процесним јединицама (Слика 37). Процесне јединице садрже апликације подељене на „сегменте простора“ или партиције, које могу чинити делови база података и апликација. Сви делови подељеног „простора“ заједно чине целину апликације. Поделом апликације у делове постиже се дистрибуираност и већа склабилност. Делови података и програма који се налазе на процесним јединицама треба да буду физички дистрибуирани на местима генерисања, обраде и складиштења података. Тиме се смањује обим комуникације и повећава брзина обраде[337].

Виртуелизован средњи слој води рачуна о поузданости система, скалабилности и управља редундантношћу. Процесним јединицама обезбеђује инфраструктуру с више паралелних канала за размену порука (*Message Grid*), инфраструктуру за дистрибуиране базе података и сесије (*Data Grid*) и инфраструктуру паралелелне обраде по *Master-Worker* моделу (*Processing Grid*). Виртуелизовани средњи слој најчешће преко једног или више апликационих програмских интерфејса корисницима нуди монолитну слику виртуелизоване апликације која може бити скалабилна и поуздана.

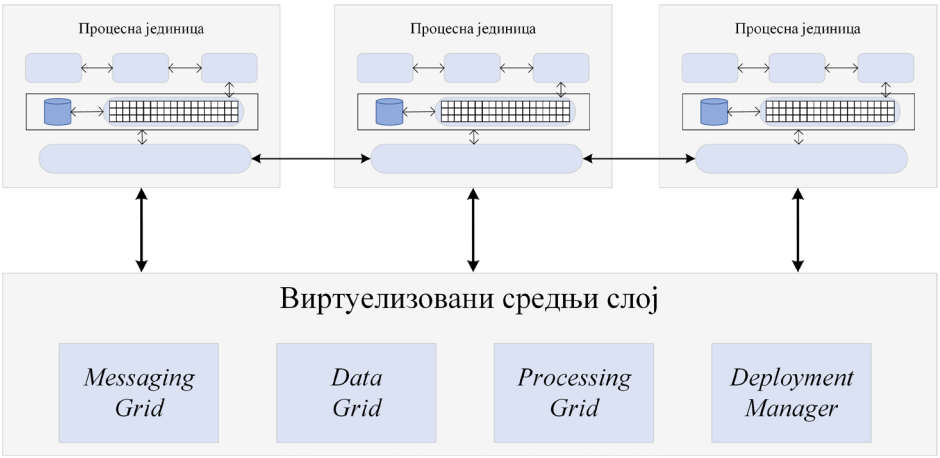
Процесна јединица садржи делове апликације, најчешће пословну логику. У мањим апликацијама, комплетна пословна логика може бити у једној процесној јединици, док се код већих, апликација може поделити на више процесних јединица. Процесних јединица исте врсте може бити више, у зависности од захтеване скалабилности и поузданости. Виртуелизован средњи слој садржи компоненте за праћење сесија корисника, контролу, синхронизацију података и обраду захтева.

За примену овог архитектурног патерна најпогодније је да се процесна јединица реализује у форми скупа микросервиса, који немају стања за која је потребно пратити сесију, и код којих је база података локална за микросервис.

---

[336] Richards, M. (2015). *Software architecture patterns*. O'Reilly Media Inc.
[337] Khan, A. A., & Le, D. N. (Eds.). (2022). *Evolving software processes: trends and future directions*. John Wiley & Sons.

Типични примери употребе овог патерна су софтверски системи који имају много корисника и велики број захтева, тј. апликације код којих је важна скалабилност и конкурентност обраде. Користи се и за имплементацију платформе као сервис (*Platform as a Service*) рачунарства у облаку.



Слика 37. Патерн виртуелизованог средњег слоја[338]

19.4.7 Патерн филтрирања тока података

Патерн филтрирања тока података (*filter-pipe*) користи се за обраду усмерених токова велике количине података. Компоненте за обраду података, у овом патерну се називају филтери, а пренос података од улаза ка излазу се назива током података (Слика 38).

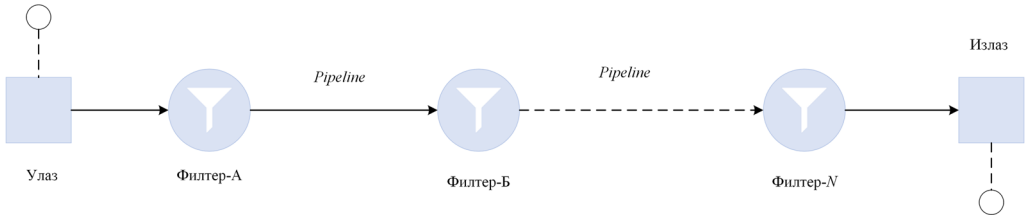
Ток података се на местима обраде „филтрира“ и резултат прослеђује комуникационим каналима до улаза наредног филтра[339].

Овај патерн се често користи за међупроцесну комуникацију у вишепрограмским и дистрибуираним рачунарским системима. Погодан је за примену у апликацијама у којима се обрађују велике количине генерисаних података, као што су обрада резултата мерења са *IoT* сензора и обрада догађаја у *ClickStream* аналитици. Често се користи за електронску размену података у појединим функционалностима оперативних система (на пример, *pipe* у *Linux* оперативном систему повезује излаз једне са улазом друге апликације).

Pipe омогућава пренос података између апликација, са апстрахованим проблемима конкурентног програмирања, као што су критични регион и сигнализација пуног

[338] Прилагођено према: Richards, M. (2015). *Software architecture patterns*. O'Reilly Media Inc.
[339] Monroe, R. T., Kompanek, A. J., Melton, R. & Garlan, D. (1996). Stylized architecture, design patterns, and objects, In *Architectural Styles, Design Paterns, and Objects*, Carnegie Mellon University Research Showcase.

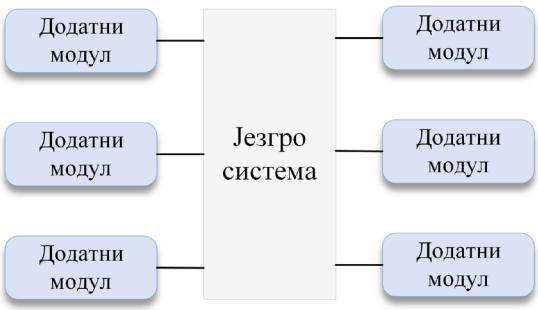
и празног бафера. Повећање скалабилности се постиже расподелом укупне обраде на појединачне „филтере“. Главни недостаци примене овог патерна везани су за могућност губитка података између филтера и брзине обраде целог система која зависи од брзине обраде најспоријег филтра у низу.



Слика 38. Патерн филтрирања тока података (filter-pipe)

19.4.8 Патерн архитектуре у форми микрокернела

Патерн архитектуре у форми микрокернела највише се користи у пројектовању објектно-оријентисаног софтвера. Састоји се од два врсте компоненти: централног дела, језгра система (*core*) и додатних модула (*plug-in*). Централни део, језгро система, налик је микрокернелу у оперативним системима[340], по коме је и добио назив. Задужен је за минимални скуп функционалности потребан да софтвер функционише. Додатни модули се користе као независни додаци за проширења и различите намене. У контексту пословних апликација, централни део система чини имплементација општих правила пословне логике, без специфичних случајева коришћења, правила или услова, док би се преко модула додавале функционалности за специфичне потребе и услове коришћења. Повезивање језгра и модула обавља се помоћу апликационих програмских интерфејса. Овај патерн архитектуре илустрован је на Слици 39.



Слика 39. Патерн архитектуре микрокернела

Овај патерн погодан је за апликације код којих се јасно могу раздвојити основне пословне рутине од додатних правила, или фиксни послови од оних који се често мењају. С друге стране, због постојања већег броја модула који од њега зависе,

[340] Büttner, E. & Richter, S. (2004). Microkernel–An Architecture Pattern. *Technische Berichte*, 18.

овај патерн не дозвољава једноставне промене у језгру. Велики изазов код овог патерна је одабир одговарајућег нивоа грануларности између кернела и модула.

19.5 Патерни пројектовања корисничког интерфејса

Процесни приступ, као екстензија објектно-оријентисаног програмирања, широко је примењиван у пројектовању корисничких интерфејса, већ од самог почетка увођења објектно-оријентисане парадигме. У оквиру корисничког интерфејса, а нарочито у виртуелној реалности, од подједнаке важности је моделовање структуре, као и моделовање динамике, па је сасвим оправдано коришћење обе парадиме у пројектовању корисничких интерфејса[341].

Примена патерна базираног на догађајима у пројектовању корисничких интерфејса није нужно везана за одређену архитектуру софтвера. Може се успешно имплементирати и у монолитним архитектурама. У вишеслојној архитектури, презентациони слој на коме се налази кориснички интерфејс, имплементира се коришћењем клијент-сервер архитектуре. Сервер обезбеђује приступ бази података презентационог слоја, инфраструктуру за обраду догађаја, размену порука и праћење клијентских сесија. Пожељно је да се што више обраде пренесе на клијентску страну (*Client Side Rendering*). Уколико припрема и графичка обрада страница корисничког интерфејса превазилази капацитет клијената, она се извршава на серверској страни (*Server Side Rendering*)[342].

Извршењем презентационог слоја апликације на клијенту, најчешће преко *AJAX*[343] технологија, обрада се у великој мери преноси са сервера на клијентски рачунар, на коме кориснички интерфејс и апликације презентационог слоја употребљава један корисник. Тиме се, поред једнокорисничког режима рада и избегавања проблема конкурентног програмирања, паралелизацијом рада више клијената на различитим рачунарима постиже повећање укупне скалабилности.

За моделовање структуре презентационог слоја користе се објекти са атрибутима из објектног модела, док се за моделовање динамике користе догађаји, микросервиси, нити и процеси. Комуникација између процеса одвија се разменом порука преко *Pub/Sub* патерна. Нека објектно-оријентисана окружења за развој корисничких интерфејса, уместо *Pub/Sub* патерна користе *Dependency Injection* патерн.

Имплементација процесног приступа у корисничким интерфејсима захтева

---

[341] Fiala, Z., Frasincar, F., Hinz, M., Houben, G. J., Barna, P. & Meissner, K. (2004). Engineering the presentation layer of adaptable web information systems. In: *Web Engineering:4th International Conference, ICWE 2004*, Munich, Germany, July 26–30, 2004. Proceedings 4 (pp. 459–472). Springer Berlin Heidelberg.

[342] Iskandar, T. F., Lubis, M., Kusumasari, T. F. & Lubis, A. R. (2020, May). Comparison between client-side and server-side rendering in the web development. In: *IOP Conference Series: Materials Science and Engineering* (Vol. 801, No. 1, p. 012136). IOP Publishing. https://iopscience.iop.org/article/10.1088/1757-899X/801/1/012136/pdf

[343] Garrett, J. J. (2005). *Ajax: A new approach to web applications*. MIT.

имплементацију система за управљање догађајима, базу података о догађајима и систем за размену порука. То су захтевне софтверске компоненте, које није упутно сваки пут појединачно развијати, већ се за имплементацију користе посебно развијени и прилагођени софтверски оквири и платформе. Најпознатији су *React*[344] и *Angular*[345].

За пројектовање корисничких интерфејса, дуго година се искључиво користио *MVC* (*Model View Controller*) патерн. Овај патерн у основној верзији има много недостатака, па је током година у значајној мери мењан и дограђиван. Последњих година се полако напушта и уступа место патернима који су из њега произашли, а који успешније решавају проблеме презентационог слоја.

19.5.1 Model View Controller

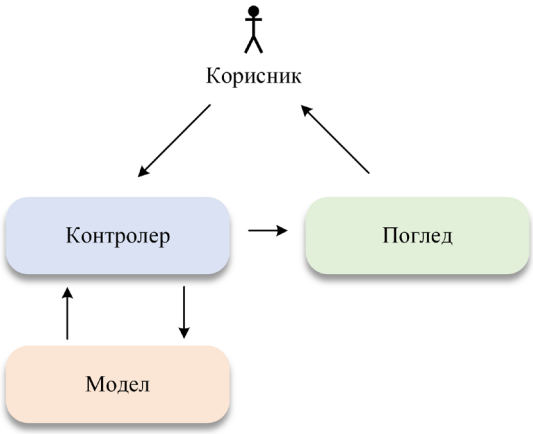
Основна идеја *MVC* патерна, као и свих његових деривата, јесте раздвајање надлежности (*separation of concern*) компоненти корисничког интерфејса. Од степена и квалитета раздвајања надлежности, у највећој мери зависи и његова употребна вредност. *MVC* патерн[346] обраду у презентационом слоју дели у три повезана сегмента (Слика 40):

- Модел (*model*), садржи базу података презентационог слоја на основу које се врши презентација података и интеракција с корисником.
- Поглед (*view*), у складу с презентационом логиком врши претраживање и филтрирање података из модела, које припрема за приказивање, прилагођава улазно-излазном уређају и приказује кориснику.
- Контролер (*controller*), прихвата акције корисника и у складу с презентационом логиком врши ажурирање погледа и модела.

Од првог објављивања[347], пре више од четири деценије, *MVC* патерн је у многим радовима критикован, мењан и допуњаван. Почетне верзије овог патерна нису водиле рачуна о стањима корисничког интерфејса – оригинални *MVC* патерн је *stateless*. Такође, није било подршке за догађаје, па се свака акција корисника морала обрађивати посебним позивима. У каснијим модификацијама, *MVC* патерн је допуњен подршком за догађаје и асинхроне процесе, као и обсервер патерном који води рачуна о усклађивању стања погледа и модела. Највећи недостатак овог патерна је лоше дефинисано раздвајање надлежности. Презентациона логика је подељена између погледа и контролера, што отежава пројектовање и тестирање[348,349].

---

[344] https://react.dev/
[345] https://angular.io/
[346] Reenskaug, T. (2003). *The Model-View-Controller (MVC) Its Past and Present*, University of Oslo. http://nandgatetech.com/files/ComputerHistory/MVC_pattern.pdf
[347] https://folk.universitetetioslo.no/trygver/1979/mvc-1/1979-05-MVC.pdf
[348] https://martinfowler.com/eaaDev/uiArchs.html
[349] Potel, M. (1996). *MVP:* Model-View-Presenter the Taligent programming model for C++ and Java. *Taligent Inc*, 20.



Слика 40. MVC патерн

У односу на почетну архитектуру, приказаној на Слици 40, новије архитектуре, које се и даље називају *MVC* и за које под тим називом постоји подршка у софтверким оквирима и платформама, поред три основне компоненте садрже и додатну архитектуру за управљање догађајима. Разликују се међусобно по начину како је имплементиран поглед, да ли је пословна логика укључена у контролер и да ли обсервер комуницира, поред погледа и модела и с контролером. У неким унапређењима *MVC* патерна, уместо обсервера се користи *Dependency Injection* патерн.

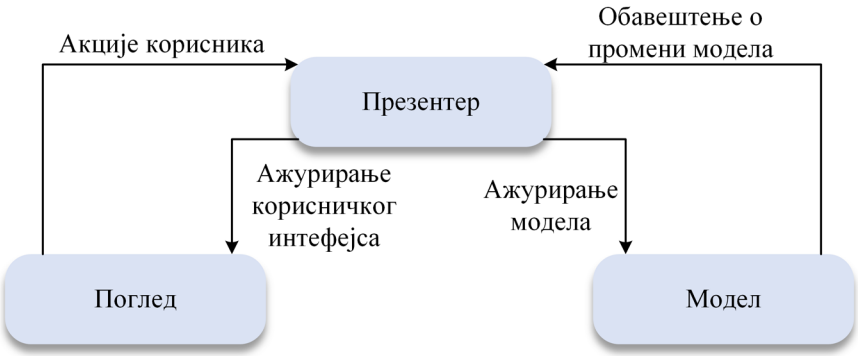
Један од покушаја да се унапреди *MVC* патерн, а да при томе његова архитектура остане јасна и једноставна, јесте *Model View Presenter* патерн.

19.5.2 Model View Presenter

Model View Presenter – MVP патерн , настао је као побољшање *MVC* патерна. У патерн су уграђени догађаји и асинхрона обрада, а комуникација између компоненти је боље структурирана и организована. Раздвајање надлежности између компоненти је јасније. *MVP* патерн садржи три компоненте (Слика 41):

- Модел (*Model*) представља базу података презентационог слоја, у оквиру које се дефинишу, смештају и претражују сви подаци који ће се приказати или на неки други начин имати утицај на кориснички интерфејс.
- Поглед (*View*) је пасивни део корисничког интерфејса који приказује податке добијене од модела и прослеђује презентеру догађаје генерисане у корисничком интерфејсу. Поглед у себи не садржи презентациону логику.
- Презентер (*Presenter*) је део корисничког интерфејса који реагује на догађаје које добија од модела и погледа, интерпретира их и реагује претраживањем података садржаних у моделу, које форматизује и доставља погледу за приказивање. Сва презентациона логика се налази у презентеру. Презентер нема директан интерфејс с корисницима, већ с њима комуницира преко погледа.

Структура *MVP* патерна је хијерархијска. Комуникацију с корисником, улаз и излаз, преузима поглед. Комплетна презентациона логика је уграђена у презентер. Презентер преузима на себе улогу посредника између модела и погледа. Предност примене овог патерна је јасно хијерархијско раздвајање надлежности између компоненти (*separation of concern*), а која олакшава пројектовање и тестирање.



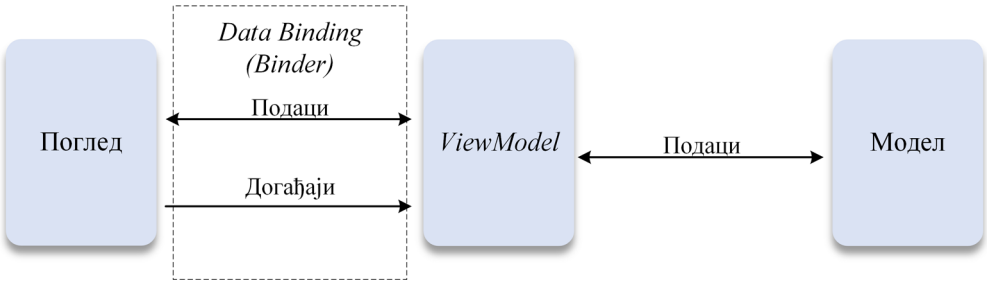
Слика 41. Model View Presenter патерн

Оргинални *MVP* патерн данас се не користи. Користе се његове варијанте: *Passive View*[350] и *Supervising controller*[351].

19.5.3 Model View View Model

У већини случајева, у бази модела *MVP* патерна помешани су подаци пословне и презентационе логике. Ради једноставнијег пројектовања и тестирања, корисно је раздвојити их у два посебна сегмента с засебним надлежностима. Најједноставнији начин да се то оствари је да се база података презентационог слоја измести у презентер, а у модел се премести део пословне логике која комуницира са корисничким интерфејсом. Када је први пут предложена 2004. године, та архитектура је названа *Presentation Model* (*PM*)[352]. Једна од варијанти такве архитектуре је *Model View View Model* (*MVVM*)[353] коју су предложили *Мајкрософтови* инжењери Кен Купер и Тед Питерс. У односу на *MVP* патерн, модел у *MVVM* патерну садржи податке пословне логике који су потребни за интеракцију с презентационим слојем и валидацију података који долазе из презентационог слоја. *View Model* садржи податке и логику презентационог слоја, док поглед садржи дефиниције понашања, податке и догађаје који су потребни за непосредну интеракцију с корисником. *MVVM* патерн је приказан на Слици 42.

[350] https://martinfowler.com/eaaDev/PassiveScreen.html
[351] https://martinfowler.com/eaaDev/SupervisingPresenter.html
[352] https://martinfowler.com/eaaDev/PresentationModel.html
[353] https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm

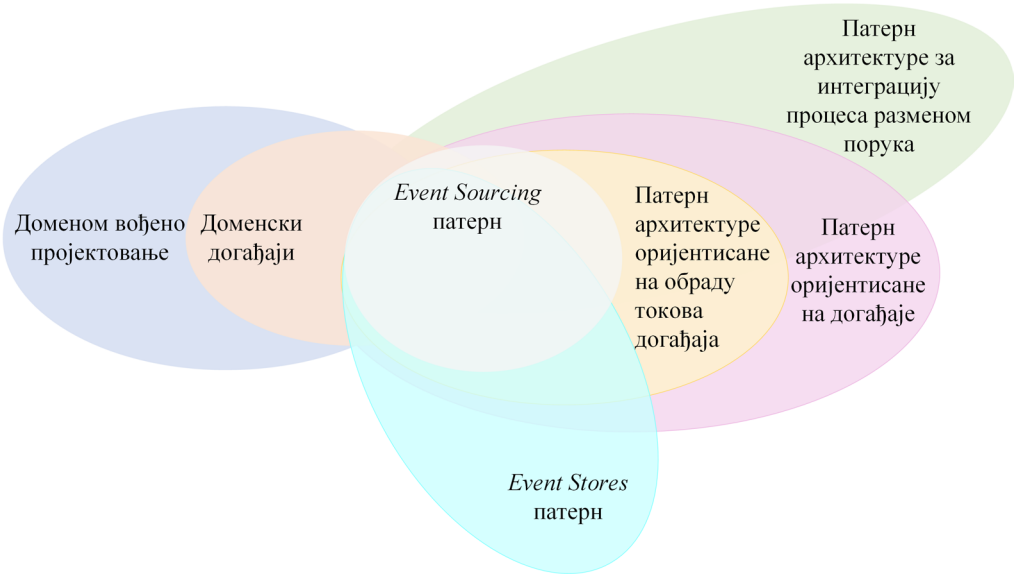


Слика 42. MVVM патерн

Овај архитектурни патерн подржавају *Angular*, *React*, *.NET*, а препоручен је и од стране *Гугла* као један од најпогоднијих начина за имплементацију корисничких интерфејса у *Android* окружењу.

20. ПАТЕРНИ АРХИТЕКТУРЕ ОРИЈЕНТИСАНЕ НА ПРОЦЕСЕ

Архитектура софтвера оријентисаног на процесе обухвата стилове и софтверске патерне проистекле из пројектовања, складиштења и обраде догађаја, као и патерне за агрегацију и интеракцију процеса. Таксономија архитектуре софтвера оријентисаног на процесе приказана је на Слици 43.



Слика 43. Таксономија архитектуре софтвера оријентисаног на процесе[354]

[354] Прилагођено према: https://www.juxt.pro/blog/event-driven-architecture/

Архитектура софтвера оријентисаног на процесе садржи:

- Патерне и архитектурне стилове пројектовања – Доменом вођено пројектовање софтвера и доменске догађаје (*Domain Driven Design*, *Domain Events*).
- Патерне за складиштење догађаја и праћење њихове историје (*Event Sourcing*, *Event Store*, *Data Lake*).
- Патерн архитектуре оријентисане на догађаје (*Event Driven Architecture*).
- Патерн архитектуре оријентисане на обраду токова догађаја (*Event Streaming*).
- Патерне архитектуре за интеграцију процеса разменом порука (*Message Driven Architecture*).

У многим случајевима више наведених патерна комбинују се у конкретну архитектуру софтвера.

20.1 Доменом вођено пројектовање софтвера

Доменом вођено пројектовање софтвера (*Domain Driven Design – DDD*) је архитектурни стил који се користи искључиво за пројектовање пословне логике у вишеслојним архитектурама веома сложених система. Не користи се за пројектовање презентационог и апликационог слоја. Не користи се ни при пројектовању једноставних система, јер његова примена подразумева високе трошкове.

Главна предност овог архитектурног стила је то што се модел пословне логике сложеног система раздваја у више домена или поддомена у којима су концепти пројектовања и језик комуницирања пројектних тимова прилагођени специфичним особинама домена или поддомена.

Користи се највише у објектно-оријентисаном програмирању. Појавом микросервиса, све више се користи и у пројектовању софтвера оријентисаног на процесе. Улаз у доменом вођено пројектовање софтвера резултат је анализе захтева, на основу *UML* дијаграма активности или, ређе, на основу *BPMN* дијаграма. Излаз је доменски модел који представља основу за архитектуру и спецификацију апликационих програмских интерфејса пословне логике.

20.1.1 Домен

Доменом вођено пројектовање софтвера представља рационализацију и прилагођавање моделом вођеног развоја софтвера и развоја софтвера вођеног подацима, конкретним доменима примене у веома сложеним системима. За разлику од метода развоја софтвера од којих је потекао, а које обухватају све фазе развоја софтвера, рационализација у доменом вођеном пројектовању софтвера огледа се у

његовом свођењу искључиво на фазу пројектовања пословне логике.

Доменом вођено пројектовање софтвера настало је из потребе да се дефинише скуп принципа, патерна и поступака у пројектовању софтвера, који би могли да се примене у пројектовању апстрактног и употребљивог доменског модела пословног система. Примарни циљ је прикупљање доменског знања неопходног за пројектовање слоја пословне логике, изолацијом доменских концепата и идентификацијом њихових узајамних веза. На основу доменског модела, који представља резултат пројектовања, дефинише се архитектура и обавља имплементација. Место доменског слоја у вишеслојној архитектури је показано на Слици 44.

Слика 44. Доменски слој у вишеслојној архитектури софтвера

Домен представља област конкретне примене софтвера. У односу на окружење, домен дефинише границе пословног система или неког његовог дела. Дефинисање домена и његових граница први је корак у пројектовању. Уколико је обим домена преширок, он се дели на више домена (поддомена). У микросервисним архитектурама се микросервис дефинише у оквиру поддомена.

Након дефинисања граница домена, следи дефинисање контекста, тј. синтаксе и семантике за концепте који се идентификују у току пројектовања. Концептима пројектовања се додељује синтакса и семантика повезана с компонентама реалног система. При додељивању синтаксе и семантике, узима се у обзир и очекивана архитектура софтвера, па се називи и значења усклађују и са архитектуром софтвера.

20.1.2 Речник термина у оквиру домена

Потпуно пресликавање синтаксе и семантике концепата реалног система на компоненте модела није могуће. За концепте модела којима се дефинише контекст, нужно је обезбедити такву синтаксу и семантику која, у највећој мери, има једнако значење за све стејкхолдере: кориснике, систем-аналитичаре, архитекте софтвера и програмере. Увођењем нових концепата у модел, генерише се речник термина који у свим фазама израде софтвера, од пројектних захтева па до кодирања, имају исту или сличну синтаксу и семантику (*Ubiquitous language*). При дефинисању концепата на нижим нивоима архитектуре софтвера, имена и језик за опис рачунарског кôда треба да прате синтаксу и семантику доменског модела. Тиме се омогућава једноставније пресликавање доменског слоја у конкретну архитектуру софтвера и рачунарски кôд.

Речник заједничких термина је нарочито користан при профилисању корисничких

дискусија (*user story*) у *DevOps*-у и агилном развоју софтвера[355]. Усвајање речника заједничких термина које употребљавају корисници, аналитичари система, програмери и други стејкхолдери значајан је разлог за прихватање овог метода пројектовања софтвера.

20.1.3 Повезан контекст

Повезан контекст (*Bounded Context*)[356] дефинише концепте на физичким или логичким границама домена или поддомена, као и интерфејсе од њих ка другим доменима и окружењу. Дефинисање повезаног контекста представља један од најважнијих поступака пројектовања. При дефинисању концепата унутар граница домена, може се десити да у различитим доменима постоје коцепти са истим називом (синтаксом), а различитом семантиком. Скуп таквих концепата описује се интерфејсима. Интерфејсе је важно исправно дефинисати јер представљају потенцијалну основу за интеграцију софтверских подсистема. На пример, „Клијент“ се као концепт појављује у домену продаје, а исто тако и у домену финансија, с различитим начинима описа његове структуре и понашања.

20.1.4 Доменски модел

Резултат процеса пројектовања је доменски модел који садржи опис пословне логике организован на начин којим се ефективно превазилази дистанца између реалности пословног система и рачунарског кôда. То подразумева да се у дефинисање доменског модела морају укључити ограничења и концепти изабране архитектуре софтвера и парадигме развоја софтвера, који ће утицати на имплементацију. Многи аутори истичу своја искуства из праксе да архитектура добро пројектованог софтвера умногоме прати архитектуру пословног система.

У време када је објављен први рад о доменом вођеном пројектовању софтвера[357], процесне екстензије и сервисно-оријентисана архитектура увелико су већ биле саставни део преовладавајућих објектно-оријентисаних програмских језика *Java*, *C#* и *Python*. У складу с тим, концепти којима се описује доменски модел дефинисани су тако да могу обухватити објектну и процесну парадигму развоја софтвера, као и сервисно-оријентисану архитектуру која је тада била веома актуелна. Концепти су постављени тако да садрже минимум семантике потребне за дефинисање објектно и процесно-оријентисаног софтвера, а истовремено омогућавају дефинисање додатне семантике, у складу с потребама концепата реалног система.

Да би могао да подржи објектну и процесну парадигму, концепт објекта из објектно-

---

[355] https://dannorth.net/whats-in-a-story/

[356] Rademacher, F., Sorgalla, J. & Sachweh, S. (2018). Challenges of domain-driven microservice design: A model-driven perspective. *IEEE Software*, 35(3), 36–43.

[357] Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

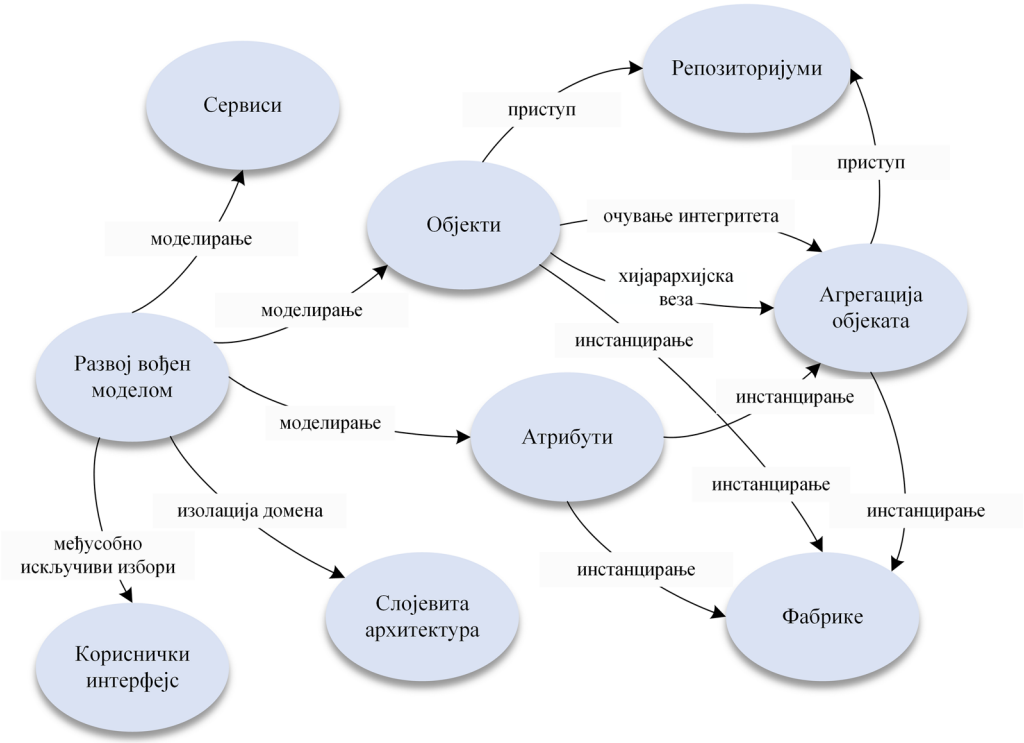
оријентисаног програмирања је разлабављен. Објекат (*Entity*) се дефинише само именовањем без прецизне дефиниције његове структуре и понашања. Тако лабаво дефинисан објекат може бити компонента система, модул, ресурс или само догађај. Модел може садржати објекат без идентитета који садржи само атрибуте (*Value Objects*). Ентитет и објекат без идентитета могу представљати и агрегацију објеката (*Aggregates*). Дефинисање објекта је раздвојено од процеса његовог инстанцирања. Инстанцирање објекта се реализује коришћењем концепта названог „фабрика“ (*Factories*). Превазилажење ограничења сиромашног модела података објектно-оријентисаног модела и искорак ка информационим моделима процесно-оријентисаног софтвера реализовани су увођењем репозиторијума (*Repositories*), који представља апстрактну базу знања с метаподацима и подацима о објектима, ресурсима, догађајима, са свим њиховим инстанцама и агрегацијама[358].

Сервиси (*Services*) у *DDD* су дефинисани као агрегација свега осталог што се не може сврстати у лабаво дефинисане објекте. Без обзира на скромну и нејасну дефиницију сервиса у *DDD* и још нејасније и спорније начине његовог коришћења, његово формално увођење у процес пројектовања је био огроман напредак. Независно од *DDD*, широка примена сервисно-оријентисане архитектуре довела је до коришћења сервиса и у сврхе за које није намењен, што је резултовало високим трошковима и компликованим одржавањем, а на крају довело и до кризе сервисно-оријентисане архитектуре и њеног постепеног напуштања. Реформа сервисно-оријентисане архитектуре и њена ревитализација учињене су увођењем *REST API* архитектурног стила који је дефинисао сервис као трансформацију стања неког објекта или ресурса. У складу с тим, дефиниција сервиса, који се до тада у пракси, а и у оргиналној спецификацији *DDD* користио као „универзална канта за ђубре“ за смештај свега што није објекат, напуштена је и трансформисана у микросервис који има форму агрегације концепата коју чине: објекат или ресурс чије стање се мења извршењем сервиса, и скуп догађаја или само рачунарски кôд који ту промену извршава. Микросервис се најчешће дефинише у оквиру издвојеног поддомена.

Мапа концепата који се користе за спецификацију доменског модела дата је на Слици 45.

Концептима доменом вођеног пројектовања у потпуности се може моделирати софтвер оријентисан на процесе: лабаво дефинисан концепт ентитета омогућава да се њиме представе догађаји; агрегације омогућавају да се њима представе сервиси, процеси, апликације, информациони систем и *B2B* мреже; неименовани објекти који садрже само вредност погодни су за представљање токова догађаја; концепт „фабрике“ омогућава раздвајање описа концепта (нпр. догађаја) од његове инстанце, тако да се из скупа метаподатака који описују концепте доменског модела могу у току извршења програма генерисати њихове инстанце.

[358] Evans, E. & Szpoton, R. (2015). *Domain-driven design*. Helion.



Слика 45. Мапа концепата за спецификацију доменског модела[359]

20.1.5 Употреба доменског модела

Доменски модел је резултат пројектовања који узима у обзир концепте анализе, планирану архитектуру и имплементацију, али не обухвата у потпуности све фазе пројектовања софтвера. Доменски модел само имплицитно дефинише модел архитектуре слоја пословне логике. Процесу пројектовања и изградње доменског модела претходи фаза анализе система, најчешће преко *UML* дијаграма активности, ређе преко *BPMN*, а следи експлицитно дефинисање потпуног модела архитектуре.

Доменом вођен развој софтвера посебно је популаран код пројектовања микросевисних архитектура. Концепт поддомена омогућава засебно моделовање сваког микросервиса и његове интеракције с другим микросервисима, коришћењем повезаног контекста и агрегације.

[359] Прилагођено према: Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional. https://msstest.sankuai.com/v1/mss_156341e12eb248878e532d-d820706c40/mall-data-init/ddd-en.compressed.pdf

20.2 Концепти моделовања софтвера оријентисаног на процесе

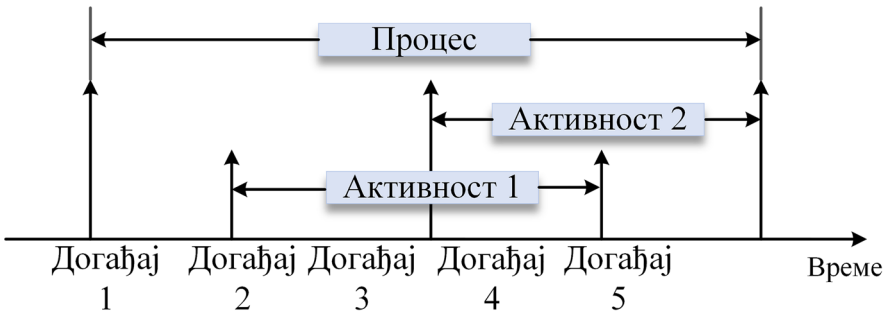
У анализи система, моделовању пословних процеса, моделовању пословне логике, дефинисању архитектуре и имплементацији, користе се концепти који су специфични за софтвер оријентисан на процесе. У оквиру овог поглавља дат је кратак преглед најчешће употребљаваних концепата, њихових апстракција, семантике, као и најнужнијих имплементационих детаља, који су потребни за њихово коришћење у процесу пројектовања софтвера.

20.2.1 Агрегације у моделима софтвера оријентисаног на процесе

Догађај је основни концепт на коме се заснива парадигма развоја софтвера оријентисаног на процесе. Најчешћи вид апстракције у развоју софтвера оријентисаног на процесе је агрегација догађаја[360]. Софтвер оријентисан на процесе може се представити хијерархијском структуром следећих концепата, од којих је догађај основни, а остали су агрегације догађаја:

1. Догађај.
2. Сервис, као агрегација догађајâ који се односе на ресурсе.
3. Процес, као агрегација догађајâ који се односе на појаве.
4. Интеракција процеса – апликација.
5. Интеграција апликација – информациони систем.
6. *B2B* мреже – интероперабилни информациони системи.

Међусобни однос догађаја, сервиса и процеса дат је на Слици 46.



Слика 46. Догађај, сервис и процес

[360] Barrett, D. J., Clarke, L. A., Tarr, P. L. & Wise, A. E. (1996). A framework for event-based software integration. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4), 378–421.

Процеси и њихове везе се могу агрегирати у апликације. Скуп апликација заједно с њиховим међусобним интеракцијама се даље може агрегирати у информациони систем. Информациони системи предузећа која пословно сарађују могу се повезати у *B2B* мреже[361].

Патерни обраде догађаја у сервисима и процесима спадају у категорију интеграционих патерна, јер користе апстракцију агрегације. Имплеметирају се у форми софтверских платформи и екосистема.

20.2.2 Догађај

Централни и атомски елемент преко којег се моделира динамика софтверског система оријентисаног на процесе, назива се догађај (*Event*). Догађај представља било коју промену у софтверу или његовом окружењу: извршење једне инструкције или дела кôда, промену вредности променљиве, појаву улаза и деловање излаза на окружење[362].

Таксономија догађаја

Према нивоу апстракције, догађаји се могу поделити на:

- **регуларне** (*ordinary event*),
- **изведене** (*notable event*), и
- **комплексне** (*complex event*).

Регуларни догађаји представљају промене стања система, док изведени догађаји представљају апстракције над променама стања и означавају појаву или најаву предстојећих проблема, настанак неке пословне прилике, достизање прага вредности неке променљиве, или њено прекомерно одступање. Изведени догађаји се најчешће генеришу при извршењу регуларних догађаја, када се испитује променљива на коју се регуларни догађај односи, упоређивањем са лимитима задатим атрибутима догађаја. Изведени догађаји се могу генерисати и корелационом и другим статистичким анализама, као и методама вештачке интелигенције које анализирајући актуелне и претходне догађаје имају за циљ да открију нове пословне могућности или да упозоре на предстојеће проблеме. Регуларни и изведени догађаји су основа за откривање комплексних догађаја. Комплексни догађаји се откривају у реалном времену, напредном аналитиком података о регуларним и изведеним догађајима.

Догађаји се на основу припадности софтверском систему могу поделити на:

---

[361] Cook, J. E., & Wolf, A. L. (1998). Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology* (TOSEM), 7(3), 215-249. https://dl.acm.org/doi/pdf/10.1145/287000.287001

[362] Schriber, T. J., & Schriber, T. J. (1974). *Simulation using GPSS* (Vol. 1). New York: Wiley.

- **ендогене**, интерно генерисане догађаје, који се односе на промену стања у софтверском систему, и
- **егзогене**, екстерне, који могу бити улазни догађаји преко којих окружење утиче на софтверски систем, и излазни догађаји преко којих софтверски систем утиче на окружење.

Ендогене и егзогене догађаје, даљом детаљнијом класификациојом по припадности неком процесу (домену), можемо поделити на:

- **доменске догађаје** (*Domain Events*), код којих се генерисање и обрада догађаја врше у истом процесу, и
- **интеграционе догађаје** (*Integration Events*) код којих су догађаји генерисани у једном процесу, а њихова обрада се обавља у другом процесу. Комуникација и интеракција процеса реализује се преко интеграционих догађаја.

Догађаји се могу комбиновати и формирати нови догађај који се тада назива **комбиновани догађај**. Они представљају комбинацију више догађаја који су повезани неком логичком операцијом, најчешће конјункцијом или дисјункцијом (*AND*, *OR*). Промена стања под утицајем комбинованог догађаја наступа у случају да су један или сви догађаји, дефинисани комбинованим догађајем, наступили и да при томе логичка операција која их повезује буде испуњена (има вредност *True*). Користе се за гранање и обједињавање токова процеса.

Дефинисање и инстанцирање догађаја

У пројектовању софтвера, термин „догађај“ се двојако користи: за спецификацију (дефинисање) врсте или типа догађаја, а такође и за именовање инстанце, индивидуалног појављивања једне врсте догађаја. Врста или тип догађаја је концепт пројектовања. Инстанца догађаја је концепт извршења програма. Атрибути догађаја се дефинишу у типовима догађаја, док се конкретне вредности атрибута догађаја додељују инстанци догађаја[363].

За један дефинисан тип догађаја током извршења програма може се генерисати више инстанци догађаја. Број инстанци једног типа догађаја може бити велики, нарочито код *IoT* уређаја. Тип догађаја може бити мерење неке физичке величине, а инстанце догађаја представљају резултати мерења током времена, којих може бити много. Такође, у електронском пословању, тип догађаја може бити приступ делу екранске форме, док су конкретни догађаји инстанце везане за приступ сваког купца појединачно, којих такође може бити много.

[363] Michelson, B. M. (2006). Event-driven architecture overview. *Patricia Seybold Group*, 2(12), 10–1571

Архитектуре интеграције и обраде догађаја

У архитектурама софтвера, догађаји се агрегацијом могу груписати по:

1. Времену настанка, и тада се таква архитектура назива **оријентисаном на догађаје**.
2. Ресурсима реалног система којима догађаји мењају стања, и тада се таква архитектура назива **оријентисаном на сервисе**.
3. Припадности појави, процесу или трансакцији, када су догађаји и сервиси груписани и уређени по хронолошком редоследу који прати животни циклус појаве, процеса или трансакције, од настанка до терминовања. Тада се таква архитектура назива **оријентисаном на процесе**.

Свака од наведених архитектура је оријентисана ка процесима. Догађаји се увек агрегирају у процесе. Називи су дати по превлађујућем начину агрегације догађаја. Механизам који се користи за агрегацију догађаја су листе догађаја. Агрегација се реализује тако што се инстанца догађаја постави у листу догађаја.

Ова подела је уведена јер је могуће обраду догађаја прилагодити начину агрегације догађаја, тако да коришћење рачунарских ресурса за сваки од начина агрегације буде близу оптималног.

20.2.3 Сервис

Синоним за сервис при моделовању пословних процеса је активност. Сервиси су апстракције које укључују ресурс (компоненту или ентитет) реалног система и опис трансформације, састављене од скупа догађаја који трансформишу стање ресурса на који се сервис односи. Пример за сервис може бити прање аутомобила: ресурс је аутомобил, први догађај с којим почиње сервис је долазак аутомобила, скуп догађаја током прања аутомобила трансформише стање ентитета аутомобил из прљавог у чисто.

Сервиси представљају спону између објектног и процесног приступа развоју софтвера. Односе се на ресурсе, ентитете у систему, али не представљају операције-методе, као у објектном приступу, већ активност, експлицитну трансформацију стања.

Дефинисање и инстанцирање сервиса

Сервис се дефинише метаподацима, док конкретне инстанце сервиса функционишу у реалном времену са својим инстанцама дефинисаних података.

20.2.4 Процес

Процес представља агрегацију, хронолошки уређену секвенцу догађаја и сервиса који прати појаву или трансакцију од почетка до завршетка. Процес је једини концепт у развоју процесно-оријентисаног софтвера који може представљати извршни рачунарски програм. Догађаји и сервиси су само његови делови.

Задатак архитектуре софтвера је да обезбеди инфраструктуру и концепте за хронолошку агрегацију догађаја и сервиса у процесе, организацију њиховог извршења и управљање. Пројектовање начина организације и агрегирања догађаја и сервиса у процесе није једнозначан процес, нити се за тај посао може користити само један архитектурни патерн. Начин реализације архитектуре софтвера оријентисаног на процесе првенствено зависи од корисничких захтева и услова рада система. За различите случајеве коришћења развијени су различити архитектурни патерни обраде догађаја.

Дефинисање и инстанцирање процеса

Процес се дефинише метаподацима, а конкретне инстанце процеса садрже атрибуте дефинисане метаподацима који егзистирају у реалном времену.

20.2.5 Интеракција процеса – апликација

Рачунарски програм или апликација представља агрегацију, скуп свих процеса у оквиру пословне активности на коју се апликација односи. Процеси у апликацији су у интеракцији, размењују информације и синхронизују свој рад.

За потпуну подршку повезивања процеса у апликације, нужно је увести и концепте за комуникацију и синхронизацију процеса. Комуникација се реализује разменом интеграционих догађаја између процеса, тј. генерисањем догађаја у једном процесу, који за одредиште и извршење има други процес. Комуникација и синхронизација процеса повезане су с проблемима конкурентног програмирања: приступом критичном региону и сигнализацијом.

Проблеми конкурентног програмирања при комуникацији и синхронизацији процеса могу се смањити лабављењем спреге између појаве догађаја и његовог извршења. Начин на који се то реализује је примена *Pub/Sub* патерна. Процес који производи догађај уопште не мора да зна какве ће последице тај догађај изазвати у другим процесима, док конзументи догађаја знају да је догађај наступио и приступају његовом извршењу, али не морају ништа знати о пореклу догађаја. Ово је ефикасан метод који проблеме конкурентног програмирања и сигнализације у највећој мери заобилази, па је постао саставни део већине патерна архитектуре оријентисане на процесе.

20.2.6 Информациони систем

Информациони систем представља агрегацију рачунарских програма – апликација у оквиру предузећа. Апликације у информационом систему, слично као и процеси у апликацијама, комуницирају разменом интеграционих догађаја. С обзиром на то да су информациони системи у већини случајева дистрибуирани, размена интеграционих догађаја се реализује преко посредника, најчешће преко система за размену порука. Да би се догађај пренео од извора до одредишта, најпре се серијализацијом трансформише у поруку, па се преко система за размену порука доставља до одредишта, где се порука распакује и догађај извршава. Поред обезбеђења дистрибуираног система за размену порука, за интеграцију апликација у информациони систем потребно је интегрисати поглед на податке и унифицирати приступ ресурсима информационог система[364]. Централну улогу у интеграцији архитектуре оријентисане на догађаје са информационим системом предузећа има окосница за интеграцију (*Enterprise Integration Backbone*). Окосница обезбеђује ресурсе за обраду догађаја, филтрирање, рутирање, трансформацију догађаја, канале за њихов транспорт, позиве сервиса, позиве пословних процеса, као и приступ осталим ресурсима информационог система предузећа.

20.2.7 B2B мреже

Информациони системи предузећа, која пословно сарађују, могу се интегрисати у *B2B* мреже. Интеграција информационих система у *B2B* мреже реализује се преко специјализованих спољних сервиса као што су *EBXML*[365], *Rosettanet*[366] и *ISO*-15000[367], који су се у прошлости у ту сврху много користили. Обезбеђење интероперабилности је највећи проблем у *B2B* мрежама. У ту сврху се користе онтологије које су специфичне за поједине привредне гране[368]. Анализа и имплементација *B2B* мрежа превазилази оквире ове књиге.

20.3 Информациони модел софтвера оријентисаног на процесе

За разлику од парадигме развоја објектно-оријентисаног софтвера, у којој се моделирају подаци и врши обрада података, у развоју софтвера оријентисаног на процесе моделирају се догађаји, који по својој природи представљају информације (инкременте знања о систему), као и њихови агрегирани концепти: сервиси и процеси. Обрада догађаја представља обраду знања. Знање о систему, подаци и метаподаци, њихова синтакса, семантика и међусобне релације имплицитно су садржани у моделу процеса.

---

[364] https://www.enterpriseintegrationpatterns.com/index.html
[365] http://www.ebxml.org/
[366] https://www.gs1us.org/resources/rosettanet
[367] https://www.iso.org/standard/79108.html
[368] Cui, Z., Jones, D. & O'brien, P. (2002). Semantic B2B integration: issues in ontology-based approaches. *ACM SIGMOD Record*, 31(1), 43–48.

Резултат моделовања софтвера оријентисаног на процесе је информациони модел састављен од метаподатака и података који описују: актере, догађаје, сервисе, процесе, компоненте реалног система, документе, ресурсе, ентитете, као и њихове међусобне односе и интеракције. Метаподаци и подаци се смештају у репозиторијум, у којем се налази комплетно знање о систему. Обрада у софтверу оријентисаном на процесе представља обраду знања и генерисање нових информација.

20.3.1 Метаподаци и подаци догађаја

У информационом моделу софтвера оријентисаног на процесе, догађај се представља слогом догађаја, информационом структуром која се састоји из три дела: заглавља (*Event Header*), тела догађаја (*Event Body*) и сервиса који се називају „прихватачи догађаја“ (*Event Handler*). Заглавље догађаја садржи метаподатке: идентификатор типа догађаја, идентификатор инстанце догађаја, приоритет догађаја, време наступања догађаја, стање догађаја, опис догађаја и начин његовог коришћења. Тело догађаја садржи скуп атрибута који се односе на стања система које догађај мења. Догађај може, али и не мора, имати атрибуте. Информација о наступању неког догађаја је некада сасвим довољна. Атрибути догађаја који описују стања система су мултимодални и могу бити: број, текст, звук, слика, видео-снимак и друго. Прихватач догађаја је процедура, операција или сервис који требају да се изврше када догађај наступи. Процедуре и сервиси прихватача догађаја током извршења могу да генеришу нове догађаје истог или различитог типа. Атрибути догађаја се по правилу не ажурирају. Када је то потребно, прихватач догађаја њихову промену врши генерисањем новог догађаја исте врсте, са ажурираним вредностима атрибута.

Слог догађаја се приликом генерисања или пристизања догађаја одмах меморише у репозиторијум, односно базу знања о догађајима. База знања о догађајима трајно се чува, не ажурира се и не брише. Слогу догађаја се приступа искључиво преко његове адресе у бази знања о догађајима.

Дефиниција догађаја са скупом одговарајућих метаподатака у бази знања се меморише као тип догађаја. Инстанца догађаја је увек повезана с типом догађаја, тако што у скупу својих атрибута држи ознаку типа догађаја од којег је потекла.

20.3.2 Метаподаци и подаци процеса

Процес се у информационом моделу представља хронолошки уређеном листом догађаја (*Event Chain*). Синоними који се у литератури користе за листе догађаја су: редови догађаја (*Event Queue*) и канали догађаја (*Event Channel*). Листа догађаја је механизам којим се реализује агрегација догађаја у процес.

Елемент листе догађаја уместо комплетног слога догађаја садржи само адресу инстанце догађаја смештеног у репозиторијуму, што је сасвим довољно за читање

слога догађаја. Тако дефинисан елемент листе догађаја заузима мало меморије, што знатно убрзава рад с листама догађаја. Листе догађаја се реализују као перзистентне *FIFO* структуре, најчешће као издвојен део базе знања.

Листа догађаја има следеће намене:

1. **Агрегирање догађаја у процес.** Приликом доласка екстерног догађаја или генерисања интерног догађаја, догађај се по хронолошком редоследу смешта у листу процеса коме припада.
2. **Успостављање хронолошког редоследа између догађаја**. Догађај се може генерисати у току рада програма, а да притом време његовог наступања (извршења) буде у будућности. Догађаји се приликом генерисања смештају у листе догађаја по хронолошком редоследу времена наступања. У листи догађаја остају све док не дође време за њихово извршење.
3. **Успостављање равнотеже између брзине генерисања догађаја и обраде**. Догађаје најчешће генеришу екстерни процеси. Уколико је интензитет пристизања догађаја већи од капацитета обраде догађаја, тада листа догађаја служи као бафер у коме догађаји остају све док се брзине њиховог пристизања и обраде не ускладе.

Скуп свих листи догађаја у апликацији или информационом систему назива се Магистрала догађаја (*Event Bus*).

Саставни део информационог модела су и системи за комуникацију и размену порука између процеса, који такође користе *Event Bus*.

Део информационог модела који се односи на меморисање догађаја, листе догађаја и размену порука, најчешће се имплементира у оквиру платформе или оквира који се користи за развој и експлоатацију софтвера. Тај део за програмера није видљив. Приступа му се преко апстрактних концепата као што су: *Event Store* и *Data Lake*, магистрала порука (*Message Bus*) и магистрала догађаја (*Event Bus*), који нуде унапред дефинисане операције над догађајима, порукама и синхронизационим концептима, и на јасан и прегледан начин уводе у софтвер асинхроне догађаје и њихове апстракције.

Процеси се могу спецификовати и непосредним повезивањем сервиса, оркестрацијом и кореографијом.

20.4 Патерни складиштења догађаја

Постоји значајна разлика у управљању подацима између архитектура оријентисаних на догађаје и објектно-оријентисаних архитектура. Објектно-оријентисане архитектуре најчешће меморишу само последње стање система (*state-oriented persistence*). Ажурирање података и брисање обавља се у складу с логиком рада апликације, најчешће *CRUD* (*create*, *read*, *update*, *and delete*) операцијама.

У архитектури оријентисаној на догађаје, након генерисања, слог догађаја се увек прво уписује у базу знања, односно репозиторијум, па се затим објављује, рутира

и извршава. Догађају се у свим операцијама приступа преко његове адресе у бази знања. Уписивање догађаја у базу знања мора да буде атомска операција. Догађаји се не ажурирају, нити се бришу. Промена стања система или исправка погрешно генерисаног догађаја реализује се једноставним генерисањем новог догађаја с новим вредностима атрибута. Актуелно стање система се реконструише на основу секвенце пређашњих догађаја који су мењали то стање.

Последица таквог приступа је да у архитектурама оријентисаним на догађаје практично не постоје операције ажурирања и брисања података[369]. То не мора да буде недостатак, ажурирање и брисање података, на пример у банкарским апликацијама, строго је забрањено. У таквим апликацијама могућност измене или брисања догађаја додатно се ограничава применом решења заснованих на *Block-chain* технологијама.

Претраживање догађаја и њихову употребу могу на различите начине вршити различити процеси. У циљу повећања ефикасности, нужно је раздвојити меморисање догађаја од претраживања. Генерисање и меморисање догађаја захтева минималне ресурсе и може се једноставно реализовати. С друге стране, претраживање базе знања о догађајима врши се на основу метаподатака, и атрибута, статистичким методама и методама вештачке интелигенције, што може бити комплексно и захтевати знатну количину ресурса. Такође, нису сви упити над базом знања једнаки.

Процесе претраживања потребно је прилагодити различитим начинима употребе и вршити посебним апликацијама. То се реализује применом *CQRS* (*Command-Query-Responsibility-Segregation*)[370] приступа у реализацији базе података о догађајима, најчешће применом *Pub/Sub* патерна у комбинацији с репозиторијумом.

Имплементација базе знања у систему оријентисаном на догађаје веома је једноставна. У пракси се најчешће састоји само од једне бинарне релације, табеле с две колоне, у које се смештају слогови догађаја, с метаподацима. У једној колони је идентификатор догађаја, а у другој агрегирани атрибути и метаподаци. Такву једноставну информациону структуру могу понудити све данас расположиве базе података. Комплексност модела није у физичкој реализацији, већ у семантици метаподатака и вредностима атрибута који описују догађаје и њихове интеракције. Историја догађаја садржи све знање о понашању система од почетка његовог животног циклуса.

Бази знања се могу придружити и каталози који садрже сортиране, индексиране и уређене метаподатке за брже претраживање и приступ подацима о догађајима. Претраживање базе знања може се вршити на основу каталога, метаподатака или откривањем знања из садржаја атрибута догађаја.

---

[369] Long, Z. (2017, October). Improvement and implementation of a high performance CQRS architecture. In: *2017 International Conference on Robots & Intelligent System (ICRIS)* (pp. 170–173). IEEE.
[370] Greg Young, CQRS Documents by Greg Young, http://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf

Без обзира на једноставност модела података (бинарна релација, табела с две колоне), расположивост комплетне историје промене стања система кроз преглед догађаја је велика предност јер омогућава да се из историје промена стања система, применом статистичких метода и метода вештачке интелигенције, у реалном времену открију везе које постоје између вредности атрибута догађаја, а које у време пројектовања софтвера нису постојале, нити су могле бити познате[371].

Меморисање и трајно чување догађаја отвара и нови скуп могућности за ретроактивно пројектовање, тестирање, оптимизацију и одржавање апликација[372]. Ретроактивно рачунарство (*Retroactive Computing*) омогућава ретроактивну анализу стања система и *post hoc* отклањања грешака[373] .

За перзистентно складиштење и претраживање догађаја, релациони модел базе података и релациона алгебра могу се користити, али нису најбољи избор. Предност је то што је сваки упис догађаја у софтверу, који подржава релациони модел, атомска операција и о томе се даље не мора водити рачуна. Релациони модел је монолитан и није дистрибуиран, што представља недостатак. Мултимодални подаци у атрибутима догађаја не могу се претраживати методама релационе алгебре. Недостатак је и одсуство више различитих модела за претраживање и читање догађаја, који су потребни за *CQRS* имплементацију. И поред наведених недостатака, релациони модел се још увек користи за перзистентно складиштење догађаја, нарочито код апликација код којих су догађаји агрегирани у сервисе[374].

