VIŠJA STROKOVNA ŠOLA ACADEMIA

MARIBOR

Primerjava sistemov Docker Swarm in Kubernetes na primeru vzpostavitve poštnega strežnika

Kandidat: Tilen Barbirič

Vrsta študija: študent izrednega študija

Študijski program: Informatika

Mentor predavatelj: mag. Dušan Brglez

Mentor v podjetju: Maksimilijan Bračič

Lektor: »Tea Erjavec, prof. geog. in slov.«

Maribor, 2024

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Podpisani/a Tilen Barbirič, sem avtor diplomskega dela z naslovom Primerjava sistemov *Docker Swarm* in *Kubernetes* na primeru vzpostavitve poštnega strežnika, ki sem ga napisal pod mentorstvom mag. Dušana Brgleza.

S svojim podpisom zagotavljam, da:

- je predloženo delo izključno rezultat mojega dela,
- sem poskrbel/a, da so dela in mnenja drugih avtorjev, ki jih uporabljam v predloženi nalogi, navedena oz. citirana skladno s pravili Višje strokovne šole Academia Maribor,
- se zavedam, da je plagiatorstvo – predstavljanje tujih del oz. misli, kot moje lastne kaznivo po Zakonu o avtorski in sorodnih pravicah (Uradni list RS, št. 16/07 – uradno prečiščeno besedilo, 68/08, 110/13, 56/15 in 63/16 – ZKUASP); prekršek pa podleže tudi ukrepom Višje strokovne šole Academia Maribor skladno z njenimi pravili,
- skladno z 32.a členom ZASP dovoljujem Višji strokovni šoli Academia Maribor objavo diplomskega dela na spletnem portalu šole.

Maribor, september 2024 Podpis študenta:

ZAHVALA

Zahvaljujem se mentorjema mag. Dušanu Brglezu in Maksimilijanu Bračiču za vso pomoč in vodenje pri ustvarjanju diplomskega dela.

Prav tako bi se rad zahvalil svoji družini in vsem bližnjim, ki so me med časom študija in pisanja diplome podpirali.

En velik hvala pa tudi vsem anonimnim ljudem na internetu, ki so mi velikokrat odgovorili na vprašanja, ko sem naletel na kakšno specifično težavo pri vzpostavitvi kakšnega servisa.

POVZETEK

Že dlje časa se ukvarjam s samogostovanjem raznih storitev v kontejnerjih. To diplomsko delo se mi je zdelo kot odlična priložnost, da svoje znanje poglobim in da prvič vzpostavim visoko razpoložljiv porazdeljen sistem. Storitve, ki tečejo na visoko razpoložljivih sistemih, uporabljamo vsak dan, pa se tega verjetno niti ne zavedamo, dokler nekaj naenkrat preneha delovati. Raziskovanje in implementacija takšnega sistema sta mi omogočila, da sem se seznanil z mnogimi izzivi in rešitvami, ki so ključni za zagotavljanje neprekinjenega delovanja storitev. Praktično vsi dandanes uporabljamo e-pošto, večina nas uporablja brezplačne rešitve velikih podjetij, kot so npr. Microsoft Outlook, Google Gmail ali Proton Mail. E-pošta je ena izmed redkih storitev, ki jih nisem še nikoli gostoval, prav tako še nisem uporabljal sistemov za upravljanje gruč, zato sem se odločil, da jih preučim in medsebojno primerjam – želel sem podrobno preučiti prednosti, slabosti, razlike in podobnosti sistemov *Kubernetes* in *Docker Swarm*. Oba sistema za upravljanje z kontejnerji sta zelo popularna. V produkciji sicer večkrat najdemo *Kubernetes* – zanimalo me je, za kaj ne *Docker Swarm*-a, ki je na prvi pogled preprostejši za konfiguracijo.

V teoretičnem delu sem raziskoval in opisoval kontejnerje, sistema *Kubernetes* in *Docker*, ker pa je predmet primerjave poštni strežnik, sem napisal še nekaj o e-pošti in e-poštnih rešitvah, dotaknil pa sem se tudi E-slame, s katero ima izkušnje prav vsak uporabnik e-pošte. Za potrebe hipoteze BTEC sem se poglobil v velike podatke, kjer sem preučil, kaj sploh veliki podatki so in kakšna je povezava med njimi in e-pošto ter kako lahko veliki podatki pozitivno ali negativno vplivajo na našo varnost na internetu ter na zasebnost.

Ker gre za relativno kompleksen projekt, sem v praktičnem delu želel zelo podrobno predstaviti vse posamezne uporabljene komponente. Najprej sem predstavil vse servise, ki sem jih postavil v obeh sistemih, navedel sem tudi razloge za mojo odločitev za uporabo le-teh. Za tem sem podrobno opisal postopke vzpostavitve izdelka, torej od vzpostavitve samih gruč do vzpostavitve posameznih servisov ter dodatnih specifik pri obeh sistemih. Kasneje sem opisal tudi kompleksnost konfiguracije poštnega strežnika s poudarkom na varnostni konfiguraciji. Na koncu sem izdelek v obeh sistemih tudi testiral z lastno testno skripto za testiranje poštnih strežnikov in opravil analizo rezultatov ter napisal še poglobljeno primerjavo glede vzpostavitve v obeh sistemih.

Ključne besede: *Kubernetes, Docker, visoka razpoložljivost, e-pošta, kontejner, Ansible.*

ABSTRACT

Comparison of Docker Swarm and Kubernetes on the example of setting up a mail server.

I have been working with self-hosting various services in containers for quite some time. This thesis seemed like an excellent opportunity to deepen my knowledge and to set up a highly available distributed system for the first time. We use services running on highly available systems every day, though we are likely not aware of it until something suddenly stops working. The research and implementation of such a system allowed me to familiarize myself with many challenges and solutions that are essential for ensuring uninterrupted service operation. Practically all of us use email nowadays, and most of us use free solutions from large companies like Microsoft Outlook, Google Gmail, or Proton Mail. Email is one of the few services I had never hosted myself, and I had also never worked with cluster management systems, so I decided to study and compare them—I wanted to thoroughly examine the advantages, disadvantages, differences, and similarities between the two. Both are very popular container orchestration systems. Kubernetes is more commonly found in production environments, so I was curious why Docker Swarm, which at first glance seems simpler to configure, is not as widely used.

In the theoretical part, I researched and described containers, the Kubernetes and Docker systems, and, since the subject of comparison is a mail server, I also wrote about email and email solutions. I touched on email spam which is something every email user has experience with. For the needs of the BTEC hypothesis, I also delved into big data, where I explored what big data is, what its connection to email is, and how big data can positively or negatively impact our online security and privacy. Since this is a relatively complex project, in the practical part I wanted to present all the individual components used in great detail. First, I introduced all the services I deployed in both systems, and I provided the reasons for my decision to use them. I described the product setup processes in detail, from setting up the clusters themselves to setting up the individual services, and the additional specifics for both systems. Later, I also described the complexity of configuring the mail server, with an emphasis on security configuration. Toward the end, I tested the product in both systems with my test script for mail servers, conducted an analysis of the results, and wrote an in-depth comparison of the setup using both systems.

Keywords: *Kubernetes, Docker, high availability, email, container, Ansible.*

KAZALO VSEBINE

1 UVOD 14

1.1 Opis področja in opredelitev problema 14

1.2 Namen, cilji in osnovne trditve 14

1.3 Predpostavke in omejitve 15

1.4 Uporabljene raziskovalne metode 16

2 Kontejnerizacija 17

2.1 Primerjava z navideznimi stroji 18

2.2 Sistem Kubernetes 19

2.2.1 Arhitektura sistema Kubernetes 20

2.2.1.1 Gradniki 20

2.2.1.1.1 Vladarji (gradniki nadzorne ravnine) 21

2.2.2 Viri sistema Kubernetes 22

2.2.2.1 Namespace 22

2.2.2.2 Workload in pod 22

2.2.2.2.1 Upravljanje delovnih obremenitev 22

2.2.2.3 Servisi 24

2.2.2.4 Trajni nosilci 24

2.2.3 Orodja 26

2.2.3.1 Kubectl 26

2.2.3.2 Helm 27

2.3 Docker 28

2.3.1 Docker Compose 28

2.3.2 Docker swarm 30

2.3.2.1 Arhitektura 31

3 e-pošta in poštni strežniki 33

3.1 Zgodovina e-pošte in poštnih strežnikov 33

3.1.1 Prvi programi in protokoli 33

3.1.2 Protokoli za prejemanje 34

3.1.2.1 Protokol POP 35

3.1.2.2 Protokol IMAP 36

3.1.3 Poštni odjemalci ... 37
3.2 E-poštne rešitve ... 38
3.3 Primerjava različnih potencialnih e-poštnih rešitev za izdelek ... 40
3.4 E-slama ... 42
3.5 Veliki podatki ... 43
3.5.1 Kaj so veliki podatki? ... 43
3.5.2 Prednosti in slabosti velikih podatkov v kontekstu e-pošte ... 44
4 Raziskovalni del / izdelek ... 45
4.1 Predstavitev izdelka ... 45
4.2 Uporabljeni virtualni stroji ... 47
4.3 Postopek vzpostavitve Kubernetes »gruče« ... 48
4.4 Postopek vzpostavitve Docker Swarm »gruče« ... 48
4.5 Opis in razlaga uporabljenih servisov ... 50
4.5.1 Podatkovna baza YugabyteDB ... 50
4.5.1.1 Primerjava z PostgreSQL ... 50
4.5.1.2 Namen uporabe v našem izdelku in kritična ocena ... 51
4.5.2 MinIO S3 objektna shramba podatkov ... 52
4.5.3 Grafana ... 56
4.5.3.1 Primer ustvarjanja tortnega diagrama ... 57
4.5.3.2 Namen in uporaba v našem izdelku ter kritična ocena ... 59
4.5.4 Zbiranje dnevniških datotek in metrik ... 59
4.5.4.1 Loki in promtail ... 59
4.5.4.2 Prometheus in node_exporter ... 60
4.5.4.3 Namen uporabe v izdelku in kritična ocena ... 60
4.5.5 HAProxy ... 61
4.5.5.1 Uporaba v našem izdelku in kritična ocena ... 62
4.5.6 VPN povezava med strežniki ... 62
4.5.7 Ansible ... 62
4.5.7.1 Sklopi orodja Ansible in primeri konfiguracijskih datotek ... 63
4.5.7.1.1 Knjige iger ... 63
4.5.7.1.2 Vloge in naloge ... 64
4.5.7.1.3 Datoteke z spremenljivkami ... 66

4.5.7.1.4 Datoteke z podatki o gostiteljih ... 67
4.5.7.1.5 Šifriranje z ansible vault ... 67
4.5.8 Konfiguracija uporabnikov v LDAP sistemu FreeIPA ... 68
4.5.8.1 Primer dodajanja uporabnika testDD ... 68
4.6 Postopek vzpostavitve in konfiguracija ... 70
4.6.1 Z Docker Swarm ... 70
4.6.1.1 Omrežje mail-network ... 71
4.6.1.2 Servisi ... 71
4.6.1.2.1 X-Common ... 71
4.6.1.2.2 Podatkovna baza yugabyteDB ... 72
4.6.1.2.2.1 Postopek vzpostavitve ... 74
4.6.1.2.3 Sistem za shranjevanje podatkov minIO ... 75
4.6.1.2.3.1 Postopek vzpostavitve ... 77
4.6.1.2.4 Poštni strežnik Stalwart ... 78
4.6.1.2.4.1 Postopek vzpostavitve ... 80
4.6.1.2.5 Grafana in zbiralniki ... 80
4.6.1.2.5.1 Grafana ... 81
4.6.1.2.5.2 Loki ... 82
4.6.1.2.5.3 Node Exporter in Prometheus ... 84
4.6.2 Z Kubernetes ... 86
4.6.2.1 Longhorn ... 86
4.6.2.2 Imenski prostori ... 88
4.6.2.3 Servisi ... 89
4.6.2.3.1 Podatkovna baza yugabyteDB ... 90
4.6.2.3.2 Minio S3 sharmba objektov ... 92
4.6.2.3.3 Poštni strežnik Stalwart ... 94
4.6.2.3.3.1 Struktura HHelm diagrama ... 94
4.6.2.3.4 Grafana, loki in Prometheus ... 98
4.6.3 Konfiguracija poštnega strežnika ... 99
4.6.3.1 Konfiguracijska datoteka ... 99
4.6.3.2 Ustvarjanje lokalnih domen ... 101
4.6.3.3 Zagotavljanje varnosti poštnega strežnika ... 102
4.6.3.3.1 DMARC, DKIM in SPF ... 102
4.6.3.3.2 Varnost povezave ... 104
4.6.3.3.3 Dušenje in stopenjsko omejevanje ... 106
4.6.3.3.4 Spam filter in SMTP omejitve ... 107

4.7 Testiranje 109
4.7.1 Vizualizacija delovanja (Grafana) 109
4.7.2 Prilagoditve konfiguracije poštnega strežnika pred testiranjem 110
4.7.3 Skripta za izvajanje testov 111
4.7.4 Test 1: Obremenitev sistema ter učinkovitost porazdeljevanja 113
4.7.5 Test 2: Obremenitev sistema ter učinkovitost porazdeljevanja pri izpadu enega vozlišča 118
4.7.6 Primerjava rezultatov obeh testov in kritična ocena 122
4.8 Primerjava obeh sistemov in načinov vzpostavitve 123
5 SKLEP 126
6 kazalo virov in literature 128

