УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Раде Пејановић

CLI алат за аутоматизовано креирање тестних окружења за платформу дистрибуираног рачунарства у облаку

ЗАВРШНИ РАД

Основне академске студије

Нови Сад, 2025

	УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА** 21000 НОВИ САД, Трг Доситеја Обрадовића 6	Број:
	ЗАДАТАК ЗА ЗАВРШНИ РАД	Датум:

(Податке уноси предметни наставник - ментор)

Студијски програм:	Софтверско инжењерство и информационе технологије		
Студент:	Раде Пејановић	Број индекса:	SV10/2021
Степен и врста студија:	Основне академске студије		
Област:	Електротехничко и рачунарско инжењерство		
Ментор:	Милош Симић		

НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ЗАВРШНИ РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:

- проблем – тема рада;
- начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;

НАСЛОВ ЗАВРШНОГ РАДА:

CLI алат за аутоматизовано креирање тестних окружења за платформу дистрибуираног рачунарства у облаку

ТЕКСТ ЗАДАТКА:

Имплементирати алат командне линије за аутоматизовано креирање тестних окружења за платформу дистрибуираног рачунарства у облаку отвореног кода. Акценат треба да буде на једноставном механизма за креирање тестних окружења на једном рачунару, употребом виртуалних машина.

Руководилац студијског програма:	Ментор рада:

Примерак за: □ - Студента; □ - Ментора

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**:		Дипломски - бечелор рад
Аутор, **АУ**:		Раде Пејановић
Ментор, **МН**:		Др Милош Симић, доцент
Наслов рада, **НР**:		CLI алат за аутоматизовано креирање тестних окружења за платформу дистрибуираног рачунарства у облаку
Језик публикације, **ЈП**:		српски/ћирилица
Језик извода, **ЈИ**:		српски/енглески
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)		6/44/31/3/33/0/2
Научна област, **НО**:		Електротехничко и рачунарско инжењерство
Научна дисциплина, **НД**:		Примењене рачунарске науке и информатика
Предметна одредница/Кључне речи, **ПО**:		виртуелизација, инфраструктура као код, аутоматизација, развојно окружење, Vagrant, Go, CLI алат
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		Овај документ представља упутство за писање завршних радова на Факултету техничких наука Универзитета у Новом Саду. У исто време је и шаблон за Typst.
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		01.01.2025
Чланови комисије, **КО**:	Председник:	Др Горан Сладић, редовни професор
	Члан:	Др Милан Стојков, доцент
	Члан:	
	Члан, ментор:	Др Милош Симић, доцент

Потпис ментора

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**:	
Author, **AU**:	Rade Pejanović
Mentor, **MN**:	Miloš Simić, Phd., asist. professor
Title, **TI**:	CLI tool for automated creation of test environments for distributed cloud computing platform
Language of text, **LT**:	Serbian
Language of abstract, **LA**:	Serbian/English
Country of publication, **CP**:	Republic of Serbia
Locality of publication, **LP**:	Vojvodina
Publication year, **PY**:	2025
Publisher, **PB**:	Author's reprint
Publication place, **PP**:	Novi Sad, Dositeja Obradovica sq. 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	6/44/31/3/33/0/2
Scientific field, **SF**:	Electrical and Computer Engineering
Scientific discipline, **SD**:	Applied computer science and informatics
Subject/Key words, **S/KW**:	Template, thesis, tutorial
UC	
Holding data, **HD**:	The Library of Faculty of Technical Sciences, Novi Sad
Note, **N**:	
Abstract, **AB**:	This document provides guidelines for writing final theses at the Faculty of Technical Sciences, University of Novi Sad. At the same time, it serves as a Typst template.
Accepted by the Scientific Board on, **ASB**:	
Defended on, **DE**:	01.01.2025
Defended Board, **DB**: President:	Goran Sladić, Phd., full professor
Member:	Milan Stojkov, Phd., asist. professor
Member:	
Member, Mentor:	Miloš Simić, Phd., asist. professor

Menthor's sign

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

ИЗЈАВА О НЕПОСТОЈАЊУ СУКОБА ИНТЕРЕСА

Изјављујем да нисам у сукобу интереса у односу ментор – кандидат и да нисам члан породице (супружник или ванбрачни партнер, родитељ или усвојитељ, дете или усвојеник), повезано лице (крвни сродник ментора/кандидата у правој линији, односно у побочној линији закључно са другим степеном сродства, као ни физичко лице које се према другим основама и околностима може оправдано сматрати интересно повезаним са ментором или кандидатом), односно да нисам зависан/на од ментора/кандидата, да не постоје околности које би могле да утичу на моју непристрасност, нити да стичем било какве користи или погодности за себе или друго лице било позитивним или негативним исходом, као и да немам приватни интерес који утиче, може да утиче или изгледа као да утиче на однос ментор-кандидат.

У Новом Саду, дана _____

Ментор

_____

Кандидат

_____

Садржај

Глава 1

Увод

Савремени софтверски системи све чешће користе дистрибуирану архитектуру и микросервисе [1]. Више независних компоненти са јединственим задужењем (енгл. *single responsibility*) међусобно комуницира ради остваривања заједничке функционалности. Свака компонента система је развијана и тестирана у сопственом окружењу које је конфигурисано за њене потребе [2]. Окружење представља:

- одређену дистрибуцију и верзију оперативног система (енгл. *operating system*, OS)
- предефинисан сет системских променљивих (енгл. *environment variables*)
- скуп зависности (енгл. *dependencies*) неопходних за исправно функционисање компоненте

Како би се спречиле интерференције између компоненти, односно верзија њихових зависности и подешавања системских променљивих, потребно их је изоловати. Изолација омогућава паковање компоненте у сопствено окружење у коме је гарантовано њено исправно функционисање без интерференције остатка система [3]. Изолација се постиже на два начина:

- физичка, где свака компонента система ради на посебној физичкој машини
- логичка, где више виртуелних машина дели једну физичку, а свака виртуелна машина представља појединачну компоненту система

Физичка изолација је непрактична и финансијски неисплатива. Логичка изолација омогућава бољу расподелу ресурса и економичније коришћење хардвера [3]. Најчешће се постиже кроз виртуелизацију (енгл. *virtualization*), која омогућава покретање више независних окружења на једној физичкој машини [4].

Та окружења могу бити виртуелне машине (енгл. *virtual machines*, VMs), које се ослањају на виртуелизацију хардвера и садрже сопствени оперативни систем, или софтверски контејнери (енгл. *software containers*), који деле језгро (енгл. *kernel*) оперативног система са *host* машином и виртуелизују кориснички простор (енгл. *user space*) оперативног система [4]. Софтверски контејнери омогућавају брже и лакше покретање апликација са свим зависностима и користе мање рачунарских ресурса [4]. Ручно покретање и конфигурисање више виртуелних окружења за сложене системе је дуготрајно и подложно грешкама [5].

Како би се овај процес поједноставио и верзионисао, користи се концепт инфраструктуре као кода (енгл. *Infrastructure as Code*, IaC). Конфигурација окружења се описује декларативно (нпр. YAML или JSON). То омогућава аутоматизовано креирање, управљање и репродукцију развојних окружења. Овакви приступи смањују време потребно за постављање окружења и повећавају поузданост процеса развоја и тестирања [6].

Циљ овог рада је развој алата који омогућава једноставно креирање и управљање локалним развојним окружењима применом принципа IaC у оквиру *Constellations* [7] пројекта отвореног кода (енгл. *open source*). Алат аутоматизује:

- покретање контролне равни (енгл. *control plane*) на матичној машини (енгл. *host machine*)
- покретања унапред произвољно дефинисаног броја виртуелних машина, њихово умрежавање и управљање животним циклусом (креирање, покретање, паузирање, гашење и брисање)
- подешавање окружења инсталацијом потребних зависности и конфигурисања системских променљивих
- удаљено покретање и заустављање дефинисаних сервиса на виртуелним машинама

На овај начин се убрзава процес припреме окружења за развој и тестирање, смањује могућност грешке и обезбеђује доследност окружења на различитим системима.

1.1 Структура рада

Овај рад је организован у више поглавља која воде читаоца од теоријских основа до конкретне реализације решења. Након увода, друго поглавље представља теоријске основе и релевантна постојећа истраживања, укључујући концепте виртуелизације, контејнеризације и инфраструктуре као кода, као и преглед алата и решења за аутоматизацију развојних окружења.

Треће поглавље описује функционалне захтеве система, прецизирајући шта алат треба да обезбеди и како изгледа крајње стање окружења. Четврто поглавље се бави архитектуром система, модуларним дизајном и међусобним односима компоненти. Пето поглавље обухвата имплементацију, са детаљима о развоју CLI алата.

Рад се завршава закључком, у коме се говори о резултатима, процењују предности и ограничења решења и дају смернице за будућа унапређења.

Глава 2

Теоријске основе и постојећа решења

Ово поглавље описује основне теоријске концепте неопходне за разумевање решења (поглавље 2.1), технологије коришћене у решењу (поглавље 2.2) и постојећа решења (поглавље 2.3)

2.1 Теоријски концепти релевантни за разумевање решења

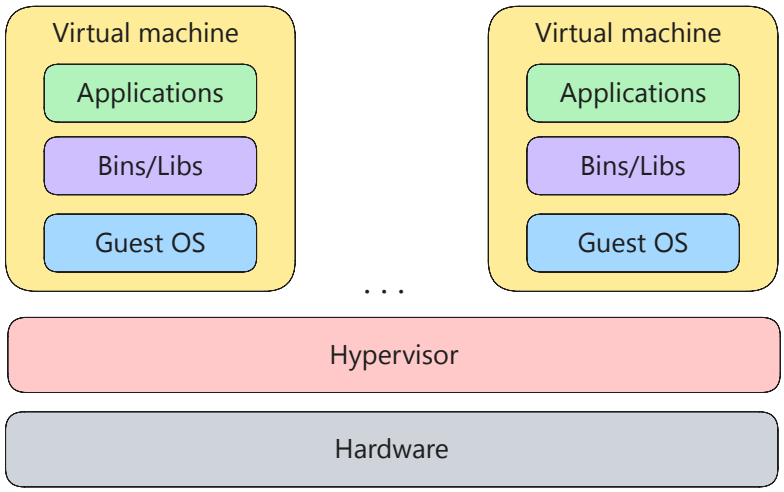
2.1.1 Виртуелизација

Виртуелизација је технологија која омогућава креирање више независних логичких окружења на једној физичкој машини. Омогућава да се један физички рачунар „подели“ на више изолованих система који функционишу као да имају сопствене ресурсе [4]. Основни системски ресурси који се виртуелизују су процесор (CPU), меморија (RAM), складиште (енгл. *storage*) и мрежни интерфејси (енгл. *network interfaces*). Основни концепти виртуелизације су виртуелне машине (енгл. *virtual machine, VM*) и хипервизори (енгл. *hipervisors*) [8].

