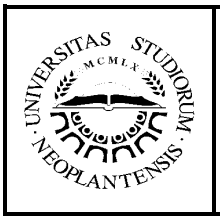


УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
НОВИ САД
Департман за рачунарство и аутоматику
Одсек за рачунарску технику и рачунарске комуникације

ЗАВРШНИ (BACHELOR) РАД

Кандидат: **Марко Кувизић**
Број индекса: **SV 38/2021**

Тема рада: **Секвенцијални оркестратор за спекулативно спајање промена софтвера у систему континуалне интеграције**

Ментор рада: **Проф. Др Мирослав Поповић**

Нови Сад, Новембар 2025.

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографска документација
Тип записа, **ТЗ**:		Текстуални штампани материјал
Врста рада, **ВР**:		Завршни (Bachelor) рад
Аутор, **АУ**:		**Марко Кувизић**
Ментор, **МН**:		**Проф. Др Мирослав Поповић**
Наслов рада, **НР**:		**Секвенцијални оркестратор за спекулативно спајање промена софтвера у систему континуалне интеграције**
Језик публикације, **ЈП**:		Српски / латиница
Језик извода, **ЈИ**:		Српски
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		**2025**
Издавач, **ИЗ**:		Ауторски репринт
Место и адреса, **МА**:		Нови Сад; трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		
Научна област, **НО**:		Електротехника и рачунарство
Научна дисциплина, **НД**:		Рачунарска техника
Предметна одредница/Кqучне речи, **ПО**:		
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		**У овом раду представљен је пројекат алгоритма за спекулативно спајање у оквиру система континуалне интеграције. Развијен је систем за динамичко састављање алгоритма из независних команди, као и распоређивач који управља њиховим извршавањем. Приказане су метрике перформанси и успешности система, уз одговарајућу анализу резултата. На крају, разматрани су могући правци даљег развоја и имплементације предложеног решења.**
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	доц. др. Миодраг Ђукић
	Члан:	проф. др. Игор Дејановић — Потпис ментора
	Члан, ментор:	проф. др. Мирослав Поповић

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:			
Identification number, **INO**:			
Document type, **DT**:		Monographic publication	
Type of record, **TR**:		Textual printed material	
Contents code, **CC**:		Bachelor Thesis	
Author, **AU**:			
Mentor, **MN**:			
Title, **TI**:			
Language of text, **LT**:		Serbian	
Language of abstract, **LA**:		Serbian	
Country of publication, **CP**:		Republic of Serbia	
Locality of publication, **LP**:		Vojvodina	
Publication year, **PY**:			
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)			
Scientific field, **SF**:		Electrical Engineering	
Scientific discipline, **SD**:		Computer Engineering, Engineering of Computer Based Systems	
Subject/Key words, **S**/**KW**:			
UC			
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, **N**:			
Abstract, **AB**:		**This paper presents the design and implementation of a speculative merge algorithm within a continuous integration system. A system for dynamically composing the algorithm from independent commands has been developed, along with a scheduler responsible for managing their execution. Performance and success metrics of the system are presented, followed by a detailed analysis of the obtained results. Finally, potential directions for further development and implementation of the proposed solution are discussed.**	
Accepted by the Scientific Board on, **ASB**:			
Defended on, **DE**:			
Defended Board, **DB**:	President:	доц. dr. Miodrag Đukić	
	Member:	prof. dr. Igor Dejanović	Menthor's sign
	Member, Mentor:	prof. dr. Miroslav Popović	

Захвалност

Захвалан сам ментору проф. др. Мирославу Поповићу и техничком ментору др. Душану Кењићу за њихову помоћ и савете при изради овог рада.

Неизмерно сам захвалан својој породици и пријатељима, за помоћ и подршку током досадашњих студија. Посебну захвалност дугујем својим колегама и пријатељима: Стефану Маринкову, Огњену Бошковићу, Вуку Антовићу, Стефану Митићу, Александру Ивановићу, Алекси Ђурђевићу, Анти Решетару, и Дејану Попову, за њихово менторство, помоћ и пријатељство током последње године студија и рада.

САДРЖАЈ

Списак слика

Списак табела

Скраћенице

CI – (енг. *Continuous integration*) – континуална интеграција

CD – (енг. *Continuous deployment*) – континуално распоређивање

JSON – (енг. *Java script object notation*) – објектна нотација јаваскрипта

HTTP – (енг. *Hyper text transfer protocol)* – протокол преноса хипертекста

PR – (енг. *Pull request*) – захтев за спајање

1. Увод

У традиционалном току развијања софтвера, инжењери развијају код на својим рачунарима. Тестови се покрећу локално – ако су уопште присутни. Спајање на главну грану се врши после људске ревизије (енг. *Code review*), која може укључивати статичку анализу кода или покретање тестова. Овај приступ производи грешке због:

• Мањка тестова

• Неусаглашених окружења

• Неусаглашених скупова тестова

• Недостатака у интеграцији [1]

Увођењем система континуалне интеграције (енг. *CI system, Continuous integration system*) у процес развијања софтвера, осигурава се минимални скуп тестова који се морају успешно извршити, да би код био спојен на главну грану и заједничко окружење у ком ће се тестови покретати. Ови процеси побољшавају брзину итерације (производње нове верзије софтвера), стабилност и транспарентност у развијању софтвера [2]. Ова побољшања нарочито долазе до изражаја у тимовима са великим бројем инжењера, али се могу успешно применити и при индивидуалном развоју софтвера.

При тестирању кода у систему континуалне интеграције, оцењује се једна промена (енг. *Pull request* за *Github* или *commit* за *Gerrit*). Потребно је спојити ову промену на привремену грану, како би се над њеним кодом покренули тестови. Да би се утврдила тачност кода, промена се спаја заједно са свим променама које ће бити спојене на главну грану пре ње. Овај процес утврђивања тачног редоследа зове се спекулативно спајање [3].

Овај рад ће се фокусирати на пројекат и имплементацију једног алгоритма спекулативног спајања. Имплементација је независна од платформе. Решење подржава пројекте са више међузависних репозиторијума. Приоритети приликом пројектовања су:

• Интелигентно проналажење грешака (енг *Intelligent failure routing*) – изоловати погрешну промену и осигурати да само она буде оцењена негативно

• Брзина – минимизовати време чекања на повратну вредност система и осигурати да су оцене у сваком тренутку тачне

• Скалабилност – временска сложеност се скалира линеарно са бројем отворених промена

Из перспективе континуалне интеграције пројекти се значајно разликују. Постоје једноставни пројекти који се састоје из једне апликације, на којима раде мањи тимови. Други пројекти се састоје из великог броја међузависних апликација, библиотека и помоћних решења. Из овог разлога, за потребе решења приказаног у овом раду, неће бити довољно осмислити један алгоритам и применити га на све пројекте. Потребно је осмислити и имплементирати прилагодљив радни оквир који омогућава динамичко састављање и примену различитих алгоритама на различите пројекте.

Коначно, решење мора бити независно од било какве платформе континуалне интеграције, или сервера за аутоматизацију који представљају индустријски стандард у овој области. Овакво решење пружа прилагодљивост, могућност да се систем у неком тренутку у потпуности пребаци на другу платформу, без губитка перформанси.

2. Теоријске основе

За разумевање одређених аспеката захтева и концепта решења биће потребно објаснити неке концепте континуалне интеграције. У даљем тексту овог поглавља, биће описани основни појмови из домена континуалне интеграције, као и алати који су коришћени на пројекту. Осим тога, биће представљен кратак преглед комерцијално доступних решења. Из овог прегледа биће аргументована одлука да се развија независно решење, на основу захтева представљених у оквиру поглавља 1.

2.1 Концепти контроле кода

При развијању софтвера у већем тиму, користе се системи контроле кода (енг. *Source control management system, SCM system*). Индустријски стандард за ове сврхе је Гит (енг. *Git*), те је неопходно објаснити основне термине из контроле кода, везане за Гит.

При развоју, канонски код пројекта чува се на удаљеном серверу. Овај сервер може садржати код произвољног броја пројеката. Код се дели на **репозиторијуме**. Код једноставних пројеката, један репозиторијум може садржати код целог пројекта. Сложенији пројекти се могу састојати из више апликација, библиотека, и помоћних алата. У овом случају пројекат се дели на више међузависних репозиторијума.

Развој се дели на **промене** (енг. *Commit*). У општем случају, један инжењер развија једну функционалност, и сав код неопходан за ту функционалност се енкапсулира у једној промени. Вреди споменути да постоје различите филозофије о томе како тачно треба поделити развој на промене.

Код унутар репозиторијума се даље може поделити на **гране**. Гране представљају одвојене токове развоја. У општем случају, једна грана је означена као главна, а остале

на неки начин представљају подршку за главну грану. Такође, у неким тимовима, различите гране могу садржати различите верзије истог софтвера.

Инжењер на својој промени прво ради локално. У том локалном окружењу, преводи и повезује софтвер, тестира ручно, или покреће неки скуп аутоматизованих тестова. Када је закључио да његова промена решава одређени проблем, он промену поставља на удаљени репозиторијум, на некој грани. У овом стању промена не мора одмах бити спојена на главну грану удаљеног репозиторијума, већ се на њој могу вршити даљи тестови на дељеном окружењу. У овом стању, сматраћемо да је промена **активна**.

2.2 Концепти специфични за удаљени сервер

Иако је Гит индустријски стандард за контролу кода, он је компатибилан са многим серверима контроле кода (Гитхаб, Герит, БитБакет). Сваки од ових сервера имају мало другачији приступ контроли стања промене, и мало другачију терминологију. На овом пројекту коришћен је Герит, и сва терминологија је преузета од њега.

На Гериту, када промена први пут стиже на удаљени сервер, она добија стање **неактивне** промене (енг. *Work in progress*). Затим се она може пребацити у стање **активне** или **напуштене** промене. Такође, на променама је могуће додатно радити када се оне већ налазе на удаљеном репозиторијуму. У том случају, сваки додатак на промену се прати одвојено, и назива **закрпа** (енг. *Patchset*). Коначно, промена се може у било ком тренутку спојити на главну грану, чиме она постаје део канонског кода пројекта, и прелази у стање **спојена**.

Други удаљени сервери најчешће подржавају ове концепте, или неки њихов надскуп. На пример, Гитхаб такође подржава промене и закрпе, али се њихови енглески називи разликују (*Pull request, commit* уместо *commit, patchset*). Стога, ови изрази Герита ће пружити генерализовани интерфејс решења са удаљеним серверима, а за сервере који их не подржавају у потпуности, додатни слој адаптације ће бити имплементиран на модуларни начин. Коначно решење ће бити независно од удаљеног сервера који се користи, али ће зависити од Гита.

Герит подржава и проширење по имену Герит **Вебхук**. Оно омогућава Герит серверу да направи *HTTP* захтев аутоматски када се деси неки догађај. Вебхук ће се користити да би Герит сервер при настанку нове промене, или додавању закрпе на стару промену *HTTP* захтевом прозвао оркестрациони сервер, чиме се започиње оркестрација за ту промену.

