УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Миљан Бакић

ТЕХНОЛОГИЈА MONGODB CHANGE STREAMS У ДИЗАЈНУ АРХИТЕКТУРЕ Е-ТРГОВИНЕ ВОЂЕНЕ ДОГАЂАЈИМА

мастер рад

Београд, 2026.

Ментор:

доц. др Мирко Спасић
Универзитет у Београду, Математички факултет

Чланови комисије:

проф. др Александар Картељ
Универзитет у Београду, Математички факултет

др Иван Ристовић
Универзитет у Београду, Математички факултет

Датум одбране: _____

Наслов мастер рада: Технологија MongoDB Change Streams у дизајну архитектуре е-трговине вођене догађајима

Резиме: Рад испитује примену механизма *MongoDB Change Streams* као основе за изградњу архитектуре вођене догађајима (*event-driven*) у микросервисном окружењу е-трговине. Иако *MongoDB* није првенствено пројектован као брокер догађаја, показује се да осматрање тока промена над дневником операција (енг. *oplog*) скупа реплика, уз механизам поновног настављања помоћу жетона (енг. *resume token*), омогућава поуздану реактивну обраду, репликацију стања ентитета и обавештавање корисника у реалном времену. Рад упоредно анализира *MongoDB Change Streams* наспрам платформе *Apache Kafka* и базе *EventStoreDB* и вреднује решење у погледу кашњења, скалабилности и једноставности одржавања. Анализа је изведена кроз студију случаја продукционог система који обрађује стотине милиона докумената. Уз њу је развијен и самосталан демонстратор (*OmniChannel.Catalog*), изграђен над истим механизмима, над којим су спроведена мерења.

Кључне речи: MongoDB, Change Streams, архитектура вођена догађајима, CQRS, Event Sourcing, Change Data Capture, микросервиси, скуп реплика, реактивна обрада, е-трговина

Садржај

Глава 1

Увод

Савремене апликације е-трговине све чешће се граде као скупови слабо спрегнутих микросервиса. Такав приступ доноси независно развијање и распоређивање компоненти, али уводи и суштински проблем: како одржати конзистентан поглед на стање ентитета (производа, залиха, цена, статуса објаве) распоређено преко више сервиса, и како кориснику приказати промене у реалном времену. Стање тих ентитета такви системи по правилу чувају у нерелационој бази података, а распрострањен избор за то складиште јесте *MongoDB* [23], у којој се сваки ентитет чува као самосадржајан документ. Класичан одговор на проблем усклађивања јесте *архитектура вођена догађајима* (енг. *event-driven architecture*), у којој сервиси не позивају једни друге синхроно, већ реагују на догађаје који описују промене стања [32, 30].

Такву размену посредује наменски међуслој (енг. *middleware*), то јест брокер догађаја, који догађаје прима од произвођача и прослеђује их потрошачима. Основна одлика тог посредовања јесте раздвајање (енг. *decoupling*) произвођача и потрошача: они међусобно не знају ништа осим облика догађаја, па се развијају, распоређују и надограђују независно. Комуникација се води моделом објављивања и претплате (енг. *publish/subscribe*): произвођач објављује догађај на именовани канал, а сваки заинтересовани потрошач се на тај канал претплаћује и прима сопствену копију догађаја. Пошто ниједна страна не чека другу, обрада се скалира простим додавањем потрошача, а привремена недоступност једног потрошача не зауставља ни произвођача ни остале потрошаче.

Догађаји се обично преносе наменским посредником као што је *Apache Kafka* [1] или се трајно чувају у специјализованом складишту догађаја попут базе

EventStoreDB [4]. Обе технологије су моћне, али уводе додатну компоненту у оперативни стек: још један дистрибуирани систем који треба поставити и одржавати. За системе који *MongoDB* већ користе као примарно складиште података то значи компоненту чија је једина сврха пренос догађаја. Питање којим се овај рад бави јесте да ли је такав посредник неизбежан, или се поуздана архитектура вођена догађајима може изградити и над механизмима које база *MongoDB* већ нуди.

Од верзије 3.6 *MongoDB* нуди механизам *Change Streams*, ток промена који апликацији у реалном времену испоручује сваку измену над колекцијом, ослањајући се на *дневник операција* (енг. *oplog*, скраћено од *operations log*), интерни дневник репликације скупа реплика [22]. Уз механизам поновног настављања тока помоћу *жетона* (енг. *resume token*), *Change Streams* обезбеђује особине које се традиционално очекују од брокера догађаја: поуздану испоруку, настављање после отказа и могућност реконструкције стања. *MongoDB* није првенствено пројектован за размену порука, а рад испитује колико се та технологија ипак може прилагодити таквој улози.

1.1 Мотивација

У микросервисној архитектури податке о истом пословном ентитету често обрађује више сервиса: један уписује намеру корисника, а други је прослеђује спољном систему и прати шта се са њом заиста догодило. Одржавање конзистентног и ажурног погледа на такав ентитет, и то у реалном времену, нетривијалан је задатак. Директно синхроно позивање сервиса ствара крхке ланце зависности, а наивно објављивање догађаја отвара проблем двоструког уписа (одељак 2.3). Наменски брокер догађаја попут платформе *Kafka* тај проблем решава, али по цену постављања и одржавања још једног дистрибуираног система. Организацијама чији системи већ користе *MongoDB* поставља се питање да ли је та цена неопходна.

Мотивација за овај рад произлази из стварног продукционог система за управљање оглашивачким кампањама на платформама *Meta*, *Snapchat* и *TikTok*, који је у потпуности изграђен над механизмом *MongoDB Change Streams*, без платформе *Kafka* или сличног посредника. Тај систем у продукцији одржава базу величине преко **511 GB** са преко **деведесет колекција**, у којима поједине колекције садрже и преко **318 милиона докумената** (глава 3).

Систем тог обима у продукцији функционише без брокера догађаја, што је полазна емпиријска основа овог рада.

Домен оглашавања и домен е-трговине структурно су слични: у оба постоји хијерархија ентитета над којима постоје два независна извора истине. *Жељено стање* (енг. *desired state*) јесте намера власника података, то јест оно што трговац или уредник хоће да ентитет буде. *Посматрано стање* (енг. *observed state*) јесте оно што спољни канал или платформа заиста пријављује о том истом ентитету. Та два стања не морају се поклапати, и управо у томе је суштина проблема.

Најпростији пример јесте промена цене. Трговац у свом каталогу постави нову цену производа: то је жељено стање и оно је у систему видљиво одмах. Продајни канал, међутим, ту измену не прихвата истог тренутка: он је прима, проверава и потврђује, што траје од неколико секунди до неколико сати, а може и да је одбије. Док потврда не дође, канал и даље пријављује стару цену: то је посматрано стање. Систем зато мора да чува обе вредности и да их непрекидно *разрешава* (енг. *reconcile*) у јединствен, материјализован поглед погодан за читање и приказ у реалном времену, поглед у коме је истовремено видљиво и шта је затражено и шта је стварно на снази.

1.2 Предмет и циљ рада

Предмет рада је примена механизма *MongoDB Change Streams* у изградњи архитектуре вођене догађајима у микросервисном окружењу е-трговине. Циљ је да се кроз практичну имплементацију и упоредну анализу покаже како *Change Streams* могу да послуже као основа такве архитектуре, те да се идентификују услови под којима је такав избор оправдан, а под којима није.

Полазни методолошки приступ јесте раздвајање стања сваког ентитета на жељено, посматрано и тренутно (енг. *desired*, *observed*, *current*), уз двостепену (каскадну) обраду токова промена. Тај образац се испитује као могућа замена за платформу *Apache Kafka* и базу *EventStoreDB*, а из њега се изводе смернице за скалабилне руковаоце догађајима (енг. *handler*) и за механизме реконструкције стања (енг. *replay*) из дневника догађаја. Решење се затим вреднује мерењима, у погледу кашњења, скалабилности и једноставности одржавања. Уз рад је развијен и демонстратор *OmniChannel.Catalog* (глава 4) који материјализује све наведене механизме на домену е-трговине и служи као тестна

платформа над којом су спроведена мерења.

1.3 Структура рада

Глава 2 уводи кључне концепте: архитектуре вођене догађајима, обрасце *CQRS* и *Event Sourcing*, хватање промена података (енг. *Change Data Capture*) и детаљан приказ механизма *MongoDB Change Streams*. Глава 3 описује студију случаја (енг. *case study*) продукционог система, укључујући измерене податке о његовој величини и обрасце архитектуре из којих је теза настала. Глава 4 описује дизајн и имплементацију демонстратора. Глава 5 доноси вредновање и упоредну анализу механизма *MongoDB Change Streams* наспрам платформе *Apache Kafka* и базе *EventStoreDB*. Глава 6 сумира доприносе, ограничења и правце даљег рада.

Глава 2

Кључни концепти

Ова глава уводи појмове неопходне за разумевање остатка рада: архитектуру вођену догађајима и сродне обрасце, проблем двоструког уписа и хватање промена података, механизам *MongoDB Change Streams* и његове гаранције, као и алтернативне технологије са којима се касније упоређује.

2.1 Архитектура вођена догађајима

У архитектури вођеној догађајима компоненте комуницирају производњом и потрошњом (енг. *producer–consumer*) *догађаја*, непроменљивих чињеница које описују нешто што се догодило у систему [30]. Уместо да позивалац синхроно чека одговор, произвођач емитује догађај и наставља рад, док заинтересовани потрошачи независно реагују. Овим се постиже слаба спрегнутост (енг. *loose coupling*): произвођач ради независно од потрошача, па се нови потрошачи могу додавати без измене произвођача.

Последица тога је *евентуална конзистентност* (енг. *eventual consistency*) [33]: материјализовани погледи изведени из догађаја не морају бити тренутно ажурни, али конвергирају ка исправном стању у кратком временском прозору. Тај компромис није ствар избора нити га је могуће надоместити: Гилберт и Линч показали су да систем који трпи мрежне поделе (енг. *partition tolerant*) не може истовремено бити и потпуно конзистентан и потпуно доступан [7], па је попуштање конзистентности цена веће доступности. За велику класу апликација е-трговине то је прихватљива размена. Насупрот томе, синхроне зависности између микросервиса стварају спрегнутост у времену извршавања (енг. *temporal coupling*): ако један сервис није доступан, ланац позива пуца.

Догађаји ту зависност уклањају, јер потрошач обрађује догађај када буде у могућности.

Идеја да се дневник базе искористи као извор истине и као ток догађаја централна је у литератури о системима који интензивно користе податке. Крепс је популаризовао дневник као обједињавајућу апстракцију система у реалном времену [17], а Клепман и Крепс га разрађују као пројектантски принцип: сложене системе градити компоновањем реплицираног дневника и оператора над токовима [16]. Клепман детаљно образлаже дуалност табеле и тока (енг. *table–stream duality*), у којој је дневник примарна апстракција, а материјализовани погледи изведене величине [15]. Стопфорд разрађује пројектовање система вођених догађајима над платформом *Kafka* [31], а Њуман даје шири контекст микросервисних архитектура и образаца интеграције [27]. Тај однос дневника и изведеног погледа у овом раду постоји између *append-only* колекција стања и колекција тренутног стања.

2.2 Обрасци *CQRS* и *Event Sourcing*

Образац *CQRS* (*Command Query Responsibility Segregation*) раздваја модел за упис (команде које мењају стање) од модела за читање (упити) [6]. Модел за читање се обично одржава као денормализован, материјализован поглед оптимизован за приказ, који се ажурира као одговор на догађаје произведене на страни уписа. Раздвајање омогућава да се две стране независно скалирају и различито моделују: страна уписа чува нормализовану истину, а страна читања држи облик погодан за конкретан приказ.

Event Sourcing иде корак даље: уместо да чува само тренутно стање, систем чува *целокупан низ догађаја* који су до тог стања довели [5]. Тренутно стање је тада изведена величина: добија се *редукцијом* (енг. *fold*) низа догађаја. Ова особина доноси неколико предности: потпуни ревизиони траг (свака промена је забележена), могућност извођења нових материјализованих погледа накнадно (пуштањем истих догађаја кроз нову пројекцију) и, за овај рад најважније, *реконструкцију* (енг. *replay*): брисањем материјализованог погледа и поновним пролазом кроз дневник догађаја стање се детерминистички поново изграђује.

Цена те особине јесте везаност за облик записаних догађаја. Показано је да је конверзија података при промени схеме један од најтежих проблема систе-

ма заснованих на догађајима, јер стари догађаји морају остати читљиви новом коду или бити преведени у нови облик [28]. Исти аутори су у анкети 25 инжењера из 19 система заснованих на догађајима издвојили пет инжењерских изазова, међу којима је еволуција схеме најтежи [29]. Та студија, међутим, обухвата само системе изграђене над наменским складиштима догађаја, па систем описан у овом раду, изграђен над нерелационом базом опште намене, у њен обухват не улази.

Систем описан у овом раду користи варијанту овог обрасца са наредном претпоставком: колекције стања су непроменљиве и искључиво додајуће (енг. *append-only*), то јест сваки нови податак се уписује као нови документ, а постојећи се не мењају. Хеланд образлаже зашто је непроменљивост постала изводљива основа складиштења: цена простора пала је довољно да се историја може чувати, а непроменљиви записи уклањају потребу за координацијом при упоредном приступу [8].

Материјализовани погледи, то јест колекције тренутног стања, изводе се из тих дневника догађаја *функцијом помирења* (енг. *reconciliation function*), описаном у одељку 3.3. За разлику од чистог обрасца *Event Sourcing*, где су догађаји основни и самостални ентитети, овде су догађаји записи стања, а редослед се одржава временским ознакама.

То је прагматичан избор када је основно складиште нерелациона база, али и место где се губи део гаранција: физичке временске ознаке не дефинишу узрочни редослед у дистрибуираном систему онако како то чине логички сатови [19]. У систему из овог рада тај недостатак ублажава то што ознаке долазе са једног примарног чвора, а обрада одбацује догађаје старије од последње примењеног (одељак 4.6).

Нека, на пример, трговац најпре постави цену на 100 динара, а одмах затим се предомисли и постави 110. Ако због поновног повезивања осматрача старији догађај (онај са ценом од 100 динара) стигне на обраду после новијег, обрада ће видети да је у тренутно стање већ примењен новији догађај и старији ће одбацити, па цена остаје 110 динара. Без те провере тренутно стање би се вратило на 100 динара и остало трајно погрешно.

2.3 Проблем двоструког уписа и образац исходишног сандучета

Чест проблем при увођењу догађаја јесте *двоструки упис* (енг. *dual write*): апликација треба да упише промену у базу и да објави одговарајући догађај. Ако су то две одвојене операције, отказ између њих оставља систем у недоследном стању: база је ажурирана, а догађај није објављен (или обрнуто). Пошто не постоји заједничка трансакција преко базе и брокера порука, гаранција атомичности је нарушена.