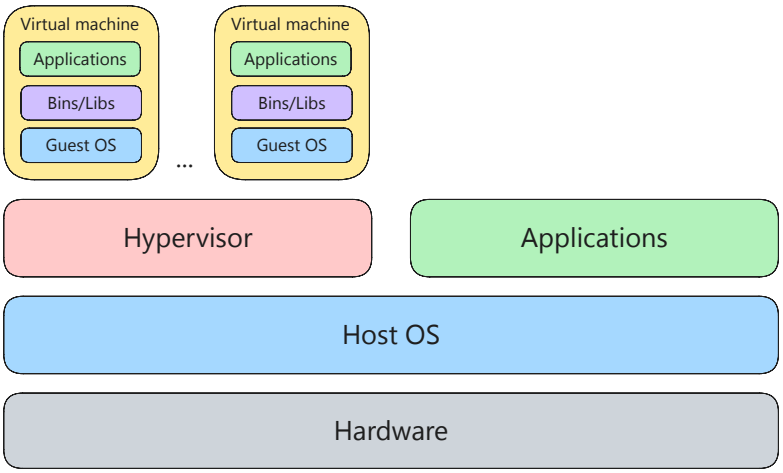
Виртуелна машина представља софтверски дефинисано изоловано окружење са сопственим оперативним системом, меморијом, процесором и мрежним интерфејсима. Ресурси виртуелне машине (енгл. *guest machine*) се добијају од физичке машине (енгл. *host machine*), али су логички издвојени тако да њихово окружење функционише независно од других виртуелних машина [9].

Хипервизор, познат и као менаџер виртуелних машина (енгл. *virtual machine monitor*, VMM), представља софтвер који омогућава деобу физичких ресурса рачунара на више виртуелних окружења. Он управља креирањем, доделом ресурса и радом виртуелних машина. Посредује између физичког хардвера (енгл. *host*) и виртуелних система (енгл. *guests*) који користе његове ресурсе [8].

Постоје две врсте хипервизора. Тип 1 (енгл. *native, bare-metal*) на слици 1, инсталира се директно на хардвер уместо оперативног система. Ресурси виртуелних машина директно се мапирају на хардвер. Овај тип је популаран у ентерпрајз окружењима и серверским системима. Тип 2 (енгл. *hosted*) на слици 2, инсталира се као апликација на постојећи *host* оперативни систем. Ресурси и инструкције виртуелних машина најпре се мапирају на *host* оперативни систем који их потом прослеђује хардверу [8], [9].



Слика 1: Хипервизор тип 1 (*native, bare-metal*).



Слика 2: Хипервизор тип 2 (*hosted*).

Бенефити виртуализације се огледају у:

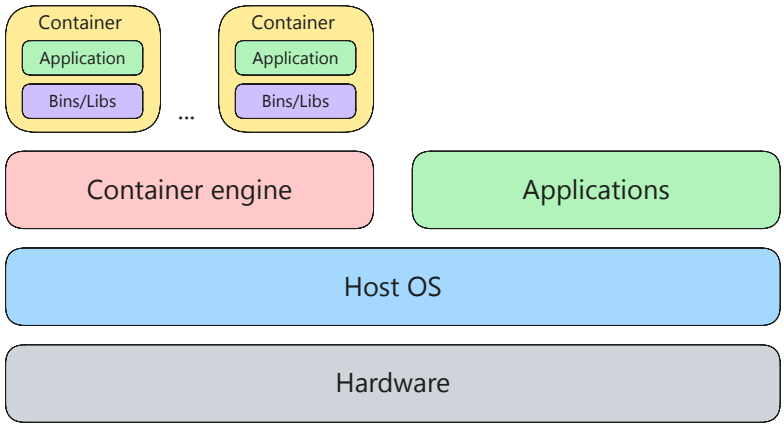
- боље искоришћење физичког хардвера
- смањењу времена неисправности (енгл. *downtime*), у случају неке катастрофе или неочекиваног гашења физичког рачунара, виртуализована окружења се лакше и брже премештају и постављају на друге физичке машине
- повећању ефикасности и продуктивности тимова који одржавају инфраструктуру
- једноставнијем тестирању и креирању тест окружења која су идентична продукционим
- еколошком унапређењу и штедњи електричне енергије [10]

2.1.2 Контејнеризација

Контејнеризација представља технологију која омогућава покретање апликација у изолованим окружењима под називом софтверски контејнери (енгл. *software containers*). За разлику од виртуелних машина, које емулирају цео хардвер и покрећу сопствени оперативни систем, софтверски контејнери деле језгро (енгл. *kernel*) *host* оперативног система и користе заједничке системске ресурсе, али су логички изоловани један од другог. Оваква архитектура омогућава значајно мању потрошњу ресурса, брже покретање и лакше распоређивање апликација у различитим окружењима [4].

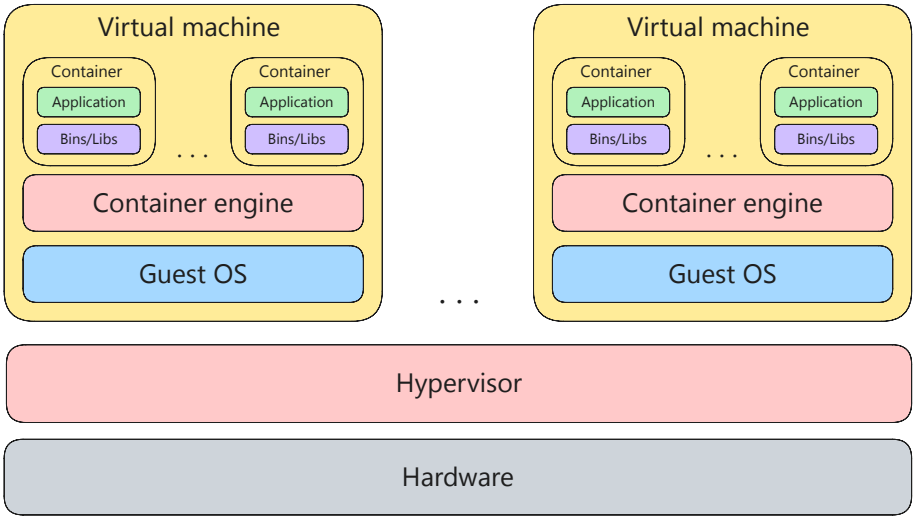
Софтверски контејнер је стандардизована јединица софтвера која обједињује апликацију и све њене зависности, библиотеке, конфигурационе датотеке и системске променљиве у један преносиви пакет. Захваљујући томе, апликација ће се понашати исто без обзира на то да ли се извршава на локалном рачунару, серверу или облаку [11].

Главна предност контејнеризације у односу на класичну виртуелизацију јесте њена ефикасност [4]. Будући да се више софтверских контејнера може покретати на једном оперативном систему без потребе за додатним хипервизором, трошак меморије и процесорских ресурса је знатно мањи, а време покретања апликација драстично краће. Сваки софтверски контејнер се извршава као посебан процес у корисничком простору (енгл. *user space*), док све виртуелне машине захтевају покретање комплетног оперативног система, што их чини знатно „тежим“ и споријим. Овакав приступ омогућава већу густину апликација по систему и једноставније оркестрирање [11].



Слика 3: Контејнеризација.

Иако представљају различите технологије, софтверски контејнери и виртуелне машине се често користе заједно, јер се њихове предности међусобно допуњују [12]. Виртуелне машине обезбеђују пуну изолацију и сигурност на нивоу хардвера, док софтверски контејнери пружају брзину, лакоћу распоређивања и скалабилност апликација. Комбиновањем ова два приступа (слика 4) могуће је покретати више контејнеризованих апликација унутар једне виртуелне машине, чиме се постиже већа флексибилност, боља искоришћеност ресурса и лакше управљање у хетерогеним окружењима [13].



Слика 4: Комбинација виртуелних машина и софтверских контејнера.

2.1.3 Инфраструктура као код

Инфраструктура као код (енгл. *Infrastructure as Code*, IaC) представља начин аутоматизације подизања и управљања инфраструктуром. Обухвата конфигурисање, постављање (енгл. *deployment*) и управљање виртуелним окружењима на физичким машинама. Виртуелна окружења могу бити виртуелне машине или софтверски контејнери, а њихова конфигурација се описује машински читљивим кодом [6].

Најчешћи формати дефинисања конфигурације јесу YAML и JSON датотеке. Конфигурацјију представљају дефиниције виртуелних машина, софтверских контејнера, мрежних ресурса, складишта и зависности апликација [6].

IaC се може реализовати на два начина:

- декларативно (енгл. *declarative*), дефинише се жељено стање у ком систем треба да се налази, а IaC алат сам закључује како да га постигне [14] (“желим 3 сервера са следећом конфигурацијом” [15])

- императивно (енгл. *imperative*), дефинише се сет инструкција (енгл. *playbooks*) које IaC алат треба да испрати корак по корак [14] ("прво креирај сервер, онда инсталирај софтвер, па конфигуриши подешавања" [15])

Представљање инфраструктуре помоћу кода омогућава примену свих концепата који су до сада били незаобилазни у развоју софтвера, као што су верзионисање, ревизија и праћење промена, примена агилних метода, аутоматизовано тестирање и интеграција. Инфраструктурне промене је могуће третирати на исти начин као промене у апликацијском коду, што повећава поузданост, репродуктивност и лакоћу одржавања окружења [6].

2.1.4 Рачунарство у облаку

Рачунарство у облаку (енгл. *cloud computing*) представља испоруку рачунарских ресурса и складишних капацитета као услуге преко интернета. Релизује се тако што корисник приступа удаљеним серверима складиштеним у огромним центрима података (енгл. *data centers*) у власништву провајдера услуге. На овај начин елиминише се потреба за локалним хардвером и омогућава скалабилност, висока доступност и флексибилност коришћења ресурса [13].

Главна мана оваквог приступа јесте центрлизовано управљање комплетним рачунарским ресурсима (рачунарском снагом и складиштеним подацима) које доводи до следећих проблема [16]:

- кашњење (енгл. *latency*) у трансферу и обради података, као последица удаљености између крајњег корисника и облака
- зависност од интернет конекције, без које корисник нема приступ сопственим подацима и рачунарским ресурсима у облаку
- загушеност мреже (енгл. *bandwidth load*), све популарнији паметни системи (енгл. *internet of things*, IoT), паметни аутомобили, авиони и остали уређаји који шаљу континуиране метрике на облак додатно оптерећују мрежу
- безбедносни ризици, подаци се обрађују и складиште на удаљеним серверима што отвара питање приватности и контроле
- регулаторна и правна ограничења, неке организације не смеју да складиште податке ван државе или сопственог система

2.1.5 Рачунарство на ивици

Ови изазови постају све израженији са растом броја повезаних уређаја и потребом за обрадом података у реалном времену, што је подстакло развој нових концепата као што је рачунарство на ивици (енгл. *edge computing*). Суштина овог приступа јесте премештање рачунарских ресурса ближе крајњим корисницима, односно изворима података који се налазе на ивици мреже (енгл. *network edge*). На тај начин се смањује кашњење у комуникацији, побољшава одзив система и омогућава ефикаснија обрада података у децентрализованом окружењу [17].