2.3 Континуална интеграција

Континуална интеграција у ширем смислу, представља скуп пракси при развијању софтвера. Циљ ових праски је да се омогући континуална валидност софтвера, то јест да у сваком тренутку постоји канонска верзија софтвера, са најновијим интегрисаним променама, која ради, по неком критеријуму. Овај критеријум најчешће подразумева пролазак алгоритма превођења, заједно са неким минималним скупом тестова.

2.4 Алгоритам превођења

Алгоритам превођења представља поступак изградње апликације из кода. У ужем смислу ово је чисто превођење програмског кода у машински. У континуалној интеграцији, превођење се односи на шири скуп операција. Типичан алгоритам превођења почиње са превођењем у машински код, затим спаја све неопходне апликације и библиотеке, и на крају позива одређени скуп тестова, најчешће секвенцијално. Такође се могу укључити и статичка анализа кода, тестови перформансе, и слично. Детаљност тестирања зависи од захтева пројекта. Улаз у алгоритам превођења представља листа промена које треба тестирати, организованих временски или према међусобним зависностима. Током извршавања алгоритма, ове промене се редом којим су прослеђене спајају на привремену грану ради тестирања. Излаз је позитивна или негативна оцена. Излаз алгоритма у случају да само превођење није успело ће увек бити негативан. У случају да неки тестови нису успели, може се дефинисати праг пролазности тестова, неопходан да би излаз био позитиван. Имплементација алгоритма превођења је ван обима овог рада, и надаље ће се посматрати као црна кутија.

2.5 Ланац зависности

На датом репозиторијуму могуће је имати више отворених промена. У том случају, Герит омогућава експлицитно дефинисање зависности између две или више промена. Дефинисањем **експлицитног ланца зависности** инжењери назначују да су промене међусобно зависне, тако што раде на истој функционалности, или решавају исти проблем на неки начин. Како су све промене у једном експлицитном ланцу увек на истом репозиторијуму, ове зависности су у главном хронолошке природе – на пример, функционалност Б је имплементирана после функционалности А, и ослања се на неки начин на свог претходника. Најчешће неће све промене бити у једном експлицитном ланцу, већ ће бити потребно спојити више експлицитних ланаца, заједно са променама које уопште нису у ланцу, у један **имплицитни ланац зависности**, да би се утврдио коначан редослед којим промене треба спојити ради тестирања.

2.6 Тандем промене

Одељак 2.5 објашњава концепт експлицитног ланца зависности, на примеру Герита. Овакве експлицитне зависности имају смисла само у оквиру једног репозиторијума, јер су само у том контексту подржане.

Међутим, код сложенијих пројеката који обухватају више међусобно повезаних репозиторијума, често постоји потреба за експлицитним повезивањем промена из различитих репозиторијума. Пример за то може бити веб апликација чији су серверски и клијентски део раздвојени у посебне репозиторијуме. У таквом случају, једна промена може уводити нову функционалност на серверу, док друга промена у клијентској апликацији ту функционалност користи.

Из овог разлога потребно је увести концепт **тандем промена**. Тандем промене су промене на различитим репозиторијума, које не смеју бити оцењене једна без друге. При спекулативном спајању биће потребно осигурати да имплицитни ланац зависности никада не садржи подскуп неког скупа тандем промена, и да су све тандем промене у скупу оцењене истоветно. Ако је једна промена оцењена позитивно, све њене тандем промене морају такође бити оцењене позитивно, и обрнуто.

2.7 *Jenkins* дистрибуирани систем

***Jenkins* сервер за аутоматизацију** представља једно од најраспрострањенијих решења у области континуалне интеграције. Он омогућава тимовима да аутоматизују процес развоја софтвера: од преузимања изворног кода из репозиторијума, преко изградње и тестирања, до имплементације готовог производа у продукционо окружење. На овај начин се повећава ефикасност развојног процеса, смањује могућност људске грешке и убрзава време испоруке нових верзија софтвера.

Jenkins је изграђен као **дистрибуирани систем**, што значи да омогућава расподелу послова на више рачунара ради боље искоришћености ресурса и паралелног извршавања. Основни градивни елементи овог система су **чворови**, који се деле на две главне врсте: **контролер** и **агент**.

Контролер представља централни сервер који управља целокупним системом. Он обезбеђује кориснички интерфејс (веб апликацију) преко кога се конфигуришу задаци, прате извршавања и анализирају резултати. Поред тога, контролер је задужен за планирање и распоређивање послова на агенте, као и за комуникацију са репозиторијумима кода.

Агенти су рачунари или виртуелне машине који обављају стварни рад, тј. извршавање задатака које им контролер додели. Сваки агент може бити специјализован

за одређену врсту посла (нпр. превођење кода, тестирање, паковање артефаката) и може радити на различитим оперативним системима, што омогућава прилагодљивост у изградњи комплексних процеса.

У оквиру система изложеног у овом раду, Jenkins је коришћен као механизам за **аутоматско извршавање алгоритма превођења**. То значи да сваки пут када се покрене процес изградње, Jenkins контролер шаље захтев агентима да покрену тај алгоритам. На овај начин се обезбеђује поуздано, поновљиво и потпуно аутоматизовано извршавање превођења, без потребе за ручном интервенцијом. У даљем раду када се каже да нека компонента прозива алгоритам превођења, подразумева се да она шаље *HTTP* захтев према *Jenkins* контролеру, који даље распоређује посао агентима на којима се превођење извршава.

2.8 Преглед комерцијално доступних решења

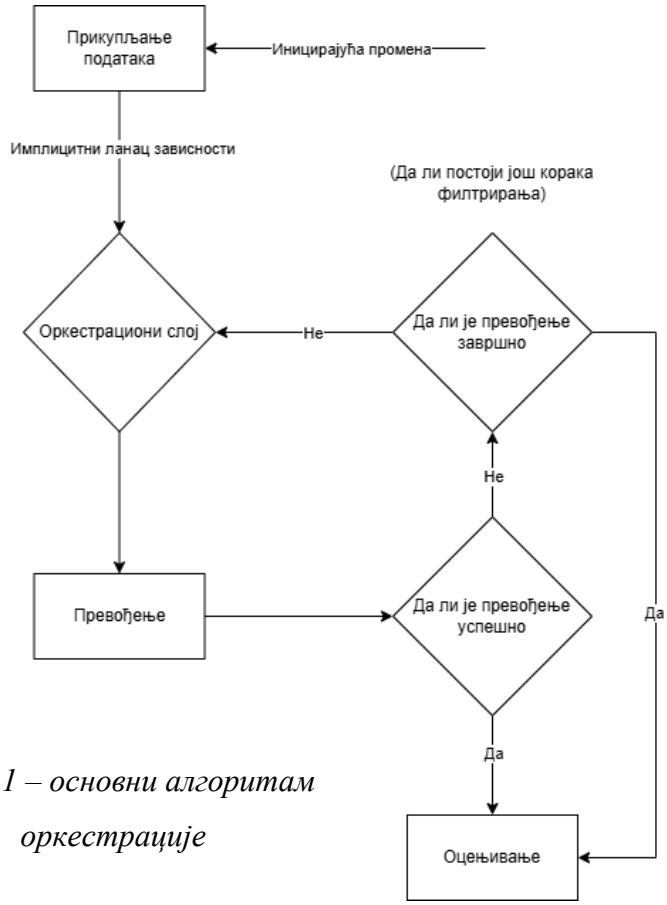
Многи алати континуалне интеграције се за сврхе спекулативног спајања ослањању на извршавање у турама (енг. *batching*). Овде спадају Мергифај и ГитЛаб. Основна идеја је да се промене скупљају до одређеног временског периода. Затим се, у регуларним интервалима, покреће превођење целог ланца промена. Уколико је превођење успешно, све промене добијају позитивну оцену. У супротном се неким алгоритмом претраге долази до промене са грешком и она се оцењује негативно. Овај приступ има предности у односу на наш - захтева много мањи број превођења и због тога мање ресурса. Међутим, ови приступи не дају повратну информацију у реалном времену. Повратно време система би било око 1 радни дан, што представља значајно ограничење.

Други скуп решења се ослања на динамички приступ. Овде спадају ГитХаб, и неке верзије ГитЛаба. Код њих постоје алгоритми који се покрећу сваки сваки пут када дође до промене на коду. Проблем је што многи од ових система посматрају промене независно. Други подржавају неки ниво спекулативног спајања, али често на малом подскупу отворених промена, те неће захватити грешке на целом ланцу. Коначно, постоје решења која су довољно прилагодљива да би постигла наше захтеве. Ипак, овај приступ би захтевао конфигурисање комерцијалног система на начин за који није осмишљен, што изискује велики труд. Из ових разлога имплементирано је потпуно ново решење.

3. Концепт решења

Ово поглавље пружа концептуални преглед решења. Одељак 3.1 излаже један пример алгоритма оркестрације. Одељак 3.2 користи овај алгоритам као прототип, и на њему објашњава концепт динамичког генерисања оркестрационог тока. Одељак 3.3 третира оркестрационе токове као независне процесе, и бави се њиховим распоређивањем.

3.1 Алгоритам оркестрације

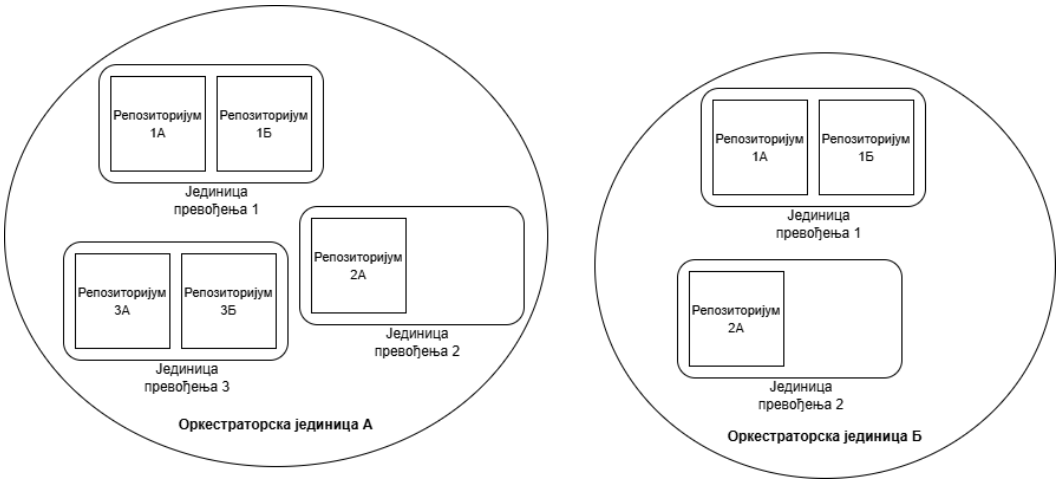


Слика 1 – основни алгоритам оркестрације

На слици 1 приказан је ток рада алгоритма оркестрације. Алгоритам се покреће при отварању нове промене. Улазни подаци алгоритма су детаљи о промени која је изазвала покретање и конфигурација пројекта. Алгоритам прикупља све отворене промене на релевантним репозиторијумима. Затим се одлучује о канонском редоследу промена (конструише имплицитни ланац зависности). На крају се позива минимални неопходни број превођења, да би се дошло до коначне оцене промене (негативне или позитивне).