Класично решење је образац *исходишног сандучета* (енг. *outbox pattern*): догађај се уписује у посебну табелу (сандуче) *у истој трансакцији* као и промена стања, а засебан процес затим чита сандуче и објављује догађаје. Тиме упис постаје атомичан, а објава поуздана. Хватање промена података (одељак 2.4) представља генерализацију ове идеје: уместо експлицитног сандучета, чита се сам интерни дневник базе, па је сваки упис аутоматски „у сандучету“.

2.4 Хватање промена података

Хватање промена података (енг. *Change Data Capture*, скраћено *CDC*) је техника којом се измене над базом претварају у ток догађаја [15]. Постоје два основна приступа:

Приступ заснован на упитима (енг. *query-based*). Периодично се анкетира база (нпр. по колони временске ознаке) и траже измењени редови. Једноставан је, али уводи кашњење (одређено периодом анкетирања), оптерећује базу и не хвата брисања поуздано.

Приступ заснован на дневнику (енг. *log-based*). Чита се интерни дневник трансакција базе (нпр. *binlog* у бази *MySQL*, *WAL* у бази *PostgreSQL*, дневник операција у бази *MongoDB*, одељак 2.5). Кашњење је ниско, оптерећење минимално, а хватају се све операције, укључујући брисања.

Индустријски најзрелију имплементацију тог приступа представља *Debezium*, који стандардизује хватање промена из релационих база и базе *MongoDB*, чита њихове дневнике репликације и објављује промене на теме платформе *Kafka* [3]. Његов конектор за *MongoDB* као извор промена интерно користи

баш *Change Streams*, што значи да и „канонски“ *CDC* алат почива на механизму који је предмет овог рада. Разлика је само у томе да ли се догађаји даље преносе кроз наменски брокер или се обрађују непосредно уз базу. Механизам *MongoDB Change Streams* јесте управо хватање промена засновано на дневнику, уграђено у саму базу и изложено апликацији као ток догађаја.

2.5 База *MongoDB*: документи, скуп реплика и дневник операција

MongoDB је нерелациона база података опште намене [23]. За разлику од релационих база, где су подаци разложени у редове табела са унапред утврђеним колонама, основна јединица податка овде је *документ*, самосадржајан запис у облику скупа парова *кључ–вредност*, у коме вредност може бити број или текст, али и уграђени документ или низ докумената. Документ је по облику близак *JSON*[1] објекту, а физички се чува у бинарном облику *BSON* (енг. *Binary JSON*), који уз *JSON* типове познаје и типове као што су цео број, децимални број, датум, идентификатор и бинарни низ.

Документи се групишу у *колекције*, а колекције у *базе*. Колекција је по улози најближа табели релационог модела, али јој не одговара у потпуности: она не намеће схему, па два документа у истој колекцији могу имати различит скуп поља и модел се проширује додавањем поља, без превођења постојећих записа. Сваки документ носи обавезно поље `_id`, које је његов јединствен кључ у оквиру колекције и над којим база сама одржава индекс; додатни индекси могу се завести над било којим пољем, укључујући и поља уграђених докумената.

Пошто су подаци у документима денормализовани, оно што би у релационом моделу захтевало спајање више табела овде је по правилу читање једног документа. Подаци се читају упитима над колекцијама и *током агрегације* (енг. *aggregation pipeline*), низом фаза кроз које документи пролазе и у коме се филтрирају, преобликују, групишу и агрегирају (нпр. збир или просек по групи). Тај је појам за овај рад важан јер је, како се види у одељку 2.6, ток промена управо посебна фаза тока агрегација, па се и над њим могу приме-

[1] *JSON* (енг. *JavaScript Object Notation*) је текстуални формат за размену података у коме се вредности записују као парови кључа и вредности, уз низове и угнежђене објекте.

нити исте фазе филтрирања. Догађај који ток промена испоручује такође је документ, чија су поља описана у истом одељку.

MongoDB постиже високу доступност кроз *скуп реплика* (енг. *replica set*), то јест групу чворова од којих је један примарни (прима уписе), а остали секундарни (реплицирају податке) [24]. Репликација се одвија преко дневника операција, посебне ограничене (енг. *capped*) колекције у којој примарни чвор бележи сваку операцију уписа, у идемпотентном облику, редоследом извршавања. Секундарни чворови читају дневник операција и примењују исте операције, чиме се одржавају усклађеним.

Дневник операција треба разликовати од *дневника догађаја*, о коме је реч у остатку рада. Први је интерни дневник репликације саме базе и њиме управља *MongoDB*, а други је *append-only* колекција коју апликација сама одржава као део свог модела података (одељак 3.3). Прво је инфраструктура базе, друго је модел података.

Идемпотентни облик записа у дневнику операција је важан: нпр. операција увећања поља се бележи као постављање на конкретну вредност, тако да поновна примена истог записа не мења резултат. Управо та особина омогућава да секундарни чвор може безбедно да поново примени дневник операција од произвољне тачке, а исту особину користи и механизам поновног настављања тока промена.

Дневник операција је временски ограничен: када достигне своју максималну величину, најстарији записи се одбацују, јер се ограничена колекција понаша као кружни бафер. То одбацивање најстаријих записа назива се *ротација* дневника операција и оно је уобичајен, непрекидан рад базе, а не грешка. Временски прозор који дневник операција покрива (нпр. неколико сати) одређује колико уназад је могуће наставити ток промена. Сама величина је подесива командом `replSetResizeOplog`, а од верзије 4.4 може се задати и најкраће време задржавања записа (`storage.oplogMinRetentionHours`), па се прозор димензионише према потребама система. Ипак, прозор је увек коначан, што је суштинско за разумевање механизма поновног настављања (одељак 2.6).

2.6 Механизам *MongoDB Change Streams*

Change Streams је механизам који апликацији омогућава да *осматра* (енг. *watch*) промене над колекцијом, базом или целим кластером, тако што интерно

прати дневник операција и испоручује структуриране догађаје промена [22]. Механизам захтева скуп реплика: над самосталним чвором није доступан, јер дневник операција не постоји. У основи, ток промена је специјалан ток агрегација са фазом `$changeStream`, што значи да се на њега могу надовезати уобичајене фазе агрегације (нпр. `$match`, `$project`).

Документ промене и врсте операција

Сваки догађај је документ типа `ChangeStreamDocument` који, између осталог, садржи: *тип операције* (`operationType`: `insert`, `update`, `replace`, `delete`, `invalidate`, ...), кључ документа (`documentKey`), време на кластеру (`clusterTime`)[2], опис измене (`updateDescription` са листом измењених и уклоњених поља) и, опционо, пуни документ. Осматрач може да филтрира ток на самом серверу, фазама агрегације (нпр. да прима само додавања нових докумената), чиме се смањује саобраћај и оптерећење. Ова могућност је искоришћена у пројекторима демонстратора и продукционог система, који осматрају *append-only* дневнике и филтрирају само догађаје додавања (`insert`).

Опције пуног документа и слике пре промене

Подразумевано, догађај измене садржи само опис измењених поља. Уз опцију `FullDocument = UpdateLookup` *MongoDB* додатно доставља тренутни цео документ. То је, међутим, документ у тренутку *потраге* (енг. *lookup*), а не нужно у тренутку измене: база уз опис измене накнадно прочита и сам документ из колекције, па ако је у међувремену стигла још једна измена истог документа, потрага враћа већ ту, новију садржину. При брзим узастопним изменама то значи да међустања могу остати невидљива: ако цена истог производа у кратком интервалу пређе са 100 на 110, а затим на 120, оба испоручена догађаја могу носити пуни документ са вредношћу 120. Обрада којој су потребне тачне међувредности или сама делта не сме се стога ослонити на пуни документ, него на опис измене (`updateDescription`) или на слику пре промене.

[2]Поље `clusterTime` је логичка временска ознака коју *MongoDB* додељује свакој операцији у скупу реплика. Састоји се од броја секунди од почетка епохе, то јест од 1. јануара 1970. у 00:00:00 по времену *UTC*, и редног броја операције унутар те секунде, чиме се добија вредност која у оквиру кластера строго расте и једнозначно одређује место операције у дневнику операција.

Од верзије 6.0 доступна је и *слика пре промене* (`FullDocumentBeforeChange`), корисна за рачунање делти и за руковање брисањима, где пуни документ више не постоји. Обе опције захтевају да буду омогућене на нивоу колекције (`changeStreamPreAndPostImages`), што демонстратор ради при иницијализацији.

Жетон настављања: састав и поновно настављање

За поуздану обраду најважнији је *жетон настављања* (енг. *resume token*). Уз сваки испоручени догађај везан је жетон који кодира позицију у току, и то тако да за апликацију нема унутрашњу структуру: она га не тумачи, него га само чува и враћа бази онаквог каквог га је добила. У пракси је изведен из времена на кластеру (`clusterTime`) и идентификатора документа и серијализован у облик погодан за поређење. Апликација може у одређеним тренуцима да га трајно сачува, па се после отказа или поновног покретања ток отвара уз опцију `ResumeAfter` (или `StartAfter`) са последњим сачуваним жетоном и наставља тачно одатле, без губитка догађаја. У даљем тексту се за овај механизам користи само термин *жетон*.

Уз *идемпотентну* обраду, ово даје семантику испоруке „најмање једном“ (енг. *at-least-once*): после отказа може доћи до поновне испоруке малог броја већ обрађених догађаја (оних од последњег сачуваног жетона до тренутка отказа), али пошто обрада не мења резултат при понављању, коначно стање остаје исправно. Семантика „тачно једном“ (енг. *exactly-once*) на нивоу испоруке није загарантована; она се постиже тек идемпотентношћу на страни потрошача.

Ограничење произлази из прозора дневника операција: ако је жетон старији од најстаријег записа у дневнику операција (нпр. систем је био угашен дуже од трајања прозора, или је дошло до наглог прилива уписа који је „прогутао“ прозор), настављање није могуће. *MongoDB* тада враћа грешку (код `286 ChangeStreamHistoryLost` или `260 InvalidResumeToken`), коју апликација мора да препозна и да се опорави поновном синхронизацијом (одељак 3.5).

Гаранције испоруке

Системи за размену догађаја разликују се по гаранцији испоруке (табела 2.1). „Највише једном“ (енг. *at-most-once*) допушта губитак, „најмање једном“ (*at-least-once*) допушта дупликате, а „тачно једном“ (*exactly-once*) не допушта ни једно ни друго, али је најскупља и на нивоу самог транспорта често немогућа без сарадње потрошача. У пракси се „тачно једном“ по правилу постиже комбинацијом испоруке „најмање једном“ и *идемпотентне* обраде на страни потрошача, што је приступ и у овом раду: жетон даје „најмање једном“, а идемпотентни уписи са провером временске ознаке чине поновну обраду безопасном.

Табела 2.1: Семантика испоруке уз одговарајући механизам

Семантика	Ризик	Како се постиже
at-most-once	губитак поруке	без потврде и настављања
at-least-once	дупликати	потврда и настављање (жетон, померај)
exactly-once	—	„најмање једном“ и идемпотентна обрада

Редослед, скалирање и хоризонтално партиционисање

Над појединачним скупом реплика *Change Streams* гарантује *тотални редослед* догађаја (редослед из дневника операција). У кластеру подељеном *хоризонталним партиционисањем* (енг. *sharding*) [25], догађаји са различитих партиција спајају се по времену на кластеру, уз координацију преко усмеривача, што уводи додатну сложеност и благо кашњење ради обезбеђивања глобалног редоследа. За разлику од платформе *Kafka*, где је паралелизам потрошача уграђен кроз партиције и потрошачке групе, код механизма *Change Streams* хоризонтално скалирање потрошача мора се решити на нивоу апликације. Демонстратор развијен у оквиру овог рада то чини усмеравањем догађаја на више радника по хешу идентификатора ентитета, чиме се очувава редослед по кључу уз паралелизам међу различитим ентитетима (одељак 4.5).

2.7 Обавештавање у реалном времену: библиотека *SignalR*

SignalR је библиотека платформе *ASP.NET Core* за двосмерну комуникацију сервера и клијената у реалном времену [20]. Апстрахује транспорт, бирајући најбољи доступни (*WebSocket*, а уз деградацију *Server-Sent Events* или *long polling*), и нуди појам *гр̄уп̄а* конекција, што омогућава циљано слање порука подскупу клијената. У архитектури из овог рада *SignalR* чини последњи корак: промене уочене над материјализованим погледом (преко *Change Streams*) прослеђују се повезаним клијентима груписаним по релевантном кључу (нпр. продајном каналу или налогу), тако да сваки клијент прима само промене које га се тичу.

2.8 Алтернативне технологије

Платформа *Apache Kafka*

Apache Kafka је дистрибуирана платформа за токове догађаја заснована на партиционисаном, реплицираном, трајном дневнику [18, 1]. Догађаји се уписују у *т̄ем̄е* (енг. *topics*) подељене на партиције ради хоризонталне скалабилности; редослед је загарантован у оквиру партиције, а потрошачи прате свој померај (енг. *offset*) независно, што омогућава да више потрошачких група независно чита исти ток. Уз *Kafka Connect* и *Debezium*, платформа *Kafka* служи и као *CDC* инфраструктура. Њена је снага у изузетно високом протоку, трајном задржавању и уграђеном скалирању потрошача, али по цену засебног кластера (брокери и координација) о ком се надаље мора водити оперативна брига.

База *EventStoreDB*

EventStoreDB је база података специјализована за *Event Sourcing*: догађаји се чувају у токовима (енг. *streams*) по агрегату, уз глобални ток `$all` и уграђену подршку за претплате (*catch-up* и *persistent*) и серверске пројекције [4]. За разлику од базе *MongoDB*, где су догађаји изведени из промена стања, у бази *EventStoreDB* је догађај примарни концепт који се експлицитно уписује. Ово доноси природну подршку реконструкцији и *временском п̄ут̄овању* (енг. *time*

travel), то јест читању стања агрегата какво је било у произвољном тренутку у прошлости, али уводи специјализовано складиште које се, по правилу, користи упоредо са базом за материјализоване погледе.

За разлику од претходно наведених приступа, који претежно разматрају релационе изворе и наменска складишта догађаја, а платформу *Kafka* као транспорт, овај рад посматра *MongoDB Change Streams* као *самодовољан* механизам: он је истовремено извор промена, гарант редоследа и основа за поновно настављање. Детаљно поређење тог механизма са наведеним алтернативама, укључујући непосредно мерење латенције и смернице о томе када је оперативно једноставнији приступ оправдан, дато је у глави 5.