Развој рачунарства на ивици не замењује класично рачунарство у облаку, већ га допуњује и омогућава координисан рад две парадигме. Рачунарство у облаку и даље има кључну улогу у централизованој обради великих количина података, дубинској аналитици и дугорочним одлукама, док рачунарство на ивици преузима обраду података ближе извору, смањује кашњење, оптерећење мреже и омогућава обраду у реалном времену. У пракси, чворови на ивици (енгл. *edge nodes*) могу локално обрадити и филтрирати податке, а резултате слати у облак за даљу анализу [17].

2.1.6 Микро рачунарство у облаку

У последњих неколико година све је популарније постављање мањих *data center*-а на ивици мреже. Географски распоређени сервери мањих размера, организовани у оквиру микро рачунарства у облаку (енгл. *micro clouds*). Инфраструктура се дефинише независно од физичких машина и њихових опера-

тивних система, применом IaC-а и концепата виртуелизације и контејнеризације. Такав приступ омогућава ефикасније коришћење ресурса чак и у хетерогеним кластерима [18].

2.2 Технологије коришћене у решењу

Алат је развијен у *Go* програмском језику. Кориснички интерфејс је реализован као CLI апликација уз употребу библиотеке *Cobra* [19], која омогућава једноставно дефинисање команди и аргумената. Архитектура система је модуларна, састоји се од централног (енгл. *core*) дела који дефинише опште интерфејсе и позадинских модула који могу бити дефинисани од стране корисника и прикључени у алат помоћу конфигурационе датотеке.

У склопу алата имплементиран је један позадински модул који користи *Vagrant* [20] и *go-vagrant* [21] библиотеку за подизање и управљање животним циклусом чворова. Конфигурациони фајл је дефинисан у YAML формату, док су скрипте за провизионисање (енгл. *provisioning*) и покретање сервиса реализоване као *Shell* скрипте. *VirtualBox* [22] хипервизор је коришћен као провајдер виртуелизације. *Control plane* и сервиси покретани на виртелним машинама су спаковани у *Docker* [23] софтверске контејнере и њихово покретање је вршено помоћу *Docker Compose* [24] алата.

2.2.1 *Cobra*

Cobra је *Go* библиотека отвореног кода за креирање CLI апликација. Омогућава дефинисање команди, подкоманди и аргумената. Подржава обавезна и опциона поља, као и логичку организацију команди у хијерархију. У алату се користи за имплементацију интерфејса који омогућава покретање и управљање виртуелним окружењима и сервисима на њима.

2.2.2 *Vagrant*

Vagrant је алат отвореног кода за аутоматизовано креирање и управљање виртуелним машинама. Омогућава дефинисање конфигурације виртуелних окружења у машински читљивим фајловима (енгл. *Vagrantfile*), а затим аутоматско подизање, провизионисање и уништавање тих окружења. Подржава различите хипервизоре као што су *VirtualBox*, *VMware* и други. Омогућава доследно и преносиво развојно окружење за више платформи .

Vagrantfile се пише у *Ruby* програмском језику. У њему се одређују основне карактеристике виртуелних окружења, као што су хипервизор, системски ресурси, мрежна подешавања и скрипте или алати за провизионирање. Такође омогућава аутоматско извршавање скрипти за инсталацију зависности и подешавања окружења. Могуће је дефинисање конфигурације за произвољан број окружења.

Vagrant долази са унапред подешеним опцијама за удаљени приступ виртуелним окружењима помоћу SSH протокола.

2.2.3 *VirtualBox*

VirtualBox је хипервизор типа 2 (енгл. *hosted*), инсталира се на *host* оперативни систем као апликација. Нуди емулацију хардвера и подршку за различите *guest* системе. Корисник може да покрене *guest* оперативни системе као независне виртуелне машине у оквиру *host*-а.

Поред основне виртуелизације, *VirtualBox* подржава више мрежних режима (NAT, *bridged*, *host-only*), дељење директоријума и пренос датотека између *host* и *guest* система, као и подршку за USB уређаје и снимке стања (енгл. *snapshots*) који омогућавају враћање виртуелне машине у претходно стање.

У оквиру овог алата, *VirtualBox* служи као хипервизор и провајдер виртуелних машина у склопу *Vagrant*-а.

2.2.4 *Docker* и *Docker Compose*

Docker је платформа за контејнеризацију. Омогућава паковање апликације и свих њених зависности у преносиви софтверски контејнер. Софтверски контејнери се извршавају из *Docker* слика (енгл. *images*),

које представљају шаблоне са свим неопходним зависностима, оперативним окружењем, библиотекама и конфигурацијама за покретање апликације. *Docker* слике се дефинишу помоћу *Dockerfile* датотека. *Docker* такође подржава употребу дељених директоријума (енгл. *volumes*) за трајно складиштење података изван животног циклуса софтверског контејнера.

Docker Compose је алат који омогућава дефинисање и управљање више повезаних *Docker* софтверских контејнера као једним системом. Конфигурација се дефинише у YAML датотеци (`docker-compose.yml`), у којој се описују појединачни сервиси, њихове слике, мрежне поставке и дељени директоријуми. Овакав приступ поједностављује оркестрацију комплексних апликација које се састоје од више међусобно зависних софтверских контејнера и омогућава покретање целокупног окружења једном командом.

Коришћени су у склопу *Shell* скрипти за провизионирање, за покретање претходно докеризованих сервиса из *Constellations* пројекта. Такође се користе и за покретање *Control plane*-а на *host* машини.

2.3 Постојећа решења

У овом поглављу је обрађено неколико неколико постојећих алата који омогућавају аутоматизовање покретања и управљање виртуелним окружењем. Ниједно од предложених решења не одговара на функционалне захтеве проблема у целости или то ради на сувише комплексан начин.

2.3.1 *Terraform*

Terraform [25] је IaC алат који омогућава декларативно управљање инфраструктуром. Користи сопствени језик HCL (енгл. *HashiCorp Configuration Language*) за опис жељеног стања система, што омогућава да алат сам израчуна кораке потребне за постизање тог стања. *Terraform* може да креира и управља ресурсима на различитим провајдерима, као што су платформе у облаку (AWS, *Azure*, GCP) или локални провајдери као што је *VirtualBox*.

2.3.2 *Ansible*

Ansible [26] је IaC алат за аутоматизацију конфигурације и управљање системима. Користи декларативне YAML *playbook*-ове за дефинисање корака које треба извршити на једном или више *host*-ова, као што су инсталација софтвера, копирање фајлова и покретање скрипти. *Ansible* се ослања на SSH за повезивање са *host*-овима и не захтева инсталиран агент на њима, што га чини погодним за управљање постојећим серверима или виртуелним машинама.

Глава 3

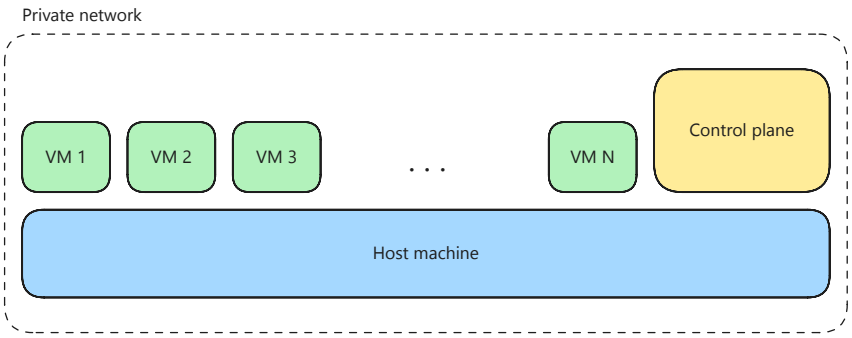
Функционални захтеви система

Потребно је креирати алат за покретање унапред произвољно дефинисаног броја виртуелних машина одједном. Свака виртуелна машина треба да има инсталирану лагану (енгл. *lightweight*) дистрибуцију *Linux* оперативног система. Конфигурација мреже самих машина треба да буде таква да машине буду међусобно видљиве, да виртуелне машине вуду видљиве са *host* машине, као и да *host* машина буде видљива из виртелних машина (слика 5).

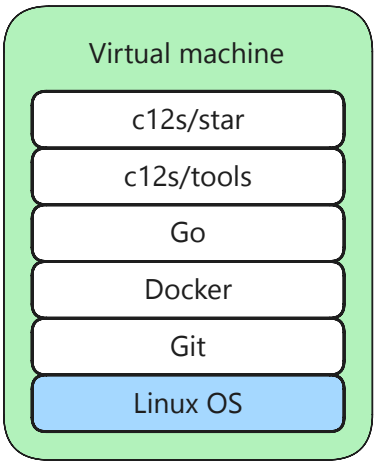
На свакој машини је потребно инсталирати систем за контролу верзија (енгл. *Version Control System*, VCS) *Git* [27], *Docker* и *Go* програмски језик, који представљају основно развојно окружење за компоненте *Constellations* пројекта. Потребно је клонирати `c12s/tools` [28] репозиторијум који садржи опште алате у оквиру пројекта *Constellations*. Након тога потребно је покренути инсталационе скрипте из `c12s/tools` репозиторијума и подесити параметре окружења како би се инсталирали неопходни сервиси. Након инсталације потребно је покренути `c12s/star` [29] сервис на свакој машини (слика 6).

На *host* машини је потребно инталирати и покренути *control plane* која обухвата скуп сервиса у склопу *Constellations* пројекта који се такође ослањају на употребу `c12s/tools` репозиторијума. Након покретања виртуелних машина и подизања неопходне инфраструктуре тестирање се врши путем `c12s/cockpit` [30] алата који се налази у оквиру *control plane*-а.

Конфигурација окружења треба да се дефинише помоћу YAML датотеке. Алат такође треба да подржи неколико различитих дистрибуција *Linux* оперативног система (са и без графичког интерфејса).



Слика 5: Инфраструктура локалног развојног окружења.



Слика 6: Поставка система на виртуелној машини.

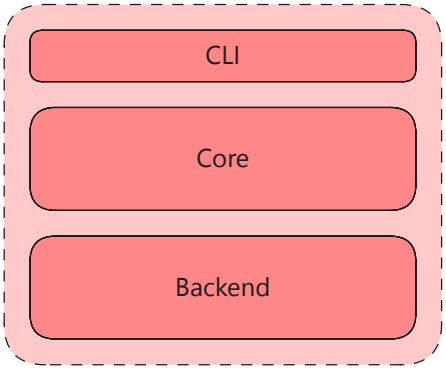
Глава 4

Архитектура система

Ово поглавље приказује преглед концептуалног модела решења (поглавље 3.1) као и преглед свих фукционалности решења (поглавље 3.1). Решење је реализовано као CLI алат који се компајлира у бинарну датотеку и на тај начин пружа сет команди за интеракцију са самим алатом.

4.1 Концептуални модел решења

Архитектура система заснива се на модуларном дизајну који омогућава једноставно проширивање функционалности и јасно раздвајање одговорности између компоненти. Систем је организован у три логичка слоја: кориснички интерфејс (CLI), језгро система (*Core*) и механизам за управљање виртуелним машинама (*Backend*) (слика 7).