За перзистентно складиштење догађаја, нарочито су погодне базе података у форми дистрибуиране бинарне релације (*key-value databases*), нпр. *Amazon DynamoDB*[375], код којих се идентификатор догађаја неким хеш алгоритмом трансформише у његову дистрибуирану адресу у бази података. Приступ таквим базама података веома је брз. *MongoDB*, који је оријентисан на документе, има недостатак да је упис атомска операција за цео документ. Документ може садржати агрегације више догађаја, нарочито у микросервисним архитектурама. Упис новог догађаја своди се на поновни упис целог документа са свим догађајима, што непотребно троши ресурсе и усложњава обраду. *Apache Kafka* гарантује атомичност поступка уписивања нових догађаја само за партицију у коју се догађаји уписују, па у примени то ограничење треба имати у виду[376].

---

[371] Zhong, Y., Li, W., & Wang, J. (2019, February). Using event sourcing and CQRS to build a high performance point trading system. In *Proceedings of the 2019 5th International Conference on E-Business and Applications* (pp. 16-19).

[372] Müller, M. (2016). Enabling retroactive computing through event sourcing. https://d-nb.info/1115251406/34

[373] Meissner, D., Erb, B., Kargl, F. & Tichy, M. (2018, June). Retro-λ: An event-sourced platform for serverless applications with retroactive computing support. In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems* (pp. 76–87).

[374] https://martendb.io/

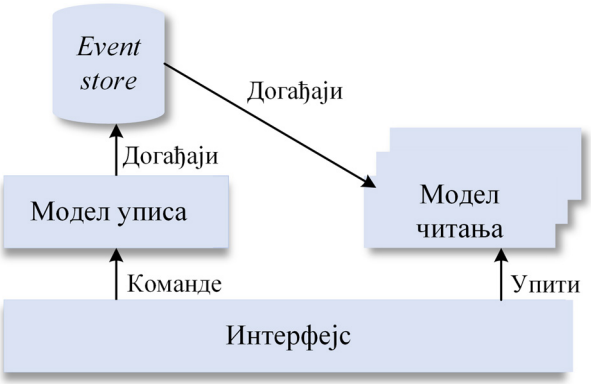
[375] https://aws.amazon.com/blogs/database/build-a-cqrs-event-store-with-amazon-dynamodb/

[376] Overeem, M., Spoor, M., Jansen, S., & Brinkkemper, S. (2021). An empirical characterization of event sourced systems and their schema evolution—Lessons from industry. *Journal of Systems and Software*, 178, 110970. https://www.sciencedirect.com/science/article/pii/S0164121221000674

У већим дистрибуираним системима, догађаји се осим у бази података привремено меморишу и у системима за управљање дистрибуираним порукама (*Message Broker*) у којима се догађај серијализацијом смешта у поруку, а затим се порука рутира до одредишта, где се догађај из поруке распакује и извршава.

20.4.1 Event store

За ефикасно перзистентно складиштење и управљање догађајима неопходно је објединити дистрибуиране базе података о догађајима и системе за размену порука у једну целину, са заједничким концептима и приступом преко јединственог апликационог програмског интерфејса. Обједињен систем, поред меморисања догађаја, треба да подржи *Pub/Sub* патерн, тј. раздвајање модела уписивања догађаја у базу података, од модела за читање и претраживање података о догађајима и *CQRS* патерн претраживања[377], што је приказано на Слици 47.



Слика 47. Event Store модели уписа, претраживања и читања догађаја (CQRS)

Системи развијени за ту намену називају се *Event Store*. Имплементирани су тако што су дистрибуиране базе података за складиштење догађаја проширене концептима *Pub/Sub* архитектуре. Тиме је постигнута уштеда у ангажованим ресурсима, а програмери су добили јединствен *API* за већину послова у вези с догађајима. Имплементације које се користе су: *EventstoreDB*[378], *Eventuate*[379] и многе друге.

Варијанта архитектуре оријентисане на догађаје (*EDA*), у којој се за складиштење догађаја користи *Event Store*, а за претраживање и извршење догађаја *CQRS*, назива се *Event Sourcing*[380].

[377] Young, G. (2010). CQRS documents. https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
[378] https://www.eventstore.com/
[379] https://eventuate.io/
[380] Betts, D., Dominguez, J., Melnik, G., Simonazzi, F. & Subramanian, M. (2013). *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure.* Microsoft.

20.4.2 Информационо језеро

Информационо језеро (*Data Lake*) представља савремен концепт за складиштење података у архитектурама софтвера оријентисаног на процесе. Оно замењује претходно описане концепте за складиштење догађаја, а користи се и за интеграцију података у информационом систему предузећа. Представља пуну имплементацију концепта репозиторијума[381].

Поред смештања метаподатака и података процесно-оријентисаног софтвера, информационо језеро се може користити и за интеграцију свих података у информационом систему предузећа. Интеграција података је перманентан изазов са којим се суочавају пројектанти информационих система, као и корисници у својим свакодневним активностима. Функционална организација и подела предузећа на организационе јединице довела је до тога да информациони систем предузећа представља збир информационих система организационих јединица, тзв. информационих силоса (*data silos*), у којима су подаци семантички различито дефинисани и међусобно неинтероперабилни.

Преласком на процесну, а касније и организацију управљану променама, кренуло се ка интеграцији података потребних за извршење пословних процеса, а који се налазе у различитим „силосима“[382]. За проблем интеграције података нема ефикасног ни једнозначног решења. Једно од могућих решења је да се сви подаци смештају у јединствени репозиторијум предузећа, с метаподацима који описују њихову семантику у тренутку складиштења, а да се надаље кроз њихов животни циклус структуришу и обрађују[383]. Такав приступ је назван информационо језеро (*Data Lake*), и представљен је на Слици 48.

Информационо језеро, или у духу српског језика семантички исправније: информациони бунар, представља централизовани репозиторијум намењен интеграцији, обједињавању и складиштењу структурираних и неструктурираних података било којег обима или намене. Подаци се складиште одмах по приспећу, без додатног структурирања[384]. Каталог језера података садржи евиденцију података уређену по метаподацма и приступним правима[385]. Подацима у информационом језеру се приступа на основу метаподатака из каталога.

---

[381] Nargesian, F., Zhu, E., Miller, R. J., Pu, K. Q., & Arocena, P. C. (2019). Data lake management: challenges and opportunities. *Proceedings of the VLDB Endowment*, 12(12), 1986-1989. https://www.vldb.org/pvldb/vol12/p1986-nargesian.pdf

[382] Speiser, S. & Harth, A. (2010, September). Taking the lids off data silos. In P*roceedings of the 6th International Conference on Semantic Systems* (pp. 1–4).

[383] DalleMule, L. & Davenport, T. H. (2017). What's your data strategy. *Harvard Business Review*, 95(3), 112–121. http://globaltraining.com.ec/files/i1.pdf

[384] Madera, C. & Laurent, A. (2016, November). The next information architecture evolution: the data lake wave. In: *Proceedings of the 8th international conference on management of digital ecosystems* (pp. 174–180).

[385] Dibowski, H., Schmid, S., Svetashova, Y., Henson, C. & Tran, T. (2020). Using Semantic Technologies to Manage a Data Lake: Data Catalog, Provenance and Access Control. In *SSWS@ ISWC* (pp. 65–80).

Слика 48. Информационо језеро[386]

Имплементација информационог језера је једноставна. За смештање података у информационо језеро може се користити објектно складиште (*object storage*) које представља стандардну услугу мрежног сервиса датотека, нпр: *Amazon S3 cloud object storage*[387], *LakeFS*[388], *Ceph*[389] или *MiniIO*[390]. Може се користити и *Hadoop*[391]. За сваки податак који се смешта у репозиторијум, *object storage server* обезбеђује: адресу приступа, простор за метаподатке и простор за податке. Добијена адреса за приступ и метаподаци чувају се у каталогу информационог језера. Подацима у информационом језеру приступа се претраживањем метаподатака из каталога. Као резултат претраживања добијају се адресе за приступ изабраним подацима. Већина *object storage server*-а подржавају *S3 REST API* за приступ подацима. Такође, већина има стандардизоване интерфејсе за *Hive*, *Spark*, *Presto*, *AWS Athena* и друге софтверске пакете за претраживање, анализу и извештавање.

Инфраструктура за прелазак са силоса на складиштење података у информационом језеру не захтева велика улагања. У већини случајева то су пакети отвореног кôда и мрежни сервиси датотека који су већ расположиви у оквиру информационог система предузећа. Поред тога, ако је послована стратегија искључиво коришћење рачунарства у облаку, информационо језеро као платформу и као сервис нуди већина провајдера рачунарства у облаку.

Главни изазов у информационом језеру је како дефинисати метаподатке, тј. онтологију којом ће се описивати подаци и како те метаподатке у реалном времену

[386] Прилагођено према: https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake/
[387] https://aws.amazon.com/s3/
[388] https://lakefs.io/
[389] https://ceph.io/en/
[390] https://min.io/
[391] Khine, P. P. & Wang, Z. S. (2018). Data lake: a new ideology in big data era. In: *ITM web of conferences* (Vol. 17, p. 03025). *EDP Sciences.* https://www.itm-conferences.org/articles/itmconf/pdf/2018/02/itm-conf_wcsn2018_03025.pdf

придруживати каталогу. Поред обезбеђења семантичке конзистентности података и метаподатака, информациона језера треба да имају системе за управљање, заштиту података, доделу приступних права и контролу приступа[392] .

У информационом језеру складиште се подаци о догађајима и информационим токовима, па се информационо језеро може користити као база података за имплементацију већине патерна архитектуре заснованих на догађајима. Може се користити и за обраду података коришћењем *map-reduce* алгоритма. Подаци у информационом језеру представљају информациону базу за апликације, алате пословне аналитике, статистике и вештачке интелигенције. Информационо језеро је основни начин складиштења података у модерним архитектурама за обраду великих података као што су *Lambda* и *Kappa*.

20.5 Патерни обраде догађаја

Архитектура оријентисана на догађаје (*Event Driven Architecture – EDA*) спада у најпопуларније архитектуре у пројектовању софтвера. Употребна вредност је у томе што менаџменту предузећа нуди информацију о догађају од значаја за пословање у тренутку његовог настанка и тиме омогућава брзу реакцију[393], за разлику од унапред дефинисаних процедура у другим архитектурама софтвера, које свој рад базирају на стању пословног система меморисаном у базама података, код којих се идентификација догађаја значајног за пословање може пролонгирати до тренутка генерисања извештаја, па из тих разлога догађај може остати непримећен[394].

На додатни замах прихватању ове архитектуре софтвера утицао је и енорман раст електронског пословања, посебно *B2B* повезивања пословних система, у којима су пословне трансакције између партнера у већини случајева оријентисане ка догађајима.

У архитектуру софтвера оријентисаног на догађаје укључена су два екстерна актера:

1. Извор догађаја, креатор, произвођач, генератор догађаја.
2. Одредиште обраде, корисник, потрошач, конзумент догађаја.

Извори и одредишта догађаја могу бити конкретни ресурси информационог система предузећа: пословни процеси, сервиси, микросервиси, базе података, аутоматизовани агенти, људи који могу генерисати догађаје и извршавати акције спецификоване појавом догађаја. Топологија конкретне архитектуре оријентисане

[392] Giebler, C., Gröger, C., Hoos, E., Schwarz, H. & Mitschang, B. (2019). Leveraging the data lake: Current state and challenges. In: *Big Data Analytics and Knowledge Discovery:21st International Conference*, DaWaK 2019, Linz, Austria, August 26–29, 2019, Proceedings 21 (pp. 179–188). Springer International Publishing.

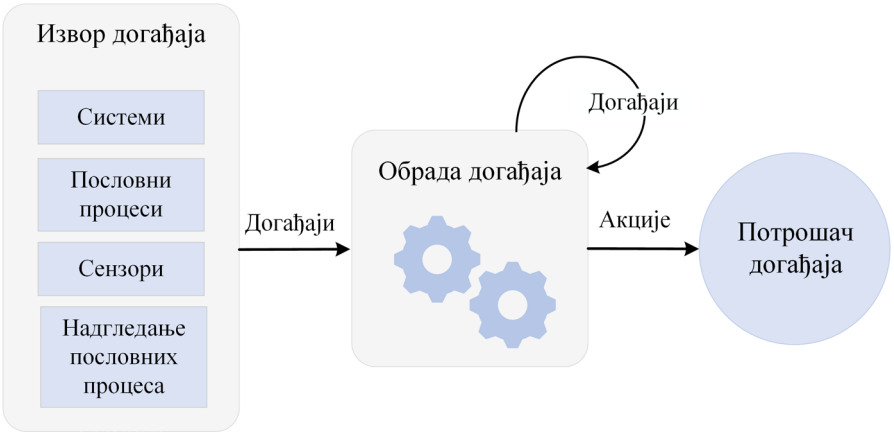
[393] Michelson, B. M. (2006). Event-driven architecture overview. *Patricia Seybold Group*, 2(12), 10–1571. https://elementallinks.com/el-reports/EventDrivenArchitectureOverview_ElementalLinks_Feb2011.pdf

[394] Schulte, R. W. (2003). The growing role of events in enterprise applications. *Gartner Research*.

на догађаје може варирати у зависности од токова догађаја, њиховог интензитета и начина дистрибуирања[395].

Најзначајнија особина архитектуре оријентисане на догађаје је изразито слаба веза између појединачних догађаја. Извор догађаја нема информације о корисницима, заинтересованим странама, даљој обради догађаја и последицама које догађај може изазвати. Креатор догађаја једино зна да се догађај десио. Као последицу слабе везе између догађаја, ову архитектуру карактерише и могућност једноставног дистрибуирања и повећања скалабилности.

Архитектура софтвера оријентисана на догађаје приказана је на Слици 49.



Слика 49. Архитектура софтвера оријентисана на догађаје[396]

Кључни део архитектуре оријентисане на догађаје је начин имплементације обраде догађаја. Имплементација обраде догађаја није једнозначна и највише зависи од услова рада система и захтева у конкретном случају коришћења.

У случају када се у процесу конзумирања догађаја врши и обрада догађаја, имплементација је једноставна. Уколико је систем дистрибуиран, тражи се висок ниво паралелизације, као и пословна аналитика у реалном времену, и тада се користе сложенији модели архитектуре, који су најчешће подржани посебним, за ту намену креираним софтверским платформама.

У зависности од начина спецификације процеса, интензитета наступања догађаја, времена актуелности догађаја и начина њихове обраде, архитектуре софтвера оријентисаног на процесе можемо поделити на:

[395] Michelson, B. M. (2006). Event-driven architecture overview. *Patricia Seybold Group*, 2(12), 10–1571. https://www.elementallinks.com/2006/02/06/event-driven-architecture-overview/, репринт референце 319 после 5 година од публиковања, са коментарима.
[396] Прилагођено према: https://www.ibm.com/docs/en/developer-for-zos/9.1.1?topic=concepts-what-is-event-processing

1. Архитектуре у којима су процеси спецификовани регуларним догађајима. Регуларни догађаји се одмах након наступања складиште у репозиторијум и обрађују основном обрадом догађаја.
2. Архитектуре у којима су процеси спецификовани токовима регуларних догађаја с високим интензитетом наступања и малим временским интервалима актуелности догађаја. Регуларни догађаји се не меморишу, обрађују се у реалном времену, по групама и у временским прозорима. Резултати обраде регуларних догађаја представљају изведене догађаје. Изведени догађаји се складиште у репозиторијум и даље обрађују статистичким и методама вештачке интелигенције.
3. Архитектуре у којима постоје регуларни догађаји ниског интензитета наступања и дугог периода актуелности, паралелно са интензивним токовима догађаја с малим интервалима актуелности. Обрада таквих догађаја комбинује се серијском обрадом (*Batch*) и обрадом токова догађаја (*Stream*).
4. Архитектуре у којима су процеси спецификовани комуникацијом микросервиса, применом оркестрације или кореографије.

За сваки од наведених модела дефинисани су одговарајући архитектурни патерни који се користе у спецификацији архитектуре, имплементацији и извршењу процеса.

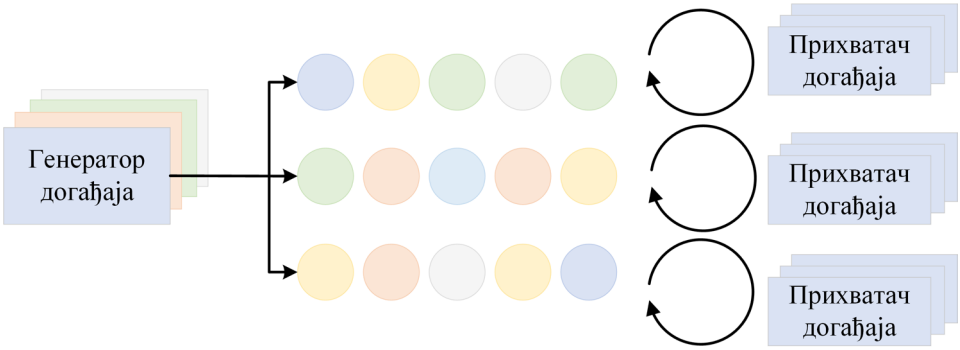
20.5.1 Патерн основне обраде догађаја

Патерн обраде догађаја агрегираних у један процес, с једном листом догађаја, назива се патерном основне обраде догађаја (*Simple Event Processing*). У овом патерну нема експлицитно дефинисаног начина интеракције процеса. Интеракција процеса се реализује преко улазних догађаја који потичу из других процеса.

Обрада догађаја је синхрона и заснива се на коришћењу *Reactor*[397] патерна, а обавља се у оквиру једне нити. Догађаји се обрађују један по један, по редоследу из листе догађаја и извршењем процедуре прихватача догађаја. Након извршења догађаја, скенира се листа догађаја. Уколико нема догађаја у листи догађаја, тада се процес скенирања блокира, тј. нит у којој се скенирање врши. Нит се одблокира када се деси неки улазни догађај и листа догађаја више не буде празна. Блокирање и одблокирање нити реализује се неким од системских средстава конкурентног програмирања (региони, семафори, догађаји и услови).

За сваки процес у оквиру апликације, комплетна обрада, од прихватања догађаја до извршења процедуре прихватача догађаја, извршава се у петљи, у оквиру једне нити (Слика 50). С обзиром на то да се за сваки процес уводи посебна нит, извесна паралелизација је могућа. Већина процесорског времена у овом патерну троши се на извршење процедура прихватача догађаја.

[397] Schmidt, D. C. The Design and Implementation of the Reactor. http://sunsite2.icm.edu.pl/pub/programming/ace/ACE/PDF/Reactor2-93.pdf



Слика 50. Патерн основне обраде догађаја[398]

Значајно побошање перформанси могло би се добити паралелизацијом извршења догађаја. У ту сврху се примењује *Proactor*[399] патерн. Процедуре прихватача догађаја се у овом патерну извршавају асинхроно и у засебним нитима.

Асинхроно извршење догађаја, иако значајно утиче на повећење паралелизма обраде и побољшање перформанси, доводи и до проблема. Догађаји се морају извршавати синхроно и у секвенци. Асинхроно извршавање може довести до тога да се догађај који се касније десио раније обради, тј. изврши. Да би се очувао хронолошки редослед, догађаји се организују у две листе: листу будућих догађаја у коју се смештају догађаји који чекају на извршење, и листу текућих догађаја у коју се смештају догађаји који се асинхроно извршавају. Обе листе догађаја се скенирају. Догађај је коначно извршен тек када се извршење његовог прихватача догађаја заврши и догађај уклони из листе текућих догађаја.

Догађаји из листе текућих догађаја уклањају се по хронолошком редоследу. Иако је процедура прихватача догађаја завршена, догађај неће бити уклоњен из листе текућих догађаја све док постоји догађај с ранијим временом наступања, а извршење његовог прихватача догађаја није завршено. Такви догађаји се називају блокираним догађајима, који чекају на извршење других догађаја како би били уклоњени из листе текућих догађаја.

Свако уклањање елемента листе текућих догађаја иницира поновно скенирање листе будућих догађаја, јер је могуће да је у оквиру процедуре прихватача догађаја генерисан неки нови догађај који може да поремети редослед у листи будућих догађаја. Детаљан опис и графички приказ функционисања овог патерна могуће је наћи у 7. поглављу референце[400].

---

[398] Прилагођено према: https://solace.com/blog/developer-guide-event-driven-development/

[399] Hu, J., Pyarali, I. & Schmidt, D. C. (1998, October). Applying the proactor pattern to high-performance web servers. In: *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems*. https://www.dre.vanderbilt.edu/~schmidt/PDF/proactor-jaws.pdf

[400] Раденковић, Б., Станојевић, М. и Марковић, А. (1999). *Рачунарска симулација*. Факултет организационих наука, Саобраћајни факултет, Београд. https://elab.fon.bg.ac.rs/wp-content/uploads/2018/10/knjiga.pdf

Даља побољшања перформанси патерна основне обраде догађаја могућа су разбијањем јаке зависности између догађаја и прихватача догађаја. То се у имплементацији овог патерна у објектно-оријентисаним окружењима може реализовати применом *Dependency Injection*[401] патерна који омогућава да се у времену извршења програма, догађају додели конкретан прихватач догађаја. Овај приступ побољшању перформанси често се користи у оквирима за развој *Java* и *Android* апликација[402] оријентисаних на догађаје.

Најпознатији софтвер који за свој рад користи *Proactor* патерн у основној обради догађаја је *NginX*[403]. Патерн основне обраде догађаја саставни је део сваког апликационог програмског интерфејса, оквира или платформе оријентисане на догађаје, нпр. *React*[404], *Angular*[405] и други. Патерн основне обраде догађаја значајан је и због тога што представља компоненту завршне обраде догађаја у сваком од сложенијих патерна.

20.5.2 Комуникација и синхронизација процеса – Pub/Sub патерн

Посебан проблем у развоју процесно-оријентисаног софтвера представља имплементација комуникације између процеса. Кардиналност комуникације између процеса може бити 1:1 или 1:*M*. Комуникација 1:1 је једноставна и остварује се разменом догађаја. Излазни догађај једног процеса је улазни догађај другог процеса.

Кардиналност комуникације 1:*M* доводи до необичне ситуације: догађај у једном процесу може бити узрок *M* различитих последица у другим процесима. Последица тога је да репрезентација догађаја, заједно с прихватачем догађаја, није адекватна за овај случај комуникације те се мора тражити архитектонско решење које би омогућило да се појава догађаја раздвоји од прихватача догађаја, којих у овом случају може бити више и сви могу бити различити.

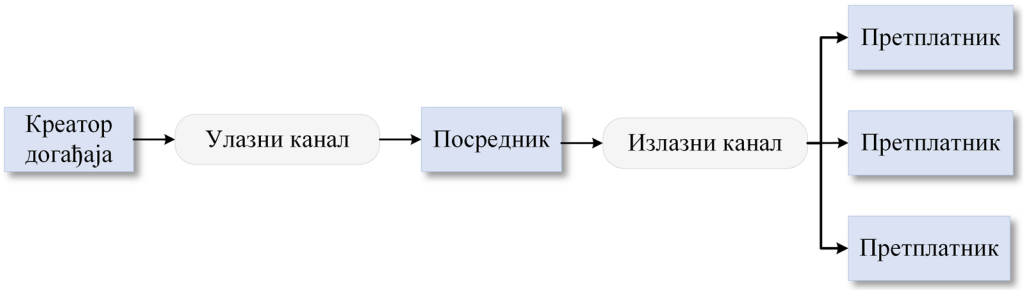
Поред овога, проблем комуникације обично је повезан с проблемом синхронизације, нпр. када један процес мора да сачека да други процес заврши посао, како би наставио рад. За синхронизацију је неопходно увести неке концепте конкурентног програмирања који омогућавају имплицитно или експлицитно блокирање процеса[406].

Publish/Subscribe (*Pub/Sub*) је најчешће коришћен патерн за решење наведених проблема. Заснива се на принципу *receive and react* и представља основну и

[401] https://developer.android.com/training/dependency-injection
[402] https://dagger.dev/
[403] https://www.f5.com/go/product/welcome-to-nginx
[404] https://react.dev/
[405] https://angular.io/
[406] Dijkstra, E. W. (2002). *Cooperating sequential processes* (pp. 65–138). Springer, New York. https://web.archive.org/web/20180722115415id_/https://pure.tue.nl/ws/files/4279816/344354178746665.pdf

рудиментарну имплементацију концепата реактивног програмирања[407]. Први пут се у литератури појавио у форми концепта конкурентног програмирања који је назван „догађај – *event*", уграђен у кернел оперативног система, с базом података која се састоји од индикатора догађаја, листом процеса блокираних на догађају и примитивних операција *waitevent*, *setevent* и *clearevent*. Разрађиван је кроз дефинисање монитора типа коначни бафер као структурног концепта за комуникацију процеса[408] . Касније је овај концепт уграђен у патерн *Observer*. Иако овај патерн има много недостатака који проистичу из његове главне предности – слабљења везе између компоненти[409], пун развој и имплементација овог архитектурног патерна достигнути су у софтверским системима посредницима за дистрибуирану размену порука. Типични представници оваквих софтвера су *RabbitMQ*[410] и *ActiveMQ*[411].

Актери у овом архитектурном патерну, приказани на Слици 51, јесу: процеси креатори догађаја – издавачи (*Publishers*), посредник за размену порука између процеса (*Message Broker*), и процеси конзументи догађаја – претплатници (*Subscribers*).



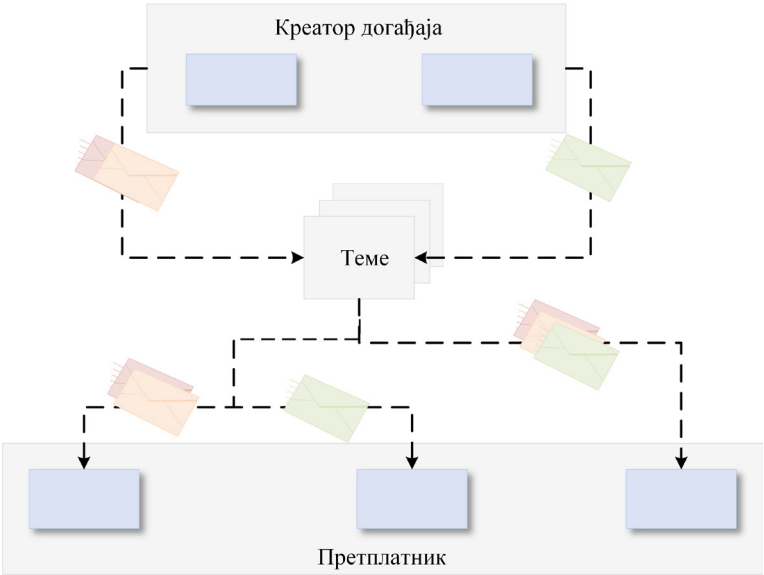
Слика 51. Актери и њихов међусобни однос у Pub/Sub архитектури

Pub/Sub архитектура има за циљ посредовање у размени података између процеса. Јединица размене података је порука (*message*). Порука садржи серијализован слог догађаја у *JSON* или *XML* нотацији.

Област посредовања у размени порука назива се Тема (*Topic*). Тема представља виртуелни канал комуникације процеса. Једна тема обично представља једну врсту догађаја. Процеси своју заинтересованост за теме реализују пријављивањем на *Pub/Sub* посредник. Процеси који креирају догађаје пријављују се као издавачи, а процеси који су конзументи догађаја као претплатници. *Pub/Sub* посредник одржава

[407] Bainomugisha, E., Carreton, A. L., Cutsem, T. V., Mostinckx, S. & Meuter, W. D. (2013). A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4), 1–34.
[408] Hansen, P. B. (2002). The invention of concurrent programming. *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, 3–61.
[409] Köppe, C. (2010, October). Observations on the observer pattern. In: *Proceedings of the 17th Conference on Pattern Languages of Programs* (pp. 1–14).
[410] https://www.rabbitmq.com/
[411] https://activemq.apache.org/

базу података о темама, издавачима и претплатницима. Посредник функционише на тај начин што анализира сваку поруку која пристигне од издавача, и уколико порука припада одређеној теми, прослеђује копију поруке свим процесима који су се пријавили у својству претплатника за ту тему (Слика 52). Овај поступак се назива рутирање порука.

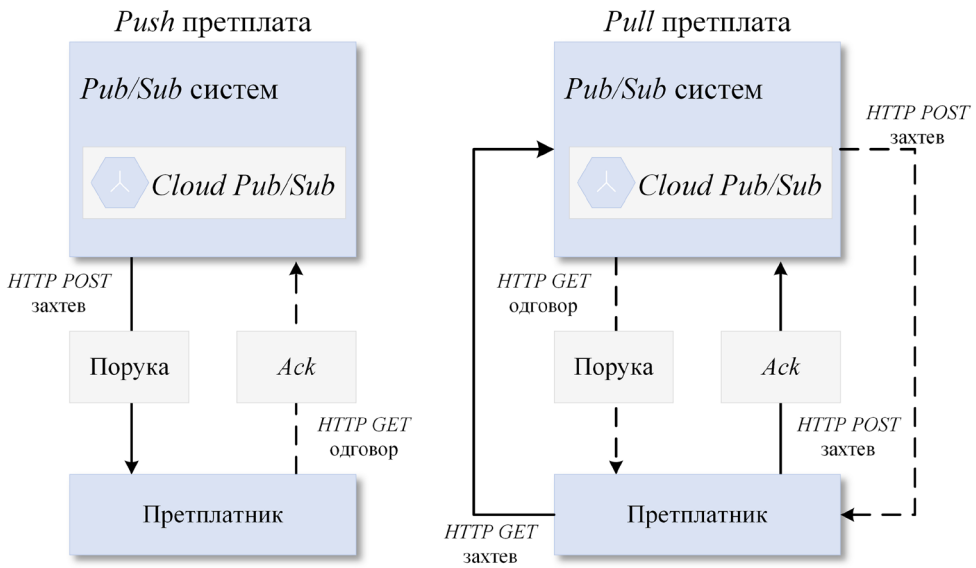


Слика 52. Рутирање порука у Pub/Sub архитектури

Pub/Sub архитектура у себи садржи *FIFO* листе с порукама за сваку тему као и механизме за њихово складиштење и рутирање. *Pub/Sub* архитектура апстрахује и смањује већину проблема конкурентног програмирања који би настали директном комуникацијом процеса. Притом, смањује и скоро у потпуности елиминише чврсту повезаност између процеса који комуницирају, а тиме омогућује боље управљање скалабилношћу.

Лоше особине су везане за латенцију у испоруци порука и могући губитак поруке због грешака у рачунарској мрежи. И поред тога што *Pub/Sub* архитектура баферује поруке и користи *FIFO* листе за разврставање порука по темама, није могуће ускладити велике разлике у брзини генерисања и обраде порука.Да би се ти проблеми ублажили, уводе се два модалитета: *Push* и *Pull* претплате на догађаје (Слика 53).

У системима базираним на *Push* претплати (*push-based systems*), порука се одмах након доласка прослеђује од издавача до претплатника. Претплатник треба да има могућност обраде поруке у реалном времену. Уколико претплатник у неком тренутку не може да обради поруку, долази до проблема који могу довести до тога да се порука изгуби.



Слика 53. Push и Pull претплата на догађаје[412]

Системи базирани на *Pull* претплати (*pull-based systems*), проблеме брзине обраде на страни претплатника решавају тако што *Pub/Sub* посредник баферује поруку и задржава је све до момента док претплатник не буде у могућности да је преузме и обради. Такав приступ доводи до повећања кашњења при испоруци поруке и губитка могућности обраде у реалном времену, али може бити употребљив уколико је потребно неко време за обраду поруке, нпр. слике, али је још увек могуће у неком временском периоду ускладити брзине генерисања и обраде догађаја.

Без обзира на бројне недостатке[413], већина софтверских пакета, оквира, апликационих програмских интерфејса, платформи и екосистема, који су оријентисани на догађаје, има уграђен овај патерн. Програмирање оријентисано на догађаје тешко је замислити без примене овог патерна.

20.5.3 Обрада токова догађаја

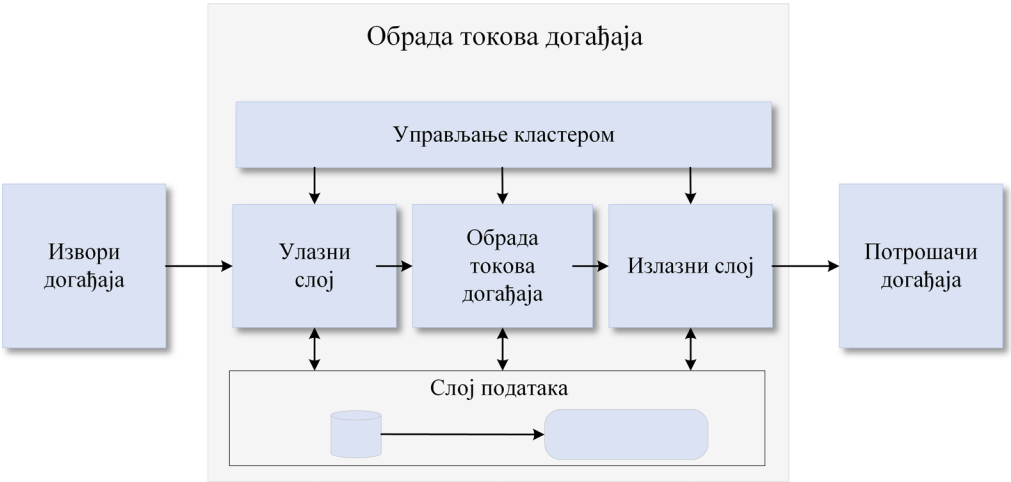
Обрада токова догађаја (*Event Stream Processing*) користи се за обраду у реалном времену када су извори догађаја разнолики и дистрибуирани с високим интензитетом генерисања догађаја. То су најчешће мерења са *IoT* уређаја, кликстримови са веб сајтова, финансијске трансакције и догађаји на друштвеним медијима. Ток догађаја (*event stream*) континуална је и бесконачна серија догађаја.

[412] Прилагођено према: https://levelup.gitconnected.com/event-driven-systems-a-deep-dive-into-pubsub-architecture-39e416be913c

[413] Happ, D. & Wolisz, A. (2016, November). Limitations of the Pub/Sub pattern for cloud based IoT and their implications. In: *2016 Cloudification of the Internet of Things (CIoT)* (pp. 1–6). IEEE.

Догађаји у серији су уређени по времену њиховог настанка. Почетак тока догађаја може бити негде у прошлости, пре него што је почела обрада тока догађаја, а крај тока је негде у непознатом тренутку у будућности. Обрада токова догађаја заснива се на агрегацији, филтрирању и евалуацији догађаја[414].

Улазни догађаји су регуларни догађаји. Акценат у овом приступу је на генерисању изведених догађаја, на основу анализе токова улазних догађаја. У овом моделу се врши пријем и припремна обрада дистрибуираних улазних догађаја (*Ingestion layer*), обрада токова догађаја (*Stream processing layer*) и припрема за рутирање излазних резултата (*Output layer*). Генерисање догађаја и коришћење резултата најчешће су дистрибуиране апликације изван софтвера за обраду токова догађаја. Особина овог патерна је висока паралелизација обраде, која у појединим случајевима може бити подржана и хардверским акцелераторима. Патерн обраде токова догађаја приказан је на Слици 54.



Слика 54. Патерн обраде токова података[415]

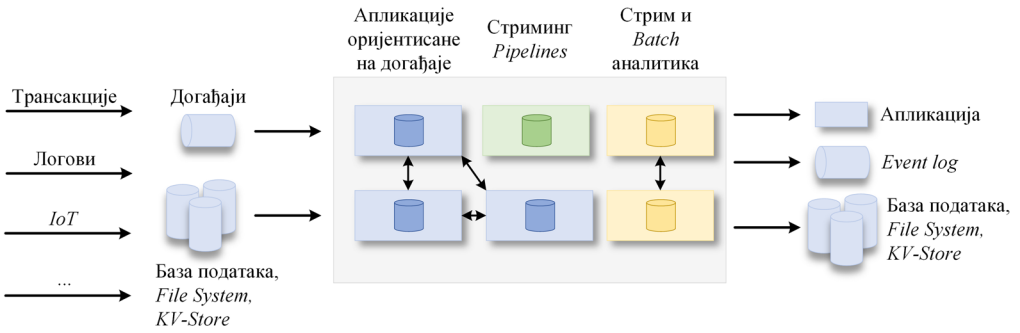
Инфраструктура за овај архитектурни патерн је комплексна и није у потпуности подржана у стандардним програмским језицима, већ се за то користе посебно развијене платформе (*Event Streaming Platform – ESP*)[416,417], које обезбеђују одговарајуће алате за управљање дистрибуираним системом, програмске моделе за обраду токова података, упитне језике и алате за извештавање[418]. Већина провајдера рачунарства у облаку има своје платформе за подршку овој архитектури[419,420].

[414] Cugola, G. & Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3), 1–62.
[415] Прилагођено према: https://dzone.com/refcardz/event-stream-processing-essentials
[416] https://www.ibm.com/cloud/event-streams
[417] https://kafka.apache.org/
[418] https://spark.apache.org/
[419] https://aws.amazon.com/kinesis/
[420] https://www.confluent.io/

Изазов за овај приступ је имплементација агрегације дистрибуираних догађаја који долазе комуникационим каналима различитог кашњења. Догађаји у процесу агрегације треба да се укомбинују у хронолошки уређен низ. Метода која се за ту намену најчешће користи је *Watermarking*[421]. Нарочито код система у реалном времену код којих се генерише велика количина догађаја, изазов је како ускладити брзину улазне обраде (*Ingestion layer*) с брзином генерисања догађаја, а да се при томе обрађује само мањи број најактуелнијих догађаја. Метода која се за ту сврху најчешће користи је *Sliding Windows*[422,423]. За обраду токова догађаја и генерисање излаза најчешће се користе методе и алати пословне аналитике.

20.5.4 Обрада комплексних догађаја

Патерн обраде комплексних догађаја (*Complex Event Processing – CEP*) оријентисан је на откривање знања у динамици догађаја. Поред анализе и обраде токова догађаја, укључује и анализу догађаја из трансакционих база података и фајл система. Регуларни и изведени догађаји у токовима догађаја и трансакционим базама података су основа за откривање комплексних догађаја. Комплексни догађаји се откривају у реалном времену. Посебна компонента у процесору догађаја надгледа појаве регуларних и изведених догађаја, и бави се откривањем шаблона, комбинација и корелација догађаја и њихових атрибута, који међусобно могу имати узрочне временске или просторне везе (*causal*, *temporal*, *spatial*) које нису експлицитно дефинисане. Резултат анализе представља комплексан догађај који представља повод за одговарајућу пословну одлуку или акцију. Обрада комплексних догађаја илустрована је на Слици 55, на примеру *Apache Flink* софтвера.



Слика 55. Обрада комплексних догађаја у Apache Flink[424]

[421] Juma, H., Kamel, I. & Kaya, L. (2008, December). Watermarking sensor data for protecting the integrity. In: *2008 International Conference on Innovations in Information Technology* (pp. 598–602). IEEE.
[422] Shankar, A. U. (1986). A verified sliding window protocol with variable flow control, C*M SIGCOMM Computer Communication Review*, 16(3), 84–91.
[423] https://kafka.apache.org/27/javadoc/org/apache/kafka/streams/kstream/SlidingWindows.html
[424] Прилагођено према: https://flink.apache.org/

20.5.5 Lambda и Kappa архитектуре

Lambda и *Kappa* архитектуре[425,426] представљају савремене архитектуре платформе као сервис рачунарства у облаку. Један од најпознатијих корисника *Lambda* архитектуре је *Netflix*[427]. *Kappa* архитектуру користе *Twitter*, *Uber*[428] и многи други.

Први рад[429] који је описао архитектуру која се данас назива *Lambda* потиче из 1983. године. О потреби унификације разнородних архитектура за обраду комплексних догађаја конкретно се размишља од друге деценије 21. века[430]. За такву архитектуру нешто касније се почео користити термин *Transactional/Analytical Processing* (*HTAP*)[431,432]. Назив *Lambda* је настао у жаргону, из потребе да се на једноставан начин, једним словом грчког алфабета, означи функција трансформације података у резултате упита[433].

Lambda архитектура је намењена обради комплексних догађаја. Подаци у *Lambda* архитектури су систематизовани и налазе се у информационом језеру. Не користе се друге базе података. Подаци су неструктурирани и потичу из различитих извора. Могу потицати из делова информационог система који раде у реалном времену и генеришу интензивне токове догађаја, које треба у реалном времену обрадити. Део података је стационаран, или пак подаци пристижу спорије.

Обрада је систематизована у три слоја: слој за обраду података у реалном времену (*Speed Layer*), слој за серијско извршење програма за обраду података (*Batch Layer*) и слој за генерисање извештаја и пословно извештавање (*Serving Layer*). Слој за серијско извршење програма за обраду података чува главни скуп података у дистрибуираном складишту отпорном на отказе. За обраду података из овог слоја типично се користи *map-reduce* програмски модел. Слој за обраду података у реалном времену обрађује само нове податке који континуално пристижу у токовима. Резултати обраде у реалном времену чувају се кратко, све до њиховог пребацивања у слој за серијско извршење програма за обраду података. Слој за

---

[425] Lin, J. (2017). The lambda and the kappa. *IEEE Internet Computing*, 21(05), 60–66. https://cs.uwaterloo.ca/~jimmylin/publications/Lin_IEEE2017b.pdf

[426] Feick, M., Kleer, N. & Kohn, M. (2018). Fundamentals of real-time data processing architectures Lambda and Kappa. *SKILL 2018-Studierendenkonferenz Informatik*. https://dl.gi.de/bitstream/handle/20.500.12116/28983/SKILL2018-05.pdf?sequence=1&isAllowed=y

[427] https://aws.amazon.com/solutions/case-studies/netflix-and-aws-lambda/

[428] https://www.uber.com/en-IT/blog/kappa-architecture-data-stream-processing/

[429] Lampson, B. W. (1983, October). Hints for computer system design. In *Proceedings of the ninth ACM symposium on Operating systems principles* (pp. 33–48). https://dl.acm.org/doi/pdf/10.1145/800217.806614

[430] N. Marz (2011) How to Beat the CAP Theorem, Thoughts from the Red Planet, blog, 13 Oct. 2011. http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html

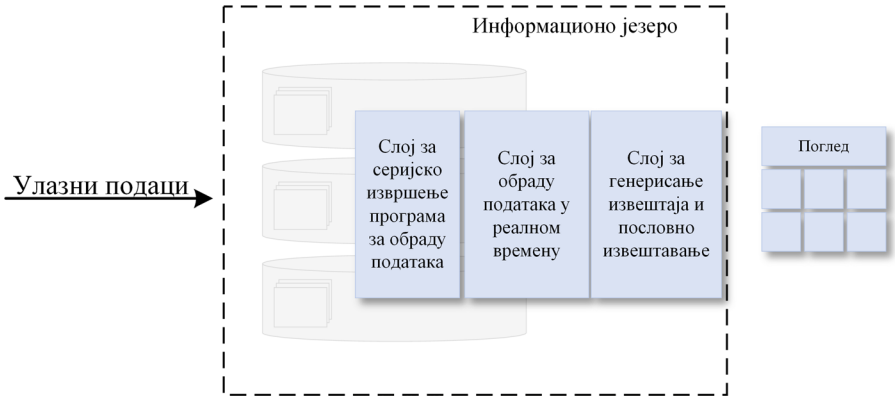
[431] https://www.gartner.com/en/documents/2657815

[432] Özcan, F., Tian, Y. & Tözün, P. (2017, May). Hybrid transactional/analytical processing: A survey. In: *Proceedings of the 2017 ACM International Conference on Management of Data* (pp. 1771–1775). https://pages.cs.wisc.edu/~yxy/cs839-s20/papers/htap-survey.pdf

[433] Lin, J. (2017). The lambda and the kappa. *IEEE Internet Computing*, 21(05), 60-66. https://cs.uwaterloo.ca/~jimmylin/publications/Lin_IEEE2017b.pdf

генерисање извештаја користи резултате обраде слоја за обраду података у реалном времену и слоја за серијско извршење програма за обраду података, зависно од потреба. Раздвајање слоја за обраду у реалном времену од слоја за серијско извршење програма за обраду података омогућује континуално додавање нових података, обезбеђујући притом да постојећи програми за обраду података не буду измењени.

Слојеви *Lambda* архитектуре су приказани на Слици 56.



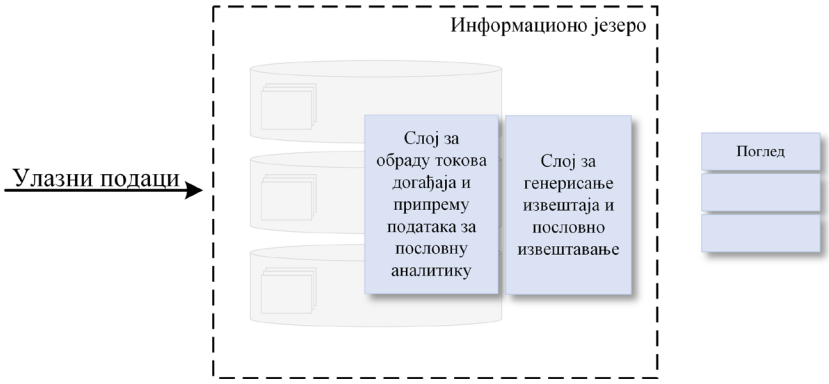
Слика 56. Lambda архитектура

Lambda архитектура сматра се погодном за примене у *big data* системима где су чести *ad hoc* упити, а потребни брзи и поуздани одговори. Недостаци се односе на комплексност и тешкоће у миграцији[434].

Kappa архитектура представља унификацију разних патерна обраде токова догађаја. Једноставнија је за реализацију од *Lambda* архитектуре[435] и садржи само два слоја (Слика 57): слој за обраду података у реалном времену (*Streaming Layer*) и слој за генерисање извештаја и пословно извештавање (*Serving Layer*).

Слој за обраду података у реалном времену обрађује токове догађаја, и припрема податке за пословну аналитику. Слој за генерисање извештаја и пословно извештавање извршава алгоритме пословне аналитике, статистике и вештачке интелигенције. *Kappa* архитектура типично се користи за апликације попут друштвених медија, система за надгледање, *IoT* система и сличних, када је потребна брза обрада текућих података. Одсуство слоја за серијско извршење програма за обраду података може довести до чешћих грешака, па је неопходно имплементирати процедуре за управљање грешкама и изузецима[436].

[434] https://towardsdatascience.com/a-brief-introduction-to-two-data-processing-architectures-lambda-and-kappa-for-big-data-4f35c28005bb
[435] Kreps, J. (2014). *Questioning the Lambda Architecture*, O'Reilly, https://www.oreilly.com/radar/questioning-the-lambda-architecture/
[436] https://towardsdatascience.com/a-brief-introduction-to-two-data-processing-architectures-lambda-and-kappa-for-big-data-4f35c28005bb



Слика 57. Карра архитектура

21. МИКРОСЕРВИСИ

Микросервис је архитектурни стил, скуп препорука и ограничења за реализацију сервиса, дефинисан у настојању да се по могућности реше проблеми у коришћењу и одржавању монолитних апликација и сервисно-оријентисане архитектуре[437].

Примена сервисно-оријентисане архитектуре у монолитним програмским језицима само је делимично решила скуп проблема монолитне архитектуре. Највећи проблеми су модуларност и интеграција. Модули су делови монолитне архитектуре и не могу се дистрибуирати и независно извршавати. До смањења употребне вредности сервисно-оријентисане архитектуре довело је произвољно дефинисање сврхе сервиса, која одступа од основне намене да сервис извршава трансформацију стања само једног ентитета или ресурса[438]. Примена сервиса у објектно-оријентисаним програмима, који *CRUD* операцијама бришу историју, може довести до проблема повезаних с конкурентним приступом подацима.

Са свим тим проблемима су се софтверски инжењери свакодневно суочавали у пракси. Да би се број проблема смањио, а предности сервисно-оријентисане архитектуре што више искористиле, у литератури се може наћи низ радова с препорукама, од начина како да се методама које су примерене објектно-оријентисаном развоју софтвера препознају делови система који могу бити кандидати за ефикасне сервисе[439], па све до предлога новог архитектурног стила

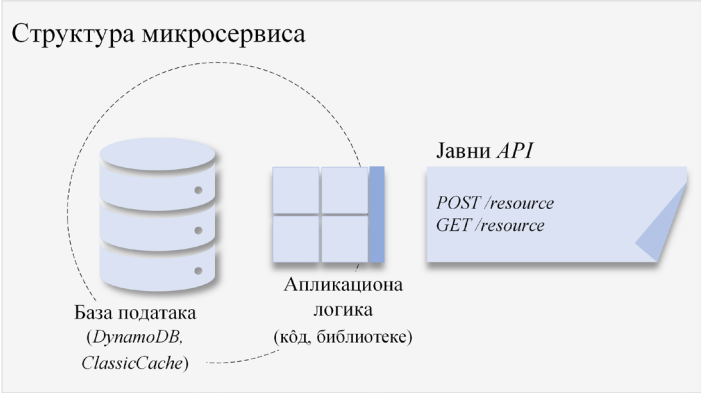
[437] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. & Safina, L. (2017). Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, 195–216. https://arxiv.org/pdf/1606.04036.pdf

[438] Xiao, Z., Wijegunaratne, I. & Qiang, X. (2016, November). Reflections on SOA and Microservices. In: *2016 4th International Conference on Enterprise Systems* (ES) (pp. 60–67). IEEE.

[439] Stojanovic, T. D., Lazarevic, S. D., Milic, M. & Antovic, I. (2020). Identifying microservices using structured system analysis, *24th International Conference on Information Technology* (IT), Zabljak, Montenegro, 2020, pp. 1–4, doi: 10.1109/IT48810.2020.9070652

названог микросервис и новијег концепта рачунарства у облаку: функција као сервис[440].

Термин микросервис[441] први пут је уведен на конференцији о архитектури софтвера 2011. године. Микросервисни архитектурни стил није стандардизован, већ представља скуп препорука и ограничења за реализацију и повезивање микросервиса. Примерен је за дистрибуиране апликације. Предложена структура микросервиса приказана је на Слици 58.



Слика 58. Структура микросервиса

Миграција ка микросервисној архитектури се мотивише скалабилношћу, одрживошћу, независним тимовима за делове апликације и лакшим и чешћим имплементацијама. Микросервисне архитектуре подржавају модуларизацију и дистрибуираност, а микросервиси се могу користити и за дефинисање архитектуре интеграције.

21.1 Комуникација микросервиса са окружењем

Микросервис је самостална, независна, извршна програмска јединица чије се извршење иницира позивом преко апликационог програмског интерфејса. Окружење из којег долази позив може бити други микросервис, посредник, процес или апликација.

При синхроној комуникацији између микросервиса и његовог окружења користе се синхрони протоколи као што су *REST*, *gRPC* и други. Поруке се преносе стандардним *http* протоколом за мрежну комуникацију, методама овог протокола: *POST*, *GET*, *PUT*, *PATCH* и *DELETE*. Након позива микросервиса, позивалац зауставља рад и остаје у блокираном стању све док се микросервис не изврши. Уколико је време извршења микросервиса кратко, то се може толерисати. Међутим,

[440] Baresi, L. & Garriga, M. (2020). Microservices: The evolution and extinction of web services?. *Microservices: Science and Engineering*, 3–28.
[441] Fowler M. & Lewis J. (2014) Microservices, https://martinfowler.com/articles/microservices.html

уколико микросервису треба дуже времена да се изврши, тада је боље изабрати асинхрону комуникацију. Одржавање апликације с много синхроно повезаних микросервиса веома је тешко, па се и у таквим случајевима препоручује примена неког од метода асинхроне комуникације.

Патерни асинхроне комуникације омогућавају изоловање микросервиса у јединицу која ради асинхроно, а која се може независно развијати, тестирати и постављати у продукционо окружење. Реализује се тако што се између микросервиса и његовог окружење поставља *Pub/Sub* посредник за размену порука (нпр. *RabbitMQ*).

Микросервис комуницира са окружењем искључиво преко апликационог програмског интерфејса (*API*). Комуникација разменом података преко заједничких слогова у бази података се не препоручује. Микросервис за функционисање користи своју базу података или искључиво своје слогове у заједничкој бази података, којима окружење нема приступ.

Посебну пажњу је потребно посветити прилагођавању *API* спецификације намени микросервиса и њиховом унифицирању за групе сервиса сличних намена. Уколико се спецификације не унифицирају, након неког времена и више ажурираних верзија микросервиса могу се појавити несагласности при позивању микросервиса и његове *API* спецификације. У таквим случајевима појављују се грешке, што отежава одржавање[442].

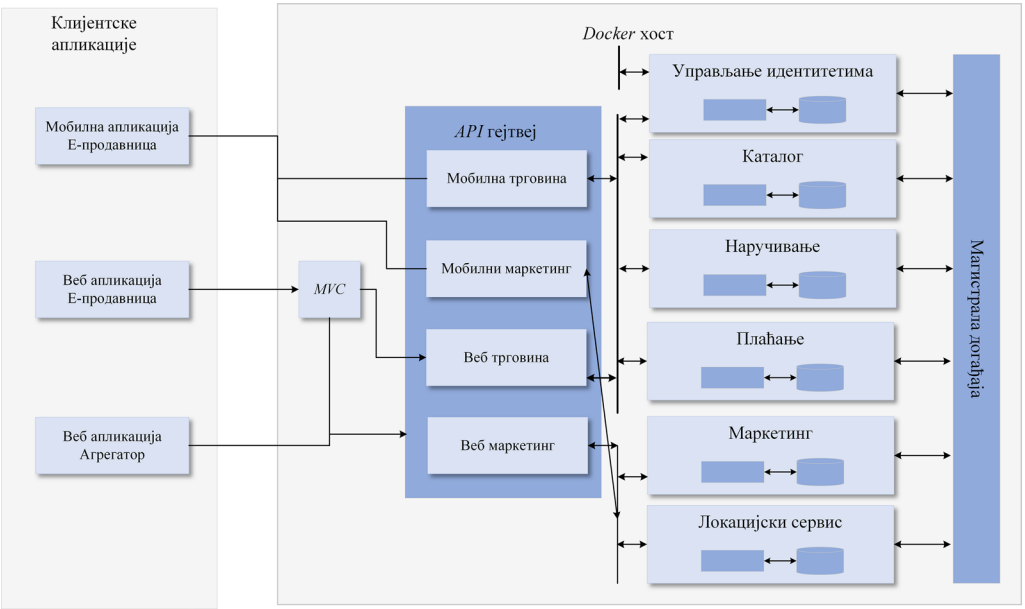
21.2 Апликациона логика микросервиса

Микросервис је мала апликација која има само једну сврху или намену, и може се поставити у продукционо окружење, независно скалирати и тестирати. Намена микросервиса за једну сврху је важна јер повећава кохезију микросервиса. При одржавању је потребан само један разлог за промену и ажурирање микросервиса, а такође само један разлог за његову замену.

Поред основне намене, да својом функционалношћу мења стање неког ресурса или ентитета, намена микросервиса може да буде захтев за одређеном функционалношћу, извршење једног мањег дела пословне логике или неког нефункционалног захтева[443]. Пример апликације у микросервисној архитектури приказан је на Слици 59.

[442] Schneider, S., & Scandariato, R. (2023). Automatic extraction of security-rich dataflow diagrams for microservice applications written in Java. *Journal of Systems and Software*, 202, 111722. https://arxiv.org/pdf/2304.12769.pdf

[443] Thönes, J. (2015). Microservices. *IEEE software*, 32(1), 116–116. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7030212



Слика 59. Пример апликације у микросервисној архитектури[444]

Микросервиси се пројектују као независне јединице које се једноставном оркестрацијом и кореографијом могу интегрисати у процесе који чине апликацију. Развијају се, тестирају и постављају у продукционо окружење независно, уз висок ниво аутоматизације у развоју и одржавању, применом *DevOps* метода и алата.

У микросервисним архитектурама, контејнери представљају стандардно продукционо окружење. Примена контејнера омогућава гранулацију модуларности од једног сервиса по контејнеру, без губитка на перформансама. Тиме је омогућена паралелизација и скалирање перформанси микросервиса у реалном времену.

21.3 Типови микросервиса

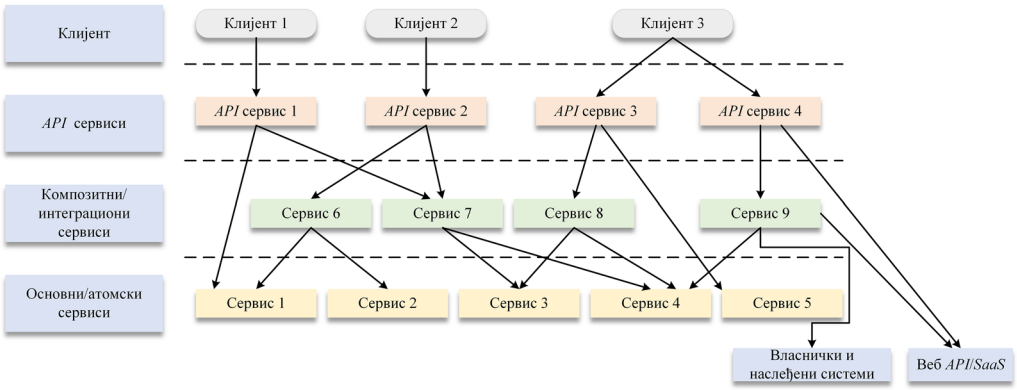
Микросервисне архитектуре се могу класификовати у више типова сервиса, категоризованих по слојевима, функционалности и грануларности[445]:

1. Основни/атомски сервиси.
2. Композитни/интеграциони сервиси.
3. *API* сервиси.