Глава 3

Студија случаја: продукциони систем

Ова глава описује продукциони систем компаније *HunchAds* за управљање оглашивачким кампањама, изграђен у потпуности над механизмом *MongoDB Change Streams* [9, 10, 11, 12, 13, 14]. Наводи о архитектури у овој глави изведени су из изворног кода тих сервиса и из непосредног рада аутора на њиховом развоју и одржавању, а наводи о обиму из мерења над продукционом базом; репозиторијуми су интерни, па им је приступ ограничен. Систем служи као емпиријска потврда да је ова технологија применљива у реалном обиму и као извор архитектонских образаца који су затим пренети у демонстратор (глава 4).

3.1 Домен и обим

Систем управља хијерархијом оглашивачких ентитета на више платформи (*Meta*, *Snapchat*, *TikTok*): *кампања* садржи *скупове огласа* (енг. *ad set*), а они *огласе* (енг. *ad*). Продавац туристичких аранжмана тако за лето 2026. године води једну кампању са задатим укупним буџетом. У њој један скуп огласа циља породице са децом, а други парове без деце, при чему сваки скуп садржи по неколико огласа, различитих слика, наслова и позива на акцију, међу којима платформа расподељује буџет тог скупа. За сваки од тих ентитета систем прати жељено и посматрано стање (одељак 1.1), то јест оно што уредник намерава и оно што платформа заиста пријављује, и мири их у јединствен материјализован поглед.

Обим система у продукцији документован је непосредним мерењем над базом на платформи *Meta* (табела 3.1). Ради се о бази од преко **511 GB** са преко **90 колекција**, на *MongoDB* верзији 8.0, у скупу реплика са три члана (примарни, секундарни, арбитар). Величина дневника операција је око 47 GB, што даје временски прозор реда неколико сати; тај параметар директно одређује политику поновног настављања и опоравка.

Табела 3.1: Обим кључних колекција продукционе базе (платформа *Meta*)

Колекција	Број докумената	Величина
`adHourByAdvertiserTimeZoneBreakdowns`	318.256.526	932 GB
`adMetrics`	33.970.490	130 GB
`adsProprietaryStates`	15.180.596	24 GB
`adsObservedStates`	13.280.924	15 GB
`adsMetaProprietaryStates`	12.369.616	8 GB
`adsProcessingStatuses`	6.023.820	2 GB
`adsCurrentState`	1.558.084	3 GB
`adSetsCurrentState`	335.289	0,5 GB
`campaignsCurrentState`	62.310	0,1 GB

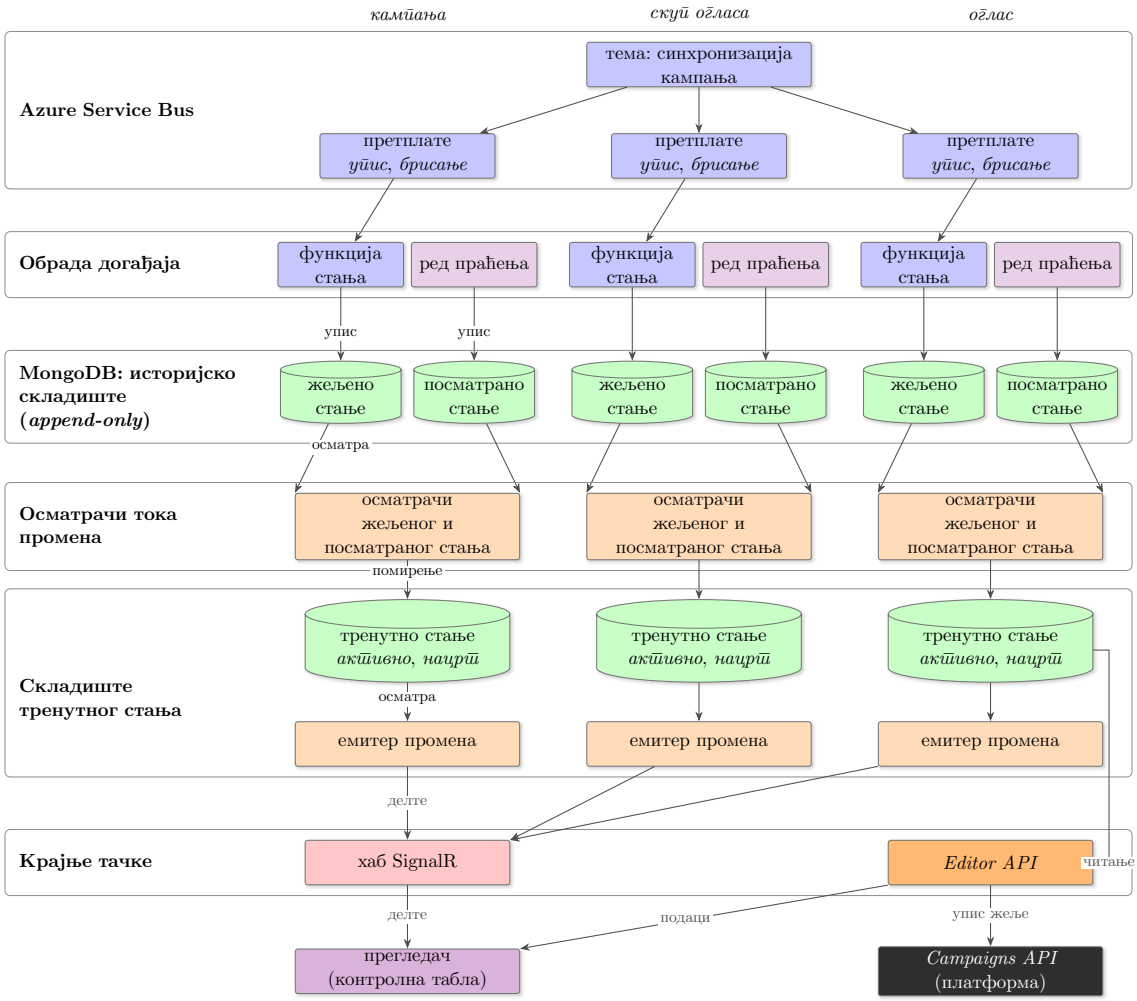
Однос између дневника догађаја и изведеног погледа овде је очигледан: преко 15 милиона докумената жељеног стања и преко 13 милиона докумената посматраног стања материјализује се у око 1,5 милиона докумената тренутног стања на нивоу огласа. Тренутно стање је, дакле, сажет поглед изведен над великим непроменљивим дневником догађаја, што одговара обрасцу *Event Sourcing*. Цела обрада одвија се без платформе *Kafka* или сличног посредника.

3.2 Архитектура

Целокупан ток података приказан је на слици 3.1. Три колоне на њој одговарају трима нивоима хијерархије оглашивачких ентитета (кампања, скуп огласа, оглас), а исти образац понавља се независно на свакој платформи. Догађаји улазе преко посредника *Azure Service Bus*[1] и спољног праћења платформи; смештају се у непроменљиве колекције стања (историјско складиште догађаја); одатле их осматрачи тока промена мире у колекције тренутног ста-

[1] *Azure Service Bus* је услуга размене порука на платформи *Microsoft Azure*, са редовима, темама и моделом објављивања и претплате [21].

ња; а друга група осматрача над тренутним стањем емитује промене према слоју *SignalR*, који их испоручује клијентима у реалном времену.



Слика 3.1: Ток података продукционог система, од догађаја до обавештавања у реалном времену.

3.3 Модел стања: жељено, посматрано и тренутно

Методолошки приступ почива на раздвајању стања сваког ентитета на три семантички различите врсте:

Жељено стање (енг. *desired state*). Намера система, то јест оно што уредник жели да ентитет буде (назив, буџет, статус објаве). Уписује

се као нови документ у *append-only* колекцију; постојећи записи се не мењају. У самом продукционом систему то стање носи наслеђену интерну ознаку *proprietary*, која се задржала у називима колекција и поља; за сам појам се у овом раду користи термин *desired state*.

Посматрано стање (енг. *observed state*). Оно што платформа заиста пријављује преко свог јавног апликативног интерфејса (енг. *application programming interface*, скраћено *API*), то јест скупа крајњих тачака преко којих спољни систем нуди приступ својим подацима (стварни статус, време измене). Такође *append-only*.

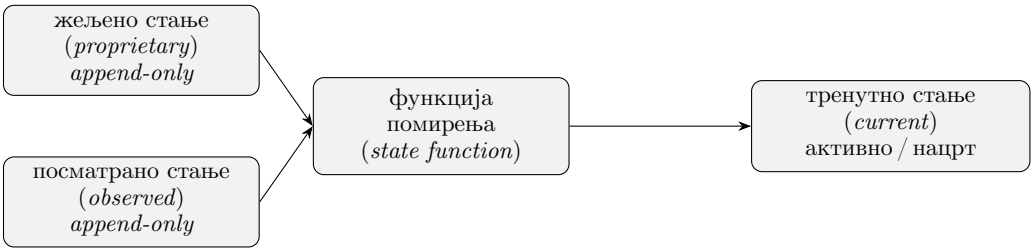
Тренутно стање (енг. *current state*). Јединствен, денормализован и помирен поглед који обједињује жељено и посматрано стање. Ово је модел за читање који користи кориснички део апликације (енг. *frontend*).

Тренутно стање користи поље са двоструком вредношћу (`DualStateField`) које истовремено носи *активну* вредност (објављену) и *нацрт* (енг. *draft*, измену која чека потврду). Тиме је могуће приказати кориснику и тренутно живо стање и неспроведену измену, а измена се „објављује“ (нацрт постаје активна вредност) тек када је платформа потврди.

Слика 3.2 приказује однос три врсте стања. Два независна *append-only* извора (жељено и посматрано стање) улазе у функцију помирења, која их редукује у јединствен документ тренутног стања. Овакво раздвајање има важну последицу: намера система (шта желимо) и стварност (шта платформа пријављује) чувају се одвојено и никада се међусобно не преписују; помирење их спаја тек у изведеном погледу. Систем зато у сваком тренутку зна и шта је тражено и шта је стварно затечено, што је основа за поуздано поновно слање команди, детекцију неслагања и приказ нацрта уз живо стање.

Разлику међу трима врстама стања најјасније показује пример из продукције. Уредник у 12:00 подигне дневни буџет једног огласа са 50 на 80 евра. Тај упис одмах ствара нови документ у колекцији жељеног стања: жељени буџет је 80 евра. Платформа, међутим, измену не примењује истог тренутка: она је прима, проверава и објављује, што траје, а може и да је одбије (нпр. због правила о садржају или недовољних средстава на налогу). Док се то не догоди, праћење платформе и даље пријављује буџет од 50 евра и статус „активан“, и то се као нови документ уписује у колекцију посматраног стања. Тренутно стање је документ који функција помирења изводи из обе колекције: у њему

је *активна* вредност буџета 50 евра (оно што је стварно на снази на платформи), а *нацрт* 80 евра (оно што је затражено и чека потврду). Када платформа пријави буџет од 80 евра, помирење нацрт „објављује“, па активна вредност постаје 80, нацрта више нема и три стања су поново усклађена. Ако платформа измену одбије, жељено стање остаје 80, посматрано 50, а тренутно стање носи оба броја уз статус одбијања, чиме је неслагање видљиво и уреднику и систему.



Слика 3.2: Три врсте стања и функција помирења.

3.4 Каскадни токови промена

Обрада је организована у два степена који чине *каскаду* токова промена: први ток уписује у колекцију коју осматра други.

Први степен: осматрачи дневника стања. Осматрачи над *append-only* колекцијама жељеног и посматраног стања (филтрирани само на додавања) примењују функцију помирења (енг. *state function*) из одељка 3.3 и редукују сваки нови догађај у документ тренутног стања. У стварном систему основни модел се притом рашчлањава на више колекција: један сервис истовремено осматра *четири*. Уз две носеће, жељено и посматрано стање из одељка 3.3, тамо постоје и две помоћне: колекција статуса обраде, у коју се бележи исход сваког слања ка платформи (успех, грешка, чекање), и колекција жељеног стања са платформе, у којој се чува оно што је платформа примила као намеру, а што се због делимично примењених измена може разликовати од оног што је уредник затражио. Све четири су *append-only* и све четири улазе у исту функцију помирења, која их мири у једну колекцију тренутног стања, уз проверу временских ознака која спречава регресију стања при догађајима ван редоследа.

Други степен: емитери (енг. *change handlers*). Осматрачи над колекцијама тренутног стања (свих типова операција, уз слику пре промене) рачунају делту промене, групишу је по налогу и шаљу је ка слоју за обавештавање. У продукцији се делте прослеђују преко посредника *Azure Service Bus*, одакле их засебна компонента испоручује клијентима преко хаба *SignalR*.

3.5 Реконструкција, задржавање и резилијентност

Поновно настављање

Сваки осматрач трајно чува свој жетон у наменској колекцији (по имену сервиса и тока), у одређеним тренуцима и при уредном гашењу. При покретању ток се наставља од сачуваног жетона. Пошто су уписи у тренутно стање идемпотентни (`upsert`[2] уз проверу временске ознаке), поновна обрада већ примењених догађаја је безбедна. Идемпотентност при томе не потиче од самог `upsert`-а: упис изведен из затечене вредности, попут увећања бројача, поновном применом даје другачији резултат. Носе је два својства конкретног уписа. Прво, догађај се уписује као пуна замена документа конкретним вредностима, а не као измена изведена из затеченог стања. Друго, обрада унапред одбацује догађај старији од последње примењеног, па поновљени догађај или уписује исту вредност или бива одбачен.

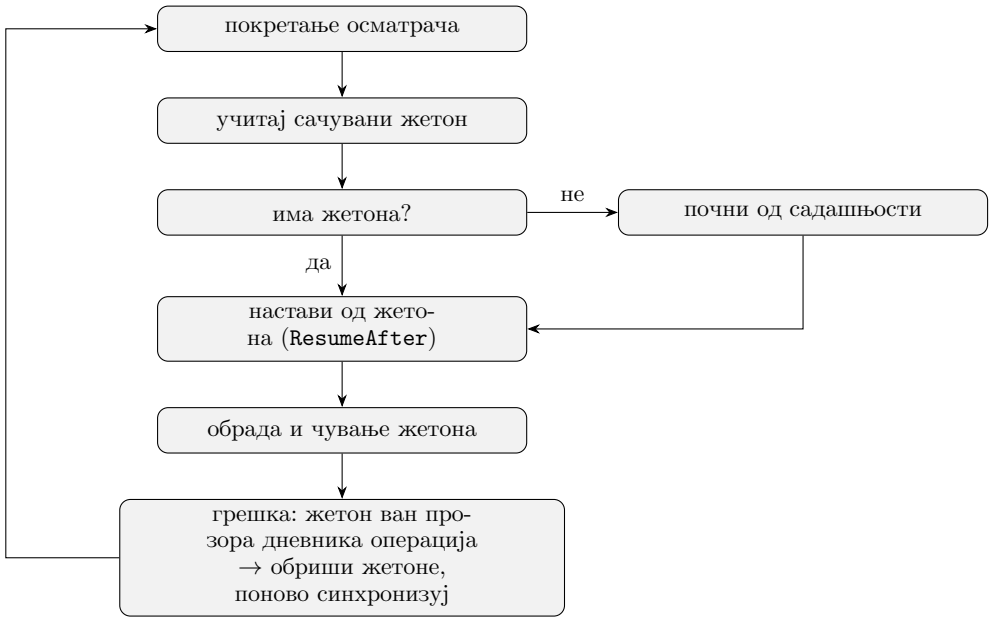
Опоравак од ротације дневника операција

Ако је жетон старији од прозора дневника операција, то јест ако је ротацијом (одељак 2.5) одбачен запис на који жетон показује, систем препознаје грешку (одељак 2.6), брише жетоне и поново се синхронизује из извора. Ово је свесно прихваћен компромис проистекао из ограниченог трајања дневника операција.