KAZALO SLIK

SLIKA 1 PRIMERJAVA KONTEJNERIZIRANIH APLIKACIJ Z APLIKACIJAMI NA NAVIDEZNIH STROJIH 18

SLIKA 2 VZORČEN PRIMER ARHITEKTURE GRUČE *KUBERNETES* 20

SLIKA 3 PRIMER DELOVNE OBREMENITVE TIPA JOB 23

SLIKA 4 PRIMER DEFINICIJE SERVISA 24

SLIKA 5 PRIMERI UPORABE ORODJA KUBECTL 26

SLIKA 6 PRIMERJAVA VZPOSTAVITVE APLIKACIJE V VEČ OKOLJIH Z IN BREZ ORODJA HELM 27

SLIKA 7 PRIMER DATOTEKE DOCKER COMPOSE 29

SLIKA 8 ARHITEKUTRA GRUČE SWARM 31

SLIKA 9 DIAGRAM, KI PRIKAZUJE DELOVANJE PROTOKOLOV POP3 IN IMAP 34

SLIKA 10 IZDELEK Z SISTEMOM *KUBERNETES* 45

SLIKA 11 IZDELEK Z SISTEMOM DOCKER SWARM 46

SLIKA 12 UKAZ ZA PRIKAZ VOZLIŠČ GRUČE *KUBERNETES* 48

SLIKA 13 SKRIPTA ZA NAMESTITEV SISTEMA *DOCKER* NA STREŽNIK 49

SLIKA 14 USTVARJANJE GRUČE *DOCKER SWARM* IN DODAJANJE UPRAVLJALCEV. 49

SLIKA 15 ARHITEKTURA GRUČE SISTEMA *MINIO* 53

SLIKA 16 PRIMER DATOTEKE Z POLITIKO DOSTOPA DO VEDRA *STALWART-BUCKET* 54

SLIKA 17 PRIMER GRAFA USTVARJENEGA V PLATFORMI *GRAFANA* 56

SLIKA 18 PRIMER GRADNIKOV POIZVEDBE V GRAFANI 57

SLIKA 19 PRIMER TRANSFORMACIJ V *GRAFANI* (1/2) 57

SLIKA 20 PRIMER TRANSFORMACIJ V *GRAFANI* (2/2) 58

SLIKA 21 IZSEK IZ KONFIGURACIJSKE DATOTEKE HAPROXY 61

SLIKA 22 PRIMER KNJIGE IGER ZA VZPOSTAVITEV POŠTNEGA STREŽNIKA IN OSTALIH ODVISNOSTI V SISTEMU *DOCKER SWARM* 63

SLIKA 23 PRIMER DATOTEKE Z NALOGAMI *CONFIGUREDBREPLICAS* V VLOGI *DEPLOYMENT_MANAGEMENT* 64

SLIKA 24 PRIMER DATOTEKE Z SPREMENLJIVKAMI MONITORING_VARS.JSON 66

SLIKA 25 PRIMER DATOTEKE Z PODATKI O GOSTITELJIH 67

SLIKA 26 UKAZI ZA ŠIFRIRANJE IN DEŠIFRIRANJE DATOTEK Z ANSIBLE-VAULT 67

SLIKA 27 PRIJAVNA STRAN SISTEMA FREEIPA 68

SLIKA 28 SEZNAM UPORABNIKOV V SISTEMU *FREEIPA* 69

SLIKA 29 OKNO ZA DODAJANJE UPORABNIKA V SISTEM FREEIPA 69

SLIKA 30 DODAJANJE DODATNEGA E-POŠTNEGA NASLOVA ZA UPORABNIKA V SISTEMU FREEIPA 70

SLIKA 31 *ANSIBLE* NALOGA ZA USTVARJANJE OMREŽJA *MAIL-NETWORK* 71

SLIKA 32 PRIMER »SKUPNE« KONFIGURACIJE ZA *YB-MASTER* KONTEJNER 71

SLIKA 33 KONFIGURACIJA ZA VLADARJA YB-MASTER1 72

SLIKA 34 KONFIGURACIJA ZA DELAVCA *YB-TSERVER-1* 73

SLIKA 35 DIAGRAM POTEKA VZPOSTAVITVE *YUGABYTEDB* V SISTEMU *DOCKER SWARM* 74
SLIKA 36 SKRIPTA *INIT.SQL* 75
SLIKA 37 SKRIPTA *ANSIBLEINIT.SH* 75
SLIKA 38 SKUPNA KONFIGURACIJA KONTEJNERJEV *MINIO* 75
SLIKA 39 KONFIGURACIJA KONTEJNERJA *MINIO* 76
SLIKA 40 DIAGRAM POTEKA VZPOSTAVITVE SERVISA *MINIO* V SISTEMU *DOCKER SWARM* 77
SLIKA 41 KONFIGURACIJA ZA ENEGA IZMED KONTEJNERJEV POŠTNEGA STREŽNIKA *STALWART* 78
SLIKA 42 SKUPNA KONFIGURACIJA ZA KONTEJNERJE POŠTNEGA STREŽNIKA *STALWART* 79
SLIKA 43 DIAGRAM POTEKA ZA VZPOSTAVITEV POŠTNEGA STREŽNIKA V *DOCKER SWARM* 80
SLIKA 44 KONFIGURACIJSKA DATOTEKA ZA KONTEJNER *GRAFANA* V SISTEMU *DOCKER SWARM* 81
SLIKA 45 KONFIGURACIJSKA DATOTEKA ZA KONTEJNER *LOKI* V SISTEMU *DOCKER SWARM* 82
SLIKA 46 SKRIPTA ZA IZVOZ SKRIVNOSTI V OKOLJSKE SPREMENLJIVKE 82
SLIKA 47 KONFIGURACIJSKA DATOTEKA *CONFIG.YAML* ZA *LOKI* (*DOCKER SWARM*) 83
SLIKA 48 KONFIGURACIJSKA DATOTEKA ZA KONTEJNER *PROMETHEUS* V SISTEMU *DOCKER SWARM* 84
SLIKA 49 KONFIGURACIJSKA DATOTEKA ZA KONTEJNERJE *NODE_EXPORTER* V SISTEMU *DOCKER SWARM* 84
SLIKA 50 UKAZI ZA VZPOSTAVITEV SHRAMBE *LONGHORN* 86
SLIKA 51 UREJANJE KONFIGURACIJE *CONFIGMAP* ZA *LONGHORN* 87
SLIKA 52 PRIMER USTVARJANJA IMENSKEGA PROSTORA V SISTEMU *KUBERNETES* 88
SLIKA 53 DIAGRAM POTEKA, KI PRIKAZUJE VZPOSTAVITEV SERVISOV V SISTEMU *KUBERNETES* 89
SLIKA 54 DIAGRAM POTEKA ZA VZPOSTAVITEV IN KONFIGURACIJO *YUGABYTEDB* V SISTEMU *KUBERNETES* 90
SLIKA 55 DATOTEKA Z VREDNOSTMI ZA *MINIO* V SISTEMU *KUBERNETES* 92
SLIKA 56 STRUKTURA HHELM DIAGRAMA ZA POŠTNI STREŽNIK STALWART 94
SLIKA 57 DATOTEKA *SECRETS.YAML* 95
SLIKA 58 DATOTEKA *STATEFULSET* ZA POŠTNI STREŽNIK *STALWART* 96
SLIKA 59 DATOTEKA Z VREDNOSTMI ZA HHELM DIAGRAM POŠTNEGA STREŽNIKA *STALWART* 97
SLIKA 60 DODAJANJE DOMENE PREKO SPLETNEGA VMESNIKA 101
SLIKA 61 DIAGRAM SPF, DKIM IN DMARC 103
SLIKA 62 DKIM, SPF, DMARC ZAPISI, KI JIH USTVARI *STALWART* 104
SLIKA 63 PRIMER KOMUNIKACIJE Z IMPLICITNIM TLS 105
SLIKA 64 PRIVZETE NASTAVITVE DUŠENJA V KONFIGURACIJSKI DATOTEKI 106
SLIKA 65 PRIMER GLAVE SPOROČILA, KI JE BILO OZNAČENO KOT E-SLAMA 107
SLIKA 66 SMTP AVTENTIKACIJA 107
SLIKA 67 NADZORNA PLOŠČA ZA SISTEM *KUBERNETES* 109
SLIKA 68 SEKCIJA ZA NADZOR POŠTNEGA STREŽNIKA V *GRAFANI* 110
SLIKA 69 IZSEK IZ KONFIGURACIJE Z ONEMOGOČENIMI VARNOSTNIMI NASTAVITVAMI 110
SLIKA 70 TESTNA SKRIPTA SPAM.SH 111
SLIKA 71 IZSEK IZ DATOTEKE Z POŠILJATELJI 112

SLIKA 72 ZASEDENOST PROCESORJA PRI TESTU 1 (*KUBERNETES*) 113
SLIKA 73 UTILIZACIJA PISANJA NA DISK PRI TESTU 1 (*KUBERNETES*) 113
SLIKA 74 ZASEDENOST PROCESORJA PRI TESTU 1 (*DOCKER SWARM*) 115
SLIKA 75 UTILIZACIJA PISANJA NA DISK PRI TESTU 1 (*DOCKER SWARM*) 115
SLIKA 76 ZASEDENOST PROCESORJA PRI TESTU 2 (*KUBERNETES*) 118
SLIKA 77 UTILIZACIJA PISANJA NA DISK PRI TESTU 2 (*KUBERNETES*) 118
SLIKA 78 ZASEDENOST PROCESORJA PRI TESTU 2 (*DOCKER SWARM*) 119
SLIKA 79 UTILIZACIJA PISANJA NA DISK PRI TESTU 2 (*DOCKER SWARM*) 120

KAZALO TABEL

TABELA 1 PRIMERJAVA RAZLIČNIH E-POŠTNIH REŠITEV 41
TABELA 2 UPORABLJENI VIRTUALNI STROJI 47
TABELA 3 E-POŠTNI PROTOKOLI IN TLS 105
TABELA 4 REZULTATI TESTA 1 (*KUBERNETES*) 114
TABELA 5 REZULTATI TESTA 1 (DOCKER SWARM) 116
TABELA 6 REZULTATI TESTA 2 (*KUBERNETES*) 119
TABELA 7 REZULTATI TESTA 2 (DOCKER SWARM) 120

1 UVOD

1.1 Opis področja in opredelitev problema

Živimo v svetu, v katerem vsakodnevno uporabljamo mnogo različnih aplikacij, ki so nam vedno na voljo. Velikokrat niti ne pomislimo o možnosti nedelovanja določene aplikacije, dokler se le-to ne zgodi – primer: izpad Googlovih storitev avgusta 2022 (Gadgets 360 Staff, 2022). Razlog za to je, da večina storitev in aplikacij, ki jih uporabljamo, temelji na konceptu visoke razpoložljivosti, kar pomeni, da v primeru izpada enega strežnika za dano storitev njegovo nalogo prevzamejo drugi, kar znatno poveča razpoložljivost in možnosti za celoten izpad zelo zmanjša.

Pri visoki razpoložljivosti pa zelo pomaga tudi koncept kontejnerjev. Ti so v zadnjih letih postali praktično standardni način za vzpostavljanje aplikacij. Njihova arhitektura omogoča, da servise in aplikacije hitro ponovno vzpostavimo, s čim lahko tudi znatno zmanjšamo čas izpada storitev.

V splošnem vlada prepričanje, da se visoka razpoložljivost »izplača« le v primerih, ko ima veliko število uporabnikov in ko bi izpad našega servisa pomenil večje stroške, kot so stroški vzpostavitve in vzdrževanja takšnega sistema, kar je čisto razumljivo. Sam verjamem, da obstajajo uporabne in smiselne možnosti za visoko razpoložljivost tudi v manjših razsežnostih. Zato ta diplomska naloga temelji ravno na vzpostavitvi visoko razpoložljivega poštnega strežnika. Če sami gostujemo spletno pošto, si res ne želimo, da pride do situacije, ko nam sporočilo ni bilo dostavljeno zaradi nedelovanja. Prav tako pa bi rad preveril težavnost in vzpostavitev takšnih sistemov v domačem okolju in ne v oblaku.

1.2 Namen, cilji in osnovne trditve

Že dolgo časa sem imel željo po vzpostavitvi (HA) sistema z veliko razpoložljivostjo, vendar zaradi mojega neznanja o sistemih za upravljanje gruč in težavnosti njihove uporabe se (še) do zdaj za to nisem odločil.

Diplomsko delo preučuje sistema *Kubernetes* in Docker Swarm, razlike med njima, ter prednosti in slabosti uporabe obeh sistemov. Cilj je postaviti poštni strežnik z vsemi pomožnimi servisi v obeh sistemih in primerjati delovanje, odzivnost, ter težavnost postavitve in uporabe obeh sistemov.

Za pomoč pri pisanju tega diplomskega dela in namene preverjanja mojih mnenj sem si postavil naslednje hipoteze:

- *H1* Vzpostavitev *Docker Swarm* »gruče« bo preprostejša kot vzpostavitev »gruče« *Kubernetes*.
- *H2* Sistem *Kubernetes* omogoča naprednejše upravljanje s kontejnerji kot sistem *Docker Swarm*.
- *H3* Prilagajanje obsega poštnega strežnika bo pri *Kubernetes* »gruči« boljša kot pri »gruči« *Docker Swarm*.
- *H4* Nadzor in sledenje dogodkov poštnega strežnika bo pri *Kubernetes* »gruči« boljše kot pri »gruči« *Docker Swarm*.
- *H5* Sistem *Kubernetes* ima večjo uporabniško skupnost kot sistem *Docker Swarm*.
- *H6* Poraba sistemskih virov bo pri sistemu *Kubernetes* večja kot pri sistemu *Docker Swarm*.
- *H7 Big data will negatively impact the privacy of users in the long run* – dodatna hipoteza – BTEC 2023 »Big Data«.

1.3 Predpostavke in omejitve

Pri pisanju diplomskega dela in ustvarjanju izdelka imamo tako tehnične kot jezikovne omejitve. Pri tehničnih bi izpostavil, da bosta gruči obeh sistemov tekli v virtualnih strojih na dveh fizičnih računalnikih, virtualni sistemi bodo tudi omejeni z virtualnimi omrežji, s katerimi bomo simulirali območno postavitev. Sistem *Kubernetes* se v veliki načini uporablja preko upravljanih sistemov v oblaku (GKE – Google, AKS – Microsoft), to pomeni omejeno število dokumentacije za lokalno vzpostavitev visoko razpoložljivih servisov in gruč.

Z jezikovnimi omejitvami pa želim izpostaviti to, da žal v slovenskem jeziku ne obstaja veliko literature na to temo, tudi dokumentacija za sisteme in servise je v glavnem na voljo le v angleškem jeziku, tako da bo večina uporabljene literature v angleščini.

1.4 Uporabljene raziskovalne metode

V diplomskem delu smo uporabili naslednje raziskovalne metode:

1. Preučevanje literature – ker imam o kontejnerizaciji le osnovno predznanje, ter sistemov *Docker Swarm* in *Kubernetes* ne poznam ravno dobro, sem preučil mnogo različnih člankov, za vsak servis sem se globoko poglobil v dokumentacijo, da bi razumel kako stvari delujejo.
2. Študija primera – v diplomskem delu primerjam in analiziram prednosti, slabosti in učinkovitost obeh sistemov.
3. Primerjalna analiza – v diplomskem delu primerjam različne vidike sistemov *Kubernetes* in *Docker Swarm*; od funkcionalnosti, arhitekture vse do enostavnosti uporabe in prilagajanja obsega.

2 KONTEJNERIZACIJA

Kontejner (angl. *software container*) je programska enota, ki vsebuje vso kodo in njene odvisnosti. Zagotavlja, da aplikacija deluje tekoče v raznih računalniških okoljih. (Docker Inc., n.d.)

Kontejnerji omogočajo, da našo programsko kodo, konfiguracije, in odvisnosti zavijemo v en objekt – sliko. Kontejnerji si delijo operacijski sistem, ki je nameščen na napravi, in tečejo kot procesi z izoliranimi viri, kar omogoča hitre, zanesljive in stalne postavitve, neodvisne od okolja, v katerem tečejo.

Prednosti kontejnerjev so, da lahko tečejo kjerkoli (ker so koda in odvisnosti zapakirane v en objekt), boljši nadzor nad porabo virov (izolacija procesov omogoča, da za vsak kontejner nastavimo točno koliko pomnilnika, procesorske moči in prostora na disku lahko uporabi) in hitrejše prilagajanje obsega (vsak kontejner teče kot ločen proces na operacijskem sistemu na napravi, kar omogoča hiter vklop in izklop kontejnerja). (Hewage, 2020)

Proces delovanja in vzpostavitve lahko opišemo v 4. korakih:

1. Ustvarjanje kontejnerske slike:

   Programsko kodo, konfiguracije in odvisnosti zavijemo v kontejnersko sliko.

2. Samodejno testiranje in postavitev:

   Ta korak sicer za delovanje ni nujen, ampak je ponavadi ključen del življenjskega cikla kontejnerja. Z raznimi avtomatskimi testi in CI/CD vodovodi kodo sproti preverjamo in avtomatiziramo njeno postavitev v želeno okolje.

3. Postavitev in zagon v kateremkoli okolju:

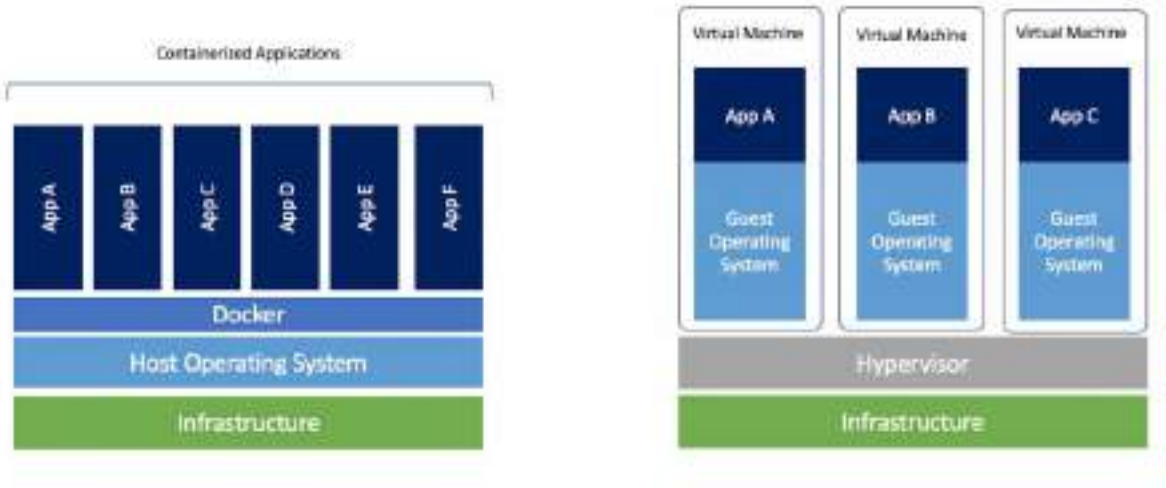
   Kontejner bo deloval enako ne glede na operacijski sistem.

4. Hitro prilagajanje obsega:

   Kontejnerji se zaženejo in ugasnejo zelo hitro, kar omogoča prilagajanje obsega naših servisov na katerokoli velikost.

2.1 *Primerjava z navideznimi stroji*

Navidezni stroj (angl. *virtual machine*) je računski vir, ki namesto fizičnega računalnika za izvajanje programov in aplikacij uporablja programsko opremo. Na enem fizičnem gostitelju lahko teče en ali več gostov. Vsak navidezni stroj uporablja svoj ločen operacijski sistem in funkcije ter teče neodvisno od ostalih »gostov«. To pomeni, da lahko na enem fizičnem stroju (gostitelju), tečejo vzporedno na primer MacOS, Linux ter Windows navidezni stroji. (Broadcom Inc., n.d.)



Slika 1 primerjava kontejneriziranih aplikacij z aplikacijami na navideznih strojih

Vir: (Broadcom Inc., n.d.)

Kontejnerji in navidezni stroji imajo sicer podobne prednosti glede izolacije in določanja virov, vendar se bistveno razlikujejo po tem, da kontejner virtualizira operacijski sistem, navidezni stroj pa strojno opremo. (Hewage, 2020)

Kombinacija navideznih strojev in kontejnerjev nam omogoča, da postavimo več nivojsko medsebojno izoliranih sistemom s podrobnim nadzorom nad določanjem in porabo virov.

2.2 Sistem Kubernetes

Kubernetes je prenosljiva, razširljiva, odprto kodna platforma za upravljanje kontejneriziranih delovnih obremenitev (angl. *workload*), ki olajša tako deklarativno konfiguracijo kot avtomatizacijo. Zaradi velike razširjenosti sistema so programi, podpora in orodja lahko dostopni. (The Kubernetes Authors, n.d.)

Inspiracija za *Kubernetes* je Googlov interni sistem za upravljanje gruč, Borg. (Google Inc., n.d.)

Prednosti *Kubernetes* sistema, kot so navedene v dokumentaciji Googlovega oblaka (Google Inc., n.d.):

1. *Avtomatizacija*

   Kubernetes ima vgrajene ukaze za opravljanje veliko težkega dela, ki je vloženo v upravljanje aplikacij, kar nam omogoča avtomatizacijo dnevnih operacij (angl. *day-to-day operations*).

2. *Abstrakcija infrastrukture*

   Ko je sistem *Kubernetes* nameščen, sam upravlja z računskimi, omrežnimi ter skladiščnimi operacijami v sklopu naših delovnih obremenitev. Prednost tega je, da se lahko razvijalci osredotočijo na razvoj aplikacij in ne rabijo skrbeti glede okolja, v katerem bo aplikacija tekla.

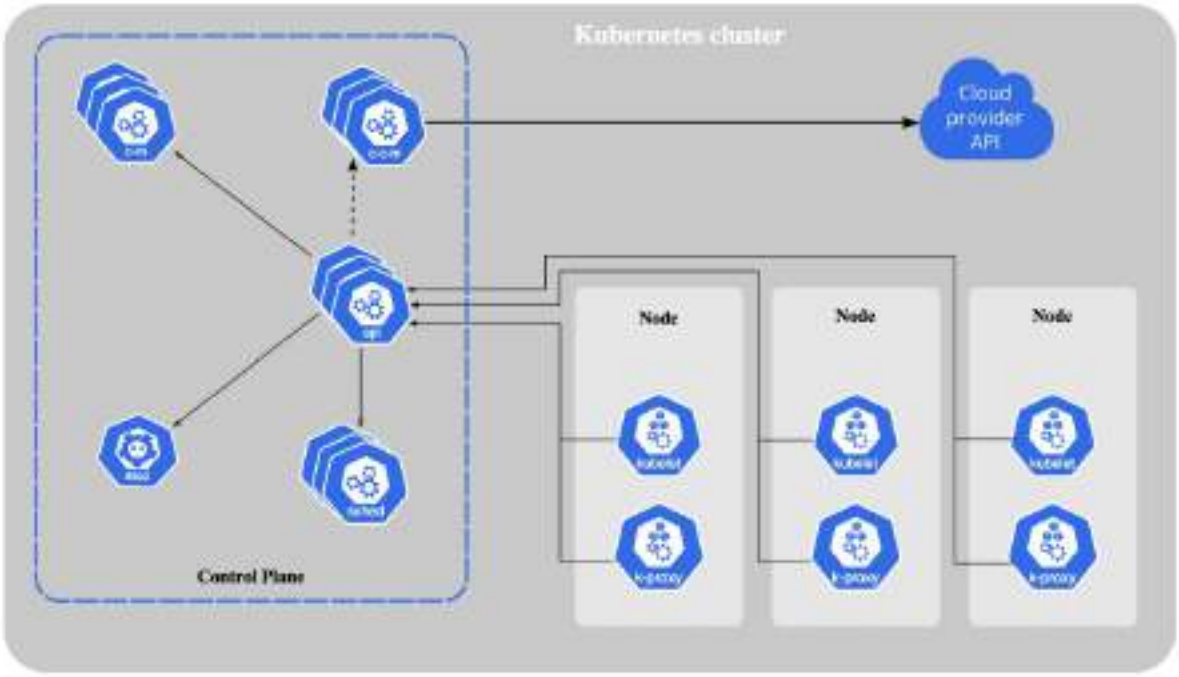
3. *Nadzor zdravja servisov*

   Kubernetes sproti izvaja zdravstvene preglede (angl. *health-checks*) servisov, ki tečejo na njem. Kontejnerje, ki so v neuspešnem stanju ali so neodzivni, samodejno ponovno zažene. Servisi postanejo končnim uporabnikom na voljo komaj takrat, ko sistem potrdi da vsi tečejo brez napak.