3.1.1 Конфигурација

Конфигурација система у овом контексту се односи на постојаће репозиторијуме, њихову организацију у групе и дефиницију њихових међузависности. Слика 2 приказује једну теоретску конфигурацију система.



Слика 2 – Пример конфигурације пројекта

Репозиторијуми су груписани у **јединице превођења** (израз који ми уводимо, у даљем тексту ЈП). ЈП подразумева све репозиторијуме који се преводе заједно, те не представљају смислену целину један без другог. ЈП-ове је корисно посматрати као међусобно дискретне апликације, које могу бити део истог пројекта и имати зависности.

Оркестрациона јединица (такође израз који ми уводимо) представља скуп свих ЈП-ова који се оцењују заједно. Када два ЈП-а имају зависност, они се могу превести независно један од другог, али се не могу тако тестирати. Да би се нека промена правилно оценила, није довољно само проверити да ли пролази тестове.

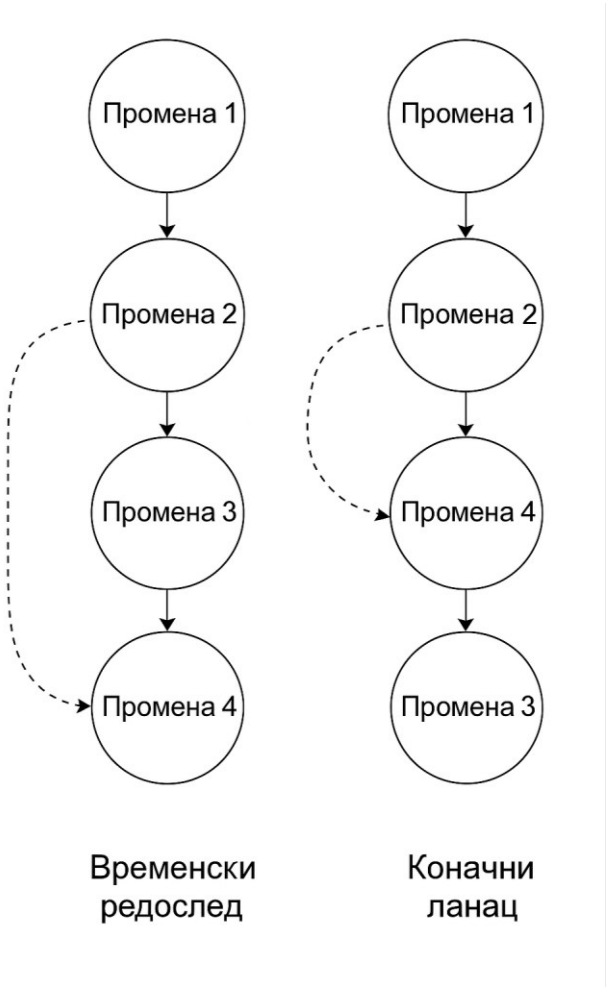
Такође је важно утврдити које су најновије верзије свих ЈП-ова од којих та промена зависи. Потребно је проверити да ли промена исправно функционише са тим верзијама.

3.1.2 Прикупљање података

Слој прикупљања података (енг. *Gating layer*) као улаз има податке о промени која је изазвала покретање алгоритма (енг. *Triggering change*, у даљем тексту иницирајућа промена) и конфигурацију система. Задатак овог слоја је да прикупи све промене од којих ће зависити оцена иницирајуће промене и утврди њихов тачан редослед.

За релевантне промене сматрамо оне које припадају репозиторијумима у истој оркестрационој јединици као иницирајућа. Да би се добавиле, потребно је ишчитати из конфигурације све ЈП-ове који припадају оркестрационој јединици, и добавити њихове репозиторијуме. Потом се за сваки репозиторијум добављају отворене промене.

Тачан редослед промена зависи примарно од времена стварања промене - старије промене иду прве. Такође, потребно је омогућити корисницима експлицитни ланац зависности (ручно дефинисање зависности између промена). Тамо где овакав ланац постоји, потребно га је испоштовати са већим приоритетом од временског редоследа. Слика 3 илуструје један пример овог концепта.



Слика 3 – Илустрација рада са експлицитним ланцем зависности

3.1.3 Конструисање ланца за превођење – слој оркестрације

Циљ оркестрационог слоја је да на основу најмањег могућег броја превођења утврди оцену промене. Ово постижемо процесом секвенцијалног превођења и филтрирања.

Улаз оркестрационог слоја је цео, нефилтрирани ланац промена. Превођења се увек позивају прослеђивањем подланца свих промена закључно са иницирајућом – нема потребе да се преводе промене изнад ње у ланцу.

Први корак је покушај превођења без филтрирања. Ако је превођење успешно, то значи да иницирајућа промена ради (пролази све тестове), као следећа у ланцу свих отворених промена. У овом случају, може се одмах оценити позитивно.

Уколико је прво превођење неуспешно, не можемо одмах закључити да је то због иницирајуће промене. Следећи корак је филтрирање свих познато лоших промена у ланцу. Ово су промене које су већ оцењене негативно, и стога знамо да ће изазвати негативну оцену ако их уврстимо у ланац. Затим преводимо поново.

Уколико је и друго превођење неуспешно, потребно је филтрирати све промене које су експлицитно означене као неактивне - нису још спремне да буду оцењене. Након овог филтрирања, ланац који остаје се састоји искључиво од позитивно оцењених промена и иницирајуће промене, која још нема оцену. Уколико је и треће превођење неуспешно, знамо да је ово сигурно због грешке у иницирајућој промени и можемо јој дати негативну оцену.

Ако је у било којем тренутку алгоритам затражи оцену активне промене која још није оцењена, систем прелази у неактивно стање док та промена не добије оцену. Конфигурација система гарантује да ће се ово десити - ако је промена активна оркестрација за њу је већ покренута, и оцена ће ускоро бити додељена.

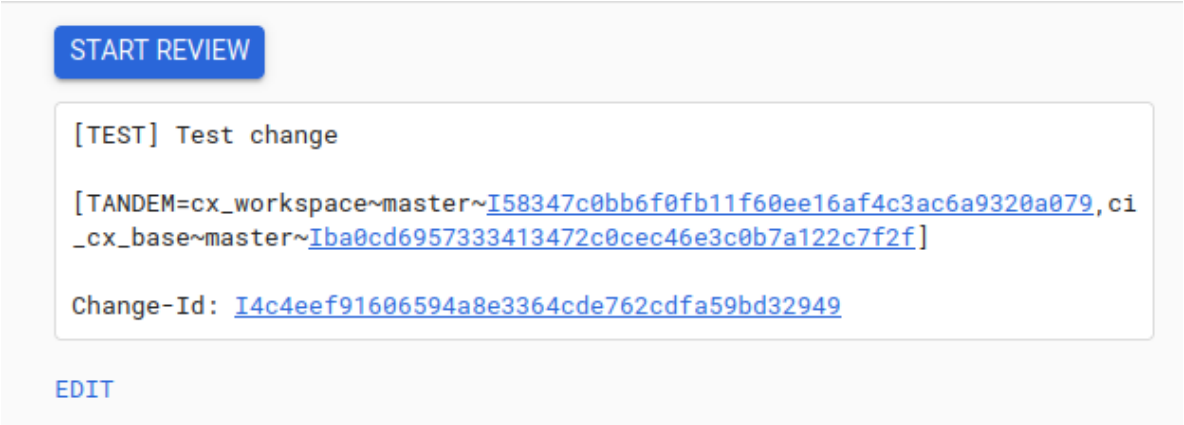
3.1.4 Поново покретање за већ оцењене промене

Ланац промена може укључивати и промене изнад иницирајуће. Један пример овога је када се мењају већ отворене промене (додају датотеке, поправљају грешке). У овом случају, већ отворену промену је потребно поново превести, и доделити јој нову оцену. Како она није на врху ланца, постоје промене које зависе од ње. Све промене које зависе од иницирајуће, потребно је оценити поново, са новом верзијом иницирајуће промене. Ово се постиже секвенцијалним поновним покретањем оркестрације, за сваку промену почевши од промене директно изнад иницирајуће. Овај процес осигурава константну ажурност оцена.

3.1.5 Функционалности везане за тандем промене

Одељак 2.6 описује концепт тандем промене. Ако бисмо ове промене посматрали независно, на начин описан у одељцима 3.1–3.4, могло би се десити да се приликом тестирања једне промене друга промена занемари. Из разлога описаних у одељку 2.6, ово би изазвало падање тестова.

При раду са тандем променама, потребно је назначити зависност између њих на експлицитан начин који није увек подржан уграђеним функционалностима удаљеног сервера за контролу кода. Стога је потребно пружити кориснику ову функционалност. При раду са системом, корисник ће назначити зависности између тандем промена додавањем специјалне ознаке у опис своје промене. Ознака мора бити окружена угластим заградама, и почињати са кључном речи *TANDEM=*. Непосредно после кључне речи долази списак промена које садрже тандем зависности са тренутном променом. Промене су наведене по репозиторијуму и грани на којој се налазе, и јединственом идеинтификатору те промене. Слика 4 приказује пример описа промене, који садржи ознаку тандема са две тандем промене.



Слика 4 – Опис промене са два тандема

Овакав приступ раду са тандем променама је прилагодљив, пошто се не ослања ни на једну функционалност удаљеног сервера, већ на коренску функционалност Гита – описа промене. Међутим, јавља се један проблем – редослед настајања промена. Ако корисник (инжењер) жели да направи тандем промене, он их мора направити једну за другом, промене неће настати у исто време. У тренутку настајања прве промене, корисник не може знати идентификатор других промена (он се генерише у тренутку настајања промене). Стога, он ће направити све промене из тандем скупа, и затим се вратити и редом им додати ознаке тандема. Ако нека од промена постане активна у периоду када нема тандем ознаку, њени превођење ће сигурно пасти. Да бисмо избегли непотребно превођење промена које су у процесу постављања, омогућавамо корисницима да назначе да ће нека промена имати тандем промене, без директног навођења њиховог идентификатора. Слика приказује једну такву промену.

Слика 5 – Промена са 2 тандем промене, наведене имплицитно, тако да оне још увек не постоје

У случају да се покрене оркестрација за промену која има назначене тандем промене, али оне још не постоје, потребно је одмах прекинути оркестрацију и оценити промену негативно. Такође је потребно оценити све њене тандем промене које већ постоје на исти начин. Затим, када последња тандем промена из скупа настане, ако је успешно преведена и оцењена позитивно, потребно је поново покренути оркестрацију за све њене тандем промене, да би и оне добиле реалне оцене, за случај да су прво оцењене негативно. У случају да је било која промена из тандем скупа оцењена негативно, све промене из тандем скупа треба оценити негативно.