Слојевита архитектура микросервиса с различитим типовима микросервиса приказана је на Слици 60.

[444] Прилагођено према: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/microservice-application-design

[445] Indrasiri, K., & Siriwardena, P. (2022). *Microservices for the Enterprise: Designing. Developing, and Deploying*, USA: Apress.



Слика 60. Микросервисна слојевита архитектура[446]

Основни/атомски сервиси

На доњем слоју се налазе грануларни, самостални сервиси, који немају екстерне зависности, а који се углавном састоје од пословне логике и врло мало, или чак нимало, логике мрежне комуникације.

Већина постојећих технологија се фокусира на изградњу ових сервиса. Стога су доступни бројни оквири и библиотеке за имплементацију ове врсте сервиса (нпр. *SpringBoot*).

Композитни/интеграциони микросервиси

Атомски микросервиси не могу се често мапирати у функционалности пословне логике јер су исувише грануларни. Стога је неретко случај да специфична функционалност може захтевати композицију више атомских сервиса. Средњи слој предложене архитектуре састоји се од таквих композитних или интеграционих микросервиса. Ови сервиси често морају да подржавају значајан део *ESB* функционалности као што су рутирање, трансформације, оркестрације итд.

Композитни/интеграциони микросервиси су такође грануларни, али су елементи од којих се састоје „крупнији“ у односу на атомске сервисе. Они садрже пословну логику (рутирање, информације о томе које сервисе би требало позвати, логику за мапирање типа података итд.) и логику мрежне комуникације (комуникација између сервиса путем различитих протокола). Код композитних микросервиса сервиса, важно је стриктно раздвојити сервисе који имплементирају пословну логику, од сервиса за интеграцију који врше њихово повезивање.

Већина интеграционих послова реализују се у архитектури микросервиса на

[446]Прилагођено према: https://medium.com/microservices-in-practice/microservices-layered-architecture-88a7fc38d3f1

слоју композитних/интеграционих сервиса. Овакви сервиси се такође користе да премосте друге застареле и власничке системе (нпр. *SAP ERP*), спољне веб *API*-је (нпр. *Salesforce*), дељене базе података итд.

У већини постојећих имплементација микросервиса користи се исти скуп технологија за изградњу, и атомских и композитних сервиса. Међутим, већина оквира и платформи намењених развоју микросевиса није пројектована за изградњу сервиса композитног типа. Имплементација овог типа сервиса је најизазовнији задатак у реализацији микросервисне архитектуре – велика количина посла је потребна за имплементацију различитих интеграционих патерна од нуле.

Да би се превазишао проблем сложене имплеметације композитних сервиса, постоје нове технологије као што је *Ballerina*[447] програмски језик дизајниран за интеграцију сервиса и мрежне интеракције, или алати за који имплементирају *API* гејтвеј и *Service Mesh* патерн. *Service Mesh* пружа могућност да се мрежна комуникација пребаци на мрежу *API* гејтвеја, тако да се фокус микросервиса може усмерити само на функционалности и пословну логику.

API сервиси

Композитни микросервиси се користе за обезбеђење функционалних и нефункционалних захтева. Могу комуницирати с микросервисима који се налазе на више различитих слојева архитектуре, инфраструктурном, доменском и апликационом слоју. *API* таквих сервиса може бити компликован. Да би се окружењу и другим сервисима пружио чист и јединствен начин приступа, који се фокусира само на функционалности, уводи се слој *API* сервиса.

API сервиси обједињују доменску и апликациону логику композитних сервиса са архитектуром за интеграцију и обезбеђивање нефункционалних захтева. Суштина примене *API* сервиса је енкапсулација стриктно раздвојених сервиса за нефунционалне захтеве и микросервиса пословне логике.

API сервиси су обично фокусирани на минималну доменску или апликациону логику. Велики број пројектаната и програмера при имплементацији микросервиса покушава да стави огромну количину пословне логике у један микросервис, задржавајући притом монолитну архитектуру. Тиме се нарушавају основни принципи модуларности и микросервисне архитектуре, што доводи до озбиљних проблема у експлоатацији. Постоје два могућа приступа за превазилажење ових проблема:

1. Пребацивање интеграције на слој композитних сервиса, а монолитни гејтвеј се може користити само за њихово излагање у виду *API*-ја.
2. Примена микрогејтвеј концепта, тј. подела времена извршавања *API* гејтвеја тако да се сваки *API* сервис може имплементирати и управљати независно.

[447] https://ballerina.io/

Исте технологије које се користе за изградњу композитних сервиса такође се могу користити и у развоју *API* сервиса. Већина постојећих (монолитних) решења за управљање *API*-јем и гејтвејем још увек еволуирају у правцу подршке концептима као што су микрогејтвеји.

21.4 Примена микросервиса у развоју софтвера оријентисаног на процесе

Микросервиси се у архитектури оријентисаног на процесе користе за имплементацију догађаја и активности/сервиса, дефинисаних при моделовању пословних процеса. Један микросервис може извршавати један или више догађаја.

Композитни микросервиси се користе за повезивање микросервиса у процесе, тако да се комплетан софтвер оријентисан на процесе може реализовати применом микросервисне архитектуре.

REST API је у великој мери усмерио коришћење микросервиса искључиво на промену стања ресурса или ентитета. Активности и сервиси су агрегације догађаја који мењају стање неког ресурса или ентитета, тако да микросервис представља идеалан концепт за имплементацију активности у оквиру пословног процеса.

Обрада у микросервису организује се као секвенцијално повезан скуп свих догађаја који мењају стање ресурса на који се односи микросервис. Догађаји које извршава микросервис су увек нови, тако да не постоји могућност било каквог паралелног приступа или спреге преко података с другим инстанцама истог или различитог типа микросервиса. Тиме се избегавају проблеми конкурентног програмирања.

Након извршења микросервиса, сви подаци и метаподаци о догађајима које је микросервис извршио, трајно су меморисани у бази знања, и расположиви су за будућу анализу и извештавање. Извршење микросервиса даје потпуно исти резултат као да су се догађаји у оквиру микросервиса, један по један, независно извршавали.

Рационала увођења микросервиса огледа се у томе да се агрегацијом обраде избегавају операције узимања догађаја са листе догађаја, анализе метаподатака и позивање процедуре прихватача догађаја, већ се сви прихватачи догађаја одмах и у континуитету извршавају. У случају активности с више догађаја, значајно се може убрзати извршење.

Изоловање микросервиса у јединицу која ради асинхроно, која се може независно развијати, тестирати и постављати у продукционо окружење, реализује се тако што се у архитектури оријентисаној на догађаје микросервиси раздвајају и постављају у улоге извора и конзумента догађаја, који су повезани преко *Pub/Sub* посредника.

Pub/Sub посредник се може користити за кореографију и оркестрацију микросервиса у реалном времену, повезујући микросервисе каналима. Тиме је омогућено ажурирање кореографије микросервиса у реалном времену и континуални развој,

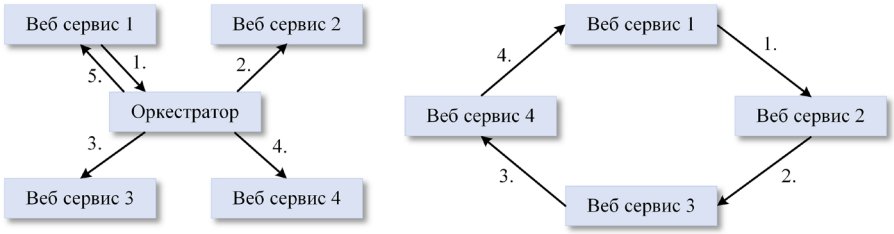
испорука и постављање апликације у продукционо окружење.

У софтверу оријентисаном на процесе, асинхрона комуникација микросервиса преко *Pub/Sub* посредника представља уобичајен и стандардан модел агрегације догађаја и активности у процес. *Pub/Sub* посредник је описан у претходним поглављима ове књиге.

Доследна примена микросервиса у архитектурама оријентисаним на процесе доноси многе предности: једноставно дистрибуирање коришћењем клијент-сервер архитектуре, једноставна паралелизација, повећање скалабилности и поузданости, избегавање проблема конкурентног програмирања, асинхроно извршење микросервиса, могућност развоја и тестирања независно од продукционог окружења, једноставна модуларизација у оквиру контејнера, и лако укључивање у *CI/CD DevOps* токове[448].

21.5 Кореографија и оркестрација микросервиса

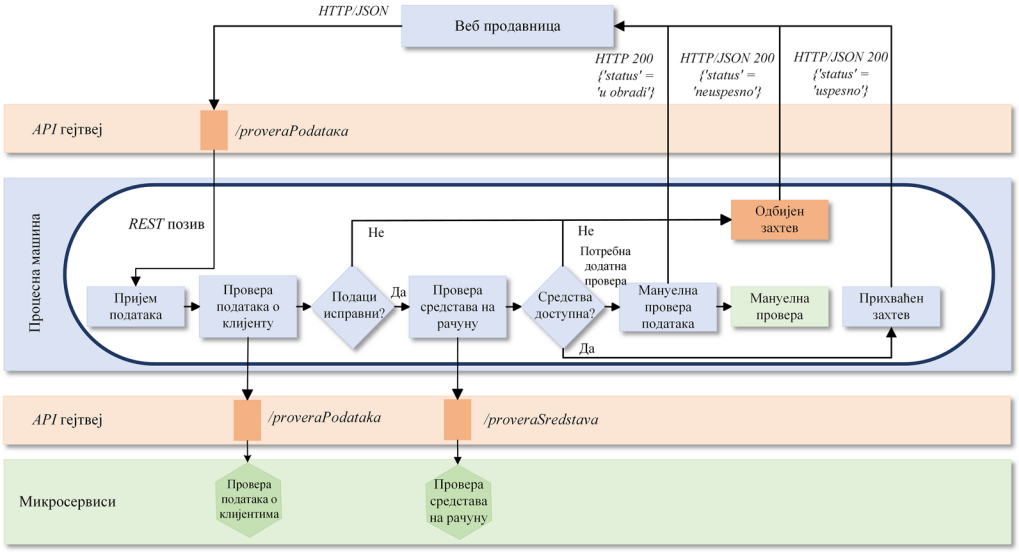
Да би се спецификовали процеси и апликације у процесно-оријентисаном развоју софтвера, неопходно је дефинисати начине организације и интеграције микросервиса који чине процес. Организација и интеграција микросервиса у процесе може се реализовати применом поступака који се називају: оркестрација и кореографија (Слика 61).



Слика 61. Оркестрација и кореографија микросервиса

Оркестрација је поступак у коме се дефинише редослед позивања микросервиса да би се реализовао процес од почетка до краја. Апликација се добија хијерархијским повезивањем микросервиса, где један сервис, оркестратор, има задатак да позива подређене сервисе по неком редоследу и тиме реализује процес. Оркестрација се реализује преко описа секвенцијалног позивања микросервиса у некој нотацији, као што је нпр. *BPEL*. Алати који се користе за оркестрацију микросервиса укључују *Zeebe*, *Netflix Conductor*, *Camunda*, *ING Baker*, *Uber Cadence*, *Pinterest Pinball*, *Azkaban*[449]. Пример оркестрације, тј. комбиновања микросервиса у пословни процес, приказан је на Слици 62.

[448] https://www.developer.com/design/event-driven-microservices/
[449] Valderas, P., Torres, V. & Pelechano, V. (2020). A microservice composition approach based on the choreography of BPMN fragments. *Information and Software Technology*, 127, 106370.



Слика 62. Компоновање сервиса у пословне процесе – синхрона оркестрација[450]

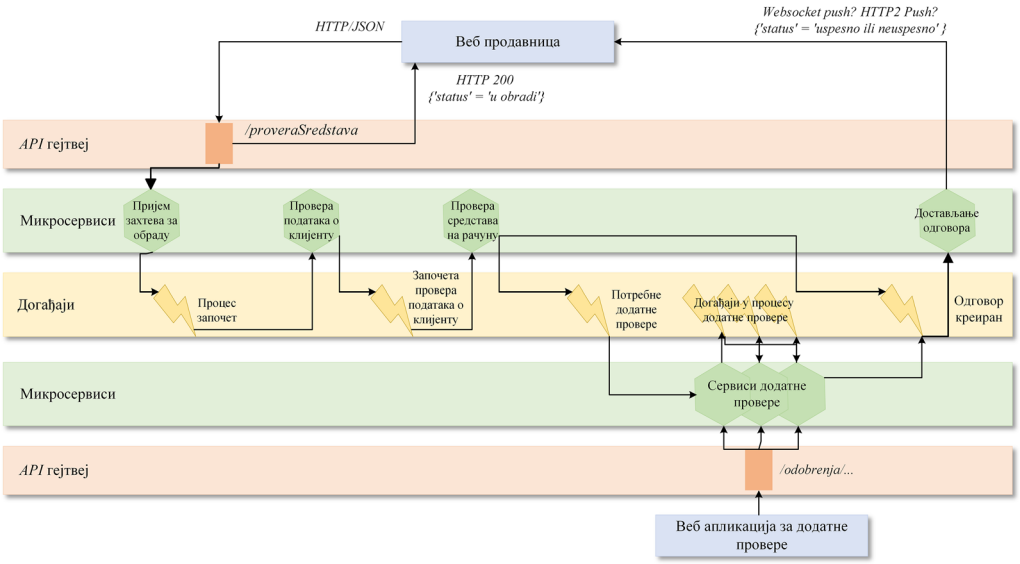
Кореографија је поступак за организацију и интеграцију сложеног пословног процеса мрежним повезивањем микросервиса и описом улоге коју сваки од микросервиса има у процесу. Кореографијом се дефинишу обрасци и редослед за размену порука (позива) између микросервиса. Учесници у кореографији треба да буду обавештени о процесима, операцијама које треба да изврше, порукама које треба да размене и о времену размене тих порука[451]. Предност примене кореографије је дистрибуирано управљање пословним процесом, проширивост и могућност комбиновања микросервиса из различитих пословних домена. Главни недостатак примене кореографије је у чињеници да нема управљачког места с којег се води рачуна о глобалном стању тока пословног процеса. Дистрибуирано управљање није једноставно и тешко је за отклањање грешака, за мерење перформанси, као и за примену *DevOps* праксе континулане интеграције и испоруке (*CI/CD*).

Да би се ови недостаци кореографије барем делимично уклонили, боље је користити *Pub/Sub* патерн уместо чврстих веза (*http* позива) између појединачних микросервиса. Такође, код комплексних апликација, неопходно је да се кореографија аутоматизује пресликавањем модела пословних процеса на микросервисе. Преглед приступа за кореографију микросервиса може се наћи у раду Валдареса и коаутора[452].

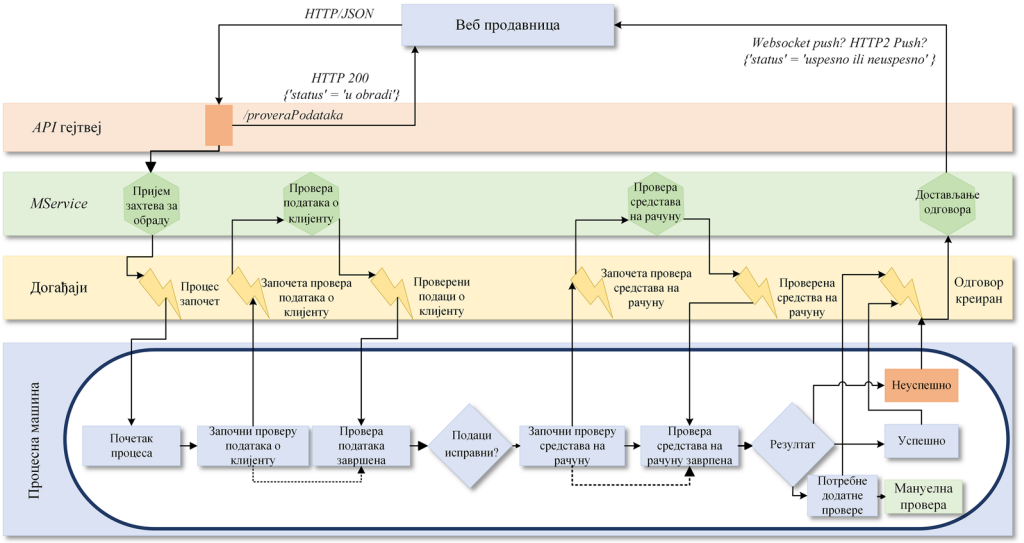
Потреба за аутоматизацијом извршавања пословних процеса захтева и стандарде и специјализован језик за композицију сервиса у пословне процесе. Језик треба да обезбеди могућност представљања пословних процеса на стандардизован

[450] Прилагођено према: https://www.soa4u.co.uk/2018/02/is-bpm-dead-long-live-microservices.html
[451] Лукић, J. *Интероперабилност Б2Б система заснована на процесно оријентисаном развоју софтвера* (магистарска теза), ФОН, 2010.
[452] Valderas, P., Torres, V. & Pelechano, V. (2020). A microservice composition approach based on the choreography of BPMN fragments. *Information and Software Technology*, 127, 106370.

начин коришћењем општеприхваћене нотације. Најбитнији део, или срце система, представља процесна машина, која извршава пословни процес написан у *BPEL* или неком другом језику извршне спецификације. Примери кореографије без и с применом процесне машине приказани су на Слици 63 и Слици 64.



Слика 63. Компоновање микросервиса у пословне процесе – асинхрона кореографија без процесне машине[453]



Слика 64. Компоновање микросервиса у пословне процесе – асинхрона кореографија с процесном машином[454]

[453] Прилагођено према: https://www.soa4u.co.uk/2018/02/is-bpm-dead-long-live-microservices.html
[454] Прилагођено према: https://www.soa4u.co.uk/2018/02/is-bpm-dead-long-live-microservices.html

21.6 Управљање подацима у микросервисним архитектурама

Главна одлика микросервисне архитектуре је стриктна енкапсулација података и избегавање дељења локалних података микросервиса (*stateless architecture*). По стриктној енкапсулацији података, микросервис је сличан објектима. За разлику од објеката, чијим подацима се приступа преко метода, подацима микросервиса приступа се преко апликационог програмског интерфејса, који омогућава дистрибуирану комуникацију микросервиса са окружењем.

Микросервис добија податке из окружења и обрађује их искључиво коришћењем *API*. Микросервиси не деле међусобно податке из својих локалних база података. Иако се у оквиру микросервисних архитектура многи проблеми конкурентног програмирања тиме избегавају, не значи да ти проблеми не постоје, већ су измештени у окружење, односно изван микросервиса[455].

Потребно је нагласити разлику између стања сесије микросервиса и стања ресурса које се мења извршењем сервиса, јер се ти појмови често мешају. Микросервиси немају подршку за чување стања сесије између позива микросервиса (*stateless session*), тако да праћење стања сесије спада у надлежност клијентске апликације.

С друге стране, промена стања ресурса врши се у оквиру микросервиса и представља његову примарну функцију. Ресурси информационог система предузећа су глобално доступни. У случајевима када микросервис врши обраду догађаја или промену стања ресурса, проблеми конкурентног програмирања у оквиру микросервисне архитектуре не могу се избећи.

Промене стања ресурса могу се меморисати и пратити на два начина: да се памти само агрегирано стање ресурса, након последње промене стања (*stateful resource*), или да се памти комплетна историја промена стања, при чему је свако стање након извршене промене меморисано као део историје промена, и за даљу обраду је непроменљиво (*stateless*, *read-only resorce*).

CRUD операције над *stateful* ресурсима доводе до проблема с конкурентним програмирањем, паралелизацијом, скалабилношћу и не препоручују се у микросервисним архитектурама. Преферирани начин је праћење историје промена стања (*stateless*, r*ead-only resorce*).

Да би се проблеми конкурентног програмирања свели на најмању меру, нужно је у микросервисним архитектурама, у случајевима када је то могуће, обраду података глобално доступних ресурса свести на обраду податка који су били валидни у једном тренутку, и сву обраду референцирати на тај тренутак. То се постиже праћењем и меморисањем комплетне историје промена. Промене података се дешавају у дискретним тренуцима времена. Након сваке промене, измењени податак постаје

[455] Luo, S., Lin, C., Ye, K., Xu, G., Zhang, L., Yang, G., ... & Xu, C. (2024). Optimizing Resource Management for Shared Microservices: A Scalable System Design, *ACM Transactions on Computer Systems*, 42(1–2), 1–28. https://dl.acm.org/doi/pdf/10.1145/3631607

део историје, надаље се не мења, постаје *stateless* и *read-only*. Праћењем историје промена и достављањем микросервису на обраду *read-only* глобално доступних историјских података, микросервисне архитектуре постају *stateless* и ослобођене су многих проблема конкурентног програмирања.

У току извршења микросервиса, генерише се ново стање ресурса које се меморише у базу података као нови слог историје промена стања ресурса. Након меморисања, то ново стање је *read-only*. Једина атомска операција и критични регион у микросерверским архитектурама је меморисање и складиштење последње промене стања ресурса, које траје врло кратко. У савременим базама података упис новог податка је по правилу атомска операција, па пројектанти и програмери о томе не морају водити рачуна, и ту операцију могу безбедно уградити у микросервис. Недостатак овог приступа је повећано коришћење капацитета базе података, који је у великој мери компензован могућношћу коришћења историје промене стања у сврху пословне аналитике.

Микросервис, поред приступа глобално доступним ресурсима, може имати и своју локалну базу података. Подаци у локалној бази података валидни су само за време извршења микросервиса. Након извршења микросервиса, локална база података се брише. Захтеви за перформансама локалне базе података микросервиса нису тако строги. Локалну базу података користи само један микросервис, тако да нема конкурентног приступа подацима. За ту сврху могу се користити и једноставније базе података, као што су *MySQL* или *MongoDB*.

21.6.1 CAP теорема

За складиштење глобално доступних података у микросервисним архитектурама, могу се користити и релационе базе података, али се тада губе могућности дистрибуираности и паралелизације микросервиса, па се микросервиси морају извршавати у монолитним архитектурама.

Да би се то избегло, у микросервисним архитектурама најчешће се користе дистрибуиране *NoSQL* базе података и репозиторијуми. У случају да је потребна паралелизација микросервиса на више географских локација, користи се дистрибуирана база података, која обезбеђује читање текућег стања ресурса (које је *read-only*) у сваком тренутку, на сваком месту и за све сервисе.

Дистрибуирање база података реализује се географском репликацијом. За репликацију и синхронизацију података потребно је извесно време, које се назива латенцијом. Латенција је већа уколико је веће географско растојање између сервера на којима су постављени микросервиси. Чак и када су локације повезане брзим оптичким везама, потребно је више милисекунди, чак и више секунди да се подаци пренесу до удаљене локације. То значи да микросервиси који раде паралелно, након промене неког податка, у периоду који је дефинисан латенцијом, приступају различитим верзијама тог податка. Имплементација модела конзистентности

података у том случају може бити комплексна, јер је за очување *ACID* особина трансакција неопходна координација између микросервиса.

Ерик Бруер је у свом предавању о дистрибуираним системима 2000. године[456] формулисао *BASE* карактеристике трансакција у *NoSQL* базама података (*Basically Available*, *Soft-state*, *Eventual consistency*), којима се, насупрот *ACID* карактеристикама у релационим базама података, захтеви конзистентности и изолације апстрахују, а приоритет има висока доступност. У овом контексту је предложио тзв. *CAP* теорему, којом тврди да се у дистрибуираним *NoSQL* базама података не могу истовремено реализовати захтеви конзистентности (*consistency*), високе доступности (*availability*) и отпорности на отказе појединачних компоненти (*partition tolerance*[457]), што је илустровано Сликом 65.

Имајући у виду да се у дистрибуираним системима отпорност на отказе појединачних компоненти јавља као главни захтев, приликом пројектовања мора се одабрати конзистентност или висока доступност као важнији захтев, а затим на неки начин решавати проблеми настали због изостанка оног који није изабран. На пример, у системима код којих је приоритетна висока доступност, конзистентност се не може постићи у реалном времену, већ с латенцијом у коначном времену (*eventual consistency*). Развојем *NewSQL* база података, као што је *Гуглов Spanner*[458], учињен је корак ка развоју скалабилних, глобално дистрибуираних база података које имају подршку за дистрибуиране трансакције које задовољавају *ACID* услове. Иако овакве базе задовољавају превасходно *CA* захтеве, оне су дистрибуиране, па се сматрају системима који у високој мери, али не у потпуности, задовољавају сва три услова *CAP* теореме[459].

Проблеме конзистентности података код дистрибуираних микросервиса морају узети у обзир пре свега пројектанти, потом и програмери, како би механизме за обезбеђење конзистентности уградили у архитектуру система. Део проблема може се решити применом одговарајућих патерна за управљање трансакцијама у микросервисним архитектурама.

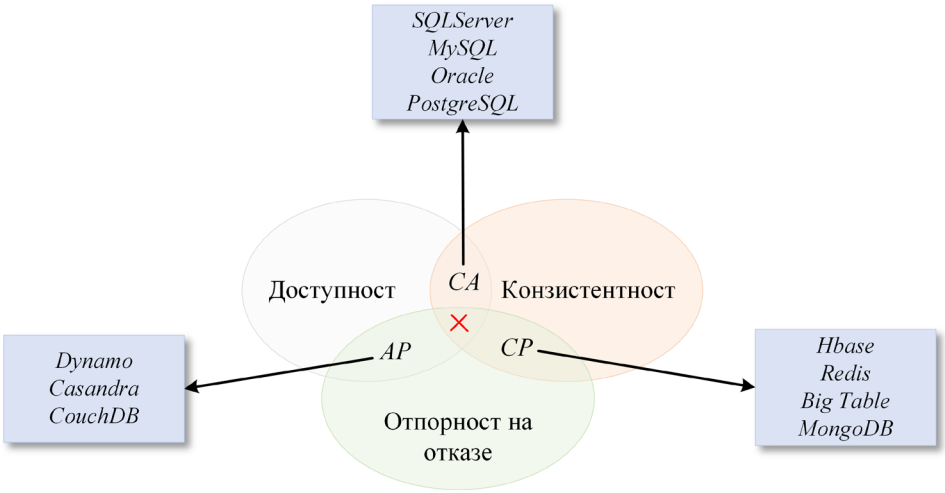
---

[456] Brewer, E. A. (2000, July). Towards robust distributed systems. *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, Vol. 7, No. 10.1145, pp. 343477–343502).

[457] Израз *partition tolerance* односи се на способност система да настави рад, упркос отказима појединих делова система.

[458] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., ... & Woodford, D. (2013). Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3), 1–22.

[459] Brewer, E. (2017). Spanner, truetime and the CAP theorem, https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45855.pdf



Слика 65. CAP теорема

21.7 Управљање трансакцијама у микросервисним архитектурама

У микросервисним архитектурама, типични патерни за управљање дистрибуираним трансакцијама су: Двофазни протокол извршавања трансакција и Сага патерни.

21.7.1 Двофазни протокол извршавања трансакција – 2PC

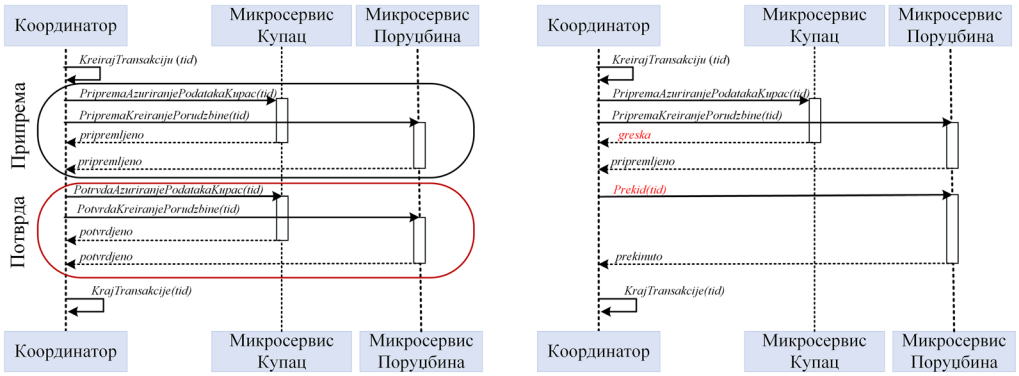
Двофазни протокол извршавања трансакција, познат и као двофазни комит (*2PC – two phase commit*) патерн, заснива се на теорији да постоји један процес који има улогу координатора извршења трансакације[460]. Током извршења трансакције, остали процеси комуницирају искључиво с координатором. Трансакције и процеси имају глобалне идентификаторе.

Трансакција почиње када координатор од корисничке апликације добије захтев за извршењем трансакције. У првој фази координатор, процесима који извршавају појединачне делове дистрибуиране трансакције, шаље поруку о захтеву за трансакцијом. Тек када координатор добије потврду од свих процеса који учествују у извршењу трансакције да је могуће извршити трансакцију, прелази се на другу фазу. У другој фази, координатор шаље поруку појединачним процесима да изврше свој део трансакције.

Уколико било који процес не може да изврши свој део дистрибуиране трансакције, информацију о томе ће доставити у првој фази *2PC* патерна. У том случају, трансакција неће бити извршена, а претходно заузети ресурси ће бити ослобођени.

[460] Mohan, C. & Lindsay, B. (1985). Efficient commit protocols for the tree of processes model of distributed transactions. *ACM SIGOPS Operating Systems Review*, 19(2), 40–52.

Примери успешног и неуспешног извршења трансакције приказани су на Слици 66.



Слика 66. 2PC патерн – успешно и неуспешно извршавање[461]

У оквиру сваког процеса чувају се логови који садрже: идентификаторе процеса, координаторе, трансакције и информације о ресурсима које трансакција користи. Приликом сваке трансакације креирају се два записа, по један за сваку фазу протокола.

Као главне предности *2PC* патерна издвајају се конзистентност и *read-write* изолација, што овај приступ чини погодним за *OLTP* (*online transaction processing*) системе, али уз ограничену скалабилност. Главни недостаци су синхрона комуникација и смањење доступности, могућност стварања *deadlock*-а и недостатак подршке за дуге трансакције, што овај приступ чини мање погодним за *B2B* апликације.

21.7.2 Сага патерн

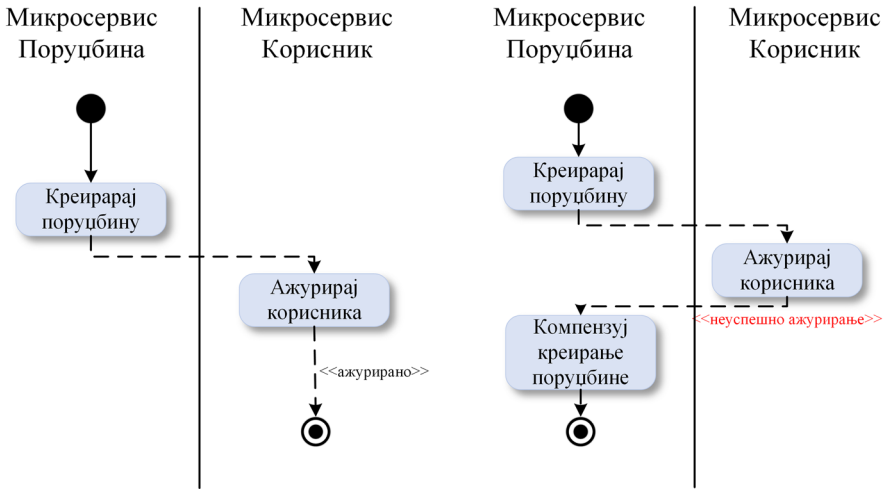
Други приступ за управљање дистрибуираним трансакцијама заснива се на Сага патерну[462]. У овом приступу, дистрибуирана трансакција се извршава кроз више локалних трансакција, које се обављају асинхроно. Трансакција се дели на низ операција од којих се свака може извршити као трансакција која гарантује *ACID* особине. Резултати извршења локалних трансакција чувају се у бази података, што омогућава праћење података о међустањима.

За имплементацију Сага патерна неопхнодне су најмање две локалне трансакције. Свака локална трансакција ажурира само податке у својој бази података. Локалне трансакције комуницирају асинхроно, преко порука или магистрале догађаја. Када једна локална трансакција заврши свој део посла, шаље поруку другој. Уколико

[461] Прилагођено према: https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture

[462] Garcia-Molina, H. & Salem, K. (1987). Sagas. *ACM Sigmod Record*, 16(3), 249–259.

је и друга локална трансакција успешна, промене су трајно забележене. Ако је неуспела, шаље поруку првој како би поништила резултате претходно извршене локалне трансакције (Слика 67).



Слика 67. Сага патерн – успешно и неуспешно извршавање[463]

Имајући у виду асинхрону комунукацију код примене Сага патерна, могуће је да систем доспе у неконзистентно стање. За постизање конзистентности система потребно је применити неку од техника за компензацију и постизање конзистентности у неком периоду (*eventual consistency*). Ако је трансакција била неуспешна због недоступности инстанце микросервиса, конзистентност се постиже када инстанца постане доступна (*startup reconciliation*). Конзистентност се може постићи на захтев клијента (*on demand reconciliation*) или у планираном времену (*scheduled reconciliation*).

Сага патерн је погодан за дуге трансакције, чије се извршење мери сатима, па и данима. Такође, значајне предности Сага патерна су скалабилност и висока доступност, јер нема закључавања ресурса у локалним трансакцијама. Главни недостаци односе се на потешкоће у дебаговању и недостатак *read-write* изолације, па је могуће прочитати погрешну вредност податка.

У случају да се дистрибуирана трансакција извршава кроз већи број локалних трансакција, неопходни су квалитетни механизми за координацију и синхронизацију, као што су оркестрација и кореографија[464].

[463] Прилагођено према: https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture

[464] Ozkaya, M. (2021). Saga Pattern for Microservices Distributed Transactions, https://medium.com/design-microservices-architecture-with-patterns/saga-pattern-for-microservices-distributed-transactions-7e95d0613345

21.8 Патерни постављања микросервиса у продукционо окружење

Постављање монолитног софтвера у продукционо окружење је једноставно, јер окружење најчешће представља један рачунар или кластер рачунара. Када су у питању микросервисне апликације, често постоји велики број међузависности компоненти, па је постављање микросервисног софтвера у продукционо окружење компликован и непредвидив процес. Ипак, постоји неколико патерна постављања у продукционо окружење (*deployment patterns*) који се могу применити на микросервисне архитектуре[465].

Постављање више инстанци микросервиса на један сервер

Патерни постављања у продукционо окружење више инстанци микросервиса по серверу (*Multiple Service Instances per Host pattern*) реализује се тако што се инстанце микросервиса из пословних процеса, који чине апликацију, постављају на један или више физичких или виртуелних сервера. Постоје две варијанте овог патерна: у првој се све инстанце микросервиса, које се односе на један пословни процес, постављају на један сервер. У другој варијанти, када је потребна већа скалабилност апликације и паралелан рад више истоветних микросервиса, микросервиси пословног процеса се расподељују на више сервера тако што се на сваки сервер постави више инстанци истог микросервиса који раде паралелно.

Овај патерн се једноставно примењује, омогућава брзо пуштање у рад микросервиса, ефикасан је у ангажовању ресурса, не захтева сложено продукционо окружење и има ниске трошкове експлоатације. Недостатак овог патерна је то што свака инстанца микросервиса треба да ради у изолованом окружењу кернела, као посебан процес оперативног система. Ангажовање ресурса сервера је због ангажовања ресурса кернела оперативног система тешко одредити и пратити, нарочито у случајевима када се апликација састоји из више паралелних пословних процеса, с више паралелизованих скалабилних инстанци микросервиса, а да се при томе свака инстанца микросервиса извршава као посебан и заштићен процес кернела оперативног система.

Постављање једне инстанце микросервиса на један сервер

Патерн постављања у продукционо окружење једне инстанце микросервиса на један сервер (*Service Instance per Host pattern*) је приступ који се заснива на поједностављењу процеса управљања скалабилношћу апликације тако што се на једну серверску јединицу поставља једна инстанца сервиса. Уколико је повећање скалабилности потребно, могуће је у реалном времену за микросервисе који су оптерећени увести више серверских јединица. Када се оптерећење смањи, тада се додатне серверске јединице могу искључити, а тиме се у реалном времену постиже

[465] https://www.developer.com/design/deployment-patterns-microservices/

управљање скалабилношћу и трошковима експлоатације с грануларношћу на нивоу микросервиса. Недостатак је повећана потрошња серверских ресурса по микросервису. Серверска јединица може бити виртуелна машина или контејнер на којима се покреће само један микросервис. Зависно од серверске јединице, постоји неколико начина примене овог патерна, који су описани у наставку.

Постављање једне инстанце микросервиса по виртуелној машини

Патерн постављања једне инстанце микросервиса по виртуелној машини (*Service Instance per Virtual Machine*) реализује се тако што се за сваки микросервис припреми посебна виртуелна машина. Повећање скалабилности се врши увођењем у рад нових виртуеланих машина с додатним инстанцама захтеваног микросервиса. Предност је грануларност управљања скалабилношћу на нивоу једног микросервиса и могућност имплементације виртуелне машине коришћењем инфраструктуре као сервиса, која се може добити на захтев од провајдера рачунарства у облаку. Недостаци примене овог патерна су повећана потрошња рачунарских ресурса по једном микросервису за износ потрошње свих ресурса потребних за рад оперативног система виртуелне машиине, повећана латенција у комуникацији између микросервиса јер се за комуникацију користе спољни ресурси рачунарске мреже, и увођење имплементационих детаља апликације у структуру виртуелне машине, што смањује ниво модуларности и могућност поновног коришћења исте виртуелне машине за друге апликације. Најзначајнији недостатак, поред великог коришћења ресурса, јесте споро покретање нових виртуелних машина, за чију потпуну фукционалност приликом пуштања у рад треба да прође време које се може мерити минутама.

Постављање једне инстанце микросервиса у један контејнер

Контејнери представљају средство за модуларну испоруку софтвера, са стандардизованим интерфејсима који апликацијама нуде интерфејсе директним приступом ресурсима кернела оперативног система, тако да при имплементацији једног микросервиса по контејнеру нема значајнијег додатног оптерећења за сервер у односу на имплементацију микросервиса на серверу без контејнера. То је велика предност у односу на случај када се један микросервис поставља на једну виртуелну машину. Уобичајено је да се контејнери инсталирају на виртуелној машини за коју је унапред дефинисан максимални капацитет ресурса који се могу ангажовати. Унутар расположивих ресурса на виртуелној машини, *IT* инжењери који су задужени за експлоатацију софтвера могу подизати више инстанци истог контејнера или додатне инстанце других контејнера. И у том случају је очувана грануларност управљања скалабилношћу на нивоу једног микросервиса, без губитака на перформансама или смањења флексибилности. Овај патерн је посебно погодан за имплементацију микросервиса, који се брзо и једноставно могу постављати у продукционо окружење. Свака инстанца микросервиса ради у свом

контејнеру. С друге стране, уколико није потребно управљање грануларношћу скалабилности на нивоу једног микросервиса, могуће је у оквиру једног контејнера поставити више повезаних микросервиса, па и комплетну апликацију.

Поред многих предности, недостатак примене овог патерна је отежано ажурирање микросервиса у реалном времену и ограничена могућност за континуалну испоруку софтвера. Када се уводи нова верзија микросервиса, потребно је угасити све контејнере са старијим верзијама микросервиса, како би се избегло функционисање апликације с више инстанци микросервиса различитих верзија које раде упоредо. Апликација је у неком периоду без функционисања микросервиса, што може довести до поремећаја у континуалном раду апликације. Иако су контејнери технологија, која је постигла висок ниво зрелости и која се користи у многим апликацијама, виртуелне машине су технологија с вишим нивоом зрелости и мање очекиваним проблемима. Ово треба имати у виду пре избора патерна за постављање сервиса у продукционо окружење код посебно важних апликација.

Постављање микросервиса у serverless окружење

Развојем рачунарства у облаку, провајдери су по повољним условима почели да нуде услуге инфраструктуре, софтверске платформе или услуге софтверског пакета као сервиса, који се могу закупити и платити на основу обима ангажоване услуге. За корисника је инфраструктура апстрахована и он о њој не мора да брине, као ни о физичким серверима, мрежној инфраструктури, подешавању и одржавању апликационих сервера и база података. Таква инфраструктура се назива *serverless* окружењем за извршење микросервиса. *Serverless* окружења која нуде провајдери рачунарства у облаку нису без физичких сервера. Физички сервери постоје, али о њиховом функционисању, поузданости, редундантности и скалабилности брине провајдер.

Инфраструктура као сервис нуди се у форми унапред припремљених виртуелних машина или контејнера, као и система за њихово управљање. Платформа као сервис се нуди у виду комплетне платформе са одговарајућим апликационим интерфејсима, преко којих се приступа скалабилним ресурсима. Најпознатије платформе које се користе за *Serverless* развој и постављање софтвера у продукционо окружење су: *Google App Engine*, *Google Cloud Platform* (*GCP*), *AWS Lambda*, *Oracle's cloud infrastructure*, *Azure serverless*, а на располагању су и многе друге.

Уколико се жели већа контрола над подацима, развојем и управљањем животним циклусом софтвера, уз истовремено коришћење контејнера као сервиса, данас се у великој мери користи *Kubernetes* (*K8S*)[466]. Услуге *Kubernetes* кластера нуди већина провајдера. *Kubernetes* је софтвер отвореног кôда и може се инсталирати и користити интерно за процесе развоја и одржавања софтвера, а постављање контејнера у продукционо окружење се може вршити на кластеру провајдера. На

[466] https://kubernetes.io/

тај начин се избегава претерано везивање за платформу провајдера, омогућава већа портабилност софтвера, уз већину предности постављања сервиса у *serverless* продукционо окружење.

Serverless окружења за контејнерске апликације веома су брзо постала популарна. Данас је то један од најбржих и најједноставнијих начина за постављање микросервиса у продукционо окружење. Смањују се трошкови експлоатације и управљања инфраструктуром. Смањује се ризик и повећава агилност развојних тимова јер се већи део поступака софтверског процеса, који се односе на издавање нове верзије софтвера и постављање софтвера у продукционо окружење ,може аутоматизовати. Управљање инфраструктуром могуће је апстраховати процедурама у облику скриптних језика. Такав приступ се назива инфраструктура као кôд (*Infrastructure as code – IaC*). Инфраструктура као кôд поједностављује постављање микросервиса у *serverless* продукционо окружење и у великој мери се користи у пракси.

IV ИМПЛЕМЕНТАЦИЈА

У овој секцији биће представљена изабрана поглавља која се односе на имплементацију и интеграцију софтвера оријентисаног на процесе. Највећи део секције биће посвећен композитним *API* сервисима, њиховој архитектури, пројектовању и имплементацији.

Архитектура софтвера дефинише модуле, њихове функционалности и *API*, али се не бави детаљном спецификацијом архитектуре и конкретном имплементацијом модула и *API* сервиса. Детаљно пројектовање модула, кодирање, тестирање и постављање у продукционо окружење врши се у фази имплементације. *API* сервиси имају своју интерну, најчешће вишеслојну архитектуру, па је потребно ту архитектуру посебно пројектовати, кодирати и тестирати, тако да се фазе пројектовања архитектуре, интеграције и имплементације преплићу и циклично утичу једна на другу.

У софтверу оријентисаном на процесе, који се у већој мери заснива на агрегацији и интеграцији, добар део послова у фази имплементације, уместо кодирања, састоји се од конфигурисања интеграционе инфраструктуре *codeless* или *low-code* софтверских платформи[467]. Специфичности везане за конкретан случај коришћења могу да се уграде применом инверзије и уметања зависности, преко апстрактних *API*-ја, које дефинише и нуди произвођач платформе, а имплементира корисник. Тиме се процес имплементације софтвера оријентисаног на процесе знатно поједностављује, а истовремено се отварају могућности за примену нових и иновативних пословних модела, како за произвођаче софтвера који софтверске модуле могу нудити на тржишту као готове производе, у форми софтверских оквира, платформи и екосистема, тако и за кориснике који могу њиховим коришћењем знатно поједноставити процес имплементације и на једноставан начин обезбедити скалабилан, добро тестиран, квалитетан и поуздан софтвер.

Коришћење софтверских оквира, платформи и екосистема на инфраструктури рачунарства у облаку отвара нове могућности за имплементацију, тестирање, континуалну интеграцију и испоруку софтвера.

---

[467] https://powerapps.microsoft.com/en-us/low-code-no-code-development-platforms/

22. АПЛИКАЦИОНИ ПРОГРАМСКИ ИНТЕРФЕЈСИ

Апликациони програмски интерфејс (*Application Programing Interface,* скр. *API*) је средство комуникације између компоненти, модула и сервиса савремених дистрибуираних софтверских апликација. Представља „мост“ који различитим програмским ентитетима омогућава да ефикасно комуницирају и размењују податке[468]. *API* апстрахује комуникацију, али такође апстрахује и енкапсулира податке, знање и функционалности модула или сервиса на који се односи[469]. Поред основне намене, да омогући и модерише комуникацију софтверских компоненти, *API* омогућава да се унапреде и додају нове функционалности и реализује интеграција с другим апликацијама и сервисима.

Данашњи софтвер готово увек захтева повезивање компоненти, тако да је *API* постао синоним за модуларност, а развој *API*-ја је постао један од најважнијих поступака у развоју софтвера. Велики део данашњих пословних, мобилних и веб екосистема зависи од имплементације одговарајућих *API*-ја.

На еволуцију *API* највише су утицале парадигме развоја софтвера и њихова способност да имплементирају модуларност и дистрибуираност. Историја и еволуција *API* непосредно је повезана са историјом развоја модуларизације и историјом развоја комуникационих технологија. У педесетим годинама двадесетог века, британски научници Морис Вилкс и Дејвид Вилер активно су радили на развоју модуларне софтверске библиотеке за један од првих британских рачунара *EDVAC*. Током истраживања, креирали су документацију која би се данас могла сматрати првим *API*-јем. Иако се ова документација и технички и визуелно знатно разликовала од савремених, већина историчара информационих технологија сматра је својеврсним прототипом. *API* се први пут помиње у књизи под називом *The Preparation of Programs for an Electronic Digital Computer*[470], објављеној 1951. године. *API* је први пут прецизније дефинисан и описан у научној литератури 1968. године у раду *Data structures and techniques for remote computer graphics*[471]. У то време су се *API*-ји користили за комуникацију различитих рачунарских система, укључујући терминале и штампаче.

Током осамдесетих година, појавила се потреба за дистрибуираним приступом библиотекама које се налазе на удаљеним рачунарима. Већ тада је било јасно да фокус објектно-оријентисаног програмирања на енкапсулацију података само унутар објекта представља велико ограничење, и да се енкапсулација мора проширити на дистрибуиране компоненте и модуле. Програмски језик *Java* је као

[468] https://www.techtarget.com/searchapparchitecture/tip/What-are-the-types-of-APIs-and-their-differences
[469] https://hackernoon.com/the-basics-of-designing-an-api-architecture
[470] Wilkes, M. V., Wheeler, D. & Gill, S. (1951). The Preparation of Programs for an Electronic Digital Computer: With special reference to the EDSAC and the Use of a Library of Subroutines. Addison-Wesley Press.
[471] Cotton, I. W. & Greatorex Jr, F. S. (1968, December). Data structures and techniques for remote computer graphics. In: *Proceedings of the December 9–11, 1968, fall joint computer conference*, part I (pp. 533–544).

одговор на такве трендове увео концепт пакета (*Package*)[472], области именовања у оквиру пакета (*Namespaces*), и позив удаљених процедура (*Remote Procedure Call – RPC*) као подршку за дистрибуираност.

Широким прихватањем парадигме процесно-оријентисаног развоја софтвера, микросервисних архитектура и интернет технологија, *API* постаје централна компонента око које се организује развој софтвера. У периоду после 2010. године, *API*-ји су наставили да се интензивно развијају, пре свега због повећања броја дистрибуираних система.

Организације су се ослањале на добро пројектоване *API*-је како би искористиле предности скалабилности, перформанси и безбедности у оквиру дистрибуираних система. Данас су апликациони програмски интерфејси централна компонента за комуникацију модула у дистрибуираним окружењима[473,474].

22.1 Пројектовање API

Пројектовање апликационих програмских интерфејса је активност у оквиру софтверског процеса, која се реализује на граници између фаза пројектовања архитектуре и имплементације. Састоји се од пројектовања структуре и функционалности *API*-ја, као и обликовања начина интеракције са околином.

Термин *API* се у жаргону софтверског инжењерства користи као синоним којим се апстрахује комплетан модул или сервис са интерфејсом ка окружењу. У том смислу, под имплеметацијом *API*, подразумева се пројектовање и имплементација комплетног модула или *API* сервиса са интерфејсом.

Приликом пројектовања *API*-ја, потребно је јасно дефинисати интерфејсе који омогућавају интеграцију различитих компоненти, модула, апликација и сервиса. Стандарди за документацију и примери употребе, у фази пројектовања *API*-ја, обезбеђују јасна упутства за пројектовање и имплементацију.

Процес пројектовања *API*-ја почиње спецификацијом корисничких захтева и случајева коришћења које *API* треба да подржи. При пројектовању, важно је обезбедити јасноћу и доследност у спецификацији, како би се олакшала интеграција микросервиса и његово коришћење. Руте (адресе и усмеравање) за приступ функционалностима *API*-ја треба да буду прецизно и недвосмислено дефинисане, коришћењем *http* метода (*get*, *post*, *put*, *delete*). Одговор такође треба да буде доследно дефинисан, најчешће у *JSON* или *XML* формату. Тиме се корисницима омогућава да једноставно прихвате и обрађују податке које добијају од *API*-ја[475].

Последњи корак у фази пројектовања *API*-ја је писање документације.

[472] https://www.javatpoint.com/package
[473] https://traefik.io/blog/the-history-and-evolution-of-apis/
[474] https://medium.com/transparent-data-eng/short-history-of-api-from-cabinet-to-big-boom-894c56b2c332
[475] https://www.postman.com/api-platform/api-design/

Документација олакшава интеграције и одржавање *API*-ја. Документација треба да садржи информације о свакој рути, подржаним методама, очекиваним параметрима, форматима одговора и могућим грешкама.

Добро пројектован *API* треба да буде једноставан за учење, тако да корисници могу брзо да разумеју и прихвате његове основне концепте, функционалности и начине коришћења. Добар *API* не само што обезбеђује функционалност, већ чини интеракцију интуитивном за кориснике[476]. Рачунарски кôд за имплементацију *API* треба да буде јасан и лак за одржавање. Такође, *API* треба да буде лако проширив, како би омогућио једноставно ажурирање и додавање нових функционалности.

22.1.1 Патерни архитектуре API

Патерни архитектуре *API* представљају структурне и функционалне оквире који се користе за решавање уобичајених проблема у пројектовању *API* сервиса и њихове комуникације. Представљају примере најбоље праксе и искуства у области развоја софтвера, обезбеђујући тиме конзистентан и оптимизован приступ пројектовању *API*-ја. Коришћење патерна у пројектовању *API* олакшава процес пројектовања и доприноси повећању читљивости кôда и лакоће ажурирања апликационог програмског интерфејса.

Архитектура *API* се пројектује у више слојева хијерархије. На највишем нивоу су патерни архитектуре *API* груписани по намени, начину коришћења и класама проблема које решавају, док се на нижим нивоима хијерархије користе патерни који су повезани са архитектурним стиловима и протоколима који се користе у конкретној имплемантацији.

API се могу груписати по намени, приступу и начину коришћења у следеће групе[477]:

1. **Јавни *API*.** Екстерни, отворени или јавни *API*, доступан је за екстерно коришћење од стране других програмера и предузећа. Предузећа која се опредељују за пословну стратегију дељења својих апликација и података са другим предузећима развијају и стављају на располагање свој јавни *API* за те послове. Овакви интерфејси обично обухватају умерени степен аутентификације и ауторизације.
2. **Партнерски *API*.** Пројектује се за коришћење од стране пословних партнера с којима предузеће послује. Омогућава приступ ограниченим функционалностима или подацима предузећа, при чему је приступ регулисан уговором. Из тих разлога, партнерски *API* укључује јачу аутентификацију, ауторизацију и безбедносне механизме.
3. **Интерни *API*.** Интерни или приватни *API* је намењен за употребу унутар предузећа. Служи као средство за комуникацију компоненти, модула и сервиса

[476] Bloch, J. (2006, October). How to design a good API and why it matters. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (pp. 506–507). https://storage.googleapis.com/pub-tools-public-publication-data/pdf/32713.pdf

[477] https://www.techtarget.com/searchapparchitecture/tip/What-are-the-types-of-APIs-and-their-differences

унутар предузећа. Овај тип *API*-ја помаже у интеграцији различитих система и апликација у оквиру информационог система предузећа. Интерни *API*-ји се карактеришу слабијом сигурношћу и аутентификацијом. Често се дешава да немају никакве безбедносне механизме и заштиту.

4. **Композитни *API*.** Композитни *API* представља комбинацију више различитих *API*-ја у један интерфејс. Он обезбеђује енкапсулацију, приступ и коришћење функционалностима и подацима из више извора. Овакав тип *API*-ја често се користи за оптимизацију употребе и ефикасно управљање подацима и сервисима у оквиру компоненти које се састоје из више модула.

По областима и класама проблема које решавају, патерни архитектуре апликационих интерфејса могу се класификовати у следеће групе[478]:

1. ***API Visibility* патерни**

   Приликом пројектовања *API*-ја, прва одлука се односи на његову видљивост и доступност. Ова одлука није само техничка, већ управљачка и организациона. Питање које се поставља је: одакле *API* треба да буде доступан, са веба, из мреже са контролисаним приступом као што је интранет, или само из дата центра? У овој групи се издвајају три кључна патерна: *Public API* патерн, *Community API* патерн и *Solution Internal API* патерн.

2. ***API Direction* патерни**

   Ова група патерна се односи на стратешка питања: које врсте система, подсистема и компоненти су интегрисане коришћењем *API*, и из којег слоја архитектуре би *API* требао да буде доступан? Одговори на ова питања помажу у одређивању опсега и карактеризацији *API*-ја између два *API direction* патерна: *Frontend Integration* патерн и *Backend Integration* патерн.

3. ***Basic Message Structure* патерни**

   Ова група патерна се односи на структуру порука које се шаљу као захтев и добијају као одговор приликом коришћења *API*-ја. Као главни патерни се издвајају *Atomic Parameter* патерн, *Atomic Parameter List* патерн, *Parameter Tree* патерн и *Parameter Forest* патерн.

4. ***Endpoint Roles and Operation Responsibilities* патерни**

   Ова група патерна се односи на улазне параметре *API* који се очекују од клијента приликом позива, као и на операције које се након позива извршавају. У овој групи се јавља већи број патерна као што су *Processing Resource* патерн, *Information Holder Resource* патерн, *Data Transfer Resource* патерн, *Link Lookup Resource* патерн и др.

5. ***Designing Request and Response Message Representations* патерни**

   Подаци представљају основни градивни блок сваке клијент-сервер комуникације. Група патерна који су усмерени ка подацима и дефинишу значења и семантику података, у порукама захтева и одговора, обухвата: *Data Element* патерн, *Metadata Element* патерн, *ID Element* патерн и *Link Element* патерн. Патерни који пружају

[478] Zimmermann, O., Stocker, M., Lubke, D., Zdun, U. & Pautasso, C. (2022). *Patterns for API design: simplifying integration with loosely coupled message exchanges*. Addison-Wesley Professional.

додатне информације о репрезентацији порука и имају комплекснија значења су *API Key* патерн, *Error Report* патерн и *Context Representation* патерн.

6. ***Refine Message Design for Quality*** **патерни**

   Ова група патерна се бави квалитетом пројектовања апликационих програмских интерфејса. Квалитет *API*-ја односи се на меру усаглашености: надлежности, перформанси, безбедности и удобности коришћења апликационог програмског интерфејса. Овде се идентификује неколико различитих патерна: *Pagination* патерн, *Embedded Entity* патерн, *Linked Information Holder* патерн, *Conditional Request* патерн, *Request Bundle* патерн, *Wish List* патерн и *Wish Template* патерн.

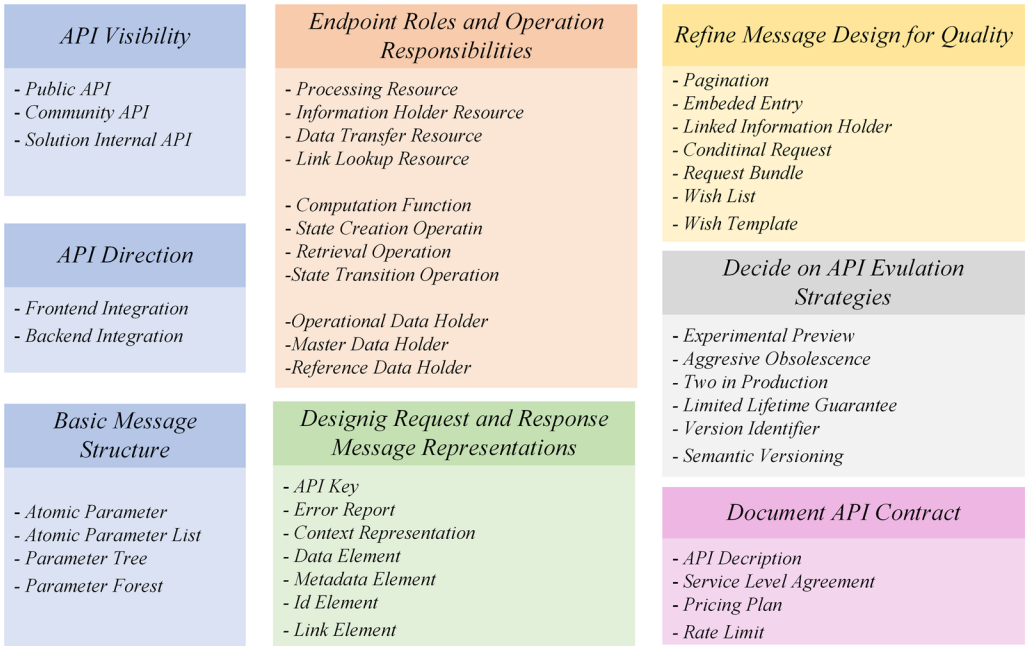
7. ***Decide on API Evolution Strategies*** **патерни**

   Ова група патерна обухвата патерне који директно или индиректно утичу на квалитет *API*-ја у области аутономије, омогућавајући *API* провајдерима и клијентима праћење животног циклуса, тако да провајдер лако и једноставно може да уведе нове верзије *API*-ја, без ремећења рада система; проширивање, унапређивање и прилагођавање *API* новим захтевима и омогућавање компатибилности, гаранције да неће доћи до семантичких неспоразума између клијента и сервера. У ову групу се убрајају следећи патерни: *Experimental Preview* патерн, *Agressive Obsolescence* патерн, *Two In Production* патерн, *Limited Lifetime Guarantee* патерн, *Version Identifier* патерн и *Semantic Versioning* патерн.