2.2.1 Arhitektura sistema Kubernetes

V grobem lahko najbolj zunanji sloj arhitekture sistema *Kubernetes* razdelimo na dva dela:

- Nadzorna ravnina (angl. *control plane*) – gradniki in komponente nadzorne ravnine sprejemajo glavne odločitve v gruči (npr. časovno razporejanje (angl. *scheduling*)), prav tako pa so odgovorni za zaznavanje in odzivanje dogodkov v gruči (npr. zagon dodatnega stroka (angl. *pod*), ko je njihovo število glede na polje »replicas« v načrtu posamezne aplikacije nezadostno). (The Kubernetes Authors, n.d.)
- Delavski gradniki (angl. *worker nodes*) – delavski gradniki so tisti strežniki, na katerih dejansko tečejo naši kontejnerji ali aplikacije, hkrati pa lahko na njih tečejo tudi komponente iz nadzorne ravnine. (Slingerland, 2023)



Slika 2 Vzorčen primer arhitekture gruče *Kubernetes*

Vir: (The Kubernetes Authors, n.d.)

2.2.1.1 Gradniki

Osnovni elementi gruče *Kubernetes*, na katerih tečejo aplikacije in kontejnerji, so gradniki (angl. *nodes*). Odvisno od posamezne gruče in specifike uvajanja gradnikov v le-to je gradnik lahko fizični ali virtualni strežnik. Gradnike ločimo na »vladarje« (angl. *masters*) in »delavce« (angl. *workers*). (The Kubernetes Authors, n.d.)

2.2.1.1.1 Vladarji (gradniki nadzorne ravnine)

Vladarji so gradniki nadzorne ravnine. Na njih tečejo naslednje komponente (The Kubernetes Authors, n.d.):

- *Kube-scheduler* – preverja in dodeljuje novo ustvarjene stroke na najbolj optimalne gradnike.
- *Kubelet* – enako kot pri delavskem gradniku tudi tukaj zagotavlja, da »workloads« tečejo tako, kot morajo. Ima pa še dodatno nalogo registracije novih gradnikov v »*Kubernetes* API strežnik« (komponento nadzorne ravnine).
- *Kube-apiserver* – izpostavlja (angl. *exposes*) *Kubernetes* API. Omogoča horizontalno širjenje, kar pomeni, da lahko v gručo dodamo več gradnikov, ki si naloge enakomerno prerazporedijo.
- *Etcd* – shramba tipa ključ-vrednost (angl. *key-value store*), ki deluje kot visoko dostopna varnostna kopija za podatke v gruči.
- *Kube-controller-manager* – na tej komponenti tečejo procesi nadzornikov (npr. nadzornik vozlišč, nadzornik nalog, …).

2.2.2 Viri sistema Kubernetes

2.2.2.1 Namespace

Imenski prostori (angl. *namespaces*) omogočajo, da gručo razdelimo na preglednejše logične segmente. V njih najdemo servise, skrivnosti, stroke, lahko pa jim dodajamo tudi označbe. (The Kubernetes Authors, n.d.) Delo brez imenskih prostorov pomeni, da imamo vse dele v privzetem imenskem prostoru, kar pa zelo hitro postane nepregledno in nam lahko povzroča težave.

2.2.2.2 Workload in pod

Delovne obremenitve (angl. *workloads*) so aplikacije, ki tečejo na sistemu *Kubernetes*. Sestavljene so lahko iz ene ali več komponent, ki delujejo skupaj. Te aplikacije tečejo na enem ali v setu strokov (angl. *pods*). (The Kubernetes Authors, n.d.)

Strok je najmanjša delovna enota, ki jo lahko ustvarimo v sistemu *Kubernetes*. Strok je skupina enega ali več kontejnerjev, v katerih tečejo komponente naših aplikacij. Vsi kontejnerji v stroku si delijo omrežno konfiguracijo, dostop do pomnilnika ter specifikacije, kako naj tečejo. Kontejnerji v stroku so vedno ko-locirani in vzpostavljeni skupaj ter tečejo v deljenem kontekstu. (The Kubernetes Authors, n.d.)

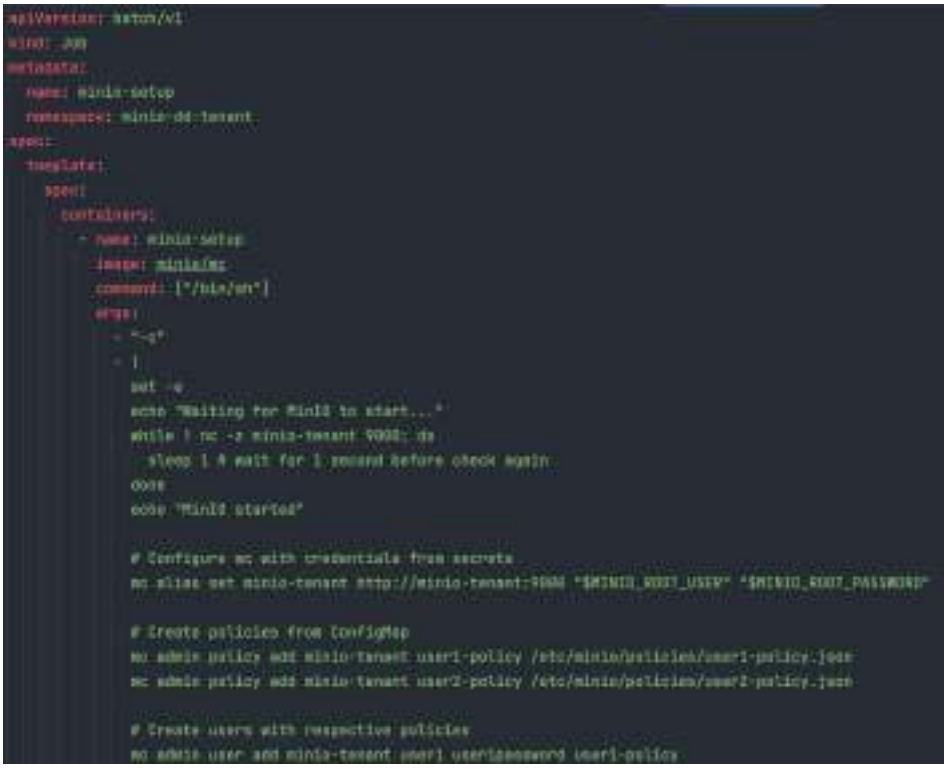
2.2.2.2.1 Upravljanje delovnih obremenitev

Delovne obremenitve lahko postavimo na različne načine, odvisno od potreb naših aplikacij. So en abstraktni nivo višje kot stroki, uporabljamo jih za definiranje navodil postavitve, števila in načina delovanja strokov. Primeri različnih delovnih obremenitev, ki so privzeto na voljo v sistemu *Kubernetes* (The Kubernetes Authors, n.d.):

- *Deployment* – je osnovni način za vzpostavljanje aplikacij v gruči. Uporablja se primarno za aplikacije brez stanja (angl. *stateless*), kjer so stroki med seboj zamenljivi in ne potrebujejo trajnega shranjevanja podatkov.
- *ReplicaSet* – set replik je podoben kot *deployment*, le da pri njem definiramo željeno število replik, ki jih potrebujemo v gruči. Če ne definiramo, na katerih vozliščih naj replike tečejo, bo *Kubernetes* samodejno poizkusil replike enakomerno prerazporediti čez vse delavske gradnike v gruči.
- *StatefulSet* – se uporablja, ko želimo, da ima vsak strok svojo ločeno identiteto in dostop do trajne shrambe podatkov. V njem definiramo enega ali več strokov, ki dobijo vsak svoj

ločen dostop do sistemskih virov. Razlika med *statefulSet* in *deployment* je, da stroki pri *StatefulSet* niso medsebojno zamenljivi, kar pomeni da so primernejši za aplikacije, ki potrebujejo trajnost pri delovanju (npr. baze podatkov).

- *DaemonSet* – zagotavlja, da bo na vsakem delavskem gradniku na voljo ena replika stroka. Primer uporabe je recimo, ko želimo, da aplikacija teče na vseh delavskih gradnikih, ki imajo dostop do določenega sistemskega vira (npr. ML-pospeševalnika), ali pa ko določen servis potrebujemo na vsakem vozlišču (npr. zbiratelj dnevniških knjig).
- *Job* – naloge so enkratne delovne obremenitve, ki izvedejo določeno nalogo le enkrat (npr. ustvarjanje dostopa za administratorskega uporabnika v aplikaciji).



Slika 3 Primer delovne obremenitve tipa job

Vir: lasten vir

Obseg delovne obremenitve lahko tudi avtomatsko prilagodimo, kar omogoča, da lahko avtomatsko ustvarimo dodatne stroke, če obstoječi stroki na gradnikih presežejo določene omejitve pri količini dodeljenih virov, in s tem aplikaciji povečamo delovno kapaciteto. (The Kubernetes Authors, n.d.)

2.2.2.3 Servisi



Slika 4 Primer definicije servisa

Vir: lasten vir

Servis (angl. *service*) se uporablja za omrežni dostop do naših delovnih obremenitev. Vsak servis ima definirano omrežno politiko, v kateri nastavimo vrata, na katerih naj posluša. Poznamo več različnih tipov servisov (The Kubernetes Authors, n.d.):

- *ClusterIP* – do servisa lahko dostopamo preko internega naslova IP v gruči, kar pomeni, da servis ni dostopen izven gruče. Ta tip je primeren, ko določene delovne obremenitve potrebujejo dostop do drugih, ne potrebujemo oz. ne želimo pa zunanjega dostopa.
- *NodePort* – servis je dostopen tudi zunaj gruče preko IP naslova vozlišča na vnaprej določenih vratih v rangu 30000-32767. Za potrebe dostopa v gruči pridobi servis tudi interni naslov IP.
- *LoadBalancer* – servis je dostopen zunaj gruče s pomočjo zunanjega izravnalnika obremenitve. *Kubernetes* sam po sebi te funkcionalnosti nima, izravnalniki obremenitve so na voljo pri oblačnih ponudnikih, pri lokalnih gručah pa lahko uporabimo rešitve kot so npr. *MetalLB, ServiceLB* ter *kube-vip*.

2.2.2.4 Trajni nosilci

Trajni nosilci (angl. *persistent volumes*) v sistemu *Kubernetes* omogočajo, da kontejner namerno zaustavi ali preneha delovati, obdrži dostop do datotek, ki jih je ustvaril oz. spremenil.

(The Kubernetes Authors, n.d.). Trajna shramba podatkov je pomembna pri *stateful* aplikacijah, ki za svoje delovanje potrebujejo dostop in trajnost podatkov. Brez trajnih nosilcev bi lahko uporabljali le t. i. *stateless* aplikacije, kar pa bi tako razvijalce kot uporabnike zelo omejilo pri možnostih uporabe.

Sistem *Kubernetes* podpira mnogo različnih tipov trajnih nosilcev, med njimi so:

- *hostPath* – trajni nosilec ustvarimo v mapi direktno na datotečnem sistemu vozlišča. Prednosti tega nosilca so, da je zelo lahek za vzpostavitev, vendar je potrebno zagotoviti, da bodo stroki, ki dostopajo do nosilca, tekli na istem vozlišču, kot je nosilec.
- *configMap* – nosilec tipa *configMap* omogoča, da v stroke posredujemo konfiguracijske podatke. V glavnem se uporablja za konfiguracijske datoteke, ni pa primeren za posredovanje občutljivih informacij, kot so gesla in privatni ključi.
- *Nfs* – omrežni datotečni sistem (angl. *network file system*) omogoča, da podatke hranimo na omrežni delitvi na sistemu izven gruče. Velika prednost pri tem sistemu je, da lahko stroki tečejo na kateremkoli vozlišču, če imajo vsa vozlišča dostop do sistema, ki ponuja to omrežno delitev.
- *Secret* – skrivnosti uporabljamo za posredovanje občutljivih informacij v stroke. Skrivnost lahko v stroku namestimo kot datoteko, lahko pa njeno vsebino pretvorimo v okoljske spremenljivke (angl. *environmental variables*).

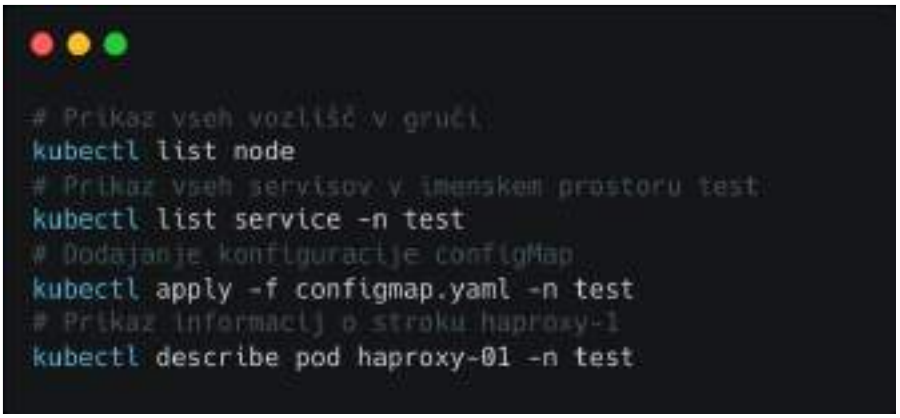
Ker je sistem *Kubernetes* izredno razširljiv, lahko podpira tudi alternativne nosilce, kot je na primer *Longhorn. Longhorn* je servis, ki omogoča kreacijo trajnih nosilcev v sistemu *Kubernetes*. Z *Longhorn* lahko ustvarimo trajni nosilec, ki je potem (če želimo) repliciran na vsa ostala vozlišča v gruči. (Cloud Native Storage with Longhorn: Overview, n.d.) navaja naslednje funkcije:

- *Longhorn* nosilce lahko uporabljamo kot trajno shrambo za distribuirane aplikacije z ohranjanjem stanja.
- Replikacija blokovnega shranjevanja čez več vozlišč za povečanje razpoložljivosti.
- Shranjevanje varnostnih kopij podatkov na zunanje sisteme, kot sta NFS ali AWS S3.
- Obnavljanje nosilcev iz varnostnih kopij.

2.2.3 Orodja

2.2.3.1 Kubectl

Kubectl je program v ukazni vrstici sistema *Kubernetes* namenjen za komunikacijo z nadzorno ravnijo. Z orodjem *kubectl* lahko nadziramo, dodajamo in odstranjujemo vse vire v gruči. Je osnoven pripomoček, ki je nujen za upravljanje z gručo.

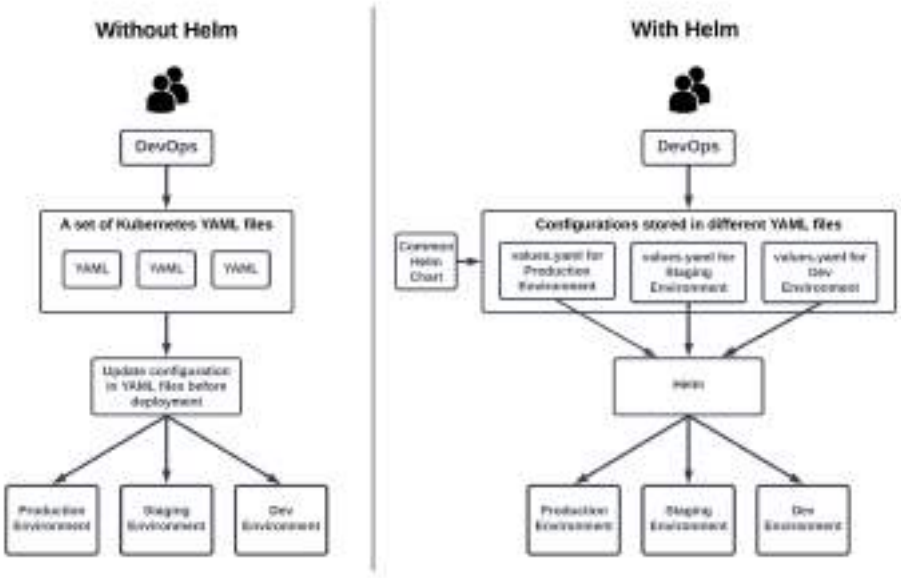


Slika 5 Primeri uporabe orodja kubectl

Vir: lasten vir

2.2.3.2 Helm

Helm je orodje, ki z avtomatizacijo združi ustvarjanje, pakiranje, konfiguracijo aplikacij v sistemu *Kubernetes* v en skupen paket, ki ga imenujemo Helm diagram (angl. Helm *chart*). (Schmitt, 2023)



Slika 6 Primerjava vzpostavitve aplikacije v več okoljih z in brez orodja Helm

Vir: (Schmitt, 2023)

Zgornji diagram prikazuje vzpostavitev aplikacije v več okoljih. Brez orodja Helm je potrebno za vsako ciljno okolje pripraviti ločene *Kubernetes* konfiguracijske datoteke, kar je pri kompleksnih aplikacijah zelo časovno zahtevna naloga. Helm ta proces poenostavi tako, da vrednosti loči od konfiguracijskih datotek, kar pomeni, da je potrebno pripraviti le toliko datotek z vrednostmi, kot imamo ciljnih okolij. Datoteka z vrednostmi lahko med drugim vsebuje:

- skrivnosti,
- podatke o želeni lokaciji naših strokov (*NodeAffinity, NodeAntiAffinity*),
- željeno število replik,
- konfiguracijske datoteke.

2.3 Docker

Docker je odprto kodna platforma, ki nam omogoča gradnjo, nameščanje, ter posodabljanje kontejnerskih aplikacij. (Susnjara & Smalley, 2024)

Je zelo dobra alternativa v primerjavi z sistemi navideznih strojev, saj porablja manj sistemskih virov. Uporablja arhitekturo tipa odjemalec – strežnik. Odjemalec sistema *docker* komunicira z prikritim procesom (angl. *daemon*), ki je odgovoren za vzpostavitev, zagon in distribucijo kontejnerjev. Tako odjemalec kot prikriti proces lahko tečeta na istem sistemu, z odjemalcem pa prikriti proces lahko upravlja tudi na daljavo. Komunikacija poteka preko aplikacijskega vmesnika REST, UNIX vtičnic ali omrežnega vmesnika. (Docker Inc., n.d.)