Слика 7: Архитектура решења на највишем нивоу.

CLI слој представља улазну тачку система. Задужен је за обраду корисничких команди и параметара конфигурације, као и за комуникацију са *Core* слојем који извршава одговарајуће операције.

Core слој чини централни део система и дефинише скуп функционалности (енгл. *interface*) које сваки *Backend* мора да имплементира. Он управља животним циклусом виртуелних машина, оркестрира операције над њима и обезбеђује апстракцију између CLI и *Backend* слојева.

Backend слој садржи конкретну имплементацију механизма за креирање и управљање виртуелним машинама. Тренутно је реализован преко *VagrantBackend*-а, али архитектура омогућава додавање других провајдера без измене осталих делова система.

Захваљујући оваквој организацији, систем је лако проширив и прилагодљив различитим окружењима. Сваки слој је независан, што омогућава развој и тестирање компоненти без међусобне зависности и поједностављује интеграцију нових функционалности.

4.2 Функционалности решења

У овом раду анализира се ток функционалности заснован на постојећој имплементацији *Backend*-а, која је реализована као омотач око *Vagrant*-а. Сви даљи описи односе се на овај конкретан *VagrantBackend*.

Свака функционалност одговара одређеној команди која се извршава из командне линије и активира специфичан ток извршавања унутар система. У табели 1 приказане су све подржане команде и њихови описи.

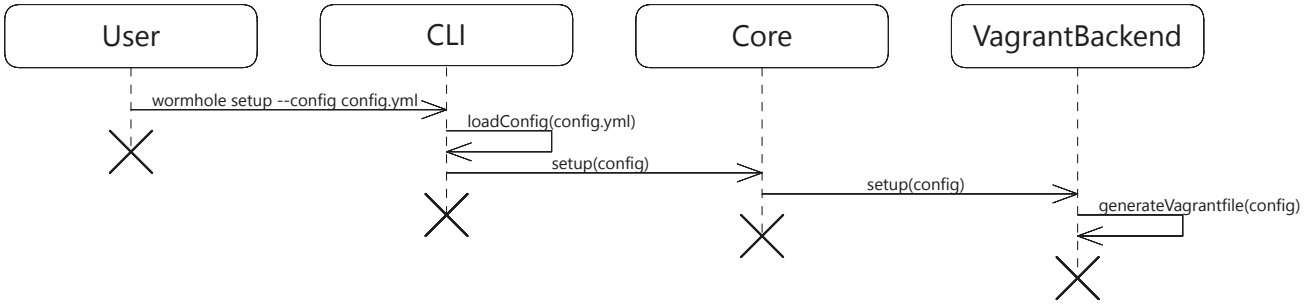
Табела 1: Преглед функционалности Wormhole алата

Функционалност	Команда	Опис
Иницијализација окружења	`wormhole setup --config <path>`	Учитава YAML конфигурациони фајл и припрема дефиницију виртуелних машина.
Креирање виртуелних машина	`wormhole create [node...]`	Креира једну или више виртуелних машина.
Заустављање виртуелних машина	`wormhole stop [node...]`	Суспендује једну или више виртуелних машина.
Наставак рада виртуелних машина	`wormhole resume [node...]`	Наставља рад једне или више виртуелних машина након суспендовања.
Поновно креирање и покретање виртуелних машина	`wormhole reload [node...]`	Поново креира виртуелне машине на основу конфигурације.
Гашење виртуелних машина	`wormhole shutdown [node...]`	Гаси виртуелне машине на контролисан начин, чувајући податке.
Брисање виртуелних машина	`wormhole destroy [node...]`	Брише једну или више виртуелних машине и све њихове податке.
Покретање сервиса	`wormhole start-nodes [node...]`	Покреће извршавање сервиса на виртуелним машинама.
Заустављање сервиса	`wormhole stop-nodes [node...]`	Зауставља извршавање сервиса на виртуелним машинама.
Покретање *control plane*-а	`wormhole start-control`	Иницијализује *control plane* сервисе на *host* машини.
Заустављање *control plane*-а	`wormhole stop-control`	Зауставља *control plane* сервисе на *host* машини.

4.2.1 Иницијализација окружења

Процес иницијализације окружења представља полазну тачку рада *Wormhole* алата. Сврха ове функционалности је да на основу конфигурационе датотеке (табела 2) припреми све потребне параметре за креирање виртуелних машина и мрежно повезивање. Резултат овог корака је окружење спремно за покретање.

Корисник прво уноси команду `wormhole setup --config config.yml` у командну линију. CLI слој учитава конфигурациону датотеку на прослеђеној путањи и позива функцију за иницијализацију окружења из *Core* слоја. *Core* мапира позив на одговарајући *Backend* (*VagrantBackend* у овом случају). *VagrantBackend* генерише *Vagrantfile* на основу прослеђене конфигурације (слика 8).



Слика 8: Дијаграм секвенце функционалности иницијализације окружења.

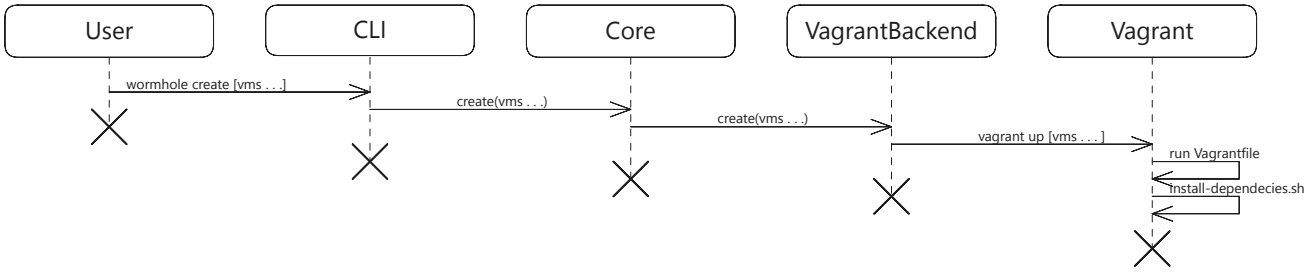
Табела 2: Структура конфигурационе YAML датотеке.

Параметар	Опис
vmCount	Број виртуелних машина које ће бити креиране.
osDistro	Идентификатор слике оперативног система.
osVersion	Верзија слике оперативног система.
cpus	Број процесорских језгара додељених свакој виртуелној машини.
memory	Количина RAM меморије (у MB) додељена свакој виртуелној машини.
gui	Булова вредност, `true` ако виртуелна машина треба да буде покренута са GUI прозором, `false` за *headless* режим.
ipBase	Почетни део IP адресе који се користи за генерисање адреса виртуелних машина.
guestPort	Порт унутар виртуелне машине за сервис који треба да буде откривен.
hostPortBase	Полазни *host* порт који се користи за мапирање *guestPort*-а виртуелних машина.
nameBase	Префикс *hostname*-а.
backendType	Тип *Backend*-а који ће управљати виртуелним машинама.

4.2.2 Креирање виртуелних машина

Процес креирања виртуелних машина представља следећи корак након успешне иницијализације окружења. Задатак ове функционалности је да на основу параметара дефинисаних у *Vagrantfile*-у (генерисаном током иницијализације), аутоматски изгради и покрене једну или више виртуелних машина које чине радно окружење.

Корисник покреће команду `wormhole create [node ...]` са опционим аргументом назива виртуелних машина уколико жели да креира само одређене виртуелне машине из конфигурације. CLI слој прихвата захтев и прослеђује га *Core* модулу, који затим позива одговарајући *VagrantBackend*, задужен за покретање виртуелних машина путем *Vagrant* API-ја.



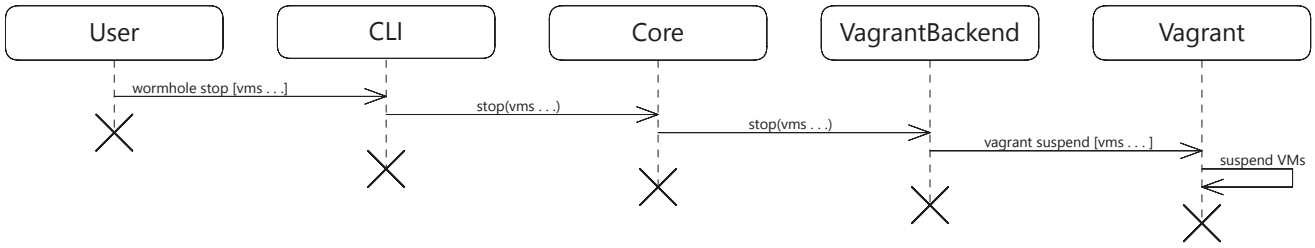
Слика 9: Дијаграм секвенце функционалности креирања виртуелних машина.

Успешним извршавањем ове функционалности, окружење добија све дефинисане виртуелне машине са инсталираним сервисима и зависностима, спремним за даље покретање (слика 9).

4.2.3 Заустављање виртуелних машина

Функционалност заустављања виртуелних машина омогућава контролисано суспендовање радног окружења ради ослобађања системских ресурса или припреме за касније настављање рада. Циљ овог корака је да се покренуте виртуелне машине у оквиру дефинисаног окружења доведу у стање мировања без губитка података или промене конфигурације.

Корисник покреће команду `wormhole stop [node ...]` са опционим аргументом који одређује једну или више виртуелних машина за заустављање. Уколико аргумент није наведен, *Wormhole* CLI аутоматски зауставља све машине дефинисане у конфигурацији. CLI слој прослеђује захтев *Core* модулу, који позива методу из *VagrantBackend*-а задужен за суспензију виртуелних машина путем *Vagrant* API-ја (слика 10).



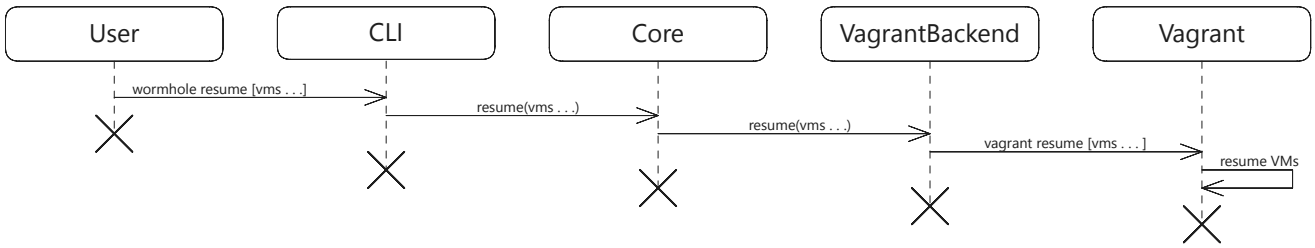
Слика 10: Дијаграм секвенце функционалности заустављања виртуелних машина.

Извршавањем ове функционалности виртуелне машине прелазе у стање *suspended*, задржавајући тренутно стање меморије и система, што омогућава брзо настављање рада у наредном кораку (*resume*).