8. ***Document API Contract*** **патерни**

   Патерни за документацију *API*-ја односе се на начин који омогућава да се функционалне могућности, својства квалитета и пословни аспекти *API*-ја документују и саопште клијентима. Ова група обухвата *API Description* патерн, *Service Level Agreement* патерн, *Pricing Plan* патерн и *Rate Limit* патерн.

На Слици 68 приказани су патерни архитектуре апликационих програмских интерфејса и њихова намена. Наведени патерни су највишег нивоа. У хијерархијској декомпозицији, архитектура *API* се надаље може разлагати по имплементационим карактеристикама повезаним с коришћеним архитектурним стиловима и протоколима.



Слика 68. Патерни архитектуре апликационих програмских интерфејса[479]

22.1.2 API архитектурни стилови и протоколи

Патерни архитектуре *API* се, поред спецификације на највишем нивоу апстракције дате на Слици 68, на нижим нивоима апстракције који су оријентисани ка комуникацији, дистрибуираности, скалабилности и поузданости, могу спецификовати коришћењем *API* архитектурних стилова и протокола.

Архитектурни стилови дефинишу препоруке и ограничења које архитектура *API* треба да задовољи, док протоколи захтеве и ограничења стандардизују у форми међународних или струковних стандарда. Најчешће коришћени архитектурни стилови и протоколи у пројектовању *API* су[480]:

1. ***REST***. Трансфер стања репрезентацијом ресурса (*Representational state transfer*) је најпопуларнији архитектурни стил за спецификацију *API*-ја. Заснован је на принципима који олакшавају скалабилну и флексибилну комуникацију између компоненти и модула. Ослања се на клијент-сервер архитектуру. Не складишти податке или статус између позива. Користи *http* протокол за пренос података и базира се на концепту ресурса који се идентификује преко универзалног идентификатора (*URI*).

[479] Прилагођено према: Zimmermann, O., Stocker, M., Lubke, D., Zdun, U. & Pautasso, C. (2022). *Patterns for API design: simplifying integration with loosely coupled message exchanges*. Addison-Wesley Professional.

[480] https://dev.to/kanani_nirav/top-6-most-popular-api-architecture-styles-you-need-to-know-with-pros-cons-and-use-cases-564j

2. ***RPC***. Протокол за позивање удаљених процедура (енгл. *Remote Procedure Call)* користи се за слање параметара дистрибуираном сервису на обраду и пријем резултата обраде. *RPC* позивају извршне акције или процесе, док се *REST* позивима углавном размењују подаци или ресурси, као што су нпр. документи.
3. ***gRPC***. Популарна имплементација *RPC* отвореног кôда, коју је развио *Гугл*. Подржава више програмских језика и нуди многе напредне функционалности у односу на класичан *RPC*. *gRPC* користи *http/2* за комуникацију, а подржава и серијализацију података у *JSON* или *Protobuf* формату. Најчешће се користи у микросервнисним архитектурама. *gRPC* не следи стандардне принципе веба, што може бити изазов за интеграцију са стандардним веб алатима и претраживачима. *gRPC* је погодан када је модел података сложен и динамичан, а сервиси зависе један од другог[481] .
4. **Сервисно-оријентисана архитектура *SOA* и *SOAP***. Сервисно-оријентисана архитектура (*SOA*) представља стандардизовани облик клијент-сервер архитектуре у којој се за спецификацију параметара позива користи *SOAP* (*Simple Object Access Protocol*). Користи се за комуникацију између веб сервиса и обезбеђује једноставан начин за слање и примање порука између софтверских компоненти.
5. ***GraphQL***. *GraphQL* је архитектурни стил и упитни језик који је развио *Фејсбук*. Омогућава креирање захтева за специфичним и тачно дефинисаним подацима, што смањује одговоре с превише или премало података. Његова флексибилност и ефикасност чине га прилагодљивим за апликације с комплексним захтевима за подацима. Једна од највећих предности *GraphQL*-а лежи у његовој универзалности. Користи сопствени језик дефиниције шеме података. *GraphQL* је сложен и захтева од корисника учење нове синтаксе и логике. Не подржава кеширање, што може утицати на перформансе.
6. ***WebSocket***. *WebSocket* је протокол који се користи за двосмерну интерактивну комуникацију клијента и сервера у реалном времену. Најчешће се имплементира у апликацијама где је потребна брза и ефикасна размена података с малим кашњењем. Примери таквих апликација су чет апликације и онлајн игрице у реалном времену. За разлику од *http* протокола, *WebSocket* користи једну перманентну везу за сву комуникацију између клијента и сервера, што га чини бржим и ефикаснијим. Подржава различите типове порука, укључујући текстуалне, бинарне и стримоване податке. Не користи шифровање или аутентификацију. Безбедност се може постићи имплементацијом *wss://* шеме која је слична *https://* за *http* протокол. Не чува информације о вези или статусу, што га чини *stateless* протоколом.
7. ***Webhooks***. *Webhook*-ови представљају асинхрону *API* архитектуру преко које сервери обавештавају клијенте о насталим догађајима у току извршења сервиса. За комуникацију се користе *http* повратни позиви или *post* захтеви како би се пренели подаци који садрже информације о насталим догађајима. Клијенти региструју своје *webhook*-ове на серверима преко *API URL* адреса. *Webhook*-ови

[481] https://dev.to/kanani_nirav/top-6-most-popular-api-architecture-styles-you-need-to-know-with-pros-cons-and-use-cases-564j

обрађују догађаје у форми *http* повратних позива и операције које се одвијају асинхроно. Ова метода комуникације је једноставна и лака за разумевање због употребе стандардних *http* метода и формата. Користи постојеће стандарде и протоколе веба, што доприноси његовој ефикасности и скалабилности. Одсуство строго дефинисаног формата порука може довести до недоследности и двосмислености у комуникацији између сервера и клијената. Не подржавају сложене упите или операције, већ преносе само једносмерна обавештења без повратних информација, и не обезбеђују поновне покушаје или потврде у случају грешака. *Webhook*-ови се најбоље примењују у ситуацијама где су потребна обавештења заснована на догађајима, када је модел података једноставан и стабилан и када сервери и клијенти не зависе један од другог. Ово је погодно за перформансе и скалабилност, смањујује оптерећење и латенцију.

8. ***HTTP streaming***. *HTTP streaming* је протокол преноса мултимедијалних података путем *http* протокола у реалном времену, где се подаци шаљу клијенту или серверу континуално, све док су доступни. Стримовање података путем серверских догађаја (*Server-Sent Events – SSE*) најзаступљенији је тип *http streaming*-а који омогућава серверу да шаље асинхрона ажурирања клијенту преко једног усмереног *http* тока. Када сервер прими нове податке, шаље их клијенту као део *SSE* догађаја. Клијент користи стандардизовани *EventSourcing API* за пријем ових догађаја и ажурирање корисничког интерфејса у складу с променама[482].

Избор *API* архитектурног стила или протокола може значајно утицати на успешност примене *API*-ја. Предузећа би требало пажљиво да бирају протокол који одговара сложености података које треба размењивати, нивоу сигурности потребном за те податке и захтеваним перформансама.

22.1.3 SOAP и REST у сервисно-оријентисаној архитектури

Сервис је агрегација догађаја који својим извршењем мења стање неког ресурса или ентитета. Синоним за сервис у моделовању пословних процеса је активност. Модул се може састојати од једног или више сервиса. Сервис се лако пројектује и програмира, може да буде независан и лако се извршава применом клијент-сервер архитектуре. Једноставно се дистрибуира, а паралелизацијом сервиса може се постићи линеарна скалабилност обраде.

После низа неуспешних покушаја да се објектно-оријентисано програмирање начини дистрибуираним и скалабилним, сервис је прихваћен у тежњи да се уведе нови концепт који би могао у највећој мери заменити метод, а поседовати могућност дистрибуираности и линеарне скалабилности.

Дистрибуирани сервис се може позивати коришћењем *http* протокола и његових метода. То значајно поједностављује начин његовог коришћења. За широку употребу, нужно је додатно дефинисати стандарде или препоруке за апликационе

[482] Jin, B., Sahni, S. & Shevat, A. (2018). *Designing Web APIs: Building APIs That Developers Love*. O'Reilly Media, Inc.

програмске интерфејсе који би се користили при позиву сервиса. *W3C* конзорцијум је стандардизовао архитектуру веб сервиса[483]. Готово сви објектно-оријентисани програмски језици су већ до почетка двехиљадите године прихватили веб сервис и сервисно-оријентисану архитектуру[484] која значајно проширује опсег њихове употребе.

SOA је донела многе новине и значајно унапредила методе пројектовања софтвера. Међутим, у примени *SOA* се наредних година претеривало. Сервис је постављен као универзални концепт модуларности и агрегације на свим нивоима апстракције, кудикамо шире од његове основне намене да реализује промену стања само једног ресурса. Такав приступ је довео до прегломазне и компликоване архитектуре интеграције, која је била тешка и скупа за одржавање[485]. Због комплексности коришћења, *SOA* се напушта и ретко се користи.

Отклон од компликоване инфраструктуре за повезивање сервиса у *SOA*, почео је 2000. године практично паралелно с њеним увођењем, предлогом новог архитектурног стила за позивање веб сервиса – *Representational state transfer* (*REST API*). Наредних година *REST API* је широко прихваћен и ингегрални је део програмског језика *Java*, од *Java* 6 верзије.

Једна од првих значајних разлика између *REST* и *SOAP* је у начину дефинисања архитектуре. Док је *REST* архитектурни стил, *SOAP* је протокол за размену података. *REST* се базира на принципима једноставности и јасности, што га чини лакшим за разумевање и флексибилнијим за коришћење. С друге стране, *SOAP* је комплексан, структуриран, строго контролисан и јасно дефинисан као стандард.

Још једна битна разлика тиче се формата порука. *REST* користи различите формате као што су *JSON* и *XML*, док *SOAP* користи искључиво *XML*. Такође, *REST* користи *http* као протокол за пренос података, док *SOAP* може користити различите протоколе, укључујући *http*, *smtp* и др. *REST* је *stateless*, што значи да сервис не чува стање између захтева и одговора. С друге стране, *SOAP* може чувати стање, што може бити од значаја у одређеним ситуацијама.

Употреба и развој ова два протокола такође се разликују. *REST* је чешће коришћен у веб апликацијама, посебно тамо где се цени једноставност и брзина. *SOAP* је традиционалнији и чешће се користи у пословним окружењима где су потребни висок ниво сигурности и стабилности. Избор између *REST* и *SOAP* зависи од специфичних потреба и захтева апликације или система. Оба протокола имају своје предности и могу се применити у различитим системима, у зависности од конкретних изазова и захтева пројекта.

---

[483] https://www.w3.org/TR/ws-arch/

[484] Лазаревић, С. (2008). Сервисно оријентисана архитектура, *Научно-стручни скуп Информатика 2008 – нови трендови у развоју информационих система*, Друштво за информатику Србије, Београд, 13. 05. 2008.

[485] Král, J. & Zemlicka, M. (2009, November). Popular SOA antipatterns. In: *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns* (pp. 271–276). IEEE.

Програмери често користе *SOAP* за интерне или партнерске *API*-је. *REST* се користи за брзу размену релативно једноставних података. *REST* може да подржи већу скалабилност. Ове карактеристике чине *REST* популарним за јавне *API*-је, које често користе мобилне апликације.

Ни *REST* не решава све проблеме модуларности. Почетком друге деценије двадесетпрвог века кренуло се ка тражењу новог концепта модуларности, који би уместо комплексних сервиса или модула који обједињују више сервиса, био основни градивни елемент апликације. Контејнер, као концепт модуларности, који има могућност једноставне и ефикасне енкапсулације, чак и појединачних једноставних микросервиса, постао је дефакто стандард за модуларизацију.

22.2 REST API

Трансфер стања репрезентацијом ресурса (*Representational state transfer – REST*) је архитектурни стил, скуп препорука и ограничења за пројектовање архитектуре сервиса и спецификацију апликационог програмског интерфејса. *REST* принципи и ограничења заснивају се већим делом на концептима архитектуре Рачунарства оријентисаног на ресурсе (*Resource Oriented Computing – ROC*)[486,487].

REST није стандардизован, резултат је докторске дисертације Томаса Филдинга из 2000. године, на Универзитету Ирвин у Калифорнији[488]. При дефинисању скупа препорука и ограничења за архитектуру сервиса, Филдинг је пошао од претпоставке да архитектура сервиса и апликациони програмски интерфејс треба у највећој мери да буду прилагођени семантици и намени умрежених сервиса преко којих се приступа дистрибураним информационим ресурсима на интернету.

Спецификација апликационог програмског интерфејса бави се приступом информационим ресурсима на интернету и начином иницијације промене њихових стања. Клијент не приступа директно ресурсима, већ датотеци с репрезентацијом ресурса која садржи тренутно актуелне линкове како да се ресурсу приступи и како да се иницирају промене стања. Индиректним адресирањем сервера, преко садржаја датотеке са описом ресурса, слаби се чврста веза између клијента и сервера и омогућава имплементација слојевите архитектуре сервиса, паралелизација, повећање поузданости, скалабилности, лакше одржавање и промена функционалности у реалном времену, а да клијент о томе ништа не мора да зна, нити да било шта мења у позиву.

---

[486] Overdick, H. (2007, July). The resource-oriented architecture. In: *2007 IEEE Congress on Services (Services 2007) (*pp. 340–347). IEEE.

[487] Lucchi, R., Millot, M. & Elfers, C. (2008). Resource oriented architecture and REST. *Assessment of impact and advantages on INSPIRE, Ispra: European Communities*. https://core.ac.uk/download/pdf/38617810.pdf

[488] Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. University of California, Irvine. PhD Thesis. https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

22.2.1 Препоруке и ограничења за REST API архитектуру

REST API се не односи само на пројектовање апликационог програмског интерфеја, већ преко скупа препорука и ограничења дефинише комплетну архитектуру за имплементацију *API* сервиса. Конкретне препоруке и ограничења које треба имати у виду при пројектовању *REST API* архитектуре су[489]:

1. **Клијент – сервер архитектура.**

   REST API се имплементира искључиво у клијент-сервер архитектури. Архитектура се састоји од клијента, сервера и ресурса на интернету чије се стање мења извршењем сервиса. Клијент, при позиву, од сервера добија датотеку са описом начина приступа ресурсу, као и хиперлинкове за иницирање промена стања ресурса.

2. **Униформан начин приступа дистрибуираном ресурсу.**

   Клијент при позиву сервиса има само адресу локације датотеке која садржи репрезентацију ресурса. Датотека с репрезентацијом ресурса садржи довољно информација намењених клијенту, за приступ и промену стања ресурса, најчешће слеђењем хиперлинкова. За приступ ресурсу, улаз или излаз (*Entrypoint, End-point*), користи се *Uniform Resource Locator* (*URL*)[490]. Репрезентације ресурса, које укључују податаке, хипермедије и линкове за прелазе стања, енкодују се коришћењем *JSON*-а, *XML*-а или *HTML*-а. За комуникацију између клијента и сервера, најчешће се користи *http* протокол, а за промену стања ресурса, његове методе.

3. ***Stateless* комуникација између клијента и сервера.**

   Комуникација између клијента и сервера је *stateless*. Сервис не меморише податке о претходно извршеним позивима клијента. Сваки захтев клијента је независан и није повезан с претходним. Податке о сесији води клијент за своје потребе, које чува у својој бази података. Клијент није упознат са стањем сервиса, нити сервис може знати детаље о стању клијента. За сваки наредни позив, клијенту се може доделити други сервер. Тиме се омогућава лакша паралелизација и управљање скалабилношћу сервиса.

4. **Кеширање података у комуникацији клијент – сервер.**

   Неки од података које генерише сервер су статичке природе, или се споро мењају, па се на клијенту могу кеширати. У ту сврху, за сваки одговор или резултат који клијент добије у току комуникације, сервер мора имплицитно или експлицитно назначити да ли постоји могућност његовог кеширања и колико дуго су кеширани подаци актуелни. Уколико сервер назначи да могућност кеширања постоји, клијент може меморисати податке из комуникације, које надаље може користи локално при наредном позиву сервиса, што смањује број интеракција између клијента и сервера.

---

[489] https://restfulapi.net/

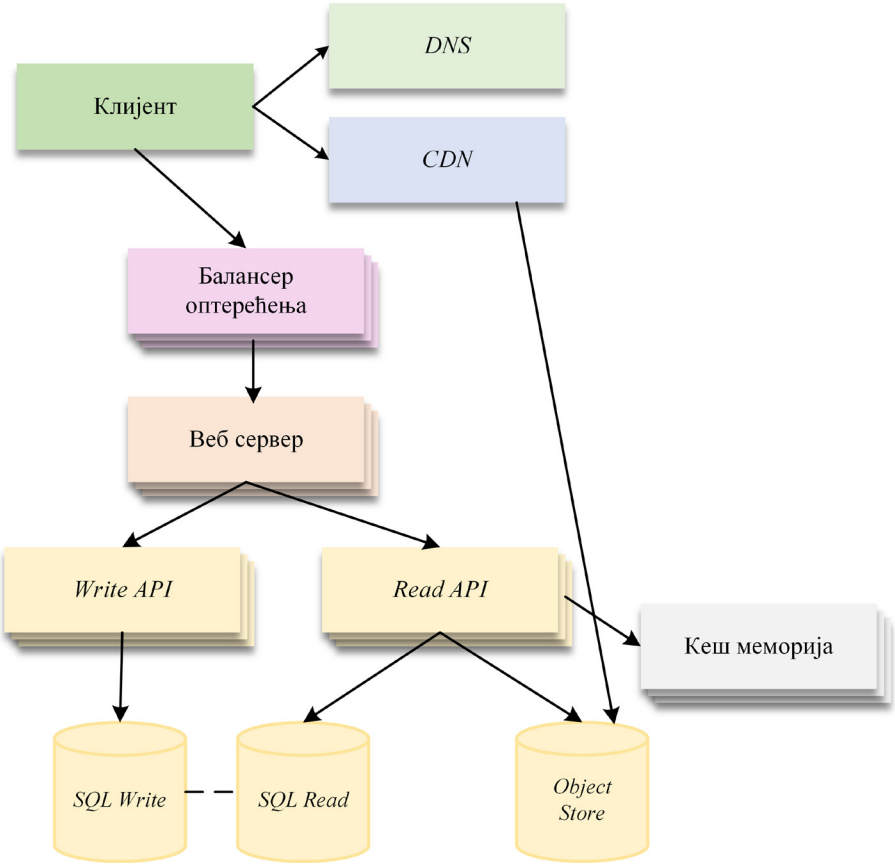
[490] Connolly, D. & Sperberg-McQueen, C. M. (2009). Web addresses in HTML 5, W3C. https://www.w3.org/html/wg/href/draft#url

5. Вишеслојна архитектура сервиса.

Сервис треба да има вишеслојну архитектуру, која би омогућила да сигурност и расподела оптерећења буде уграђена у ниже слојеве, а да за клијента који приступа вишим слојевима, ти детаљи не буду видљиви. На Слици 69 је приказана типична четворослојна архитектура: клијент, веб сервер, *API* слој с кеширањем и слој података. Између слојева се налазе посредници и балансери оптерећења.

6. Достављање кôда на захтев.

Достављање кôда на захтев, опциона је препорука (*Code on demand*) која омогућава серверу да део својих функционалности, тј. извршног кôда, пренесе на клијента. Обрадом кôда на клијенту, повећава се скалабилност апликације.



Слика 69. Типична имплементација слојева архитектуре REST API[491]

REST API није стандард ни прописани протокол. Поштујући наведене принципе и ограничења, може се имплементирати на различите начине, коришћењем различитих технологија. За *API* сервис се каже да је *RESTful* уколико је при његовом пројектовању и имплементацији поштовано свих наведених шест препорука и ограничења.

[491] Прилагођено према: https://danmartensen.svbtle.com/exploring-rest-api-architecture

22.2.2 Елементи REST API архитектуре

Елементи из којих се у процесу пројектовања дефинише *REST API* архитектура могу се грубо поделити на: елементе који се односе на комуникацију и елементе који се користе у имплементацији архитектуре. Елементи *REST API* архитектуре су[492]:

1. **Крајње тачке.**

   Крајње тачке представљају *URL* адресу преко које клијенти усмеравају захтеве ка серверској страни (*Entrypoint*), док серверска страна прима и обрађује одговоре на пристигле *http* захтеве (*Endpoint*).

2. **Модел података *API*.**

   Модел података *API* је апстрактна представа података који се размењују између клијента и сервера. Модел података, поред података који се непосредно размењују преко *API*, треба да обухвати објекте, атрибуте и релације објеката у софтверском систему, који учествују у клијент-сервер комуникацији или на које клијент-сервер комуникација има непосредан утицај. Модел података *API* на страни сервиса треба да у оквиру модела слојева *API* садржи структуре података потребне за обезбеђење скалабилности, поузданости и сигурности. При пројектовању модела података за *API*, потребно је тежити максималној флексибилности у комуникацији и имати у виду могућа будућа проширења софтверског система.

3. **Аутентификација и ауторизација.**

   Ова компонента у оквиру слојевите *API* архитектуре обезбеђује аутентификацију, потврђивање идентитета корисника или сервиса, и ауторизацију, додељивање права приступа и овлашћења корисницима или сервисима, омогућавајући им тиме да могу приступити ресурсима и обављати одређене акције. Ово се може урадити помоћу токена, *OAuth 2.0* и других метода са статусним кодовима.

4. **Верзионисање.**

   Ова функционалност архитектуре омогућава паралелно постојање различитих верзија *API*-ја, што спречава нарушавање функционисања постојећих апликација при изменама кôда, било да се ради о значајним или мањим изменама у различитим верзијама *API*. Верзионисање омогућава програмерима да пре шире дистрибуције тестирају нову верзију *API* на одабраној групи корисника, док остали корисници у исто време користе стару верзију *API*.

5. **Аналитика и праћење.**

   Мониторинг и анализа *API*-ја су кључни за брзо идентификовање проблема у перформансама или безбедносним ризицима, као и за праћење образаца коришћења и понашања корисника.

6. ***SDK*.**

   SDK (енгл. *Software Development Kits*) представља скуп компоненти, библиотека, алата и документације који програмерима олакшавају процес развоја апликација. Унапред изграђене компоненте за често коришћене патерне штеде време, јер програмери могу да користе *API* без потребе за разумевањем интерних детаља.

[492] https://programmers.io/blog/basic-designing-of-api-architecture/

22.3 Сигурност API

Због пораста популарности и употребе микросервиса и *serverless* архитектура, сајбер напади на *API*-је су све чешћи. Да би се напади предупредили, нужно је основну аутентификацију, која захтева само корисничка имена и лозинке, проширити различитим облицима безбедносних токена, мултифакторске аутентификације и *API* гејтвеја[493].

Сигурност апликационих програмских интерфејса остварује се применом метода, техника и процедура за обезбеђење интегритета, поверљивости и доступности података који се размењују између клијента и сервера. Поступци, процедуре и функционалности које се користе у обезбеђивању сигурности *API* су:

- **Аутентификација**. Безбедност *API* се у великој мери ослања на аутентификацију, којом се потврђује да клијентска апликација поседује безбедан идентитет и да је дозвољено да користи *API*.
- **Ауторизација**. Ауторизација је следећи корак који укључује одређивање којим подацима и функционалностима може приступити аутентификована апликација током интеракције са *API*-јем.
- **Правилно дефинисање скупа улазних параметара.** Пројектант *API*-ја је одговоран да обезбеди да *API* успешно валидира све неопходне параметре током позива *API*, и извести о евентуалним грешкама.
- **Чишћење корисничког уноса.** Чишћењем корисничког уноса, *HTML* и *JavaScript* ознаке се бришу, а потенцијална рањивост на *cross-site scripting* се минимизира.
- **Пригушивање** (енгл. *Throttling*). Представља механизам контроле брзине позива *API* у одређеном временском периоду. Имплементира се на серверу. Користи се како би сервер ограничио број захтева ка *API*-ју које му корисник или апликација могу послати. Тиме се постиже очување ангажованих ресурса сервера, спречава злоупотреба и одржава функционално стање *API*-ја без прекомерног оптерећења сервера.
- **Ограничавање брзине позивања сервиса** (енгл. *Rate-limiting*). Имплементира се на клијенту. Представља ограничавање максималног броја позива сервису у јединици времена, које клијент може да упути *API*-ју у нормалном раду. Тиме се ограничава обим напада на унапред дефинисан максимални број позива сервиса. *Rate-limiting* може да ублажи *Denial of Service* (*DoS*) нападе на *API*-је.
- **Праћење активности.** Безбедност *API* укључује и праћење *API* евиденције за сумњиве активности и покушаје неовлашћеног приступа.

Сигурносни системи и алати за заштиту *API*-ја су:

- **Отворена ауторизација** (*OAuth*). *OAuth* је отворени стандард за аутентификацију и ауторизацију заснован на токенима. Пружа оквир за обезбеђење приступа *API*-ју у коме апликација треће стране доделом токена одобрава кориснику приступ

[493] https://www.techtarget.com/searchapparchitecture/definition/API-security

ресурсима на серверу[494]. Корисници могу да овласте поуздане стране (*Relying Parties*) за приступ њиховим подацима код провајдера идентитета (*Identity Providers*). Користе га *Амазон*, *Фејсбук*, *Гугл* и други, што га чини једним од најчешће коришћених система за једнократну пријаву на вебу.

- **Мултифакторска аутентификација.** Мултифакторска аутентификација је безбедносни систем за проверу идентитета корисника који захтева више од једне методе аутентификације.
- **Протокол за обезбеђење сигурности транспортног слоја** (енгл. *Transport Layer Security – TLS*). Обезбеђује аутентификацију, приватност и интегритет података између апликација које међусобно комуницирају. То је најраспрострањенији безбедносни протокол за веб претраживаче и друге апликације којима је потребна безбедна размена података преко интернета.
- ***Security Assertion Markup Language – SAML***. *SAML* је отворени стандард за дељење безбедносних информација о идентитету, аутентификацији и ауторизацији у различитим системима. *SAML* обезбеђује оквир за имплементацију јединственог пријављивања и федеративних система идентитета.
- ***API* менаџери**. Надгледају *API*-је и обезбеђују безбедно и скалабилно окружење за њихов рад. Циљ управљања је надгледање животног циклуса *API*-ја и стварање услова да потребе апликација које користе *API* буду задовољене.

Најбоље праксе и препоруке за постизање сигурности *API* полазе од аутентификације и ауторизације клијента. Сви *API* позиви морају бити адекватно аутентификовани и ауторизовани, тако да само клијенти с валидним идентитетима и одговарајућим привилегијама имају приступ серверу. Обавезна је примена *SSL* и *TLS* шифровања комуникације при позиву *API*. Сви позиви ка *API* би требало да прођу процес валидације, како би се осигурало да су подаци који се преносе исправно форматирани и у складу са очекиваном шемом. За постизање веће сигурности користи се *API* гејтвеј који пружа аутентификацију, рутирање, ограничење брзине и друге безбедносне функционалности, постављајући заштитни омотач око *API*-ја. Безбедно чување *API* кључева је од суштинске важности. Кључеви би требало да остану приватни и никада не смеју бити укључени директно у *API*. Редовно одржавање *API*-ја и ажурирање је кључно за очување безбедности. Применом наведених пракси и препорука може се обезбедити висок ниво сигурности за *API*-је.

22.4 Имплементација API

Након пројектовања и избора технологије за имплементацију *API*, прелази се на имплементацију модула и сервиса којима се приступа преко *API*. Процес имплементације обухвата најчешће следеће кораке:

[494] Fett, D., Küsters, R. & Schmitz, G. (2016, October). A comprehensive formal security analysis of OAuth 2.0. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (pp. 1204–1215).

1. Кодирање.

Кодирање је генерисање изворног и извршног програмског кôда за *API*, кроз развој функционалности, управљање грешкама, израду документације, верзионисање *API*-ја и оптимизацију перформанси.

2. Развој функционалности.

Развој функционалности се односи на имплементацију пословне логике модула или *API* сервиса, задате у процесу пројектовања на основу извршне спецификације *BPML*, преко *Domain Driven Design* спецификације, коришћењем *UML* дијаграма, или на неки други начин.

3. Управљање грешкама.

Управљање грешкама се односи на понашање *API* при појави грешке у програму, грешке у подацима, раду софтвера у неочекиваним условима, мањку ресурса или губитку комуникације. Грешке у раду софтвера називају се и изузецима (*exceptions*). Тестирање током имплементације је важно за откривање и исправљање грешака. Први корак у управљању грешкама је идентификација скупа могућих грешака и дефинисање адекватних порука о грешци за кориснике и програмере. *API* би требало да има могућност идентификације грешке и враћања *http* статусних кодова с порукама о насталој грешци. При генерисању поруке о насталој грешци, може доћи до слања осетљивих информација о сервису, као што су имена сервера, оквира, класа, верзија и *SQL* упита који се користе. Хакери ове информације могу злоупотребити. Да би се то избегло, при имплементацији сервиса користи се *API* гејтвеј који стандардизује поруке о грешкама и филтрира слање осетљивих информација које се односе на имплементацију сервиса.

4. Израда *API* документрације.

Документација описује начин употребе *API*-ја. При изради документације, потребно је описати функционалности *API*-ја, путање захтева, параметре, формате података и одговоре сервера. Добро структурирана документација обезбеђује лак приступ информацијама о сервису и разумевање начина његове употребе. Документација треба да укључи и примере кôда за позив *API* у различитим програмским језицима. *API* се у времену може мењати у складу с потребама корисника, па се континуално одржавање и ажурирање документације подразумева. Постоје више стандарда за документацију *API*-ја. Један од њих је *OpenAPI*[495], који дефинише јединствен начин описа *REST API*, пружајући детаљан преглед ресурса, крајњих тачака, параметара и одговора.

5. Верзионисање *API*-ја.

Верзионисање *API*-ја је део имплеметације који подразумева подршку различитим верзијама *API*-ја и њиховој еволуцији без угрожавања начина функционисања постојећих клијената. Мало је вероватно да ће *API* током времена остати статичан. Како се пословни захтеви развијају, требало би у сервис додавати нове колекције ресурса, мењати односе и модификовати структуре података. Главни задатак верзионисања *API* је омогућавање постојећим клијентским апликацијама да наставе нормално функционисање, а новим корисницима коришћење

[495] https://www.openapis.org/

предности нових додатих функција/ресурса. Верзионисање омогућава да *API* клијенту назначи више својих верзија, с карактеристикама и ресурсима за сваку верзију засебно, а клијент може позиве усмеравати на одређену верзију *API*. Ово омогућава да корисници старе верзије наставе нормално да користе *API* и када се у нову верзију *API* унесу озбиљне промене. Најчешће коришћени приступи за верзионисање *API* су[496]:

1) *URI* верзионисање.

Подразумева укључивање верзије *API* у путању до ресурса (на пример, */v1/endpoint)*, чиме се омогућава лака идентификација и изолација различитих верзија.

2) Верзионисање у оквиру заглавља.

Подразумева креирање прилагођеног заглавља које указује на верзију *API* (на пример, *Custom-Header: api-version=1*). Кроз овакав приступ, клијенти могу назначити верзију *API*-ја коју користе.

3) *Media Type* верзионисање.

Клијент у оквиру *http* позива сервиса, у делу заглавља *Accept* може серверу послати формат садржаја ресурса (*Media Type*) који очекује. Приликом дефинисања садржаја ресурса, *Accept* може укључити и информације које омогућавају клијенту да назначи коју верзију ресурса очекује (на пример, *Accept: application/vnd.haveibeenpwned.v2+json*).

6. Оптимизација перформанси.

Оптимизација перформанси је последњи корак у имплементацији *API*. Побољшава брзину и ефикасност размене података између клијента и сервера. Кеширање, компресија података и управљање ресурсима су технике које се често у ту сврху примењују. Кеширање на клијенту или серверу, у оквиру путање захтев-одговор, омогућава складиштење и локално коришћење копија података којима се често приступа. Врсте кеширања које се у ту сврху користе су меморијско кеширање, диск кеширање и кеширање на нивоу мрежних сервиса. Постоји више оквира, технологија и програмских пакета који олакшавају имплементацију кеширања у меморији или на диску клијента или сервера, као што су *Memcached*[497] и *Redis*[498]. Кеширање на апликативном нивоу мрежних сервиса доприноси смањењу времена одзива за захтеве клијената и унапређује опште перформансе система (смањење оптерећења на серверу и мрежи). *HTTP* протокол садржи уграђени оквир за кеширање који се лако може имплементирати додавањем заглавља које се односи на могућност кеширања оног ресурса којем се приступа[499].

[496] https://github.com/dwyl/learn-api-design?tab=readme-ov-file#versioning-your-api
[497] https://memcached.org/
[498] https://redis.io/
[499] https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching

22.4.1 Code first и API first приступи имплементацији

Code-first

Code-first приступ имплементацији *API* почиње писањем кôда којим се имплементира логика и функционалност *API*-ја. *Code-first* приступ наглашава приоритете „основне функционалности“ уместо „интерфејса“. Суштина имплементације *API*-ја вођена је кôдом. *API* се дефинише након писања кôда. Документација за *API* се формира из самог кôда[500].

Code-first приступ подразумева да се *API* развија у складу с почетним пројектним захтевима које дефинише произвођач софтвера, не узимајући у обзир потребе будућих корисника *API*-ја. Корисник од произвођача софтвера добија готове *API* спецификације. Интеракција између корисника *API* и програмера током развоја софтвера је минимална или не постоји. Недостатак комуникације с корисником, касније доводи до многих проблема у интеграцији и експлатацији.

API-first

Наглашавање значаја апликационих програмских интерфејса у развоју софтвера довело је до *API-first* приступа у развоју софтвера. *API-first* приступ је методологија развоја софтвера која ставља нагласак на пројектовање и имплементацију софтверских решења које почиње од развоја апликационих програмских интерфејса, пре него што рад на било којој другој функционалности започне. Ова методологија се фокусира на дефинисање јасних и добро пројектованих *API*-ја пре него што се развију друге компоненте. Основна идеја иза *API-first* приступа је да омогући лакше управљање променама, побољша модуларност и омогући тимовима да паралелно раде на развоју софтвера. Ово је агилнији процес од *Code-first* приступа јер укључује кориснике *API* у развојни процес. Овај приступ такође олакшава интеграцију с другим системима, јер је *API* централна копонента за комуникацију и интеграцију различитих апликација и сервиса[501] .

API-first је новија метода развоја софтвера која није потпуно истражена ни методолошки дефинисана, као што су то друге савремене методе развоја софтвера, па у будућности треба очекивати значајна побољшања овог приступа.

22.5 Тестирање API

Тестирање *API* омогућава идентификацију и решавање потенцијалних проблема пре него што се *API* постави у продукционо окружење. За разлику од тестирања корисничког интерфејса, које се фокусира на валидацију из перспективе коришћења апликације, тестирање *API* се фокусира на анализу и проверу

[500] Tucci M. (2023, April 24). Code-First vs. Design-First: Eliminate Friction with API Exploration. Swagger. https://swagger.io/blog/code-first-vs-design-first-api/
[501] https://www.postman.com/api-first/

његове функционалности. Пажљиво конструисани тестови, који обухватају и функционалне и нефункционалне аспекте, осигуравају стабилност и квалитет *API*-ја, и омогућавају безбедну и ефикасну интеграцију различитих софтверских система.

У оквиру тестирања, анализирају се очекиване функционалности, безбедност, перформансе и поузданост. Тестови се изводе директно на *API*, или као део интеграције *API* у конкретну апликацију. Тестирање *API* се изводи упућивањем захтева на једну или више крајњих тачака и упоређивањем одговора са очекиваним резултатима[502]. Процес тестирања захтева потпуно разумевање функционисања и проверу свих операција и могућности које *API* пружа. Тестирањем и исправљањем грешака обезбеђује се да сви *API* позиви буду тачно обрађени, параметри правилно интерпретирани, а резултати прецизни. Тестирају се очекивани *http* статусни кодови за успешне и неуспешне захтеве. Дефинисање очекиваних *http* статусних кодова за неуспеле захтеве помаже у идентификацији и решавању проблема током интеграције. Јасно дефинисане поруке о грешкама у телу неуспешног захтева олакшавају дијагностику и решавање проблема. Коришћење одговарајућих алата за тестирање *API*-ја, као што су *Postman*, *Insomnia*, или слични, омогућава аутоматизацију тестова и ефикасно праћење перформанси.

За потребе тестирања генеришу се тестни случајеви (*test cases*) који дефинишу услове или променљиве помоћу којих се може утврдити да ли *API* реагује на одговарајући начин и да ли исправно ради. Тестови анализирају: потврду овлашћења, време потребно да се добије одговар са сервера, квалитет одговора, *http* статусне кодове, како *API* реагује на очекивано оптерећење корисника и да ли корисници могу на неочекиване начине утицати на апликацију. Тестни случајеви треба да обухвате перформансе и сигурност, пружајући комплетну слику о стабилности и поузданости *API*-ја.

Врсте тестирања које се користе за тестирање *API* су [503,504]:

1. **Валидационо тестирање.** Тестира се да ли *API* враћа очекиване резултате и да ли су они у исправном формату. Проверава се да ли су улазни параметри, излазни параметри и тип података тачни.
2. **Функционално тестирање.** Функционално тестирање проверава да ли *API* функционише исправно и да ли испуњава захтеване спецификације. Тестира се пословна логика, валидација улаза, излаза и управљање грешкама.
3. **Тестирање оптерећења.** Тестирање оптерећења је провера перформанси и стабилности *API*-ја у стресним условима. Симулира се велики број захтева у сценаријима великог оптерећења, како би се проверило да ли *API* може да обради потребан број истовремених захтева и како се понаша при преоптерећењу.
4. **Безбедносно тестирање.** Има за циљ да идентификује безбедносне пропусте

[502] https://www.techtarget.com/searchapparchitecture/definition/API-testing
[503] https://www.techtarget.com/searchapparchitecture/definition/API-testing
[504] https://www.imperva.com/learn/application-security/api-testing/

и недостатке и обезбеди да *API* испуњава потребне стандарде безбедности. Укључује тестирање рањивости као што су *SQL Injection*, *cross-site srcipting, CSRF* и друге.

5. ***Penetration* тестирање.** Представља врсту тестирања безбедности која укључује симулацију напада хакера, мрежно скенирање, скенирање рањивости и ручно тестирање пенетрације, да би се откриле рањивости и слабости у *API*-ју.
6. ***Fuzz* тестирање.** Ова врста тестирања подразумева намерно уношење у *API* великих количина насумичних података (који се називају шум или *fuzz*), у покушају да се изазове негативно понашање или принудно рушење сервиса. На тај начин се могу открити безбедносни пропусти или неочекивано понашање *API*-ја.
7. ***Unit* тестирање.** *Unit* тестирање (Тестирање модула) је процес тестирања у којем се модули или делови сервиса, који се називају јединице (*unit*) појединачно и независно проверавају. *Unit* тестирања *API*-ја представља тестирање појединачних крајњих тачака посебним захтевима.
8. **Интеграциони тестови.** Интеграциони тестови су врста тестова у којима се компоненте, модули и сервиси у оквиру апликације тестирају као интегрисани ентитети. Тест интеграције анализира да ли *API* на одговарајући начин повезује компоненте, модуле и сервисе и квалитет њихове интеграције у целокупни софтверски систем.

Алати за тестирање API обезбеђују интерфејсе с минималним захтевима за кодирање који омогућавају и мање искусним програмерима да примене тестове. Најчешће коришћени и доступни алати алати за тестирање *API* су:

Postman[505] је алат који се користи за тестирање и развој *API*. Омогућава корисницима да креирају *http* захтеве, прослеђују их серверу и анализирају одговоре. *Postman* обезбеђује аутоматизацију тестирања и симулацију различитих услова рада сервиса. Издваја се по могућностима за аутоматизацију тестирања, укључујући тест-скрипте и колекције за организацију *API* позива[506].

Swagger[507] је алат који се фокусира на документовање и пројектовање *API*. Он омогућава развојним тимовима да креирају чисту и јасну документацију за *API*-је, што олакшава коришћење и интеграцију *API*-ја у апликације. *Swagger* такође пружа могућност за тестирање *API*-ја преко свог интерфејса, што доприноси његовој универзалности и удобности. Фокус на документацији и *OpenAPI* стандарду разликује га од других алата[508].

Apache JMeter[509] је бесплатна Јава апликација отвореног кôда која ради на свим оперативним системима. Алат не захтева вештине програмирања и може да обрађује

[505] https://www.postman.com/automated-testing/
[506] Westerveld, D. (2021). *API Testing and Development with Postman: A practical guide to creating, testing, and managing APIs for automated software testing*. Packt Publishing Ltd.
[507] https://swagger.io/docs/
[508] https://swagger.io/
[509] https://jmeter.apache.org/

различите типове апликација, сервиса и протокола. *JMeter* се може користити за генерисање великих оптерећења реалног саобраћаја, како би *API*-је ставио под притисак. Интеграција између *JMeter*-а и *Jenkins*-а омогућава администраторима да уграде *API* тестирање у *CI/CD pipelines* и да користе *JMeter* за праћење *API*-ја.

Insomnia[510] је *API* клијент отвореног кôда за креирање, организовање, дељење и извршавање *RESTful*, *SOAP*, *GraphQL* и *gRPC* захтева из десктоп апликације. Корисници *Insomnia* алата могу да креирају токове *API* тестова. *Inso*, интерфејс командне линије апликације, омогућава корисницима да интегришу аутоматизоване *Insomnia API* тестове у своје *CI/CD pipelines* преко *GitHub*-а, *GitLub*-а или *Bitbucket*-а.

Thunder Client[511] за *VS Code* је интегрисани алат за тестирање *API* у развојном окружењу *Visual Studio Code*. Олакшава тестирање без напуштања *IDE*, што је посебно практично за развојне тимове.

22.6 Модели зрелости API архитектура

Модели зрелости *API* архитектура представљају формални оквир за оцену квалитета пројектовања и имплементације *API*. Могу се поделити у три групе, које обуватају седам нивоа зрелости, приказних у Табели 4.

Табела 4. Модели зрелости API

Нису засновани на *API*			**Делимично засновани на *API***		**Потпуно засновани на *API***	
Нивои						
1	**2**	**3**	**4**	**5**	**6**	**7**
Изоловане апликације	Неструктурирана интеграција	*Component-based* архитектура	Сервисно-оријентисане архитектуре	Приватни *API* засновани на микросервисима	Отворени *API*	*API* као пословна стратегија
Карактеристичне технологије						
Монолитне апликације	Сокети *File Sharing* *MSG Queues* *DB Replications*	*EJB* *RMI* *COM/COM+* *CORBA* Веб сервиси	*ESB* *BPEL* *CEP* *BAM* *SOAP*	*REST* *AMQP* *API* гејтвеј Контејнери	*API platform* *Dev Portal* *WAF* *REST* *OAS*	*API platform* *Service Mesh* *Cloud* *GraphQL* *gRPC*

Групе модела и нивои зрелости архитектуре *API* су:

1. **Модели зрелости архитектура које нису засноване на *API*.**

   У овој групи, с нивоима зрелости 1–3, у апликацији не постоји *API*. У неким случајевима не постоји никаква комуникација између апликација. За комуникацију апликација користе се поруке, неструктурирани веб сервиси, *Unix Socket* или *TCP/Socket* технологије. Конкретно, на првом нивоу се налазе

[510] https://insomnia.rest/
[511] https://www.thunderclient.com/

изоловане апликације без интеграције с другим системима, као што су монолитне апликације попут *ERP* и *CRM* система. На другом нивоу зрелости архитектуре *API*, интеграција је неструктурирана, није заснована на стандардима и врши се *ad hoc*, односно само када се појави проблем. Трећи ниво зрелости се односи на архитектуре софтвера засноване на компонентама које се сусрећу с проблемом лоше интероперабилности.

2. **Модели зрелости архитектура које су делимично засноване на *API*.**

   У овој групи, с нивоима зрелости 4–5, апликације су делимично засноване на *API*, примењују се *SOAP* и *RESTful* веб сервиси, али с ниским нивоом управљања, стандардизације и раздвајања одговорности. На четвртом нивоу зрелости су апликације са сервисно-оријентисаном архитектуром, док пети ниво обухвата апликације с микросервисном архитектуром. Апликације на овом нивоу користе *API* за интеграцију између *front-end* и *back-end* делова апликације. Кључно питање везано за *API*-је је управљање. Користе се само неке врсте управљања које обезбеђују *API* гејтвеји. Комуникација између микросервиса је од тачке до тачке. *API*-јима микросервиса се не управља.

3. **Модели зрелости архитрктура које су у потпуности засноване на *API*.**

   Код ове групе, с нивоима зрелости 6–7, постоји јасно раздвајање одговорности између нивоа слојева *API*, сервиса и апликација. Постоје механизми безбедности, висок ниво управљање *API*-јима, праћење, аналитика и унапређено искуство развоја *API*-ја. Шести ниво обухвата отворене *API* који се налазе изнад осталих у архитектури. На овом нивоу се користе комерцијалне или отворене платформе за *API*-је, портали за развој и модули за аналитику. *RESTful API* су стандард за спољне интеграције. Седми ниво подразумева потпуно управљање спољним и унутрашњим *API*-јима, зрелу архитектуру микросервиса и инфраструктуру засновану на *cloud*-у. Користе се комуникациони протоколи као што су *REST*, *GraphQL*, *WebSockets* и *gRPC*. Предузећа на основу таквих *API*-ја могу креирати екосистем партнерства и окружење отворених иновација.

За *RESTful API*, поред наведених модела зрелости, користи се и модел зрелости који је заснован на степену сагласности с препорукама и ограничењима за *REST* архитектуру. У пракси, пројектанти не користе све препоруке и ограничења за *RESTful API*. За одређивање нивоа зрелости архитектуре, зависно од поштовања појединих *REST* препорука, Леонард Ричардсон је предложио скалу с три нивоа[512]:

- Ниво 1, коришћење ресурса. Уместо да се приступа сложеном сервису преко једног апликационог програмског интерфејса, приступ сервису се дели на више ресурса, којима се приступа појединачно.
- Ниво 2, униформни интерфејси. Коришћење *http* метода *POST*, *GET*, *PUT*, *PATCH* и *DELETE*, уводи се стандардни скуп наредби и реакција на поруке за *CRUD* операције.
- Ниво 3, контрола стања ресурса путем хипермедија[513]. Сервер динамички

[512] Richardson, L., Amundsen, M. & Ruby, S. (2013). *RESTful Web APIs: Services for a Changing World.* O'Reilly Media, Inc.
[513] https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

припрема хиперлинкове за промену стања ресурса (*Hypertext As The Engine of Application State – HATEOAS*), уводи се могућност откривања ресурса и могућност самодокументовања.

Честа је ситуација да пројектанти софтвера тврде да су њихови апликативни програмски интерфејси *RESTful*, иако не задовољавају наведене препоруке и ограничења. Највећи број пројектованих веб сервиса задовољавају Ниво 2. За пуну имплементацију Нивоа 3 потребни су протоколи и инфраструктура који нису стандардизовани и који се по комплексности могу поредити са *SOA* архитектуром[514].

23. ПАТЕРНИ ИНТЕГРАЦИЈЕ

Информациони системи великих предузећа састоје се од стотина, ако не и хиљада, сервиса и апликација које су прилагођаване, набављене од треће стране, или су део наслеђеног система, које функционишу на више слојева различитих оперативних платформи. Интеграција неопходних компоненти и одржавање апликација, у таквим системима постаје захтевнији процес од пројектовања и развоја апликације.

Развој софтверског инжењерства у великој мери су одредиле потребе великих предузећа за уређењем модуларне архитектуре својих информационих система и патернима интеграције који су засновани на провереним примерима добре праксе.

Кренуло се од покушаја креирања једне велике апликације која покрива све пословне потребе предузећа, а која је названа *Enterprise Resource Planning* (*ERP*) систем. *ERP* системи су донели уредност у пројектовању података и апликација[515], али су се у пракси показали као затворени за проширења других произвођача софтвера и тешки за одржавање. *ERP* се заснива на интеграцији функционалности. Нефункционални захтеви који су кључни за процес интеграције су у другом плану. То је био разлог тешкоћама и високим трошковима одржавања с којима су се корисници *ERP* система суочавали.

Да би се решили проблеми интеграције и одржавања с којима су се корисници *ERP* система суочавали, кренуло се ка дефинисању нових отворенијих приступа пројектовању који су усмерени ка решавању функционалних и нефункционалних пројектних захтева на почетку процеса пројектовања, и њиховим уграђивањем у архитектуру софтвера. При имплементацији софтвера потребно је да се поштују сва правила модуларизације и интеграције. То подразумева да се за функционалне захтеве дефинишу модули, а за имплементацију нефункционалних захтева, архитектура за њихову интеграцију. Тиме се ажурирање функционалности раздваја

[514] https://martinfowler.com/articles/richardsonMaturityModel.html

[515] Beleţ, T. & Purcărea, A. A. (2017). *The evolution of enterprise resource planning systems, International Journal of Advanced Engineering, Management and Science*, 3(12), 239942. https://www.neliti.com/publications/239942/the-evolution-of-enterprise-resource-planning-systems

од ажурирања нефункционалних захтева, повећава скалабилност, квалитет и омогућава континуална интеграција. То су најзначајније одреднице савременог софтвера.

У животном циклусу софтвера, фокус се помера с пројектовања функционалности на пројектовање архитектуре интеграције, која постаје примарна у развоју и експлоатацији софтвера.

Интеграциони патерни, настали на примерима добре праксе, представљају структуриран и ефикасан оквир за комуникацију и сарадњу микросервиса. Подржавају централизовану интеракцију микросервиса с клијентима, као и оркестрацију микросервиса у паралелне процесе, омогућавајући предузећима да на једноставан начин граде отпорне, скалабилне и прилагодљиве софтверске системе[516] .

Разумевање начина функционисања патерна интеграције постаје важан фактор успешног пројектовања софтвера. Модуларност, балансирање оптерећења, побољшање скалабилности, изолације грешака, откривање сервиса и безбедност су уграђени у нефункционалне захтеве интеграционих патерна. Кроз коришћење добро дефинисаних интеграционих патерна, предузећа постижу ефикасност, скалабилност, поузданост и агилност у решавању комплексносних проблема интеграције, што их чини спремним за промене и иновације у динамичном пословном окружењу.

Проверени интеграциони патерни су кључни за успех комплексних система интеграције у модерном пословном окружењу, што се потврђује кроз успешну примену у великим мултинационалним предузећима као што су *Убер*[517], *ЛинкедИн*, *Нетфликс*[518], *Твитер*[519]. Њихова суштина лежи у пружању инфраструктуре и смерница за ефикасну интеграцију различитих апликација, сервиса и система у јединствену, функционалну целину.

У динамичном окружењу, где су сарадња и ефикасност од суштинског значаја, примена архитектурних патерна интеграције и овладавање технологијама интеграције микросервиса отвара се пут ка иновацијама и трансформацији пословања. Уколико се мудро примењују, интеграциони патерни уздижу архитектуру микросервиса из колекције независних сервиса у кохезиван и агилан екосистем.

23.1 Агрегатор

[516] Blinowski, G., Ojdowska, A. & Przybyłek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation, *IEEE Access*, 10, 20357–20374.
[517] https://medium.com/nerd-for-tech/uber-architecture-and-system-design-e8ac26690dfc
[518] https://www.geeksforgeeks.org/system-design-netflix-a-complete-architecture/
[519] https://blog.bytebytego.com/p/twitter-architecture-2022-vs-2012

Агрегатор је патерн интеграције који се користи за ефикасну обраду груписаних захтева[520]. Овај патерн се реализује у форми композитног *API* сервиса. Врши агрегирање више појединачних захтева *API* сервисима у један груписани захтев, који се затим обрађује као целина а резултат враћа клијенту.

Један од облика агрегатор патерна, *Batch Request* агрегатор патерн, посебно је користан у сценаријима где је потребно обрадити више захтева са сличним карактеристикама истовремено, оптимизујући коришћење ресурса и побољшавајући перформансе.

Агрегатор патерн се учестало користи у апликацијама друштвених медија, за састављање садржаја, тако што прикупља објаве, коментаре и медијски садржај из различитих микросервиса[521]. Такође, примењује се у апликацијама е-трговине, за формирање корпе за куповину агрегацијом производе из различитих микросервиса.

23.2 Bulkhead

Bulkhead патерн је композитни *API* сервис, који при свом извршењу дели микросервисе које позива у изоловане групе, или *bulkhead*-ове, како би се ограничили могући кварови при њиховом извршавању. Овај приступ спречава ланчане реакције, тј. спречава да проблеми у једном микросервису утичу на друге, обезбеђујући отпорност система[522]. *Bulkhead* патерн се може користити за ограничење броја истовремених позива сервисима на једној нити, изолацију нити на чији одговор се дуго чека и преусмеравање позива сервиса на другу нит која има бржи одговор[523].

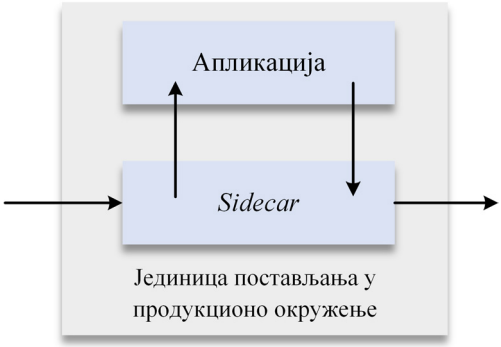
Примери употребе *bulkhead* патерна су системи финансијских трансакција, где се овај патерн користи да изолује сервисе који рукују различитим типовима трансакција. Ако се појави проблем у једном типу трансакције, он је ограничен унутар свог *bulkhead*-а, спречавајући каскадно отказивање које би могло утицати на наредне трансакције. У системима за резервисање летова, патерн *bulkhead* издваја сервисе који управљају различитим врстама резервација. Ако дође до проблема у једној врсти резервације, она остаје ограничена унутар свог *bulkhead*-а, спречавајући да се проблем прошири на друге типове резервација.

23.3 Sidecar

[520] Kousiouris, G. (2021, October). A self-adaptive batch request aggregation pattern for improving resource management, response time and costs in microservice and serverless environments. In: *2021 IEEE International Performance, Computing, and Communications Conference (IPCCC)* (pp. 1–10). IEEE.
[521] Zignani, M., Esfandyari, A., Gaito, S. & Rossi, G. P. (2016). Walls-in-one: usage and temporal patterns in a social media aggregator, *Applied Network Science*, 1, 1–24.
[522] Nygard, M. (2018). *Release it!: design and deploy production-ready software*. Pragmatic Programmers.
[523] Hosea, E., Palit, H. N. & Dewi, L. P. (2021). Fault Tolerance pada Microservice Architecture dengan Circuit Breaker dan Bulkhead Pattern, *Jurnal Infra*, 9(2), 227–233.

Sidecar патерн је композитни *API* сервис који служи за раздвајање нефункционалних захтева који се реализују у оквиру *Sidecar API* сервиса, од функционалних који се реализују у микросервису који извршава апликациону логику. Омогућава независно ажурирање апликационе логике од паралелизације, скалабилности, поузданости, сигурности, управљање мрежним саобраћајем и других послова.

Примењује се тако што се микросервисима апликационе логике додели *Sidecar* композитни *API* сервис, који прима позиве клијената, брине се о нефункционалним захтевима и управљању мрежним саобраћајем, а фунционалне захтеве прослеђује сервису апликационе логике (Слика 70)[524].



Слика 70. Sidecar патерн

Овај патерн се користи у апликацијама које захтевају висок ниво безбедности, за управљање аутентификацијом, ауторизацијом, енкрипцијом и декрипцијом, јер омогућава да се безбедносне функције имплементирају без мешања у постојећа права приступа и логику микросервиса. Такође, овај патерн се користи за послове надгледања и бележења активности у сложеним апликацијама које захтевају детаљно праћење перформанси и анализу логова. На тај начин, сваки микросервис добија свој алат за надгледање и логовање, што олакшава откривање грешака и оптимизацију перформанси система без мешања у основну логику микросервиса. Још један пример примене су медија стриминг апликације, где *sidecar* патерн олакшава транскодовање, тј. конвертовање дигиталног медијског садржаја из једног формата у други.

23.4 Прокси и реверзни прокси

[524] Koschel, A., Bertram, M., Bischof, R., Schulze, K., Schaaf, M. & Astrova, I. (2021, July). A look at service meshes. In: *2021 12th International Conference on Information, Intelligence, Systems & Applications (IISA)* (pp. 1–8). IEEE.