2.3.1 Docker Compose

Docker compose je orodje za lažjo vzpostavitev aplikacij na platformi *Docker*. Uporablja datoteke tipa *yaml,* v katerih definiramo naše servise, omrežja, podatkovne nosilce in (v primeru načina *swarm*) tudi skrivnosti. Glavna prednost dela s sistemom *compose* je, da zelo pospeši celoten deloven proces, saj avtomatizira in olajša samo vzpostavitev vseh potrebnih delov naših kontejnerskih aplikacij. V datotekah pa lahko uporabljamo tudi že prej definirane, tako da je delo z orodjem *Compose* zelo fleksibilno. (Docker Compose – What is It, Example & Tutorial, 2023)



Slika 7 Primer datoteke Docker Compose

Vir: Lasten vir

Na zgornji sliki lahko vidimo primer dela datoteke *Docker Compose*. V njej smo definirali:

- *Servis node_exporter.*
  - *Image*: kontejnerska slika in verzija, ki jo želimo uporabiti.
  - *Volumes*: podatkovni nosilci, ki jih bo kontejner uporabljal (nosilci so lahko trajnostni, ali pa kot v našem primeru poti na sistemu, ki jih bomo delili s kontejnerjem).
  - *Command*: zagonski ukaz za kontejner.
  - *Ports*: tukaj definiramo vrata, na katerih bo kontejner dostopen preko lokalnega omrežja.

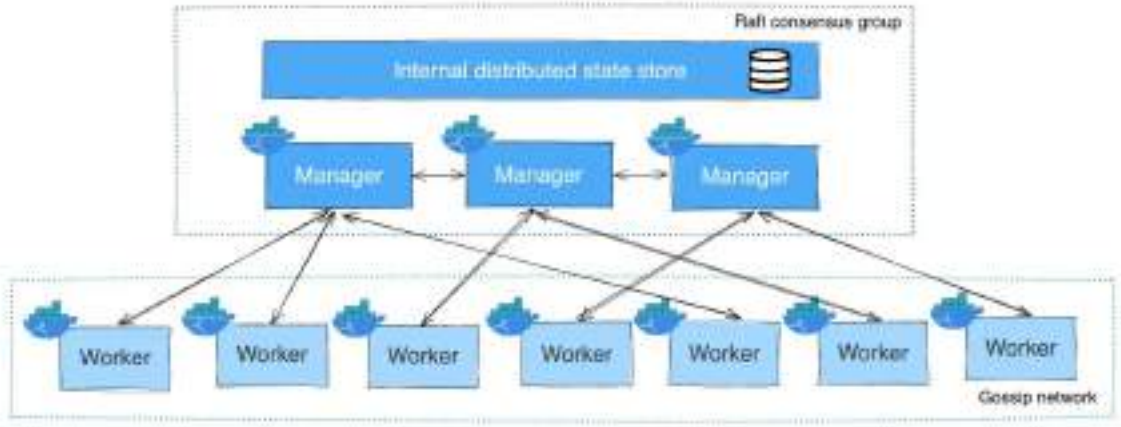
- *Target, published, protocol*: vrata na kontejnerju in vrata na gostitelju ter omrežni protokol.
- *Mode*: omrežni način v načinu *Swarm*; možnosti sta *host* (gostitelj) ali *overlay* (gruča); če izberemo način *overlay,* bo kontejner dostopen na vratih preko kateregakoli vozlišča v gruči.
  - *Restart*: tukaj izberemo način ponovnega zagona kontejnerja; možnosti so ob napaki (*on_error*), vedno (*always*), nikoli (*never*).
  - *Deploy*: samo za način *Swarm.*
- *Restart_policy:* sem vnesemo podatke o času in število poizkusov, v katerih naj bi se kontejner zagnal.
- *Mode*: število replik kontejnerja; *replicated* ali *global* (pri *replicated* v dodatnem polju še vnesemo število replik, global pa pomeni, da bo ena replika na voljo na vsakem vozlišču v gruči).
  - *Networks*: omrežja, v katera naj bo kontejner povezan.
- *Configs*: konfiguracijske datoteke za naše kontejnerje. Datoteke definiramo, nato pa jih na nivoju servisa vpnemo v kontejner.
- *Secrets*: skrivnosti z občutljivimi podatki, ki jih na nivoju servisa vpnemo v kontejner.
- *Networks*: tukaj definiramo omrežja, ki bodo v uporabi v naših kontejnerjih.

Omrežja, konfiguracije inskrivnosti so lahko tudi vnaprej definirana izven datoteke *compose*. V tem primeru pod imenom danega objekta vnesemo vrednost *external: true* in *Compose* bo avtomatsko poizkusil poiskati objekt. Če ga ne bo našel, pa nam bo sporočil napako.

2.3.2 Docker swarm

Docker Swarm je sistem za upravljanje gruč *docker* kontejnerjev. Z njim lahko vzpostavimo in upravljamo gruče sestavljene iz fizičnih ali virtualnih *docker* vozlišč. (Rahul & Casey, n.d.) Je del platforme Docker in predstavlja manj zahtevno, preprostejšo alternativo platformi *Kubernetes*.

2.3.2.1 Arhitektura



Slika 8 Arhitekutra gruče Swarm

Vir: (Docker Inc., n.d.)

Vozlišča gruče *Swarm* delimo na dva tipa; na delavce in upravljavce – vsako vozlišče lahko opravlja le eno ali pa tudi obe funkciji hkrati.

Upravljavci skrbijo za vzpostavitev in načrtovanje servisov, ohranjanje stanja gruče ter služijo kot HTTP dostopne točke za servise v gruči. Priporoča se, da je v gruči liho število upravljalnih vozlišč (min. 3), saj v primeru izgube gruča ne propade – za izračun lahko uporabimo formulo (n/2+1), kjer je *n* število upravljalnih vozlišč v gruči.

Delavci so pa vozlišča, na katerih dejansko tečejo naši servisi. Na njih lahko gledamo kot na normalne sisteme sz platformo Docker. Odgovorni so za dejansko izvajanje naših kontejnerjev, nimajo pa nobene upravljalne ali načrtovalne funkcije.

Funkcije sistema *Swarm*, kot so navedene v dokumentaciji (Docker Inc., n.d.):

- Upravljanje gruč, integrirano v *Docker engine*, omogoča, da z ukaznim orodjem *Docker* ustvarimo gručo *Docker* vozlišč, na katerih lahko tečejo naše aplikacije. Uporaba zunanjih orodij za vzpostavitev in upravljanje z gručami ni potrebna.
- Decentralizirana arhitektura omogoča vzpostavitev tako delavskih kot upravljalnih vozlišč, kar pomeni da lahko vzpostavimo celotno gručo z le eno sistemsko sliko.
- Deklarativen servisni model nam olajša delo pri vzpostavitvi samih servisov (*docker compose*).
- Za vsak servis lahko nastavimo želene nastavitve prilagajanja obsega, sistem *Swarm* pa bo le-te avtomatsko razporedil po gruči.

- Avtomatsko usklajevanje stanja v gruči – če imamo za en servis zahtevanih npr. 10 replik in dve od teh prenehata delovati, bo sistem *Swarm* avtomatsko ustvaril dve novi, da bo kvota dosežena.
- Omrežje z več gostitelji – vsak servis, ki mu dodelimo omrežno konfiguracijo tipa *overlay*, zagotavlja, da bo servis dostopen preko kateregakoli vozlišča v gruči.
- Dostopnost servisov po domenskem imenu in porazdelitev obremenitve – upravljavska vozlišča vsakemu servisu dodelijo unikatno domensko ime, kar avtomatsko porazdeli obremenitev med vse kontejnerje tega servisa.

V primerjavi z »navadnim« načinom delovanja platforme *Docker*, pa ima način *Swarm* tudi druge prednosti:

- Upravljanje z skrivnostmi: v načinu *Swarm* lahko na varen način v kontejnerje vpnemo datoteke z občutljivimi informacijami, kot so privatni ključi, gesla in uporabniški podatki.
- Upravljanje z konfiguracijski datotekami: omogoča da konfiguracijske datoteke definiramo na nivoju gruče – ni potrebe po prisotnosti konfiguracijske datoteke na vsakem vozlišču, saj so po kreaciji hranjene v upravljavcih in na zahtevo posredovane delavskim vozliščem.

V načinu *swarm* lahko več servisov združimo v en kopico (angl. *stack*). Datoteke *stack* so zelo podobne datotekam *compose*, le da lahko v njih uporabljamo še dodatne vrednosti, ki so specifične za način *Swarm*. Kopice združujejo prednosti datotek *compose* z naprednimi funkcionalnostmi sistema *Swarm*, kar naredi celoten sistem še bolj robusten in preprostejši za uporabo.

3 E-POŠTA IN POŠTNI STREŽNIKI

Sporočila, ki so poslana in prejeta preko računalniških sistemov in omrežij imenujemo e-pošta. E-poštni strežniki uporabnikom omogočajo, da preko omrežja pošiljajo besedilo in razne datoteke (kot priloge). E-poštni strežniki so računalniki, ki pošiljajo/prejemajo/posredujejo/hranijo e-poštna sporočila.

3.1 Zgodovina e-pošte in poštnih strežnikov

Prva elektronska sporočila so se preko telegrafov pošiljala že leta 1832, prva e-pošta pa se je pojavila leta 1965. Veliki računalniki *(*angl. *mainframes)* so bili omejeni na posamezna podjetja, koncept širše povezljivosti še ni obstajal. Uporabniki so do sistema dostopali preko terminalov, saj je zaradi velikosti in cene takratnih računalnikov podjetje običajno imelo le enega. Prva sporočila so se v tem času izmenjevala preko datotek na istem sistemu. (Markuson, 2021)

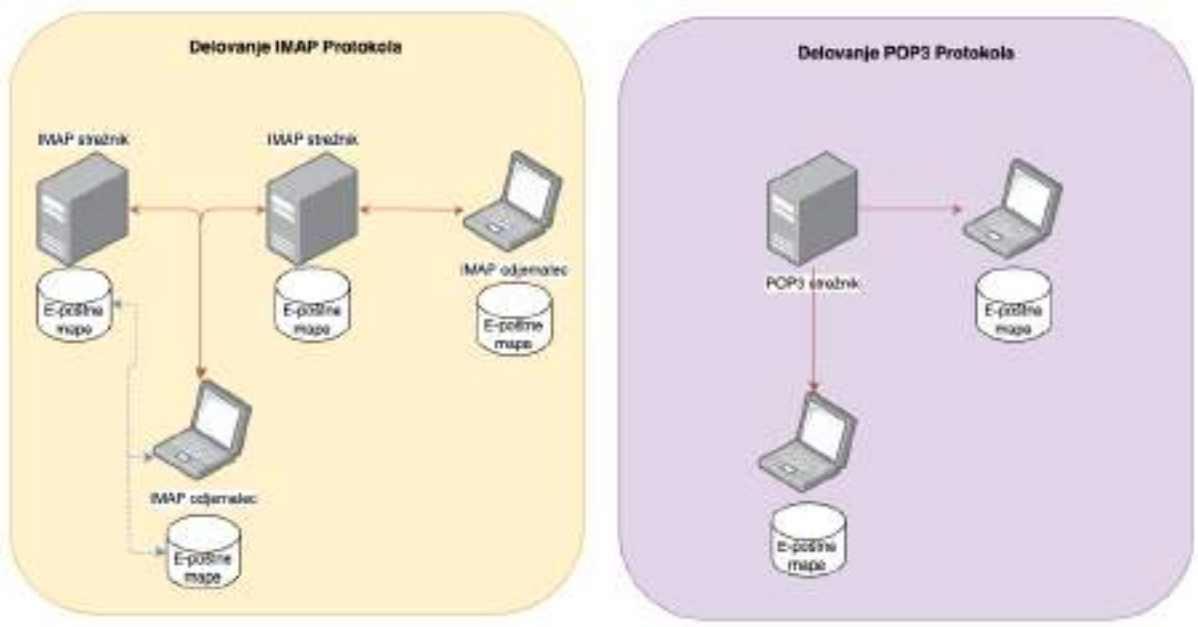
3.1.1 Prvi programi in protokoli

Kot izumitelja e-pošte mnogi štejejo Raya Tomilsona, ki je leta 1971 razvil program za pošiljanje sporočil med gostitelji preko ARPANET (predhodnik modernega interneta) SNDMSG (angl. *send message* – sln. pošlji sporočilo). Za definicijo naslova prejemnika in pošiljatelja je Tomilson izbral danes že znan format <ime>@<gostitelj>. Znak @ naj bi izbral čisto naključno, iskal je posebni znak, ki še ni bil v splošni uporabi med programerji. Prva e-poštna sporočila so bila poslana brez končnic (npr. .si, .com), saj še ni bilo potrebe po le-teh. Prejemnik je sporočilo prejel tako, da je bilo besedilo sporočila dodano v datoteko, ki je služila kot nabiralnik. Protokol, ki ga je uporabljal SNDMSG, je bil protokol za izmenjavo datotek FTP (RFC114). Skozi leta so mu dodajali vedno več komand in funkcij, leta 1982 pa se je iz njega rodil prvi namenski protokol za pošiljanje pošte SMTP (*Simple Mail Transport Protocol*). SMTP je spremenil način naslavljanja iz <uporabnik>@<gostitelj> v <uporabnik>@<domena>. (Markuson, 2021)

1983 se pojavi sistem za upravljanje imenskih strežnikov DNS (*Domain Name Service*), ki je vključeval zapisa *Mail Forwarder* MF in *Mail Destination* MD, ki sta se kasneje v letu 1987 združila v zapis MX, ki ga uporabljamo še danes. (Markuson, 2021)

3.1.2 Protokoli za prejemanje

Z prihodom osebnih računalnikov v domove se je pojavila tudi težnja po tem, da bi lahko uporabniki oddaljeno dostopali do svojih e-poštnih predalov. Ljudje do večjih sistemov niso več dostopali preko terminalov v podjetju, zato je bilo potrebno ustvariti tudi protokole za komunikacijo računalnikov s poštnimi strežniki. Najpogostejša protokola za prejemanje sporočil sta IMAP in POP. (Sanebox Inc., n.d.)



Slika 9 Diagram, ki prikazuje delovanje protokolov POP3 in IMAP

Vir: Lasten vir

3.1.2.1 Protokol POP

Protokol POP je nastal leta 1984 za potrebe uporabnikov, ki so potrebovali način do oddaljenega dostopa do svojih e-poštnih sporočil. Protokol je bil dvakrat nadgrajen – POP2 (1985) je dodal dodatne ukaze in spremenil način prenosa sporočil iz prvotnega protokola, ki je prenesel vsa sporočila hkrati, na možnost prenosa posameznih sporočil. Sledila je verzija POP3 (1988), ki je poenostavila prenos sporočil na osebnih računalnikih. POP3 je zadnja in še danes uporabljana verzija tega protokola. Protokol deluje tako, da se uporabnik identificira z uporabniškim imenom in geslom, na to pa na svojo napravo prenese e-poštna sporočila, ki so na strežniku. Ko so sporočila prenesena iz strežnika na uporabnikovo napravo, se takoj izbrišejo iz strežnika. (Sanebox Inc., n.d.)

Prednosti protokola POP3:

- Sporočila so na napravi na voljo tudi brez internetne povezave.
- Ker so sporočila shranjena na napravi kot datoteke, je postopek izdelave varnostnih kopij preprostejši.
- Veliko ponudnikov v današnjem času omogoča, da se sporočila še vedno ohranijo na strežniku, tako da imamo poleg lokalnih kopij tudi oblačne.
- Hitrost iskanja po prenesenih sporočilih je odvisna od diska v napravi in ne od hitrosti internetne povezave.

Slabosti protokola POP3:

- Če uporabljamo več naprav in želimo izbrisati sporočilo, je postopek potrebno ponoviti na vseh napravah.
- Pri prenosu sporočil na več naprav ni indikatorjev, da smo sporočilo že prebrali, označili ali izbrisali.
- Če so v sporočilih velike priponke, lahko sporočila zasedejo veliko prostora na naših napravah.
- Pri večjem številu sporočil na strežniku je lahko poraba pasovne širine pri prenosu zelo velika.

3.1.2.2 Protokol IMAP

Mark Crispin je leta 1986 na univerzi Stanford razvil protokol IMAP kot alternativo protokolu POP. IMAP je dvosmerni protokol, ki omogoča boljše funkcionalnosti za uporabnika. Protokol je bil kmalu po izidu nadgrajen z označevalnimi funkcionalnostmi, sledilo pa je še več revizij – IMAP2bis (zgodnja 90. leta prejšnjega stoletja) je dodal uporabo večnamenskih e-poštnih dodatkov (MIME), IMAP4rev1 (1996) je dodal uporabo tako šifriranih kot nešifriranih gesel in možnost šifriranja samih sporočil. (Sanebox Inc., n.d.)

Prednosti protokola IMAP:

- Sporočila so shranjena na strežniku, zato ne rabimo skrbeti za lokalne varnostne kopije sporočil.
- Podpira označevanje sporočil – prebrano, neprebrano, pomembno, …
- Podpira dvostransko urejanje sporočil in map – v primeru, ko imamo več naprav, na eni spremenimo mapo ali izbrišemo sporočilo, te spremembe pa se prenesejo na strežnik in so na voljo tudi na ostalih napravah.
- Izbris sporočila na eni napravi pomeni izbris sporočila na strežniku in na ostalih napravah.

Slabosti protokola IMAP:

- Do e-poštnih sporočil lahko dostopamo le z internetno povezavo.
- Če pomotoma izbrišemo sporočilo na eni napravi, je le-to izgubljeno saj se spremembe uveljavijo tudi na strežniku in ostalih napravah.
- Hitrost iskanja po sporočilih je odvisna od internetne povezave in strežnika.

3.1.3 Poštni odjemalci

Poštni odjemalec je program ali aplikacija, ki se uporablja za dostop do e-poštnih storitev. V grobem poznamo dva tipa – namizne poštne odjemalce (angl. *desktop mail clients*) in spletne poštne odjemalce (angl. *webmail clients*). Njihov namen je preprosto upravljanje z e-pošto in dostop do vseh e-poštnih storitev (pošiljanje, prejemanje shranjevanje,…). (Tschabitscher, 2019)

Prvi poštni odjemalec z grafičnim vmesnikom je bil Elm (1986), ki je v primerjavi z odjemalci v ukazni vrstici omogočal boljšo interaktivnost s poštnim predalom. Sledilo je vedno več grafičnih odjemalcev, med njimi tudi Pine (1992) in Mutt (1995). Odjemalci so skozi leta postajali vedno bolj prijazni za uporabo, sploh za netehnične uporabnike. (Bloom, 2017)

Prvi spletni e-poštni odjemalec je bil leta 1994 razvit na inštitutu CERN s strani Phillipa Hallam-Beckerja (Kalat, 2023). Šlo je sicer le za testno verzijo, ki ni bila nikoli na voljo širši publiki. Status prvega komercialnega *webmail* odjemalca pa pripada programu *cc:mail* podjetja Lotus, ki je izšel le leto kasneje (1995) (Bloom, 2017).