3.2 Динамичко конструисање алгоритма оркестрације

Из разлога дефинисаних у поглављу 1, није довољно једноставно пројектовати статички алгоритам и применити га једнако на сваки пројекат. Потребно је имплементирати радни оквир за динамичко конструисање алгоритма оркестрације, потенцијално другачијег за сваки пројекат.

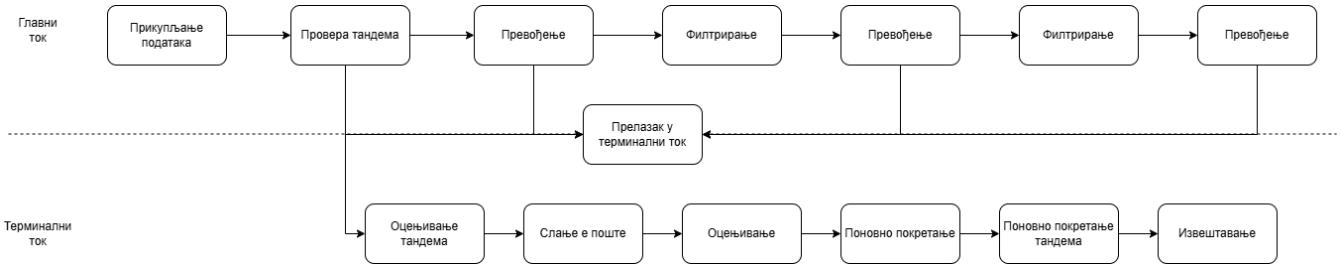
3.2.1 Концепт команде

При имплементацији динамичког конструисања алгоритма, оркестратор описан у одељку 3.1 је коришћен као прототип оркестрационог алгоритма. Да би се овај алгоритам динамички конструисао, потребно је изделити га на међусобно независне операције. Ове операције морају бити такве да њихов редослед не утиче на саму функционалност операције. Њихови интерфејси морају бити усклађени, и морају бити међусобно компатибилни. Ако су ови захтеви испуњени, било која комбинација операција ће представљати смислен оркестрациони алгоритам, мада ће само неки подскуп ових комбинација бити користан. Ове независне операције зваћемо **команде**.

3.2.2 Концепт оркестрационог тока

Један алгоритам за оркестрацију представља списак команди које треба извршити секвенцијално. Такав алгоритам зовемо **оркестрациони ток**. По опису оркестрационог тока из одељка 3.1, јасно је да постоје тачке у којима је потребно прекинути оркестрациони ток. Такође постоје команде које треба извршити на крају оркестрационог тока, без обзира на то да ли је он прекинут превремено или не. Овде спадају команде за оцењивање промене, слање извештаја, поново покретање оркестрације и слично. Из овог разлога делимо оркестрациони ток на два: главни ток и завршни ток. Команде из главног тока се извршавају секвенцијално. У тренутку када се дође до завршног стања, прекида се главни ток и прелази се на завршни ток, који се потом извршава без прекида. Свака команда садржи услов под којим систем прелази у завршно стање. На примеру команде

за превођење, ово би била ситуација у којој је превођење успешно, те је довољно само оценити промену позитивно и завршити оркестрацију. Слика 6 приказује коначан изглед оркестрационог тока за алгоритам описан у одељку 3.1.



Слика 6 – потпун оркестрациони ток из одељка 3.1, описан као главни и завршни ток команди

3.3 Распоређивач

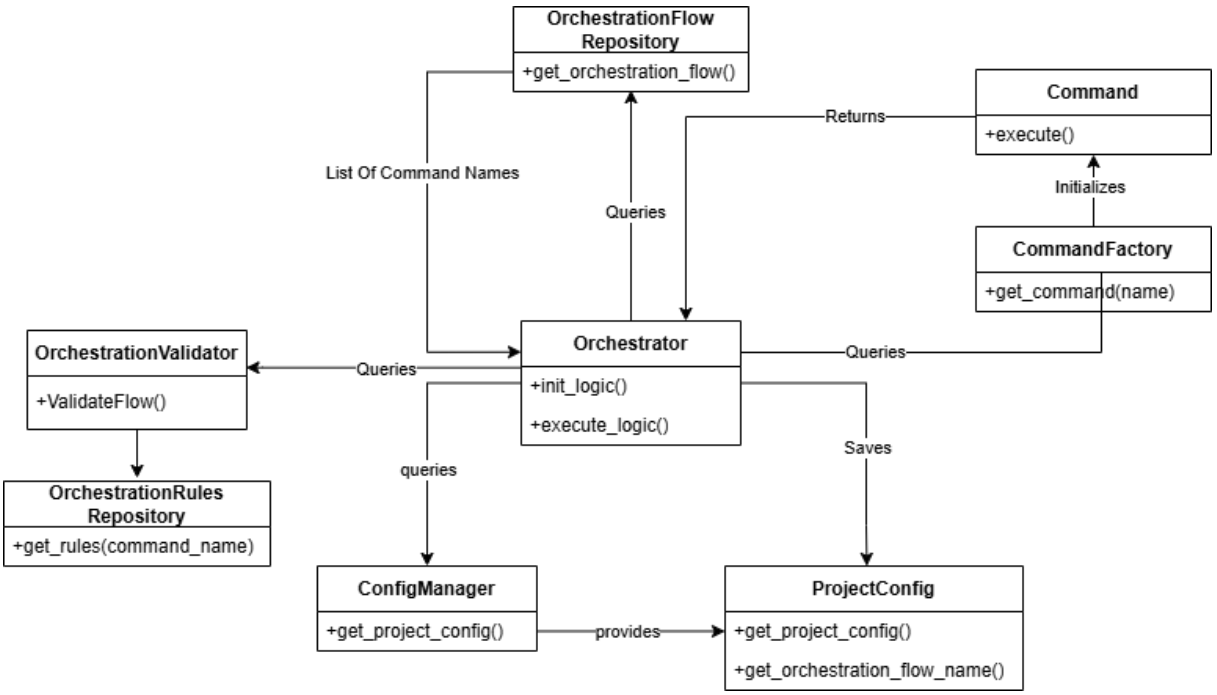
Оркестраторски сервер представља једну независну компоненту дистрибуираног система. Један ток оркестрације има за циљ да оцени једну промену, и у даљем раду ће се звати једним **оркестратором**. Један оркестратор представља динамички инстанциран поткомпоненту оркестраторског сервера. Овај оркестратор се такође сматра командом, у смислу да енкапсулира логику коју треба извршити. Ова логика се састоји од више команди које треба извршити секвенцијално. Оркестратори се извршавају међусобно паралелно. Потребно је распоредити оркестраторе на систему са ограниченим бројем нити и језгара, на начин који максимално искориштава ресурсе. Компонента која енкапсулира ову логику зове се **распоређивач**.

4. Програмско решење

Алгоритми и концепти описани у поглављу 3 су имплементирани у виду сервера у радном оквиру Flask уз помоћ програмског језика Python. Сервер је покренут константно, а оркестрација се покреће преко Герит *Webhook*-а, аутоматски, на начин описан у поглављу 2.

4.1 Преглед архитектуре оркестрационог сервера

Слика 7 приказује преглед архитектуре оркестрационог сервера. Сврха поглавља 4 је да објасни сваку компоненту ове архитектуре, уз логику њихове имплементације, и њихове међусобне интерфејсе.



Слика 7 – Архитектура оркестраторског сервера

4.1.1 Управљач конфигурације

У одељку 3.1.1 је описана основна конфигурација пројекта са примерима. Она представља распоред јединица оркестрације, јединица превођења и репозиторијума за један пројекат. Осим ове конфигурације пројекта, потребно је чувати још конфигурационих података. За конфигурацију користимо формат *JSON*, због његове читкости и лакоће парсирања. Комплетна конфигурација за све пројекте је издељена на више *JSON* датотека, по логичким целинама. Оне датотеке које се односе на специфичне пројекте се налазе на одвојеним гранама за те пројекте. Датотеке које се односе на цео систем се налазе на главној грани репозиторијума. Табела 1 представља преглед конфигурационих датотека у систему. Управљач конфигурације такође садржи и датотеке које нису описане у табели 1, али оне немају везе са оркестрацијом. Из тог разлога су ван обима рада и неће бити описане.

Табела 1 – Конфигурационе датотеке у систему, заједно са описима

ДАТОТЕКА	ЛОКАЦИЈА	ОПИС
КОНФИГУРАЦИЈА ПРОЈЕКТА	Грана чије име одговара интерном називу пројекта	Списак репозиторијума пројекта, заједно са јединицама превођења и оркестрације, на начин описан у одељку 3.1.1.
КОНФИГУРАЦИЈА РАСПОРЕЂИВАЧА	Главна грана пројектног репозиторијума	Ограничења броја нити доступних распоређивачу, по пројекту и глобално, на начин објашњен у одељку 4.6.2.
ЛОГИКА ОРКЕСТРАЦИЈЕ	Главна грана пројектног репозиторијума	Сви оркестрациони токови у систему, описани на начин из поглавља 4.3.
ВАЛИДАЦИЈА ОРКЕСТРАЦИЈЕ	Главна грана пројектног репозиторијума	Правила која описују међузависности између команди.

Да бисмо поједноставили добављање одговарајућих података о конфигурацији, имплементирана је помоћна компонента која ће се на даље звати **управљач конфигурације**. Управљач чува у својој радној меморији верзије сваке датотеке из табеле, и енкапсулира податке о њиховим локацијама, на удаљеним репозиторијумима. Његов интерфејс се састоји од метода за ажурирање сваке од конфигурационих датотека и метода за добављање садржаја ових датотека. Методе за ажурирање датотека су изложене кроз интерфејс оркестрационог сервера, који се може погодити обичним *HTTP* захтевом.

При инстанцирању управљача конфигурације, пројектни репозиторијум се тражи на прослеђеној локалној путањи. Уколико није нађен, биће клониран са удаљеног сервера, на ту локацију. У будућности све Гит операције неопходне за управљање конфигурацијом се извршавају на овој инстанци репозиторијума. Затим, управљач добавља списак свих грана са овог репозиторијума. Овај списак грана одговара у потпуности списку пројеката које систем тренутно подржава, када се изузму главна грана

и још неколико помоћних грана за документацију, које се занемарују. Управљач ће затим ажурирати конфигурације специфичне за пројекат итерацијом кроз овај списак грана, а глобалне конфигурације ће ажурирати са главне гране. Коначно, сваки пут када се промена споји на пројектни репозиторијум, одговарајуће конфигурације ће бити поново ажуриране.

У даљем раду, када се каже да нека компонента добавља конфигурацију, подразумева се да ово ради кроз управљач конфигурације, а не ручним клонирањем репозиторијума и парсирањем датотека. Овим систем добија на модуларности.

4.2 Команде

4.2.1 Структура команде

На начин описан у поглављу 3, функционалности оркестратора у ужем смислу енкапсулиране су у команде. Команде су имплементиране као класе са заједничком методом *.execute()*, која извршава комплетну функционалност команде. Команда такође може садржати помоћне методе које нису део интерфејса и сматрају се приватним, иако Пајтон не подржава ову функционалност у потпуности.