Слика 3.3 сажима логику покретања и опоравка осматрача. При покретању се учитава сачувани жетон и ток се наставља тачно одатле; у регуларном

[2] `upsert` је упис који документ ажурира ако постоји, а уписује га као нов ако не постоји (енг. *update* и *insert*).

раду жетон се чува у одређеним тренуцима. Ако сачуваног жетона нема, што је случај при првом покретању или после брисања жетона, ток се отвара без опције `ResumeAfter` и почиње од садашњости: жетон се не наслеђује него га *MongoDB* додељује тек првом наредном догађају, па измене настале раније овим путем нису доступне. Ако *MongoDB* пријави да је жетон ван прозора дневника операција, осматрач брише жетоне и поново се синхронизује из извора.



Слика 3.3: Логика покретања, настављања и опоравка осматрача.

Задржавање

Колекције дневника догађаја имају индексе *TTL* (енг. *time to live*), нпр. 60 дана, па складиште догађаја у овој поставци покрива ограничен временски прозор. Индекс *TTL* јесте обичан индекс над пољем типа датум, уз који се задаје рок трајања: посебан позадински процес базе у регуларним интервалима пролази кроз тај индекс и сам брише документе чија је вредност поља старија од задатог рока [26]. Брисање, дакле, не покреће апликација него база, па се старење дневника догађаја уређује једном одлуком о року, а не логиком у коду. То је одлука о трошку складиштења на нивоу конкретног система, а не ограничење технологије: без индекса *TTL append-only* колекције чувале би се трајно, па би и реконструкција била могућа од почетка времена. Однос

овог избора према трајном дневнику платформе *Kafka* и базе *EventStoreDB* размотрен је у глави 5.

Отпорност на отказе

Осматрачи се аутоматски поново повезују уз ограничен пораст кашњења при опорављивим грешкама (прекид везе, промена примарног чвора), користе ограничене редове са више радника ради контроле притиска (енг. *backpressure*), а вишедокументне каскадне измене извршавају унутар трансакција.

Контрола притиска и отровне поруке

Сваки осматрач пуни ограничени ред (енг. *bounded channel*) који опслужује више радника. Када радници заостану, ред се пуни и произвођач (читач тока) се блокира, чиме се природно успорава читање и спречава неограничено гомилање у меморији. Ако обрада појединог документа трајно не успева, тзв. *отровна порука* (енг. *poison message*), систем је после ограниченог броја покушаја прескаче уз бележење, померајући жетон даље, како један неисправан документ не би зауставио цео ток; тај појединачни губитак надокнађује се при следећем поновном учитавању на страни клијента. Ови обрасци пренети су и у демонстратор (одељак 4.5).

3.6 Поновљивост обрасца

Исти образац примењен је независно на три платформе: *Meta* [9, 10], *Snapchat* [11, 12] и *TikTok* [13, 14]. Сервиси су структурно готово идентични и разликују се само у прилагођивачу за спољни интерфејс *API* и у префиксу назива колекција. Механизам осматрача (петља опоравка, жетон, ограничени редови, идемпотентни уписи) понавља се као шаблон, па се нова платформа уводи само писањем новог прилагођивача за њен *API*, без измене саме обраде тока промена.

Други случај употребе: агрегација метрика

Исти образац примењен је и на суштински другачији проблем, агрегацију метрика оглашавања у реалном времену, у засебном сервису. Ту се сирови

подаци о учинку (прегледи, кликови, конверзије) сливају на нивоу појединачног огласа, а систем их мора непрекидно сабирати на ниво скупа огласа и кампање.

Решење поново користи *Change Streams*, овога пута у израженој *каскади*: први ток промена нормализује сирове метрике различитих платформи у јединствену колекцију (`adMetrics`, реда 34 милиона докумената), а други ток осматра ту колекцију и инкрементално ажурира агрегате на вишим нивоима. Разлика у односу на управљање стањем јесте у природи операције: уместо замене документа користи се *инкрементално сабирање* (`$inc`), при чему се из слике пре промене (`FullDocumentBeforeChange`) рачуна делта (разлика нове и старе вредности) и додаје на текући збир вишег нивоа. Агрегати се тако одржавају без поновног пресумирања целог скупа, што је идеја редукције догађаја пресликана на нумеричке величине.

Иако се домен разликује (нумеричка агрегација наспрам помирења стања), инфраструктура је иста: осматрачи филтрирани током агрегација, жетон за настављање, ограничени редови за контролу притиска и идемпотентни уписи. То што иста инфраструктура покрива два суштински различита проблема говори да није реч о ад-хок решењу. Наредна глава исти образац преноси и на трећи домен, кроз демонстратор развијен за потребе овог рада.

Глава 4

Дизајн и имплементација демонстратора

Ради практичне провере механизама и мерења, развијен је самосталан, репродуцибилан демонстратор *OmniChannel.Catalog*, реактивни каталог е-трговине изграђен над механизмом *MongoDB Change Streams*. Демонстратор је написан у језику *C#* на платформи *.NET 10* и испоручује се као скуп контејнерских слика, заједно са скупом реплика базе *MongoDB*, а поступак покретања и кључна подешавања дати су у додатку А.2. Целокупан изворни код, укључујући тестове и програме за мерење, објављен је као слободан софтвер под лиценцом *MIT* и доступан је у јавном репозиторијуму [2], па се сва мерења из главе 5 могу поновити.

4.1 Домен и пресликавање на продукцију

Демонстратор пресликава хијерархију и модел стања из студије случаја на домен е-трговине (табела 4.1). Трговац води каталог у три нивоа: *Производ* → *Варијанта* → *Листинг*.

Производ је артикал онако како га трговац замишља, на пример мушка памучна мајица одређеног модела: носи назив, опис и категорију, али се сам по себи не продаје. *Варијанта* је конкретна продајна изведба тог производа, та мајица у величини `L` и плавој боји, и она носи цену и залиху. *Листинг* је приказ једне варијанте на једном продајном каналу, то јест излог: иста плава мајица величине `L` може истовремено бити изложена у сопственој продавници трговца по цени од 2.500 динара и на спољном каналу по 2.700 динара, при

чему је на једном каналу објављена, а на другом још чека преглед.

Продајни канали су двојаки. Сопствена продавница (енг. *webshop*) јесте канал у власништву трговца, где он сам одлучује шта је објављено, па се листинг објављује одмах и без провере. *Електронска тржишта* (енг. *marketplace*) туђе су платформе на којима више трговаца излаже своју робу, нпр. *Amazon* или *eBay*. Таква тржишта пре објаве спроводе сопствену модерацију, могу измену и да одбију, а о свом стању извештавају са закашњењем. Управо та разлика чини раздвајање жељеног и посматраног стања потребним.

Табела 4.1: Пресликавање продукционог модела на домен демонстратора

Продукција (оглашавање)	Демонстратор (е-трговина)
`Campaign` → `AdSet` → `Ad`	Производ → Варијанта → Листинг
платформа, налог	продајни канал
жељено стање (*proprietary*)	жељено стање трговца (цена, залиха, статус)
посматрано стање (*observed*)	стање које канал пријављује
тренутно стање (уз `DualStateField`)	помирено тренутно стање (активно, нацрт)

Колекције прате исту шему као у продукцији, само на домену е-трговине. Жељено стање бележе три дневника догађаја (`productsProprietaryStates`, `variantsProprietaryStates` и `listingsProprietaryStates`), а посматрано стање само један, `listingsObservedStates`, јер о стању извештавају једино канали, и то на нивоу листинга. Помирени погледи су `productsCurrentState`, `variantsCurrentState` и `listingsCurrentState`, а колекција `resumeTokens` чува по један жетон по осматрачу, а њено складиште дато је у додатку А.1.

4.2 Модел података

Табела 4.2 даје преглед колекција и њихових улога. Дневници су *append-only* и над њима осматрачи филтрирају само додавања; колекције тренутног стања су материјализовани погледи над којима постоји јединствени индекс по пољу `entityId` (спречава дупликате и служи као заштита при упоредним уписима).

Страна за упис (`EditorApi`) изложена је као *REST*[1] интерфејс (табела 4.3). Свака измена додаје нови документ у одговарајући *append-only* дневник;

[1] *REST* (енг. *Representational State Transfer*) је стил пројектовања веб интерфејса у коме се сваки ресурс адресира сопственом путањом, а операција над њим се исказује *HTTP* методом (`GET` за читање, `POST` за креирање, `PUT` за измену).

Табела 4.2: Колекције демонстратора (у бази `omnichannelCatalog`)

Колекција	Врста	Улога
`productsProprietaryStates`	*append-only*	жељено стање производа
`variantsProprietaryStates`	*append-only*	жељено стање варијанти
`listingsProprietaryStates`	*append-only*	жељено стање листинга
`listingsObservedStates`	*append-only*	стање које канал пријављује
`productsCurrentState`	модел за читање	помирен поглед производа
`variantsCurrentState`	модел за читање	помирен поглед варијанти
`listingsCurrentState`	модел за читање	помирен поглед листинга
`resumeTokens`	контролна тачка	по један жетон по осматрачу

уписи никада не мењају постојеће записе, чиме се чува историја и омогућава реконструкција.

Табела 4.3: Кључне тачке интерфејса `EditorApi` и контроле канала

Метод	Путања	Дејство
`POST`	`/api/products`	креира производ (нови *proprietary* запис)
`PUT`	`/api/products/{id}`	измена производа (нови запис)
`POST`	`/api/listings`	креира листинг на каналу
`PUT`	`/api/listings/{id}`	измена листинга (нпр. цене → нацрт)
`POST`	`/api/listings/{id}/discard-draft`	одустајање од измене (одбацивање нацрта)
`POST`	`/api/channel/{id}/reject`	канал одбија листинг (посматрано: `rejected`)
`POST`	`/api/channel/{id}/observe`	канал пријављује произвољно стање (нпр. потврду измене)
`POST`	`/api/admin/seed`	учитава почетни каталог из скупа података у облику *CSV*
`POST`	`/api/admin/replay`	реконструкција тренутног стања из дневника догађаја
`POST`	`/api/admin/reset`	брисање свих података (за демо)
`POST`	`/api/admin/simulator`	паузирање и покретање симулираног канала

4.3 Компоненте

Систем је један процес платформе *ASP.NET Core* са више позадинских сервиса (`BackgroundService`), што прати продукциони образац:

`EditorApi`. *REST* страна за упис; уписује жељено стање у *append-only* дневник.

`ChannelSimulator`. Симулира спољни канал који у регуларним интервалима пријављује посматрано стање (одобрено, на прегледу или одбијено).

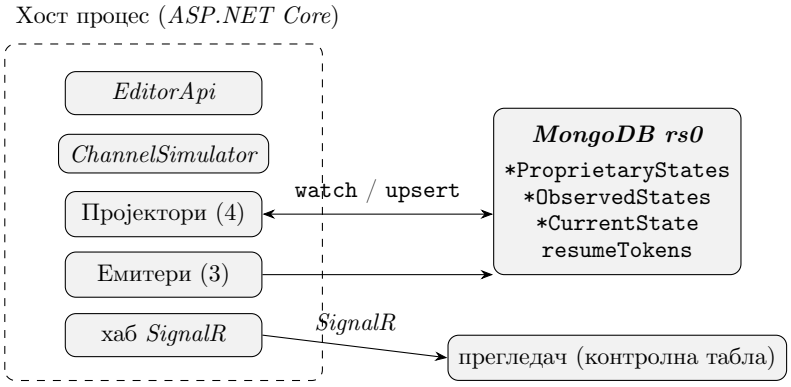
Пројектори. Осматрачи тока промена над дневницима, који мире догађаје у тренутно стање.

Емитери. Осматрачи над тренутним стањем, који промене шаљу преко библиотеке *SignalR*.

Хаб *SignalR* и контролна табла. Приказ промена у реалном времену.

Алат за реконструкцију. Поновно грађење тренутног стања из дневника догађаја, из командне линије.

Слика 4.1 приказује распоред при извршавању. Сви позадински сервиси живе унутар *једног* апликационог процеса, а једина инфраструктура поред њега је скуп реплика базе *MongoDB*: *засебног брокера догађаја нема*. Та минималност је главна практична предност решења, јер се цео систем одржава као једна апликација уз базу коју би систем и иначе имао.



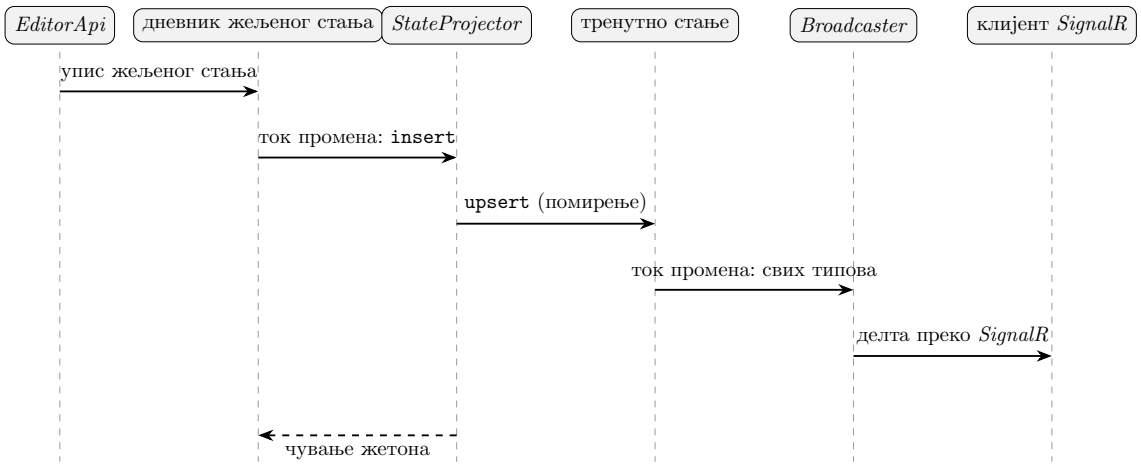
Слика 4.1: Распоред при извршавању.