3.2 E-poštne rešitve

V moderni dobi se za večino e-poštne komunikacije uporabljajo brezplačne e-poštne rešitve. V pojem e-poštne rešitve vključujem različne kombinacije strežnikov SMTP, IMAP, POP in pa spletnih, namiznih ali mobilnih odjemalcev spletne pošte večjih tehnoloških podjetij (npr. Microsoft Outlook (bivši Hotmail), Google Gmail in Protonov Proton Mail).V organizacijah pa se zaradi potreb po večji varnosti, želje po integracijah z ostalimi internimi sistemi (npr. avtentikacija, enkratna prijava (angl. *SSO – Single Sign On*)) in tudi pogojev uporabe (večina brezplačnih rešitev, kot je npr. Gmail, v pogojih uporabe določa da se le-ta ne sme uporabljati v komercialne namene) uporabljajo rešitve, ki jih lahko delimo na dve kategoriji:

- Upravljani sistemi (angl. *managed systems*) – celotna rešitev gostuje na strežnikih ponudnika, kar pomeni, da stranki ni potrebno skrbeti za strojno opremo, na kateri teče, ter varnostne in sistemske posodobitve. Ima pa tudi ostale prednosti, kot so lažja administracija, velika dostopnost in varnost podatkov. Primeri upravljanih e-poštnih sistemov so npr. Microsoft Exchange, Google G-Suite ter Zimbra.
- Neupravljani sistemi tipa »na lokaciji« (angl. *OnPrem – On premise*), kar pomeni, da stranka sama poskrbi za gostovanje, podležne strežnike in vse povezave z njimi. Ta način se pogosto uporablja v zelo velikih organizacijah, ki imajo tehnološke kapacitete in interne sisteme za prijavo, administracijo uporabnikov. Velikokrat so tudi ponovni prodajalci (angl. *reseller*), saj pri njih z e-poštnimi predali gostujejo tudi ostala podjetja. Dober primer ponovnih prodajalcev so operaterji telekomunikacijskih storitev. Te rešitve so lahko brezplačne ali plačljive (med plačljivimi so tudi gostujoče verzije nekaterih upravljanih sistemov):
    - Brezplačne rešitve: Mailu, Stalwart mail, Mailcow, iRedMail.
    - Plačljive rešitve: Microsoft Exchange, Zimbra.

Pri tem bi rad dodal, da velikost podjetja ni pogoj za smiselnost uporabe upravljanega/neupravljanega poštnega sistema, saj je skozi zgodovino veliko ljudi samih gostovalo e-poštne sisteme, a je ta trend v zadnjih letih zaradi izboljšanja dostopnosti brezplačnih rešitev, zmanjšanega zanimanja širše javnosti in hkrati vedno večje težavnosti lastnega gostovanja pričel upadati. (Sholly, 2023) je leta 2023 izvedel anketo, v kateri je ljudi, ki se ukvarjajo z samo-gostovanjem (angl. *self-hosting*) storitev povprašal med drugim tudi o tem, če gostujejo svoj e-poštni strežnik. Na vprašanje »Ali sami gostujete e-poštni strežnik?«

sta 1602 vprašana odgovorila z »Ne« in le 267 vprašanih z »Da«, kar kaže na to, da tudi med ljudmi, ki so lastnemu gostovanju storitev naklonjeni, gostovanje lastnih e-poštnih strežnikov ni popularno.

Osebno menim, da se bo ta trend počasi začel spreminjati – kot je povedal (Fenollosa , 2022): »E-pošta je zdaj oligopol – storitev, ki je čuvana s strani peščice velikih podjetij, ki ne sledijo principom internetne nevtralnosti«.

3.3 Primerjava različnih potencialnih e-poštnih rešitev za izdelek

Pri izbiri najprimernejše e-poštne rešitve sem upošteval naslednje faktorje:

- Odprta koda – e-poštna rešitev more biti odprtokodna, ker to omogoča, da tudi sam pogledam v kodo in boljše razumem delovanje.
- Možnost postavitve v Docker Swarm in *Kubernetes* – če rešitve ne moremo postaviti v enem izmed danih sistemov, ni primerna za izdelek.
- Možnost porazdeljenega delovanja – ker želim predstaviti koncept visoke razpoložljivosti in porazdeljenosti poštnega strežnika.
- Anti-spam zaščita – želim le rešitve, ki omogočajo avtomatsko filtriranje neželene pošte.

Mailu je odprto kodni poštni strežnik sestavljen iz več Docker kontejnerjev. Je odprtokoden in stremi k temu, da bi naredil sam proces vzpostavitve čim bolj dostopen. Lahko ga postavimo tako v sistemu Docker kot *Kubernetes*. Vključuje IMAP, IMAP+ in SMTP strežnike. (Mailu authors, n.d.)

Docker-mailserver je preprost odprtokoden poštni strežnik, ki ne uporablja podatkovne baze, ampak hrani vso konfiguracijo na datotečnem sistemu. (Docker Mailserver Organization, n.d.)

Poštni strežnik Stalwart je odprtokodna e-poštna rešitev, ki vključuje SMTP, JMAP, IMAP4 in POP3 strežnike ter velik nabor modernih funkcij. Napisan je v programskem jeziku Rust. (Stalwart Labs Ltd., n.d.)

Za lažjo primerjavo sem iz dokumentacij za vse tri strežnike (Docker Mailserver Organization, n.d.), (Mailu authors, n.d.), (Stalwart Labs Ltd., n.d.) izpisal pomembne funkcije in pripravil naslednjo tabelo:

Tabela 1 Primerjava različnih e-poštnih rešitev

Ime	Podprti protokoli	*Kubernetes* in Docker?	Odprta koda?	Možnost porazdeljenega delovanja?	Anti-spam zaščita?
Mailu	IMAP, IMAP+, JMAP	Da	Da	V dokumentaciji ni podatkov.	Da
Docker mail server	IMAP, SMTP, POP3	Uradno samo docker, je pa možna tudi vzpostavitev z Helm diagramom.	Da	V uradni dokumentaciji o tem ne piše nič.	Da
Stalwart mail server	IMAP4, JMAP, POP3, SMTP	Da	Da	Da	Da

Vir: lasten vir

Za izdelek sem na koncu izbral poštni strežnik Stalwart, saj vključuje vse faktorje, ki sem si jih zastavil pri iskanju rešitve. Je res eden redkih poštnih strežnikov, ki omogoča porazdeljeno delovanje brez zahtevne dodatne konfiguracije. Dobra dokumentacija, aktivna in odzivna skupnost pa sta le še dodatna razloga za izbiro.

3.4 E-slama

Izraz e-slama (angl. *spam*) je bolj znan kot *vsiljena* ali *neželena pošta*. Gre za nenaročena e-poštna sporočila, ki so ponavadi poslana v velikem številu mnogim prejemnikom hkrati. Navadno se pošiljajo iz okuženih računalnikov, ki so del večje mreže. Ljudje vsiljene pošte ne marajo, ampak so se z njo že sprijaznili, saj so je že navajeni. Vsiljena pošta pa ni le moteča, lahko je tudi nevarna, saj pošiljatelji pogosto dodajo okužene priponke ali pa nas želijo pretentati s prevaro. (Awati & Teravainen, n.d.)

Ena izmed pogostih prevar se imenuje *ribarjenje s kopjem* (angl. *spear phishing*) – pri tem napadu je tarča ponavadi oseba, ki je zaposlena v določenem podjetju. Cilj napada je pridobiti uporabnikove prijavne podatke. Napadalec pred napadom najprej nekaj časa zbira informacije in osebne podatke o tarči. (Fortinet, Inc., n.d.)

E-slama pa lahko povzroči škodo tudi na drugačne načine, kot jih navaja (Kaspersky Lab, n.d.):

- Preobremenitev komunikacijskih sistemov – velike količine neželene pošte lahko po nepotrebnem preobremenijo komunikacijske kanale in s tem znatno upočasnijo delovanje omrežja. Posredna škoda je tukaj tudi ta, da mora uporabnik za ta prenos podatkov plačati, kar pomeni nepotrebne stroške.
- Nepotrebna izguba časa – če vsiljena pošta pride do prejemnikovega nabiralnika, jo more uporabnik ročno izbrisati. V primeru, ko nekdo prejme veliko poštnih sporočil na dan (npr. poslovna komunikacija) in zraven prejme še ogromno neželene pošte, lahko porabi mnogo nepotrebnega časa za brisanje.
- Pri brisanju neželenih sporočil pa se lahko zgodi tudi, da pomotoma izbrišemo pomembno sporočilo in če tega ne ugotovimo dovolj hitro, je to sporočilo za vedno izgubljeno.

Iz pridobljenih podatkov (Petrosyan, 2024) lahko razberemo, da je bilo v letu 2023 45,6 % vseh e-poštnih sporočil označenih kot neželenih. Zato je izredno pomembno, da za ustavljanje neželenih sporočil dobro poskrbimo.

3.5 Veliki podatki

3.5.1 Kaj so veliki podatki?

Veliki podatki (angl. *big data*) so raznolike zbirke strukturiranih, nestrukturiranih in polstrukturiranih podatkov, ki s časom eksponentno naraščajo. Zaradi njihove obsežnosti, kompleksnosti in raznolikost jih tradicionalni sistemi za upravljanje podatkov ne morejo shranjevati, obdelovati in analizirati. Uporabljajo se za strojno učenje, napovedno modeliranje in za napredne analitike za reševanje poslovnih problemov. (Google LLC, n.d.)

Veliki »V«-ji (dimenzije izzivov in priložnosti, ki so značilne za analizo velikih podatkov), kot jih opisujeta (Mucci & Stryker, 2024):

Obseg (angl. *volume*) – tehnologije za velike podatke in rešitve za shranjevanje v oblaku omogočajo organizacijam, da stroškovno učinkovito shranjujejo in upravljajo te obsežne nize podatkov ter preprečujejo izgubo dragocenih podatkov zaradi omejitev shranjevanja.

Hitrost (angl. *velocity*) – zaradi izjemno hitrega ustvarjanja podatkov v današnjem času – od posodobitev družbenih omrežij v realnem času do zapisov visokofrekvenčnega trgovanja z delnicami – je zelo pomembno, da lahko te podatke hitro obdelamo. Sistemi za obdelavo velikih podatkov omogočajo, da zelo hitro pripravimo analize le-teh.

Raznolikost (angl. *variety*) – današnji podatki se pojavljajo v različnih oblikah, od strukturiranih in numeričnih podatkov v tradicionalnih bazah podatkov do nestrukturiranih besedil, videov in slik iz različnih virov. Ta raznolikost zahteva fleksibilnost sistemov za upravljanje podatkov, ki lahko obvladujejo in integrirajo različne tipe podatkov za analizo. Baze podatkov NoSQL, jezera podatkov in tehnologije za branje shem omogočajo potrebno fleksibilnost za prilagajanje raznoliki naravi velikih podatkov.

Verodostojnost (angl. *veracity*) – tehnike in orodja za čiščenje podatkov, validacijo ter verifikacijo so ključnega pomena za zagotavljanje integritete velikih podatkov, kar organizacijam omogoča, da sprejemajo boljše odločitve na podlagi zanesljivih informacij.

Vrednost (angl. *value*) – cilj analitike velikih podatkov je izluščiti uporabne vpoglede, ki ponujajo oprijemljivo vrednost. To vključuje pretvarjanje obsežnih nizov podatkov v uporabne informacije. Napredna analitika, strojno učenje in umetna inteligenca so ključni za pridobivanje informacij, ki jih vsebujejo veliki podatki.

3.5.2 Prednosti in slabosti velikih podatkov v kontekstu e-pošte

Kot eno izmed prednosti uporabe velikih podatkov vidim možnost za boljšo in hitrejšo prepoznavo neželene pošte. V članku *Spamdoop: A Privacy-Preserving Big Data Platform for Collaborative Spam Detection* (AlMahmoud, Damiani, Hadi, & Al-Hammadi, 2019) je predstavljena platforma Spamdoop, ki z uporabo velikih podatkov pomaga pri zmanjševanju nezaželene pošte. Platforma deluje v velikem obsegu in uporablja tehnike, ki varujejo zasebnost vsebine e-pošte. Z integracijo teh pristopov s tehnologijo velikih podatkov Spamdoop omogoča učinkovito obdelavo obsežnih količin podatkov o e-pošti, hkrati pa zagotavlja zasebnost in izboljšuje učinkovitost detekcije neželene pošte.

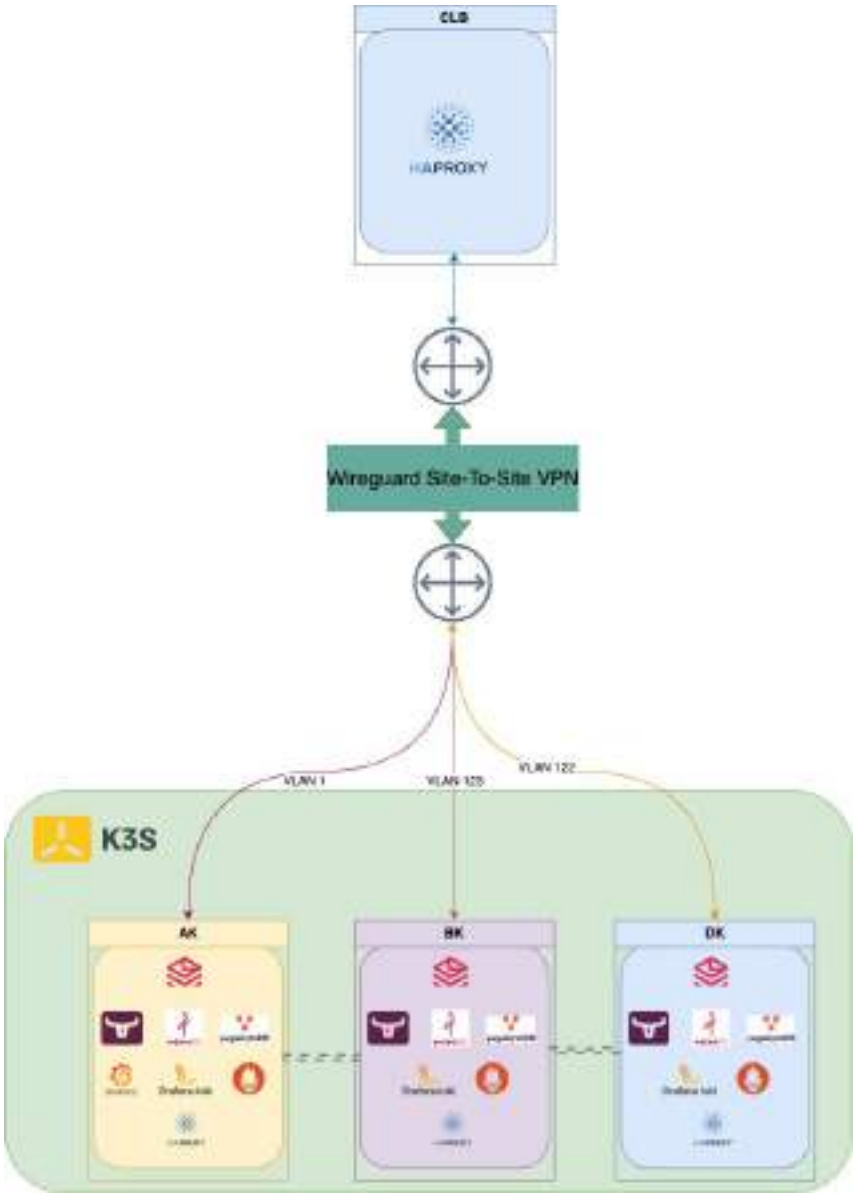
(Varghese, Jacob, & Supriya, 2017) v članku *Spam: A Big Data Challenge* razpravljajo o izzivih, ki nastajajo pri obdelovanju neželene pošte. Predstavijo tudi uporabo programske knjižnice Apache Majout, ki omogoča obdelovanje velikih podatkov s strojnim učenjem. Ugotovili so, da algoritmi Naive Bayes in Complement Naive Bayes, implementirani z uporabo Apache Mahout, nudijo zelo dobre rezultate pri klasifikaciji neželene pošte. Testiranje teh modelov na Enron podatkovnem nizu je pokazalo visoko stopnjo natančnosti pri pravilnem razvrščanju e-poštnih sporočil kot *spam* ali legitimna sporočila. Naive Bayes model je dosegel skoraj 93 % natančnost pri pravilni klasifikaciji.