Излаз команде је излазни код, у виду енумерације. Табела 2 представља излазне кодове команди и њихове ефекте. Неке команде, попут филтрирања, никад нису завршне. Друге команде садрже функтор *is_terminal*, који представља логички услов под којим команда јесте завршна.

Табела 2 – излазни кодови команди са значењима

ИЗЛАЗНИ КОД	ЗНАЧЕЊЕ
OK	Команда је извршена успешно и није завршна. Потребно је наставити са извршавањем главног тока, или прећи у завршни у случају да је ово последња команда у главном.
TERMINAL	Команда је извршена успешно али је завршна. Потребно је прескочити све преостале команде из главног тока и прећи на извршавање завршног тока
ERROR	Дошло је до круцијалне, необрађене грешке приликом извршавања команде. Потребно је одмах прекинути сво извршавање тока. Ово се ради да би се избегло слање бесмислених оцена и непотребне електронске поште преласком у завршни ток.

4.2.2 Команда прикупљања података

Команда прикупљања података мора прикупити све отворене промене на свим репозиторијумима у оркестрационој јединици. Затим је потребно сложити ове промене у имплицитни ланац зависности. Ово се постиже итерирањем кроз промене поређане хронолошким редоследом. За сваку промену, добавља се њен експлицитни ланац зависности, и умеће у имплицитни без нарушавања. Хронолошки следећа промена се додаје тек после целог експлицинтог ланца зависности хронолошки прошле команде.

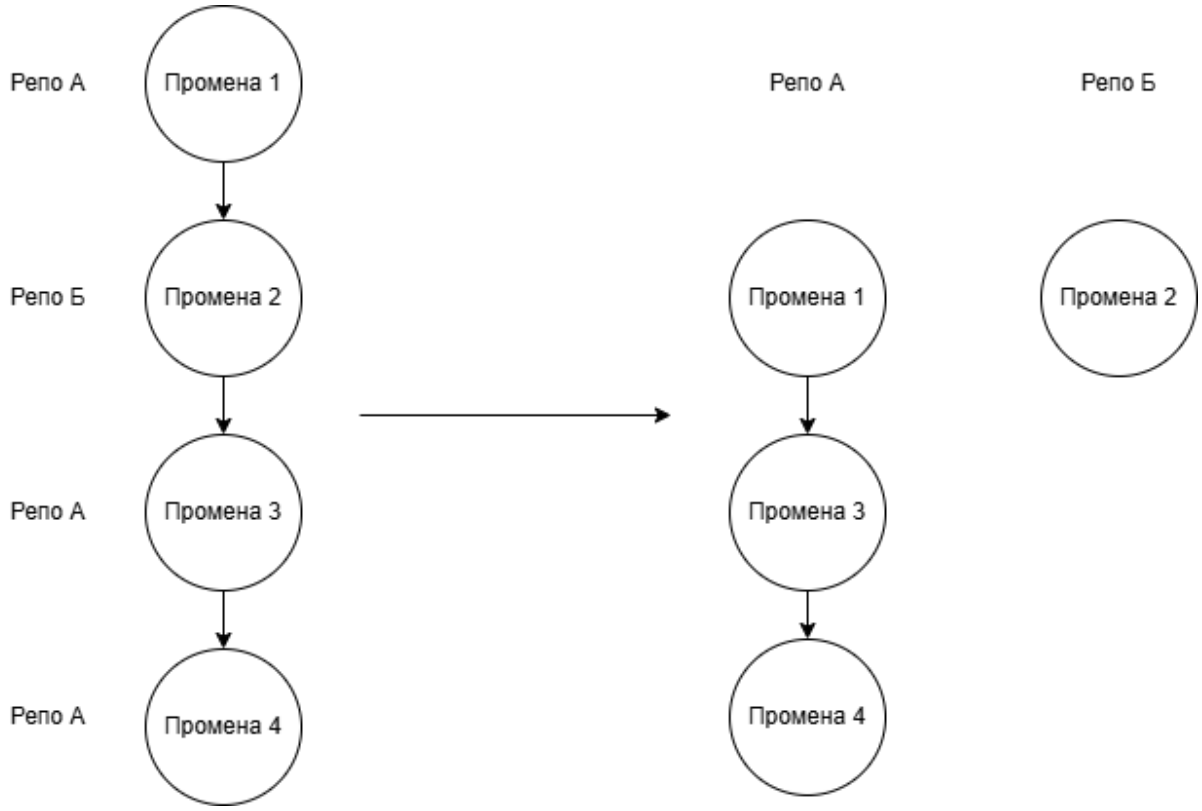
4.2.3 Команда превођења

Табела 3 садржи све улазне параметре алгоритма превођења, заједно са њиховим кратким описима. Команда превођења припрема ове параметре и позива алгоритам превођења *HTTP* захтевом према *Jenkins* серверу.

Табела 3 – улазни параметри алгоритма превођења

ПАРАМЕТАР	ОПИС
REF_CHAIN	Речник који садржи имплицитне ланце зависности за сваки репозиторијум, закључно са иницирајућом променом. Начин на који су промене референциране зависи од удаљеног сервера који се користи
SCM_PROJECT	Назив репозиторијума на којем се налази иницирајућа промена
SCM_BRANCH	Грана репозиторијума на којој се налази иницирајућа промена
SCM_PATCHSET_REVISION	Најновија закрпа иницирајуће промене у тренутку иницирања оркестрације
REPO_BUILD_NUMBERS	Речник који садржи јединствени идентификатор артефакта који ће бити коришћен уместо превођења сваког репозиторијума за који је то могуће – детаљније објашњено у одељку
REPOS	Тип сваког репозиторијума у односу на конфигурацију пројекта, објашњено у одељку

Као што је назначено у одељку 2.7, алгоритам превођења се извршава у виду тока обраде на *Jenkins* серверу. Овај алгоритам је ван опсега рада. Његов улаз је листа промена које треба спојити на привремену грану, за сваки репозиторијум у оркестрационој јединици. Да би се позвао алгоритам превођења, неопходно је разложити ланац зависности, који садржи промене из сваког репозиторијума, на појединачне ланце за сваки репозиторијум. Слика 8 је дијаграм који приказује овај концепт на реалном примеру.



Слика 8 – пример раздвајања ланца зависности

У одређеним ситуацијама није неопходно преводити сваку јединицу превођења у оркестрационој јединици. Резултат сваког успешног превођења се чува на централном серверу као артефакт. Како свака нова промена покреће оркестрацију, а самим тим и превођење, свака валидна промена ће имати одговарајући артефакт превођења. Укратко, ако је промена валидна, сигурно је успешно преведена, и стога постоји њен артефакт. Овај артефакт је заправо артефакт целе јединице превођења. За дату иницирајућу промену, довољно је поново превести само јединицу превођења којој та промена (тј. репозиторијум на ком се она налази) припада. За остале јединице превођења довољно је само тестирати да ли су компатибилне са новом верзијом јединице превођења која се променила. Ово се постиже скидањем њихових артефакта са централног сервера, и тестирањем тих артефакта. Да би алгоритам превођења могао ово да уради, потребно је

да оркестраторска команда превођења проследи речник свих репозиторијума, и за сваки репозиторијум идентификатор најновијих артефакта његове јединице превођења.

Тип сваког репозиторијума се добавља једноставном итерацијом кроз репозиторијуме оркестраторске јединице. Ови подаци постоје у конфигурацији пројекта и добављају се у складу са процедуром из одељка 4.1.1.

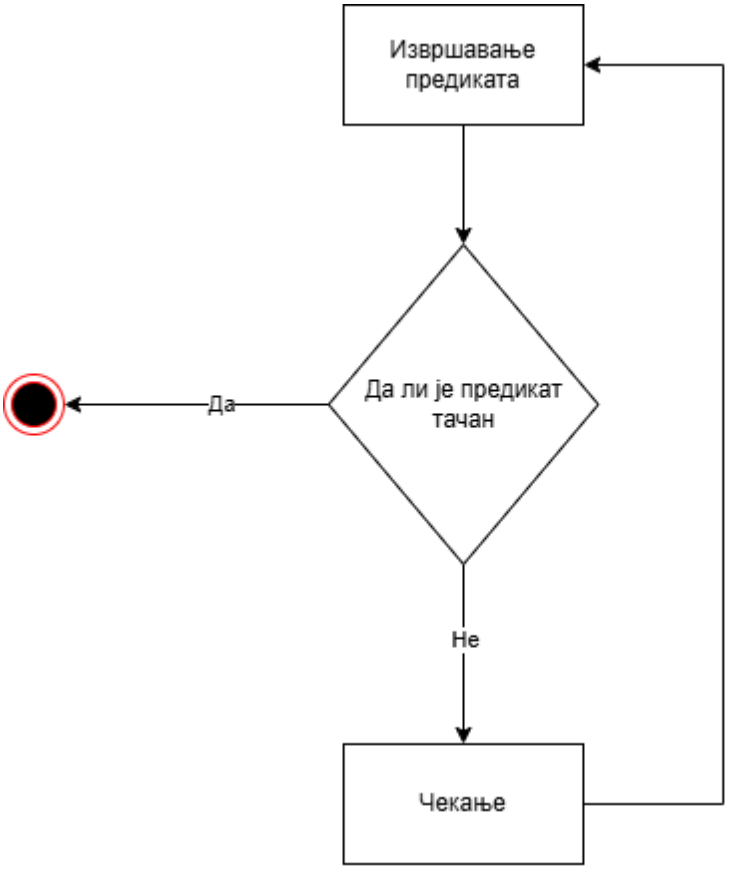
Слика представља дијаграм тока оркестраторске команде превођења. Само позивање алгоритма превођења се извршава *HTTP* позивом кроз помоћну компоненту.

4.2.4 Команда чекања

У одељку 3.1 описано је у којим тренуцима алгоритам оркестрације мора сачекати да оцена буде додељена некој промени из ланца зависности. Нажалост, Герит не подржава никакав догађај или вид аутоматског обавештења када је промена оцењена. Стога, неопходно је пребацити ток оркестрације у неактивно стање, и периодично проверавати оцену свих промена у ланцу зависности.

Команда чекања је имплементирана на генерализован начин. Она садржи произвољан предикат, и чека произвољан број секунди између провера тог предиката. Испод је један пример предиката, који се тренутно користи у продукцији.

Слика 9 представља дијаграм тока команде чекања.



Слика 9 – дијаграм тока команде чекања

4.2.5 Команда поновног покретања

У одељку 3.1.4 описани су тренуци у којима је потребно поново покренути оркестрацију за цео ланац зависности изнад иницирајуће промене. Ово се постиже иницијализацијом новог оркестраторског објекта. За овај оркестратор, није неопходно поново обављати прикупљање података. Довољно је само одрадити оркестрацију на већ постојећем ланцу зависности, али тако да је иницирајућа промена другачија.