4.2.4 Наставак рада виртуелних машина

Функционалност наставка рада виртуелних машина омогућава поновно активирање претходно суспендованих виртуелних окружења. Сврха овог корака је да се систем врати у стање у којем је био пре заустављања, како би се наставио рад без потребе за поновним креирањем или иницијализацијом окружења.

Корисник покреће команду `wormhole resume [node ...]` са опционим аргументом који одређује једну или више виртуелних машина које треба поново покренути. Уколико аргумент није наведен, Wormhole CLI аутоматски наставља рад свих виртуелних машина дефинисаних у конфигурацији. CLI слој прослеђује захтев *Core* модулу, који затим позива методу из *VagrantBackend*-а задужену за наставак рада виртуелних машина путем *Vagrant* API-ја (слика 11).



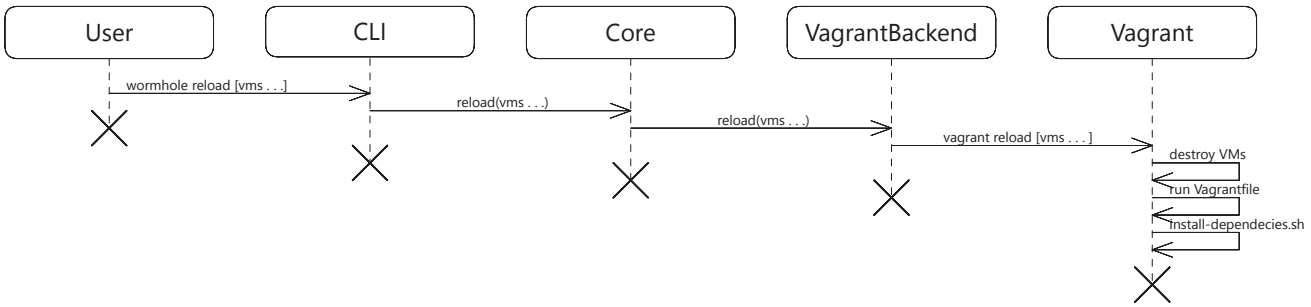
Слика 11: Дијаграм секвенце функционалности наставка рада виртуелних машина.

Извршавањем ове функционалности виртуелне машине се враћају из стања *suspended* у активно, при чему се задржава претходно стање система и подаци унутар сваке машине. На тај начин омогућен је континуитет развојног окружења без потребе за поновним покретањем сервиса.

4.2.5 Поновно креирање и покретање виртуелних машина

Функционалност поновног креирања и покретања виртуелних машина омогућава обнављање окружења на основу најновије конфигурације дефинисане у *Vagrantfile*-у. Сврха овог корака је да се обезбеди освежавање система након измена у конфигурационој датотеци или ажурирања софтверских компоненти, без потребе за ручним брисањем и поновним подизањем окружења.

Корисник покреће команду `wormhole reload [node ...]` са опционим аргументом који одређује једну или више виртуелних машина које треба поново креирати и покренути. Уколико аргумент није наведен, *Wormhole* CLI извршава поступак над свим машинама дефинисаним у конфигурацији. CLI слој прослеђује захтев *Core* модулу, који затим позива методу из *VagrantBackend*-а задужену за поновно креирање и покретање виртуелних машина путем *Vagrant* API-ја (слика 12).



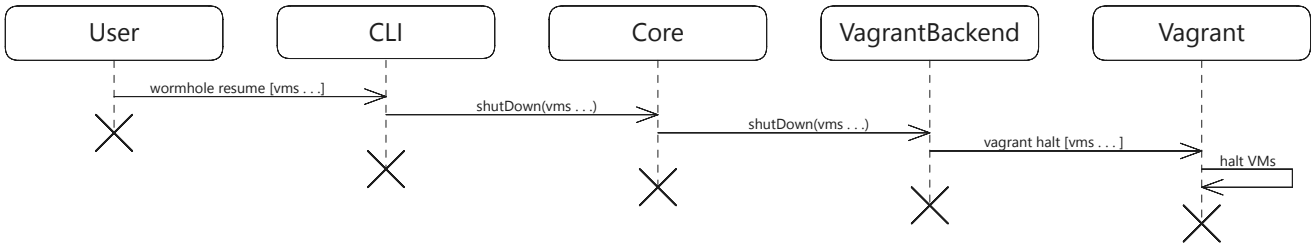
Слика 12: Дијаграм секвенце функционалности поновног креирања и покретања виртуелних машина.

Извршавањем ове функционалности све виртуелне машине се гасе и затим поново покрећу, при чему се ажурирају сви параметри дефинисани у конфигурационој датотеци.

4.2.6 Гашење виртуелних машина

Функционалност гашења виртуелних машина омогућава контролисано искључивање радног окружења. За разлику од суспендовања (*stop*) где се стање меморије чува, гашењем се машине у потпуности искључују, што омогућава ослобађање свих системских ресурса и чини овај корак погодним када се окружење више не користи у текућој сесији.

Корисник покреће команду `wormhole shutdown [node ...]` са опционим аргументом који одређује једну или више виртуелних машина које треба искључити. Уколико аргумент није наведен, Wormhole CLI извршава поступак над свим машинама дефинисаним у конфигурацији. CLI слој прослеђује захтев *Core* модулу, који затим позива методу из *VagrantBackend*-а задужену за гашење виртуелних машина путем *Vagrant* API-ја (слика 13).



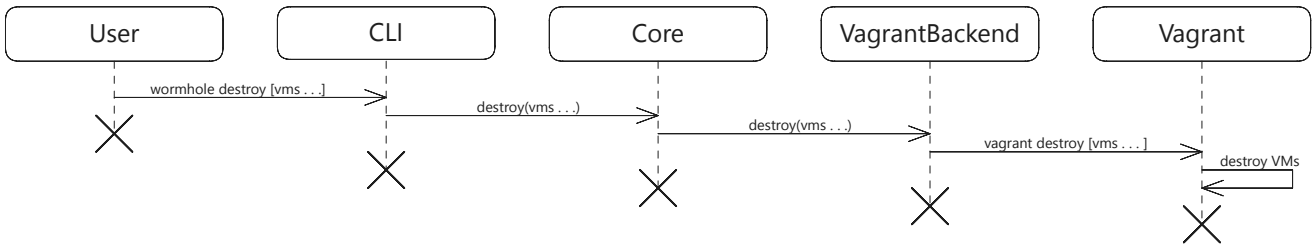
Слика 13: Дијаграм секвенце функционалности гашења виртуелних машина.

Извршавањем ове функционалности све виртуелне машине се уредно искључују, при чему се њихово стање не чува у меморији. Овај корак представља крај активне сесије рада у *Wormhole* окружењу и претходи евентуалном уништавању окружења (*destroy*).

4.2.7 Брисање виртуелних машина

Функционалност брисања виртуелних машина представља завршни корак у управљању радним окружењем. Њена сврха је трајно уклањање свих виртуелних машина и њихових података, чиме се систем враћа у почетно стање и ослобађају се сви заузети ресурси. Овај корак се најчешће користи након завршетка развојног или тестног циклуса када више није потребно задржати претходно окружење.

Корисник покреће команду `wormhole destroy [node ...]` са опционим аргументом који одређује једну или више виртуелних машина које треба поново креирати и покренути. Уколико аргумент није наведен, *Wormhole* CLI извршава поступак над свим машинама дефинисаним у конфигурацији. CLI слој прослеђује захтев Core модулу, који затим позива методу из *VagrantBackend*-а задужену за уништавање виртуелних машина и уклањање њихових придружених ресурса путем *Vagrant* API-ја (слика 14).



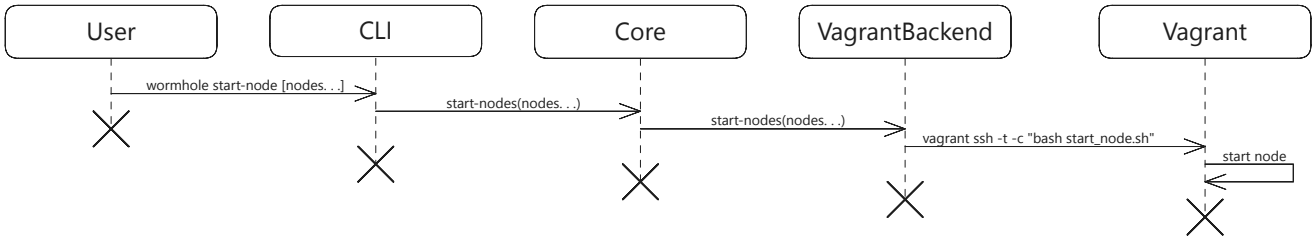
Слика 14: Дијаграм секвенце функционалности брисања виртуелних машина.

Извршавањем ове функционалности све виртуелне машине и њихови повезани подаци се трајно бришу. Након овог корака, систем остаје без активних инстанци, што омогућава поновну иницијализацију окружења од самог почетка или потпуно искључивање развојне инфраструктуре.

4.2.8 Покретање сервиса на чворовима

Функционалност покретања чворова представља корак у којем се активирају сервиси унутар сваке виртуелне машине дефинисане у окружењу. Циљ овог процеса је да се након успешног креирања и покретања инфраструктуре омогући подизање софтверских компоненти које чине радне чворове система.

Корисник покреће команду `wormhole start-nodes [node ...]` са опционим аргументом којим може ограничити извршавање само на поједине чворове. CLI слој прихвата захтев и прослеђује га *Core* модулу, који позива методу из *VagrantBackend*-а задужену за даљу оркестрацију. За разлику од претходних функционалности, *VagrantBackend* у овом случају не управља директно животним циклусом виртуелних машина, већ користи механизам удаљеног приступа путем `vagrant ssh` команде ради извршавања скрипте за покретање сервиса унутар сваке машине (слика 15).



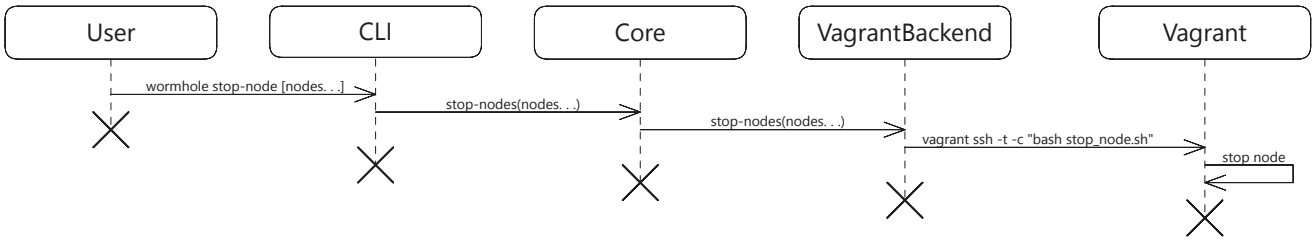
Слика 15: Дијаграм секвенце функционалности покретања сервиса на чворовима.

Извршавањем ове функционалности, сви дефинисани чворови у окружењу покрећу своје сервисе и успостављају комуникацију са *control plane*-ом, чиме окружење постаје потпуно оперативно и спремно за даљи развој и тестирање.