Obdelovanje velikih podatkov ima lahko tudi negativne posledice za našo zasebnost. Podjetja pogosto zbirajo podatke o strankah, pri čemer te stranke niso ustrezno informirane o obsegu zbiranja podatkov. Poleg tega že samo shranjevanje tolikšne količine podatkov lahko privabi kibernetski kriminal – vdori v podatkovne sisteme lahko omogočijo nepooblaščen dostop do naših osebnih in občutljivih informacij, kar pogosto vodi do kraje identitete, prevar in izgube finančnih sredstev. Z analizo podatkov o uporabnikih lahko podjetja razlikujejo posameznike glede na njihove preference, spol, raso, spolno usmerjenost, kar lahko vodi do diskriminacije pri razlikah v spletnih cenah, oglaševanju in celo pri potencialnih zaposlitvah. (IAC.ai, 2023)

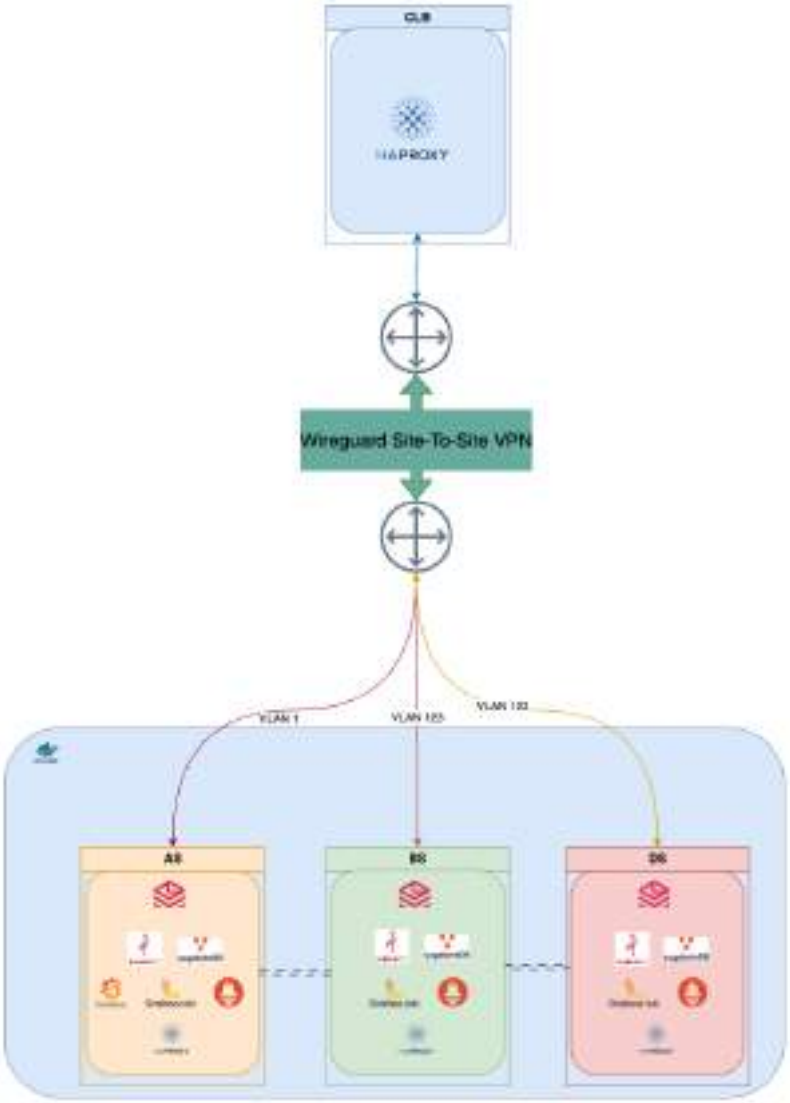
4 RAZISKOVALNI DEL / IZDELEK

4.1 Predstavitev izdelka

Izdelka sta dva. Eden je vzpostavljen s sistemom *Kubernetes*, drugi pa s sistemom Docker Swarm. Na spodnjih dveh slikah sta prikazani arhitekturi obeh izdelkov, bolj podrobno pa ju bom opisal v posameznih sklopih.



Slika 10 Izdelek z sistemom *Kubernetes*



Slika 11 Izdelek z sistemom Docker Swarm

4.2 Uporabljeni virtualni stroji

Naziv	vCPU	RAM	Disk	OS	Namen
AS	4	8 GB	20 + 40 GB	Ubuntu	Vozlišče Swarm
BS	4	8 GB	20 + 40 GB	Ubuntu	Vozlišče Swarm
DS	4	8 GB	20 + 40 GB	Ubuntu	Vozlišče Swarm
AK	4	8 GB	20 + 40 GB	Ubuntu	Vozlišče *Kubernetes*
BK	4	8 GB	20 + 40 GB	Ubuntu	Vozlišče *Kubernetes*
DK	4	8 GB	20 + 40 GB	Ubuntu	Vozlišče *Kubernetes*
CLB	4	4 GB	40 GB	Ubuntu	Izenačevalnik obremenitve
AG	2	4 GB	10 GB	OPNSense	Usmerjevalnik
CG	2	2 GB	10 GB	OPNSense	Usmerjevalnik

Tabela 2 Uporabljeni virtualni stroji

Vir: Lasten vir

Virtualni stroji AS, BS, DS so bili uporabljeni kot vozlišča za gručo Docker Swarm. Vsak od njih ima identične sistemske vire in vsak od njih je v ločenem podomrežju (za simulacijo treh različnih območij). Enako velja za stroje AK, BK, DK, le da so uporabljeni kot vozlišča za gručo *Kubernetes*. Vloga virtualnih strojev AG in CG je usmerjanje paketov, kreacija virtualnih omrežij in oddaljena VPN povezava med njima. Virtualni stroj CLB služi kot izenačevalnik obremenitve, na katerem je nameščen HA-Proxy v Docker kontejnerju. Vsi sistemi razen CG in CLB so gostovani na dveh fizičnih strežnikih na naši fizični lokaciji, CG in CLB pa sta virtualna stroja v oblaku ponudnika Hetzner.

4.3 Postopek vzpostavitve Kubernetes »gruče«

Za našo gručo sem izbral distribucijo sistema *Kubernetes* imenovano K3s. K3s je sistem, ki je popolnoma kompatibilen s sistemom *Kubernetes*, njegove prednosti pa so, da porablja manj sistemskih virov (velikost celotne izvršilne datoteke je manj kot 100 MB), je lažji za administracijo, prav tako pa ima vključeno rešitev za porazdeljevanje obremenitev ServiceLB.

Za vzpostavitev sem uporabil prosto dostopno orodje, ki temelji na knjigah iger orodja Ansible *ansible-k3s*. Skripta omogoča preprosto vzpostavitev gruč *Kubernetes* na operacijskih sistemih Debian, Ubuntu, Raspberry Pi OS, RHEL, SUSE in Arch Linux. (k3s-ansible authors, n.d.) Za tak način vzpostavitve sem se odločil, ker v primeru težav omogoča zelo hitro odstranitev ter ponovno vzpostavitev gruče *Kubernetes*.

Najprej smo v našo datoteko gostiteljev vnesli IP naslove in gesla za vse tri sisteme. To datoteko smo potem z *ansible-vault* šifrirali. Nato *ansible k3s* avtomatsko prenese vse potrebne skripte in pripravi prvo vozlišče v gruči. Ko je prvo vozlišče pripravljeno, sledi konfiguracija preostalih; to naredi z ukazom »*curl -sfL https://get.k3s.io | K3S_URL=https://myserver:6443 K3S_TOKEN=KLJUČ sh –«* kjer je ključ, ključ za pridružitev, ki ga dobi ob inicializaciji prvega vozlišča. Ko so vsa vozlišča pripravljena, lahko preverimo njihovo delovanje z ukazom na spodnji sliki:

```
k3u3@ubuntu-k3s-3:~$ kubectl get nodes
NAME           STATUS   ROLES                       AGE   VERSION
ubuntu-k3s-1   Ready    control-plane,etcd,master   9d    v1.26.9+k3s1
ubuntu-k3s-2   Ready    control-plane,etcd,master   9d    v1.26.9+k3s1
ubuntu-k3s-3   Ready    control-plane,etcd,master   9d    v1.26.9+k3s1
```

Slika 12 Ukaz za prikaz vozlišč gruče *Kubernetes*

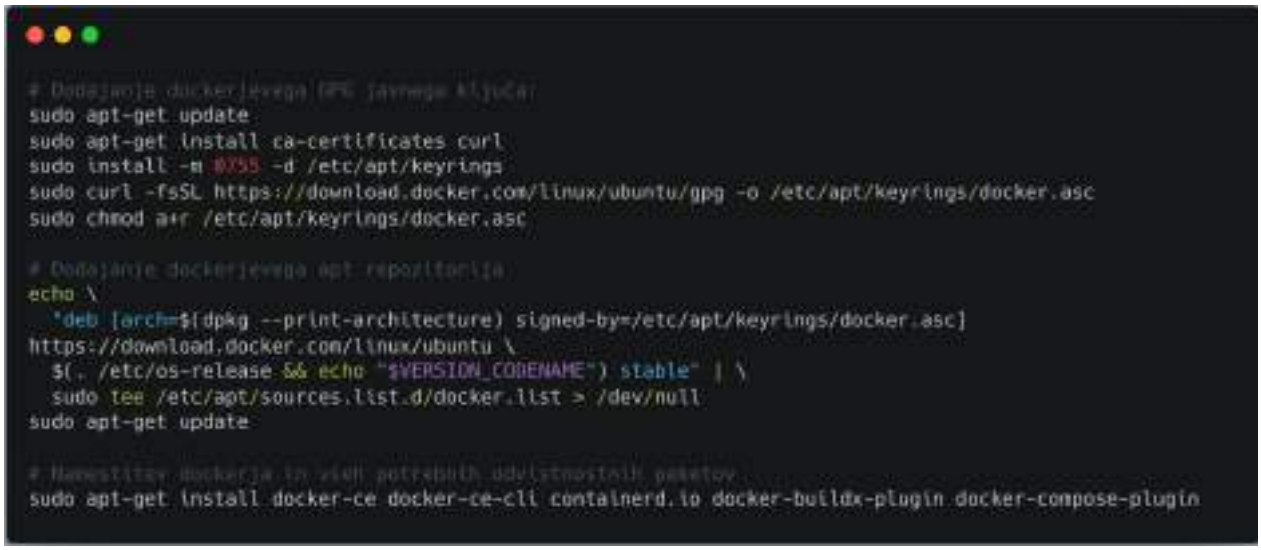
Vir: lasten vir

Če imajo vsa vozlišča status *Ready*, pomeni, da je bila inštalacija gruče uspešna.

Skripta *k3s-ansible* nam zelo olajša delo – moji prvi poizkusi so bili z ročno inicializacijo z orodjem *kubeadm* in težava se je vedno pojavila pri ponovni inicializaciji, saj je potrebno počistiti vse od Linuxovih tabel IP do raznih datotek, ki jih *Kubernetes* ustvari na vozlišču.

4.4 Postopek vzpostavitve Docker Swarm »gruče«

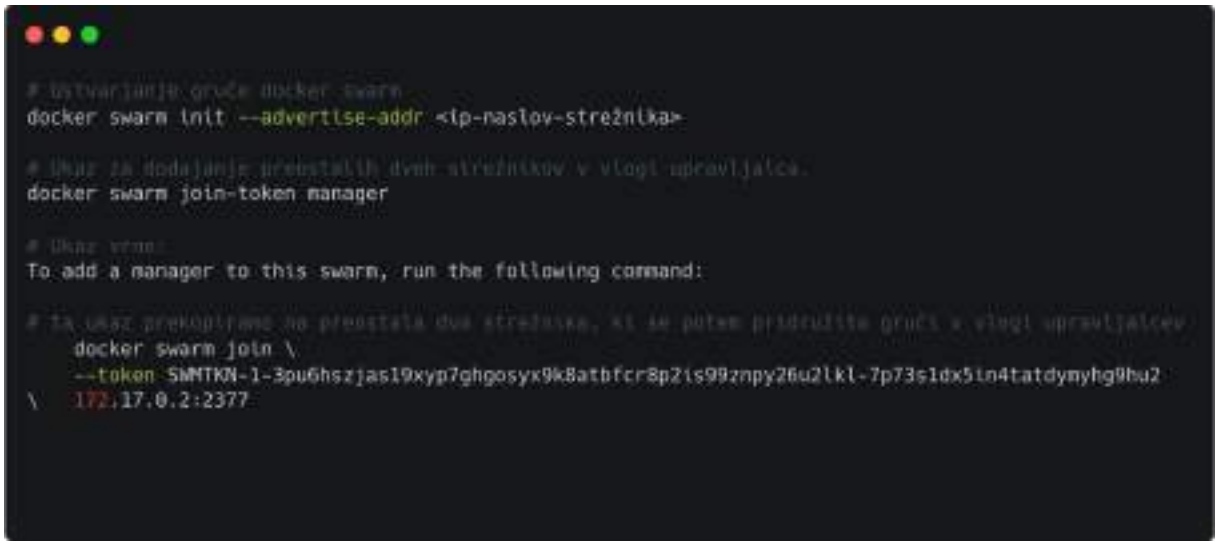
Vzpostavitev gruče *docker swarm* smo pričeli z namestitvijo sistema *docker* na vse tri strežnike. Uporabili smo naslednje ukaze:



Slika 13 Skripta za namestitev sistema *docker* na strežnik

Vir: lasten vir

Nato je sledilo ustvarjanje gruče. Najprej smo na enem strežniku ustvarili gručo z ukazom *docker swarm init* in nato dodali v gručo še preostala dva strežnika z ukazom *docker swarm join \ --token <ključ>*.



Slika 14 Ustvarjanje gruče *docker swarm* in dodajanje upravljalcev.

Vir: lasten vir

Sama vzpostavitev gruče *swarm* je bila izredno preprosta in hitra. Za postavitev sem porabil manj kot 10 minut.

4.5 Opis in razlaga uporabljenih servisov

4.5.1 Podatkovna baza YugabyteDB

YugabyteDB je odprto kodna porazdeljena SQL podatkovna baza. Združuje prednosti relacijskih podatkovnih baz (npr. ACID transakcij) s prednostmi globalno-porazdeljenih avtomatsko-drobljenih shramb (npr. NoSQL zbirke). (Mihalcea, 2023)

Ker je *yugabyteDB* oblačno-privzeta (angl. *cloud-native*) podatkovna baza, je zelo prilagodljiva glede na postavitev – lahko imamo veliko število na različnih geografskih lokacijah, kar je odlično za redundantnost in preklope v primeru izpada, podpira pa tudi velike količine podatkov v velikostih po več terabajtov. (Yugabyte, Inc, n.d.)

Sistem pa nima le SQL funkcionalnosti, podpira tudi YCQL (*Cassandra query language*) in YEDIS (*YugabyteDB* ekvivalenca shrambe v spominu *redis*), a teh dodatnih funkcionalnosti v našem primeru nismo uporabili.

4.5.1.1 Primerjava z PostgreSQL

Za to podatkovno bazo sem se odločil, ker je v osnovi baza *Postgres,* s katero imam že veliko izkušenj. Ampak pripraviti *Postgres* bazo za veliko razpoložljivost ekvivalentno sistemu *YugabyteDB* je glede na moje raziskave kompleksen postopek, za katerega moramo uporabiti orodja, kot so npr. *pgpool*, *yugabyteDB* pa je že privzeto pripravljen za način visoke razpoložljivosti.

YugabyteDB je zasnovan za rešitev potrebe po visoki razpoložljivosti, za katero monolitne baze podatkov, kot je *PostgreSQL*, niso bile nikoli zasnovane. To v osnovi pomeni, da se posodobitve izvedejo v enem dodatnem neodvisnem območju odpovedi v primerjavi s *PostgreSQL*. Nobeno izmed vozlišč v *YugabyteDB* ni absolutni »vodja«, ki bi bil odgovoren za posodobitve vseh podatkov v bazi. Obstaja več particij in te particije so razporejene med več vozlišči v gruči. Vsako vozlišče ima nekaj vodilnih in nekaj sledilnih particij. Odgovornost za pisanje nosi vodilna particija, ki nato uporabi protokol replikacije *Raft* za potrditev pisanja vsaj še eni sledilni repliki, preden nam uspešnost pisanja dokončno potrdi. Ko vozlišče odpove, se nekatere od vodilnih particij izgubijo, preostali dve sledilni repliki (na še vedno dostopnih vozliščih) pa v nekaj sekundah samodejno izvolita novega vodjo. Pri takšnih volitvah ima prednost replika z najnovejšimi podatki. To vodi do izjemno nizkega časa nedosegljivosti

pisanja in samozdravilnega sistema z avtomatskimi značilnostmi odpovedi (angl. *auto-failover characteristics*). (Yugabyte, Inc., n.d.)

4.5.1.2 Namen uporabe v našem izdelku in kritična ocena

V našem izdelku bosta v obeh sistemih vzpostavljeni bazi za poštni strežnik *Stalwart* in za sistem za vizualizacijo podatkov *Grafana*. Za oba sistema bo potrebno ustvariti ločena uporabnika za dostop.

V primeru sistema *Grafana* bo v bazi hranjena celotna konfiguracija, kar pomeni, da lahko sistem *Grafana* teoretično teče na kateremkoli vozlišču v gruči.

Pri poštnem strežniku pa bodo v bazi hranjena poštna sporočila, »vrsta sporočil« (angl. *message queue*), uporabljena pa bo tudi za shrambo vpogledov (angl. *lookup store*) ter kot sistem za »polno iskanje besedila« (angl. *full text search*). Shranjevanje vrste sporočil v bazo omogoča, da se pri večjem številu sporočil v vrsti le-ta enakomerno razporedijo med vozlišča. Uporaba baze kot shrambe vpogledov in kot sistem za polno iskanje besedila pa pomeni manjšo obremenitev procesov poštnega strežnika.

Hitra replikacija ter učinkovito deljenje pa v primeru izpada vozlišča pomenijo, da je verjetnost za izgubo ali nedostopnosti podatkov minimalna.

Zaradi vseh zgoraj navedenih prednosti ocenjujem, da je *YugabyteDB* odlična izbira v primerjavi z normalno bazo *PostgreSQL*. Ker je naš sistem razdeljen na tri »virtualna« oddaljena območja, bomo z uporabo te baze lahko dosegli odlično visoko razpoložljivost in redundanco z zelo dobro odzivnostjo ter zmogljivostjo. Velik faktor pri moji odločitvi pa je tudi odlična dokumentacija, ki ima zelo dobre napotke tako za postavitev v sistemu *Docker* kot v sistemu *Kubernetes*.

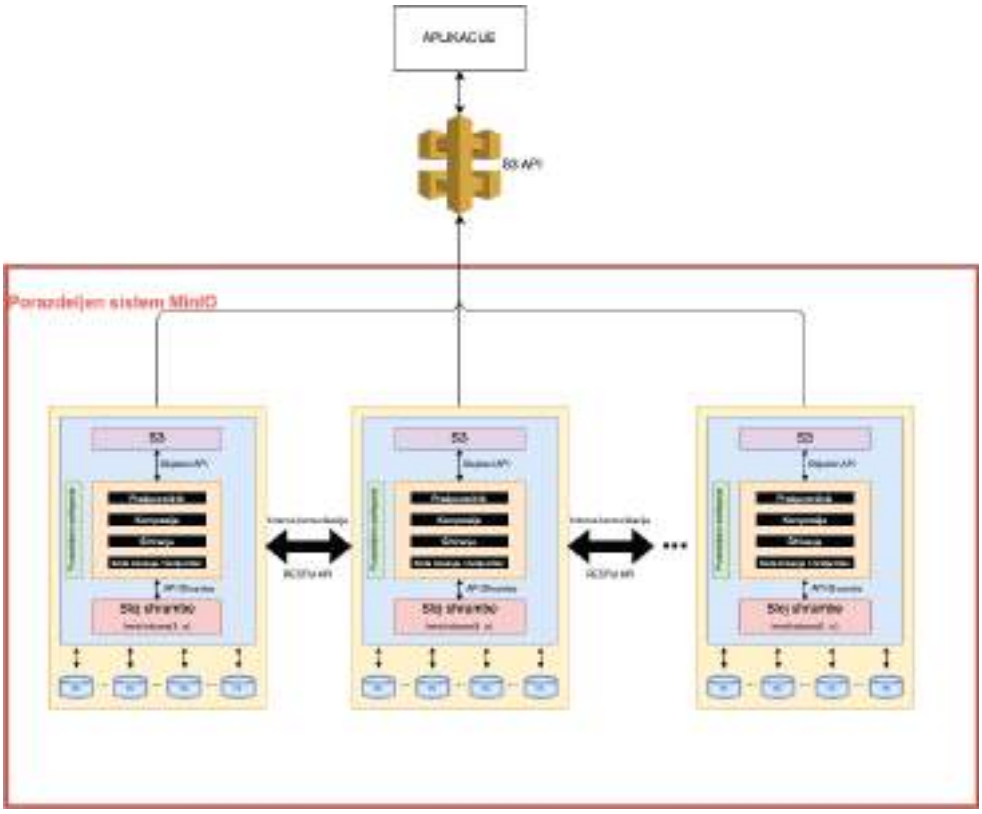
4.5.2 MinIO S3 objektna shramba podatkov

Sistem *MinIO* je odprto kodna porazdeljena objektna shramba podatkov. Njegov namen je priskrbeti funkcionalnosti shrambe tipa S3 v zasebnih oblačnih sistemih. Namestimo ga lahko v različnih konfiguracijah, z raznimi shrambami – lokalni disk, *Kubernetes* PVC in objektnimi shrambami javnih oblakov kot so Microsoftov Azure, ter Googlov GCP. Podatke replicira, deli in razširi čez več diskov, kar omogoča funkcionalnosti visoke razpoložljivosti in povečano zanesljivost. (Singh Gill, 2022)

Objekt je sestavljen iz binarnih podatkov, ki jim včasih rečemo »veliki binarni objekti« (angl. *Binary Large Object – blob*). *Blob* je lahko slika, zvočna datoteka, preglednica ali pa celo binarna izvršilna koda (.exe). (MinIO, Inc, n.d.)