4.4 Ток догађаја кроз систем

Слика 4.2 је дијаграм секвенци који приказује пут једне измене кроз систем, од уписа жељеног стања до испоруке клијенту, кроз две узастопне фазе осматрања токова промена (каскаду).

Ток има пет корака. Уредник шаље измену крајњој тачки компоненте `EditorApi`, која је уписује као нови документ у *append-only* дневник жељеног стања. У том тренутку измена је трајно забележена, али још није видљива у моделу за читање. Тај упис база бележи у дневнику операција, одакле га као догађај типа `insert` преузима први осматрач, `StateProjector`. Он на догађај примењује функцију помирења и резултат уписује у колекцију тренутног стања операцијом `upsert`. Тај упис је сам по себи нова промена, па га као догађај види други осматрач, `Broadcaster`, овог пута над свим типовима операција и уз слику пре промене, што му омогућава да израчуна делту. Делту најзад прослеђује повезаним клијентима преко хаба *SignalR*, груписаним по продајном каналу, тако да сваки клијент прима само промене које га се тичу.

Кључна је особина да ниједан корак не позива наредни: свака компонента само уписује у своју колекцију, а следећа реагује на промену коју је база објавила. Испрекидана стрелица означава чување жетона у одређеним тренуцима, које осматрачу омогућава да после отказа настави тачно одатле. Пуна имплементација петље осматрања дата је у додатку А.1.



Слика 4.2: Дијаграм секвенци: пут измене кроз каскаду токова промена.

4.5 Језгро: осматрач тока промена

Централну апстракцију чини генеричка класа `ChangeStreamProcessor`, параметризована типом документа, коју наслеђују сви осматрачи. Она обједињује учитавање жетона, отварање тока, контролу притиска и опоравак. Листинг

4.1 приказује срж отварања тока: примену сачуваног жетона (`ResumeAfter`), захтев за пуним документом (`UpdateLookup`) и чување жетона у одређеним тренуцима.

Листинг 4.1: Отварање тока промена уз жетон настављања

```
var options = new ChangeStreamOptions
{
    FullDocument = FullDocument,
    MaxAwaitTime = TimeSpan.FromMilliseconds(Settings.MaxAwaitTimeMs),
    BatchSize = Settings.BatchSize
};
var saved = await resumeTokens.GetLatestAsync(ServiceName, stoppingToken);
if (saved != null)
{
    options.ResumeAfter = saved;
}

using var cursor = await Collection.WatchAsync(BuildPipeline(), options, stoppingToken);
while (!stoppingToken.IsCancellationRequested
    && await cursor.MoveNextAsync(stoppingToken))
{
    foreach (var change in cursor.Current)
    {
        var partition = (GetEntityId(change).GetHashCode() & int.MaxValue) % workers;
        await partitions[partition].Writer.WriteAsync(change, stoppingToken);
        if (++processedSinceSave >= Settings.ResumeTokenSaveInterval)
        {
            await resumeTokens.SaveAsync(ServiceName, change.ResumeToken, stoppingToken);
            processedSinceSave = 0;
        }
    }
}
```

Догађаји се усмеравају ка N радника по хешу идентификатора ентитета, чиме се промене над истим ентитетом *унутар једног тока* обрађују редом, док различити ентитети иду упоредо. Међутим, у једну колекцију тренутног стања уписује *више* осматрача (нпр. листинг мире и пројектор жељеног и пројектор посматраног стања), који раде у одвојеним нитима; хеш-усмеравање их не серијализује међусобно. Да не би дошло до изгубљене измене (енг. *lost update*) при истовременом уписивању истог документа, помирење се додатно штити *закључавањем по идентификатору ентитета*: читање–измена–упис за дати ентитет извршава се атомично без обзира на то из ког тока долази. Ограничени редови (енг. *bounded channel*) обезбеђују контролу притиска: када су радници презаузети, произвођач се блокира и успорава читање тока.

Жетон се притом чува тек након што су сви догађаји до дате позиције

стварно обрађени (баријера која сачека да се редови испразне пре чувања жетона), чиме се очувава семантика испоруке „најмање једном“: после отказа се поново обрађује само прозор од последњег безбедног чувања, а идемпотентност чини поновну обраду безопасном.

Пројектори филтрирају ток серверски само на додавања (*append-only* дневници), што приказује листинг 4.2.

Листинг 4.2: Ток агрегација филтриран само на додавања

```
new EmptyPipelineDefinition<
       ChangeStreamDocument<ListingProprietaryState>>()
    .Match(change =>
       change.OperationType == ChangeStreamOperationType.Insert);
```

4.6 Помирење стања и поље са двоструком вредношћу

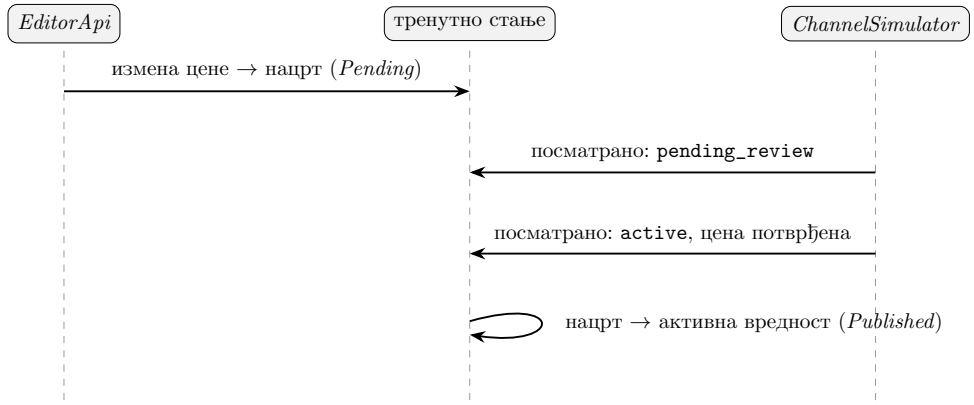
Функција помирења (енг. *state function*) редукује сваки догађај у документ тренутног стања. Жељена измена цене поставља *нацрт*; тек када канал (преко посматраног стања) потврди ту цену, нацрт се „објављује“ и постаје активна вредност (листинг 4.3). Тако се верно моделује разлика између намере трговца и стварног стања на каналу. Цела метода помирења посматраног стања, из које је овај исечак издвојен, дата је у додатку А.1.

Листинг 4.3: Објављивање нацрта при потврди са канала

```
if (state.EffectiveStatus == ChannelStatus.Active)
{
    if (current.Price.HasDraft && state.ObservedPrice.HasValue
        && current.Price.Draft == state.ObservedPrice.Value)
    {
        current.Price.Publish();
    }
}
```

Слика 4.3 приказује ток помирења: жељена измена постаје нацрт (статус *Pending*), канал је потврђује (преко посматраног стања), а функција помирења тада објављује нацрт, чиме статус прелази у *Published*.

Уписи су идемпотентни и заштићени провером временске ознаке: догађај старији од последњег примењеног се одбацује, чиме је обрада отпорна на догађаје ван редоследа и на поновну обраду при настављању тока.

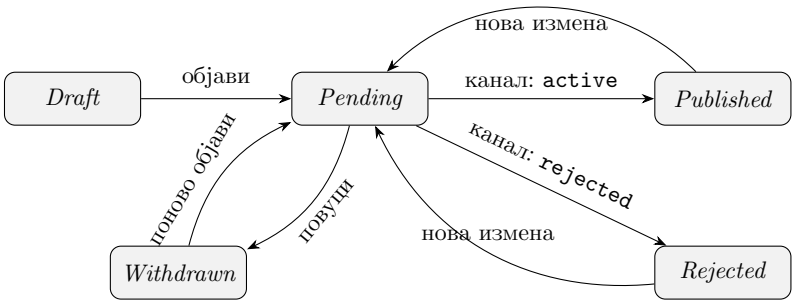


Слика 4.3: Помирење жељеног и посматраног стања.

Резултат помирења сажет је у пољу статуса објаве, чији су прелази приказани на слици 4.4. Листинг почиње као *Draft*; захтевом за објавом прелази у *Pending* (измена постоји као нацрт, али је канал још није потврдио); потврдом канала прелази у *Published*, а одбијањем у *Rejected*; повлачење или уклањање води у *Withdrawn*. Нова измена већ објављеног листинга поново ствара нацрт и враћа статус у *Pending*, чиме циклус „измена–потврда“ постаје видљив кроз једно поље.

Нацрт се може и одбацити пре него што га канал потврди: догађај `discardDraft` у дневнику жељеног стања поништава све нацрте листинга, а статус се тада поново изводи из последњег извештаја канала (потврђен листинг се враћа у *Published*, одбијен остаје *Rejected*). Одустајање, дакле, поништава само сопствену намеру; оно што је канал рекао остаје нетакнуто, па одбацивање нацрта нема јединствен циљни статус и није приказано као засебан прелаз на слици 4.4.

Повлачење је, као и објављивање, двострано: промена жеље у `withdrawn` одмах поставља статус *Withdrawn*, али је канал о томе тек потребно обавестити. Симулатор зато посматра и обрнути смер, то јест листинге чија је жеља `withdrawn` (или који су уклоњени), а које канал још увек пријављује као активне, и за њих генерише посматрано стање `paused`, чиме се ефективни статус усклађује са жељом. Повратак из *Withdrawn* захтева нову жељу за објавом, која статус враћа у *Pending* и тиме листинг поново уводи у циклус потврде; без тог прелаза листинг би остао изван обраде, јер симулатор бира кандидате управо по статусу *Pending*.



Слика 4.4: Прелази статуса објаве листинга.

4.7 Реконструкција стања (replay)

Пошто су дневници непроменљиви, тренутно стање је у потпуности изведива величина. Компонента `CatalogReplayer` брише материјализоване погледе и поново их гради пролазом кроз све догађаје у хронолошком редоследу, примењујући исту функцију помирења. Реконструкција је идемпотентна: поновна реконструкција даје идентично стање, што је потврђено интеграционим тестом (одељак 4.10).

4.8 Обавештавање у реалном времену

Емитери осматрају колекције тренутног стања (свих типова операција, уз слику пре промене), рачунају делту преко `EntityDeltaFactory` и шаљу је повезаним клијентима преко хаба *SignalR*. Контролна табла (једна страна у облику *HTML*) приказује живу табелу стања по каналима и нуди управљачке акције (измена цене и одустајање од измене, потврда и одбијање на каналу, повлачење и враћање у продају, реконструкција), чиме демонстрира узрочно-последични ток: измена → пут кроз токове промена → ажурирање приказа у реалном времену.

Табла омогућава и паузирање симулираног канала. Пошто симулатор потврђује тражене вредности већ у наредном циклусу (подразумевано након две секунде), нацрт по правилу постоји прекратко да би се над њим ручно демонстрирало одбијање. Са паузираним каналом листинг остаје у стању *Pending* неограничено дуго, па се сценарији дивергенције (одбијање измене, потврда различите вредности од тражене) могу извести контролисаним темпом, што је прикладно за приказ уживо.

Испорука је груписана: клијенти се придружују групи хаба *SignalR*, а емитер шаље делту само припадницима одговарајуће групе. У демонстратору табла прати све промене, али исти механизам омогућава да се, као у продукцији, промене усмере само заинтересованим клијентима (нпр. по продајном каналу или налогу), чиме се избегава непотребан саобраћај и сваки клијент прима искључиво податке који га се тичу. Последњи корак архитектуре тако пресликава груписање из теоријског прегледа (одељак 2.7) на конкретан домен е-трговине.

4.9 Репродуцибилност

Целокупан систем се подиже једном командом (`docker compose up`), која покреће *једночлани* скуп реплика базе *MongoDB*, то јест скуп реплика са само једним чвором. Такав скуп нема стварну репликацију, али има дневник операција, што је довољно јер *Change Streams* захтева скуп реплика, а не и више чворова (одељак 2.5). Уз базу се покреће и апликација. Демонстратор тако не зависи ни од чега из продукционе инфраструктуре.

Почетни каталог се учитава из *скупа података* у облику *CSV*, а не из кода: сваки ред описује једну варијанту (производ, опис, категорија, величина, боја, цена, залиха) и списак канала на којима се она излаже (колона `kanali`). Крајња тачка `POST /api/admin/seed?dataset=име` чита изабрану датотеку из фолдера `catalogs/` и уписује одговарајуће жељено стање, док `GET /api/admin/datasets` излистава доступне каталоге (контролна табла нуди њихов избор). Различити примери каталога, припремљени у програму за табеле и извезени у облику *CSV*, учитавају се тако без измене кода, што олакшава понављање сценарија над разним подацима.

4.10 Тестови

Исправност је проверена јединичним и интеграционим тестовима. Интеграциони тестови користе библиотеку *EphemeralMongo*, која подиже стваран једночлани скуп реплика, па се *Change Streams* и трансакције извршавају као у продукцији; тестови тако проверавају понашање, а не само логику у изолацији. Табела 4.4 сумира покривене случајеве. Сви тестови пролазе.

Табела 4.4: Покривеност тестовима демонстратора

Тест	Шта проверава
`DualStateField`	семантику активно/нацрт, објаву и одбацивање нацрта
Материјализација	да упис у дневник догађаја доводи до документа тренутног стања
Објава нацрта	да потврда са канала (посматрано стање) објављује одговарајући нацрт
Повлачење	да повлачење листинга (жеља `withdrawn`) поставља статус *Withdrawn*
Одустајање од измене	да одбацивање нацрта враћа стање које је канал последње потврдио
Обустава на каналу	да посматрано стање `paused` усклађује ефективни статус са повлачењем
Поновна објава	да жеља за објавом повучени листинг враћа у *Pending*
Ван редоследа	да старији догађај не регресира новије стање
Идемпотентност	да поновна примена истог догађаја не мења резултат
Каскада	да уклањање производа уклања варијанте и листинге
Реконструкција	да реконструкција из дневника догађаја даје идентично стање (и да је идемпотентна)
Настављање тока	да пројектор наставља од сачуваног жетона после рестарта, без дупликата

4.11 Одступања од продукционог система

Ради јасноће и самодовољности, демонстратор свесно упрошћава неке аспекте продукционог система; та одступања су наведена да поређење буде поштено:

Обавештавање без међуслоја. Емитери шаљу промене директно преко библиотеке *SignalR*, док продукција убацује посредника *Azure Service Bus* између емитера и те библиотеке. Међуслој доноси трајно задржавање порука у реду и одвајање испоруке од обраде (испорука преживљава рестарт клијентског слоја), али уводи додатну компоненту; демонстратор бира једноставнији пут јер је фокус на механизму токова промена.