4.2.9 Заустављање сервиса на чворовима

Функционалност заустављања чворова омогућава контролисано заустављање свих сервиса покренутих унутар виртуелних машина. Сврха овог корака је да се радни чворови система уредно искључе, како би се спречили губитак података и неконзистентност током гашења или реконфигурације окружења.

Корисник покреће команду `wormhole stop-nodes [node ...]` са опционим аргументом којим може ограничити извршавање само на поједине чворове. CLI слој прихвата захтев и прослеђује га *Core* модулу, који позива одговарајућу методу из *VagrantBackend*-а. Као и код покретања чворова, *VagrantBackend* користи механизам удаљеног приступа путем `vagrant ssh` команде ради извршавања скрипте за гашење сервиса унутар сваке машине (слика 16).



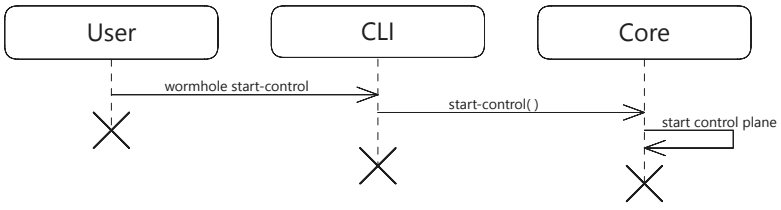
Слика 16: Дијаграм секвенце функционалности заустављања сервиса на чворовима.

Извршавањем ове функционалности, сви активни чворови у окружењу уредно заустављају своје сервисе и прекидају комуникацију са *control plane*-ом, чиме се систем доводи у стабилно и безбедно стање мировања.

4.2.10 Покретање *control plane*-а

Функционалност покретања *control plane*-а представља корак у којем се иницијализује централни део система задужен за управљање и координацију свих чворова унутар окружења. Сврха овог процеса је успостављање сервиса који омогућавају надзор, комуникацију и оркестрацију између различитих компоненти *Constellations* пројекта.

Корисник покреће команду `wormhole start-control`, чиме се иницира процес подизања *control plane*-а на *host* машини. CLI слој прихвата захтев и прослеђује га *Core* модулу, који у овом случају директно извршава скрипту за покретање сервиса без посредовања *VagrantBackend*-а. Ова скрипта, која се налази у оквиру `c12s/tools` репозиторијума, покреће све неопходне контејнере и сервисе у *Docker* окружењу (слика 17).



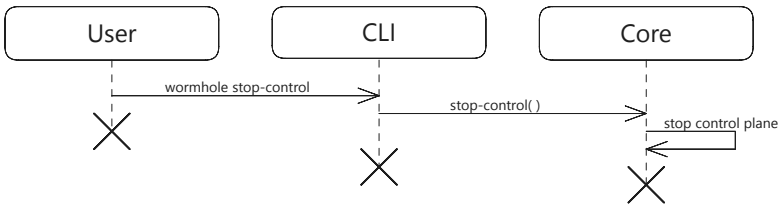
Слика 17: Дијаграм секвенце функционалности покретања *control plane*-а.

Извршавањем ове функционалности, *control plane* постаје активан и спреман за комуникацију са чворовима, чиме се успоставља потпуно функционално окружење за управљање системом.

4.2.11 Заустављање *control plane*-а

Функционалност заустављања *control plane*-а омогућава контролисано искључивање централног дела система задуженог за управљање чворовима. Сврха овог процеса је уредно заустављање свих активних сервиса и ослобађање ресурса *host* машине, без нарушавања интегритета података или конфигурације окружења.

Корисник покреће команду `wormhole stop-control`, којом се иницира поступак гашења *control plane*-а. CLI слој прихвата захтев и прослеђује га *Core* модулу, који директно извршава скрипту за заустављање сервиса без посредовања *VagrantBackend*-а. Скрипта, која је део `c12s/tools` репозиторијума, зауставља све активне контејнере и сервисе покренуте у `Docker` окружењу (слика 18).



Слика 18: Дијаграм секвенце функционалности заустављања *control plane*-а.

Извршавањем ове функционалности, сви сервиси у оквиру *control plane*-а се уредно искључују, чиме се систем доводи у стабилно стање мировања и припрема за могуће поновно покретање или потпуно гашење окружења.

Глава 5

Имплементација система

Имплементација представља фазу у којој је дефинисана архитектура система добила своју конкретну софтверску форму. Циљ је био развити CLI алат који омогућава једноставно подизање, конфигурисање и управљање виртуелним машинама у локалном окружењу. Развој је заснован на модуларном приступу који раздваја функционалности на више логичких целина, CLI, *Core* и *Backend* слој, чиме се постиже лакше одржавање и проширивост решења.

Имплементација је реализована у програмском језику Go, који омогућава компајлирање у једну извршну датотеку. *Backend* је заснован на *Vagrant*-у и *VirtualBox*-у, док се параметри окружења дефинишу путем YAML конфигурационих датотека.

У наставку поглављу биће описана имплементација система, укључујући организацију пројекта и садржај директоријума (поглавље 5.1), *Core* компоненту и интерфејсе (поглавље 5.2), учитавање и валидацију конфигурационих фајлова (поглавље 5.3), CLI компоненту (поглавље 5.4) и *VagrantBackend* компоненту (поглавље 5.5). Посебно су обрађене *shell* скрипте за иницијализацију окружења, покретање и заустављање чворова и управљање *control plane*-ом (поглавље 5.6).

5.1 Организација пројекта

Пројекат је организован по Go конвенцијама, са јасно раздвојеним пакетима по функционалним целинама. Таква структура омогућава лакшу навигацију кроз код, изолацију компоненти и једноставно проширивање система новим функционалностима.

У табели 3 приказана је структура пројекта и намена сваког директоријума. Табела пружа јасан преглед организације изворног кода и служи као оријентир за даље разумевање имплементације система.

Табела 3: Структура пројекта.

Директоријум	Опис
`/aliases`	Садржи дефиниције алијаса за CLI команде.
`/cmd`	Главна улазна тачка CLI апликације.
`/constants`	Садржи глобалне константе које укључују описе команди и подразумеване вредности.
`/internal/core`	Језгро система, овде се налази централни интерфејс *Backend* и основне структуре.
`/internal/backends`	Имплементације конкретних *backend*-а. Тренутно постоји само *VagrantBackend*.
`/config`	Садржи опис и учитавање конфигурације.
`/utils`	Помоћне функције.
`/vagrant`	Радни директоријум у ком се налазе све скрипте које се преносе на виртуелне машине и генерисани *Vagrantfile* који се користи при подизању окружења.

5.2 *Core* компонента

Core компонента представља апстракциони ниво система. Налази се у `/internal/core` директоријуму. Дефинише заједнички интерфејс који сваки *Backend* мора да имплементира. Помоћу интерфејса се

обезбеђује јединствен начин управљања инфраструктуром без обзира на конкретну технологију која стоји иза ње.

На листингу 1 приказан је `Backend` интерфејс који садржи све функционалности управљања животним циклусом виртуелних машина и покретање и заустављање сервиса на њима, као што је описано у претходном поглављу у табели 1.

```
type Backend interface {
  Setup(conf *config.Config) error
  Create(vms ...string) error
  Stop(vms ...string) error
  Resume(vms ...string) error
  Reload(vms ...string) error
  ShutDown(vms ...string) error
  Destroy(vms ...string) error
  StartNodes(vms ...string) error
  StopNodes(vms ...string) error
}
```

Листинг 1: Дефиниција `Backend` интерфејса.

У *Core* компоненти такође је дефинисана метода фабрике (енгл. *factory method*) *Backend*-а, помоћу које се добија исинтанца одговарајуће имплементације *Backend*-а на основу параметара конфигурационе датотеке. На листингу 2 приказана је та метода.

```
func NewBackend(name string) (Backend, error) {
  switch name {
  case "vagrant":
    return backends.NewVagrantBackend()
  default:
    return nil, fmt.Errorf("unsupported backend: %s", name)
  }
}
```

Листинг 2: Метода фабрике *Backend*-а.

5.3 Конфигурација

Конфигурациона датотека у YAML формату дефинише параметре окружења, као што су број виртуелних машина, дистрибуција оперативног система, ресурси и мрежне поставке. На овај начин се омогућава лако прилагођавање и репродукција развојног окружења. Опис структуре конфигурационе датотеке дат је у табели 2 у претходном поглављу.

Конфигурациона структура, `Config`, њено учитавање и валидација се налазе у директоријуму `/config`. За потребе учитавања конфигурационе YAML датотеке коришћена је *gopkg.in/yaml.v3* библиотека и њена `Unmarshal()` метода.

Валидација се врши у складу са типом и значењем сваког поља, како би се обезбедила исправност и компатибилност конфигурације са системом. Примењене провере укључују:

- `vmCount`, `cpus`, `memory` - позитивни цели бројеви
- `osDistro`, `osVersion`, `nameBase`, `backendType` - непразни стрингови
- `guestPort`, `hostPort` - позитивни цели бројеви између 1024 и 65535
- `gui` - булова вредност

На листингу 3 приказан је пример исправне конфигурационе YAML датотеке која се парсира и учитава у структуру `Config`. На листингу 4 приказана је дефиниција конфигурационе структуре `Config`.

```yaml
vmCount: 3
osDistro: "ubuntu/jammy64"
osVersion: "20241002.0.0"
cpus: 4
memory: 4096
gui: false
ipBase: "192.168.56."
guestPort: 6739
hostPortBase: 11000
nameBase: "node"
backendType: "vagrant"
```

Листинг 3: Пример исправне конфигурационе YAML датотеке.

```go
type Config struct {
  VMCount      int    `yaml:"vmCount"`
  OSDistro     string `yaml:"osDistro"`
  OSVersion    string `yaml:"osVersion"`
  CPUs         int    `yaml:"cpus"`
  Memory       int    `yaml:"memory"`
  GUI          bool   `yaml:"gui"`
  IPBase       string `yaml:"ipBase"`
  GuestPort    int    `yaml:"guestPort"`
  HostPortBase int    `yaml:"hostPortBase"`
  NameBase     string `yaml:"nameBase"`
  BackendType  string `yaml:"backendType"`
}
```

Листинг 4: Дефиниција конфигурационе структуре `Config`.

5.4 CLI компонента

Командни интерфејс представља улазну тачку система и омогућава кориснику да управља свим функционалностима алата кроз командну линију. Свака команда активира специфичан ток извршавања унутар система, који се преко *Core* слоја прослеђује одговарајућем *Backend*-у. Тако је обезбеђена апстракција између корисничког интерфејса и логике управљања виртуелним машинама и *control plane*-ом.

Организација кода CLI компоненте прати модуларан приступ. Свака команда је дефинисана у свом *Go* фајлу у директоријуму `/cmd`. Алијаси, кратки и дуги описи команди дефинисани су у посебним датотекама унутар `/aliases` и `/constants` директоријума респективно. То омогућава једноставно додавање или модификацију команди без измене основне логике.