Прокси

Прокси патерн је интеграциони патерн у форми композитног *API* сервиса. Не мора бити део апликације, већ може бити део инфраструктуре која опслужује више различитих корисника, апликација и сервиса.

Прокси опслужује више клијената на локалној мрежи, с више улазних тачака, чије позиве може преусмерити на више сервера на интернету или некој другој мрежи. Примарни циљ му је да као посредник раздвоји чврсту везу између клијента и сервера и тиме побољша модуларност, интеграцију, скалабилност и поузданост. Може на себе преузети и послове аутентификације, ауторизације и криптовања мрежног саобраћаја. Прокси се може употребити и за расподелу оптерећења тако што распоређује захтеве на више истоветних инстанци сервиса, оптимизујући коришћење ресурса и спречавајући преоптерећење појединачних инстанци сервиса.

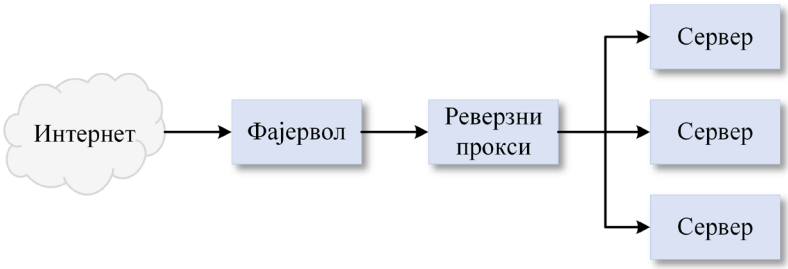
Прокси патерн није ограничен на *API* позиве и може се користити за преусмеравање различитих врста мрежног саобраћаја, као што су веб сервери, електронска пошта, фајл сервери и друго.

Реверзни прокси

Реверзни прокси је интеграциони патерн у форми композитног *API* сервиса, који се користи за рутирање екстерних захтева ка дистрибуираним серверима на локалној мрежи (Слика 71). За разлику од прокси сервера, где се клијент налази на локалној мрежи, код реверзног прокси сервера, клијент се налази на интернету или екстранету, а сервери су на локалној мрежи. Одатле потиче и његов назив. Намењен је широком скупу апликација (имејл, веб, дистрибуирани фајл системи). Може се користити и за рутирање приступа *API*. Транспарентан је за екстерне процесе с јединственом мрежном *API* адресом.

За реверзни прокси се користи и израз *Ingress Controller* (нпр. у *Kubernetes*-у). Неке од познатих имплементација су: *Envoy*[525], *HAProxy*[526], *Kong*[527], *Nginx*[528], *Traefik*[529] и други.

[525] https://www.envoyproxy.io/
[526] https://www.haproxy.org/
[527] https://konghq.com/
[528] https://nginx.org/ru/
[529] https://traefik.io/traefik/



Слика 71. Реверзни прокси

Осим расподеле оптерећења, примена реверзног проксија доприноси заштити од напада. Одредишна адреса сервиса, није јавно изложена, њу зна само прокси, тако да нападачи тешко могу директно нападати сервис. Нападима је изложен реверзни прокси, па посебну пажњу треба посветити његовој заштити.

Реверзни прокси може да кешира садржај који је у *REST* позиву дозвољен за кеширање, што резултује бољим перформансама. Постоји могућност конфигурације реверзног проксија за обављање *SSL* енкрипције, чиме се одредишни сервер растерећује бављења енкрипцијом.

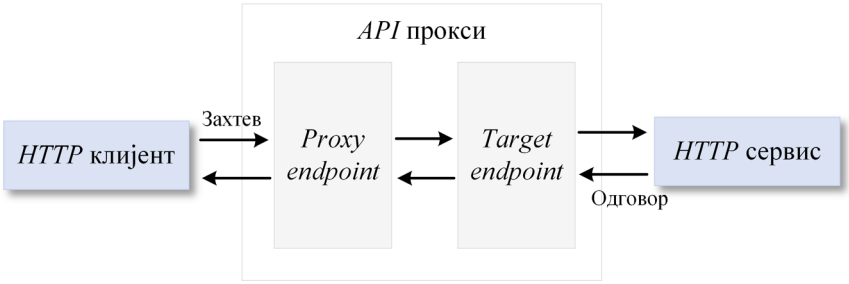
Недостатак реверзног проксија при рутирању *API* позива је то што није у могућности да манипулише аргументима и врши трансформацију и прилагођавање различитим *API* на одредишном серверу. За такве намене се користи архитектурни патерн прилагођен *API* рутирању, који се назива *API* прокси.

23.5 API прокси

API прокси[530] је интеграциони патерн који је намењен искључиво преусмеравању саобраћаја према *API* сервисима. Раздваја *API* који се излаже клијентским апликацијама и *API* који се користи за приступ *back-end* сервисима (Слика 72). Део *API* изложен клијентским апликацијама назива се *ProxyEndpoint* и дефинише начин на који клијентске апликације користе *API*. Садржи *API* прокси *URL*, начин приступа (*http*, *https*) и сигурносне политике (контрола приступа, токени, квоте). Део изложен *back-end* сервисима назива се *TargetEndpoint* и дефинише интеракцију *API* проксија са *back-end* сервисима.

API прокси омогућава паралелизацију, расподелу оптерећења између микросервиса, повећање скалабилности, сигурности, поузданости и дозвољава могућност промене *API* спецификације *back-end* микросервиса у реалном времену, а да то нема утицаја на клијентску апликацију. За компликованије случајеве позива сервиса, у којима се захтева и трансформација протокола позива, или врсте сервиса, користи се *API* гејтвеј[531].

[530] https://konghq.com/learning-center/api-management/what-is-an-api-proxy
[531] https://www.f5.com/glossary/api-gateway



Слика 72. API прокси

API прокси се на основу сервиса које нуди, и критеријума усмеравања позива, може реализовати преко једноставнијих и специфичнијих патерна, као што су: *Gateway Routing* патерн, *Chained Microservice* патерн и *Branch* патерн.

***Gateway Routing* патерн**

Gateway Routing патерн је упрошћени облик *API* прокси патерна који има само једну улазну тачку за позиве микросервиса (једну *IP* адресу на једној мрежи) које на основу дефинисаног критеријума преусмерава (рутира) ка више микросервиса на другим мрежама. Користи се када је потребно изложити више микросервиса путем једне крајње тачке и усмерити их ка интерним позадинским микросервисима. Често се користи у сложеним апликацијама где се више микросервиса комбинује како би се пружила одговарајућа функционалност клијентима.

Може се користити за усмеравање микросервиса на *cloud*-у, тако што се захтеви динамички усмеравају на основу локације, стања сервиса или специфичних захтева клијената[532]. *Gateway Routing* патерн се може применити и за усмеравање корисника према одговарајућим микросервисима на основу претходних куповина, преференција производа или локације.

***Chained Microservice* патерн**

C*hained Microservice* патерн је облик *API* прокси патерна који оркестрира секвенцу микросервиса, како би се обавила одређена операција или пословни процес. Сваки микросервис у ланцу обавља различиту функцију, доприносећи целокупном задатку. Ланац микросервиса не би требало да буде предугачак. Термин *singleton chain* односи се на ланац који садржи један микросервис и може се проширивати у будућности. Овај патерн примењује се у системима за обраду поруџбина, за управљање секвенцом микросервиса одговорних за валидацију поруџбине, проверу инвентара и обраду плаћања; у системима за управљање корисничким налозима, за координацију низом микросервиса који се баве регистрацијом

[532] Swarnalatha, K.S., Mallya, A., Mukund, G. & Ujwal Bharadwaj, R. (2023). Solving Problems of Large Codebases: Uber's Approach Using Microservice Architecture. In: Shetty, N.R., Patnaik, L.M., Prasad, N. H. (eds): *Emerging Research in Computing, Information, Communication and Applications. Lecture Notes in Electrical Engineering*, vol 928. Springer, Singapore. https://doi.org/10.1007/978-981-19-5482-5_57

корисника, провером идентитета и управљањем привилегијама приступа, и у другим системима.

***Branch* патерн**

Branch патерн је облик *API* прокси патерна који на основу дефинисаног критеријума омогућава обраду више различитих захтева ка једном микросервису и њихово усмеравање ка различитим позадинским микросервисима. То је нарочито корисно код верзионисања микросервиса, када *Branch* патерн омогућава различитим апликацијама да извршавају различите верзије истог микросервиса које су њима прилагођене. Гранање микросервиса се може имплементирати коришћењем верзионисаних *API*-ја или одвојених микросервиса. У е-трговини, *branch* патерн може се користити за персонализоване препоруке производа на основу историје куповине и преференција корисника.

23.6 API гејтвеј

API гејтвеј је најзначајнији и најкомплетнији интеграциони патерн и компонента инфраструктуре која има за циљ прилагођавање и стандардизовање комуникције између клијената и сервиса који функционишу на различитим апликацијама, платформама и с различитим типовима сервиса у оквиру једног предузећа. Његова примарна улога је да делује као јединствена улазна тачка која, независно од технологије, типа сервиса, начина организације и складиштења података, омогућава стандардизовани процес интеракције између апликација и сервиса. *API* гејтвеј патерн помаже у решавању проблема као што су[533]: сигурност, ограничавање брзине, кеширање, избегавање непотребне комуникације између клијената и микросервиса, хетерогеност клијената, прослеђивање захтева позадинским микросервисима, откривање функционалних инстанци микросервиса и детектовање недоступних инстанци микросервиса.

API гејтвеј прима *API* позиве из интерних и екстерних извора, по потреби их пакује у више захтева и усмерава их до одговарајућих микросервиса. Истовремено прима одговоре микросервиса, пакује их и испоручује као стандардизоване одговоре клијенту који је упутио захтев. *API* гејтвеј може да обезбеди аутентификацију, ауторизацију, надгледање, балансирање оптерећења и руковање одговорима микросервиса. На Слици 73 је приказан пример функционисања једноставног *API* гејтвеја.

[533] Gadge, S. & Kotwani, V. (2018). *Microservice architecture: API gateway considerations*, GlobalLogic Organisations, Aug-2017, 11.

Слика 73. Једноставан API гејтвеј

Примена *API* гејтвеја је значајно утицала на развој софтвера, интеграцију и управљање софтверским процесом. Многе његове предности су обликовале модеран приступ у развоју и испоруци софтвера. Примена *API* гејтвеја као централне тачке контроле приступа *API* сервисима и управљање нефункционалним захтевима везаних за дистрибуираност, паралелизацију, скалабилност и поузданост, значајно су поједноставили и усавршили обраду *API* захтева. Тиме је смањењена сложеност и убрзана испорука апликација, и омогућено једноставније решавање експлоатационих проблема у реалном времену, путем детаљних показатеља, контролних табли, анализе података и увида у перформансе апликација. Наведене особине чине *API* гејтвеј веома важном компонентом *IT* инфраструктуре предузећа[534] .

Примери употребе овог патерна су платформе е-трговине, где *API* гејтвеј патерн консолидује захтеве са веб и мобилних клијената и усмерава их ка микросервисима који се баве каталогом производа, аутентификацијом корисника и обрадом поруџбина. Ова централизација поједностављује интеракције с клијентима и побољшава сигурност кроз јединствену улазну тачку. *API* гејтвеј користи се и у *cloud* апликацијама, као централна тачка за интеракцију са спољним сервисима.

[534] https://www.techtarget.com/whatis/definition/API-gateway-application-programming-interface-gateway

Веб, мобилни и *IoT* клијенти шаљу захтеве преко *API* гејтвеја, који затим прослеђује те захтеве одговарајућим микросервисима на *cloud*-у[535]. Микросервиси се баве задацима као што су обрада података, анализа, складиштење и приказивање информација корисницима. Овај патерн омогућава скалабилност, сигурност и једноставно управљање саобраћајем у сложеним системима.

23.7 Откривање микросервиса

У архитектури софтвера оријентисаног на процесе, откривање микросервиса (*Microservices Discovery*) представља један од најозбиљнијих задатака и изазова. *API* прокси, *API* гејтвеј и *Service Mesh*, кључне компоненте интеграције, за обезбеђивање паралелизације, скалабилности и доступности откривају и покрећу неопходне микросервисе у реалном времену.

Откривање микросервиса није стандардизовано. Може бити на страни клијента[536], на страни сервера[537], или неко од хибридних решења [538,539]. За њихово функционисање потребно је у архитектуру апликације уградити регистар микросервиса (*Microservices Registry*)[540]. Сервиси се могу самостално регистровати (*self-registraction*) или може постојати посебна компонента за регистрацију (*third-party registration*). Сервиси се откривају у реалном времену, па је нужно пратити индикаторе квалитета њиховог рада, и уколико је потребно заменити их. Индикатор који значајно утиче на квалитет је латенција откривених и покренутих сервиса.

Откривање микросервиса може се имплементирати на више начина[541]:

- **Имплементација заснована на *DNS*-у.** Овај приступ заснива се на употреби стандардне *DNS* библиотеке као клијента. За сваки микросервис се уноси *DNS* запис. Систем за откривање микросервиса лоцира инстанце микросервиса претрагом *DNS* записа (*DNS lookups*). Овај приступ је погодан за све програмске језике, јер не тражи значајне измене у кôду. Највећи недостатак је кашњење у пропагирању *DNS* записа. Такође, трошкови одржавања могу бити високи, јер је потребна посебна инфраструктура за проверу доступности микросервиса и кеширање.
- **Имплементација заснована на бази података.** Овај приступ се заснива на евиденцији микросервиса у конзистентној кључ-податак бази података. Микросервис се повезује на систем за откривање микросервиса помоћу локалног проксија, и преузима информације потребне за конфигурисање *sidecar* проксија. Комуникација се може реализовати помоћу *REST*-а и транспарентна

[535] Stortelder, R. (2019). Enabling centralized access to a reactive architecture for hardware control systems (Master's thesis, University of Twente).
[536] https://microservices.io/patterns/client-side-discovery.html
[537] https://microservices.io/patterns/server-side-discovery.html
[538] https://microservices.io/patterns/self-registration.html
[539] https://microservices.io/patterns/3rd-party-registration.html
[540] https://microservices.io/patterns/service-registry.html
[541] https://hansheng0512.medium.com/service-discovery-in-microservices-19850a75f52d

је за програмера. Међутим, овај приступ се може користити само за откривање микросервиса на страни сервера, уз комплексну конфигурацију и одржавање *sidecar* проксија.

- **Имплементација заснована на библиотекама.** Овај приступ користи посебну библиотеку помоћу које се на клијентским апликацијама имплементирају функционалности за комуникацију са серверским компонентама система за откривање микросервиса. Функционалности библиотеке су директно изложене програмеру, и позив микросервиса се мора директно имплементирати као *API* позив. За сваки програмски језик је потребно да постоји посебно развијена библиотека.

С обзиром на то да откривање микросервиса није стандардизовано, а да је у савременим микросервисним архитектурама веома важно, за ту сврху развијени су многи системи отвореног кôда, као што су *Consul*[542], *Etcd*[543], *Apache ZooKeeper*[544], *Netflix Eureka*[545] и други. Откривање микросервиса је стандардна опција платформи за развој микросервисних апликација. Провајдери рачунарства у облаку нуде своје верзије регистара микросервиса. *Kubernetes* има уграђен регистар микросервиса базиран на *Etcd*[546].

Избор начина откривања микросервиса зависи од величине софтверског система и одабране архитектуре. Кључни критеријуми при избору су висока доступност и скалабилност. Избором развојне платформе, избор начина откривања микросервиса је у великој мери унапред одређен. Уколико регистар сервиса на изабраној платформи не задовољава, могу се изабрати нека од решења отвореног кôда. Грешке у пројектовању и каснији развој могу негативно утицати на доступност и скалабилност софтвера[547].

23.8 Обезбеђење паралелизације скалабилности и поузданости

У савременим архитектурама софтвера оријентисаног на процесе, проблеми скалабилности, паралелизације и расподеле оптерећења, измештају се изван апликационе логике и премештају у ниже слојеве архитектуре који су ближи инфраструктурном слоју. Тиме се постиже да се паралелизацијом и скалабилношћу може управљати као инфраструктуром, која је независна од корисничких сесија и апликационе логике.

---

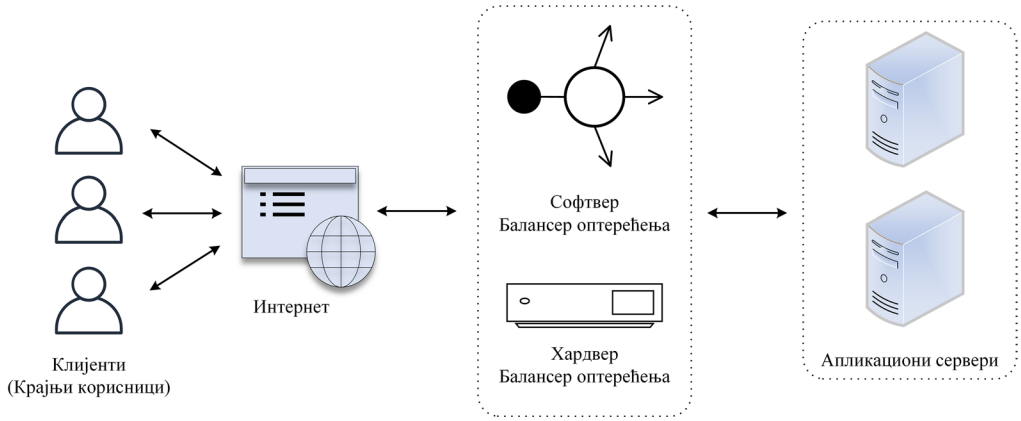
[542] https://www.consul.io/
[543] https://etcd.io/docs/v3.5/dev-internal/discovery_protocol/
[544] https://zookeeper.apache.org/
[545] https://github.com/Netflix/eureka
[546] https://dev.to/nomzykush/basic-guide-to-kubernetes-service-discovery-dmd
[547] De Alwis, A. A. C., Barros, A., Fidge, C. & Polyvyanyy, A. (2019). Availability and scalability optimized microservice discovery from enterprise systems. In: *On the Move to Meaningful Internet Systems: OTM 2019 Conferences: Confederated International Conferences: CoopIS, ODBASE, C&TC 2019,* Rhodes, Greece, October 21–25, 2019, Proceedings (pp. 496–514). Springer International Publishing. https://eprints.qut.edu.au/132556/1/132556.pdf

С обзиром на то да су клијенти дистрибуирани и за приступ *API* најчешће користе *ISO/OSI* мрежне сервисе апликационог нивоа: за пренос података *http*, а за адресирање ресурса *dns*, најпогодније је да се паралелизација и расподела оптерећења врши коришћењем архитектурних патерна који су засновани на особинама ова два протокола.

REST API садржи уграђене механизме за измештање паралелизације у ниже слојеве архитектуре. Често коришћени архитектурни патерни за расподелу оптерећења *REST API* сервиса су: реверзни прокси и *API* прокси, а при комбиновању паралелних сервиса различитих типова (нпр. *REST* и *SOAP*) користи се *API* гејтвеј. За обезбеђење поузданости, у комбинацији с паралелизацијом, могу се користити и протоколи нижих нивоа *ISO/OSI* модела, као што је нпр. *VRRP*[548].

У софтверу оријентисаном на процесе, скалабилност и поузданост се обезбеђују расподелом оптерећења на микросервисе који се паралелно извршавају. Балансер оптерећења (*Load Balancer*) је инфраструктурна компонента која се налази између клијента и сервера, која прима позиве од клијената и усмерава их ка серверима који паралелно извршавају сервисе[549]. При томе се води рачуна да сви сервери буду равномерно оптерећени, и да се увођењем нових сервера у реалном времену обезбеди потребна скалабилност.

Функционисање балансера оптерећења приказано је на Слици 74.



Слика 74. Балансирање оптерећења у NGINX[550]

Балансер оптерећења се ретко среће као независна компонента. Најчешће је део инфраструктурних компоненти као што су инверзни прокси, *API* прокси и *API* гејтвеј, који поред основне функционалности нуде и расподелу и балансирање оптерећења.

[548] https://datatracker.ietf.org/doc/html/rfc5798
[549] https://kemptechnologies.com/what-is-load-balancing
[550] Прилагођено према: https://www.f5.com/glossary/load-balancer

Расподела се врши на основу алгоритма за балансирање оптерећења. Алгоритам омогућава да се оптерећење равномерно распореди, оптимално искористе ресурси, смањи време одзива и обезбеди висока доступност апликација.

Постоје две категорије алгоритама за расподелу оптерећења: статички и динамички. Статички алгоритми се ослањају на фиксна правила и не зависе од тренутног стања сервера, док динамички алгоритми прате тренутно стање сервера и оно утиче на начин дистрибуције саобраћаја. Тиме се омогућава дистрибуирање оптерећења на начин који оптимизује перформансе система и спречава преоптерећење појединачних сервера[551].

Неки од алгоритама за балансирање оптерећења, који се користе у софтверским системима попут *Kubernetes*-а, *NGINX*-а и других, јесу: *Round Robin*, *Least Connections*, *Hashing* и *Least Time*[552]:

Round Robin је алгоритам који прослеђује захтеве клијената циклично и једнако ка свим доступним серверима. Ово је ефикасна метода када су сервери приближно једнаки у капацитету и оптерећењу, јер сваки сервер добија захтев по реду, без обзира на своје тренутно оптерећење. Предности овог приступа су његова једноставност и лакоћа имплементације. Међутим, проблеми могу настати ако сервери нису једнако способни да обраде захтеве, јер *Round Robin* не узима у обзир стварно оптерећење сервера.

Least Connections је алгоритам који усмерава захтеве клијената ка серверу с најмање активних конекција. Број конекција је приближан броју сервиса који се извршавају на серверу, па овај алгоритам тежи ка изједначавању оптерећења по серверима. Овај приступ је добар када сесије на серверу имају варијабилно трајање или различите захтеве према ресурсима, јер сервери с мање конекција могу брже обрадити нове захтеве. Предност је боља расподела оптерећења у реалном времену, али овај алгоритам може захтевати више ресурса за праћење и ажурирање броја конекција, што може бити изазов у врло великим системима.

Hashing алгоритам користи *hash* функцију за одређивање сервера на основу атрибута захтева, као што су *IP* адреса или *URL* клијента. Ово омогућава да исти клијент конзистентно буде усмерен на исти сервер, што је корисно за одржавање корисничких сесија. Предности су: конзистентност у обради захтева и ефикасност у кеширању на серверима. Главни недостатак овог метода је што може доћи до неравномерне расподеле захтева по серверима ако балансер није правилно конфигурисан, посебно у сценаријима где се саобраћај значајно мења.

Least Time алгоритам комбинује број активних конекција и време одзива сервера, да би изабрао оптимални сервер за обраду захтева. Овим се омогућује да се захтеви усмеравају не само према мање заузетим серверима, већ и према онима који брже врше обраду, што доводи до бољих перформанси и бржег времена одзива

[551] https://aws.amazon.com/what-is/load-balancing/
[552] https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/

за кориснике. Предности овог приступа укључују динамично прилагођавање променама у мрежи и оптерећењима сервера, али изазови могу настати у његовој сложености имплементације и потреби за сталним праћењем перформанси сервера како би алгоритам био ефикасан.

У пракси се за балансирање оптерећења и управљање скалабилношћу највише користи *NGINX*. Као стандардну услугу, балансирање оптерећења нуде и провајдери рачунарства у облаку *Амазон*[553], *Мајкрософт*[554], *Гугл*[555] и други.

24. СОФТВЕРСКИ ОКВИРИ, ПЛАТФОРМЕ И ЕКОСИСТЕМИ

Софтверски оквири, платформе и екосистеми представљају окружења за развој апликација које су базиране на принципима модуларности и интеграционим архитектурама. Нуде унапред припремљен скуп модула за одређене домене примене у којима се специфичности корисничких захтева имплементирају путем инверзије и уметања зависности.

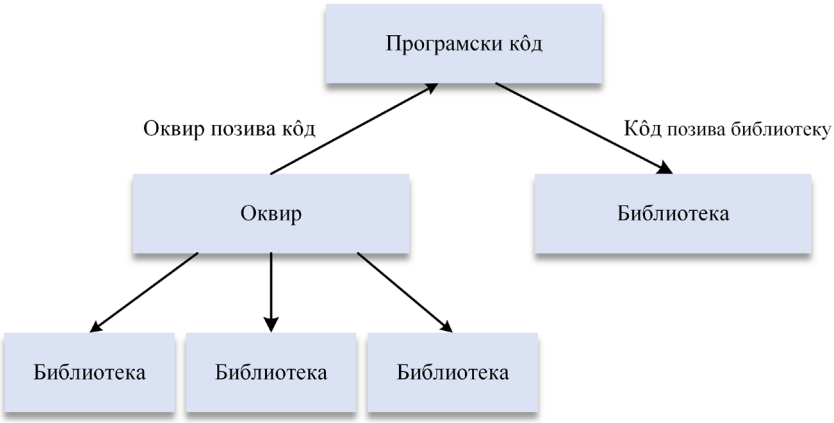
24.1 Софтверски оквири

У контексту софтверског инжењерства, оквир (енгл. *framework*) је колекција софтверских модула и компоненти које олакшавају развој нових апликација.

Софтверски оквири садрже модуле с кôдом који је заснован на одређеним софтверским стандардима и протоколима. Оквири могу дефинисати и примењивати одређена правила архитектуре софтвера, тако да нове апликације могу бити развијене на унапред дефинисан начин[556].

Библиотека је колекција класа или модула који се могу користити у кôду да би се постигле одређене функционалности без поновног кôдирања, док је оквир шири појам и представља апстрактну структуру која омогућава вишеструку употребу кôда и интеракцију компоненти у различитим апликацијама. Разлика између библиотеке и оквира је приказана на Слици 75.

[553] https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html
[554] https://learn.microsoft.com/en-us/azure/load-balancer/load-balancer-overview
[555] https://cloud.google.com/use-cases/load-balancing
[556] https://aws.amazon.com/what-is/framework/



Слика 75. Разлика између оквира и библиотеке

Осим што укључује већи број библиотека, оквир поседује механизам који управља радом и односима између библиотека. У комбинацији с додатним *plugin*-овима и алатима, оквири омогућавају оптимизацију апликација, кеширање, управљање верзијама софтвера, унапређен рад с базом података и сл. Неке од кључних предности употребе оквира су[557]:

- **Повећана ефикасност и брзина развоја софтвера.**

  Оквири представљају скуп библиотека, шаблона и алата које значајно убрзавају развој софтвера, омогућавајући програмерима да се фокусирају на пословну логику и специфичне карактеристике апликације.

- **Доступност ресурса (документација, форуми, специјализовани сајтови) и подршка.**

  Популарни оквири често имају велике и активне заједнице програмера, обимну документацију, подршку, форуме итд. Програмери могу имати користи од дељења знања у заједници и добити подршку за развој апликације. Овакав екосистем програмерима пружа богатство ресурса и убрзава процес учења.

- **Квалитет и конзистентност кôда.**

  У оквире су често уграђена искуства из најбоље праксе, чиме се обезбеђује конзистентна и одржива основа кôда. Пружају структуриран приступ програмирању, примењујући упутства за организацију кôда, обављање уобичајених задатака и одржавање модуларне архитектуре.

- **Модуларност и могућност поновног коришћења.**

  Употреба оквира омогућава раздвајање функционалности у различите модуле, што обезбеђује бољу организацију и управљање кôдом. Модули се могу поново користити у различитим пројектима или у различитим деловима истог пројекта, што убрзава развој и спречава понављање кôда. Оваква архитектура олакшава сарадњу тимова и доприноси побољшању одрживости апликација.

[557] https://www.spiceworks.com/tech/tech-general/articles/what-is-framework/

- **Сигурност и поузданост.**

  Оквири често имају уграђене безбедносне мере и искуства из најбоље праксе који помажу у заштити од уобичајених безбедносних пропуста. Користећи широко прихваћене и распрострањене оквире, програмери се могу ослонити на експертизу аутора оквира, и заједнице која оквир користи.

- **Интероперабилност.**

  Употреба широко прихваћених оквира у већој мери осигурава усаглашеност са стандардима и омогућава интероперабилност и интеграцију. Интеграција с постојећим системима или спољним *API*-јима је једноставнија, јер многи оквири нуде уграђену подршку за различите технологије и протоколе.

Истовремено, постоје и одређени изазови у примени оквира: ниједан оквир не може да обухватити све проблеме и изазове, може се јавити проблем перформанси, нису сви оквири једноставни за учење, нема пуне контроле над кôдом, развој сопствених делова кôда захтева познавање оквира.

24.1.1 Врсте оквира

Оквири могу бити подељени на две основне категорије: широко распрострањени и кастомизовани оквири. Широко распрострањени оквири укључују оне који су развијени од стране великих софтверских предузећа и оне који су настали из великих заједница програмера. Примери у овој категорији укључују *Angular* (*JavaScript* оквир развијен од стране *Гугла*), *React* (*JavaScript* оквир развијен од стране *Фејсбука*), и *Django* (*Python* оквир развијен од стране *Python* заједнице).

Кастомизовани оквири се често развијају у специфичним контекстима. Могу бити развијени као активности истраживања и развоја (*R&D*), где су најчешће у фокусу нове технологије и иновације. Такође, предузећа могу развити кастомизоване оквире за своје специфичне потребе, обезбеђујући алате и функционалности које су прилагођене њиховим пројектима и производима. Исто тако, владе и организације могу инвестирати у развој кастомизованих оквира који одговарају њиховим потребама, укључујући безбедносне захтеве и регулаторне стандарде.

Оквири се могу класификовати по следећим категоријама[558] :

1. **Оквири за веб апликације (*front-end* и *back-end* оквири).**

   Оквири за веб апликације обухватају широк спектар алата за развој веб страница и веб апликација. *Front-end* оквири као што су *React*, *Angular* и *Vue* (засновани на *JavaScript* програмском језику) омогућавају програмерима да креирају интерактивни кориснички интерфејс и управљају динамичким садржајем веб странице. На другој страни, *back-end* оквири, као што су *Django*, *Flask* (засновани на *Python* програмском језику), *Express*, *Next* (засновани на *JavaScript* језику) или *Laravel*, *Symfony*, *CodeIgniter* (засновани на *PHP* језику) пружају инфраструктуру за развој серверске логике, обраду захтева и комуникацију с базама података.

[558] https://www.spiceworks.com/tech/tech-general/articles/what-is-framework/

2. **Оквири за мобилне апликације.**

   Оквири за мобилне апликације су дизајнирани за развој апликација које се извршавају на мобилним уређајима. Укључују *Flutter* и *React Native* који омогућавају програмерима да развијају хибридне апликације користећи језике попут *JavaScript* или *Dart*.

3. **Оквири за тестирање.**

   Оквири за тестирање пружају алате и библиотеке за аутоматизацију тестирања софтвера, укључујући функционално, регресионо, тестирање перформанси и тестирање корисничког интерфејса. Популарни оквири у овој области укључују *Selenium*, *Jest* и *JUnit*.

4. **Оквири за развој игара.**

   Оквири за развој игара обухватају алате и библиотеке за креирање игара, укључујући *game engine*, алате за графику, звук и др. Примери су: *Unity*, *Unreal Engine* и *Phaser*.

5. **Оквири за машинско учење.**

   Оквири за машинско учење пружају алате и библиотеке за развој, обуку и примену модела машинског учења. *TensorFlow*, *PyTorch* и *Scikit-learn* су популарни оквири у овој области.

6. **Оквири за развој информационих система предузећа.**

   Оквири за развој информационих система предузећа (енгл. *Enterprise application frameworks*) пружају инфраструктуру за развој комплексних апликација које подржавају пословне процесе и функције предузећа. Примери укључују *Spring* оквир за *Java* програмски језик и *.NET* за *C#*.

7. **Оквири за графички кориснички интерфејс.**

   Оквири за графички кориснички интерфејс омогућавају програмерима да дизајнирају и развијају корисничке интерфејсе. Примери укључују *React* за веб апликације и *SwiftUI* за *iOS* апликације.

24.2 Софтверске платформе

Појам платформе користи се за производ, који не мора бити софтверски, а који је пројектован тако да се може лако модификовати и прилагођавати корисницима додавањем, изменом или уклањањем функционалности[559]. Овај приступ може се применити и у развоју комплексних софтверских система, где се на основној платформи могу развијати додатни и опциони модули.

[559] Wheelwright, S. C. & Clark, K. B. (1992). Creating project plans to focus product development, *Harvard Business Review* (March-April): 70–82.

24.2.1 Врсте софтверских платформи

Платформе се по намени грубо могу поделити на[560]:

1. Технолошке платформе (енгл. *Technology Platforms*).
2. Рачунарске платформе (енгл. *Computing Platforms*).
3. Услужне платформе (енгл. *Utility Platforms*).
4. Платформе за мрежне интеракције (енгл. *Interaction Networks*).
5. Електронска тржишта (енгл. *Marketplace*).
6. Платформе за услуге на захтев (енгл. *On-demand Service Platforms)*.
7. Платформе за crowdsourcing садржаја (енгл. *Content Crowdsourcing Platforms*).
8. Платформе за прикупљање података (енгл. *Data Harvesting Platforms*).
9. Платформе за дистрибуцију садржаја (енгл. *Content Distribution Platforms*).

Технолошке платформе представљају инфраструктурне или софтверске оквире који пружају основне алате, услуге или ресурсе који се могу користити у развоју различитих дигиталних производа или апликација. Примери технолошких платформи су *Amazon Web Services* (*AWS*), *Microsoft Azure* и *Twilio*. Ове платформе нису двосмерна тржишта и не повезују учеснике платформе. Технолошке платформе своје услуге продају развојним тимовима и обично су невидљиве крајњим корисницима. Омогућавају брже и ефикасније креирање и скалирање софтвера, пружају готове компоненте, као што су: ресурси рачунарства у облаку, *API*-ји, базе података и различити алати за развој софтвера. На пример, *Нетфликс* стримовање видео садржаја пружа преко *Amazon Web Services* платформе, али корисници директно комуницирају само с *Нетфликсом*.

Рачунарске платформе, за разлику од технолошких платформи, омогућавају интеракцију између корисника платформе и развојних тимова трећих страна. Примери рачунарских платформи су: *Apple iOS*, *Google Android* и *Microsoft Windows*. На модерним рачунарским платформама, веза између корисника и развојних тимова остварује се путем продавница апликација, што олакшава откривање, препоруке, активацију и монетизацију апликација.

Услужне платформе, као што су *Гугл* претрага, *Kayak* и *Zenefits*, ради привлачења што већег броја корисника пружају корисну, најчешће бесплатну услугу. Када постоји критична маса корисника, платформа се отвара и за друге типове учесника, као што су оглашивачи, код *Гугл* претраге, авио-компаније у случају *Kayak* или осигуравајуће компаније у случају *Zenefits*. Лансирање услужних платформи обично је прилично једноставно – кључно је имати корисну услугу која генерише поновну употребу и има занемарљиве маргиналне трошкове. Када постоји ресурс који је створен критичном масом корисника (нпр. подаци, циљано ангажовање итд.), платформа се отвара предузећима како би се омогућила монетизација путем различитих механизама, укључујући оглашавање и провизије.

[560] https://medium.com/platform-hunt/the-8-types-of-software-platforms-473c74f4536a

Платформе за умрежавање, пословне и социјалне интеракције представљају дигиталне платформе које олакшавају комуникацију и интеракцију између специфичних учесника, било да су у питању појединци или предузећа. Примери ових платформи су *Фејсбук*, *WeChat* и *Bitcoin*. Кључна карактеристика овог типа платформи је њихова способност да посредују у дигиталним интеракцијама различитих формата, укључујући поруке, гласовне позиве, слике или трансфер новца. Идентитет корисника представља темељ, јер су све интеракције повезане са одређеним корисничким налозима. Основни ефекат умрежавања преко ових платформи је међусобно привлачење корисника, чиме се подстиче даљи раст корисничке базе.

Електронска тржишта попут *eBay*-а, *Amazon Marketplace*-а, *AirBnB*-а или *UpWork*-а представљају дигиталне платформе које повезују понуду с потражњом и делују као двосмерни системи. Омогућавају трансакције између учесника на страни потражње (купци) и учесника на страни понуде (продавци). Цене производа и услуга које се нуде на платформи најчешће одређују учесници на страни понуде. Идентитет корисника, за разлику од платформи за мрежне интеракције, игра секундарну улогу на платформи. Купци траже одређени производ или услугу, а не одређеног продавца. Производ/услуга може бити понуђен од стране више продаваца који се такмиче по питању цене, репутације и искуства.

Платформе за услуге на захтев омогућавају корисницима да захтевају и добију различите услуге обично преко мреже независних пружалаца услуга. Пример овакве платформе је *Убер*. Ове платформе интегришу проналажење, наручивање, плаћање, испуњавање, сертификацију и потврду услуге. Цене и стандарде квалитета поставља платформа, док корисник обично има мало слободе у избору конкретног пружаоца услуге или начина испоруке. Доступност и предвидљивост услуге су кључне метрике квалитета платформи за услуге на захтев. Ефекат умрежавања преко ових платформи манифестује се у доступности услуге, где више пружалаца услуга на платформи означава бољу доступност услуге.

Платформе за *crowdsourcing* садржаја омогућавају корисницима да доприносе и конзумирају различите типове садржаја. Примери платформи овог типа су *YouTube* и *TripAdvisor*. Корисници на овим платформама сакупљају, објављују и деле садржај, као што су видео-записи, рецензије и оцене различитих услуга и дестинација. Ефекат умрежавања на овим платформама се огледа у томе да већи број креатора садржаја привлачи већи број корисника, што директно увећава вредност платформе.

Платформе за прикупљање података, као што су *Waze* и *OpenSignal*, представљају категорију дигиталних платформи које корисницима пружају услуге, док истовремено прикупљају податке о њима. Ефекат умрежавања се заснива на акумулацији и употреби података, а не само на броју корисника. Док су корисници у интеракцији с платформом, они генеришу податке који доприносе расту скупа података. С већим бројем података унапређују се функционалности и релевантност платформе, што доприноси привлачењу нових корисника који желе приступ обогаћеним функционалностима платформе.

Платформе за дистрибуцију садржаја, као што су *Google AdSense*, *Outbrain* и *Millennial Media*, повезују власнике веб сајтова, мобилних апликација и друге са креаторима садржаја који желе да поделе садржај (или огласе) корисницима. Што је више садржаја доступно на платформи, то је платформа привлачнија за кориснике.

24.2.2 Low-code софтверске платформе

Low-code платформе су софтверске платформе које омогућавају развој и имплементацију комплетних апликација с минималном употребом програмског кôда. Ове платформе се имплементирају применом концепта Платформа као Сервис (енгл. *Platform as a Service, PaaS*) рачунарства у облаку. Корисници могу приступити алатима и ресурсима за развој апликација преко интернета, без инсталирања локалног софтвера. Карактеристика *low-code* платформи је употреба напредних графичких корисничких интерфејса и визуелних апстракција, уместо писања сложеног процедуралног кôда. Ово чини процес развоја софтвера једноставнијим, бржим и приступачнијим особама које немају техничко знање програмирања[561] .

Једна од кључних карактеристика *low-code* платформи је висок ниво апстракције, што омогућава корисницима да се фокусирају на логику и функционалност апликација, а не на детаље програмског кôда. Ове платформе често укључују библиотеке и готове модуле, што додатно убрзава процес развоја. Такође, *low-code* платформе често подржавају интеграцију с другим системима путем *API*-ја и омогућавају аутоматизацију пословних процеса. Важно је имати у виду да *low-code* приступ може имати ограничења у погледу комплексности и скалабилности апликација[562].

Low-code платформе се састоје из четири главна слоја архитектуре[563]:

1. **Апликативни слој.**

   Овај слој обухвата графичко окружење с којим корисници директно комуницирају, како би спецификовали апликације. Виџети и различити алати који се користе за изградњу корисничког интерфејса су део овог слоја. Такође, овде се дефинишу механизми аутентификације и ауторизације. Корисници могу користити модуларне компоненте на овом слоју како би спецификовали понашање апликације.

2. **Слој интеграције сервиса.**

   Овај слој омогућава повезивање с различитим сервисима коришћењем

[561] Robert Waszkowski. Low-code platform for automating business processes in manufacturing, IFAC-PapersOnLine, 52(10):376–381, 2019.

[562] Frank, U., Maier, P. & Bock, A. (2021). *Low code platforms: Promises, concepts and prospects. A comparative study of ten systems* (No. 70). ICB-Research Report.

[563] Sahay, A., Indamutsa, A., Di Ruscio, D. & Pierantonio, A. (2020, August). Supporting the understanding and comparison of low-code development platforms. In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (pp. 171–178). IEEE.

одговарајућих *API*-ја и механизама аутентификације. Користи се за добијање података из екстерних извора, за манипулацију подацима и њихову анализу.

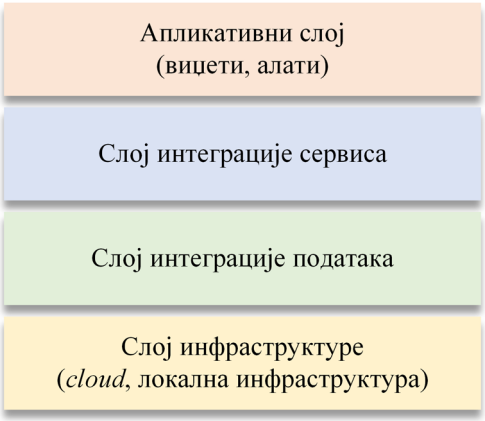
3. **Слој интеграције података.**

   Овај слој омогућава операције над подацима чак и када су извори података хетерогени. Бави се интеграцијом података из различитих извора.

4. **Слој инфраструктуре.**

   На овом слоју се развијене апликације могу имплементирати на инфраструктури рачунарства у облаку или локалним окружењима. У овом слоју се обављају процеси контејнеризације и оркестрације апликација, континуалне интеграције и постављања у продукционо окружење.

Архитектура *low-code* платформи приказана је на Слици 76.



Слика 76. Архитектура low-code платформи

Модуларна архитектура софтверске платформе омогућава децентрализовано делегирање задатака и укључивање већег броја учесника у развој компоненти платформе, чиме платформа може прерасти у отворени екосистем, тј. комплексну мрежу повезаних предузећа и појединаца, чија се интеракција одвија у физичком и дигиталном свету.

24.3 Софтверски екосистеми

Софтверски екосистем могуће је дефинисати као скуп стејкхолдера и њихових софтверских пројеката, који су развијени, функционишу, еволуирају и генеришу нову вредност на заједничкој технолошкој платформи[564].

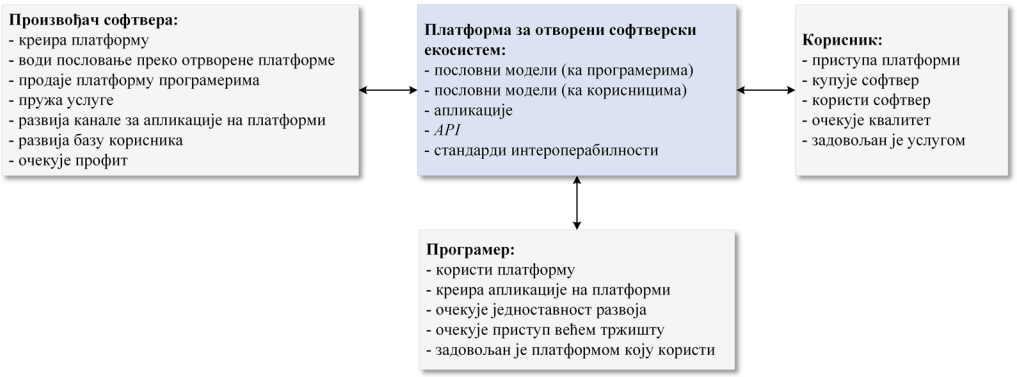
У софтверски екосистем по правилу је укључен већи број умрежених учесника:

[564] Lungu, M., Robbes, R. & Lanza, M. (2010, September). Recovering inter-project dependencies in software ecosystems. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering* (pp. 309–312).

произвођачи софтвера и хардвера, програмери-фриленсери, друга софтверска и несофтверска предузећа, провајдери садржаја на интернету, сервиси за дигитална плаћања, сервиси друштвених медија, провајдери инфраструктуре, телекомуникациона предузећа и други[565].

Произвођачи софтвера, који промовишу платформу, најчешће преузимају стратешку улогу у координацији екосистема[566], развијају нове пословне моделе који су засновани на отвореним иновацијама и *crowdsourcing*-у, одлучују у којој мери ће отворити своје производе (отворени *API*, софтвер отвореног кôда), и развијају нове стандарде и правила за сарадњу и интероперабилност на техничком и организационом нивоу[567].

Везе између учесника у отвореном софтверском екосистему су бројне и испреплетане (Слика 77), а један учесник може имати једну или више улога у екосистему. На пример, корисник више није пасивни потрошач производа или услуга, већ може активно учествовати у пројектовању или развоју (*prosumer*).



Слика 77. Везе између учесника у отвореном софтверском екосистему[568]

Типични примери екосистема су мобилни екосистеми *Андроид* и *Еплов iOS*[569] (Слика 78), затим *Salesforce*[570], *Alibaba* и други.

---

[565] Messerschmitt, D. & Szyperski, C. (2005). S*oftware Ecosystem: Understanding An Indispensable Technology and Industry* [M](Cambrige, USA).

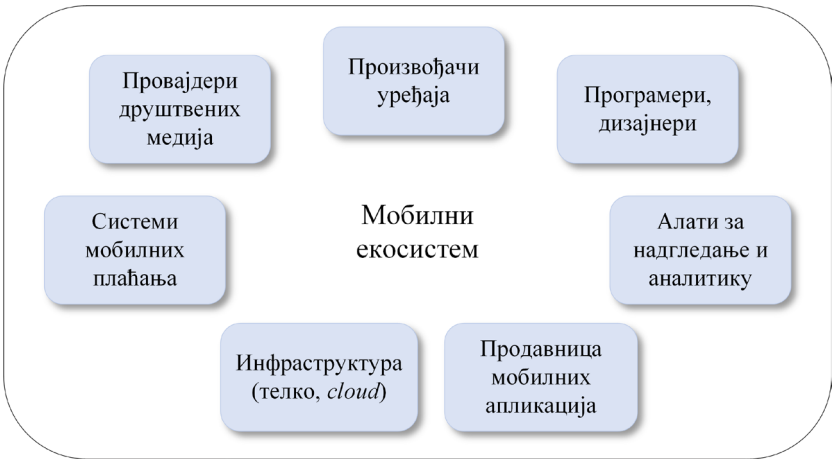
[566] Jansen, S., Finkelstein, A. & Brinkkemper, S. (2009, May). A sense of community: A research agenda for software ecosystems. In: *2009 31st International Conference on Software Engineering-Companion Volume* (pp. 187–190). IEEE.

[567] Manikas, K. & Hansen, K. M. (2013). Software ecosystems – A systematic literature review. *Journal of Systems and Software*, 86(5), 1294–1306.

[568] Прилагођено према: Yu, E. & Deng, S. (2011, June). Understanding software ecosystems: A strategic modeling approach. In: *Iwseco-2011 software ecosystems 2011. proceedings of the third international workshop on software ecosystems*. Brussels, Belgium (pp. 65–76).

[569] H. Sadi, M. & Yu, E. (2015). Designing software ecosystems: How can modeling techniques help?. In: *Enterprise, Business-Process and Information Systems Modeling: 16th International Conference*, BPMDS 2015, 20th International Conference, EMMSAD 2015, Held at CAiSE 2015, Stockholm, Sweden, June 8–9, 2015, Proceedings (pp. 360–375). Springer International Publishing.

[570] https://www.salesforce.com/blog/salesforce-ecosystem-explained/



Слика 78. Пример софтверског екосистема

Софтверски екосистеми се развијају у многим облицима, али, у основи, омогућавају учесницима да прошире вредност основне платформе до крајњих корисника, других учесника у екосистему, и наравно, до саме основне платформе.

24.3.1 Примери софтверских екосистема

***iOS* екосистем**

Екосистем *iOS*-а, развијен од стране компаније *Епл* обухвата различите уређаје, софтвер и услуге, нудећи корисницима квалитетно интегрисано искуство[571] . Кључне компоненте овог екосистема су:

1. ***iOS* уређаји.**

   iOS екосистем је изграђен око породице премијум хардверских уређаја дизајнираних за различите сврхе као што су *iPhones*, *iPads*, *MacBooks*, *Apple Watch*, *Apple TV*.

2. ***iOS* оперативни систем.**

   У средишту *iOS* екосистема налази се *iOS* оперативни систем. *iOS* је познат по добро прилагођеном корисничком интерфејсу, ефикасном коришћењу хардверских ресурса, стабилности и безбедности. Кључни концепти и функционалности који се издвајају су: редовна ажурирања, продавница апликације (*App Store*), приватност, безбедност и једноставна интеграција.

3. **Екосистем апликација.**

   Еплова продавница апликација (*App Store*) је кључна компонента *iOS* екосистема, пружајући приступ за преко два милиона апликација. Апликације развијају предузећа или индивидуални програмери, уз могућности монетизације у складу с правилима продавнице.

[571] https://www.linkedin.com/pulse/exploring-ios-ecosystem-comprehensive-guide-saurabh-anand-ocq5f/

4.*Епл* сервиси.

Епл је проширио свој екосистем уводећи различите сервисе који се једноставно интегришу са *iOS* уређајима. Неки од сервиса су *iCloud*, *Apple Music*, *Apple Pay*, *Apple Arcade*, *Apple TV+*.

***Salesforce* екосистем**

Salesforce је направљен као платформа за управљање односима с клијентима (енгл. *Customer Relationship Management, CRM*) и аутоматизацију продајних процеса. Платформа је потом проширена компонентама за аутоматизацију маркетишких активности и корисничку подршку у *B2B*, *B2B2C* и *B2C* е-трговини. Главни елементи *Salesforce* екосистема су[572]:

- *CRM* софтверске компоненте. *Salesforce* екосистем се заснива на софтверским компонентама које се корисницима нуде у форми *Software-as-a-Service* (*SaaS*) модела.
- Додатне функционалности. Укључују додатне сервисе за подршку пословним процесима продаје, услуга и маркетинга: *Sales Cloud*, *Service Cloud*, *Marketing Cloud*.
- Продавница апликација. *AppExchange marketplace* нуди могућност за продају независно развијених компоненти.
- Заједница програмера. *Salesforce* омогућује развој апликација коришћењем *low-code* платформе, уз добру подршку и доступну платформу за учење.
- Заједница партнера. Партнерска предузећа пружају услуге развоја и консалтинга у области *CRM*-а и увођења *Salesforce* решења.

25. КОНТИНУАЛНА ИНТЕГРАЦИЈА, ИСПОРУКА И ПОСТАВЉАЊЕ СОФТВЕРА У ПРОДУКЦИОНО ОКРУЖЕЊЕ

25.1 Увод и дефиниције

Континуална интеграција, континуална испорука, као и континуално постављање софтвера у продукционо окружење (*Continuous Integration and Continuous Delivery or Deployment – CI/CD*) представљају скуп поступака којима се од изворног кôда долази до активне и употребљиве апликације[573] . Континуална испорука софтвера и континуално постављање софтвера у продукционо окружење су алтернативни поступци *DevOps*-а, исте намене и функционалности, али се разликују по степену аутоматизације и учешћу људског фактора.

Континуална интеграција односи се на праксу програмирања код које се промене

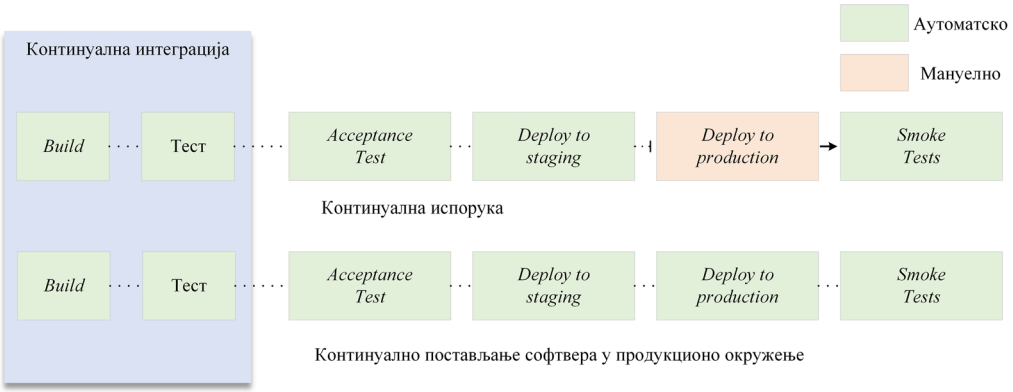
[572] https://www.linkedin.com/pulse/demystifying-salesforce-ecosystem-aakrin/
[573] https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

у деловима кôда у кратким интервалима интегришу са остатком апликације, најчешће коришћењем неког од алата за управљање верзијама кôда. Свака промена се затим валидира, интегрише и тестира, у контексту целе апликације. С обзиром на то да промене могу бити честе, неопходан услов за континуалну интеграцију је аутоматско тестирање софтвера након сваког *build*-а.

Континуална испорука софтвера укључује људски фактор у доношење одлука о фазама испоруке софтвера. Фазе континуалне испоруке софтвера најчешће представљају пролаз апликације кроз постављање и тестирање у низ одвојених окружења, од развојних, тестних, до коначног продукционог окружења у коме софтвер ради. У свакој од фаза доносе се одлуке о наредним корацима, све до коначног постављања софтвера у продукционо окружење.

Континуално постављање софтвера у продукционо окружење у потпуности аутоматизује све поступке, без директног учешћа људског фактора. Када се заврше процеси континуалне интеграције, софтвер се аутоматски поставља у продукционо окружење, аутоматизовано тестира, и уколико фаза тестирања буде успешна, софтвер постаје доступан корисницима. Континуално постављање софтвера у продукционо окружење је добар начин да се убрза добијање повратних информација од корисника и смањи притисак рока испоруке за развојни тим. Програмери се могу фокусирати на развој функционалности, а свака завршена и тестирана функционалност може се испоручити корисницима у року од неколико минута.

Разлика између континуалне испоруке и континуалног постављања у продукционо окружење приказана је на Слици 79.



Слика 79. Континуална испорука и континуално постављање у продукционо окружење [574]

[574] Прилагођено према: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

Специфичноси *CI/CD* пракси огледају се у следећем:

- Неопходно је писати аутоматске тестове за сваку функционалност, промену или поправку кôда. Очекивани резултат је софтвер с мање грешака и нижи укупни трошкови тестирања. Ова пракса смањује обим мануаленог посла *QA* тима, али се тражи веће експертско знање за писање аутоматских тестова у односу на традиционалне методе тестирања софтвера.
- Потребна је *CI/CD* инфраструктура, тј. одговарајућа софтверска решења за сваки од *CI/CD* елемената, нпр. сервер за интеграцију. Сви проблеми приликом интеграције се могу открити и исправити рано, што убрзава испоруку и смањује трошкове интеграције. Сервер за интеграцију се може поставити локално (нпр. *Jenkins*), или се може користити као *cloud* сервис (нпр. *GitLab CI/CD*).
- Потребно је да програмери усвоје праксу честих *commit*-ова, најмање једном дневно, а по потреби и чешће. Уколико промена коју су извршили утиче на *build* целе апликације, биће одмах обавештени и грешка се лакше може детектовати и исправити.
- Комплексност испоруке нових верзија софтвера је смањена. Организационе и техничке припреме за испоруку се могу аутоматизовати, па је већи фокус на чешћим испорукама и на комуникацији с корисницима. Потребно је посветити одговарајућу пажњу изради документације.

25.2 Континуална интеграција

Настанак појма континуалне интеграције везује се за софтвер *Cruise Control*[575], Јава оквир за аутоматизацију *build* фазе развоја софтвера. Током година, континуална интеграција је постала стандардна пракса у великом броју софтверских предузећа, јер се њеном применом смањују кашњења у испоруци софтвера, смањује се укупан напор потребан за интеграцију, и ствара окружење за развој квалитетног софтвера[576].

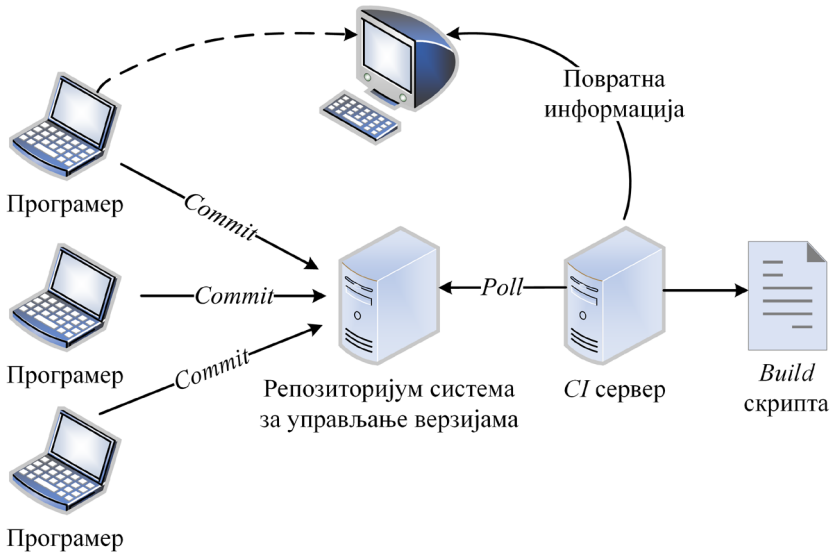
Основне компонтне система за континуалну интеграцију приказане су на Слици 80. Сценарио континуалне интеграције може се описати у неколико корака:

1. Програмер комитује кôд у репозиторијум система за управљање верзијама. Добра пракса је да се претходно повуче актуелна верзија кôда с репозиторијума и уради локални *build*, па тек ако је све у реду, промене сниме на заједнички репозиторијум. Сваки committ у дељени репозиротијум треба да иницира аутоматски *build*.
2. *CI* сервер на сваких неколико минута преузима промене из репозиторијума. Уколико је било промена, интегрише се нови кôд у целокупну апликацију тако што се покреће тзв. *build* скрипта. Интеграција се ради на главној грани система за управљање верзијама. Примери *CI* алата су *Jenkins*, *GitHub Actions*, *Circle CI*.
3. На основу резултата интеграције, *CI* сервер генерише поруку и шаље је члановима тима. Порука се може послати као имејл, као директна порука преко неког

[575] https://cruisecontrol.sourceforge.net/
[576] https://www.martinfowler.com/articles/continuousIntegration.html

канала за комуникацију (нпр. *Slack*) или на други начин. Ако се током *build*-а догоди грешка, добра пракса је да тада исправка грешке буде задатак од највећег приоритета. Препорука је да се грешка исправи у року од највише десет минута, што може значити и враћање на претходну верзију софтвера.