Симулиран канал. Спољни продајни канал је симулиран, уместо да се користи прави интерфејс платформе: његову улогу преузима класа `ChannelSimulator` (додатак А.1), чиме је демонстрација детерминистичка и репродуцибилна.

Обим стања. Изостављена је грана „жељено стање са платформе“ (`meta-proprietary`) присутна у продукцији, јер не доприноси приказу основног механизма.

Задржавање. Демонстратор не примењује истицање дневника догађаја преко индекса *TTL*; у продукцији дневници истичу (нпр. 60 дана) ради контроле трошка складиштења, што тамо сужава прозор реконструкције (одељак 3.5).

Ниједно од ових одступања не мења суштину механизама (токови промена, жетон, помирење, реконструкција), који су пренети верно.

Глава 5

Вредновање и упоредна анализа

Ова глава вреднује решење засновано на *MongoDB Change Streams* и упоређује га са еквивалентним решењима која уместо тог механизма користе платформу *Apache Kafka* (уз *Debezium* као *CDC* слој), односно базу *EventStoreDB*. Поређење се води по кашњењу и протоку, скалабилности потрошача, гаранцијама испоруке, могућности реконструкције и оперативној цени решења.

5.1 Методологија мерења

Мерења су спроведена над демонстратором (глава 4), над скупом реплика базе *MongoDB* у контејнеру. Сва мерења изведена су на истој машини: *MacBook Pro* (2023) са процесором *Apple M3 Pro* (11 језгара: 5 учинковитих и 6 економичних), 18 GB радне меморије, под *macOS* 26.5 и *Docker Engine* 27.4. Пошто су све три технологије мерене на истом хардверу и у истом раду машине, апсолутне вредности зависе од конфигурације, али је поређење међу њима валидно. За сваки догађај мери се *кашњење материјализације*, то јест разлика између тренутка настанка догађаја жељеног стања (поље `LastProprietaryAt` у документу тренутног стања) и тренутка када га је функција помирења материјализовала (поље `UpdatedAt`). Пошто пројектор ради у истом процесу, обе временске ознаке долазе са истог часовника, па је кашњење мерено без утицаја анкетирања (енг. *polling*). Уз то, мери се и проток, укупан број догађаја обрађених у секунди, укључујући и секвенцијални упис на страни клијента. Под *клијентом* се овде подразумева сам мерни програм, то јест процес који уписује догађаје у *append-only* дневник и тиме покреће обраду. Он није део посматраног система него његов произвођач оптерећења. Мерење је аутома-

тизовано програмом `OmniChannel.Catalog.Bench`, а поставка и подешавања дати су у додатку А.2.

Свако мерење почиње *загревањем*: пре мереног дела обради се 500 догађаја чији се резултати одбацују. Тиме се из узорка уклањају почетна, „хладна“ мерења при успостављању тока, попуњавању кеша система за управљање базом (енг. *storage engine*), то јест подсистема који физички чува документе и индексе, и раду преводиоца у току извршавања (енг. *just-in-time compiler*), која иначе несразмерно подижу реп расподеле на малим узорцима. Након загревања свако оптерећење мери се три пута, а извештава се медијана та три мерења. Латенција се рачуна из временских ознака у документима, које *BSON* чува с тачношћу од једне милисекунде, па је то и резолуција мерења. Вредности медијане реда 1–2 ms треба читати као „на граници резолуције“, а не као тачне вредности.

Поред тога, спроведено је и *непосредно упоредно мерење* латенције испоруке над све три технологије (*MongoDB Change Streams*, *Apache Kafka*, *EventStoreDB*), програмом `OmniChannel.Catalog.LatencyProbe`. Мери се иста величина: време од производње догађаја (са утиснутом монотоном временском ознаком) до његовог пријема код претплаћеног потрошача. Произвођач и потрошач деле исти процес и исти монотони часовник (`Stopwatch`), то јест часовник који само напредује и на који не утичу накнадна усклађивања системског времена, па су резултати директно упоредиви и изолују путању испоруке. Догађаји се производе у равномерном режиму (са малим размаком) како би се измерила латенција по догађају, а не ефекат групне испоруке. За сваку технологију коришћена је конфигурација ниске латенције (нпр. подешавање `fetch.wait.max.ms` $= 10$ на платформи *Kafka*, аналогно подешавању `MaxAwaitTime` у бази *MongoDB*). Сва три система подигнута су локално у контејнерима, свако као једночлана инстанца, дакле као и скуп реплика из одељка 4.9, са једним чвором и без репликације (сценарији у фолдеру `comparison/`): `mongo:8.0`, `confluentinc/cp-kafka:7.7.1` уз `debezium/connect:2.7.3.Final` и `eventstore/eventstore:24.10.0-jammy`. Мерни програми су изграђени на *.NET 10*.

Радно оптерећење пресликано је из стварног продукционог понашања: из колекције жељеног стања огласа продукционог система (глава 3) анализиран је дводневни прозор са 252.912 забележених акција, па је из статуса уређивања и објаве (`DraftAdded`, `DraftUpdated`, `DraftRemoved`; `Published`,

`FailedToPublish`, `Deleted`) изведен однос операција: око 99,8 % измена, 0,18 % брисања и 0,01 % додавања. Мерење репродукује тај однос, а садржај догађаја чини узорак стварних докумената из истог прозора, просечне величине око 1,7 kB, идентичан за све три технологије. Пошто су ти документи интерни, у јавном репозиторијуму [2] налази се њихов анонимизован узорак: исте схеме, истог броја докумената и истоветне расподеле величина, са замењеним идентификаторима и називима, тако да мерење остаје поновљиво без излагања продукционих података.

Технологије се при том понашају различито: над базом *MongoDB* операције (додавање, измена, брисање) заиста се извршавају над колекцијом, а догађаји се *изводе* из тока промена; на платформи *Kafka* и у бази *EventStoreDB* еквивалентни догађаји се *експлицитно објављују*, јер те технологије саме по себи не виде стање базе.

Оваква поставка има и своја ограничења, која одређују домет изведених закључака. Све три технологије мерене су као једночлане, локалне инстанце на истој машини, па резултати не одражавају понашање у дистрибуираном, географски распоређеном распореду, где мрежно кашњење и репликација утичу на све три, али не нужно подједнако. Мерено је 20.000 догађаја по технологији у равномерном режиму, док се понашање под трајно високим оптерећењем (засићење) и у тренуцима наглог скока може разликовати. Предност платформе *Kafka* у протоку долази до изражаја тек при обимима већим од овде мерених, а утицај величине циљне колекције мерен је засебно (одељак 5.2). Свака технологија подешена је за ниску латенцију, али је простор подешавања велик, па би другачија подешавања (величине пакета, потврде, број партиција) дала другачије бројеве. Циљ није био пронаћи оптималну конфигурацију сваке технологије, него их упоредити под сличним подешавањима усмереним на ниску латенцију.

Ограничења носе и сами подаци и мерена величина. Догађаји носе стварне продукционе документе, али узорковане из ограниченог скупа (100 докумената из дводневног прозора), па би знатно већи документи (нпр. са угнежђеним метрикама) повећали трошак серијализације и преноса код све три технологије, мада не нужно подједнако. Пошто *BSON* временске ознаке чува с тачношћу од једне милисекунде, вредности медијане реда 1–2 ms леже на самој граници резолуције и не треба их читати као тачне, а максимум ($p100$) показао се нестабилним између поновљених мерења, због чега су закључци заснивани на

$p99$ као последњем перцентилу који је под овим условима поуздано упоредив. Најзад, у упоредном мерењу мерена је латенција *испоруке* (производња до пријема), а не пуна материјализација са сложеном функцијом помирења. Та обрада додаје кашњење које не зависи од избора транспортне технологије.

5.2 Резултати мерења

Табела 5.1 приказује измерене вредности за пет оптерећења, при чему највеће одговара обиму од 20.000 догађаја коришћеном у упоредном мерењу из одељка 5.2. Приказани су $p95$ и $p99$, јер за кашњење меродаван је реп расподеле, а не типичан случај. Медијана је на свим оптерећењима износила 1–2 ms, дакле на самој граници резолуције мерења.

Кашњење материјализације је ниско и стабилно: $p95$ износи 3 ms на свим оптерећењима од 1000 догађаја навише, а $p99$ остаје у опсегу 6–8 ms. Ни са двестоструким растом оптерећења, од 100 на 20.000 догађаја, реп расподеле се не помера. Благо виши $p95$ при само 100 догађаја (5 ms) последица је малог узорка, у коме и појединачно одступање мења перцентил. Проток се изравнава око 1000 догађаја у секунди. Он је овде ограничен секвенцијалним уписом на страни клијента, где се сваки догађај уписује засебно, а не самим током промена. Сам механизам осматрања при овим оптерећењима није био засићен, што је засебно проверено мерењем са више упоредних уписивача (одељак 5.2).

Табела 5.1: Кашњење материјализације кроз ток промена (демонстратор)

Оптерећење	$p95$ (ms)	$p99$ (ms)	Проток (дог/s)
100 догађаја	5,0	6,0	801
1000 догађаја	3,0	8,0	995
5000 догађаја	3,0	7,0	1029
10.000 догађаја	3,0	7,0	1044
20.000 догађаја	3,0	8,0	986

Упоредно мерење латенције испоруке

Табела 5.2 приказује измерену латенцију испоруке за све три технологије, под идентичном методологијом (одељак 5.1), преко 20.000 мерених догађаја по технологији, уз претходно загревање од 500 догађаја (медијане три поновљена

мерења, а по сваком мерењу 4 додавања, 19.960 измена и 36 брисања, у складу са продукционим односом).

Загревање овде готово ништа не мења: без њега су исте величине износиле 2,3, 4,3 и 7,1 ms за *MongoDB Change Streams*, што потврђује да на узорку од 20.000 догађаја неколико почетних, „хладних“ мерења (типичних за успостављање претплате и потрошача, енг. *cold start*) већ пада испод прага деведесет деветог перцентила. Управо зато је велики узорак и изабран. На неколико стотина догађаја исти би ефекат био видљив.

Табела 5.2: Латенција испоруке догађаја: непосредно поређење (једночлане инстанце, вредности у милисекундама)

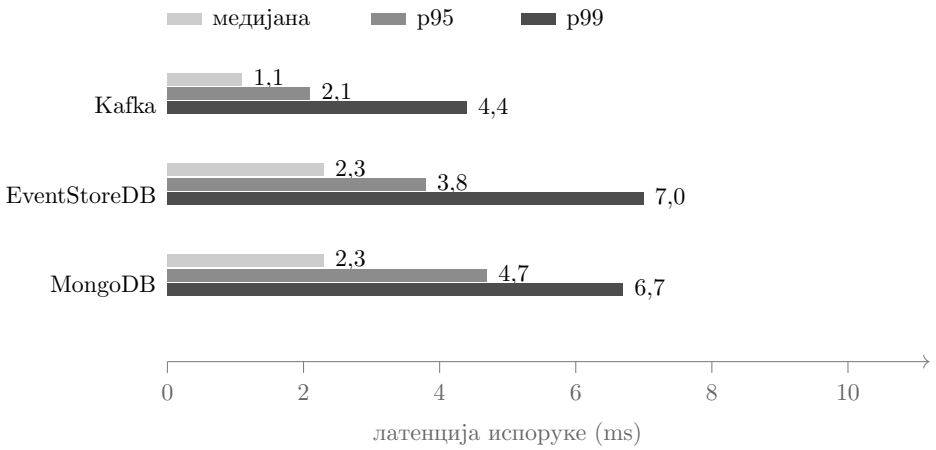
Технологија	Медијана	$p95$	$p99$	Максимум
Apache Kafka	1,1	2,1	4,4	17–115
EventStoreDB	2,3	3,8	7,0	112–558
MongoDB Change Streams	2,3	4,7	6,7	19–165

Медијана, $p95$ и $p99$ дати су као медијане три поновљена мерења. Максимум је, међутим, приказан као *распон* преко та три мерења, јер медијана трију екстрема нема јасно значење: свака гранична вредност у тој колони најгоре је кашњење виђено у једном пролазу од 20.000 догађаја.

Тако приказан, максимум не треба читати као својство технологије: за исту технологију он варира и за ред величине, што га чини мером застоја окружења (сакупљање отпада, распоређивање процеса), а не путање испоруке. Реп расподеле стабилизује се тек на $p99$, који је између поновљених мерења знатно постојанији: за *MongoDB Change Streams* износио је 5,5, 6,7 и 6,7 ms, а за базу *EventStoreDB* 6,9, 7,4 и 7,0 ms. Ни он, међутим, није потпуно стабилан: за платформу *Kafka* измерено је 2,8, 16,0 и 4,4 ms. Закључци који следе зато се заснивају на $p99$, али уз разлике мање од неколико милисекунди које не треба тумачити као поредак међу технологијама.

Слика 5.1 исте податке приказује графички, за медијану, $p95$ и $p99$. Максимум је изостављен јер би својим редом величине сажео остале три вредности у неупотребљиво уске ступце. Све три технологије испоручују догађаје у уском опсегу једноцифрених милисекунди, уз малу разлику између медијане и $p99$. Платформа *Kafka* је најбржа по свим трима величинама, што је очекивано за наменски пројектован транспорт догађаја. *MongoDB Change Streams* и база *EventStoreDB* практично се не разликују: медијана им је иста (2,3 ms),

$p95$ је нешто нижи код базе *EventStoreDB* (3,8 наспрам 4,7 ms), а $p99$ нешто нижи код механизма *Change Streams* (6,7 наспрам 7,0 ms). Другим речима, механизам уграђен у нерелациону базу опште намене мери се са наменским складиштем догађаја, при чему то кашњење укључује и потврду трајности догађаја на нивоу већине (*majority-commit*), о чему детаљније ниже у тексту.



Слика 5.1: Латенција испоруке по технологији: медијана, $p95$ и $p99$, у милисекундама (мање је боље).

Из мерења не следи да је једна од технологија просто „најбржа“. Све три испоручују догађаје у једноцифреним милисекундама, а за домен е-трговине разлика од неколико милисекунди практично је неприметна крајњем кориснику, будући да мрежни пут до прегледача уноси знатно веће кашњење. Платформа *Kafka* задржава најнижу медијану, што је очекивано за наменски пројектован транспорт догађаја, али са стварним продукционим документима као садржајем разлика у односу на *Change Streams* пада испод једне милисекунде.