За управљање стаблом команди и аргументима командне линије коришћена је библиотека *Cobra*, која омогућава лаку дефиницију подкоманди и аутоматско генерисање помоћи (енгл. *help*).

На листингу 4 приказана је `/aliases/aliases.go` датотека, која дефинише алијасе свих команди. Овим се омогућава да *Cobra* библиотека препозна и аутоматски понуди исправне облике команде у случају погрешног корисничког уноса.

```
package aliases

const (
  CreateAlias     = "create"
  CreteAlias      = "crete"
  CrateAlias      = "crate"
  // ...
)

var (
  CreateAliases      = []string{CreateAlias, CreteAlias, CrateAlias}
  // ...
)
```

Листинг 5: Приказ садржаја `/aliases/aliases.go` датотеке.

На листингу 6 приказана је `/constants/short.description.go` датотека, која дефинише кратке описе свих команди. Овим се омогућава да *Cobra* библиотека, приликом приказа помоћи или листању доступних команди, уз сваку прикаже сажет опис намене.

```
package constants

const (
  ShortCreateDesc     = "Create one or more virtual machines"
  // ...
)
```

Листинг 6: Приказ садржаја `/constants/short.description.go` датотеке.

На листингу 7 приказана је `/constants/long.description.go` датотека, која дефинише детаљне описе свих команди. Ови описи се приказују када корисник затражи проширену помоћ за одређену команду, пружајући додатне информације о њеној употреби, параметрима и очекиваном понашању.

```
package constants

const (
  LongCreateDesc = `Creates the virtual machines specified by name.
If no names are provided, all virtual machines in the environment will be created.

Example:
- wormhole create
- wormhole create node0 node1`
  // ...
)
```

Листинг 7: Приказ садржаја `/constants/long.description.go` датотеке.

На листингу 8 приказан је пример имплементације *Cobra* команде. Овај пример илуструје употребу претходно дефинисаних константи за алијасе, кратке и дуге описе команди.

```
var CreateCmd = &cobra.Command{
  Use:     "create [VM names...]",
  Aliases: aliases.CreateAliases,
  Short:   constants.ShortCreateDesc,
  Long:    constants.LongCreateDesc,
  Run: func(cmd *cobra.Command, args []string) {
    // ...
  },
}
```

Листинг 8: Пример имплементације *Cobra* команде.

5.5 *VagrantBackend* компонента

Backend слој представља део система задужен за директну комуникацију са платформом за виртуелизацију и извршавање операција над виртуелним машинама. У оквиру овог решења реализован је *VagrantBackend*, који се ослања на алат *Vagrant* и библиотеку *go-vagrant* ради аутоматизације креирања, конфигурисања и управљања виртуелним окружењима. Ова компонента спаја функционалности дефинисане у *Core* компоненти са конкретним механизмима које обезбеђује *Vagrant*.

5.5.1 Генерисање *Vagrantfile*-а

Генерисање *Vagrantfile*-а представља централну функционалност овог модула. На њему се заснива дефинисање окружења које ће се касније покретати помоћу *Vagrant*-а.

VagrantBackend користи предефинисан *Vagrantfile* шаблон, смештен у `/internal/backends/vagrantfile_template.rb` датотеку. На основу шаблона и конфигурационе YAML датотеке генерише излазну *Vagrantfile* датотеку која се користи за покретање окружења.

Користи *text/template* библиотеку како би уградио дефинисане вредности у шаблон. Замењује делове датотеке који се налазе у двоструким витичастим заградама (`{{...}}`) са одговарајућим атрибутом структуре `TemplateData` која садржи опис окружења прилагођен *Vagrantfile* шаблону. На листингу 9 приказан је садржај *Vagrantfile* шаблона са претходно описаним деловима за замену, обухваћеним двоструким витичастим заградама.

```
Vagrant.configure("2") do |config|
    config.vm.box = "{{ .Box }}"
    config.vm.box_version = "{{ .BoxVersion }}"

    # Resource configuration
    config.vm.provider "virtualbox" do |vb|
        vb.memory = {{ .Memory }}
        vb.cpus = {{ .CPUs }}
        vb.gui = {{ .GUI }}
    end

    # Common provisioning script for all VMs
    config.vm.provision "shell", name: "install-dependencies", path:
"install_dependencies.sh"

    # Create node VMs
    {{- range .VMs}}
    config.vm.define "{{ .Name }}" do |node|
        node.vm.hostname = "{{ .Hostname }}"
        node.vm.network "private_network", ip: "{{ .IP }}"
    end
    {{- end }}
end
```

Листинг 9: *Vagrantfile* шаблон.

Како се шаблон не би морао учитавати са диска сваки пут и како би се омогућило да једном компајлирана извршна датотека не зависи од њега, *Vagrantfile* шаблон се уграђује у њу помоћу *embed* библиотеке. На листингу 10 приказан је део кода у склопу *VagrantBackend*-а задужен за то.

```
//go:embed vagrantfile_template.rb
var vagrantfileTemplate string
```

Листинг 10: Уграђивање *Vagrantfile* шаблона у извршну датотеку.

5.5.2 Имплементација функционалности

VagrantBackend имплементара сваку функцију дефинисану у *Backend* интерфејсу. Функционалности су имплементиране ослањајући се на *go-vagrant* библиотеку или на директно позивање *Vagrant* команди. Функционалности су подељене у два скупа:

- за управљање животним циклусом виртуелних машина
- за покретање и засустављање сервиса на њима.

Функционалности за управљање животним циклусом виртуелних машина имплементиране су тако да се ослањају на `VagrantClient` атрибут структуре `VagrantBackend` приказане на листингу 11.

```
type VagrantBackend struct {
  client *vagrant.VagrantClient
}
```

Листинг 11: Дефиниција `VagrantBackend` структуре.

Функционалности за покретање и заустављање сервиса су имплементиране као омотачи око удаљених позива *shell* скрипти на виртуелним машинама помоћу `vagrant ssh` команде, уместо ослањања на *VagrantClient* као у претходном случају.

На листингу 12 приказана је поједностављена имплементација функције `StartNodes`. *Flag*-ови `-t` и `-c` служе за алокацију псеудо TTY-а (енгл. *teletypewriter*) и извршавање наведене команде без отварања интерактивне SSH сесије респективно, односно омогућавају да се *ouput* покренутих скрипти преусмери на терминал, али да терминал не остане заробљен у SSH сесији.

```
func (v *VagrantBackend) StartNodes(vms ...string) error {
  // ...
    cmd := exec.Command("vagrant", "ssh", "-t", vm, "-c", "bash /vagrant/node_start.sh")
  // ...
}
```

Листинг 12: Имплементација функције `StartNodes`.

5.6 *Shell* скрипте

У овом поглављу описане су основне *shell* скрипте које омогућавају подизање и конфигурацију развојног окружења. Скрипте покривају инсталацију зависности, покретање и заустављање сервиса на виртуелним машинама, као и подизање *control plane*-а на *host* машини. Скрипте које се покрећу на виртуелним машинама налазе се у директоријуму `/vagrant` који се монтира на истоимени директоријум на виртуелним машинама.

- `install_dependencies.sh` - скрипта која инсталира све неопходне зависности и сервисе на виртуелној машини, користи се током иницијализације сваке виртуелне машине у склопу *Vagrantfile* провизионисања (листинг 9).
- `node_start.sh` и `node_stop.sh` - скрипте за покретање и заустављање сервиса на виртуелним машинама, конкретно *c12s/star* сервиса, извршавају се преко *Vagrant* SSH сесије.
- `node.env` - конфигурациони фајл који дефинише променљиве окружења специфичне за сваку виртуелну машину, користи се приликом извршавања `install_dependencies.sh` скрипте.
- `control_plane_start.sh` - скрипта која покреће *control plane* сервисе на *host* машини и омогућава да CLI алат управља свим чворовима у систему.

Ово поглавље ће детаљније приказати садржај и ток извршавања сваке скрипте.

5.6.1 Инсталација зависности

Инсталација зависности се одвија приликом првог покретања виртуелне машине у склопу провизионисања. Реализована је у склопу `install_dependencies.sh` скрипте.

Процес инсталације обухвата следеће кораке:
1. додавање званичног *Docker* GPG кључа
2. инсталација *Docker* и *Docker Compose* алата
3. покретање *Docker*-а
4. додавање предефинисаног *Vagrant* корисника у Docker групу
5. инсталација последње верзије *Go*-а и *Git*-а
6. клонирање *c12s/tools* репозиторијума на путању `/home/vagrant/tools` уколико већ не постоји
7. покретање инсталационе скрипте `install.sh` из *c12s/tools*-а како би се инсталирали сви неопходни *Constellations* сервиси
8. конфигурисање `node.env` датотеке на основу дефинисаних параметара виртуелне машине
9. додавање дозволе за покретање `node_start.sh` скрипти

На листингу 13 приказане је део промењљивих окружења дефинисаних у датотеци `node.env`. Оне се аутоматски генеришу за сваки чвор током процеса иницијализације и садрже информације неопходне за правилно повезивање чворова са *control plane*-ом и међусобну комуникацију у оквиру кластера.

```
STAR_HOSTNAME # назив или адреса чвора
NATS_HOSTNAME # назив или адреса чвора на ком је покренут control plane (host)
NATS_ADDRESS # назив или адрса чвора на ком је покренут control plane + порт 4222
BIND_ADDRESS # адреса на којој је чвор доступан осталим чворовима
```

Листинг 13: Промењљиве `node.env` датотеке.

5.6.2 Покретање сервиса на виртуелним машинама

Скрипта `node_start.sh` задужена је за покретање свих потребних *Constellations* сервиса на појединачном чвору. Извршава се након што је успешно завршено провизионисање и конфигурисање окружења.

Процес покретања обухвата следеће кораке:
1. извоз (енгл. *export*) вредности из `node.env` датотеке у окружење виртуелне машине
2. брисање и поновно креирање директоријума на путањи `/etc/c12s` са постављеним свим пермисијама за све кориснике (0777)
3. покретање сервиса дефинисаних у `node.yml` датотеци помоћу *Docker Compose*-а
4. компајлирање *c12s/star* сервиса и креирање `star` извршне датотеке на `/home/vagrant/star` путањи
5. покретање `star` извршне датотеке у позадини и записивање PID-а у `/etc/c12s/star.pid` датотеку

На листингу 14 приказани су сервиси дефинисани у оквиру `node.yml` датотеке који се покрећу на сваком чвору помоћу *Docker Compose*-а.

```yaml
version: "3.8"

services:
  node_exporter:
    # ...
  cadvisor:
    # ...
  starometry:
    # ...
```

Листинг 14: Сервиси дефинисани у `node.yml` датотеци.

5.6.3 Заустављање сервиса на виртуелним машинама

Скрипта `node_stop.sh` служи за контролисано заустављање свих *Constellations* сервиса који су покренути на појединачном чвору. Извршава се ручно или као део поступка гашења кластера. Њена основна улога је да обезбеди чисто и безбедно гашење процеса како би се избегли проблеми при поновном покретању система.