Shrambe objektov, kot je *MinIO*, uporabljajo za organizacijo objektov t. i. vedra (angl. *buckets*). Vedro je idejno podobno mapi v datotečnem sistemu – vsako vedro lahko vsebuje večje število objektov. Vedra v sistemu *MinIO* imajo identične funkcionalnosti kot vedra v sistemu AWS S3.

Da je shramba objektov kompatibilna s S3 API, pomeni, da lahko z njo upravljamo preko REST API, ki je kompatibilen z Amazonovo S3 Shrambo (Amazon je S3 Shrambo predstavil leta 2006 (Amazon.com, Inc, 2006)), od takrat pa se je pojavilo še mnogo konkurenčnih rešitev, ki jih uporabljajo zaradi kompatibilnosti in statusa kot *de-facto* standard za shrambe objektov.



Slika 15 Arhitektura gruče sistema *MinIO*

Vir: Lasten vir

MinIO je oblačno zasnovan in lahko teče v kontejnerjih, ki jih upravljamo z zunanjimi orodji kot npr. *Kubernetes* ali v javnih oblakih. Najpogosteje se pojavlja v zasebnih oblakih z lokalno priklopljenimi diski (JBOD/JBOF). Vsi strežniki v gruči so enakovredno zmogljivi (polno simetrična arhitektura). Podatki in metapodatki so pri pisanju združeni v objekte, kar omogoča, da sistem deluje brez baze podatkov. Sistem pa vse funkcije (koda brisanja, preverjanje gnitja bitov (angl. *bitrot check*) in šifriranje) izvaja kot vrstične, striktno konsistentne operacije. (MinIO, Inc., n.d.)

Ta arhitektura omogoča, da sistem teče v velikem obsegu, z možnostjo horizontalnega prilagajanja obsega. (MinIO, Inc., n.d.)

Šifriranje na strani strežnika *MinIO* (SSE) ščiti objekte kot del operacij pisanja, kar omogoča odličen izkoristek strežnika za zaščito objektov na ravni shranjevanja (šifriranje v mirovanju (angl. *encryption at rest*)). SSE prav tako zagotavlja ključne funkcionalnosti za izpolnjevanje regulativnih zahtev, povezanih z varnim zaklepanjem in brisanjem. Za izvajanje varnih kriptografskih operacij uporablja storitev šifriranja ključev (KES) in zunanjo storitev za

upravljanje ključev (KMS). Podpira tudi upravljanje ključev s strani strank, pri čemer zunanja aplikacija (npr. *HashiCorp KeyVault, AWS SecretsManager, Google Cloud SecretManager*, ...) prevzame popolno odgovornost za ustvarjanje in upravljanje šifrirnih ključev, ki se uporabljajo z *MinIO* SSE. (MinIO, Inc, n.d.)

Za omejitve dostopov sistem *minIO* uporablja politike nadzora dostopa (angl. *policy-Based Access Control – PBAC)*. V vsaki politiki so opisani pogoji in ena ali več akcij, ki so povezane z dovoljenji uporabnikov oz. skupin uporabnikov.



Slika 16 Primer datoteke z politiko dostopa do vedra *stalwart-bucket*

Vir: Lasten vir

Datoteka s politiko je strukturirana v obliki zapisa JSON. Sestavljena je iz polj:

- Name – ime politike.
- Statement – izjava politike:
  - Effect – ali želimo dovoliti dostop do akcije X za vedro Y?
  - Resources – tukaj definiramo vedra, na katera politika vpliva, lahko pa tudi omejimo dostope do posameznih map v samem vedru.
  - Conditions – pogoji: Tukaj opcijsko definiramo pogoje, pod katerimi se politika izvaja, npr. glede na izvorni IP zahteve, uporabniško ime, identifikacijsko številko uporabnika, …
  - Actions – akcije, na katere vpliva politika zasebnosti (npr. *s3:GetObject*, *s3:DeleteObject*, *s3:PutObject*, …)

Politike nadzora dostopa lahko dodelimo na posamezne uporabnike, skupine uporabnikov, prav tako ima lahko ena politika več »izjav«, tako da lahko res natančno določimo pogoje dostopa za izbrana vedra, akcije in uporabnike.

V našem primeru smo ustvarili dve politiki zasebnosti, eno za servis Loki in eno za servis Stalwart – za oba servisa smo ustvarili tudi uporabnika, na katera smo naslovili politiki.

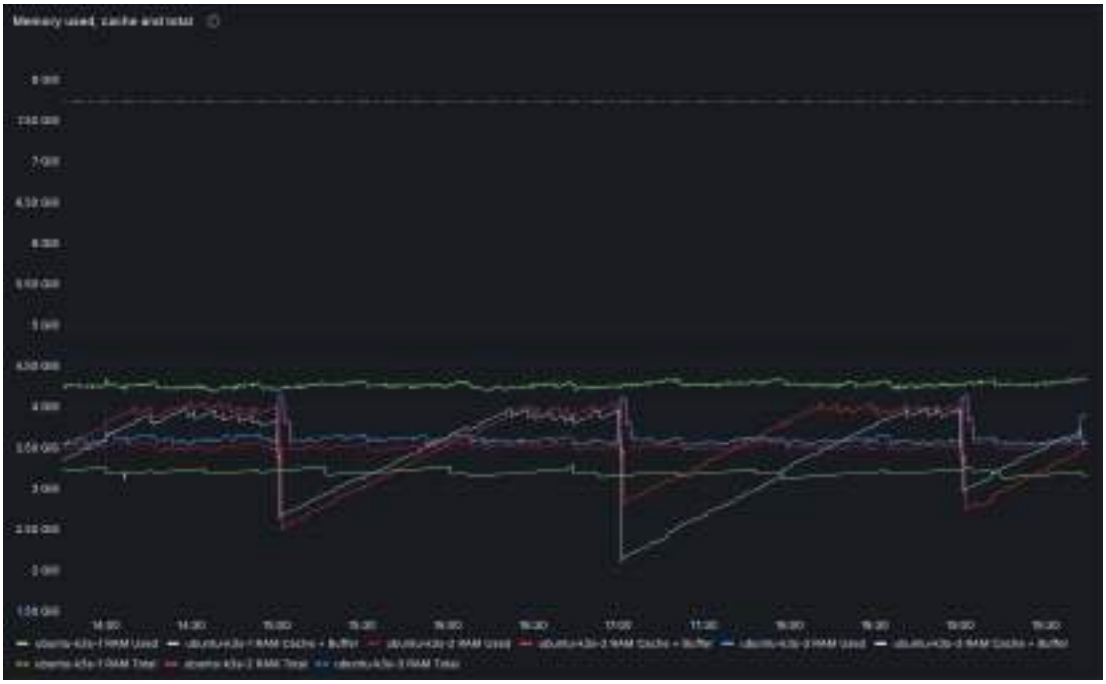
Ker sta visoka razpoložljivost in redundanca ena izmed ključnih delov našega izdelka, je pomembno, da smo izbrali shrambo, ki le-to tudi omogoča. Pomembna faktorja pa sta bila tudi preprostost vzpostavitve in konfiguracije ter kompatibilnost s tako *Docker Swarm* kot *Kubernetes*. V gruči *Kubernetes* za podložno shrambo uporabljamo sistem *Longhorn*, ki dejansko že sam po sebi zagotavlja tako replikacijo kot porazdeljenost podatkov, vendar na nivoju datotečnega sistema (PVC), ker pa je na voljo le v sistemu *Kubernetes*, ga žal ne moremo uporabiti v gruči *Docker Swarm*. *MinIO* je na voljo v obeh sistemih, pa tudi konfiguracija in sama vzpostavitev je relativno preprosta. Ostale alternative bi bile še npr. NFS, *GlusterFS* ali *sshfs*, vendar je za le-te potrebna konfiguracija izven gruče, kar poveča zahtevnost vzpostavitve in poveča težavnost večkratnih ponovnih vzpostavitev.

MinIO za shrambo uporabljamo pri sistemu za zbiranje dnevniških datotek Loki, za deljeno shranjevanje dnevniških datotek ter pri poštnem strežniku *Stalwart* za shrambo *BLOB* (*Binary Large OBject*), v katero shranjujemo priponke e-poštnih sporočil, saj te datoteke niso primerne za shranjevanje v podatkovno bazo.

Menim pa tudi, da je sistem *MinIO* zelo dober primer visoko razpoložljivega, porazdeljenega servisa, ki bo tekel na vseh vozliščih obeh gruč, hkrati pa bo za prej navedena dva servisa zagotavljal konsistentno, zanesljivo ter dostopno oblačno shrambo.

4.5.3 Grafana

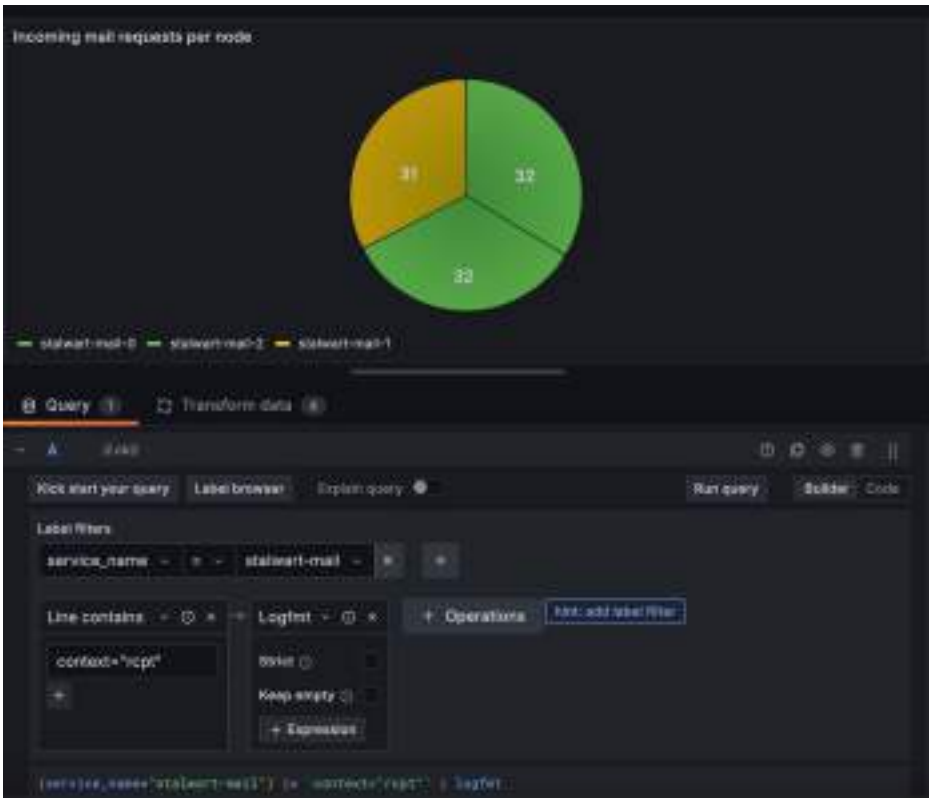
Grafana je odprto kodna interaktivna platforma za vizualizacijo podatkov. Uporabnikom omogoča, da predstavijo podatke z grafi in tabelami združenimi v eno ali več nadzornih plošč (angl. *dashboard*). Omogoča nam tudi, da ustvarimo lastne poizvedbe in opozorila za naše podatke, ne glede na to, kje so shranjeni – klasični strežniki, gruče *Kubernetes*,… Z vzpostavitvijo platforme *Grafana* imamo boljši pregled nad našimi podatki in tudi lažjo vizualizacijo delovanja naših aplikacij. (Red Hat, Inc., 2022)



Slika 17 Primer grafa ustvarjenega v platformi *Grafana*

Vir: lasten vir

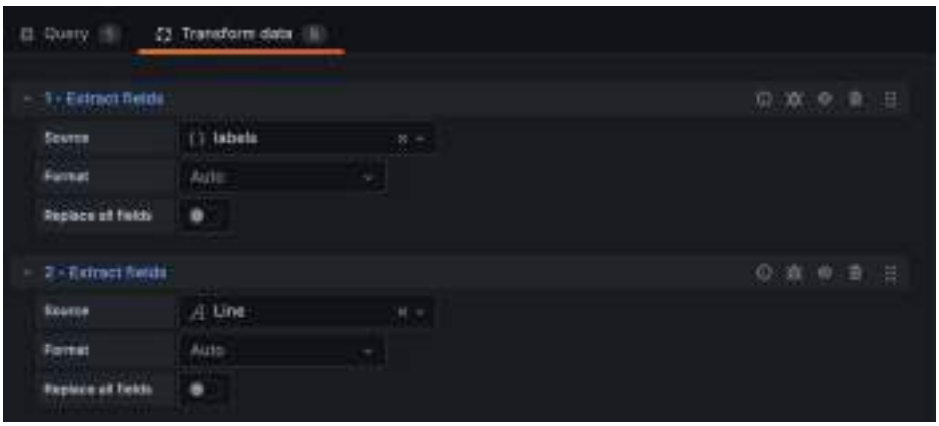
4.5.3.1 Primer ustvarjanja tortnega diagrama



Slika 18 Primer gradnikov poizvedbe v Grafani

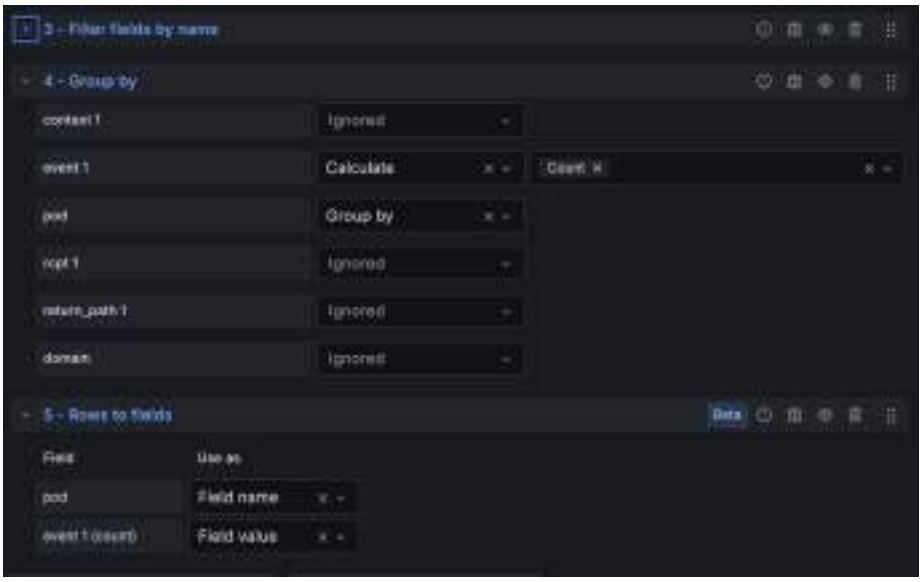
Vir: lasten vir

Ustvarjanje diagrama pričnemo s pisanjem poizvedb. V tem primeru smo za vir izbrali zbiralnik dnevniških datotek *Loki*. S filtri smo izbrali servis »stalwart-mail«, za podrobnejšo izbiro vrstic dnevniških zapisov pa smo dodali še filter »context=rcpt«, ki bo izbral vrstice z dogodki, ki se zgodijo ob prejemu sporočila. Operacija *Logfmt* pretvori dnevniške vrstice v obliko zapisa, ki je primerna za obdelavo z *Grafano*.



Slika 19 Primer transformacij v *Grafani* (1/2)

Vir: lasten vir



Slika 20 Primer transformacij v *Grafani* (2/2)

Vir: lasten vir

Po poizvedbah pa je bilo potrebno neobdelane podatke še transformirati in izvleči le podatke, ki jih potrebujemo. To sem dosegel z uporabo naslednjih transformacij:

- *Extract fields* – s to transformacijo smo iz stolpcev *labels* in *Line* pridobili polja z vrednostmi.
- *Filter fields by name in Group by* – s tema transformacijama smo izbrali polji *pod* in skupno število dogodkov *event*.
- *Rows to fields* – privzeto so podatki razvrščeni v vrstice, jaz pa sem želel poenostaviti izpis in sem zato le-te pretvoril v polja.

Rezultat je diagram iz prve slike te točke, na katerem vidimo, koliko sporočil je prispelo na posamezno instanco poštnega strežnika *Stalwart* v gruči *Kubernetes*.

4.5.3.2 Namen in uporaba v našem izdelku ter kritična ocena

Grafana podpira veliko različnih virov za podatke – izmed vseh možnih povezav bomo uporabili naslednji:

- *Loki* (zbiralnik dnevniških datotek).
- *Prometheus* (zbiralnik metrik delovanja).

Delo z *Grafano* je izjemno preprosto, kreacija grafov in tabel ne vzame veliko časa, omogoča pa tudi prilagajanje grafov in metrik, ki jih želimo vizualizirati. Omogoča tudi, da platformo postavimo z visoko razpoložljivostjo, a zaradi omejitev z našo podatkovno bazo (*YugabyteDB* ne podpira *Postgres* funkcije LOCK_ON_WRITE) takšnega načina postavitve ne bomo izbrali, kar niti ni tako kritično, ker gre le za vizualizacijsko orodje in ne za funkcionalni del našega visoko razpoložljivega poštnega sistema. Menim, da je *Grafana* zelo primerna za opazovanje in vizualizacijo našega sistema, in sicer zaradi dobre dokumentacije, preprostosti vzpostavitve, ter možnosti izvoza in uvoza posameznih nadzornih plošč, kar bo v fazi testiranja prihranilo mnogo časa.