MongoDB Change Streams испоручује догађаје са медијаном испод 2 ms и уском, предвидивом расподелом ($p99$ око 5 ms), што је нижа медијана и практично исти реп као код наменског складишта догађаја, базе *EventStoreDB*. То кашњење притом *укључује гаранцију трајности*: ток промена испоручује догађај тек када је он потврђен на нивоу већине (*majority-committed*) и више не може бити поништен. Иста гаранција омогућава поуздано настављање преко жетона и рад без лажних догађаја. У домену е-трговине, где догађаји носе цене, залихе и статусе објаве, то је предност: систем не реагује на измену ко-

ја би накнадно могла бити поништена, па не долази до недоследности попут приказане цене која не одговара потврђеној на каналу.

Та се латенција постиже без иједне додатне компоненте, поновном употребом базе која у систему већ постоји. *CDC* пут платформе *Kafka* (*Debezium*) и сам интерно користи *Change Streams* као извор промена (одељак 5.3), па би избор те платформе значио слагање брокера и конектора *поврх* истог механизма који је овде измерен. За системе који већ користе *MongoDB* то значи поуздану испоруку уз гаранцију трајности, у једноцифреним милисекундама и без додатног дистрибуираног система.

Проток

Мерење материјализације (табела 5.1) показало је проток реда 800–1050 догађаја у секунди при секвенцијалном упису. Проток је ту ограничен *клијентом*, јер је сваки догађај уписиван засебно и синхроно, а не механизмом токова промена. Да та тврдња не би остала само образложење, мерење је поновљено са више упоредних уписивача (режим `parallel` истог програма): исти број догађаја уписује N нити, свака над сопственим низом ентитета. Мере се две величине одвојено, брзина самог уписа и брзина материјализације кроз ток промена, јер тек њихова разлика показује која страна ограничава проток.

Табела 5.3: Проток у зависности од броја упоредних уписивача, уз осам радника пројектора. Проток је дат у догађајима у секунди, кашњење у милисекундама.

Уписивача	Упис	Материјализација	$p95$	$p99$
1	1042	1042	3	7
2	1452	1452	7	13
4	2226	2225	30	45
8	3720	2226	3255	3334
16	5844	2269	5250	5401
32	9223	2210	6619	6858

Проток уписа расте са 1042 на 9223 догађаја у секунди, што потврђује да је граница из табеле 5.1 заиста била на страни клијента. Материјализација, међутим, престаје да прати упис изнад четири уписивача и зауставља се на око 2200 догађаја у секунди. Од те тачке ограничени редови (одељак 4.5) спроводе

контролу притиска: догађаји чекају у реду, ниједан се не губи и меморија не расте, али кашњење материјализације расте на неколико секунди.

Тај плафон није својство механизма токова промена него подешавања демонстратора. Пројектор подразумевано користи осам радника, а над сваким догађајем изводи претрагу и упис уз закључавање по кључу. Табела 5.4 показује шта се дешава када се број радника повећа, уз фиксирана 32 уписивача.

Табела 5.4: Проток у зависности од броја радника пројектора, уз 32 упоредна уписивача. Проток је дат у догађајима у секунди, кашњење у милисекундама.

Радника	Упис	Материјализација	$p95$	$p99$
8	8723	2228	6512	6726
16	6726	2612	4565	4682
32	5760	3238	2668	2767
64	5390	3877	1542	1557
128	4356	3671	933	959
256	4057	4057	68	90

Материјализација расте са 2228 на 4057 догађаја у секунди, а при 256 радника изједначава се са уписом: пројектор тада више није уско грло, ништа се не гомила у реду и $p99$ пада са 6726 на 90 ms. Граница се, дакле, помера подешавањем на нивоу апликације, што је у складу са запажањем из одељка 2.6 да се код механизма *Change Streams* хоризонтално скалирање потрошача решава у апликацији. Преостали проток од око 4000 догађаја у секунди не одређује више пројектор него сама мерна машина.

Уз ово мерење иду две ограде. Прво, проток уписа опада како расте број радника, са 8723 на 4057 догађаја у секунди, јер уписивачи, пројектор и једночлана база деле исти процесор и исту инстанцу, а у стварном распореду ти послови живе на различитим чворовима, па тог надметања нема. Друго, у мерењу сваки догађај погађа различит ентитет, па нема надметања око закључавања по кључу. То је најповољнији случај за паралелизам: код оптерећења у коме се исти ентитет често мења, серијализација по кључу ограничила би корисни паралелизам без обзира на број радника.

Продукциони систем (глава 3), у коме више произвођача истовремено уписује догађаје, потврђује да стварни проток вишеструко превазилази ове бројеве: колекције жељеног и посматраног стања броје десетине милиона докумената, а систем их материјализује без засебног брокера.

Проток је, с друге стране, тачка у којој платформа *Kafka* има архитектонску предност: партиционисање и потрошачке групе дају проток и хоризонтално скалирање потрошача који превазилазе оно што један скуп реплика нуди. Капацитет скупа реплика није јединствен број, јер зависи од хардвера, величине докумената, броја индекса и режима потврђивања уписа, па га за конкретну поставку треба одредити мерењем. Ред величине се, међутим, може проценити из продукционих података из главе 3: тамошњи дневник операција од 47 GB покрива прозор реда неколико сати, што при просечној величини записа од око 1,7 kB одговара трајном протоку реда неколико хиљада уписа у секунди, и то са резервом, јер прозор није попуњен до границе. Тај систем над скупом реплика од три члана опслужује обим описан у одељку 3.1, па за обиме типичне за е-трговину капацитет једног скупа реплика по правилу није ограничење. Ако захтеви прелазе тај ред величине, партиционисање платформе *Kafka* постаје одлучујућа предност.

Латенција при великом обиму колекције

Мерења из табеле 5.1 спроведена су над релативно малим колекцијама, што оставља отвореним да ли кашњење материјализације расте када циљна колекција нарасте до продукционих размера. Ради провере, колекција тренутног стања унапред је напуњена са **10 милиона** докумената (око 4,4 GB логичке величине), након чега је истом методологијом (одељак 5.1) измерено кашњење материјализације низа од 2000 нових догађаја над тако великом колекцијом.

Пошто редослед мерења сам по себи може да утиче на резултат, мерење је постављено тако да се тај утицај искључи. Изведено је укупно три пута: над празном колекцијом пре пуњења, над напуњеном колекцијом, и поново над празном колекцијом пошто је велика колекција испражњена. Свакој серији претходи загревање, а свака се понавља три пута. Ако би редослед био узрок разлике, контролно мерење на крају одступало би од основног на почетку. Табела 5.5 приказује сва три резултата.

Кашњење је остало непромењено: све три серије дају исте вредности у оквиру резолуције мерења. Кључно је да се контролно мерење над празном колекцијом, изведено *после* мерења над десет милиона докумената, не разликује од основног изведеног пре пуњења. Редослед мерења, дакле, није извор разлике, па се измерена једнакост може приписати самом систему, а не за-

Табела 5.5: Кашњење материјализације у зависности од величине циљне колекције

Величина циљне колекције	Медијана (ms)	$p95$ (ms)	$p99$ (ms)
празна (основа, пре пуњења)	2,0	3,0	7,0
10.000.000 докумената	2,0	3,0	7,0
празна (контрола, после пражњења)	1,0	3,0	8,0

грејаности процеса или кеша. Да величина колекције не утиче на кашњење има и архитектонско објашњење. Функција помирења над сваким догађајем изводи претрагу по идентификатору ентитета (`Find` по јединственом индексу над пољем `entityId`) и идемпотентан `upsert`, обоје сложености $O(\log n)$. Јединствени индекс над 10 милиона докумената и даље је плитак и стаје у кеш, па је појединачна измена једнако брза као над празном базом. Уз то, осматрање тока промена чита дневник операција независно од величине циљне колекције. Мерење је у складу са продукционом студијом (глава 3), у којој колекције тренутног стања броје милионе докумената.

5.3 Упоредна анализа

Табела 5.6 сажима квалитативно поређење три технологије кроз призму изградње архитектуре вођене догађајима.

Из табеле се види да разлика међу технологијама није у томе шта се може постићи, него по коју цену. Све три нуде испоруку у једноцифреним милисекундама и реконструкцију стања из трајног записа, али се разилазе у оперативном терету: *Change Streams* не уводи ниједну компоненту поред базе која у систему већ постоји, док платформа *Kafka* и база *EventStoreDB* захтевају засебан кластер о ком се надаље води брига. Заузврат, платформа *Kafka* једина доноси уграђено хоризонтално скалирање потрошача, а база *EventStoreDB* једина третира догађај као примарни ентитет. Наредни пододељци разлажу те разлике по појединачним димензијама.

Кашњење и проток

Непосредно мерење (табела 5.2) показује да *Change Streams* испоручује промене са медијаном од 2,3 ms и $p99$ испод 7 ms, при чему та вредност укљу-

Табела 5.6: Поређење технологија за архитектуру вођену догађајима

	MongoDB Change Streams	*Kafka* (уз *Debezium*)	*EventStoreDB*
Улога	*CDC* уграђен у базу	трајни дневник и *CDC* конектори	наменско складиште догађаја
Додатна компонента	нема (користи постојећу базу)	засебан кластер	засебна база
Задржавање догађаја	подесиво (*TTL* по избору)	подесиво, дугорочно	трајно
Гаранције	*at-least-once*, редослед по кључу	*at-least-once*, редослед по партицији	строг редослед по току
Скалирање потрошача	ручно (партиционисање у апликацији)	уграђено (партиције, групе)	претплате, пројекције
Реконструкција	из *append-only* дневника, док се чува	у оквиру задржавања дневника	изворно (`$all`)
Одржавање	најмање (само база)	највише	средње
Крива учења	блага (ако се већ користи *MongoDB*)	стрма	средња

чује потврду трајности догађаја (одељак 5.2). Најнижу сирову латенцију има платформа *Kafka*, док се база *EventStoreDB* и *Change Streams* практично не разликују. Разлика у протоку (одељак 5.2) плаћа се додатном мрежном путањом произвођач–брокер–потрошач и компонентама које треба одржавати.

Скалабилност

Предност је на страни платформе *Kafka*: партиције и потрошачке групе дају изворно хоризонтално скалирање потрошача. Код механизма *Change Streams* паралелизам се постиже на нивоу апликације (у демонстратору усмеравањем по хешу идентификатора ентитета на више радника), што захтева пажљив дизајн, али је изводљиво: мерење из одељка 5.2 показује да се проток материјализације подиже самим повећањем броја радника, а продукциони обим (глава 3) да тај приступ носи и знатно веће системе.

Реконструкција и задржавање

Концептуална разлика тиче се два одвојена прозора. Први је прозор дневника операција, који ограничава *настављање тока* преко жетона; он је подесив (одељак 2.5), али увек коначан. Други је задржавање самог дневника догађаја (*append-only* колекција), а оно је ствар избора, не технологије: без индекса *TTL* дневник се чува трајно, па је реконструкција од почетка времена могућа читањем колекције, независно од дневника операција, што демонстратор и показује. База *EventStoreDB* и (уз одговарајуће задржавање) платформа *Kafka* трајан дневник нуде као подразумевани режим; у бази *MongoDB* трајно задржавање захтева свесну одлуку о трошку складиштења, а продукциони систем из главе 3 определио се за *TTL* од 60 дана, чиме је прозор реконструкције тамо сужен.

Уз временски прозор постоји и друго ограничење реконструкције, независно од технологије: реконструкција је могућа само док текући код уме да прочита старе записе. Промена облика догађаја захтева или превођење историје или задржавање подршке за старије верзије, што је у литератури издвојено као један од најтежих проблема система заснованих на догађајима [28]. Демонстратор и продукциони систем ту држе исти, најједноставнији приступ: схема записа стања се проширује само додавањем поља са подразумеваним вредностима, чиме стари записи остају читљиви без превођења.

Једноставност одржавања

Решење над механизмом *Change Streams* не уводи ни један нови дистрибуирани систем у апликативни стек (енг. *application stack*). Тим који већ одржава скуп реплика базе *MongoDB* добија архитектуру вођену догађајима без нове оперативне површине, а са њом и без пратећег трошка и ризика.

Када су Change Streams прави избор

MongoDB Change Streams је оправдан избор када: (1) *MongoDB* већ постоји као примарно складиште; (2) обим и захтеви за протоком су у оквиру капацитета скупа реплика (што, као што показује глава 3, укључује и веома велике системе); и (3) дневник догађаја може да живи у самој бази, трајно или уз *TTL* по избору. Засебан брокер или складиште догађаја оправданији су када проток или број независних потрошачких група прелазе оно што један скуп

реплика подноси. Исто важи када догађаје треба размењивати преко граница више база и тимова, или када је потребно поновно читање историје као тока, са уграђеним претплатама.

Упркос ограничењима мерне поставке (одељак 5.1), главни налаз остаје недвосмислен: латенција механизма *MongoDB Change Streams* истог је реда величине као код наменских технологија, а решење за то не тражи никакву додатну инфраструктуру.

Глава 6

Закључак

Рад је испитао применљивост механизма *MongoDB Change Streams* као основе архитектуре вођене догађајима у микросервисном окружењу е-трговине. Полазећи од продукционог система који у обиму од преко 511 GB и стотина милиона докумената функционише само над механизмом *Change Streams*, извучен је методолошки образац и пренет у демонстратор на домену е-трговине.

6.1 Доприноси

Потврђено је да раздвајање стања на жељено, посматрано и тренутно, у комбинацији са двостепеном каскадом токова промена, чини јасан и поновљив методолошки приступ изградњи реактивних система над базом *MongoDB*. Показано је како се жетоном настављања постиже поуздано настављање и семантика испоруке „најмање једном“, те како се уз идемпотентне, временски заштићене функције помирења и *append-only* дневнике омогућава детерминистичка реконструкција стања. Упоредна анализа наспрам платформе *Apache Kafka* и базе *EventStoreDB* идентификовала је услове под којима је сваки приступ оправдан.

Као препоруке за дизајн истичу се: усмеравање догађаја по идентификатору ентитета ради редоследа по кључу уз паралелизам међу ентитетима; ограничени редови ради контроле притиска; идемпотентни уписи са провером временске ознаке; и експлицитно руковање ротацијом дневника операција као редовним, а не изузетним сценаријем опоравка.

6.2 Ограничења

Главно ограничење тиче се настављања тока: жетон важи само у оквиру прозора дневника операција, који је подесив, али увек коначан. Задржавање самог дневника догађаја, с друге стране, ствар је избора: *append-only* колекције могу се чувати и трајно, а продукциони систем их ограничава индексима *TTL* ради контроле трошка складиштења, чиме сужава прозор реконструкције. Реконструкција је ограничена и обликом записаних догађаја: промена схеме захтева или превођење историје или задржавање подршке за старије верзије [28], што демонстратор не решава него избегава, дозвољавајући само додавање поља. Скалирање потрошача захтева дизајн на нивоу апликације, за разлику од уграђеног партиционисања платформе *Kafka*. Демонстратор, ради јасноће, поједностављује неке аспекте продукционог система (нпр. изостаје међуслој посредника *Azure Service Bus* између емитера и библиотеке *SignalR*).