4. *CI* сервер наставља да скенира промене у репозиторијуму система за управљање верзијама.



Слика 80. Компоненте система континуалне интеграције[577]

Предуслов за ефективну континуалну интеграцију је квалитетан и добро конфигурисан систем за управљање верзијама софтвера. Дељени репозиторијум би требало да члановима тима, осим изворног кôда, обезбеди и ресурсе или линкове ка ресурсима као што су тестови, шема базе података, подаци за тестирање, конфигурациони фајлови, параметри за конфигурацију развојног окружења, скрипте за инсталацију, библиотеке и друго.

Приликом тестирања интеграције софтвера неопходно је проверити интеракције и размену података између различитих модула. Типично се реализује након модуларног тестирања (*unit testing*) а пре тестирања целокупног система. Постоји више типова тестирања интеграције[578]:

- ***Big-Bang*** приступ. Заснива се на комбиновању свих модула софтвера и провери функционалности интеграције након тестирања појединачних модула. Типично се користи када су зависности између модула мале. Овај приступ је ефективан само у системима с мањим бројем модула, јер је приликом тестирања тешко одредити који модул проузрокује грешку.

[577] Прилагођено према: Duvall, P. M., Matyas, S. & Glover, A. (2007). *Continuous integration: improving software quality and reducing risk*. Pearson Education.
[578] https://www.geeksforgeeks.org/software-engineering-integration-testing/

- ***Bottom-Up*** приступ. Тестира се интеграција сваког модула на нижем нивоу хијерархије с модулима на вишем нивоу хијерархије. Циљ је тестирати појединачне интерфејсе између различиитх модула. Овај приступ омогућује паралелизацију тестирања појединих подсистема, једноставно креирање тестова и анализу резултата.
- ***Top-Dow***n приступ. Примњује се тако што се прво тестирају модули на вишем нивоу хијерархије, а симулира се понашање модула на нижем нивоу хијерархије који још увек нису интегрисани. Овај приступ омогућује независно дебаговање модула, једноставну изолацију грешака на интерфејсима, и брзо проналажење грешака. Могући су проблеми приликом тестирања модула на нижим нивоима хијерархије.
- ***Хибридни*** приступ. Представаља комбинацију *bottom-up* и *top-down* приступа. Погодан је за велике пројекте с већим бројем подсистема.

Појам континуалне интеграције у контексту модуларних, микросервисних архитектура односи се и на интеграцију хетерогених компоненти у јединствен софтверски систем. У зависности од предмета интеграције, могу се разликовати[579]:

- **Интеграција апликација.** Односи се на спајање различитих апликација независно од платформи или технологија у којима су развијене. Интеграција апликација омогућује хармонизацију и аутоматизацију пословних процеса, и конзистентност размене података. Пример може бити интеграција *CRM* система с различитим сервисима за комуникацију с клијентима, као што су имејл, позивни центар, друштвени медији и др.
- **Интеграција података.** Односи се на хармозацију података из различитих извора, стварајући свеобухватни поглед на информационе ресурсе. Интеграцијом података спајају се и по потреби трансформишу подаци из различитих база података, фајлова, апликација, екстерних извора и др., учитавају у погодне структуре (нпр. у информационо језеро) и припремају у форму погодну за доношење одлука.
- **Интеграција *API*.** Омогућује софтверским компонентама да комуницирају и сарађују.
- ***Cloud* интеграција.** Односи се на проблеме интегрисања софтверских компоненти које су дистрибуиране на *cloud* и на локалној инфраструктури. Велики број организација користи хибридни приступ, па део апликација хостује на локалној инфраструктури а део измешта у *cloud*. Неопходно је да све апликације функционишу као једна целина, без обзира на инфраструктуру на којој су постављене.
- ***IoT* интеграција.** У савремене апликације је често потребно интегрисати податке из реалног света који се прикупљају преко различитих сензора или извршити управљачке акције над *IoT* уређајима.
- ***AI* интеграција.** Односи се на могућност интерације различитих *AI* софтверких компоненти, као што су базе знања, модели машинског учења, софвер за синтетисање говора и сл. с другим софтверским компонентама информационих система.

[579] https://www.codium.ai/developers-hub/integration-of-different-software-components/

25.3 Интеграциона платформа као сервис

Интеграциона платформа као сервис (*Integration Platform as a Service, iPaaS*) је *cloud* сервис који омогућује развој, извршење и управљање пословима интеграције апликација, *API*-ја и података[580]. Користе се као замена за *Enterprise Service Bus* у *cloud* окружењу. Могу се користити за интеграцију софтверских компоненти хостованих на локалној инфраструктури предузећа, у *cloud*-у, или ширем екосистему. С обзиром на то да омогућују аутоматозацију послова интеграције апликација и података, сматрају их једном од кључних технологија за дигиталну трансформацију предузећа.

Интеграционе платформе се типично развијају у форми *low-code* платформи. Корисницима нуде графички интерфејс путем којег се могу креирати шеме за аутоматизацију интеграције апликација или података, без или с мало кодирања. Поседују уграђене конекторе ка различитим апликацијама и базама података, механизме за спецификацију потребних *ETL* процеса, подржавају *API* протоколе попут *REST*, *SOAP*, *GraphQL*, протоколе за пренос података и фајлова попут *http*, *ftp*, *sftp*, различите формате датотека попут *XML*, *JSON*, *Parquet*, и главне *cloud* системе за складиштење фајлова, као што су *Amazon S3*, *Windows Azure Storage Blob* и друга. Примери *iPaaS* решења су *MuleSoft* (*Salesforce*), *Amazon AppFlow*, *Azure Logic Apps*, и друга.

25.4 Патерни постављања софтвера у продукционо окружење

Без обзира на све претходне фазе *DevOps*-а, у којима су анализирани ризици и извршено тестирање нове верзије софтвера, у пракси се често дешава да се при постављању нове верзије појаве грешке, које се у ходу морају исправити или вратити претходна верзија софтвера. Да би се ризик и трошкови смањили, а поступак у највећој мери аутоматизовао[581], предложене су и препоручене стратегије постављања софтвера у продукционо окружење (*deployment strategies*), настале на основу искуства и препорука најбоље праксе. У литератури се ове препоруке често називају и патернима постављања софтвера у продукционо окружење (*deployment patterns*)[582]. Избор стратегије или патерна, који ће се користити, зависи од пројектних захтева, процењеног ризика, понашања софтвера при надоградњи, имплементационог окружења и изабраног скупа алата[583].

---

[580] https://www.snaplogic.com/blog/what-is-ipaas-integration-platform-as-a-service

[581] Harzenetter, L., Breitenbücher, U., Falkenthal, M., Guth, J., Krieger, C. & Leymann, F. (2018, December). Pattern-based deployment models and their automatic execution. *In: 2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)* (pp. 41–52). IEEE.

[582] Eilam, T., Elder, M., Konstantinou, A. V. & Snible, E. (2011, May). Pattern-based composite application deployment. In: *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011)* and Workshops (pp. 217–224). IEEE.

[583] Bayona-Oré, S., Calvo-Manzano, J. A., Cuevas, G. & San-Feliu, T. (2014). Critical success factors taxonomy for software process deployment, *Software Quality Journal*, 22, 21–48.

Најчешће коришћени патерни су[584]:

- ***In-place deployment*** је патерн у коме се врши ажурирање на нову верзију софтвера без замене било које инфраструктурне компоненте. Претходна верзија софтвера се у оквиру постојеће инфраструктуре на сваком рачунарском ресурсу зауставља, инсталира се нова верзија, пушта се у рад и врши се њена валидација. Применом овог патерна нема потребе за додатном инфраструктуром. Лоше стране су да софтвер није расположив током преласка на нову верзију, а ризик и трошкови повратка на претходну верзију (*rollback*) могу бити високи.
- ***Rolling deployments*** је патерн у коме се претходна верзија софтвера, корак по корак, замењује новом верзијом софтвера, при чему се истовремено са новом верзијом замењује инфраструктура на којој софтвер ради. Погодан је и често се примењује када се софтвер испоручује у форми контејнера. Контејнери претходне верзије, један по један, мењају се контејнерима који покрећу нову верзију софтвера. При примени овог патерна не постоји изолација инфраструктурног окружења између претходне и нове верзије. То омогућава брзо постављање софтвера у продукционо окружење, али повећава ризик и усложњава поступак повратка на претходну верзију, уколико поступак не буде успешан.
- ***Blue/green deployment*** је патерн у коме се креирају два истоветна инфраструктурна окружења за рад софтвера. На једном (*blue*) се извршава претходна верзија софтвера. На другом (*green*) се ради и тестира нова верзија. Када се заврши тестирање, корисници и њихов саобраћај се најчешће преко *proxy* сервера аутоматски преусмеравају на green окружење, а *blue* окружење се напушта и гаси. Применом ове стратегије повећава се доступност софтвера и смањују се ризици отказа поједностављењем процеса повратка на претходну верзију софтвера. Недостатак је повећана цена за ангажовање инфраструктурних ресурса за паралелан рад два истоветна продукциона окружења.
- ***Canary release*** је патерн у коме се нека од нових особина софтвера доставља мањој групи, пре него што се достави свим корисницима. Изабрана група корисника се о томе обавештава и доставља им се упутство за тестирање, Уколико се изабрани корисници о измени не обавесте, варијанта овог патерна се назива *Dark Launches.* Примена овог патерна омогућава развојном тиму да прати перформансе и изврши све неопходне измене пре него што софтвер постане доступан свим корисницима. Овај приступ, побољшава корисничко искуство, смањује време недоступности софтвера, као и ризик.

Ово су најчешће коришћени патерни. У пракси се користи већи број патерна који су прилагођени специфичним пројектним захтевима и имплементационим окружењима[585]. Провајдери рачунарства у облаку нуде алате за аутоматизацију процеса постављања софтвера у продукционо окружење за већину наведених патерна[586].

---

[584] https://levelup.gitconnected.com/deployment-patterns-the-most-common-explained-25d89c3ecfab

[585] Humble, J., Read, C. & North, D. (2006, July). The deployment production line. In: *AGILE 2006 (AGILE'06)*, IEEE.

[586] https://docs.aws.amazon.com/whitepapers/latest/overview-deployment-options/welcome.html

V ИНФРАСТРУКТУРА

IT инфраструктура представља окосницу имплементације софтвера оријентисаног на процесе. Састоји се из сложеног склопа хардвера, софтверских решења и мрежних ресурса. Инфраструктура је стратешка компонента која се у пуном капацитету користи тек на крају софтверског процеса, у експлоатацији софтвера, али од почетка пројектовања утиче на софтверска решења и обликује поступке којима ће се подаци обрађивати, начин како ће се чувати и комуницирати.

Сервери, складиштење података и мрежна опрема основни су елементи инфраструктуре. Њихова синхронизација и међусобна повезаност су кључни за постизање оптималних перформанси, и као такви подлежу континуалним процесима оптимизације и модернизације.

Дата центар (*data center*) је физичка локација или виртуелна инфраструктура која пружа централизовано управљање и смештај *IT* ресурса, укључујући сервере, мрежне компоненте и складишне капацитете, са циљем ефикасне обраде, дистрибуције и чувања података[587]. Елементи за складиштење, обраду и дистрибуирање података играју виталну улогу у одржавању флуидности и отпорности на отказе целокупног информационог система, брзом алокацијом и ефикасним управљањем *IT* ресурсима, креирајући на тај начин синхронизовани екосистем[588].

Виртуелизација је једна од најважнијих технологија у овом контексту. Трансформацијом физичких ресурса у виртуелне инстанце омогућава се боље искоришћење хардвера и смањује потреба за физичким простором. Тиме се смањују оперативни трошкови, повећава се енергетска ефикасност и оперативна флексибилност[589].

---

[587] Kaiya, H. (2018). Meta-requirements for information system requirements: lesson learned from software ecosystem researches, Procedia Computer Science, 126, 1243–1252.

[588] Sulistio, A., Reich, C. & Doelitzscher, F. (2009). Cloud infrastructure & applications–CloudIA. In: *Cloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1–4, 2009. Proceedings 1* (pp. 583–588). Springer Berlin Heidelberg.

[589] Alsbatin, L., Öz, G. & Ulusoy, A. H. (2017, September). An overview of energy-efficient cloud data centres. In: *2017 International Conference on Computer and Applications (ICCA)* (pp. 211–214). IEEE.

Софтверски дефинисане мреже (*SDN*) омогућавају централизовану контролу и динамичко прилагођавање мрежног протока, чиме се ствара платформа за ефикаснију комуникацију између различитих ресурса и апликација. Овакав приступ управљању мрежном инфраструктуром омогућава дата центрима да постигну високе нивое скалабилности и доступности. Управо овај напредак у мрежној оптимизацији чини софтверски дефинисане мреже не само корисним, већ и апсолутно неопходним за очување високих стандарда перформанси и ефикасности у овим комплексним екосистемима.

Савремена решења за складиштење података, посебно софтверски дефинисано складиштење (*SDS*), од виталне су важности за ефикасно управљање све већом количином података која се генерише у дата центрима. Таква складишна решења омогућавају адаптивну алокацију ресурса и капацитета, стварајући потребне предуслове за оптимизацију и скалирање целокупног система.

Контејнеризација, с фокусом на изолацији и управљању апликацијама, постаје саставни део архитектурних решења, не само у оквиру дата центара, већ и у процесима пројектовања софтвера. Поред убрзаног развоја и лакше имплементације апликација, контејнеризација омогућава боље искоришћење ресурса и стварање ефикаснијих радних процеса, оркестрацијом и аутоматизацијом распореда контејнера.

Рачунарство у облаку (*cloud computing*) постаје инструмент за брзо и ефикасно скалирање апликација и сервиса. Трансформацијом *IT* инфраструктуре ка рачунарству у облаку омогућена је демократизација приступа напредним технолошким ресурсима, који су сада доступни и малим предузећима. Улога рачунарства у облаку протеже се даље од једноставног пружања ресурса; постаје есенцијални део стратешког планирања организације која тежи оптимизацији и иновацији. Рачунарство у облаку је тиме постало основа глобалне *IT* инфраструктуре[590].

Савремена *IT* инфраструктура активно подржава дигиталну трансформацију омогућавајући организацијама да се брзо прилагоде променљивим тржишним условима и технолошким трендовима. У оваквим технолошки динамичним окружењима, *DevOps* методе управљања софтверским процесом и аутоматизација његових појединих фаза, постају све значајније. *DevOps* култура, уз подршку напредних комуникационих алата и сервиса, омогућава бржу и ефикаснију комуникацију између развојних и оперативних тимова, док аутоматизација убрзава фазе животног циклуса софтвара.

[590] Li, D., Huang, H. & Wang, K. (2023). Information technology infrastructure and earnings management strategy: evidence from a quasi-natural experiment, *China Journal of Accounting Studies,* 11(1), 108–133.

26. ВИРТУЕЛИЗАЦИЈА

Виртуелизација је врста апстракције у којој се ресурси рачунарског система апстрахују софтверским компонентама, које корисницима у вишекорисничком раду обезбеђују илузију њиховог ексклузивног коришћења. Посматрано с практичне стране, виртуелизација представља акт креације виртуелних верзија рачунарског хардвера, складишних уређаја и мрежних ресурса у циљу бољег искоришћења, смањења трошкова електричне енергије, смањења потребног простора, повећања опсега функционалности и побољшања сигурности апликација које користе виртуелне рачунарске ресурсе[591].

Поред рачунарског хардвера, могу се виртуелизовати процеси (контејнери), складишни ресурси (*SAN*, *NAS*, глобални фајл системи), базе података (дистрибуиране *NoSQL* базе података, информациона језера) и мрежни ресурси (софтверски дефинисане мреже)[592].

Централна компонента виртуелизације је хипервизор (*Hypervisor*, *Virtual Machine Manager-VMM*), софтвер који управља виртуелизацијом ресурса и њиховим коришћењем. Сервер на коме ради хипервизор назива се *host*. Корисник виртуелизованих ресурса, виртуелна машина (*virtual machine – VM*) назива се *guest*. Виртуелне машине представљају међусобно изолована окружења која у раду користе виртуелизоване ресурсе *host*-а. Један хипервизор може виртуелним ресурсима паралелно опслуживати више guest виртуелних машина.

Посебан допринос развоју хипервизора дао је пројекат *libvirt*[593] који је за већину популарних хипервизора дефинисао стандардизован интерфејс и имплементирао већину менаџмент функција.

26.1 Еволуција и историја развоја виртуелизације

Виртуелизација хардверских ресурса је од почетка рачунарства била уграђивана у структуру кернела вишекорисничких оперативних система. Кернел оперативног система корисницима омогућава илузију ексклузивног коришћења рачунарских ресурса.

За виртуелизацију процесора задужен је део кернела за управљање процесима који сваком процесу корисника додељује одређен сегмент времена рада процесора и на тај начин ствара илузију да сваки процес има свој процесор. Доделу меморијских ресурса програмима корисника обавља део кернела који се назива алокатор меморијског простора који води рачуна о виртуелној меморији и њеној расподели

[591] Huang, D. & Wu, H. (2018). Chapter 2-virtualization, *Mobile Cloud Computing*. Morgan Kaufmann, 31–64.
[592] Radenković, B. & Kočović, P. (2014). From mainframe to cloud. In: *Handbook of Research on High Performance and Cloud Computing in Scientific Research and Education* (pp. 1–30). IGI Global.
[593] https://libvirt.org/

на процесе корисника. Приступ хард дисковима контролише виртуелни диск, који сваком процесу може обезбедити илузију да има ексклузивни приступ хард диску.

Историја развоја оперативних система представља историју виртуелизације. Већ у раним данима рачунарства, седамдесетих година двадесетог века, на рачунару *IBM*370 постојала је могућност пуштања више независних оперативних система у истом хардверском окружењу. У то време, седамдесетих и осамдесетих година, уместо паралелних погледа на хардвер (контејнер) развијају се монолитни оперативни системи који централизовано управљају хардвером. Виртуелне машине у овој фази развоја рачунарства емулирају комплетан физички рачунар, укључујући и оперативни систем.

Употреба виртуелних машина донела је неколико изазова. Свака виртуелна машина покреће сопствену копију оперативног система, па је за сваку од њих потребна значајна количина ресурса. Осим тога, за управљање виртуелним машинама користе се посебни управљачки алати, хипервизори, што повећава сложеност управљања инфраструктуром.

Као одговор на ове изазове, кренуло се на истраживања технологија контејнеризације које би користиле исти кернел оперативног система за покретање више изолованих апликација и тиме смањиле потрошњу ресурса у поређењу с виртуелним машинама.

Већ 1979. у седму верзију оперативног система *UNIX*, а нешто касније и у *BSD* 4.2, уведен је *chroot*[594] као концепт који омогућава изолацију фајл система за различите процесе. Унапређење овог концепта и значајан корак ка контејнеризацији начињен је деведесетих година, у пројекту *Plan*9[595], који је у кернел оперативног система *UNIX* увео независне адресне шеме за означавање ресурса у различитим апликацијама. То је била основа да се у *Linux* оперативни систем почетком двадесетпрвог века уведе *chroot* и учине први кораци ка изолацији процеса. Најзначајније иновације у структури кернела, које су омогућиле савремене облике виртуелизације и контејнере, јесу: контролне групе *cgroups*, као концепт за контролу и ограничавање коришћења ресурса хијерархијски повезаних група процеса, и *namespaces* за области именовања и изолацију ресурса које те групе процеса користе.

Термин контејнер је 2005. први пут употребио *SUN Microsystems*[596], као синоним за Соларис зоне[597], један од начина *chroot* имплементације. Значајан допринос развоју контејнерских технологија дали су *LXC* контејнери[598] који користе

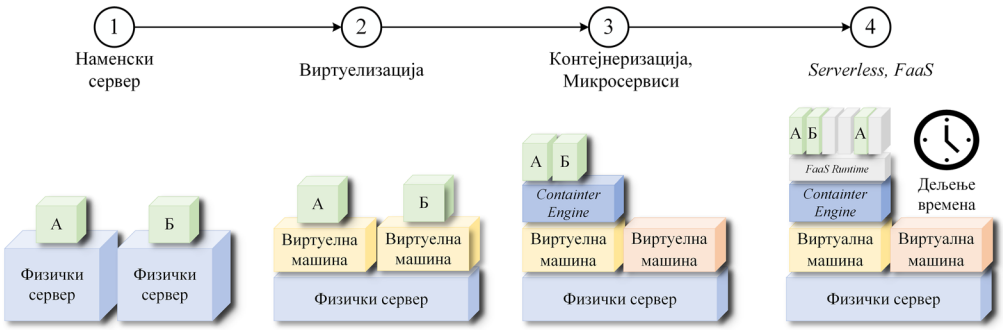
[594] https://bsdimp.blogspot.com/2020/06/whither-chroot.html
[595] https://9p.io/plan9/
[596] Galvin, P. (2005). Solaris 10 containers, *USENIX login*, 11–14.https://www.usenix.org/system/files/login/issues/login_october_2005.pdf#page=13
[597] Price, D. & Tucker, A. (2004, November). Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In: *LISA* (Vol. 4, pp. 241–254). https://www.usenix.org/legacy/publications/library/proceedings/lisa04/tech/full_papers/price/price.pdf
[598] https://linuxcontainers.org/lxc/introduction/

искључиво ресурсе *Linux* кернела. Унапређена верзија, *LXD* контејнери[599] користе се у најновијим верзијама *Ubuntu Linux* оперативних система и у *Kubernetes*-у.

Широко коришћење контејнера почело је објављивањем *Docker*-а 2013. године. *Docker*[600] је поједноставио управљање контејнерима. У извршном окружењу за рад контејнера користи *namespaces* и *cgroups* функције *Linux* кернела како би омогућио контролу ресурса и висок ниво изолације између процеса. Поред тога, садржи и практичан командни интерфејс и *API* за покретање контејнера, као и услугу регистра (*Docker Hub*[601]) за дељење слика контејнера између корисника.

Усвајање микросервисне архитектуре додатно је подстакло прихватање контејнера. Контејнери пружају идеално окружење за постављање, скалирање и управљање микросервисима. Управо у контексту примене микросервиса, оркестрација контејнера постаје неопходна. Оркестрација контејнера односи се на аутоматско распоређивање, управљање и координацију контејнера у контејнерским апликацијама. Ово је нарочито важно за апликације које користе микросервисну архитектуру, јер она омогућава ефикасно управљање и скалирање великог броја контејнера који често комуницирају. *Kubernetes*[602], отворена платформа за оркестрацију контејнера, уведен је како би се аутоматизовала имплементација, скалирање и управљање контејнеризованим апликацијама. Оркестрација контејнера, предвођена *Kubernetes*-ом, постала је кључна за успешно постављање, управљање и скалирање контејнерских апликација, посебно оних које користе микросервисну архитектуру, чиме је допринела већој ефикасности, скалабилности и поузданости у развоју и операцијама софтвера.

Еволуција платформи за виртуелизацију и контејнере приказана је на Слици 81.



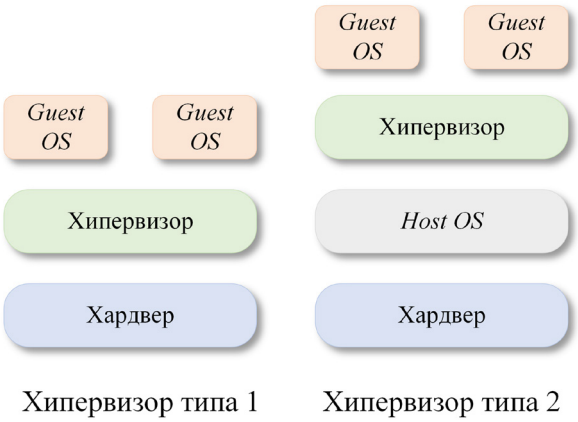
Слика 81. Еволуција платформи за виртуелизацију и контејнере[603]

[599] https://ubuntu.com/lxd
[600] https://www.docker.com/
[601] https://hub.docker.com/
[602] https://kubernetes.io/
[603] Прилагођено према: Kratzke, N. (2018). A brief history of cloud application architectures. *Applied Sciences*, 8(8), 1368. https://www.mdpi.com/2076-3417/8/8/1368

Технологија контејнера се развила од скромних почетака, до тога да је постала темељ модерних софтверских архитектура. Контејнери, комбиновани с напретком у функцијама *Linux* кернела, трансформисали су начин на који се апликације пакују, дистрибуирају и извршавају. Успон *Docker*-а и *Kubernetes*-а додатно је учврстио улогу контејнера у постизању ефикасних, скалабилних и преносивих софтверских решења, посебно у ери микросервиса и рачунарства у облаку. Контејнери и микросервиси представљају основу на којој се заснива *serverless computing*[604] и функција као сервис (*function as a service – FaaS*) рачунарства у облаку.

26.2 Типови виртуелизације

Уколико хипервизор директно приступа хардверским ресурсима, без посредовања оперативног система на *host* рачунару, тада се класификује као хипервизор типа I (*bare-metal*, или *Type-I virtualization*). Уколико хипервизор приступа хардверским ресурсима посредовањем кернела оперативног система и ради као посебан програм у његовом окружењу, назива се хипервизор типа II (*Type-II virtualization*). Слика 82 приказује разлике између хипервизора типа I и типа II.



Слика 82. Типови хипервизора

С обзиром на то да новије верзије *Linux* кернела подржавају пропагацију (*passthrough*) хардверских ресурса[605], од хардвера, преко кернела, хипервизора, па све до виртуелне машине, подела типова хипервизора није тако стриктна. Хипервизори типа II могу да имају и особине хипервизора типа I. На основу начина функционисања хипервизора, виртуелизацију можемо даље грубо поделити у следеће групе: потпуна виртуелизација, виртуелизација подржана кернелом, виртуелизација подржана хардвером и паравиртуелизација.

[604] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., ... & Suter, P. (2017). Serverless computing: Current trends and open problems, *Research advances in cloud computing*, 1–20. http://arxiv.org/abs/1706.03178.pdf

[605] https://developer.ibm.com/tutorials/l-pci-passthrough/

26.2.1 Потпуна виртуелизација

Потпуна виртуелизација (*full virtualization*) обезбеђује ресурсе и подршку за немодификоване виртуелне машине чија се архитектура и процесор могу разликовати од архитектуре и процесора хипервизора. У том сценарију, хипервизор омогућава виртуелним машинама емулацију потпуног окружења и интерпретирање наредби процесора виртуелне машине. То се реализује по цену успорења рада виртуелне машине, које може бити и десетинама пута спорије од природног хардверског окружења. Због тога се овај вид виртуелизације ређе користи и то у развоју и тестирању софтвера, нпр. за развој и тестирање андроид апликација (*Android Studio*[606], *BlueStacks*[607] и други), или када природно окружење није доступно, када је застарело, или га је тешко набавити, а постоји потреба за коришћењем апликација које раде у том окружењу, нпр. *DOS-Box*[608], емулатор *MS DOS* оперативног система. Највише коришћен систем за потпуну емулацију архитектура заснованих на различитим процесорима је *QEMU* (*Quick Emulator*)[609].

26.2.2 Виртуелизација подржана кернелом

Виртуелизација подржана кернелом (*Kernel-level Virtualization*, у жаргону контејнера, често називана и *Lightweight Virtualization*), ослања се на концепте кернела оперативног система намењене изолацији и управљању процесима. У овом типу виртуелизације, у реалности постоји само кернел *host* рачунара. Виртуелизација ресурса кернела обавља се на нивоу процеса. Виртуелизованом процесу се додељују реални ресурси *host*-а, који су изоловани од других процеса. Изолација процеса и сложено управљање концептима кернела, који су прилично ниског нивоа (*cgroups* и *namespaces*), представља највећи изазов у примени овог вида виртуелизације. С друге стране, предност је да између кернела и виртуелизованих процеса који раде у изолованим окружењима не мора бити посредника, па нема ни непотребног трошења ресурса за виртуелизацију. На овај вид виртуелизације ослања се *Kernel-based Virtual Machine – KVM*[610], често коришћен хипервизор у *Linux* окружењу. Контејнери такође користе овај тип виртуелизације. Раст популарности контејнерских технологија, посебно *Docker*-а и *Kubrenetes*-а, довео је до тога да се у пракси овај тип виртуелизације најчешће користи.

[606] https://developer.android.com/studio
[607] https://www.bluestacks.com/
[608] https://www.dosbox.com/
[609] https://www.qemu.org/
[610] https://www.linux-kvm.org/

26.2.3 Виртуелизација подржана хардвером

Виртуелизација подржана хардвером (*Hardware-assisted Virtualisation*) користи специјализоване архитектурне екстензије и посебне скупове наредби процесора, као што су *Intel VT-x* и *AMD-V*, које подржавају виртуелизацију и изолацију ресурса. Ове архитектурне екстензије процесора омогућавају хипервизору и његовом кернелу да за послове виртуелизације, уместо својих софтверских компоненти, користе ресурсе процесора, делегирајући најважније послове емулације, изолације и управљања виртуелним машинама, на ниво хардвера. Виртуелизација подржана хардвером односи се на унапређење функционисања хипервизора, чиме се повећавају перформансе и скалабилност, а у исто време не утиче се на начин функционисања или структуру виртуелне машине. Будући да не захтева модификације оперативног система виртуелне машине, овај тип виртуелизације се широко примењује и компатибилан је с различитим окружењима за виртуелизацију (*VMware*, *Xen*, *KVM*, *Microsoft Hyper-V* и други). Међутим, овај тип виртуелизације је условљен применом компатибилних процесора, што може ограничити његову примену на старијим серверима[611].

26.2.4 Паравиртуелизација

Паравиртуелизација (*Paravirtualization*) је приступ виртуелизацији у којем хипервизор, поред директне емулације конкретних хардверских компоненти које се могу наћи на тржишту, као што су диск контролери, графичке картице, мрежне картице и друге, нуди сопствене виртуелизоване верзије контролера, које заснивају рад на коришћењу ресурса кернела, који директно приступа хардверским ресурсима *host*-а. Виртуелним машинама се на тај начин, уз посредовање кернела, нуде виртуелни интерфејси који имају највише до 10% успорења у односу на перформансе хардвера, што је веома добар резултат у односу на емулацију. Међутим, драјвери за такве интерфејсе не постоје, тако да је потребно уградити их накнадно у виртуелне машине. Као подршка ширем коришћењу овог приступа, дефинисан је *virtio*, отворени стандард за дефинисање паравиртуелизованих интерфејса. Његовим уграђивањем у *Linux* кернел[612], широм су отворена врата за паравиртуелизацију, тако да данас скоро сви доступни хипервизори подржавају паравиртуелизацију коришћењем *virtio* интерфејса.

26.3 Виртуелизација рачунарских мрежа

Виртуелизација рачунарских мрежа омогућава стварање виртуелних, логички изолованих мрежних ресурса на физичкој мрежној инфраструктури. Ова технологија је променила начин на који организације планирају, конструишу и

[611] Fisher-Ogden, J. (2006). Hardware support for efficient virtualization. *University of California, San Diego, Tech. Rep*, 12, 1–12. https://cseweb.ucsd.edu/~jfisherogden/hardwareVirt.pdf
[612] https://docs.kernel.org/driver-api/virtio/virtio.html

управљају својим рачунарским мрежама, омогућавајући већу флексибилност и оптимизацију ресурса. Коришћењем мрежне виртуелизације, корисници и апликације могу функционисати као да су на одвојеним мрежама, иако физички деле исту инфраструктуру.

Виртуелизација рачунарских мрежа омогућава бољу искоришћеност мрежних хардверских ресурса, смањење трошкова и омогућава брже развојне циклусе мрежне инфраструктуре. Виртуелне мреже се могу брзо постављати, модификовати или уклањати без потребе за физичким реконфигурисањем мрежних уређаја. Омогућавају бољу адаптацију на различите захтеве корисника и апликација, што је корисно у динамичним окружењима где се услови често мењају[613].

Софтверски дефинисане мреже (*Software Defined Networking – SDN*) су технологије које се ослањају на виртуелизацију рачунарских мрежа, али иду корак даље у централизованом управљању ресурсима. Софтверски дефинисане мреже омогућавају прецизну контролу мрежних ресурса и динамичко адаптирање на различите оперативне услове, што је кључно за примену у рачунарству у облаку[614].

Основне компоненте инфраструктуре *SDN* су: апликације, контролери, уређаји и протоколи. Апликације у *SDN* инфраструктури шаљу мрежне захтеве ка контролеру преко програмабилних интерфејса, тј. *API*-ја. То могу бити захтеви који се тичу управљања *IP* адресама, управљања квалитетом сервиса, аналитике, безбедности, расподеле оптерећења итд. *SDN* контролер је компонента задужена да апликацијама пружи апстрактан поглед на мрежне ресурсе. *SDN* уређаји су мрежни уређаји, који се од традиционалних разликују по софтверским компонентама које омогућују имплементацију кориснички дефинисаних мрежних функција. Мрежни протоколи и *API*-ји у *SDN* су компоненте које омогућују програмирање управљања саобраћајем у мрежи. *API* интеграције се често категоризују на тзв. *northbound* и *southbound* интерфејсе, где су први намењени за интеграцију апликација и контролера, а други за интеграцију контролера и уређаја. *Northbound API*-ји се могу имплементирати коришћењем различитих протокола, укључујући *REST*. Пример *southbound* протокола је *OpenFlow*[615].

У рачунарству у облаку, *SDN* игра кључну улогу у оптимизацији и аутоматизацији мрежних функционалности. С обзиром на комплексност инфраструктуре рачунарства у облаку, *SDN* омогућава брзо и ефикасно скалирање, аутоматско распоређивање оптерећења и интегрисане безбедносне протоколе. Ова флексибилност је кључна за подршку великом броју апликација и услуга које платформе рачунарства у облаку могу понудити.

---

[613] Chowdhury, N. M. K. & Boutaba, R. (2009). Network virtualization: state of the art and research challenges. *IEEE Communications magazine*, 47(7), 20–26. http://www.csc.villanova.edu/~nadi/csc8580/S11/DeeptiNune.pdf

[614] https://www.cisco.com/c/en/us/solutions/software-defined-networking/overview.html

[615] Rischke, J. & Salah, H. (2020). Software-defined networks. In: *Computing in Communication Networks* (pp. 107–118). Academic Press.

Коришћењем *SDN*, организације могу значајно смањити оперативне трошкове и повећати ефикасност. Централизовано управљање мрежним ресурсима омогућава администраторима да имају преглед и контролу над свим мрежним ресурсима, што је кључно за ефикасно распоређивање ресурса и обезбеђивање високе доступности и перформанси рачунарске мреже.

Карактеристичан пример за софтверски дефинисане мреже је *Open Stack Neutron*[616] софтвер отвореног кôда.

26.4 Виртуелизација складиштења података

Виртуелизација складиштења података представља технолошки процес који апстрахује физичке складишне ресурсе и групише их у један или више виртуелних складишних простора. Основни циљ виртуелизације складишта података је да се обезбеди независност локације података раздвајањем физичког и логичког простора на коме се подаци чувају. Кључне технолошке компоненте у овом процесу укључују *SAN* (*Storage Area Network*), *NAS* (*Network Attached Storage*) и различите врсте *RAID* конфигурација. Ове технологије омогућавају изолацију и сегментацију складишног простора[617].

SNIA (*Storage Networking Industry Association*) дала је системску класификацију виртуелизације складишта података, по више критеријума[618]:

- по томе шта се виртуелизује: блокови података, дискови, магнетне траке, фајл системи, фајлови;
- по томе где се виртуелизација ради: на серверу, на мрежи, на уређају за складиштење података;
- по начину на који се виртуелизација ради: *in-band*, *out-of-band*.

Основне предности виртуелизације складишта података су повећана скалабилност, ефикасност и флексибилност у управљању подацима. С виртуелном архитектуром, администратори система могу лакше преносити, алоцирати и реалоцирати складишне ресурсе без потребе за мануелним интервенцијама и прекидима у раду[619].

Као карактеристични примери за виртуелизацију складиштења података, могу се навести *OpenStack*, који користи *Cinder*[620] модул за управљање виртуелним

[616] https://wiki.openstack.org/wiki/Neutron
[617] Morris, R. J. & Truskowski, B. J. (2003). The evolution of storage systems. *IBM systems Journal*, 42(2), 205–217. https://cs.unibg.it/arrigoni/impianti/articoli/Morris_2003.pdf
[618] Bunn, F., Simpson, N., Peglar, R. & Nagle, G. (2003). Storage Virtualization, *SNIA Technical Tutorial*, https://www.snia.org/sites/default/files/sniavirt.pdf
[619] Gupta, P. K. & Asia, C. S. S. (2008). Storage Virtualization: What, Why, Where and How. *The Storage Networking Industry Association (SNIA)*. https://mediabuzz.com.sg/mediabuzz_supplements/the_virtual_world/Virtualization_I.pdf
[620] https://docs.openstack.org/cinder/latest/

складиштима блокова података и *Swift*[621] модул за управљање виртуелним објектним складиштима, као и *Amazon S3*[622] *storage*. Подршка за виртуелизацију фајл система уграђена је у *Linux* оперативни систем[623].

26.5 Избор хипервизора и типа виртуелизације

Анализом наведених типова виртуелизације, јасно је да ниједан приступ није универзалан и оптималан за све сценарије. Одабир типа виртуелизације и хипервизора захтева пажљиву анализу организационих потреба, техничких капацитета и дугорочних стратешких циљева. Сваки тип има своје предности и мане које се морају анализирати у контексту специфичних захтева и ресурса који су на располагању[624].

Избор између различитих типова виртуелизације зависи од специфичних захтева и капацитета организације у погледу хардверских, софтверских и администраторских ресурса. Виртуелизација подржана кернелом пружа висок ниво изолације и контролу виртуелних ресурса, али захтева додатни напор у погледу управљања и одржавања. Паравиртуелизација је оптимизована за перформансе, али може захтевати додатна прилагођавања *guest* оперативног система и инсталацију паравиртуелизованих драјвера за уређаје. Виртуелизација подржана хардвером нуди добар баланс између перформанси и компатибилности, али је његова примена условљена хардверским карактеристикама процесора на доступним серверима[625].

Посебно је важно правилно одабрати методе виртуелизације складиштења података и виртуелизације рачунарских мрежа и ускладити их с потребама организације.

27. КОНТЕЈНЕРИ

Контејнер је средство виртуелизације које омогућава паковање, транспорт и извршење независних паралелних апликација на ресурсима вишекорисничких оперативних система. Контејнер има више функција: виртуелизација ресурса, паралелно извршавање више апликација у раздвојеним областима именовања (*namespaces*), стандардизација управљања процесима, управљање меморијом и управљање уређајима у оквиру апликације.

---

[621] https://docs.openstack.org/swift/latest/
[622] https://aws.amazon.com/s3/
[623] https://docs.kernel.org/filesystems/vfs.html
[624] Agesen, O. (2007). Performance Aspects of x86 Virtualization. http://vglab.cse.iitd.ac.in/~sbansal/csl862-virt/2010/readings/PS_TA68_288534_166-1_FIN_v5.pdf
[625] Liu, S. & Jia, W. (2015). A survey: main virtualization methods and key virtualization technologies of CPU and memory. *The Open Cybernetics & Systemics Journal*, 9(1). https://benthamopen.com/contents/pdf/TOCSJ/TOCSJ-9-350.pdf

Контејнери решавају следеће проблеме у развоју апликација:

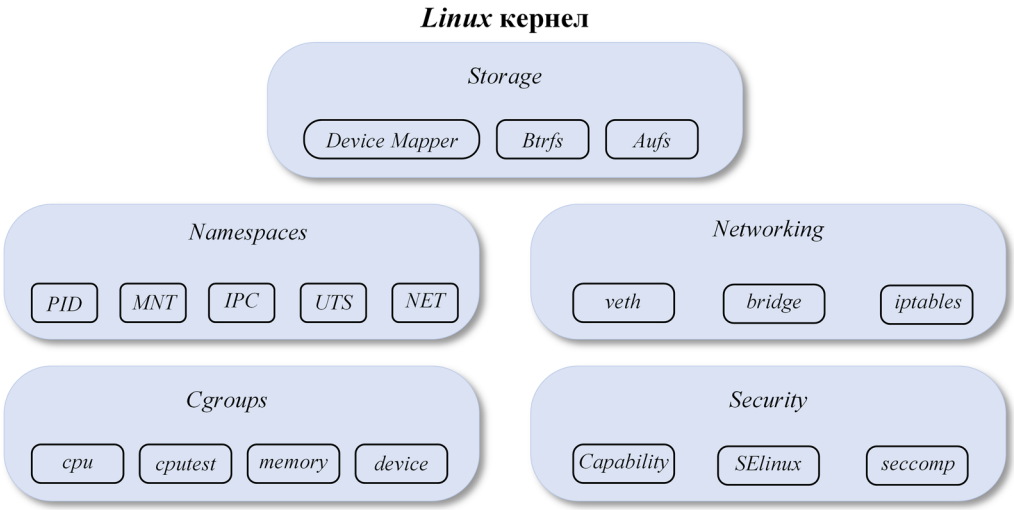
- компатибилност сервиса с верзијама оперативног система и библиотекама,
- употребу различитих окружења за развој, тестирање и продукцију,
- дуго време за постављање развојног, тестног и продукционог окружења,
- различите преференције програмера по питању оперативног система.

Контејнери као концепт имају дугу историју, али су тек недавно стекли широку популарност и прихваћеност због своје ефикасности, преносивости и једноставности имплементације.

27.1 Концепти Linux кернела у контејнерској виртуелизацији

Контејнери представљају изоловану апстракцију оперативног система прилагођену апликацији која се у контејнеру извршава. Контејнери су по својој суштини само *Linux* процеси. Сходно томе, управљање процесима у оквиру контејнера врши се на исти начин и помоћу истих инструмената кернела оперативног система као и за све друге апликације.

Компоненте *Linux* кернела, које се користе у савременим технологијама виртуелизације и контејнерима, приказане су на Слици 83.



Слика 83. Компоненте Linux кернела које се користе у виртуелизацији и контејнерским технологијама[626]

[626] Прилагођено према: https://q15928.github.io/2021/01/09/container-101/

27.1.1 Linux процеси

У оквиру *Linux* оперативног система, процеси представљају инстанце покренутих програма које се извршавају под надзором кернела. Кернел управља процесима, распоређује их и додељује им ресурсе на основу различитих атрибута и приоритета. Креирање, планирање и управљање процесима је сложен задатак који укључује додељивање јединствених идентификатора процеса, праћење информација о власницима процеса и постављање ограничења за коришћење ресурса. Управљање процесима омогућава да се више паралелних процеса извршава на оптималан начин, уз максимално искоришћавање и дељење ресурса.

Контејнере кернел посматра као одвојене процесе, за које обезбеђује међусобну изолацију и оптимизацију перформанси. Процеси се, у зависности од организације обраде, могу извршавати паралелно или секвенцијално. Паралелан рад процеса подразумева међусобно дељење рачунарских ресурса и евентуалну међупроцесну комуникацију, о чему брине кернел оперативног система.

Процеси могу бити привилеговани и непривилеговани. Привилеговани процеси припадају *root* кориснику (*UID* 0), који има неограничено право приступа ресурсима. Непривилеговани процеси припадају осталим корисницима.

27.1.2 Креирање процеса

При креирању, процесу се додељују атрибути: идентификатор процеса, *Process ID* (*PID*), идентификатор корисника *User ID* (*UID*), затим се дефинишу приоритет и друга ограничења ресурса. Атрибути процеса се налазе у структури која се зове *task struct*.

Најчешћи начин креирања хијерархије *Linux* процеса је гранање преко *fork()* системске функције, која се позива у оквиру неког постојећег процеса. Овом функцијом надређени процес креира ново „дете“ процес, који је егзактна копија родитељског процеса и који се извршава паралелно с родитељем. Гранањем процеса наслеђују се атрибути родитељског процеса, његове променљиве и дозволе. Једини атрибут који ово „дете“ процес има, различито у односу на процес који га је генерисао, јесте другачији идентификатор процеса. Како би се постигло да „дете“ процес заправо има сопствени контекст и програм који извршава, мора се позвати једна од *exec* функција (нпр. *execl()*, *execp()*, *execle()*, итд.). Циљ ових функција је да учитају и покрену одређени део меморијског простора на коме се налази жељени програм.

Алтернатива копирању је клонирање, које се врши *clone()* функцијом. Ова функција омогућава већу флексибилност и контролу дељења ресурса између родитељског и клонираног процеса. При позиву *clone()* функције, декларишу се ресурси који ће бити „преузети“ од стране новог процеса, као што су меморија, *file system*, *networks namespace* итд. Користи се за дефинисање изолованих нити и инстанци контејнера.

27.1.3 Организација процеса

Унутар *Linux* оперативног система, организација процеса је хијерархијска. Процес „родитељ“ може имати нула, једно или више „деце“ процеса. Сваки од процеса унутар ове хијерархије има свој *PID* којим се јединствено идентификује као и *PID* свога родитеља (*PPID*). Процес који ће бити први покренут и постарати се да остали системски процеси буду покренути, зове се *init process*, и има *PID* 1. Специфичност *init* процеса је да, уколико дође до терминирања неког процеса који има децу, сва његова деца биће „превезана“ на *init* процес који представља њиховог универзалног претка. Таква деца се називају „сирочићи“ или *orphan process*. У случају да процес терминира, али се и даље систему пријављује као да је „жив“, такав се процес назива *zombie* процес.

Упоредо са хијерархијском организацијом, процеси се могу организовати и у другачије логички повезане целине. Један од начина организовања су процесне групе. Процесне групе се користе у циљу груписања сродних процеса који се баве извршењем заједничког задатка. Свакој групи се додељује идентификатор групе: *process group ID* (*PGID*). Процеси који припадају истој групи, као један од својих атрибута чувају *PGID* групе. Сесије представљају један од начина организовања процесних група. Сесије су групе које имају сопствени *Session ID* (*SID*). *SID* је обично исти као и *ID* процеса који је започео сесију.

Linux процеси, односно инстанце покренутих програма, основни су градивни блокови који омогућавају да контејнери функционишу. Гранање, клонирање и остале технике за креирање и управљање процесима представљају основу за извршавање апликација у контејнерима, док хијерархијска организација процеса олакшава управљање и комуникацију између њих. Кроз груписање процеса и сесија, може се лакше управљати и контролисати извршавање контејнера и ресурса које они користе.

27.1.4 Међупроцесна комуникација

Међупроцесна комуникација (*Inter Process Communication – IPC*) је неопходна уколико више паралелних процеса ради у оквиру једног посла или апликације. Да би се избегли проблеми конкурентног програмирања, кернел на контролисан начин уводи своје интерне ресурсе за подршку комуникацији и синхронизацији процеса, којима процеси могу приступити преко одговарајућих апликационих програмских интерфејса кернела. Најчешћи облик међупроцесне комуникације је пренос података између процеса. За ту сврху кернел процесима нуди специјалну датотеку која може бити доступна свим процесима, а која се назива *Unix Socket*. Један програм може да пише у ту датотеку, док други може да чита уписане податке. Кернел води рачуна о проблемима истовременог приступа и сигнализацији читања из празне датотеке и писања у пуну датотеку. При томе је комуникација једноставна, јер се користи постојећи фајл систем, и веома је брза јер

се као преносни медијум користи *RAM* меморија. За дистрибуирану комуникацију процеса, путем рачунарских мрежа, користи се *TCP Socket*, исте функционалности као и *Unix Socket*, али с рачунарском мрежом као преносним медијумом. Процеси могу комуницирати и преко заједничке, дељене меморије. С обзиром на то да је меморијски простор процеса строго ограничен и изолован, да би се ова форма комуникације ипак омогућила, кернел одваја један део свог меморијског простора који се назива дељена меморија (*shared memory*), којој процеси могу да приступе посредовањем кернела. За избегавање проблема конкурентног програмирања приликом приступа дељеној меморији, као и за сигнализацију одређених стања између процеса, користе се семафори (*semaphores*) којима процеси такође могу приступити посредовањем кернела.

27.1.5 Контрола и ограничавање ресурса – контролне групе (cgroups)

Контролне групе, скраћено *cgroups*, јесу функција кернела која омогућава алокацију и ограничавање ресурса као што су *CPU*, меморија, диск и мрежни пропусни опсег. Контролне групе омогућавају поделу и ограничавање ресурса између различитих процеса или група процеса, укључујући контејнере, тако да ниједан процес или контејнер не може потрошити све ресурсе и тиме нарушити перформансе других.

Основна функција контролних група је контрола ресурса, нпр. везивање језгара *CPU* за конкретне процесе, како би се обезбедило равномерно распоређивање или, ако је то потребно, ограничење *CPU* ресурса. Контролне групе контролишу и употребу меморије и *I/O* уређаје и мрежног саобраћаја. Поред контроле ресурса, контролне групе мере и прате употребу ресурса ради праћења перформанси процеса, ради наплате у окружењима рачунарства у облаку, или за оптимизацију и отклањање грешака.

Контролне групе су хијерархијски организоване, тако да прате процесе. Процеси унутар контролних група потомака наслеђују лимите и контроле од родитељских контролних група. Када се контролна група креира, она наслеђује параметре родитељске контролне групе. Управљање контролним групама често се обавља путем псеудофајл система, обично монтираног на */sys/fs/cgroup/*, омогућавајући конфигурацију контролне групе читањем и писањем у датотеке.

Контролне групе у контејнерским окружењима претежно служе за гаранцију одређене количине ресурса за сваки контејнер. Ово је кључно за одржавање стабилног и предвидивог окружења. Поред иницијалне доделе ресурса, контролне групе се могу користити и да спрече прекорачења додељених ресурса (*quota*) од стране појединих контејнера. Коришћењем ова два механизма осигурава се стабилнији рад и изолација ресурса у раду с више контејнера.

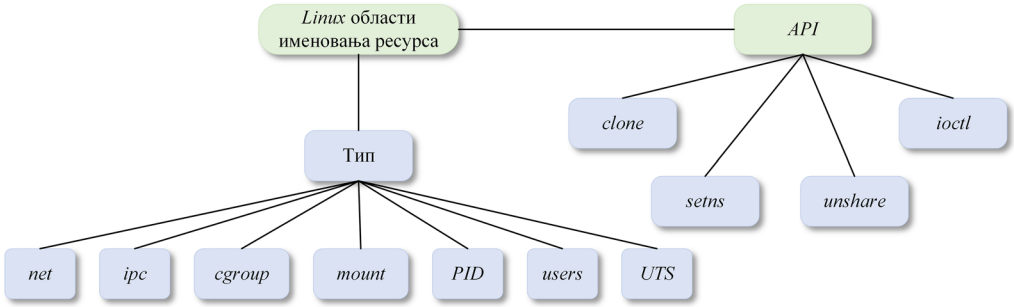
27.1.6 Изолација ресурса – области именовања (namespaces)

Контејнер представља изоловано окружење за извршавање апликација, у којем апликација има приступ само ресурсима који су јој додељени, тако да не омета рад других апликација. Изолација контејнера у односу на окружење је кључна за постизање поузданости и сигурности, посебно у окружењима где више корисника и организација дели исту инфраструктуру.

Изолација контејнера је поступак виртуелизације, у коме се апликација издваја и смешта у контејнер. Контејнер приступа ресурсима рачунара преко виртуелизованих интерфејса. Изолација контејнера се постиже коришћењем контролних група (*cgroups*) за ограничавање ресурса и области именовања (*namespaces*) за контролу приступа. Област именовања је основна технологија изолације ресурса у *Linux* кернелу која омогућава стварање виртуелних инстанци, као што су процеси, меморија, мрежни интерфејси и фајл системи. Инстанце ресурса у различитим областима именовања логички су одвојене, чиме се омогућава сегментација коришћења ресурса и изолација апликација.

Изолација ресурса као концепт и област именовања и средство реализације, омогућавају апликацијама и процесима да се извршавају у виртуелном окружењу које апликацијама изгледа као да су једини корисници ресурса који су им додељени.

Области именовања у *Linux* кернелу подељене су по врсти ресурса које контролишу, а приказане су на Слици 84.



Слика 84. Области именовања ресурса (namespaces) у Linux кернелу[627]

Контејнеру се придружује област именовања за сваки тип ресурса који користи: процесе, мрежне интерфејсе, корисничке идентитете и друге ресурсе који су изоловани од осталих контејнера и од самог *host* система. Области именовања за виртуелизацију системских ресурса у *Linux* кернелу су:

- Изолација процеса (*PID: Process ID*).
- Изолација мрежних ресурса (*NET: Networking*).
- Управљање међупроцесном комуникацијом (*IPC: Inter Process Communication*).

[627] Прилагођено према: https://8gwifi.org/docs/linux-namespace.jsp

- Изолација тачака монтирања фајл система (*MNT: Mount*).
- Изолација идентификације система (*UTS: Unix Timesharing System*).
- Изолација идентификатора корисника и група (*Users: User ID*).

Све области именовања, примењене заједно, омогућавају висок степен изолације и независности у окружењима с контејнерима, чиме се постиже сигурност и ефикасност у раду са апликацијама и ресурсима.

PID област именовања (Process ID)

Сврха *PID* области именовања је да обезбеди изолацију процеса и јединствени идентификатор процеса за контејнер. Као и процеси у *Linux*-у, област именовања процеса хијерархијски је организована. Када се креира нова област именовања процеса, додељује јој се први процес са *PID*-ом 1, који је аналоган *init* процесу оперативног система и представља тачку уласка у контејнер (*Entrypoint*). Због ове особине могуће је користити већ постојеће *PID* бројеве, који су унутар надређене области именовања. Процес ће видети само друге процесе у оквиру своје области именовања или своје деце. Може да их адресира по *PID*-овима, који су јединствени само за ту област именовања. Ова функционалност додатно је подржана и од стране *Linux* фајл система, где се у */proc/PID* директоријуму налази садржај зависан од области именовања.

Да би се постигла изолованост и на нивоу корисника, *PID* област именовања мора имати интеграцију с простором именовања корисника, како би се додатно ограничила могућност комуникације процеса који нису креирани од стране истог корисника.

Треба нагласити да постоји могућност превода *PID*-ова и њихове видљивости преко глобалне табеле процеса. Ова могућност је битна у софтверу за управљање контејнерима који се извршавају на *host*-у или у родитељском контејнеру и морају преводити *PID*-ове између различитих области именовања. Овим се омогућава алатима да повежу изоловане контејнерске процесе са онима који се налазе на самом *host* рачунару.

NET област именовања (мрежа)

NET област именовања виртуелизује приступ мрежним ресурсима тако да свака област именовања има независан скуп мрежних интерфејса, *IP* адреса, рута, виртуелне мрежне интерфејсе итд. Виртуелизацијом мрежних ресурса омогућено је да контејнер има мрежну конфигурацију која је независна од окружења. Тиме се побољшава сигурност и изолација мрежних ресурса. Изолацијом *Socket*-а у оквиру једне области именовања осигурава се да процеси у различитим *NET* областима именовања не могу комуницирати путем *IPC* механизама, осим ако то није експлицитно дозвољено. Тиме се осигурава да контејнери не могу ненамерно

комуницирати или ометати једни друге, што је кључно за сигурност и стабилност мрежних апликација.

Уз мрежне просторе именовања, један од кључних концепата који доприносе изолацији мрежних ресурса је виртуелни *ethernet* (*veth*) интерфејс, који је део *Linux* кернела. Применом *veth* интерфејса можемо повезати мрежну област именовања с главним мрежним простором, чиме се омогућава комуникација између контејнера и спољног света. Мрежни интерфејси могу бити додељени одређеним областима именовања, а сваки интерфејс у једном тренутку може бити део само једне области именовања. Ово омогућава прецизну контролу умрежавања и приступа мрежним ресурсима.

MNT област именовања (тачке монтирања директоријума)

Сврха *MNT* области именовања је изолација фајл система контејнера у односу на окружење. *MNT* област именовања омогућава сваком контејнеру независан поглед на фајл систем, чиме се ствара илузија различитих распореда фајл система за процесе који припадају различитим областима именовања.

Ова изолација омогућава контејнеру да види само део фајл система који је одређен *MNT* облашћу именовања. У оквиру слике контејнера, дефинисан је фајл систем који се монтира у његову *MNT* област именовања. На овај начин, сваки контејнер има приступ само свом фајл систему.

Монтирање (*mount*) и демонтирање (*unmount*) фајл система може утицати на друге области именовања, па је из тог разлога потребно управљати пропагацијом монтирања фајл система. Коришћењем *MNT* области именовања, за свако монтирање може се дефинисати да ли ће бити дељено, *slave*, приватно или независно. Приватне монтаже не пропагирају догађаје монтирања у друге просторе именовања, док дељене монтаже пропагирају догађаје у све просторе именовања који деле ту монтажу.

Доделом контејнера неком *MNT* простору именовања можемо постићи још виши ниво изолације променом *root*-а фајл система који ће бити видљив контејнеру, путем функције *pivot_root*. Овом функционалношћу се служи *Container Runtime*, који при креирању контејнера мења *root* фајл систем контејнера на нови, који је дефинисан у слици контејнера. На овај начин, сваки контејнер може имати независан *root* фајл систем који му омогућава да функционише као одвојена јединица са својим ресурсима.