Ова команда итерира кроз део ланца зависности изнад иницирајуће промене. За сваку промену у овом делу, конструише оркестратор. Ови оркестратори имају специјални оркестрациони ток – ток поновног покретања. Овај ток нема команду прикупљања података, већ се извршава на већ формираном ланцу зависности. Затим, ови оркестратори се прослеђују распоређивачу. Распоређивач их третира исто као оркестраторе настале као одговор на настанак нове промене, с тим што се као део команде може конфигурисати нижи ниво приоритета за њих. Овај концепт значи да је поновно оцењивање већ оцењених промена једини корак алгоритма оркестрације који се извршава асинхроно. Конструисање оркестратора ће се извршити као део команде, али сама оркестрација је подложна распоређивању, и извршиће се тек када нит постане доступна.

4.2.6 Помоћне команде

Осим команди наведених у одељцима 4.2.2-4.2.5 систем садржи велики број помоћних команди. Оне не решавају неки логички проблем, већ једноставно пружају спрегу са неким удаљеним сервером, или другом компонентом система. Табела 4 представља примере ових команди, заједно са кратким описом њихових функционалности.

Табела 4 – помоћне команде

КОМАНДА	ОПИС
СЛАЊЕ ЕЛЕКТРОНСКЕ ПОШТЕ	Из одговарајуће конфигурације добавља податке о корисницима које треба обавестити о стању пројекта, при свакој оркестрацији. Затим позива модул за слање електронске поште, који конструише *HTML* код потребан за изглед електронског писма, и шаље га на све наведене оадресе
ОЦЕЊИВАЊЕ ПРОМЕНЕ	Позива модул за оцењивање промена. Прослеђује стање оцене (позитивна или негативна) и идентификатор промене коју треба оценити.
ОЦЕЊИВАЊЕ ТАНДЕМ ПРОМЕНА	Оцењује све промене у тандем скупу иницирајуће промене, истоветно са иницирајућом променом.
ИЗВЕШТАВАЊЕ	Уписује метаподатке о извршавању оркестрације у базу података. Ово представља помоћну функционалност која помаже у праћењу система.

4.2.7 Команде тандем промена

Систем такође препознаје три команде специфичне за обрађивање тандем промена. Не подржава сваки пројекат тандем промене, тако да се ове команде не користе у сваком току и не сматрају се делом главног оркестрационог алгоритма, али их вреди објаснити овде.

Табела 5 – команде везане за тандем промене

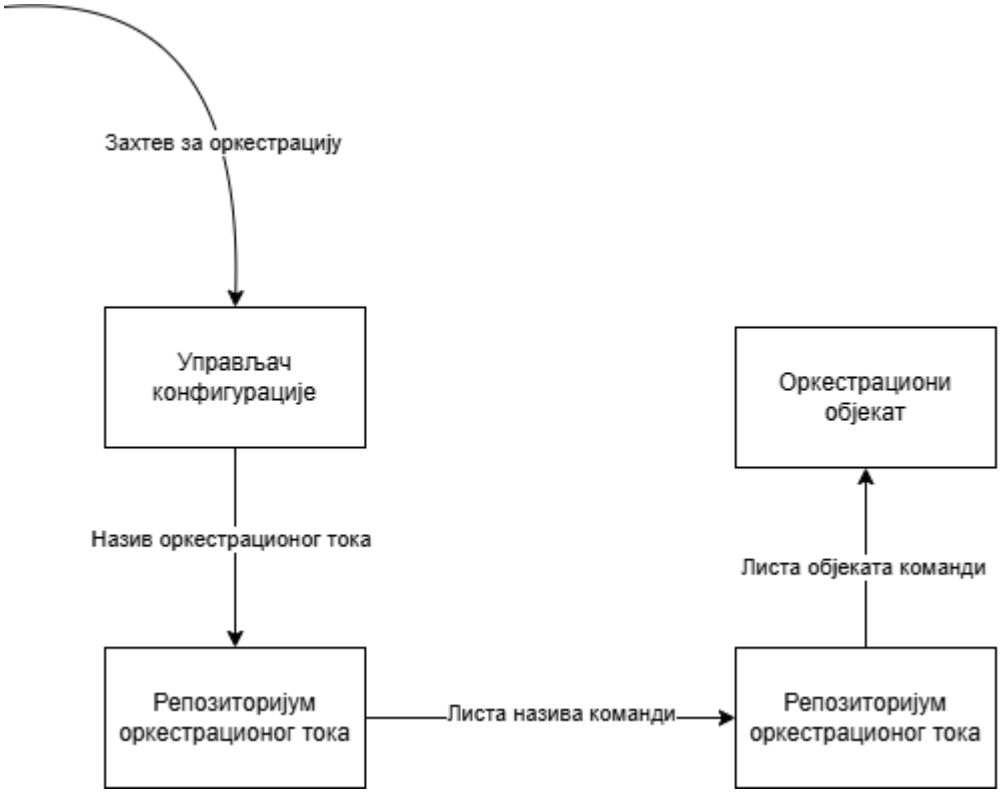
КОМАНДА	ОПИС
ОЦЕЊИВАЊЕ ТАНДЕМ ПРОМЕНА	Оцењује све промене у тандем скупу иницирајуће промене, истоветно са иницирајућом променом.
ПОНОВО ПОКРЕТАЊЕ ТАНДЕМ ПРОМЕНА	Функционише на исти начин као и обична команда поновног покретања. Једина разлика је што уместо итерирања по ланцу зависности, итерира кроз листу промена у тандем односу са иницирајућом променом, и ствара оркестраторске објекте за ове промене
ПРОВЕРА ТАНДЕМА	Проверава за сваку промену у тандем односу са иницирајућом променом, да ли она постоји и да ли се може наћи на удаљеном репозиторијуму. У случају да се и једна промена не може наћи, претпоставља се да је тандем скуп и даље у фази постављања. Све промене из тандем скупа добијају негативне оцене и прелазе у неактивно стање.

4.3 Оркестрациони ток

4.3.1 Конструисање оркестрационог тока

Конструисање оркестрационог тока се састоји од учитавања конфигурације оркестрационог тока, налажења и парсирања одговарајућег тока, и инстанцирања објекта за сваку од наведених команди. Потом се попуњавају листе команди које представљају главни и завршни ток оркестрације. По архитектури система на слици 7 из одељка 4.1, за овај ток одговорне су компоненте **фабрика команди, репозиторијум оркестрационог тока и валидатор оркестрационог тока.** Помоћну улогу игра управљач конфигурације описан у одељку 4.1.

Слика 10 представља дијаграм тока конструисања оркестрационог тока. Конструисање се иницира инстанцирањем оркестрационог објекта из распоређивача. Из конфигурације пројекта се добавља назив оркестрационог тока. Затим се из репозиторијума оркестрационог тока добавља ток у виду листе команди, референцираних по називу команде. Овај ток се затим валидира по правилима из валидатора оркестрационог тока, а по процедури из поглавља 4.3.2. Имена команди се потом прослеђују фабрици команди, која енкапсулира логику инстанцирања сваке од команди. На пример, филтрирање негативних оцена и филтрирање неактивних промена су оба инстанце исте класе, али се предикат филтрирања разликује.



Слика 10 – дијаграм тока конструисања оркестрационог тока

4.3.2 Валидација оркестрационог тока

Циљ имплементације оркестрационог тока јесте да било који инжењер, без доменског знања може да напише нови оркестрациони алгоритам за специфичне потребе пројекта, без додавања новог кода. Ово се постиже конфигурацијом у *JSON* датотеци. Сваки оркестрациони ток садржи листу команди у главном току, и другу листу у завршном току.

Иако су све команде концептуално независне, постоје оркестрациони токови који немају смисла. Најједноставнији пример је оркестрациони ток без иједне команде која конструише ланац зависности. Други пример су команде специфичне за тандем промене,

описане у одељку 4.2.7, које немају смисла једна без друге. Да би се избегла збуњеност и потреба за комплексном документацијом сваког од ових примера, имплементиран је **валидатор тока оркестрације.**

Валидатор тока оркестрације је имплементиран по принципу **правила.** Правило се састоји од једне (унарно) или две (бинарно) команде, и логичког предиката. Табела 6 приказује све логичке предикате тренутно имплементиране над валидатором, и њихово значење. За бинарни предикат, прва команда (лева) се сматра **изворном** а друга команда (десна) се сматра **циљном**. За сврхе валидације сматра се да се завршни ток наставља на главни ток линеарно, тј. сматра се да прва команда из завршног тока долази непосредно после последње команде главног тока.

Табела 6 – Предикати валидације оркестраторског тока

ПРЕДИКАТ	**ОПИС**
REQUIRED	Унаран, прослеђена команда се **мора** наћи у сваком оркестрационом току
REQUIRED_BEFORE	Изворна команда се **мора** наћи **пре** циљне команде у оркестрационом току. Ако циљна команда није у оркестрационом току, правило се сматра испуњеним
REQUIRED_AFTER	Изворна команда се **мора** наћи **после** циљне команде у оркестрационом току. Ако циљна команда није у оркестрационом току, правило се сматра испуњеним
INCOMPATIBLE	Изворна и циљна команда се **не смеју** наћи заједно у оркестрационом току. Ако се ниједна команда не налази у оркестрационом току правило се сматра испуњеним

Осим ових предиката, подржани су и логички оператори **и** и **или** између правила. Операција или је изражена оператором ||, по угледу на Ц-олике програмске језике. Операција и је имплицитна, сукцесивним навођењем правила се постиже да сва правила морају бити испуњена. Слика 11 представља пример табеле валидације оркестраторских токова, са зависностима израженим у табели 6.

Command_Dependencies										
	Build	**CheckTandem**	**FilterInvalid**	**FilterWip**	**Gating**	**Retrigger**	**RetriggerTandem**	**Review**	**ReviewTandem**	**SendEmail**
Build	x									
CheckTandem		x								
FilterInvalid			x							
FilterWip				x						
Gating					x					
Retrigger						x				
RetriggerTandem		Required before					x			
Review	Required before							x		
ReviewTandem		Required before							x	
SendEmail										x

Слика 11 – пример зависности валидације оркестраторског тока

4.4 Протокол безбедног завршетка

У неким ситуацијама неопходно је нагло завршити оркестрацију. Ово може бити због корисничког захтева, позивом команде из текстуалног интерфејса или HTTP захтева. Такође, завршетак се може захтевати програмски, из система надгледања оркестрације, описаног у одељку 4.5.

За ове сврхе имплементиран је **протокол безбедног завршетка.** Свака команда (оркестратор се сматра командом), садржи поље које назначава да ли би та команда требала да буде завршена. Команда је слободна да у тачкама када је то безбедно провери ово поље, и заврши свој ток на безбедан начин, прекидајући било какав рад.

4.4.1 Аргумент за безбедни завршетак

Један пример небезбедног завршетка јесте при команди превођења. Она покреће ток рада превођења на *Jenkins* серверу, и потом прелази у неактивно стање. Затим периодично проверава да ли је превођење готово, након чега може да настави рад. Ако би ова команда просто била завршена у произвољној тачки, њен ток би престао, али би се превођење наставило. Овим би се непотребно трошили ресурси система на превођење (најзахтевнија тачка система), које неће имати одговарајућу оцену.