Процес заустављања обухвата следеће кораке:
1. извоз (енгл. *export*) вредности из `node.env` датотеке у окружење виртуелне машине
2. читање PID-а из `/etc/c12s/star.pid` датотеке
3. прора да ли прочитани PID припада `star` процесу
4. убијање `star` процеса
5. гашење сервиса дефинисаних у `node.yml` датотеци помоћу *Docker Compose*-а

5.6.4 Покретање *control plane*-а

Скрипта `control_plane_start.sh` задужена је за иницијализацију и покретање централних сервиса који чине *control plane* систем. Ова компонента представља управљачки слој који координише рад свих чворова у оквиру кластера, обезбеђује размену порука и синхронизацију путем NATS сервиса.

Процес покретања обухвата неколико корака:
1. извоз (енгл. *export*) вредности из `control_plane.env` датотеке у окружење *host* машине
2. покретање свих неопходних сервиса дефинисаних у `control_plane.yml` датотеци помоћу *Docker Compose*-а
3. покретање *scylla* сервиса
4. покретање *Grafana* сервиса

5.6.5 Заустављање *control plane*-а

Скрипта `control_plane_stop.sh` задужена је за контролисано гашење свих сервиса који чине *control plane*. Њена улога је да обезбеди правилно ослобађање ресурса и спречи неконзистентност у раду система након заустављања.

Процес заустављања обухвата следеће кораке:
1. извоз (енгл. *export*) вредности из `control_plane.env` датотеке у окружење *host* машине
2. гашење *Grafana* сервиса
3. гашење *scylla* сервиса
4. гашење свих сервиса дефинисаних у `control_plane.yml` датотеци помоћу *Docker Compose*-а

На листингу 15 приказани су сервиси дефинисани у оквиру `control_plane.yml` датотеке који се покрећу на *host* машини помоћу *Docker Compose*-а.

```yaml
version: "3.8"

services:
  magnetar:
    # ...
  oort:
    # ...
  kuiper:
    # ...
  quasar:
    # ...
  agent_queue:
    # ...
  apollo:
    # ...
  vault:
    # ...
  neo4j:
    # ...
  nats:
    # ...
  etcd:
    # ...
  kuiper_etcd:
    # ...
  quasar_etcd:
    # ...
  scylla:
    # ...
  lunar-gateway:
    # ...
  consul:
    # ...
  rate_limiter_service:
    # ...
  meridian:
    # ...
  meridian_neo4j:
    # ...
  pulsar:
    # ...
  pulsar_etcd:
    # ...
  protostar_healthcheck:
    # ...
  protostar_metrics:
    # ...
  prometheus_healthcheck:
```

Листинг 15: Сервиси дефинисани у `control_plane.yml` датотеци.

Глава 6

Закључак

У закључку дајте кратак преглед онога шта урађено, са осврtом на проблеме који су решени, предности и мане решења и правце даљег развоја.

Списак слика

Списак листинга

Списак табела

Списак коришћених скраћеница

Скраћеница	Опис
API	Application Programming Interface (апликациони програмски интерфејс)
AWS	Amazon Web Services (Амазон веб сервиси)
CI/CD	Continuous Integration / Continuous Delivery (континуирана интеграција / континуирана испорука)
CORS	Cross-Origin Resource Sharing (размена ресурса између извора и дестинације различитог порекла)
CSS	Cascading Style Sheets (језик за описивање стилова)
DOM	Document Object Model (објектни модел документа)
DTO	Data Transfer Object (објекат за пренос података)
HTTP	HyperText Transfer Protocol (протокол за пренос хипертекста)
JSON	JavaScript Object Notation (формат за размену података)
JWT	JSON Web Token (сигурносни токен заснован на JSON формату)
RLS	Row-Level Security (сигурност на нивоу реда)
REST	Representational State Transfer (скуп правила за комуникацију између клијента и сервера)
RPC	Remote Procedure Call (позив удаљене процедуре)
SQL	Structured Query Language (структурирани упитни језик)
TLS	Transport Layer Security (безбедност транспортног слоја)
UML	Unified Modeling Language (језик за моделовање дијаграма)
URL	Uniform Resource Locator (јединствени идентификатор и локатор ресурса)
UI	User Interface (кориснички интерфејс)
UUID	Universally Unique Identifier (универзално јединствени идентификатор)
WAL	Write-Ahead Logging (записивање операција унапред)

Списак коришћених појмова

Појам	Објашњење
Асихнрони рад	Рад који се изводи независно од главног тока извршавања, омогућавајући наставак других операција без чекања на његов завршетак.
Bucket (S3)	Логичка јединица за складиштење у AWS S3 сервису, која организује фајлове у облаку.
Read-Only	Режим рада у коме су подаци само за читање, без могућности измене.
Cross-platform	Способност софтвера да се извршава на више различитих оперативних система из истог кода.
Connection pool	Механизам за управљање и поновну употребу веза са базом података како би се побољшале перформансе апликације.

Биографија

Раде Пејановић рођен је 2002. године у Новом Саду. Првих шест разреда основне школе завршио је у ОШ „Јован Дучић“ у Петроварадину, а преостала два у ОШ при Гимназији „Јован Јовановић Змај“ у Новом Саду. Паралелно је похађао и нижу музичку школу „Јосип Славенски“ у Новом Саду, одсек кларинет.

Средњошколско образовање наставио је у Гимназији „Јован Јовановић Змај“ на смеру за ученике са посебним способностима за физику. За изузетан успех у основној и средњој школи добио је Вукову диплому.

Факултет техничких наука у Новом Саду, смер Софтверско инжењерство и информационе технологије, уписао је 2021. године и успешно завршио све испите предвиђене студијским програмом.

Овим радом завршава основне академске студије на Факултету техничких наука у Новом Саду.

Литература

[1] W. Lamersdorf, „Paradigms of Distributed Software Systems: Services, Processes and Self-organization“, у *E-Business and Telecommunications — International Conference (ICETE 2011), Communications in Computer and Information Science, vol. 314*, M. S. Obaidat, J. L. Sevillano, и J. Filipe, Ур., Springer, Berlin, Heidelberg, 2012, стр. 33–40. doi: 10.1007/978-3-642-35755-8_3.

[2] J. Thönes, „Microservices“, *IEEE Software*, том 32, изд. 1, стр. 116, 2015, doi: 10.1109/MS.2015.11.

[3] S. Nanda и T.-c. Chiueh, „A Survey on Virtualization Technologies“, *Technical Report TR179, Department of Computer Science, Stony Brook University*, 2005, [На Интернету]. Доступно на http://comet.lehman.cuny.edu/cocchi/CMP464/papers/VirtualizationSurveyTR179.pdf

[4] P. Sharma, L. Chaufournier, P. Shenoy, и Y. C. Tay, „Containers and Virtual Machines at Scale: A Comparative Study“, у *Proceedings of the 17th International Middleware Conference*, у Middleware '16. Trento, Italy: Association for Computing Machinery, 2016. doi: 10.1145/2988336.2988337.

[5] S. Chinamanagonda, „Automating Infrastructure with Infrastructure as Code (IaC)“, *SSRN Electronic Journal*, 2019, doi: 10.21275/SR24829170834.

[6] K. Morris, *Infrastructure as Code*. O'Reilly Media, 2021. [На Интернету]. Доступно на https://books.google.rs/books?id=UW4NEAAAQBAJ

[7] c12s, „c12s — GitHub Organization“. 2025.

[8] R. Hat, „What is Virtualization?“. Приступљено: 05. Новембар 2025. [На Интернету]. Доступно на https://www.redhat.com/en/topics/virtualization/what-is-virtualization

[9] AWS, „What is Virtualization?“. Приступљено: 05. Новембар 2025. [На Интернету]. Доступно на https://aws.amazon.com/what-is/virtualization/

[10] J. Shamir, „The Benefits of Virtualization“. [На Интернету]. Доступно на https://www.ibm.com/think/insights/virtualization-benefits

[11] Docker, „What is a Container?“. Приступљено: 05. Новембар 2025. [На Интернету]. Доступно на https://www.docker.com/resources/what-container/

[12] I. Buchanan, „Containers vs Virtual Machines“. Приступљено: 05. Новембар 2025. [На Интернету]. Доступно на https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms

[13] H. Almohammadi, J. L. Hernández, J. M. Gavaldà, и F. G. à, „The Goodness of Nesting Containers in Virtual Machines for Server Consolidation“, *Journal of Grid Computing*, том 22, 2024, doi: 10.1007/s10723-024-09782-2.

[14] C. Feng, „Imperative vs Declarative IaC: Ansible and Terraform Insights“. Приступљено: 06. Новембар 2025. [На Интернету]. Доступно на https://www.casperfeng.com/blog/imperative-vs-declarative-iac-ansible-and-terraform-insights

[15] J. Holdsworth и A. Badman, „What Is Infrastructure as Code (IaC)?“. [На Интернету]. Доступно на https://www.ibm.com/think/topics/infrastructure-as-code

[16] P. A. Abdalla и A. Varol, „Advantages to Disadvantages of Cloud Computing for Small-Sized Business“, у *2019 7th International Symposium on Digital Forensics and Security (ISDFS)*, 2019, стр. 1–6. doi: 10.1109/ISDFS.2019.8757549.

[17] K. Cao, Y. Liu, G. Meng, и Q. Sun, „An Overview on Edge Computing Research“, *IEEE Access*, том 8, стр. 85714–85728, 2020, doi: 10.1109/ACCESS.2020.2991734.

[18] M. Simić, G. Sladić, M. Zarić, и B. Markoski, „Infrastructure as Software in Micro Clouds at the Edge“, *Sensors*, том 21, изд. 21, стр. 7001, 2021, doi: 10.3390/s21217001.

[19] spf13, „Cobra: A Commander for modern Go CLI interactions“. 2025.

[20] HashiCorp, „Vagrant – Development Environments Made Easy“. [На Интернету]. Доступно на https://developer.hashicorp.com/vagrant

[21] B. Matcuk, „go-vagrant: A golang wrapper around the Vagrant command-line utility“. 2021.

[22] O. Corporation, „Oracle VM VirtualBox — Powerful open source virtualization“. [На Интернету]. Доступно на https://www.virtualbox.org/

[23] I. Docker, „Docker — Accelerate how you build, share, and run applications“. [На Интернету]. Доступно на https://www.docker.com/

[24] I. Docker, „Docker Compose Documentation“. [На Интернету]. Доступно на https://docs.docker.com/compose/

[25] HashiCorp, „Terraform — Automate Infrastructure on Any Cloud“. [На Интернету]. Доступно на https://developer.hashicorp.com/terraform

[26] A. Community, „Ansible Documentation“. [На Интернету]. Доступно на https://docs.ansible.com/

[27] G. Community, „Git — Distributed version control system (official site)“. Приступљено: 06. Новембар 2025. [На Интернету]. Доступно на https://git-scm.com/

[28] c12s, „tools: Scripts and utilities used by the c12s organization“. 2025.

[29] c12s, „star: A Go module by c12s (GitHub repository)“. 2024.

[30] c12s, „cockpit: A CLI tool that communicates with the Constellations gateway“. 2024.