6.3 Правци даљег рада

Правци даљег рада укључују: аутоматизовано хоризонтално скалирање осматрача уз координацију преузимања партиција (нпр. закупом власништва над опсезима идентификатора); интеграцију трајног дневника (архивирањем истеклих догађаја у јефтиније складиште) за системе који се определе за задржавање преко индекса *TTL*; проширење упоредне анализе мерењима под високим оптерећењем и у дистрибуираном, географски распоређеном скупу реплика; те увођење међуслоја за поруке и мерење његовог утицаја на трајност испоруке насупрот додатој латенцији.

Отворен је и хибридни приступ: задржати *MongoDB Change Streams* као примарни механизам обраде уз базу, а на његовом излазу прикачити брокер тек тамо где су потребни трајан дневник или бројне независне потрошачке групе, чиме би се спојиле једноставност једног и скалабилност другог приступа.

Рад показује да *MongoDB Change Streams* може да послужи као језгро архитектуре вођене догађајима, иако база није првенствено пројектована као брокер. Предност је највећа тамо где се уз латенцију вреднује и укупна оперативна цена решења.

Додатак А

Изворни код и конфигурација демонстратора

Овај додатак садржи изворни код најважнијих компоненти демонстратора и његова кључна подешавања, ради потпуности и поновљивости.

А.1 Изворни код кључних компоненти

Наведени су само делови који су суштински за механизме описане у раду; целовит пројекат доступан је у јавном репозиторијуму [2].

Петља опоравка осматрача тока промена

Наредни листинг приказује спољну петљу генеричке основе `ChangeStream Processor`. Она разликује три исхода: уредан прекид, неважећи жетон (при ротацији дневника операција жетон се брише и креће се из садашњости) и опорављиву грешку (поновно повезивање уз ограничен, растући застој). Овим је осматрач отпоран на прекиде везе и промене примарног чвора, а истовремено се штити од бесконачног понављања.

Листинг А.1: Петља опоравка и класификација грешака

```
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    var attempts = 0;
    while (!stoppingToken.IsCancellationRequested)
    {
        try
        {
```

```
            await RunAsync(stoppingToken);
            attempts = 0;
        }
        catch (OperationCanceledException) when (stoppingToken.IsCancellationRequested)
        {
            break;
        }
        catch (Exception ex) when (IsResumeTokenInvalid(ex))
        {
            await resumeTokens.DeleteAsync(ServiceName, CancellationToken.None);
            attempts = 0;
        }
        catch (Exception ex) when (IsRecoverable(ex))
        {
            attempts++;
            if (attempts > Settings.MaxReconnectAttempts)
            {
                throw;
            }
            await Task.Delay(Settings.ReconnectDelayMs * attempts, stoppingToken);
        }
    }
}

private static bool IsRecoverable(Exception ex) =>
    ex is MongoConnectionException or MongoExecutionTimeoutException
        or MongoNotPrimaryException or MongoNodeIsRecoveringException
        or MongoCursorNotFoundException or TimeoutException
    || (ex.InnerException != null && IsRecoverable(ex.InnerException));

private static bool IsResumeTokenInvalid(Exception ex)
{
    if (ex is MongoCommandException command && command.Code is 286 or 260)
    {
        return true;
    }
    return ex.Message.Contains("resume point may no longer be in the oplog",
            StringComparison.OrdinalIgnoreCase)
        || (ex.InnerException != null && IsResumeTokenInvalid(ex.InnerException));
}
```

Складиште жетона

Жетони се чувају по имену сервиса, уз `upsert`, тако да сваки осматрач има тачно један ред који се ажурира. Читање враћа последњи сачувани жетон, а брисање се користи при ротацији дневника операција.

Листинг А.2: ResumeTokenRepository

```
public async Task SaveAsync(string serviceName, BsonDocument token, CancellationToken ct)
{
    var filter = Builders<ResumeToken>.Filter.Eq(t => t.ServiceName, serviceName);
    var update = Builders<ResumeToken>.Update
        .Set(t => t.ServiceName, serviceName)
        .Set(t => t.Token, token)
        .Set(t => t.Timestamp, DateTime.UtcNow)
        .SetOnInsert(t => t.Id, ObjectId.GenerateNewId());
    await _collection.UpdateOneAsync(filter, update,
        new UpdateOptions { IsUpsert = true }, ct);
}

public async Task<BsonDocument?> GetLatestAsync(string serviceName, CancellationToken ct)
{
    var token = await _collection.Find(t => t.ServiceName == serviceName)
        .SortByDescending(t => t.Timestamp).FirstOrDefaultAsync(ct);
    return token?.Token;
}
```

Помирење посматраног стања

Функција помирења посматраног стања ажурира ефективни статус и објављује нацрте чије вредности је канал потврдио, уз заштиту од догађаја ван редоследа временском ознаком.

Листинг А.3: ListingCurrentStateRepository.ApplyObservedAsync (сржж)

```
if (current.LastObservedAt.HasValue && state.ObservedAt < current.LastObservedAt.Value)
{
    return;
}
current.EffectiveStatus = state.EffectiveStatus;
current.ObservedPrice = state.ObservedPrice;
current.LastObservedAt = state.ObservedAt;
if (state.EffectiveStatus == ChannelStatus.Active)
{
    if (current.Price.HasDraft && state.ObservedPrice.HasValue
        && current.Price.Draft == state.ObservedPrice.Value)
    {
        current.Price.Publish();
    }
    if (current.Available.HasDraft && state.Available.HasValue
        && current.Available.Draft == state.Available.Value)
    {
        current.Available.Publish();
    }
}
current.PublishStatus = MapPublishStatus(state.EffectiveStatus, current);
```

```
22 await _listings.ReplaceOneAsync(x => x.EntityId == state.EntityId, current,
23     new ReplaceOptions { IsUpsert = true }, cancellationToken);
```

Симулатор канала

Симулатор проналази листинге чије жељено стање канал још није потврдио и генерише посматрано стање. Повучени (или уклоњени) листинзи које канал још пријављује као активне добијају статус `paused`; при првом контакту непознат канал враћа `pending_review`, а затим потврђује тражене вредности статусом `active`.

Листинг А.4: ChannelSimulator: генерисање посматраног стања

```
1  private static ListingObservedState BuildObservation(ListingCurrentState listing)
2  {
3      var now = DateTime.UtcNow;
4      if (listing.Removed || listing.DesiredStatus == PublishStatus.Withdrawn)
5      {
6          return new ListingObservedState
7          {
8              EntityId = listing.EntityId, Channel = listing.Channel,
9              EffectiveStatus = ChannelStatus.Paused,
10             ObservedAt = now, CreatedAt = now
11         };
12     }
13     var firstContact =
14         listing.LastObservedAt == null && listing.Channel != SalesChannel.Webshop;
15     if (firstContact)
16     {
17         return new ListingObservedState
18         {
19             EntityId = listing.EntityId, Channel = listing.Channel,
20             EffectiveStatus = ChannelStatus.PendingReview,
21             ObservedAt = now, CreatedAt = now
22         };
23     }
24     return new ListingObservedState
25     {
26         EntityId = listing.EntityId, Channel = listing.Channel,
27         EffectiveStatus = ChannelStatus.Active,
28         ObservedPrice = listing.Price.Effective,
29         Available = listing.Available.Effective,
30         ObservedAt = now, CreatedAt = now
31     };
32 }
```

А.2 Конфигурација и покретање

Компоновање контејнера

Демонстратор се подиже једном командом (`docker compose up`). Датотека за компоновање подиже једночлани скуп реплика (обавезан за *Change Streams*), чија провера доступности (енг. *healthcheck*) истовремено иницијализује скуп реплика, и апликациони процес који се покреће тек када је база доступна.

Листинг А.5: `docker-compose.yml` (срж)

```
services:
  mongo:
    image: mongo:8.0
    command: [''mongod'', ''--replSet'', ''rs0'', ''--bind_ip_all'']
    ports: [''27018:27017'']
    healthcheck:
      test: [''CMD'',''mongosh'',''--quiet'',''--eval'',
        ''try { rs.status().ok } catch (e) { rs.initiate().ok }'']
      interval: 5s
      retries: 30
  host:
    build: { context: ., dockerfile: src/OmniChannel.Catalog.Host/Dockerfile }
    ports: [''8080:8080'']
    environment:
      - MongoDb__ConnectionString=mongodb://mongo:27017/?replicaSet=rs0&directConnection=
          true
    depends_on:
      mongo: { condition: service_healthy }
```

Кључна подешавања

Понашање осматрача подешава се преко класе `MongoDbSettings` (табела А.1). Подразумеване вредности одабране су за ниску латенцију уз разумну контролу притиска.

Осматрачи усмеравају догађаје на `ParallelWorkers` радника по хешу идентификатора ентитета: тиме је редослед по кључу очуван (промене истог ентитета иду истом раднику), док различити ентитети напредују упоредо. Повећање броја радника подиже пропусност уз задржавање гаранције редоследа по ентитету.

Табела А.1: Кључни параметри осматрача (класа `MongoDbSettings`)

Параметар	Подразум.	Улога
`BatchSize`	1000	максимална величина пакета при читању тока
`MaxAwaitTimeMs`	1000	најдуже чекање на нове догађаје (одзивност)
`ChannelCapacity`	1000	капацитет ограниченог реда (контрола притиска)
`ParallelWorkers`	8	број радника (паралелизам међу ентитетима)
`ResumeTokenSaveInterval`	50	на колико догађаја се чува жетон
`MaxReconnectAttempts`	5	број покушаја поновног повезивања
`ReconnectDelayMs`	2000	основни застој између покушаја (растући)

Литература

[1] Apache Software Foundation. Apache kafka documentation, 2024. https://kafka.apache.org/documentation/.

[2] Miljan Bakić. OmniChannel.Catalog — демонстратор уз мастер рад. Извор отвореног кода, лиценца *MIT*, 2026. https://github.com/miljanb99/omnichannel-catalog.

[3] Debezium Community. Debezium documentation, 2024. https://debezium.io/documentation/.

[4] Event Store Ltd. Eventstoredb documentation, 2024. https://developers.eventstore.com/.

[5] Martin Fowler. Event sourcing, 2005. https://martinfowler.com/eaaDev/EventSourcing.html.

[6] Martin Fowler. Cqrs, 2011. https://martinfowler.com/bliki/CQRS.html.

[7] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[8] Pat Helland. Immutability changes everything. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[9] HunchAds. meta-campaign-events-processor. Интерни репозиторијум, *Azure DevOps* (приступ ограничен). https://dev.azure.com/HunchAds/manage/_git/meta-campaign-events-processor.

[10] HunchAds. meta-campaign-tracking-service. Интерни репозиторијум, *Azure DevOps* (приступ ограничен). https://dev.azure.com/HunchAds/manage/_git/meta-campaign-tracking-service.

[11] HunchAds. snapchat-campaign-events-processor. Интерни репозиторијум, *Azure DevOps* (приступ ограничен). `https://dev.azure.com/HunchAds/manage/_git/snapchat-campaign-events-processor`.

[12] HunchAds. snapchat-campaign-tracking-service. Интерни репозиторијум, *Azure DevOps* (приступ ограничен). `https://dev.azure.com/HunchAds/manage/_git/snapchat-campaign-tracking-service`.

[13] HunchAds. tiktok-campaign-events-processor. Интерни репозиторијум, *Azure DevOps* (приступ ограничен). `https://dev.azure.com/HunchAds/manage/_git/tiktok-campaign-events-processor`.

[14] HunchAds. tiktok-campaign-tracking-service. Интерни репозиторијум, *Azure DevOps* (приступ ограничен). `https://dev.azure.com/HunchAds/manage/_git/tiktok-campaign-tracking-service`.

[15] Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

[16] Martin Kleppmann and Jay Kreps. Kafka, samza and the unix philosophy of distributed data. *IEEE Data Engineering Bulletin*, 38(4):4–14, 2015.

[17] Jay Kreps. The log: What every software engineer should know about real-time data's unifying abstraction. LinkedIn Engineering, 2013. Изворна адреса више није доступна. Архивирана копија: `https://web.archive.org/web/2026/https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying`.

[18] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB)*, Athens, Greece, 2011.

[19] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[20] Microsoft. Overview of asp.net core signalr, 2024. `https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction`.

[21] Microsoft. What is azure service bus?, 2024. `https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview`.

[22] MongoDB, Inc. Change streams — mongodb manual, 2024. `https://www.mongodb.com/docs/manual/changeStreams/`.

[23] MongoDB, Inc. Mongodb manual — databases and collections, 2024. `https://www.mongodb.com/docs/manual/core/databases-and-collections/`.

[24] MongoDB, Inc. Replication — mongodb manual, 2024. `https://www.mongodb.com/docs/manual/replication/`.

[25] MongoDB, Inc. Sharding — mongodb manual, 2024. `https://www.mongodb.com/docs/manual/sharding/`.

[26] MongoDB, Inc. Ttl indexes — mongodb manual, 2024. `https://www.mongodb.com/docs/manual/core/index-ttl/`.

[27] Sam Newman. *Building Microservices*. O'Reilly Media, 2 edition, 2021.

[28] Michiel Overeem, Marten Spoor, and Slinger Jansen. The dark side of event sourcing: Managing data conversion. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, page 193–204. IEEE, 2017.

[29] Michiel Overeem, Marten Spoor, Slinger Jansen, and Sjaak Brinkkemper. An empirical characterization of event sourced systems and their schema evolution — lessons from industry. *Journal of Systems and Software*, 178:110970, 2021.

[30] Chris Richardson. *Microservices Patterns: With Examples in Java*. Manning Publications, 2018.

[31] Ben Stopford. *Designing Event-Driven Systems*. O'Reilly Media, 2018.

[32] Hugh Taylor, Angela Yochem, Les Phillips, and Frank Martinez. *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise*. Addison-Wesley, 2009.

[33] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

Биографија аутора

Миљан Бакић рођен је 1999. године у Аранђеловцу. Дипломирао је 2022. године на Математичком факултету Универзитета у Београду, где је потом уписао мастер студије. Професионално се бави развојем софтвера у *C#* и *Node.js* технологијама на платформи *Azure*, радећи на дистрибуираним системима за обраду података у реалном времену, међу којима је и систем заснован на технологији *MongoDB Change Streams* који је у овом раду послужио као студија случаја.