Ако на овај пример применимо концепт безбедног завршетка, ток изгледа другачије. Команда превођења прима захтев за завршетак. Затим, први пут када се пробуди из неактивног стања, проверава да ли је у међувремену примила захтев за завршетак, и прелази у ток терминације. Ток завршетка садржи додатну логику, која прекида ток превођења на *Jenkins* серверу, чиме се избегава непотребно трошење

ресурса. Применом безбедног завршетка се омогућава да ова логика остане енкапсулирана унутар команде која се терминише. Компонента која захтева завршетак не мора да буде свесна детаља имплементације команде. Безбедни завршетак се сматра добром праксом у великим дистрибуираним системима, попут система описаног у овом раду.

4.5 Систем надгледања оркестрације

Оркестрација је дуготрајан процес. Током једне оркестрације, могуће је да ће за иницирајућу промену инжењери додати закрпу. У овом случају покренуће се поновна оркестрација, за верзију промене са најновијом закрпом, а стара оркестрација постаје непотребна. Ако се дозволи да стара оркестрација заврши са радом, непотребно се троше ресурси система. Осим тога, могуће је да ће старија оркестрација трајати дуже од нове, и преписати оцену нове оркестрације. Јасно је да је у случају додавања нове закрпе на већ активну промену неопходно прекинути стару оркестрацију те промене (протокол безбедног забршетка), ако она постоји.

Такође, добра пракса би била прекинути оркестраторе, као и команде превођења и чекања, које су премашиле неки праг времена извршавања. Ако се тај праг постави разумно, ово ће сигурно бити због неке врсте отказа, те ће се прекидањем ослободити ресурси система.

Ове функционалности енкапсулиране су у **систем надгледања оркестрације.** Свака команда, укључујући и оркестраторску команду, садржи листу својих **надгледача**. Свака команда је слободна да обавести своје надгледаче периодично, у тренуцима када је то безбедно. Овом приликом им прослеђује пакет метаподатака о себи. Овај концепт је сличан концепту безбедног завршетка, и тренуци у којима се обавештавају надгледачи се скоро увек поклапају са тачкама безбедног завршетка. Сваки надгледач имплементира своју методу која се покреће када је он обавештен. Надгледач може само чувати податке, проверити неки услов, или позвати протокол безбедног завршетка над командом која га је прозвала.

4.6 Распоређивач

4.6.1 Општи концепт распоређивача

Распоређивачка компонента представља централни део система који управља извршавањем више оркестратора. Њена основна улога јесте да организује, покреће и надгледа рад појединачних јединица извршавања, водећи рачуна о њиховој међусобној независности и правилној употреби ресурса. Сваки задатак, односно објекат који треба

извршити, распоређивач додељује посебној нити, чиме се обезбеђује паралелно извршавање и боља искоришћеност процесорског времена. На тај начин систем може истовремено да обрађује више активности без међусобног чекања и блокирања. Распоређивач резервише једну нит за своје операције. Ову нит ћемо звати **распоређивачка нит.** У овом контексту реч нит се односи на нити оперативног система, у Пајтону представљене апстракцијама *Thread* і *ThreadPool.*

Јединице извршавања распоређивача су команде. Главна одговорност распоређивача јесте да позове *.execute()* методе команди које су му прослеђене смисленим редоследом, сваку на својој нити, уз поштовање додатних ограничења.

4.6.2 Ограничења, конфигурација распоређивача

Главни ресурс распоређивача јесу **нити**. Потребно је ограничити распоређивач на број нити који је доступан на серверу на ком се оркестрациони сервер извршава. На овако ограниченом систему, сви пројекти би користили исти скуп нити. У неким случајевима, ово има смисла, али постоје и ситуације у којима ово представља проблем. Узмимо за пример конфигурацију која опслужује два пројекта са ограниченим бројем нити. Ако у кратком периоду, инжењери пројекта А направе велики број промена које треба оценити, ове промене могу заузети све нити у систему. Ово би значило да ниједна промена из пројекта Б не може добити оцену.

Из овог разлога, осим глобалног ограничења, распоређивач подржава и ограничења броја нити по пројекту. Ова ограничења се чувају у конфигурационој датотеци ***scheduler_config.json***. Пројекти су наведени по именима, и наведен је њихов број нити. Ако неки пројекат није наведен у конфигурацији, сматра се да он користи све нити система, тј. за њега важи једино ограничење броја нити у целом систему.

4.6.3 Адаптивни семафор

Овако ограничење броја нити ћемо постићи имплементацијом **семафора**. Семафор је софтверска компонента која издаје ограничен број **дозвола.** Када друга компонента затражи дозволу за извршавање, семафор ће проверити да ли је број доступних дозвола већи од нуле, издати дозволу, и смањити број дозвола за један. Компонента када заврши извршавање мора обавестити одговарајући семафор, како би се дозвола вратила у скуп доступних дозвола и била поново издата.

У имплементацији семафора коришћена је класа која енкапсулира бројач дозвола и механизам за блокирање нити када ресурс није доступан. Методе *aquire()* и *release()* омогућавају контролисано приступање заједничком ресурсу. Прва смањује бројач и потенцијално зауставља нит док се ресурс не ослободи, док друга повећава бројач и

сигнализира другим нитима да могу наставити рад. На овај начин се обезбеђује синхронизација и избегавају конфликтне ситуације.

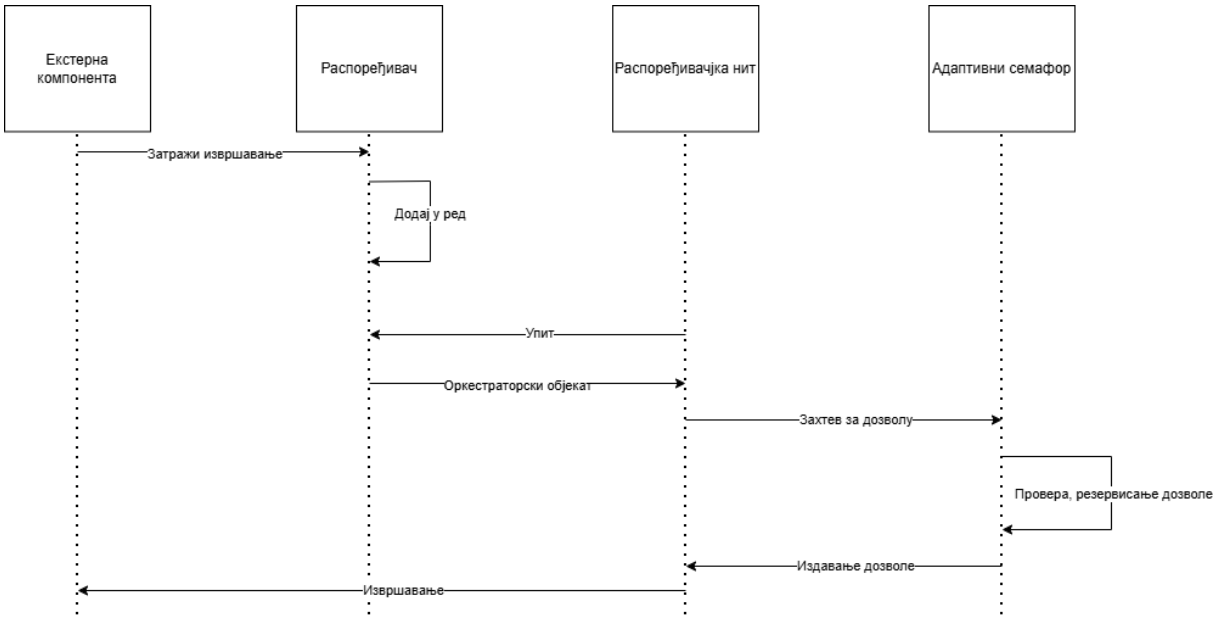
За потребе распоређивача, инстанцираћемо по један семафор за сваки пројекат, са бројем дозвола који одговара броју нити предвиђених конфигурацијом за тај пројекат. Такође ћемо инстанцирати један семафор који ће пратити максимални број дозвола (нити) у систему.

Као што је назначено у одељку 4.1.1, конфигурацију распоређивача је могуће променити у било ком тренутку, једноставном изменом датотеке на удаљеном репозиторијуму. Локална копија конфигурације ће онда бити аутоматски измењена кроз управљач конфигурације. У овом случају потребно је динамички изменити број дозвола на семафору. Имплементације семафора доступне у стандардним библиотекама најчешће не подржавају овакво динамичко мењање конфигурације. Из овог разлога имплементирамо **адаптивни семафор.** Ово постижемо проширавањем класе семафора описане у прошлом пасусу, са методама за измену броја дозвола, уз провере које онемогућавају немогућа стања (нпр. број дозвола мањи од броја тренутно заузетих дозвола).

4.6.4 Ток рада распоређивача

Било која компонента система може прозвати интерфејс распоређивача, и проследити му команду за извршавање заједно са приоритетом те команде. Објекат команде ће затим бити прослеђен **реду са приоритетима**. При популисању реда, распоређивачка нит излази из неактивног стања и почиње са радом.

Распоређивачка нит затим налази **први објекат максималног приоритета** из реда приоритета. У случају да је ово оркестраторски објекат, добавља се име његовог пројекта из пројектне конфигурације. Затим се добавља одговарајући семафор пројекта, и тражи се дозвола од њега. У случају да дозволе нема, објекат се враћа у ред, на прво место, одакле је и преузет. У случају да постоји дозвола, позива се ***.execute()*** метода објекта, и распоређује на одговарајућу нит. Слика 12 представља дијаграм тока извршавања распоређивања.



Слика 12 – дијаграм тока распоређивача

У тренутној конфигурацији пројекта, оркестраторски ток се сматра атомском јединицом извршавања. Ово значи да се токови распоређују на нити, и једном када је ток распоређен, извршава се до краја. Како се токови састоје од команди, а распоређивач ради над интерфејсом команди, било би могуће појединачно распоредити команде у оркестраторском току. У овом случају, било би могуће да се изврши једна команда, и затим поново чека слободна нит. Међутим, тај приступ би био комплескнији за праћење.

5. Резултати

5.1 Експериментална поставка

Алгоритам описан у досадашњем делу рада је примењен као део ширег компанијског алата континуалне интеграције. Овај алат је до сад коришћен на пет комерцијалних пројеката унутар фирми *RT-RK* и *Iwedia*. Само један од ових пројеката је завршен, те само за њега постоје комплетни подаци. За овај пројекат, у даљем тексту *AIS* (интерни назив пројекта), урађена је комплетна анализа перформанси и коришћења система континуалне интеграције. Ови подаци ће бити основа анализе решења, а некомплетни подаци са осталих пројеката ће бити коришћени само тамо где података за АИС нема. Обављени су и неформални разговори са инжењерима из ових тимова, поготово вођама тимова. Садржај ових разговора ће пружити додатни контекст подацима.

На свим наведеним пројектима у употреби је систем за верзионисање Герит. Инжењери су развијали софтвер локално, и потом слали измене на централни сервер. Промене су при отварању биле означене као неактивне и постајале активне на захтев инжењера кроз кориснички интерфејс Герит платформе. Преласком промене из неактивне у активну се покретао систем континуалне интеграције, а сваком додатном изменом отворене промене систем се поново покретао.