IPC област именовања (међупроцесна комуникација)

Изолацијом механизама међупроцесне комуникације (*IPC*) можемо осигурати независност комуникацијских канала између процеса. Кроз *IPC* просторе именовања, објекти *System V IPC*-а (семафори, редови порука, дељена меморија)

као и *POSIX* тредови порука постају изоловани. То значи да свака област именовања може креирати *IPC* објекте са истим идентификаторима, али без сукобљавања са *IPC* објектима у другим областима именовања. Ова изолација је кључна за неометану и сигурну комуникацију између процеса унутар одвојених контејнера.

Пожељно је да се за контејнер ограничи приступ *IPC*, поготово у системима с више корисника, где је важно да један контејнер не може видети комуникацију *IPC*-а другог контејнера. Апликације унутар контејнера могу користити механизме *IPC*-а, изолацијом њихових кључева, што омогућава да се два контејнера која покрећу исти софтвер неће сударати у погледу *IPC* кључева. Ово је посебно важно у ситуацијама где се користе контејнери који се темеље на истим сликама и имају сличне конфигурације, али је потребно да раде независно један од другог.

Као што је то случај с процесима, *IPC* ресурси могу постати напуштени када процеси који их користе заврше рад или се прекину. Унутар *IPC* области именовања, постоје механизми који осигуравају да се ови ресурси чисте како би се спречило исцрпљивање ресурса и одржала стабилност система. За дуготрајне контејнере или системе са честим стварањем/уништавањем контејнера, неопходно је осигуравање правилног чишћења ресурса *IPC*-а, како би се спречила исцрпљеност ресурса и омогућио несметан рад и комуникација контејнера.

UTS област именовања (UNIX Timesharing System)

Идентификација система представља кључни аспект конфигурације рачунарских система и њихове комуникације с другим системима. *UTS* област именовања омогућава изолацију идентификационих параметара, као што је име *host*-а, тако да процеси у различитим областима именовања могу видети различите излазе *uname* команде. За апликације које бележе информације о систему или за услуге које су свесне имена *host*-а, *UTS* област именовања омогућава сваком контејнеру да поседује свој идентитет. Ово омогућава да логови, ревизије и понашање сервиса буду доследни и изоловани унутар сваког контејнера.

USER област именовања (USER ID)

Контејнери користе *USER* области именовања за изолацију корисника и група између различитих контејнера. Посебна пажња се посвећује подешавању *root* привилегија у оквиру контејнера. Ако непривилеговани процес унутар контејнера добије *root* привилегије, јавља се сигурносни ризик због могућности приступа ресурсима изван контејнера. Такви случајеви се избегавају применом *USER* области именовања, која осигурава да процеси имају само неопходне привилегије, које важе једино унутар контејнера.

USER област именовања омогућава мапирање *UID*-а (*User ID*) и *GID*-а (*Group ID*) између *host*-а и других области именовања. Ово мапирање омогућава да процес

унутар једне области именовања има привилегије *root* корисника (*UID* 0), али се мапира на непривилегованог корисника изван те области. Таква изолација је кључна за сигурност и ограничавање могућности неовлашћеног приступа рачунарским ресурсима.

Поступак дефинисања области именовања при креирању контејнера

Поступак дефинисања области именовања је следећи:

1. Креира се *USER* област именовања.
2. Ремапира се *UID* процес на креирану *USER* област именовања.
3. Креира се *UTS* област именовања.
4. Промени се *hostname* у креираној *UTS*, тако да одговара називу контејнера.
5. Креира се *IPC/CGROUP* област именовања за процесе и међупроцесну комуникацију.
6. Креира се *NET* област именовања за мрежне ресурсе.
7. Креирају се парови виртуелних мрежних интерфејса за умрежавање.
8. Креира се *PID* област именовања за процесе у оквиру контејнера.
9. Креира се *MNT* област именовања за фајл системе у оквиру контејнера.
10. Премести се *root* директоријум за апликацију на фајл систем у контејнеру.
11. Монтира се */proc* директоријум унутар новог *root* директоријума контејнера.
12. Стари *root* директоријум се искључи (*unmount*).
13. Премешта се процес на нови *root* директоријум у оквиру контејнера.

27.2 Екосистем контејнера

Компоненте контејнерског екосистема омогућавају креирање, покретање и управљање радом контејнера. Свака компонента екосистема има специфичну улогу и функцију:

- **Слика контејнера** (*container image*) је основна јединица за креирање инстанце контејнера и представља бинарну датотеку која садржи апликацију и све њене зависности, укључујући библиотеке и конфигурационе датотеке. На основу слике контејнера креирају се **инстанце контејнера** које се извршавају. Слика контејнера се састоји из низа слојева, од којих сваки слој представља логички скуп промена над фајл системом слике контејнера. Ови слојеви се слажу један преко другог како би се формирала коначна слика контејнера. Слој слике контејнера се не може мењати након креирања, али је промену слике контејнера могуће постићи додавањем нових слојева на постојећу слику, што омогућава брзу и ефикасну израду нових верзија апликације.
- ***Container runtime*** је софтверска компонента одговорна за покретање и управљање контејнерским инстанцама креираним из слика контејнера. *Runtime* сарађује

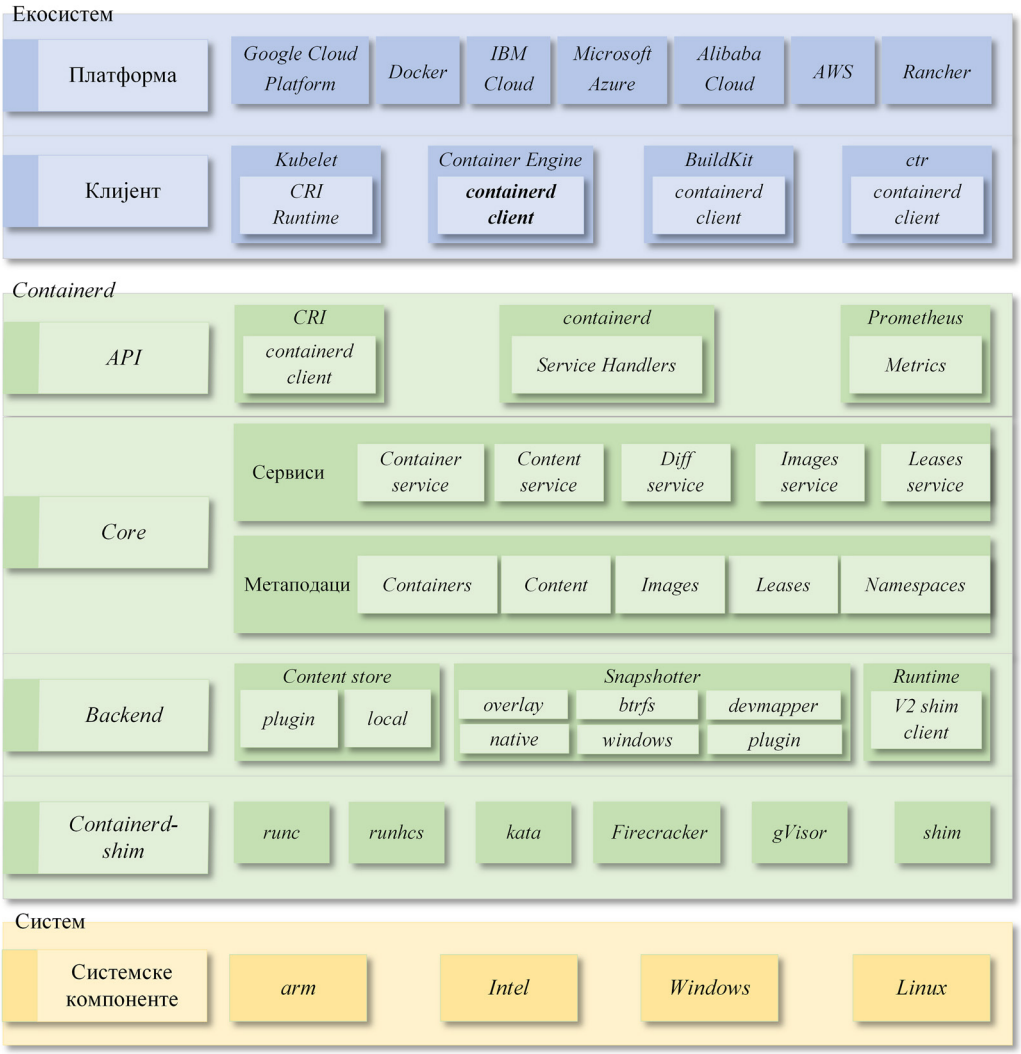
с кернелом оперативног система како би осигурао функције за изолацију и управљање ресурсима, као што су простори именовања и контролне групе.

- ***Container engine*** представља проширење у односу на *container runtime* и пружа додатне функционалности за управљање контејнерима, као што су креирање, покретање, заустављање и брисање контејнера, али и управљање његовим сликама. *Container engine* се често користи за аутоматизацију процеса управљања контејнерима и олакшава рад путем практичних *CLI* и *API* позива.
- **Регистар контејнера** је сервис који омогућава складиштење и дељење слика контејнера. Регистар често служи као централно место за дистрибуцију слика контејнера између тимова и организација. Такође, омогућава преношење слика контејнера на различите рачунаре, где се могу користити за покретање контејнерских инстанци. Регистар контејнера користи *RESTful API* за преузимање и отпремање слика контејнера. Омогућава верзионисање слика контејнера, као и аутентификацију и ауторизацију корисника. Постоје јавни регистри, попут *DockerHub*-а, а компаније често креирају сопствене регистре доступне само корисницима унутар компаније.

Контејнерски екосистем са својим компонентама омогућава флексибилност и ефикасност у управљању апликацијама и инфраструктуром. Он омогућава брзо и лако постављање апликација и њихових зависних компоненти, без потребе за комплексном конфигурацијом и управљањем. Уз то, контејнерски екосистем омогућава преносивост апликација између различитих окружења и платформи, што олакшава развој и имплементацију софтвера у различитим контекстима[628].

Контејнерски екосистем се у пракси може преклапати с компонентама других екосистема. Приказ контејнерског екосистема (*Docker*) дат је на слици 85.

[628] Syed, M. H. & Fernandez, E. B. (2018, August). A reference architecture for the container ecosystem. In: *Proceedings of the 13th international conference on availability, reliability and security* (pp. 1–6 https://www.researchgate.net/profile/Eduardo-Fernandez-40/publication/327002766_A_reference_architecture_for_the_container_ecosystem/links/5b9c3bb8299bf13e6031734d/A-reference-architecture-for-the-container-ecosystem.pdf



Слика 85. Екосистем контејнера[629]

27.3 Животни циклус контејнера

Животни циклус контејнера представља серију корака и стања кроз која контејнер пролази током свог постојања. Први корак у животном циклусу контејнера је креирање слике контејнера, на основу дефиниције која се налази у текстуалној датотеци, а која се по конвенцији назива *Dockerfile*. Ова датотека садржи низ инструкција које дефинишу слику на којој ће контејнер бити заснован, као и

[629] Прилагођено према: https://workshops.qnib.org/container-in-hpc-workshop-101/container/docker/index.html#docker-ecosystem

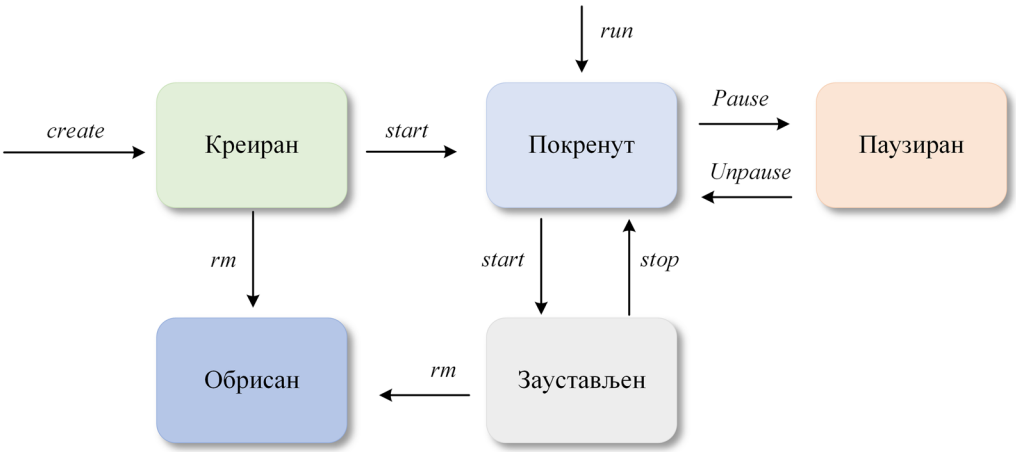
низ додатних конфигурација, софтверских пакета, библиотека и апликација инсталираних у контејнер. Након што је ова датотека креирана, може се користити за изградњу слике контејнера уз помоћ алата за прављење контејнерских слика.

Када је, на основу *Dockerfile*, слика контејнера направљена, може се складиштити у контејнерском регистру, који представља централизовано складиште слика контејнера. Контејнерски регистри омогућавају чување и дељење слика контејнера, што олакшава дистрибуцију и употребу контејнера од стране више корисника у различитим окружењима.

Покретање контејнера почиње креирањем инстанце контејнера, на основу слике контејнера. Овај процес укључује иницијализацију коју спроводи *container runtime* и подразумева подешавање потребних ресурса и њихову изолацију, пре него што покрене контејнерску апликацију у изолованом окружењу контејнера.

Када је контејнер покренут, он прелази у активно стање и почиње извршавање апликације или услуге дефинисане у контејнеру. Контејнер може остати у активном стању докле год је је то потребно за функционисање апликације. Стањем контејнера може се манипулисати путем *container engine*-а, помоћу којег је могуће паузирати рад контејнера, поново га покренути или трајно зауставити. Када је извршавање контејнера завршено, прекида се извршавања апликације унутар контејнера, те контејнер прелази у неактивно стање и може бити заустављен и уклоњен из система. Уклањање контејнера обухвата брисање инстанце контејнера из система и чишћење свих повезаних ресурса и конфигурација.

Стања контејнера у оквиру животног циклуса приказана су на Слици 86.



Слика 86. Животни циклус контејнера

Управљање животним циклусом контејнера може бити сложен задатак, посебно када се ради о великим и динамичним контејнерским окружењима у којима постоје многе инстанце контејнера које су у међусобној зависности. Алати за оркестрацију контејнера, попут *Kubernetes*-а, омогућавају аутоматизацију и оптимизацију

управљања контејнерским инстанцама током њиховог животног циклуса. Ови алати пружају могућност за аутоматско распоређивање, скалирање и управљање контејнерима, што олакшава постављање, управљање и одржавање контејнерских апликација у различитим окружењима и на различитим инфраструктурним платформама.

27.4 Умрежавање контејнера

Поред управљања процесима, *Linux* оперативни систем пружа могућност за управљање мрежним ресурсима. Функционално повезивање контејнера на мрежном нивоу обезбеђује се коришћењем ресурса кернела, као што су: виртуелни *ethernet* интерфејс, виртуелни *bridge*, а за контролу рутирања мрежног саобраћаја *IPTables*.

Изолација мрежних ресурса постиже се применом *network* области именовања (*NET namespaces*), који виртуелизује *network stack*, тако што свака *network* област именовања има своје мрежне ресурсе као што су: мрежни интерфејси, *IP* адресе, табеле рутирања, тунели и *firewall*. Правило додато у *network* област именовања преко *IP Tables*, примењује се само на мрежне пакете који долазе и одлазе из те области именовања.

Виртуелни *Ethernet device* (*veth*) представља виртуелизовани мрежни интерфејс, сличан мрежној картици. Виртуелна мрежна картица повезује контејнер са окружењем. *Veth* се увек прави у повезаним паровима. Један *veth device* је доступан у оквиру контејнера, док је други доступан *host* систему. Комуникација између ова два интерфејса је у облику тунела, у смислу да пакети који су послати на један интерфејс, бивају прослеђени до његовог пара.

Network namespace обезбеђује механизам за распоређивање мрежних ресурса у независне и изоловане целине, чији је једини механизам међусобне комуникације преко *veth* интерфејса. Међутим 1:1 мапирање између *veth* интерфејса и потенцијално великог броја контејнера је непрактично. Потребно је да се, поред *veth* интерфејса и области именовања, уведе и могућност рутирања.

Једна од најчешће коришћених могућности за потребе међусобног повезивања виртуелних етернет интерфејса различитих контејнера, као и физичких мрежних интерфејса *host*-а, јесте коришћење *Linux bridge*, који обезбеђује, слично као физички *switch*, могућност *L2* повезивања и рутирања пакета на основу *MAC* адреса изворног и одредишног интерфејса.

Рутирање преко *MAC* адресе даје могућност рутирања пакета између различитих интерфејса, било физичких или виртуелних, који су прикључени на исти *bridge*. На овај начин, могуће је постићи мрежно повезивање два контејнера, где је сваки у оквиру свог простора именовања.

Linux bridge ради само на *L2* слоју и нема могућност рутирања пакета ван *host*

рачунара, осим прослеђивања на физички етернет интефејс *host*-а. Да би се омогућио рад на *L3* слоју, тј. рутирање између различитих мрежа, потребно је увести рутирање на *IP* протоколу. Најчешћи алат који се за то користи је *Linux IPTables*. *IPTables* рутирање ради тако што кернел прегледа *IP* пакете који пролазе кроз мрежни интерфејс, и дозвољава кориснику да декларише правила преко којих би се вршило рутирање тих пакета. Ова правила се декларишу преко табела. Најчешће коришћена табела је *Filter*, која се употребљава за рутирање пакета на основу њихове путање, преко правила за *Input*, *Forward* и *Output chain*. *Input* правила се примењују на пакете који имају одредиште у локалним интерфејсима; *forward* правила се примењују на пакете које је потребно проследити даље кроз систем; *output* се користи за локално генерисане пакете. Ова правила подразумевају одређене акције, које могу бити прихватање, одбијање или стављање пакета у ред чекања. Активација ових акција дефинише се по неком критеријуму као што је *IP* адреса, порт, протокол итд.

С популаризацијом микросервисне архитектуре, *Cluster Networking* или управљање мрежом у оквиру кластера постаје кључна ставка скалабилне комуникација између контејнера који су распоређени на различите виртуелне машине у оквиру кластера. Платформе за оркестрацију контејнера саме по себи не нуде механизам за подешавање кластерске мреже ради комуникације између контејнера. Подешавања мреже се морају вршити путем екстерних алата, који преко интеграција с платформама за оркестрацију могу аутоматизовати креирање, брисање и управљање виртуелним мрежним ресурсима. Типичан пример су два алата: *Flannel* и *Calico*, који *Kubernetes* кластерима пружају изоловану, сигурну и високоприлагодљиву мрежну комуникацију између контејнера у оквиру кластера.

27.5 Фајл системи контејнера

Основни проблем код фајл система за контејнере је како изоловати, управљати и оптимизовати фајл систем унутар контејнера, а истовремено одржати брзину и лакоћу коришћења. Традиционални *Linux* фајл системи нису пројектовани да се носе с динамичком и слојевитом природом контејнерских апликација. Да би се постигла жељена изолација користећи стандардне фајл системе, свака промена или ажурирање захтевали би копирање читавог фајл система, што би довело до непотребног губитка времена и складишног простора. Како би се превазишли ови изазови, развијен је *Union File System* (*UFS*), који омогућава слојевито складиштење, у коме се измене чувају у одвојеним слојевима, док основни фајл систем остаје непромењен. Ова метода контејнерима пружа флексибилност и ефикасност, док истовремено чува интегритет и структуру фајл система.

UFS користи различите технике како би постигао рад у слојевима, као што су гране, *Whiteout* и *Opaque*, *Copy-On-Write* (*CoW*). Такође, подржава рад с гранама, где може постојати више грана чији је корен у оргиналном фајл систему. Гране могу бити само за читање (*RO*), или за читање и писање (*RW*). Приликом уједињавања грана,

резултирајући фајл систем комбинује њихов садржај. Редослед грана је обично сложен према приоритету, при чему датотека из гране вишег приоритета може заменити исту такву датотеку у грани нижег приоритета.

При брисању датотека *UFS* користи *Whiteout* маркере којима означава брисање и *Opaque* маркере којим означава брисање целог директоријума. Модификација фајлова у оквиру *UFS*-а обавља се путем *Copy-On-Write* (*CoW*) стратегије, где приликом модификације датотеке, оригинална датотека остаје непромењена, док се измена примењује на копију у грани која је за читање и писање. Сваки слој представља скуп измењених датотека. Основни слој је најчешће минимални *OS* фајл систем, док су накнадни слојеви задужени за додавање апликација, библиотека и других датотека. Када се слика контејнера покрене, *UFS* спаја слојеве, дајући контејнеру уједињен приказ фајл система.

Осим ефикасног управљања променама, предност *UFS*-а огледа се и у оптимизацији складиштења, брзини иницијализације контејнера и могућности верзионисања. Од контејнера се очекује да деле најниже слојеве слика, тиме смањујући редундантност и оптимизујући употребу складиштеног простора. Истовремено, иницијализација контејнера постаје бржа јер није потребно копирати читав фајл систем, већ само креирати нови слој за писање. *UnionFS* такође омогућава једноставно верзионисање слика контејнера, где свака промена или ажурирање може представљати нови слој.

У екосистему контејнера, постоје разне имплементације *union* фајл система, укључујући *Overlay*, *Overlay2*, *aufs* и *btrfs*. Међу њима, *Overlay2* се показао као најпопуларнији због ефикасности и поузданости.

27.6 Управљање волуменима у контејнерском окружењу

Коришћење специјализованог фајл система решило је проблеме оптимизације и измене података у оквиру контејнера. При раду с контејнерима честа је пракса да је потребно користити екстерне складишне ресурсе, који се могу налазити или на екстерном систему, или ван контејнерске области именовања.

Други проблем који се често јавља је перзистентност података, због слојевите структуре контејнерских слика, где је само последњем слоју омогућено читање и писање, и то обично у оквиру радне меморије. Применом овог принципа рада, све промене над фајл системом унутар контејнера нестаће када се он заустави или обрише. Ово представља изазов за апликације које захтевају стални приступ подацима, као што су базе података.

Како би се превазишао проблем, уведен је концепт „волумена“, који служи као мост између контејнера и екстерног механизма перзистентности података. Применом волумена може се постићи трајно чување и приступ подацима, чак и након што се контејнер заустави или обрише.

Пошто се информације о волуменима не налазе у склопу слике контејнера, посао

рада с волуменима не може вршити *Container runtime*, већ се тај посао препушта за *Container Engine*, који нуди начин за постизање перзистентног складиштења и дељења података између контејнера и *host*-а. Са становишта програма који користи волумене, сваки волумен, независно од своје имплементације, контејнеру ће се приказивати као монтирани директоријум у оквиру његовог фајл система. Све операције уписивања или читања унутар монтираног директоријума биће усмерене ка *Container Engine*-у, који ће даље, зависно од типа волумена, спроводити те операције изван контејнера, независно од тога да ли су подаци на физичком рачунару или у оквиру екстерног кластера.

Постоје две врсте волумена: перзистентни и транзијентни. Перзистентни волумени чувају податке и након што се контејнер заустави или уништи. Ова врста волумена је есенцијална за апликације које захтевају континуални приступ подацима, као што су базе података. С друге стране, транзијентни волумени, који су привременог карактера, могу бити корисни за привремено складиштење, логове или кеширање података без потребе за дуготрајним складиштењем.

У системима с више чворова, као што су кластери, управљање волуменима постаје сложеније и скалабилност ових система је од кључне важности за перформансе дистрибуираних контејнерских апликација. Системи за оркестрацију морају превазићи проблем у коме се контејнер може покретати на било којем чвору унутар кластера, али му се мора омогућити приступ одговарајућем волумену, без обзира на то где се тачно налази физички складишни ресурс. Решење овог проблема често се постиже употребом дистрибуираних фајл система или специфичних складишних решења, који омогућавају монтирање на директоријум, независно од чвора на којем се контејнер извршава. Применом ових система, добијамо већу флексибилност у распоређивању контејнера и осигуравамо високу доступност података, јер се подаци у овим системима често реплицирају на више чворова.

27.7 Регистар контејнера

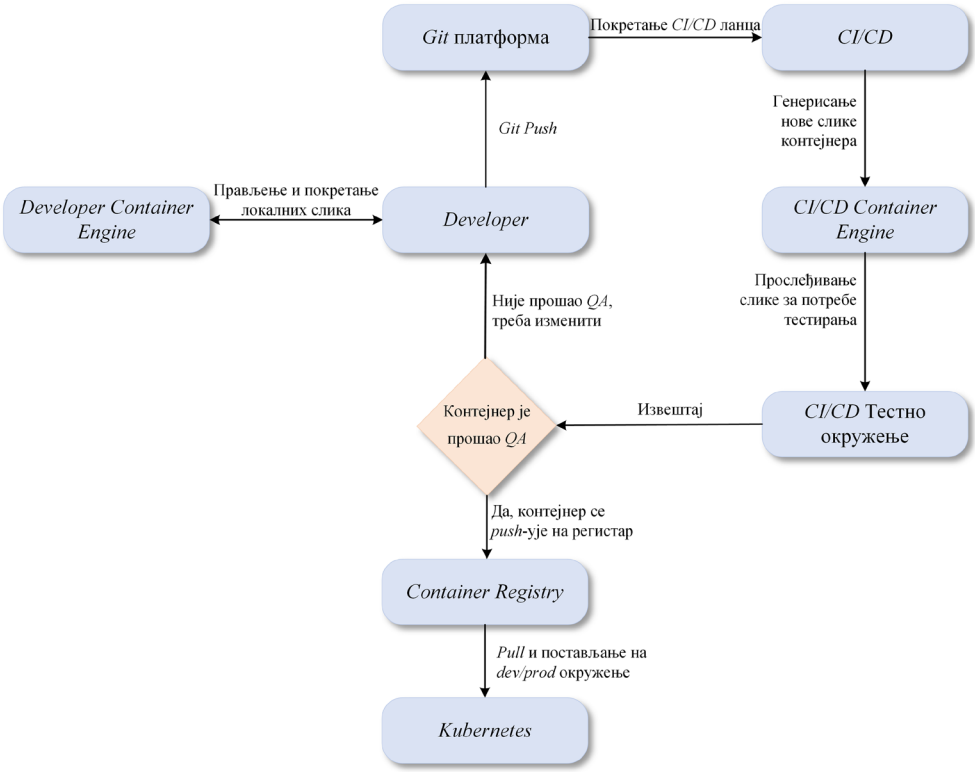
Једна од главних предности контејнера је у томе да су независни од окружења, чиме се избегава честа појава у другим окружењима да на неким платформама или рачунарима апликација ради, док на некима не ради. Употребом контејнера у развојном процесу, овај проблем се елимише, тако што се са сваком изменом кôда поново генерише нова слика контејнера која се доставља корисницима. Међутим, при раду с више корисника, јавља се проблем дистрибуције новогенерисане слике контејнера. Како би се решио проблем размене контејнерских слика, уведене су платформе специјализоване за њихово чување и дистрибуцију, у форми контејнерских регистара.

Уколико се у контејнерски екосистем укључе регистри, тада ће се ток развоја апликација променити, и то на следећи начин: након измене кôда, потребно је проверити конфигурациону датотеку (*Dockerfile*), на основу које ће се креирати слика, која ће се за сада налазити само на локалној машини у оквиру локалног

Container Engine-а. Да би другима омогућили приступ креираној слици, можемо је отпремити на контејнерски регистар путем *push* команде која ће уз креденцијале за приступ платформи поставити слику у регистар и ставити је на располагање ауторизованим члановима тима. Чланови тима преузимају нову верзију слике *pull* командом, а она ће након скидања свих слојева слике бити доступна у оквиру њиховог *Container Engine*-а.

Један од захтевнијих и компликованијих поступака у развоју софтвера је постављање апликације у продукционо окружење. Применом контејнерских регистара, ова процедура се може стандардизовати и олакшати. Продукционо окружење може бити свесно нове верзије, и након преузимања заменити стари контејнер новим.

Управо због своје улоге као централног чвориштa, контејнерски регистри често садрже низ додатних алатки и механизама који олакшавају рад са сликама контејнера. Поред основних функција отпремања и преузимања, многи регистри нуде могућности за верзионисање слика, аутоматско ажурирање и интеграцију са *CI/CD* алатима за постављање у продукционо окружење. Ове додатне функционалности омогућавају тимовима да одржавају квалитет слика контејнера које се користе у продукцији. Пример коришћења регистра контејнера у *CI/CD* приказан је на Слици 87.



Слика 87. Пример рада с контејнерима у CI/CD системима

Међу најчешће коришћеним контејнерским регистрима су: *Docker Hub*, *Google Container Registry* (*GCR*), *Amazon Elastic Container Registry* (*ECR*) и *Azure Container Registry*. Сваки од ових регистара нуди функционалности и предности које се прилагођавају различитим потребама и захтевима корисника, и обично нуде висок ниво интеграције са осталим сервисима у оквиру понуде тог *cloud* провајдера.

27.8 Стандардизација контејнерског екосистема

Стандардизација контејнерског екосистема је кључни фактор за усвајање и широко коришћење контејнера. За то највеће заслуге припадају *Open Container Initiative* (*OCI*)[630] , пројекту великог броја софтверских предузећа, који има за циљ стварање отворених стандарда за коришћење контејнерских технологија. Стандарди које је дефинисао *OCI* омогућавају интероперабилност између различитих контејнерских технологија и олакшавају интеграцију контејнерског екосистема с другим деловима *IT* инфраструктуре.

OCI је поставио два главна стандарда у контејнерском екосистему: стандард који дефинише структуру слике (*image*) контејнера и стандард који дефинише *runtime* контејнера. Стандард за слике контејнера дефинише формат и структуру слика контејнера, док стандард за *runtime* контејнера дефинише интерфејс за покретање и управљање контејнерских инстанци.

Ова два стандарда омогућавају интероперабилност између различитих контејнерских технологија и омогућавају корисницима да користе различите контејнерске алате и платформе без потребе за прилагођавањем свакој од њих[631]. Стандардизација контејнерског екосистема има више предности. Она омогућава већу преносивост слика контејнера између различитих контејнерских платформи и окружења. Олакшава интеграцију контејнерских алата и платформи с другим деловима *IT* инфраструктуре, као што су системи за аутоматизацију, оркестрацију и управљање. Подстиче сарадњу и иновације у контејнерском екосистему, јер омогућава различитим контејнерским технологијама да међусобно комуницирају.

27.8.1 OCI Image-spec

OCI Image-spec[632] представља унифициран и стандардизован оквир за паковање и дистрибуцију слика контејнера. Омогућава високу интероперабилност у функционисању алата и платформи различитих произвођача. *OCI Image-spec* користи *blob*-ове за складиштење градивних елемената слике контејнера. ***Blob*** представља информациону структуру сличну датотеци која представља скуп бинарних или текстуалних података. Стандард прописује да се **манифест** користи

[630] https://opencontainers.org/
[631] https://opencontainers.org/posts/blog/2023-07-21-oci-runtime-spec-v1-1/
[632] https://github.com/opencontainers/image-spec

за опис структуре и конфигурације слике. За управљање различитим манифестима унутар једне слике користи се ***index.json*** датотека.

Директоријум Blobs

У *OCI Image-spec*, директоријум *Blobs* представља централно складиште за *blob*-ове прописане овом спецификацијом. *Blob*-ови су адресирани и њима се приступа преко *SHA-256* хеша. *Blob*-ови представљају модуларни градивни елемент слика контејнера и садрже слојеве фајл система, метаподатке и кључне конфигурационе поставке. Управљање и идентификација *blob*-ова су стандардизовани кроз дескрипторе, који пружају додатне метаподатке и омогућавају њихову лакшу манипулацију и обраду, укључујући класификацију путем стандардизованих *MIME* медијских типова података.

Индекс слике контејнера

Једна од кључних датотека у оквиру *OCI* спецификације је *index.json*. Назива се и индекс слике контејнера. Ова датотека омогућава приступ различитим манифестима, који представљају појединачне слике контејнера за различите архитектуре, као што су *amd64*, *arm64* и друге. *Index.json* садржи референце ка манифестима, али не упућује директно на *blob*-ове који чине слике. Та функција је делегирана манифестима. На тај начин *index.json* омогућава груписање различитих манифеста слика под једним ентитетом, чиме се постиже већа флексибилност и ефикасност у управљању ресурсима.

Манифест

Манифест слике је документ који пружа детаљан опис једне конкретне слике контејнера, идентификујући слојеве и конфигурациони *blob*. Манифест је директно повезан са *index.json* датотеком и служи за склапање различитих компоненти у извршни контејнер. Манифест описује слојеве фајл система и конфигурационе *blob*-ове. На овај начин, манифест делује као посредник између *index.json* датотеке и *blob*-ова, пружајући детаљне информације о томе како се слојеви и конфигурације комбинују при креирању функционалног фајл система контејнера. Такође, манифест садржи криптографске сажетке *blob*-ова, који гарантују интегритет и аутентичност сваке компоненте слике, што је од суштинског значаја за сигурност у екосистему контејнера.

Формирање слике контејнера

Манифест, *index.json* и *blob*-ови чине основу структуре контејнерске слике. Процес формирања слике контејнера почиње анализом и интерпретирањем садржаја

датотеке *Dockerfile*. Свака инструкција у *Dockerfile* резултује креирањем новог *blob*-а, који се складишти у директоријум *Blobs,* а који ће даље бити референциран унутар креираног манифест фајла. *Container Engine* користи манифест да би комбиновао *blob*-ове у коректном редоследу, чиме се формира коначна слика контејнера. Након што су сви релевантни *blob*-ови креирани и референцирани у манифесту, *Container Engine* генерише *index.json* датотеку, којом групише манифесте различитих архитектура. *Index.json* омогућава референцирање на више верзија исте слике, сваке са својим манифестом. На овај начин *Container Engine* трансформише *Dockerfile* у *OCI* компатибилну слику контејнера, која се може даље дистрибуирати у контејнерске регистре.

27.8.2 OCI Runtime-spec

OCI Runtime-spec[633] представља стандардизовану спецификацију и стандардизован скуп параметара за извршавање контејнера. Спецификација садржи прецизне инструкције о алокацији ресурса, начинима изолације контејнера и управљању њиховим животним циклусом. Конфигурациона датотека, дефинисана у *OCI Runtime-spec*, садржи све техничке детаље потребне за иницијализацију и рад контејнера: путању до главног процеса, неопходне аргументе, поставке окружења, мрежне конфигурације, као и детаље управљања ресурсима, попут *CPU* и меморије.

Иницијализација контејнера

Ток стварања и управљања контејнером почиње дефинисањем жељених иницијализационих параметара у конфигурационом фајлу. Након што се иницијализује, контејнер прелази у фазу алокације ресурса и креирања изолованог окружења. Ова фаза укључује употребу контролних група (*cgroups*) и области именовања (*namespaces*), које су стандардизоване кроз *OCI Runtime-spec*. Њиховом применом по *OCI* стандарду омогућава се да контејнери ефикасно користе ресурсе, а да притом не ометају рад осталих процеса или контејнера на истом систему.

Управљање контејнерима

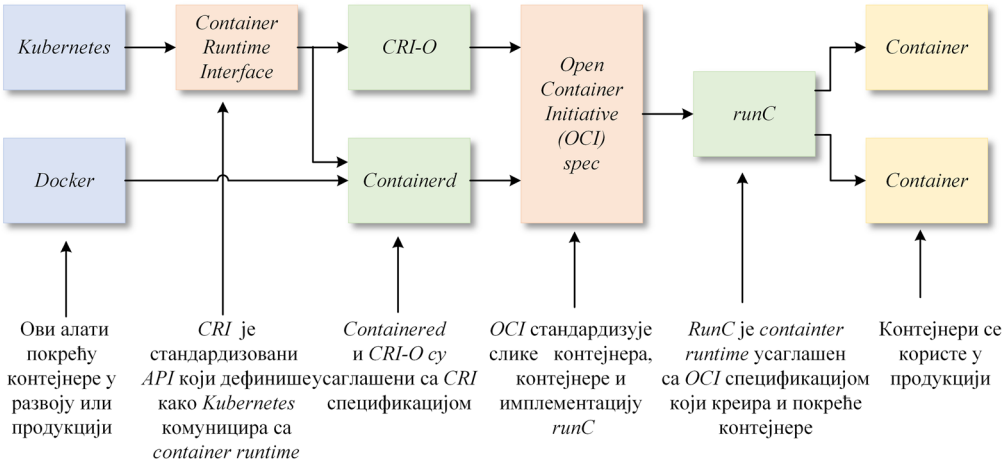
Када је окружење у којем ће се контејнер покретати иницијализовано, контејнер се покреће, поштујући дефинисане изолационе и ресурсне параметре. *OCI Runtime-spec* пружа детаљне смернице за транзицију стања контејнера и акције којима се може утицати на животни циклус контејнера, нпр. паузирање, настављање рада и заустављање контејнера. Паузирањем контејнера, сви се процеси унутар контејнера привремено обустављају, а наставак подразумева реактивацију процеса. Ако је рад контејнера завршен, било планирано или услед неке грешке, спецификација прописује поступак за његово правилно гашење и ослобађање ресурса.

[633] https://github.com/opencontainers/runtime-spec

Имплементације OCI Runtime-spec

Имплементација *OCI Runtime-spec*-а има више, међу њима су *RunC*, *Kata Containers* и *gVisor*. Свака од ових имплементација има своје карактеристике и предности. *RunC* је најпознатија и често се користи као референтна имплементација *OCI* стандарда. Ослања се директно на *Linux* кернел и омогућава лаку интеграцију са осталим алатима и системима. *Kata Containers* пружа додатни слој изолације у односу на *RunC*, кроз употребу лаких виртуелних машина, чиме се постиже боља безбедност без значајног утицаја на перформансе. С друге стране, *gVisor* користи модификовану верзију *Go runtime*-а за извршавање апликација унутар контејнера, пружајући тако снажну изолацију уз минималне губитке перформанси.

Процес оркестрације контејнера применом *OCI* стандарда приказан је на Слици 88.



Слика 88. OCI стандарди у оркестрацији контејнера[634]

27.9 Container runtime и container engine

Алати *container runtime* и *container engine* међусобно су повезани, али имају различите и специфичне функције и улоге у екосистему контејнера.

Container Runtime

Container Runtime је одговоран за непосредно креирање и управљање животним циклусом контејнера. Представља основни алат за извршавање контејнера. У свом раду користи функционалности оперативног система, као што су области

[634] Прилагођено према: https://www.techtarget.com/searchitoperations/definition/Open-Container-Initiative

именовања и контролне групе за обезбеђење изолације контејнера и управљање ресурсима. *Container Runtime* имплементира *OCI runtime* спецификацију како би се осигурала конзистентност и предвидивост у управљању контејнерима.

Container Engine

Container Engine представља софистициран софтверски алат који додатно олакшава рад с контејнерима. Активно користи *Container Runtime* за иницијализацију и управљање контејнерима, али пружа и широк спектар додатних функција, укључујући функционалности везане за слике контејнера, управљање волуменима, мрежном конфигурацијом и оркестрацију. *Container Engine* је дизајниран тако да може користити *Container Runtime* различитих произвођача. Међу познатијим имплементацијама *Container Engine*-а су *Docker* и *containerd*[635] који је иницијално био компонента *Docker*-а, али је постао независан алат отвореног кôда фокусиран на боље перформансе.

У оперативној пракси, када корисник покрене контејнер, *Container Engine* извршава низ акција: преузимање слике из контејнерског регистра, припрема слике контејнера, подешавања мрежних и складишних ресурса. На крају се позива одговарајући *runtime* да иницијализује контејнер с претходно дефинисаном конфигурацијом.

27.10 Cloud native приступ развоју софтвера

Cloud native је приступ развоју софтвера који се фокусира на изградњу и покретање апликација које користе предности инфраструктуре и сервиса рачунарства у облаку. За разлику од традиционалних модела, који се ослањају на монолитне архитектуре и локалне сервере, *cloud native* апликације су пројектоване тако да у потпуности функционишу у *cloud* окружењу у којем су скалабилност, модуларност и отпорност на грешке брига провајдера сервиса рачунарства у облаку. У сервисима рачунарства у облаку, контејнери представљају један од темељних градивних елемената, омогућавајући једноставну имплементацију микросервисне архитектуре и једноставно управљање ресурсима.

Cloud native се не односи само на технологије као што су контејнери и оркестрација, већ представља целовит приступ који обухвата различите аспекте развоја софтвера, од дизајна и имплементације, до испоруке и управљања апликацијама. Права снага *cloud native* приступа лежи у његовом екосистему, који код већине провајдера рачунарства у облаку обухвата *DevOps* примере најбоље праксе, аутоматизацију *CI/CD*, и низ додатних сервиса као што су базе података, складиштење података и аналитички алати.

[635] https://containerd.io/

Приступи развоју у *cloud native* окружењима могу бити различити, неки се ослањају на *serverless* архитектуру, други користе *Managed Kubernetes* кластере, а неки чак комбинују више *cloud* провајдера у једном решењу. Кључ је у томе да су све ове апликације дизајниране на начин да максимално искористе *cloud* окружење, од скалирања и отпорности на отказе, до интеграције са осталим *cloud* сервисима.

Cloud Native Computing Foundation (*CNCF*)[636] је организациони стуб овог динамичног екосистема, дефинишући смернице, стандарде и најбоље праксе. *CNCF* такође промовише пројекте и технологије отвореног кôда које доприносе *cloud native* екосистему, пружајући подршку за даљу иновацију и интероперабилност.

27.11 Оркестрација контејнера

Оркестрација контејнера је аутоматизовано управљање животним циклусом контејнерских апликација у *cloud native* окружењима. Аутоматизацијом се решава низ случајева коришћења који се срећу у раду са апликацијама састављених из више контејнера, као што су: распоређивање, балансирање оптерећења, аутоматски опоравак, испорука и ажурирање апликације.

Контејнери омогућавају паковање и изолацију апликација са свим њиховим зависностима, што поједностављује дистрибуцију и скалирање. Управљање мањим бројем контејнера може се обављати и манулно, међутим, када је у питању управљање великим бројем контејнера, нарочито у дистрибуираним и микросервисним архитектурама, оркестрација као аутоматизовани поступак постаје неопходна.

Распоређивање контејнера

Распоређивање контејнера је централна компонента сваког оркестрационог система. Системи као што su *Kubernetes*, *Docker Swarm* или *Apache Mesos*, одређују на којим ће физичким или виртуелним машинама контејнери бити покренути. Приликом одлуке, узимају се у обзир различити фактори као што су расположиви ресурси, зависности и специфични конфигурациони параметри.

Балансирање оптерећења

Комуникација и балансирање оптерећења такође су од есенцијалне важности за постизање скалабилности апликација. Оркестрациони алати осигуравају да контејнери могу комуницирати, независно од њихове локације, док се ресурси могу динамички алоцирати у случају повећања или смањења саобраћаја, како би се одржале оптималне перформансе и доступност.

[636] https://www.cncf.io/

Аутоматски опоравак

У случају кварова система или појединачних контејнера, оркестрација је задужена за аутоматски опоравак, иницирајући нове инстанце контејнера на алтернативним серверима. Ово омогућава континуалну доступност сервиса.

Оркестрација контејнера није само есенцијални део савремених *cloud native* архитектура, већ и стратешка технологија која омогућава предузећима да на најбољи начин искористе предности контејнеризације. Интеграцијом оркестрационих алата, предузећа могу управљати сложеним дистрибуираним системима, с већом ефикасношћу, агилношћу и отпорношћу на отказе.

27.11.1 Single host оркестрација – Docker Compose

Docker Compose је алат који омогућава дефинисање и управљање контејнерским апликацијама, коришћењем *Docker engine*-а. Иако се не може упоредити са сложенијим системима за оркестрацију, *Docker Compose* омогућава ефикасну оркестрацију на једном рачунару. Његова основна сврха је аутоматизација процеса покретања, повезивања и управљања скупом контејнера кроз једноставне конфигурационе датотеке.

Због једноставности и флексибилности, често се користи у локалним развојним окружењима, али није намењен за продукциона окружења. Његова оркестрација је ограничена на један рачунар, што значи да не пружа високу доступност, балансирање оптерећења и друге функције које су потребне у раду продукционих окружења комплексних апликација.

Docker Compose оркестрација користи централизовану конфигурациону датотеку, најчешће референцирану као *docker-compose.yml*. Ова конфигурациона датотека пружа јасну и структурирану платформу у којој корисници могу дефинисати различите компоненте апликације: контејнере, хијерархију зависности између контејнера, параметре за мрежно повезивање, конфигурације за управљање волуменима, као и низ других параметара неопходних за правилно функционисање апликације.

Могућност да из конфигурационе датотеке генерише кохерентно окружење чини *Docker Compose* посебно корисним за развојне тимове. Омогућава инжењерима да лако реплицирају упрошћена продукциона окружења на локалним машинама, чиме се гарантује конзистентност развојних окружења у свим фазама развоја софтвера.

27.11.2 Multi-node оркестрација

Multi-node оркестрација представља процес управљања радом контејнерских апликација распоређених на више рачунарских чворова, који могу бити у облику виртуелних или физичких машина. У савременом пословном окружењу, где се

апликације морају скалирати и у реалном времену одговарати на повећан број корисничких захтева, *multi-node* оркестрација контејнера постаје неопходна. *Multi-node* оркестрација омогућава апликацијама да ефикасно користе ресурсе, прилагоде се различитим оптерећењима и опораве од потенцијалних отказа, одржавајући у исто време перформансе и висок степен доступности. Да би се све то реализовало, методе и алати *multi-node* оркестрације контејнера треба да дају одговоре на следеће технолошке и пословне изазове:

- **Комуникација између контејнера**. У мрежним дистрибуираним окружењима, комуникација између контејнера на различитим чворовима може проузроковати проблеме као што су латенција и губици у преносу. Ти проблеми се избегавају применом специјализованих мрежних протокола који су прилагођени раду у оркестрационим платформама, оптимизујући рутирање и брзину комуникације. Поред тога, *service-discovery* елиминише потребу за ручном конфигурацијом, што омогућава динамичко лоцирање контејнера.
- **Аутоматско скалирање**. При варијабилном оптерећењу, аутоматско скалирање представља основу за постизање конзистентних перформанси софтвера. Први проблем који оркестрациони системи морају решити је препознавање тренутка за додавање или одузимање ресурса. Оркестрациона решења нуде одговор применом софистицираних алгоритама и метрика које интелигентно прате оптерећење система, омогућавајући проактивно скалирање ресурса, без потребе за сталним људским надзором.
- **Отказ система**. Отказ система у дистрибуираном окружењу су свакодневица и њихове последице могу бити катастрофалне. Међутим, оркестрационе платформе пружају решење за тај скуп проблема кроз редундантне конфигурације и активне провере здравља контејнера. Такви механизми не само да детектују проблеме чим се појаве, већ и аутоматски пребацују операције на исправне контејнере, минимизујући прекиде.
- **Конзистентност и доступност података**. Конзистентност и доступност података може бити веома изазован проблем у дистрибуираним апликацијама. Овај проблем настаје због потребе да подаци буду синхронизовани на више места, упркос физичкој раздвојености контејнера. Технологије као што су дистрибуирани фајл системи и висококонзистентне базе података нуде решење, гарантујући конзистентност података и брз приступ без обзира на локацију.
- **Безбедност**. Безбедност у комуникацији између чворова је основна потреба сваке апликације која има приступ интернету. Оркестрационе платформе пружају решења кроз енкрипцију података у транзиту и аутентификацију приступа, гарантујући интегритет и приватност података. Изолација мреже контејнера и правила приступа додатно јачају одбрану штитећи контејнерске ресурсе од напада.
- **Балансирање оптерећења**. С обзиром на различите капацитете и оптерећења контејнера, балансирање оптерећења је кључно за одржавање перформанси система. Интелигентни балансери оптерећења, који користе напредне метрике и алгоритме, осигуравају да се оптерећење дистрибуира оптимално, чиме се постиже континуална ефикасност и брза реакција на промене у оптерећењу.

- ***CI/CD***. У процесу развоја апликација, испорука и имплементација нових верзија софтвера може довести до проблема попут прекида, несагласности између различитих верзија или проблема с међузависностима. Модерне оркестрационе платформе нуде решење кроз континуалну интеграцију и испоруку (*CI/CD*). Ове аутоматизоване методе омогућавају ажурирање апликација с минималним прекидима, уз гаранције да ће свака нова верзија проћи кроз ригорозно тестирање и валидацију пре него што буде имплементирана у продукционом окружењу. Употребом декларативних спецификација, систем аутоматски може управљати зависностима и конфигурацијама апликација, осигуравајући њихов исправан рад.
- **Платформе за оркестрацију контејнера**. Наведени елементи нису само додатак, већ есенцијалне компоненте сваке савремене *multi-node* оркестрационе платформе. Свака платформа мора укључивати решења за наведене проблеме како би се осигурала отпорност, ефикасност и скалабилност апликација којима управља. Најпознатије платформе за оркестрацију контејнера су: *Kubernetes*, *Docker Swarm*, *Mesos* и *Amazon ECS*. За сваку од ових платформи, имплементација решења за наведене проблеме је оно што их разликује и даје им карактеристично место на тржишту оркестрационих алата.

28. KUBERNETES

Kubernetes[637] је отворена платформа за аутоматизацију распореда, скалирања и управљања контејнеризованим апликацијама. Пружа робусне и скалабилне механизме за управљање радним оптерећењима, од монолитних апликација до микросервиса. Развијан је у *Гуглу*, а сада је под окриљем *Cloud Native Computing Foundation*.

Kubernetes је постао незаобилазан алат у *DevOps* процесу развоја софтвера и налази се у примерима најбоље праксе. Његова модуларна и флексибилна архитектура лако се интегрише с различитим услугама рачунарства у облаку, складиштима података и мрежним решењима.

Kubernetes се налази у средишту токова трансформације софтвера ка *cloud native* архитектурама. *Managed Kubernetes* услуге, као што су: *Google Kubernetes Engine* (*GKE*), *Amazon EKS* и *Azure Kubernetes Service* (*AKS*), омогућавају предузећима да једноставно имплементирају апликације и управљају *Kubernetes* кластерима, потпуно интегрисаним у екосистеме рачунарства у облаку, који су већ оптимизовани за високе перформансе и скалабилност.

28.1 Kubernetes архитектура

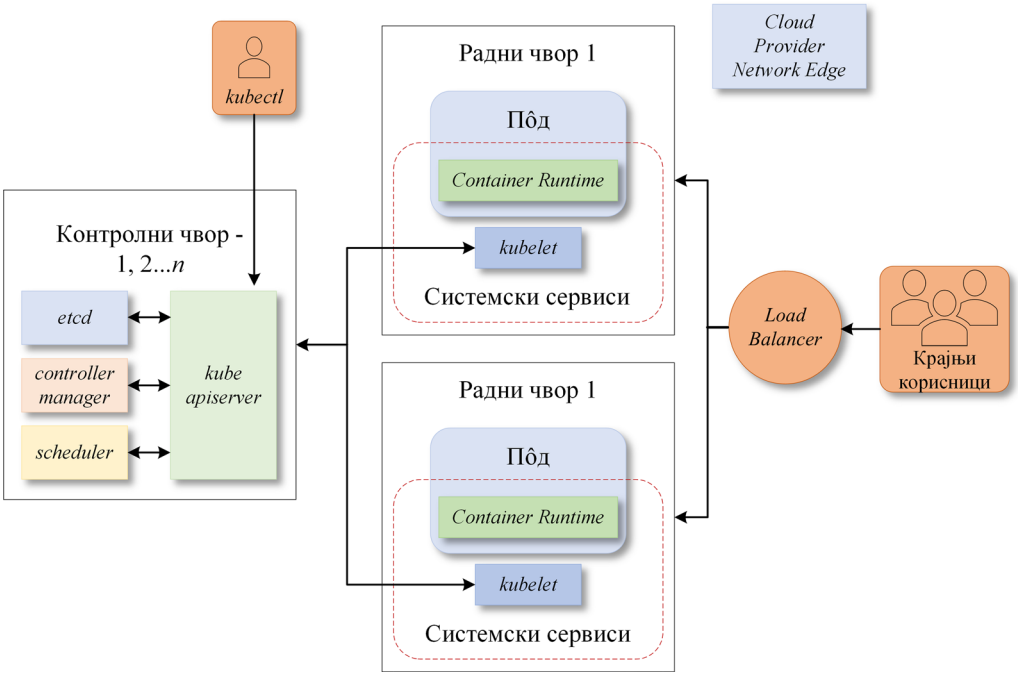
Архитектура *Kubernetes*-а представља кластер који се састоји од контролног чвора (*control plane*) и више радних чворова (*worker nodes*). Контролни чвор је

[637] https://kubernetes.io/

управљачки центар за све ресурсе и операције унутар *Kubernetes* кластера. Одговоран је за интерпретацију, извршење и оркестрацију корисничких захтева, као и за одржавање исправности и стабилности целокупног система. Радни чворови су извршне јединице које извршавају апликације у контејнерима. Они су задужени за пружање процесорских ресурса, меморије и мрежних ресурса. Контролни чвор се састоји од: *API* сервера, *etcd key-value* складишта, *Scheduler*-а и *Controller Manager*-а. Радни чворови су опремљени *Kubeletom*, *Kube-proxy*-јем и *container runtime*-ом, најчешће *containerd*[638]. Архитектура *Kubernetes* кластера приказана је на Слици 89.

28.1.1 API сервер

API сервер је централна тачка за операције и комуникације унутар *Kubernetes* кластера. Функционише као *RESTful API* посредник за размену података између клијената и кластера. *API* сервер аутентификује и ауторизује захтеве, проверава интегритет података, а затим усмерава операције ка одговарајућим компонентама, као што су *etcd* складиште или *Scheduler*, на основу њихове релевантности и приоритета.



Слика 89. Архитектура Kubernetes кластера[639]

[638] https://containerd.io/
[639] Прилагођено према: https://platform9.com/wp-content/uploads/2019/05/kubernetes-constructs-concepts-architecture.jpg

28.1.2 Etcd складиште

Etcd складиште[640] је *key-value* конфигурациона база података за *Kubernetes*. Ова компонента користи *Raft* алгоритам за одржавање високе доступности и толеранције на грешке. *Etcd* континуално синхронизује садржај и стања са *API* сервером и другим компонентама кластера, осигуравајући конзистентност и интегритет система кроз трансакционе механизме и логове, како би се увек знало тачно стање у ком се кластер налази.

28.1.3 Scheduler

Scheduler распоређује контејнере на радне чворове, анализирајући актуелне захтеве, капацитете радних чворова и остале расположиве ресурсе. Одлуку о распореду контејнера на радне чворове доноси на основу низа параметара, као што су захтеви за ресурсима, ограничења, антиафинитетне политике и друга задата правила. Након доношења одлуке, *Scheduler* ажурира *etcd* и *API* сервер, који потом инструише радне чворове да иницирају извршење контејнера.

28.1.4 Controller Manager

Controller Manager представља кључни елемент оркестрације унутар *Kubernetes* кластера, и задужен је за управљање животним циклусима ресурса и аутоматизацијом њиховог функционисања. Основна функционалност ове компоненте темељи се на контролним петљама у којима се анализирају актуелна стања ресурса у кластеру и пореде се са жељеним стањима, дефинисаним корисничким захтевима и конфигурацијама. Када *Controller Manager* идентификује било какав несклад, тада покреће одговарајуће акције. То могу бити алоцирање ресурса, премештање пôдова на друге радне чворове, или неки други поступци који су потребни за усклађивања стварног и жељеног стања. Овај процес се одвија у тесној координацији са *Scheduler*-ом и *API* сервером, осигуравајући висок степен аутоматизације и ефикасности унутар кластера.

28.1.5 Kubelet

Kubelet је агент, посредник између контролног чвора и радних чворова. Задужен је за покретање, заустављање и одржавање здравља контејнера, као и за надгледање ресурса и извештавање контролног чвора. *Kubelet* имплементира и механизме за аутоматско исправљање и решавање проблема.

[640] https://etcd.io/

28.1.6 Kube-proxy

Kube-proxy је компонента која управља мрежним саобраћајем апликационог нивоа на радним чворовима. Задужен је за усмеравање и балансирање мрежног саобраћаја између сервиса и пôдова. *Kube-proxy* користи низ механизама, као што су *iptables*, *IPVS* и *eBPF*, како би омогућио откривање сервиса, усмеравао мрежни саобраћај и примењивао алгоритме за балансирање оптерећења.