5.2 Метрике коришћења система континуалне интеграције

Табела 7 садржи податке о броју превођења по покретању алгоритма. Они нам говоре о квалитету процеса развоја софтвера. Овај број ће увек бити између један и три. У теоретском идеалном случају, када би свака промена била савршена, овај број би био један. У најгорем случају овај број би био три, што би значило да свака промена ради

једино ако је сама у ланцу. Ово би значило да се цела логика слагања ланца заобилази, свака промена се тестира сама за себе, те је алгоритам лоше пројектован. Бројке које видимо (1.6 – 1.9) су реалне и приказују окружење где се континуална интеграција користи, али није још потпуно интегрисан у процес развоја. На пример, инжењери не покрећу исти скуп тестова локално као на платформи или окружења нису савршено усклађена.

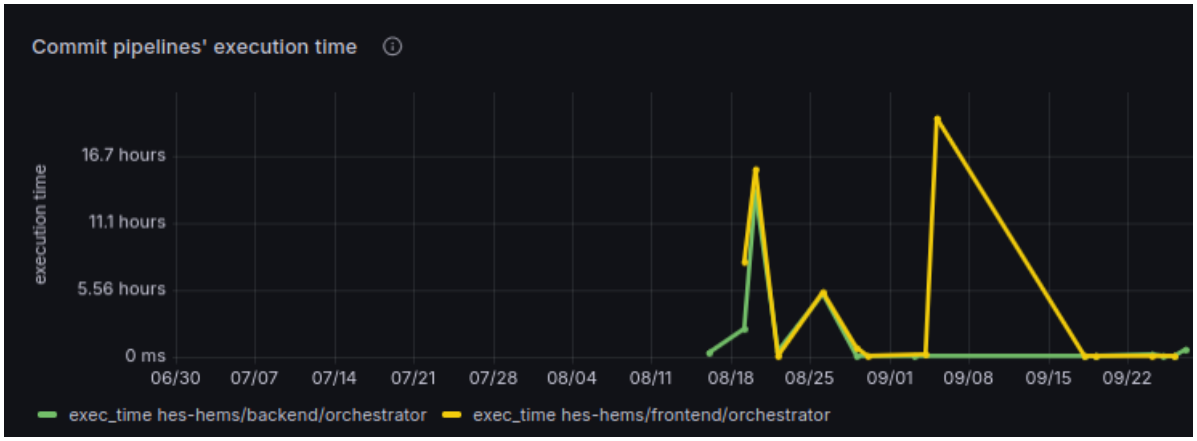
Табела 7 – метрике коришћења континуалне интеграције на пројекту АИС

Тим	*Време једног превођења*	*Промена месечно*	*Превођења по покретању алгоритма*	*Покретања алгоритма по промени*
AIS-play kotlin	~4 минута	76	1.66	4.51
AIS-play android	~4.5 минута	86	1.89	1.86
AIS-play dui	~3.5 минута	6	1.78	1.5

Друга важна метрика је број покретања алгоритма по промени. Она нам говори о томе колико је добра комуникација између система континуалне интеграције и инжењера. У случају АИС-плеј-веб тима, овај број значајно одступа од просека. Чињеница да су инжењери покретали алгоритам 4.5 пута по промени указује на неразумевање платформе. Инжењери нису довољно добро разумели повратне информације на платформи, те нису могли лако да поправе грешке. Због овога, промена је постајала активна и потом имала велики број преправки.

Резултат ове анализе је био унапређење система повратних информација. Неки примери су централизовано праћење грешака (енг. *error logging*) и увођење контролне странице где корисници могу да прате тренутно стање репозиторијума. Резултат ових промена је био значајно мањи број покретања алгоритма при промени, што је драстично смањило оптерећење система.

Слика 13 представља граф времена оркестрације на пројекту АИС, који је део контролне странице. Он указује на чињеницу да је време извршавања алгоритма углавном минимално. Међутим, постоје случајеви када оно расте. Разлог је чекање неоцењених промена. Ако дође до било какве грешке у извршавању, нпр. губитак мреже или преоптерећење сервера, промена на којој се ради у том тренутку неће бити оцењена. Ово може да изазове бесконачно чекање, где свака следећа промена чека промену која никад неће бити оцењена. Тренутно решење овог проблема је људска интервенција. У будућности ће бити имплементиран помоћни алгоритам који ће периодично проверавати да ли постоје неоцењене промене за које нема покренутих превођења.



Слика 13 – Граф времена извршавања оркестрације на пројекту АИС

Слика 14 је граф који приказује брзину спајања (енг. *Merge velocity*), током 2 месеца развоја на пројекту *AIS*. Она се дефинише као временски период од тренутка када промена постане активна, до тренутка када се промена споји на главну грану. Ово време се креће од неколико минута до 3 сата. С обзиром на чињеницу да су промене у просеку спајане за мање од 30 минута, можемо рећи да је циљно повратно време система од 30 минута постигнуто.

Слика 14 – брзина спајања на пројекту АИС

Из анализе се види да је решење успешно интегрисано у развојни процес. Инжењери активно користе систем континуалне интеграције, уз адекватну повратну спрегу са платформом. Иако перформансе још нису идеалне, нарочито у погледу чекања у систему, решење задовољава потребе тимова.

5.3 Даљи рад

Тренутни систем је осмишљен под претпоставком да ће оркестрациони сервер бити константно доступан. Као што је описано у одељку 3.1, у случају да у имплицитном ланцу зависности постоје неоцењене промене, оркестрација мора прећи у неактивни режим рада и сачекати да ове промене буду оцењене, да би се прешло у следећи корак оркестрације. У теоретској идеалној поставци система, ово није проблем, пошто ће свака промена покренути одговарајућу оркестрацију, и стога бити благовремено оцењена. Међутим, тестирање система у реалној поставци је показало да ово није увек тачно.

Реални системи могу доживети непредвиђени прекид везе, пад неког од сервера, губитак пакета, и сличне непредвидиве отказе. Реални систем би требао да буде отпоран на ове отказе. У случају система изложеног у овом раду, било која промена настала током оваквог отказа никада неће бити оцењена. Свака промена изнад ње у ланцу промена ће чекати бесконачно да та промена буде оцењена. Овај проблем је до сад решаван ручном

интервенцијом. Тривијално је лако ручно покренути оркестрацију за дату промену након што је она пропуштена, али се показало као тежак посао наћи најнижу промену у ланцу за коју оркестрација није покренута. Осим тога, откази нису увек очигледни, и у случају неприметног квара система може проћи дуже време пре него што инжењер из тима за континуалну интеграцију уочи проблем.

Из ових разлога, биће неопходно аутоматизовати процес поновног покретања за промене које настану у периоду отказа. Први би био при свакој иницијализацији оркестраторског сервера наћи све отворене неоцењене промене, и покренути појединачан оркестратор за сваку. Овакав алгоритам би решио проблем отказа оркестраторског сервера, што представља мањину свих могућих отказа.

Идеално решење за остале отказе би био некакав систем који би емитовао догађај сваки пут када дође до квара у систему. Квар у овом контексту би било довољно дефинисати као сваку оркестрацију која није резултовала валидном оценом, или безбедном терминацијом, описаном у одељку 4.4. Потом би било потребно чувати ове догађаје у некаквој перзистентној меморији, заједно са метаподацима оркестрације, и поново распоређивати оркестраторе који нису успели, периодично, све док се не дође до валидне оцене.

6. Закључак

Систем изложен у овом раду представља нетрадиционални приступ спекулативном спајању у системима континуалне интеграције, који на иновативан начин повезује процесе развоја, тестирања и имплементације софтвера. Главни циљеви система били су постизање **прилагодљивости**, **скалабилности** и **поузданости**, како би се обезбедила лакша интеграција у различита развојна окружења и прилагодљивост специфичним потребама пројеката.

Читав систем почива на комбинацији **отворених алата**, попут Jenkins сервера за аутоматизацију, Гита и Герита за контролу верзија и управљање променама, као и **затворених, прилагођених компоненти**, међу којима се издваја оркестрациони сервер развијен управо за потребе овог пројекта. Оркестрациони сервер представља централну компоненту која омогућава координацију више процеса континуалне интеграције, праћење извршавања задатака и интелигентно доношење одлука о спекулативном спајању. Као такав, систем је независан од комерцијалних лиценци и екстерне подршке, што га чини **отвореним, одрживим и економски исплативим решењем** за примену у различитим развојним тимовима.

Резултати представљени у поглављу 5. показују да је постигнута задовољавајућа спрега између система континуалне интеграције и инжењера који га користе. Анализом метрика јасно је доказано да предложени систем **значајно убрзава процес развоја**, смањује време потребно за откривање и исправку грешака и повећава **укупни квалитет софтверских решења**. На тај начин, потврђена је хипотеза постављена у уводном делу рада, да континуална интеграција, у комбинацији са прилагођеним оркестрационим сервером, представља ефикасан модел за унапређење продуктивности и поузданости у развојним процесима.

Додатна вредност предложеног система огледа се у могућности даљег проширења и интеграције са алатима за континуалну испоруку и континуално распоређивање, што отвара простор за потпуно аутоматизован циклус континуалне интеграције. Поред тога, архитектура система омогућава лако додавање нових функционалности, попут интелигентног распореда задатака, приоритизације на основу историје успешности тестова и аутоматског оптимизовања ресурса.

Будући рад може бити усмерен на унапређење механизама за спекулативно спајање, као и на интеграцију система са напредним аналитичким алатима који користе машинско учење за предвиђање исхода спајања и препознавање потенцијалних конфликата у коду. Такође, развој интерфејса за визуелизацију података и праћење метрика у реалном времену представља следећи корак ка повећању транспарентности и употребљивости система.

Сумирајући све наведено, може се закључити да предложени систем представља **значајан корак ка модернизацији процеса континуалне интеграције**, обезбеђујући стабилну основу за даљи развој и истраживање у области оркестрације процеса континуалне интеграције. Његова примена показала је да комбинација отворених технологија и прилагођених решења може резултирати моћним, поузданим и прилагодљивим системом који у потпуности одговара савременим захтевима софтверског инжењерства.

7. Литература

[1]…….. M. Shahin, M. A. Babar, and L. Zhu, “Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices,” *arXiv*, Mar. 2017.

[2]…… E. Soares, G. Sizilio, J. Santos, D. A. Costa, and U. Kulesza, “The Effects of Continuous Integration on Software Development: a Systematic Literature Review,” arXiv preprint arXiv:2103.05451, Jan. 2022.

[3]…… J. Danjou, “Announcing Speculative Merge Queues,” *Mergify Blog*, Mar. 31, 2021. https://blog.mergify.com/announcing-speculative-merge-queues/.

[4]…… Fallahzadeh, E., Bavand, A. H., & Rigby, P. C. (2023). Accelerating Continuous Integration with Parallel Batch Testing. arXiv preprint arXiv:2308.13129.

[5]…….“Why a merge commit for pull requests?” GitHub actions issue #504, May 14, 2021.:https://github.com/actions/checkout/issues/504.