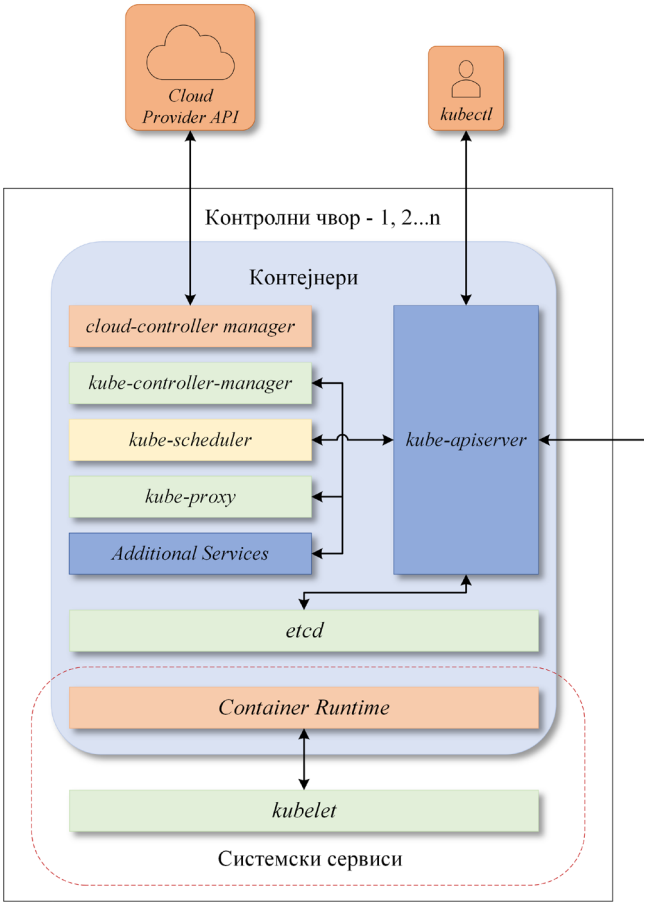
28.1.7 Container runtime

Container runtime је компонента која омогућава извршавање контејнера на радним чворовима. *Container Runtime Interface* (*CRI*) је *plugin* интерфејс који омогућава да *kubelet* користи различите *container runtime*-ове, без потребе да се рекомпајлирају и прилагођавају компоненте кластера. У овом контексту, *CRI* поставља скуп *API* спецификација који сваки *runtime* мора да испуни како би био компатибилан са *Kubernetes*-ом. *CRI* омогућава интероперабилност и стандардизацију, тако да се различите *runtime* технологије могу користити унутар истог *Kubernetes* екосистема.

28.1.8 Интеракција и синхронизација компоненти Kubernetes-а

Синхронизација и интеракција између компоненти кључне су за функционисање *Kubernetes*-а. *API* сервер служи као мост за комуникацију и синхронизацију између контролног чвора и радних чворова. Он прихвата захтеве од корисника, или интерне захтеве од других компоненти као што су *Scheduler* и *Controller Manager* и ажурира *etcd* новим планом. Ажуриране информације се пропагирају према *Kubelet*-у и *kube-proxy*-ју на радним чворовима. *Scheduler*, као интелигентни алокатор ресурса, константно комуницира са *API* сервером да би добио најновије податке о доступности и оптерећењу радних чворова. Истовремено, *Controller Manager,* који је одговоран за одржавање жељеног стања система, реагује на догађаје попут неуспеха у функционисању чвора или пôда, и врши комуникацију ових промена према *API* серверу.

Интеракција и синхронизација компоненти *Kubernetes*-а у контролном чвору приказана је на Слици 90.

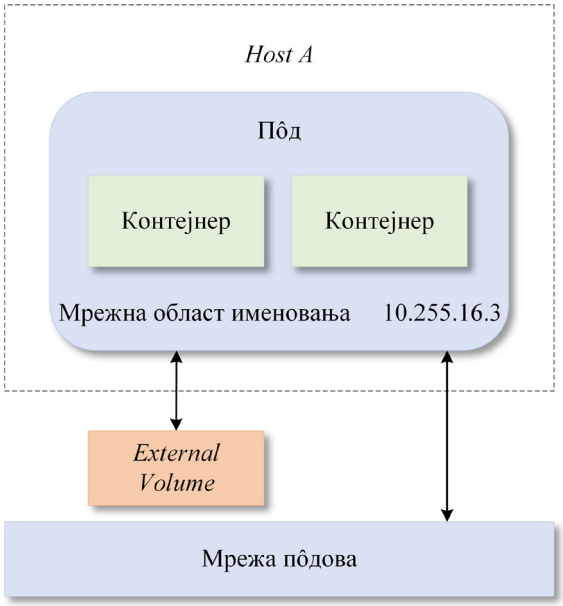


Слика 90. Интеракција и синхронизација компоненти Kubernetes-а у контролном чвору

Etcd складиште је извор података о стању кластера, омогућавајући кохерентност и интегритет кластера. Управљачке компоненте, као што су *API* сервер, *Scheduler* и *Controller Manager*, континуално ажурирају *etcd* информацијама о стању кластера. *Kubelet* и *kube-proxy* примењују конфигурације из *etcd* на радним чворовима, осигуравајући да су контејнери правилно иницијализовани, оркестрирани и надгледани. Ова симбиоза омогућава *Kubernetes*-у да ефикасно реагује на динамичке промене у захтевима окружења, док истовремено одржава високе стандарде перформанси и поузданости. Интеграција преко стања које се прати у *etcd*, омогућава не само скалабилност и отпорност на грешке, већ пружа и механизме за аутоматизовано управљање, ажурирање и опоравак система.

28.2 Пôд

Најмања извршна јединица која може бити распоређена унутар *Kubernetes* екосистема назива се пôд (*pod*). Пôд представља основни градивни блок апликације. Може садржати један или више контејнера који деле исте ресурсе и мрежну област именовања (Слика 91). Висок степен интеграције и интероперабилности омогућава оптимално коришћење ресурса и ефикасну комуникацију између контејнера у оквиру истог пôда. Пôдови на радним чворовима под директним су надзором *Kubelet*-а, а за управљање и распоређивање користе инструкције добијене преко *Kubernetes API* сервера.



Слика 91. Архитектура пôда

Пôд као концепт модуларности

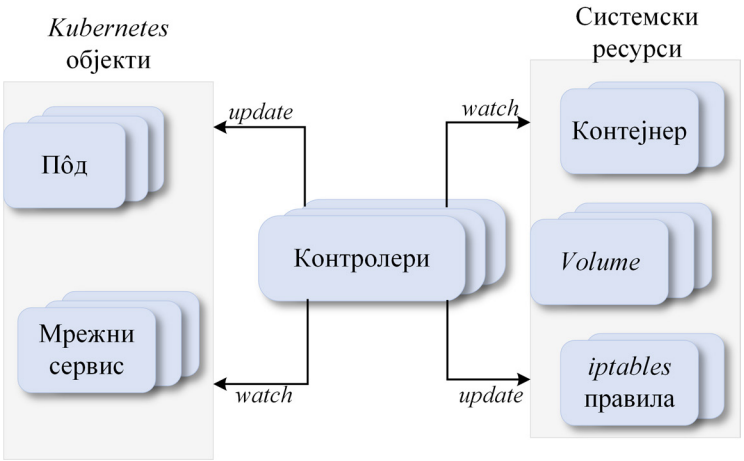
Пôд представља основни концепт модуларности у *Kubernetes*-у, еквивалентан модулу. Има сопствену област именовања, апликационе програмске интерфејсе који су видљиви унутар кластера, као и могућност аутоматске или мануелне миграције на други сервер. Пôд може енкапсулирати један или више контејнера (*Multi Container Pod*)[641], а тиме се омогућава гранулација модуларности у распону од једног микросервиса у оквиру једног контејнера по пôд-у, па све до сложених компоненти и подсистема, с више контејнера и микросервиса по пôд-у. Подела надлежности и кохезија контејнера унутар пôд-а може се имплементирати применом *sidecar*

[641] https://linchpiner.github.io/k8s-multi-container-pods.html

патерна[642], тако што се функционалности пôда поделе између главног контејнера који садржи апликациону логику и више *sidecar* контејнера који функционишу паралелно, обављајући споредне задатке као што су мониторинг, логовање, или управљање конфигурацијама. Овај начин рада омогућава да се главни контејнер фокусира на своје примарне функције без потребе за уграђивањем додатних споредних функционалности. Најчешће примене *sidecar* контејнера укључују подешавање проксија и агената за сигурност који аутоматизују задатке као што су енкрипција мрежног саобраћаја и управљање *API* захтевима у оквиру *Service Mesh*-а.

28.3 Контролери

Контролери имају кључну улогу у аутоматизацији различитих аспеката управљања стањем кластера. Контролери континуално прате стање кластера и усклађују га са жељеним стањем које је дефинисано преко *API* сервера. Они се ослањају на *etcd* за чување информација о стању и користе га као извор података приликом одлучивања о акцијама које треба предузети. На тај начин, контролери омогућавају *Kubernetes*-у да аутоматски реагује на промене и да одржава здравље кластера. Контролери у *Kubernetes*-у приказани су на Слици 92 и објашњени у наредном тексту.



Слика 92. Контролери у Kubernetes-у

28.3.1 ReplicaSet

ReplicaSet је специјализовани тип контролера који има за циљ да одржава жељени број реплика пôда унутар кластера. У случају да неки од пôдова из било којег разлога престане да функционише, *ReplicaSet* аутоматски иницира креирање

[642] https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/

новог пôда, чиме се одржава константан број активних пôдова. Ова компонента комуницира са *API* сервером ради ажурирања информација о стању и с *Kubeletom* на радним чворовима за извршење операција распоређивања.

28.3.2 Deployment Controller

Deployment Controller је део *Kubernetes* екосистема задужен за континуалну испоруку и континуално ажурирање апликација. *Deployment Controller* омогућава подешавање распоређивања и ажурирања пôдова, користећи патерне као што је *rolling updates* за постепено увођење промена, као и могућност за *rollback* ако се детектују проблеми. Осим тога, *Deployment Controller* пружа сложене механизме за контролу пôдова, укључујући стратегије за ажурирање и правила за скалирање. Овако сложена функционалност омогућава висок ниво контроле и прилагодљивости променама, док истовремено минимизује потребу за мануелним интервенцијама администратора. У комбинацији са *ReplicaSet*-ом и *API* сервером, *Deployment* је кључни елемент у оркестрацији и динамици кластера, омогућујући агилност и отпорност на отказе у динамичким, дистрибуираним окружењима.

28.3.3 StatefulSet

За управљање *stateful* апликацијама, користи се *StatefulSet* контролер[643]. За разлику од *Deployment*-а, који је идеалан за *stateless* апликације, *StatefulSet* омогућава сваком пôду да задржи своје стање при коришћењу дистрибуираних система који имају латенцију у пропагацији података. Ово је од суштинског значаја за управљање ресурсима као што су базе података, где сваки пôд треба да има конзистентно стање, као и редослед при креирању и брисању. *StatefulSet* користи *etcd* као перзистентну базу података за чување информација о стању, док *Kubelet* на радним чворовима води бригу о примени ових информација на пôдове.

StatefulSets се често користи у комбинацији с *headless* сервисима који омогућавају сваком пôду у класетру да трајно задржи своју мрежну адресу. Тиме се омогућава одржавање конзистентности у управљању подацима унутар пôда и стабилне идентификације ресурса, која се не мења у току миграције пôда, што је критично за базе података, њихове репликације и трансакционе операције[644].

28.3.4 Перзистентно складиштење података у Kubernetes-у

Складишни волумени у *Kubernetes*-у могу бити ефемерни и перзистентни[645]. За меморисање и чување стања апликације између два покретања (*stateful*), користе се перзистентни волумени (*PersistentVolume – PV*). Перзистентни волумен трајно

[643] https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/
[644] https://kodekloud.com/blog/kubernetes-headless-service/
[645] https://kubernetes.io/docs/concepts/storage/volumes/

складишти податаке на неком од физичких или мрежних медијума. Подаци у перзистентном волумену се чувају све време животног циклуса пôда коме је перзистентни волумен придружен, све док се пôд трајно не уклони из кластера.

Перзистентни складишни простор може бити организован као локални фајл систем, мрежни сервис, као што је *NFS*, или складишни простор који пружају провајдери рачунарства у облаку, као што су *Amazon Web Services*, *Google Cloud Platform* или *Microsoft Azure*[646]. Складишни простор у *Kubernetes*-у се разврстава по класама складишта (*storage classes*), на основу перформанси, пакета услуга које нуде провајдери рачунарства у облаку, карактеристикама складишног медијума, као што су нпр. *SSD* или *HDD*, и цени.

Захтев за доделу складишних ресурса за трајни волумен назива се: *PersistentVolumeClaim* (PVC)[647]. *PVC* спецификује величину и тип складишног простора који је потребан, а *Kubernetes*-ов систем за управљање складиштењем (*PersistentVolumeProvisioner*)[648] стара се о проналажењу одговарајуће класе складишта која може задовољити ове захтеве. Процес додељивања перманентних волумена се може аутоматизовати, учинити динамичким и извршавати у реалном времену.

Складишни простор се додељује у зависности од захтева за скалабилношћу, перформансама или цене. Тиме се предузећима омогућава да оптимизују трошкове и перформансе складишног простора према специфичним потребама апликација. Динамичко додељивање складишног простора (*dynamic provisioning*) смањује административне напоре везане за управљање складишним простором и олакшава скалирање апликација пружајући им потребне ресурсе складиштења по потреби.

Динамичко додељивање складишног простора интегрисано је са *Container Storage Interface* (*CSI*), чиме је олакшана и стандардизована интеракција контејнера с различитим класама складишта. *CSI* стандардизује комуникацију између *Kubernetes* кластера и складишних медијума, чиме се корисницима омогућава да искористе најбоље од специфичних карактеристика и предности које нуди сваки појединачни провајдер складишта.

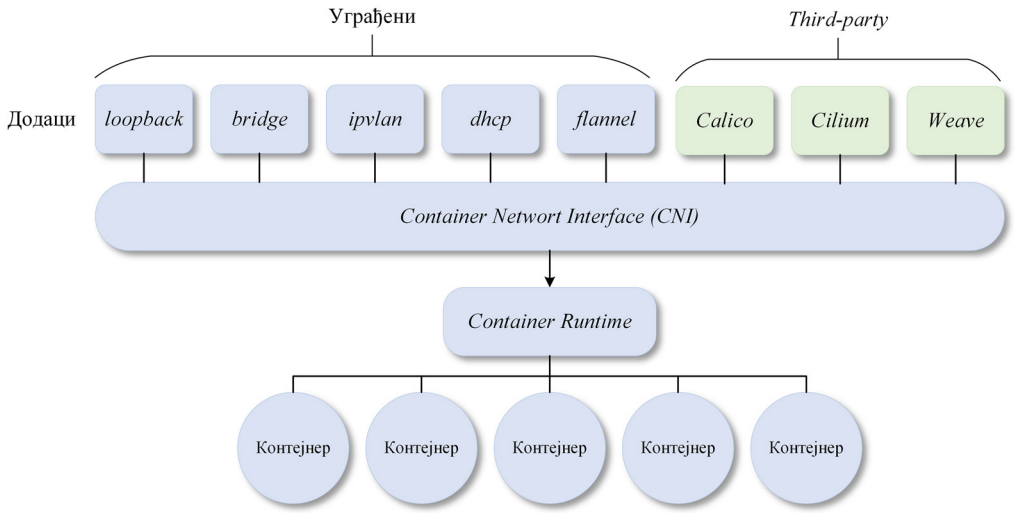
У складишном екосистему, *StatefulSet*-ови имају значајну улогу у управљању *stateful* апликацијама. *StatefulSet*-ови користе механизме динамичке доделе складишног простора и *CSI* за доделу и одржавање перманентног складишног простора за сваки пôд унутар сета. Сваком пôду динамички се додељује посебан перманентни волумен који остаје константно везан за тај пôд, чак и у случају његовог премештања или поновног покретања унутар кластера. Тиме се омогућује да апликације имају непрекидан приступ потребним подацима.

[646] https://kubernetes-csi.github.io/docs/drivers.html
[647] https://kubernetes.io/docs/concepts/storage/persistent-volumes/
[648] https://github.com/kubernetes/examples/blob/master/staging/persistent-volume-provisioning/README.md

28.4 Умрежавање у Kubernetes-у

Умрежавање између пôдова у кластеру, који се могу физички налазити на веома удаљеним серверима, реализује се преко неког од облика софтверских мрежа. Комуникација се успоставља коришћењем интерфејса који су дефинисани и реализовани коришћењем *Container Network Interface* (*CNI*)[649] спецификације са скупом одговарајућих мрежних компоненти и библиотека, прилагођених раду с контејнерима. Структура *CNI* је дата на Слици 93.

У већини случајева, поред основних мрежних компоненти (*loopback*, *bridge*, *dhcp*), *CNI* користи *Flannel*[650], који сав саобраћај енкапсулира у *VXLAN* пакете, преносиве путем интернета. За комплексније случајеве, када су мреже унутар кластера веома сложене, користи се *Calico*[651].



Слика 93. Container Network Interface (CNI)

Унутар пôда, контејнери могу комуницирати без икаквих ограничења, користећи *loopback* интерфејс. Пôдови могу међусобно комуницирати преко *IP* адреса пôда које су видљиве унутар кластера. Приступ од пôда ка сервису, или екстерног извора ка сервису, реализује се посредовањем *kube-proxy* компоненте.

CNI дефинише умрежавање на *L2* и *L3* нивоу *TCP/IP* модела. За умрежавање на апликационом нивоу, *Kubernetes* користи концепт који се назива *Services*.

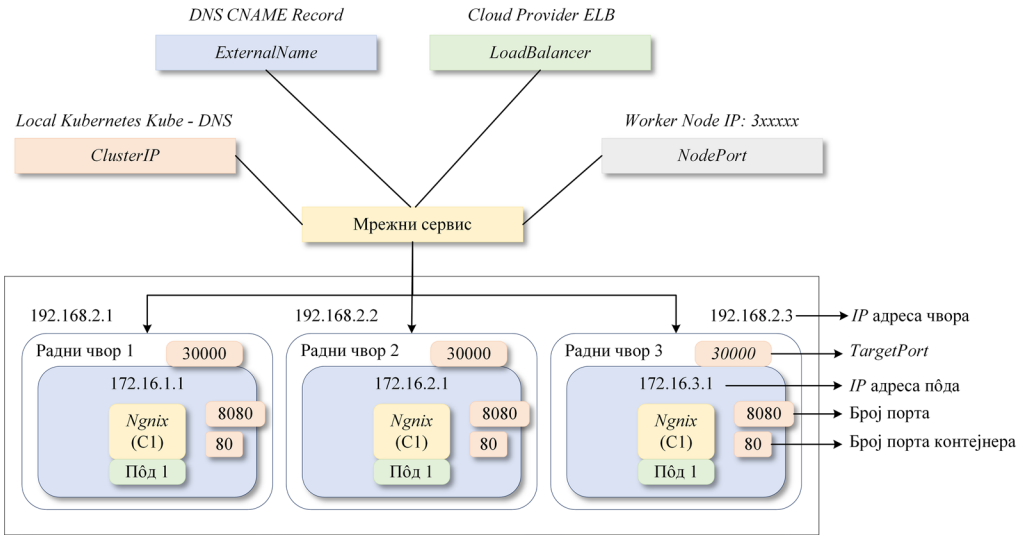
[649] https://www.cncf.io/projects/container-network-interface-cni/
[650] https://github.com/flannel-io/flannel
[651] https://docs.tigera.io/

28.4.1 Kubernetes мрежни сервиси апликационог нивоа

У *Kubernetes* екосистему, управљање мрежним саобраћајем и ресурсима на апликационом нивоу, како унутар кластера, тако и изван њега, реализује се применом концепта под називом *Services*. Ови мрежни сервиси не само да дефинишу комплексан скуп правила за интелигентно усмеравање саобраћаја ка одређеним пôдовима, већ омогућавају и високу доступност и скалабилност апликација преко система балансирања оптерећења. Ова функционална улога је интегрисана са осталим компонентама у *Kubernetes* архитектури.

Сервиси су ограничени на апликациони ниво *ISO*/*OSI* модела. Сервисима, као што су *http*, *dns*, *proxy* и балансери оптерећења, и одговарајућим *API*, приступа се на основу *IP* адресе и *Port*-а. Сервиси се могу претраживати, лоцирати и њима приступати претраживањем додатних текстуалних *DNS* записа о сервису, које одржава *Kubernetes*.

Архитектура мрежних сервиса приказана је на Слици 94.



Слика 94. Мрежни сервиси Kubernetes-а

Мрежни сервиси се описују атрибутима:

- ***ClusterIP*** излаже сервис интерној мрежи кластера преко *IP* адресе. Сервис је доступан само унутар кластера.
- ***NodePort*** излаже сервис на специфичном порту *IP* адресе радног чвора. Тиме се даје могућност постављања екстерних балансера оптерећења или сличних сервиса који нису потпуно подржани *Kubernetes*-ом.
- ***LoadBalancer*** излаже сервис екстерном *LoadBalancer*-у, којег обезбеђује *managed-Kubernetes* добављач.
- ***ExternalName*** само мапира *CNAME* слог у *DNS*-у, без икакве врсте посредовања.

Атрибути мрежних сервиса, поред спецификације конкретних особина сервиса, као додатни параметри уграђени у *DNS* записе, користе се за аутоматско проналажење сервиса.

Сервис се дефинише спецификацијом вредности атрибута и навођењем свих чворова којима сервис приступа.

Намена Сервиса као концепта у *Kubernetes*-у је да групише скуп приступних тачака *TCP*/*IP* мрежних сервиса апликационог нивоа на пôдовима у један скалабилан и поуздан *TCP*/*IP* ресурс апликационог нивоа, с једном стабилном *ClusterIP* адресом.

Функционисање сервиса се одвија на следећи начин: Клијенти шаљу захтеве ка *ClusterIP* адреси *LoadBalancer*-а. *LoadBalancer* прослеђује захтеве клијената на једну или више адреса чворова, који су спецификовани у оквиру сервиса (*IP* адресом чвора и *NodePort*-ом). *LoadBalancer* може у реалном времену исте послове распоређивати различитим чворовима, а да клијент тога не мора да буде свестан. Тиме се постиже повећање склабилности и поузданости сервиса.

Kube-proxy је компонента на сваком радном чвору која је задужена за имплементацију усмеравања мрежног саобраћаја на апликационом нивоу за тај чвор. *Kube-proxy* прима инструкције и конфигурације директно од *API* сервера и користи *etcd* као основни извор поузданих информација о актуелним *Kubernetes* мрежним сервисима који садрже правила за рутирање и балансирање оптерећења. Интеракција мрежних сервиса са осталим кључним компонентама *Kubernetes* екосистема чини *Kubernetes* мрежне сервисе изузетно робустним и прилагодљивим механизмом за контролу мрежног саобраћаја апликационог нивоа.

У *Kubernetes*-у, *Deployment* контролери користе мрежне сервисе као елемент за управљање и оркестрацију саобраћаја између пôдова. Сервиси користе селекторе који омогућавају аутоматско идентификовање и усмеравање саобраћаја апликационог нивоа ка пôдовима који припадају одређеном *Deployment*-у. На овај начин се омогућава приступ неком *Deployment*-у путем једне локално видљиве адресе, и балансира оптерећење за сав саобраћај који је усмерен на ту адресу. Ова функционалност је неопходна за динамичко скалирање и управљање апликацијама у току извршавања. На тај начин, сервиси омогућавају континуалан и конзистентан приступ пôдовима, чак и кад пролазе кроз фазу ажурирања или замене.

28.4.2 Kubernetes Ingress ресурси и мрежни сервиси

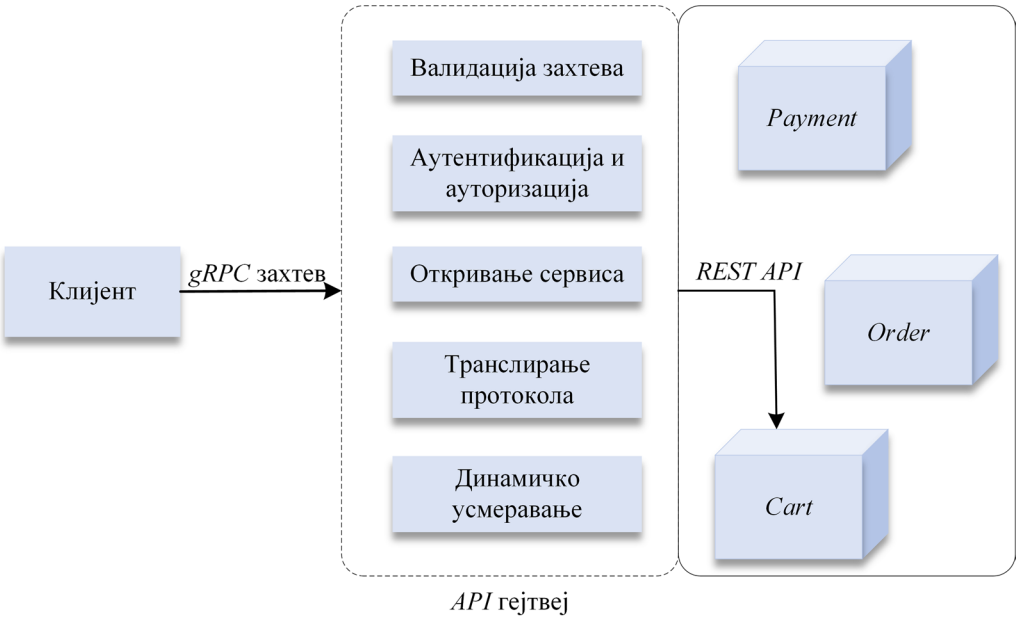
Kubernetes Ingress ресурси и мрежни сервиси, или само *Kubernetes Ingress*[652], функционишу као централна тачка за прихватање долазних *http* и *https* захтева из окружења, које обрађују, трансформишу и усмеравају ка одговарајућим сервисима унутар кластера.

[652] https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

Kubernetes Ingress мрежни сервис своје функционалности реализује преко *Kubernetes Ingress* контролера, који представља специфичну имплементацију инверзног *Proxy* сервера, *API Proxy*, или *API* гејтвеја, који се комбинују с балансерима оптерећења и прилагођавају за рад у *Kubernetes* окружењу.

Kubernetes Ingress контролер се не инсталира као подразумевани део *Kubernetes* кластера, већ се инсталира по потреби, и прилагођава конкретним апликацијама на кластеру. У оквиру кластера, зависно од потреба апликација, може бити дефинисано више *Kubernetes Ingress* контолера различитих добављача. *Kubernetes* има уграђену подршку за *AWS*[653], *GCE*[654], и *nginx*[655]. У *Kubernetes* кластер се могу уградити и контролери других произвођача софтвера као што су *Kong Ingress Controller for Kubernetes*[656], *AKS Application Gateway Ingress Controller*[657] и други.

Потребно је напоменути и да се конфигурисање *Ingress* контролера врши у оквиру *Ingress* ресурса, синтаксом која је дефинисана у *Kubernetes*-у и која је иста за све контролере. *Kubernetes* преводи *Ingress* конфигурацију у конфигурацију одабраног софтвера за контролере, па корисник о томе не мора да брине. Пример обраде захтева на *API* гејтвеју у оквиру *Kubernetes Ingress* контролера дат је на Слици 95.



Слика 95. API гејтвеј у оквиру Kubernetes кластера

[653] https://github.com/kubernetes-sigs/aws-load-balancer-controller#readme
[654] https://github.com/kubernetes/ingress-gce/blob/master/README.md#readme
[655] https://github.com/kubernetes/ingress-nginx/blob/main/README.md#readme
[656] https://github.com/Kong/kubernetes-ingress-controller#readme
[657] https://github.com/Azure/application-gateway-kubernetes-ingress?tab=readme-ov-file#application-gateway-ingress-controller

Фазе обраде захтева кроз *API* гејтвеј на *Kubernetes* кластеру су:

1. **Валидација захтева**: *API* гејтвеј проверава валидност захтева тако што контролише методе, заглавље и тело захтева, како би се обезбедило да захтев одговара унапред дефинисаним правилима у *Kubernetes*-у.
2. **Аутентификација и ауторизација** (*AuthN/Z*): Гејтвеј аутентификује и ауторизује *API* захтев провером идентификационих података, као што су *API* кључеви (аутентификација), а затим примењује методе ауторизације, као што је *RBAC* (контрола приступа на основу улога). *API* гејтвеј може користити услуге провајдера идентитета (*IdP*) за комлексније провере аутентификације и ауторизације, као што је мултифакторска аутентификација (*MFA*) и друге.
3. **Откривање сервиса:** *API* гејтвеј користи компоненту откривања сервиса како би пронашао одговарајући *back-end* сервис за примљени захтев. То ради путем упита над регистром сервиса или коришћењем динамичког *DNS*-а.
4. **Транслирање протокола:** Ако је потребно, *API* гејтвеј може преводити протоколе између клијената и сервиса. На Слици 95 је приказан клијент који шаље *gRPC* захтев који се преводи у *REST* формат. Такође, гејтвеј претвара одговор у протокол видљив споља (овде из *REST* у *gRPC* формат) пре враћања одговора клијенту.
5. **Динамичко усмеравање**: Када је *back-end* сервис лоциран и протокол захтева преведен, *API* гејтвеј рутира захтев ка одговарајућој инстанци *back-end* сервиса унутар кластера, или *load balancer*-у.

Конфигурисање *Ingress* контролера врши се у оквиру *Ingress* ресурса. Може се конфигурисати сложено усмеравање долазних захтева на основу домена и путања, *SSL*/*TLS* терминација за безбедну комуникацију, редирекција, и *layer*-7[658] балансирање оптерећења, чиме се унапређује флексибилност и контрола мрежног саобраћаја.

Интеграција *Ingress*-а са алатима за мониторинг и аналитику представља додатну погодност у управљању апликацијама. Синергија *Ingress*-а са алатима за мониторинг и аналитику омогућава администраторима кластера да добију увид у обрасце долазног и интерног мрежног саобраћаја, детектују потенцијалне сигурносне ризике и правовремено реагују на промене у понашању апликација или оптерећењу система. Мониторинг приступа систему преко *Ingress*-а може помоћи у оптимизацији перформанси и доступности апликација, нудећи податке који могу водити ка бољем разумевању и унапређењу корисничког искуства.

28.5 Функционалности Kubernetes кластера

Компоненте у архитектури *Kubernetes* система функционишу у координисаном екосистему ради постизања високе отпорности на отказе, скалабилности и

[658] https://www.f5.com/glossary/layer-7-load-balancing

ефикасности. *API* сервер посредује у комуникацији и координацији између компоненти, док *etcd* складиште гарантује доследност и трајност информација. *Kubelet* и *kube-proxy* компонентама, које се налазе на радним чворовима, управља се из контролног чвора, и у комбинацији са *Scheduler*-ом и *Controller Manager*-ом, оптимизује се животни циклус пôдова и апликација.

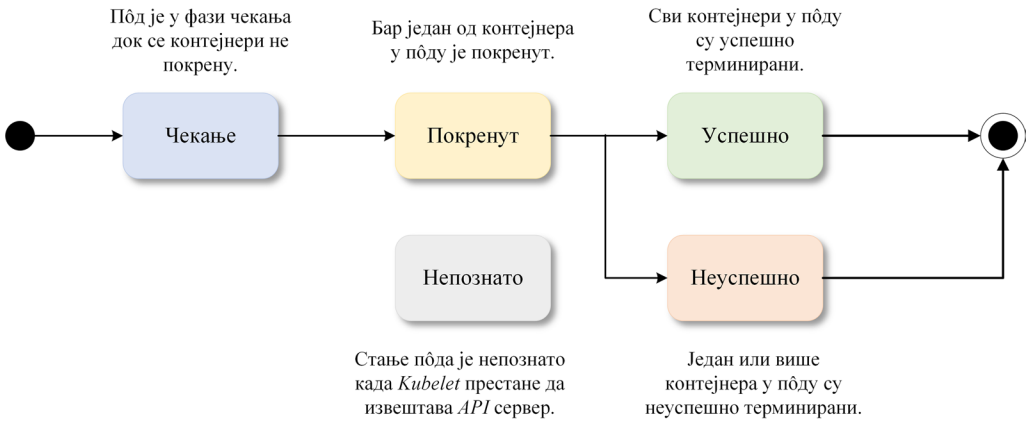
Ова међузависност и синхронизована координација кључни су елементи који *Kubernetes* платформу чине погодном за управљање сложеним апликативним екосистемима. Управљање се реализује преко скупа функционалности кластера, који су приказани на Слици 96.

Слика 96. Функционалности Kubernetes кластера

28.5.1 Генерисање и распоређивање пôдова

Генерисање и распоређивање пôдова у *Kubernetes* кластеру је важна функционалност, кључна за перформансе и отпорност на отказе целокупног система. Она укључује сарадњу између компоненти кластера, као што су *API* сервер, *Scheduler* и *Kubelet*, у циљу оптималног искоришћења ресурса и постизања високе доступности и ефикасности на свим нивоима. Када корисник иницира захтев за генерисање новог пôда, *API* сервер најпре врши валидацију тог захтева и ажурира *etcd* складиште подацима о жељеном стању пôда. Ова фаза поставља основу за даље операције и координацију унутар кластера. *Scheduler* потом преузима и анализира различите

факторе који утичу на распоређивање пôдова, као што су тренутно оптерећење радних чворова, доступност ресурса и поставке афинитета, те на основу тога доноси одлуку о томе на којем ће се радном чвору пôд покренути. Након што је одлука донета, *API* сервер ажурира *etcd* складиште новим информацијама о локацији пôда. Затим, *Kubelet* на изабраном радном чвору почиње иницијализацију пôда, користећи одабрани *container runtime* за постављање и покретање контејнера, чиме се финализује циклус креирања и распоређивања пôда. Животни циклус пôда дат је на Слици 97.



Слика 97. Животни циклус пôда

28.5.2 Балансирање оптерећења у оквиру кластера

Балансирање оптерећења унутар *Kubernetes* кластера је функционалност задужена за високу доступност и перформансе. За ову функционалност се користе *Kubernetes* мрежни сервиси и *kube-proxy*. *Kubernetes* мрежни сервиси омогућавају апстракцију и стабилан мрежни интерфејс апликационог нивоа за подове, док *kube-proxy* динамички ажурира своје конфигурације у складу с променама у мрежним сервисима и пôдовима. Када *API* сервер региструје промене, информације се брзо пропагирају до *kube-proxy*, који потом конфигурише или ажурира *iptables* или *IPVS* правила. Ови механизми ефикасно усмеравају саобраћај ка одговарајућим пôдовима, који су изложени путем *Kubernetes* мрежних сервиса. Ова координирана интеракција између *Kubernetes* мрежних сервиса, *kube-proxy*-ја и *API* сервера осигурава оптималну дистрибуцију саобраћаја, минимизујући потенцијалне тачке застоја и максимизујући искоришћење ресурса унутар кластера.

28.5.3 Скалирање перформанси

Скалирање перформанси у *Kubernetes*-у је функционалност важна за одржавање високих перформанси и ефикасности. Управља се преко координисаних акција *Controller Manager*-а, *Scheduler*-а и *API* сервера. Конкретно, *Controller Manager*,

користећи компоненте као што су *ReplicaSet* и *Deployment* контролери, динамички управља бројем пôдова, аутоматски додајући или уклањајући инстанце у складу с тренутним потребама и конфигурацијом. Ова аутоматска прилагодљивост уско је повезана с радом *Scheduler*-а, који је задужен за селективно постављање нових пôдова на најадекватније радне чворове. *API* сервер, као централна тачка комуникације, осигурава доследност и ажурност информација, синхронизујући ове промене са *etcd* складиштем које служи као основни извор података о стању кластера.

28.5.4 Обрада непланираних прекида и опоравак

Обрада непланираних прекида у раду пôдова је компонента задужена за одржавање отпорности на отказе и високе доступности *Kubernetes* кластера. Када *Controller Manager* детектује непланирани прекид у раду пôда, одмах покреће поступак за његову замену или опоравак. Ово је посебно важно ако је пôд део неке сложеније структуре као што су *ReplicaSet* или *Deployment*, где се аутоматски креира нови пôд на радном чвору који *Scheduler* идентификује као најпогоднији. *Kubelet* је задужен за иницијализацију новог пôда, док *kube-proxy* осигурава његову мрежну доступност, чиме се обезбеђује несметан рад и одржавање кластера. Сви ови процеси су блиско интегрисани и синхронизовани са *etcd* складиштем преко *API* сервера, чиме се осигурава кохерентно и поуздано стање кластера.

28.5.5 Мониторинг и логови

Kubernetes нема уграђене алате за мониторинг и логове. Корисници могу сами да одаберу и имплементирају екстерне алате који одговарају специфичним потребама њихових апликација. Тиме се омогућава прилагођавање праћења перформанси различитим типовима апликација. Лоша страна је то што имплементација ових система захтева додатну конфигурацију и одржавање.

Мониторинг у *Kubernetes*-у, користи се за одржавање исправног рада кластера и апликација. Прикупљају се метрике у реалном времену које обухватају искоришћеност *CPU*, употребу меморије, мрежни проток, латенцију и друге. Алати који се у ту сврху користе, као што је нпр. *Prometheus*[659], омогућавају детаљно праћење ових метрика, нудећи увид у перформансе индивидуалних апликација. Једна од кључних вредности мониторинга је његова употреба у аутоматском скалирању апликација. *Kubernetes* се може конфигурисати тако да аутоматски покреће додатне инстанце пôдова када детектује оптерећење на постојећим инстанцама.

За разлику од мониторинга, који је фокусиран на прикупљање метрика, *loging* се у *Kubernetes*-у бави бележењем и чувањем информација о догађајима и

[659] https://prometheus.io/

активностима унутар кластера. Имплементација ефикасног система за *loging* омогућава корисницима да анализирају и разумеју како се апликације понашају у продукционом окружењу, што може бити од помоћи у отклањању проблема. Интеграција са екстерним алатима попут *Elasticsearch*[660], *Fluentd*[661] и *Kibana* (*EFK stack*)[662] омогућава централизовано прикупљање, претраживање и анализу логова из различитих извора унутар кластера.

28.5.6 Аутоматско скалирање

Аутоматско скалирање у *Kubernetes*-у (*autoscaling workloads*)[663] омогућава прилагођавање апликација на променљиве услове оптерећења, путем аутоматског прилагођавања броја пôдова или ресурса додељеним инстанцама пôдова.

Аутоматско скалирање се у *Kubernetes*-у реализује применом *Horizontal Pod Autoscaler*-а (*HPA*)[664] и *Vertical Pod Autoscaler*-а (*VPA*)[665], који омогућавају динамичко скалирање ресурса на основу стварне потрошње и захтева апликација, уз могућност интеграције са сервисима за мониторинг или метрикама које нуде *cloud* провајдери.

Horizontal Pod Autoscaler (*HPA*) динамички мења број реплика пôда у складу с мерењима ангажовања *CPU* и меморије који врше екстерни системи за мониторинг. Екстерни системи за мониторинг прате и снимају метрике, а *HPA* их користи за одлучивање о потреби за скалирањем броја пôдова, на више или ниже, на основу дефинисаних прагова ангажовања ресурса или других критеријума. На пример, ако су пôдови конфигурисани да одржавају одређену границу искоришћености *CPU*, у случају да је оптерећење испод прага, тада *HPA* повећава број пôдова на серверу, а када се искоришћеност *CPU* повећа изнад задатог прага, број пôдова на серверу се смањује. Интеграцијом са *cloud autoscaling* сервисима и применом *cloud*-специфичних метрика може се додатно повећати флексибилност аутоскалирања у *Kubernetes*-у.

За разлику од *HPA* који оптимизује број пôдова по серверу, *Vertical Pod Autoscaler* врши финија подешавања прилагођавањем додељених ресурса које појединачни пôдови користе. *VPA* редовно процењује употребу ресурса од стране пôдова и предлаже или аутоматски прилагођава ресурсе који су им додељени, како би се омогућило пôдовима да ефикасно раде унутар својих оперативних капацитета.

Неправилна конфигурација *autoscaling*-а може довести до непотребног скалирања које оптерећује сервере и повећава оперативне трошкове, или до недовољно додељених ресурса који негативно утичу на перформансе апликација. Стога,

[660] https://www.elastic.co/elasticsearch
[661] https://www.fluentd.org/
[662] https://www.cloud4c.com/blogs/efk-logging-with-efk-stack
[663] https://kubernetes.io/docs/concepts/workloads/autoscaling/
[664] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[665] https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler

кључна је прецизна конфигурација и интеграција с поузданим мониторинг системима како би се одржала ефикасност *autoscaling*-а. Интеграција са сервисима *cloud* провајдера, користећи специфичне метрике, додатно може унапредити флексибилност и прецизност *autoscaling*-а, што апликацијама омогућава да максимизују коришћење *cloud* ресурса и минимизују оперативне трошкове у *cloud* окружењима.

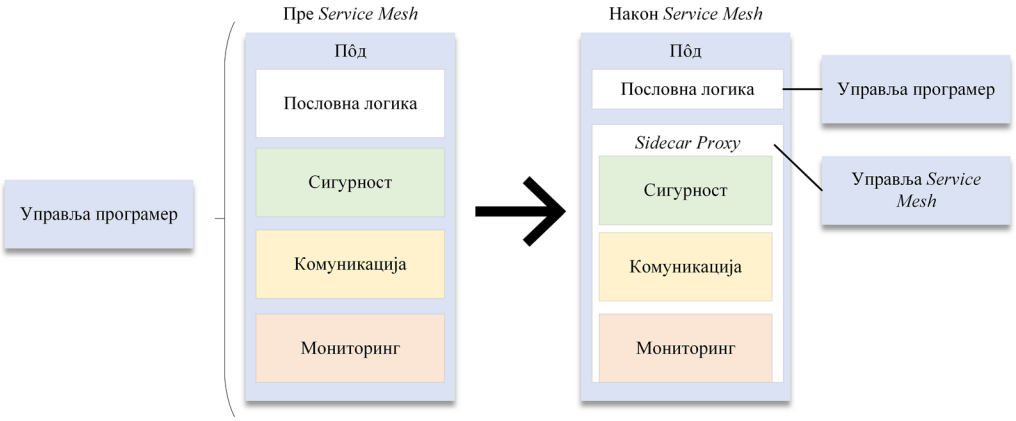
28.5.7 Service Mesh у Kubernetes-у

Аутоматско скалирање у *Kubernetes*-у врши оптимизацију ангажовања ресурса на нивоу сервера и пôда. Расподела оптерећења се тада реализује помоћу *Kube-proxy* сервиса, који на нивоу радних чворова расподељује оптерећење по пôдовима. Имајући у виду да пôд представља имплементацију једног модула апликације, може се закључити да *Kubernetes* у основи подржава аутоматско скалирање и расподелу оптерећења на нивоу модула.

Међутим, проблем може настати у случајевима када су модули, тј пôдови, организовани у више сервиса, тј. контејнера, а потребно је скалирање на нивоу једног сервиса унутар модула. Оваква функционалност није уграђена у *Kubernetes* и не може се постићи применом *Kube-proxy* сервиса. Да би се то постигло, у *Kubernetes*-у се дефинише наменски инфраструктурни слој, премештањем послова управљања мрежним саобраћајем, аутентификације, ауторизације, сигурности, поузданости, обезбеђења скалабилности и других споредних послова, из апликационог слоја микросервиса у посебне *sidecar*[666] контејнере. Сваком пôду се додељује посебан *Layer-7 Proxy* сервер који управља *sidecar* контејнерима унутар пôда. *Layer-7 Proxy*[667] сервери су међусобно умрежени. Мрежа *Proxy* сервера заједно sa *sidecar* контејнерима чини нови инфраструктурни слој који се назива Мрежа сервиса или *Service Mesh*[668], преко којег се може управљати перформансама и мрежним саобраћајем појединачних микросервиса, независно од апликационе логике (Слика 98).

Балансирање оптерећења унутар мреже сервиса пружа веће могућности у односу на стандардни *Kubernetes* сервис за балансирање оптерећења. Док *Kubernetes* примарно користи једноставне алгоритме за расподелу долазних захтева међу пôдовима, софтвер за мреже сервиса омогућавају сложеније *Layer-7* стратегије рутирања засноване на садржају захтева, тежинама рута и конфигурабилним правилима за покушаје и прекиде. Ово омогућава управљање на вишем нивоу грануларности и динамичније управљање саобраћајем, што је посебно корисно у сложеним микросервисним архитектурама.

[666] https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/
[667] https://www.f5.com/glossary/layer-7-load-balancing
[668] https://imesh.ai/blog/what-is-a-service-mesh-and-why-is-it-needed-for-kubernetes/



Слика 98. Трансформација микросервиса преко sidecar контејнера у Service Mesh[669]

Мониторинг и аналитика унутар мреже сервиса нуде детаљне увиде у перформансе, латенцију, и грешке специфичне за појединачне захтеве и токове података међу сервисима. Анализа и визуелизација омогућавају детаљну анализу мрежног саобраћаја, боље разумевање понашања система, на нивоу који превазилази стандардне алате за мониторинг у *Kubernetes*-у.

Имплементација мреже сервиса у *Kubernetes* окружењу може значајно побољшати управљање микросервисима, поједностављујући конфигурацију мрежних сервиса, унапредити сигурност и олакшати опсервабилност и анализу система. Кроз правилну имплементацију и конфигурацију, мрежа сервиса постаје незамењиви део архитектуре за апликације које захтевају комплексне мрежне операције и висок ниво сигурности.

Софтвер за управљање мрежама сервиса није део *Kubernetes*-а. У ту сврху најчешће се користе специјализовани софтверски пакети као што су: *Istio*[670], *Consul Connect*[671], *Linkerd*[672], *Envoy*[673].

28.5.8 Безбедност

Безбедност пôдова је интегрисана на више нивоа, и представља један од важних аспеката у управљању *Kubernetes* кластером. *Kubelet* има примарну улогу у спровођењу сигурносних политика на нивоу радног чвора, док *API* сервер имплементира *Role-Based Access Control* (*RBAC*) за управљање приступом и ауторизацијом на нивоу целокупног кластера. Оба ова механизма уско су повезана са *etcd* складиштем и заједно чине кохерентан оквир за очување интегритета,

[669] Прилагођено према: https://www.opsmx.com/blog/what-is-service-mesh-and-why-is-it-necessary/
[670] https://istio.io/
[671] https://developer.hashicorp.com/consul/docs/k8s/connect
[672] https://linkerd.io/
[673] https://www.envoyproxy.io/

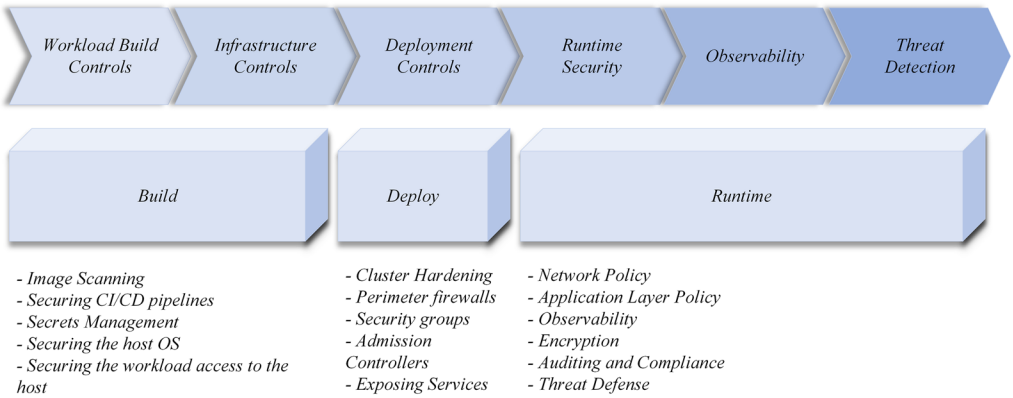
поверљивости и аутентичности података и апликација унутар кластера. Кључна безбедносна функција коју врши *kubelet* је спровођење аутентификације путем TLS (*Transport Layer Security*) сертификата и енкрипција података који пролазе кроз кластер, чиме се спречава прислушкивање пакета у рачунарској мрежи. Поред енкрипције која се врши при сваком трансферу података, подаци који се налазе у оквиру *etcd* базе података су такође енкриптовани (*data-at-rest encryption*), чиме се спречава компромитовање података у случају неауторизованог приступа систему.

Kubelet врши енкрипцију и аутентификацију на нивоу чвора. На нивоу кластера *API* сервер контролише приступ ресурсима у оквиру кластера, путем *RBAC*-а. *RBAC* осигурава конролу приступа преко пермисија које се могу доделити индивидуалним корисницима, групама корисника или сервисима. Преко контроле приступа, могуће је очувати интегритет и поверљивост индивидуалних ресурса и целих апликација.

Поред контроле приступа, која се спроводи на апликационом нивоу, постоји и контрола приступа која се може вршити на мрежном нивоу *ISO*/*OSI* модела. *Kubernetes* дозвољава дефинисање мрежних правила која ће ограничити или забранити комуникацију између одређених пôдова, или са екстерним системима. Ова мрежна правила се обично користе упоредо са *RBAC*-ом како би се ограничио приступ систему у случају напада, као што су *DDoS* или *lateral movement*.

Чест проблем на који се наилази при развоју апликација и њиховом постављању у продукционо окружење је како безбедно делити креденцијале за приступ, као што су лозинке и *SSH* кључеви. *Kubernetes* овај проблем решава путем *Kubernetes Secrets*, који све креденцијале чува шифроване унутар *etcd* складишта, и дозвољава приступ само пôдовима и сервисима којим је одобрен приступ путем *RBAC*-а. Креденцијали се додељују пôдовима при њиховом креирању, путем *mount*-ованог волумена који садржи енкриптоване креденцијале. На тај начин се смањује могућност неовлашћеног приступа осетљивим подацима и олакшава развојни процес, а програмери и пројектанти могу сигурносне мере препустити *Kubernetes*-у. Додатна предност *Kubernetes secrets* је једноставност интегрисања са *CI/CD* алатима.

Поред стандардизације управљања креденцијалима, *CI/CD* системи могу примењивати одређене безбедносне политике над кôдом и апликацијама, које пролазе кроз *CI* или *CD* ток. Безбедносне политике се примењују преко статичке анализе кôда, препознавања рањивости кôда, као и применом добрих безбедносних пракси. Преглед безбедносних мера у *Kubernetes*-у може се видети на Слици 99.



Слика 99. Безбедносне мере у Kubernetes-у[674]

28.5.9 CI/CD

За потребе постављања апликација у продукционо окружење, *Controller Manager* и његови специјализовани контролери *Deployment* и *StatefulSet*, обезбеђују висок ниво аутоматизације у управљању животним циклусом пôдова. Ови контролери омогућавају аутоматизован распоред пôдова на сервере, у складу с дефинисаним политикама и конфигурацијама. У свом раду, у циљу потпуне синхронизације између жељеног и стварног стања апликација у кластеру, координирају рад са осталим компонентама као што су *Scheduler* и *API* сервер. Аутоматизација је посебно корисна за *CI/CD* токове и *DevOps* јер оптимизује брзину и поузданост испоруке софтвера.

Процеси као што су аутоматизована ажурирања, повратак на претходно стање и континуална испорука софтвера, представљају стандардне функционалности *Kubrenetes StatefulSet*-а и *Deployment*-а. У овом контексту, *Deployment* контролер непрекидно комуницира са *API* сервером како би пратио и ажурирао стање пôдова, док *Scheduler* користи софистициране алгоритме за оптималну расподелу ресурса. Уз овај ниво аутоматизације, потреба за мануелним интервенцијама при *CI/CD* токовима знатно је смањена. Главна предност овог приступа је у смањењу људских грешака и убрзавању развојног процеса, уз одржавање стабилности и поузданости система кроз целокупни процес.

Kubernetes, иако није дизајниран са циљем да буде саставни део *CI/CD* токова, због своје могућности да веома лако креира изолована тестна окружења коришћењем пôдова и *job*-ова, постао је кључни и незаменљиви део *CI* токова. *Kubernetes* подржава цео циклус развоја, од компилације, до покретања тестова и анализа кôда. Због лакоће којом се *Kubernetes* подешава, као и интероперабилности с

[674]Прилагођено према: https://medium.com/@mostafa.elnakeb/kubernetes-security-and-observability-strategy-embracing-a-paradigm-shift-in-technology-f94cd60fba6

другим платформама, могуће је *CI* токове покретати преко система за контролу верзија кôда, или директно путем интеграције са *CI* алатима. Интеграцијом са *CI* алатима омогућава се да свака промена кôда проузрокује креирање новог изолованог окружења за тестирање, тако да се избегава конфликт између тестова, или контаминација тестног окружења. Поред стандардних тестних окружења, *Kubernetes* се такође може користити за емулирање различитих продукционих услова ради тестирања оптерећења, или емулирања одређеног сценарија.

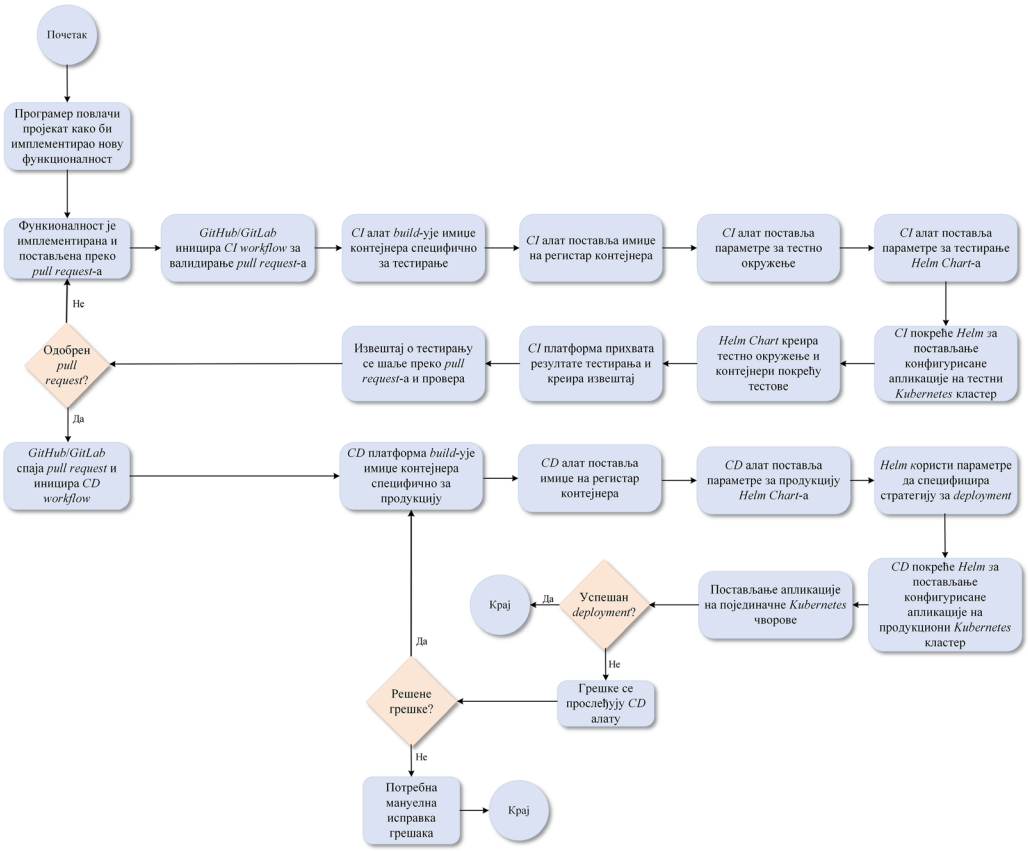
Kubernetes такође преузима главну улогу у континуалној испоруци апликација, где преко интеграције са *CD* алатима, омогућава постављање софтвера у продукционо окружење путем кастомизованих конфигурационих фајлова. *CD* процес почиње спајањем у продукциону грану путем *git*-а, где се иницирају *CD* токови који имају за циљ оркестрацију стратегија ажурирања софтвера, попут: *rolling update*, *blue-green* и *canary*. Ове стратегије ажурирања софтвера могу се кастомизовати за свако појединачно постављање у продукционо окружење, чиме се омогућава брза и сигурна имплементација нових везија софтвера, уз одржавање високог нивоа доступности и поузданости апликације. При постављању у продукционо окружење увек постоји ризик од пада сервиса или увођења нових грешака. Да би се ово спречило, *Kubernetes* омогућава прецизно управљање верзијама апликације и брз повратак на претходну стабилну верзију, уз интеграцију са *CD* алатима. Механизам *rollback*-а се може подесити тако да се различити сегменти апликације могу *rollback*-овати на претходне компатибилне верзије.

28.5.10 Helm Charts

Helm charts је алат у *Kubernetes* екосистему који омогућава ефикасно управљање апликацијама преко пакета. *Helm chart* садржи колекцију (пакет) *YAML* датотека које дефинишу *Kubernetes* ресурсе потребне за покретање апликације, укључујући пôдове, сервисе, волумене и др. Ова колекција датотека омогућава тимовима да стандардизују и аутоматизују процес постављања у продукционо окружење и осигурају конзистентност функционисања апликације у различитим радним окружењима.

Поред груписања релевантних *YAML* фајлова, *Helm charts* омогућавају групно покретање и заустављање целокупне апликације, укључујући и сегменте као што су *front-end*, *back-end,* база података. Додатна функционалност се постиже кастомизацијом *Helm charts*, у којима се прилагођавају конфигурације *YAML* фајлова за потребе различитих окружења. Вредности које се инјектују у *helm chart* могу се генерисати из разних извора, али најчешћи извор су *CI/CD* системи који путем својих токова могу кастомизовати сваки индивидуални *YAML* фајл у оквиру *chart*-а. На пример, *CI/CD* системи могу аутоматски унети вредности као што су специфични конфигурациони параметри или тајни кључеви приликом ажурирања апликације. При континуалној интеграцији, *helm charts* се могу користити за брзо постављање нових изолованих тест окружења путем конфигурације дефинисане

за сваки индивидуални тест у оквиру *CI* алата. Интеграцијом *helm charts* с популарним платформама као што су *Jenkins* и *Gitlab* омогућава се покретање *Helm* скрипти као саставног дела *CI/CD* тока. Ови токови могу параметризовати *Kubernetes deployment*, тако да користе различите стратегије ажурирања попут: *rolling updates*, *canary* или *blue-green*, где се сваки од њих може конфигурисати у складу са специфичним потребама развојног тима. Ова флексибилност при избору стратегије ажурирања омогућава тимовима да прилагоде процес испоруке специфичним захтевима апликације, чиме се минимизују потенцијални проблеми који могу угрозити доступност апликације. Пример једног *CI/CD* решења може се видети на Слици 100.



Слика 100. Пример једног CI/CD решења које користи Helm charts