4.5.4 Zbiranje dnevniških datotek in metrik

4.5.4.1 Loki in promtail

Pri nadzoru sistemov so dnevniške datoteke zelo dober vir informacij in podatkov, ki nam lahko pomagajo pri boljšem razumevanju delovanja sistema, prav tako nas opozarjajo na morebitne napake, ki nastanejo pri delovanju sistema.

Loki je zbiralnik dnevniških datotek, razvit iz strani podjetja Grafana Labs (isto podjetje, ki je razvilo sistem Grafana). Od ostalih zbiralnikov dnevniških datotek razlikuje po tem, da ne indeksira celotnih dnevniških sporočil, indeksira le oznake in metapodatke sporočil, procesiranje in poizvedbe pa prepusti sistemu Grafana. Ta način delovanja omogoča, da lahko procesira zelo veliko sporočil in hkrati porablja manj shrambe. Za dejansko zbiranje dnevniških datotek sistem *Loki* uporablja komponento *Promtail*. (Pratt, 2023)

Loki omogoča veliko različnih načinov vzpostavitve in delovanja, med njimi je tudi način visoke razpoložljivosti. Visoko razpoložljivost doseže tako, da podatke hrani v objektni shrambi S3 (v našem primeru bomo za to uporabili shrambo *MinIO* S3).

4.5.4.2 Prometheus in node_exporter

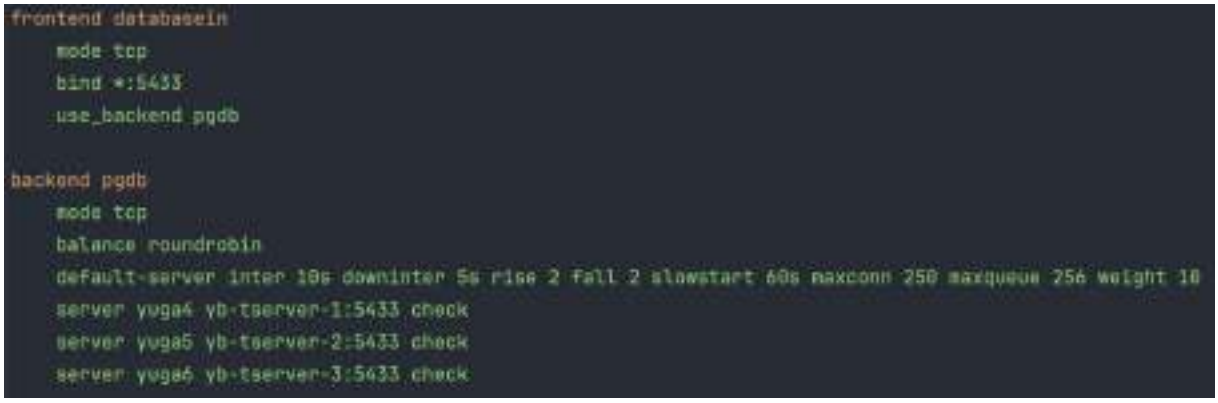
Prometheus je odprto kodna rešitev za zbiranje, organizacijo metrik naših sistemov in aplikacij. Deluje tako, da preko HTTP končnic na "tarčah" "strga" *(*angl. *Scrapes)* metrike. Podpira mnogo različnih tipov tarč, kot so sistemi za upravljanje z kontejnerji (*Kubernetes*), aplikacije ter servise (npr. sistemi za upravljanje z zbirkami podatkov). Za zbiranje sistemskih podatkov na UNIX operacijskih sistemih uporablja komponento *node_exporter*, ki deluje tako da bere metrike direktno iz jedra sistema. (Hailey, 2021)

4.5.4.3 Namen uporabe v izdelku in kritična ocena

Brez možnosti zbiranja podatkov v mojem izdelku ne bi mogel kasneje izvajati testov, za to je to eden izmed najbolj kritičnih delov samega izdelka. *Prometheus* ima zelo dobro podporo, veliko servisov izpostavlja HTTP končnice za metrike, a žal (še) med njimi ni poštnega strežnika *Stalwart*, za to je kritična uporaba zbiralca dnevniških knjig Loki. Med vzpostavljanjem izdelka sem pridobil zelo dobro razumevanje dnevniških zapisov poštenega strežnika, zato lahko to znanje in dnevniške zapise pretvorim v podatke, primerne za monitoring. Tako *Prometheus* in *Loki* podpirata delovanje v načinu visoke razpoložljivosti, kar omogoča tudi dobro redundanco v primeru izpadov. Obe rešitvi omogočata tudi zelo dobre prilagoditve, kot so npr. čas hranjenja podatkov ter tudi največja dovoljena velikost le-teh, kar omogoča, da lahko dobro delujeta tudi na sistemih z omejenimi sistemskimi viri. Podatke iz obeh sistemov bomo uporabili za vizualizacijo v sistemu *Grafana*.

4.5.5 HAProxy

HAProxy je odprto koden namestnik *(*angl. *proxy)* in *proxy* za izenačevanje obremenitve, ki podpira protokola HTTP in TCP. (LogicMonitor Inc., 2023) Poleg izenačevanja obremenitve omogoča tudi terminiranje TLS, dodajanje/ odstranjevanje HTTP glav *(*angl. *Headers)*, ščitenje pred napadi tipa DDoS in beleženje povezav. (HAProxy Community, n.d.)



Slika 21 Izsek iz konfiguracijske datoteke HAProxy

Vir: lasten vir

Konfiguracijo izvajamo preko datotek v obliki zapisa *yaml.* Na zgornji sliki je prikazan del konfiguracijske datoteke za podatkovno bazo *yugabyteDB.* V sekciji z imenom *Frontend* definiramo vrata in IP naslove, na katerih bo HAProxy poslušal, ter kam (na kateri *backend*, naj te povezave usmeri). V sekcijo *backend* pa smo zapisali podatke o strežnikih in način izenačevanja, ki ga želimo uporabiti.

Razlaga vrstic za *frontend:*

- *Mode*: protokol, ki ga želimo uporabiti (HTTP/TCP).
- *Bind*: IP naslov in vrata, na katerih naj posluša.
- *Use_backend*: *Backend,* na katerega naj bodo povezave usmerjene.

Razlaga vrstic za *backend*:

- *Mode*: protokol, ki ga želimo uporabiti (HTTP/TCP).
- **Balance**: način izenačevanja, ki ga želimo uporabiti (*round robin, leastconn,* ali *none*).
- Server <ime> <ip-ali-dns:VRATA> check : Tukaj definiramo strežnike in njihova vrata med katerimi bo potekalo izenačevanje obremenitve. “check” pomeni, da želimo da HAProxy preveri, če je strežnik dostopen.

4.5.5.1 Uporaba v našem izdelku in kritična ocena

HAProxy v izdelku uporabim tri krat:

1. Kot zunanji izenačevalnik obremenitve, za dostop do servisov v gručah Docker Swarm in *Kubernetes*
2. Kot izenačevalnik obremenitve v gruči Docker Swarm za servisa yugabyteDB in *minIO*
3. Kot izenačevalnik obremenitve v gruči *Kubernetes* za potrebe servisa yugabyteDB

HAProxy se mi za zgoraj navedene naloge zdi zelo primeren, zaradi preprostosti konfiguracije, možnosti postavitve v tako Docker Swarm kot Kuberntes gručah, ter nizke porabe sistemskih virov.

4.5.6 VPN povezava med strežniki

Obe gruči sta povezani z oddaljenim strežnikom *CLB* preko VPN povezave na usmerjevalnikih z tunelom Wireguard. Wireguard je varen omrežni tunel, ki želi zamenjati tako IPSec, kot popularnejše TLS rešitve kot je OpenVPN. V primerjavi z prej omenjenimi rešitvami je Wireguard varnejši, zmogljivejši in lažji za uporabo. (Donenfeld, 2020)

Povezava preko VPN-ja omogoča, da so servisi medsebojno dostopni brez odpiranja vrat na gostiteljih (razen vrat za VPN), omogoča pa tudi da je zunanja vstopna točka za naš poštni strežnik na tretji lokaciji, kar pomeni da vsa komunikacija teče preko tretje vstopne točke.

4.5.7 Ansible

Zaradi narave dela, kot je testiranje raznih konfiguracij, ter večkratnih ponovnih vzpostavitev sistemov, se mi je zdelo smiselno zadevo kolikor se le da avtomatizirati. Za to sem uporabil orodje Ansible.

Ansible je orodje, ki omogoča avtomatizacijo upravljanja konfiguracij, postavitve aplikacij, posodabljanja programske opreme in drugih potreb pri administrativnem delu z računalniškimi sistemi. (What is ansible?, n.d.)

Z orodjem *Ansible* lahko na zelo pregleden način v konfiguracijskih datotekah razdelimo posamezne naloge, ki jih povežemo v logične sklope.

4.5.7.1 Sklopi orodja Ansible in primeri konfiguracijskih datotek

V grobem se konfiguracijske datoteke orodja delijo na naslednje sklope:

- Datoteke s spremenljivkami (angl. *variables*) – v njih hranimo razne spremenljivke, ki jih bomo potrebovali v nalogah.
- Datoteke s podatki o gostiteljih (angl. *host files – inventory*) – tukaj hranimo.
- Knjige iger (angl. *playbooks*).
- Vloge (angl. *roles*) so mape, v katerih so razni sklopi nalog.
- Naloge (angl. *tasks*) so datoteke, v katerih izvajamo dejanske ukaze na gostiteljih.

4.5.7.1.1 Knjige iger

Najvišji sklop se imenuje »Knjiga iger« (angl. *playbook*), ki vsebuje vloge (angl. *roles*). Za primer bomo tukaj uporabili knjigo za vzpostavitev poštnega strežnika in ostalih odvisnosti v sistemu *Docker swarm*.



Slika 22 Primer knjige iger za vzpostavitev poštnega strežnika in ostalih odvisnosti v sistemu *Docker swarm*

Vir: lasten vir

V knjigo smo v našem primeru napisali:

1. Na katerih napravah želimo, da se naloge izvedejo (angl. *hosts*).
2. Če želimo, da se knjiga izvede z administratorskimi pravicami? (angl. *become*)

3. Če želimo pridobiti informacije o sistemih (angl. *gather_facts*)
4. Katere datoteke s spremenljivkami naj bodo v uporabi? (angl. *vars_files*)
5. Vloge in naloge v knjigi.

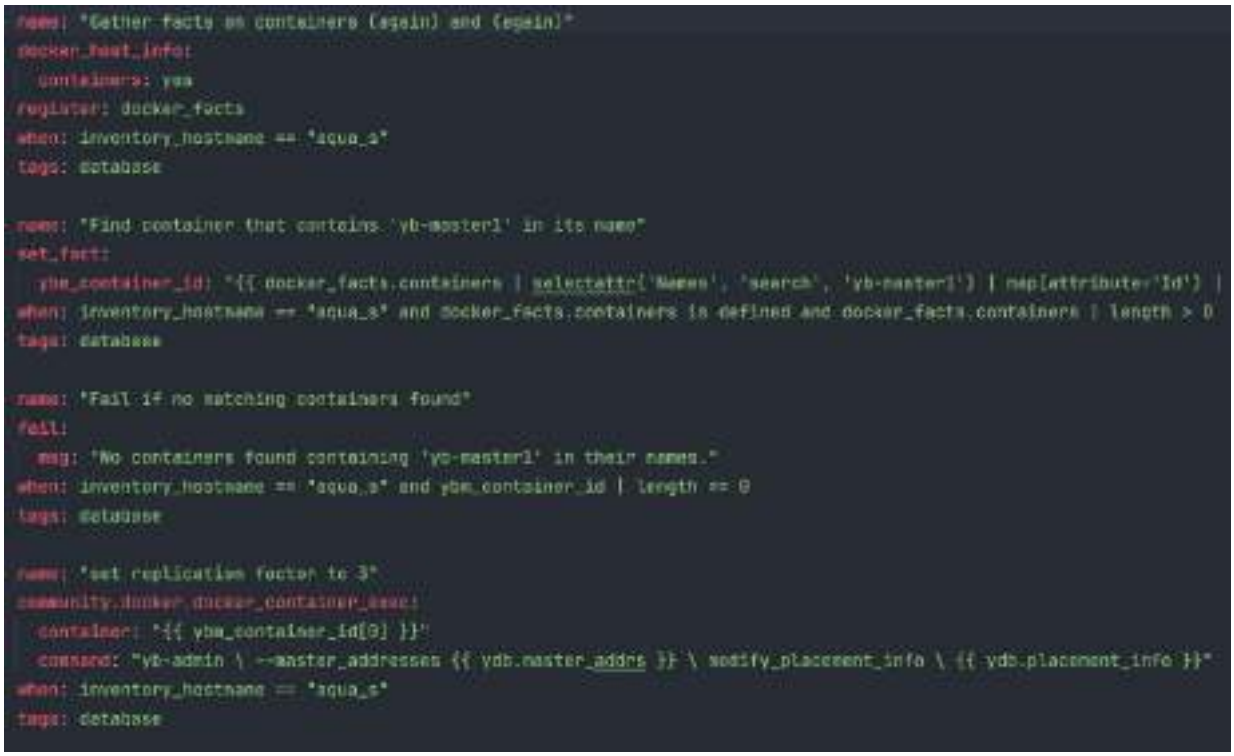
4.5.7.1.2 Vloge in naloge

Vsaka vloga vsebuje svoje naloge – lahko uporabimo vse naloge za dano vlogo, lahko pa tudi uporabimo le določene naloge, čisto odvisno od primera uporabe.

Če v knjigi pod vlogo napišemo le ime vloge, npr. *vloga*, se bodo izvedle vse naloge, ki so napisane v datoteki *main.yml*, če pa napišemo npr. *vloga/naloga* se bodo izvedle le naloge zapisane v datoteki *naloga.yml* . To nam omogoča, da posamezne naloge uporabimo le v določenih primerih in pa tudi večkrat.

Za vsako posamezno nalogo lahko tudi točno nastavimo, kdaj se naj izvede (npr. da se izvede le enkrat ali da se izvede le na določenem gostitelju), kar nam zelo olajša delo in omogoča veliko prilagodljivost pri konfiguraciji.

Kot primer bomo tukaj uporabili vlogo *deployment_management* in nalogo *configureDbReplicas*.



Slika 23 Primer datoteke z nalogami *configureDbReplicas* v vlogi *deployment_management*

Vir: lasten vir

V našem primeru datoteka *configureDbReplicas.yml* vsebuje naslednje elemente:

1. Ime naloge.
2. Ukaze (*docker_host_info*, register, *set_fact*, *fail*, *docker_container_exec*).
3. Pogoje izvedbe (*when*).
4. Oznake (*tags*).

Ukazi izvedejo določene akcije na gostitelju. Npr. *docker_container_exec* v kontejnerju *yb-master-1* izvede ukaz za označevanje posameznih strežnikov podatkovnih baz. Tukaj bi rad izpostavil še ukaz »*fail*«. Ukaz *fail* omogoča, da v določenih primerih prekinemo izvedbo knjige z namenom preprečevanja morebitnih napak – v našem primeru se to zgodi, ko orodje ne najde kontejnerja z imenom »yb-master-1«, saj če ta kontejner ne obstaja, ukazi za le-tem ne bodo delovali.

Pogoji izvedbe (*when*) omogočajo, da se posamezne naloge izvedejo le enkrat ali le na določenih gostiteljih. V našem primeru smo pogoj *when* uporabili, za to,, ker vemo, da bo kontejner »yb-master-1« vedno na voljo le na gostitelju »aqua_s«.

Oznake omogočajo, da izvedemo le ukaze za določeno kategorijo. Npr. v knjigo iger bi lahko napisali, da naj se v njej izvedejo le naloge z oznako »ime_oznake« in potem se bi izvedle le naloge z to oznako. V našem primeru smo naloge označili z oznako »database«, le za primer dobre prakse, saj oznak v naših knjigah v tem kontekstu ne uporabljamo (selekcije izvajamo na drugačne načine).

4.5.7.1.3 Datoteke z spremenljivkami



Slika 24 primer datoteke z spremenljivkami monitoring_vars.json

Vir: lasten vir

V datotekah z spremenljivkami hranimo spremenljivke, ki jih bomo uporabljali v naših nalogah. Datoteke so lahko tipa *yaml (YAML Ain't Markup Language)* ali tipa *json (JavaScript Object Notation)*. V našem primeru smo uporabili datoteke tipa *json*, v katerih hranimo podatke o mapah, ki jih bomo ustvarili na gostiteljih, skrivnosti kot so gesla in uporabniška imena za povezavo s podatkovno bazo, …

Brez uporabe teh datotek bi bili primorani te vrednosti fizično vnesti v naloge ali pa jih posredovati kot vrstične argumente pri izvedbi knjige iger, kar bi omejilo ponovno uporabnost samih nalog.

4.5.7.1.4 Datoteke z podatki o gostiteljih

V datotekah s podatki o gostiteljih hranimo podatke, ki jih bo orodje potrebovalo za povezavo na posamezne gostitelje (ip naslov, ime uporabnika, geslo za pridobitev administratorskih pravic). Posamezne gostitelje lahko dodelimo v razne podskupine, kar ponovno omogoča dodatno fleksibilnost in prilagoditve pri izvedbi knjig iger.



Slika 25 Primer datoteke z podatki o gostiteljih

Vir: lasten vir

4.5.7.1.5 Šifriranje z ansible vault

Vault je del orodja *ansible*, ki omogoča šifriranje datotek. Ker v konfiguracijskih datotekah pogosto hranimo občutljive podatke, kot so gesla, uporabniška imena in IP naslovi, je res ključen del orodja, saj bi brez njega bili prisiljeni vse te podatke hraniti kot nezaščiteno besedilo. Šifriramo lahko katerokoli datoteko – od datotek s podatki o gostiteljih, datotek z spremenljivkami pa vse do raznih konfiguracijskih datotek, ki bodo v uporabi na gostiteljih, omejitev praktično ni.

Šifriranje in dešifriranje izvedemo z naslednjimi ukazi:



Slika 26 Ukazi za šifriranje in dešifriranje datotek z ansible-vault

Vir: lasten vir

4.5.8 Konfiguracija uporabnikov v LDAP sistemu FreeIPA

FreeIPA je odprto kodna rešitev, ki združuje imeniški strežnik (*LDAP*), *MIT Kerbebors*, NTP in DNS strežnika insistem za upravljanje z certifikati. Administracijo omogoča preko spletnega uporabniškega vmesnika in preko ukazne vrstice. (FreeIPA Team, n.d.)

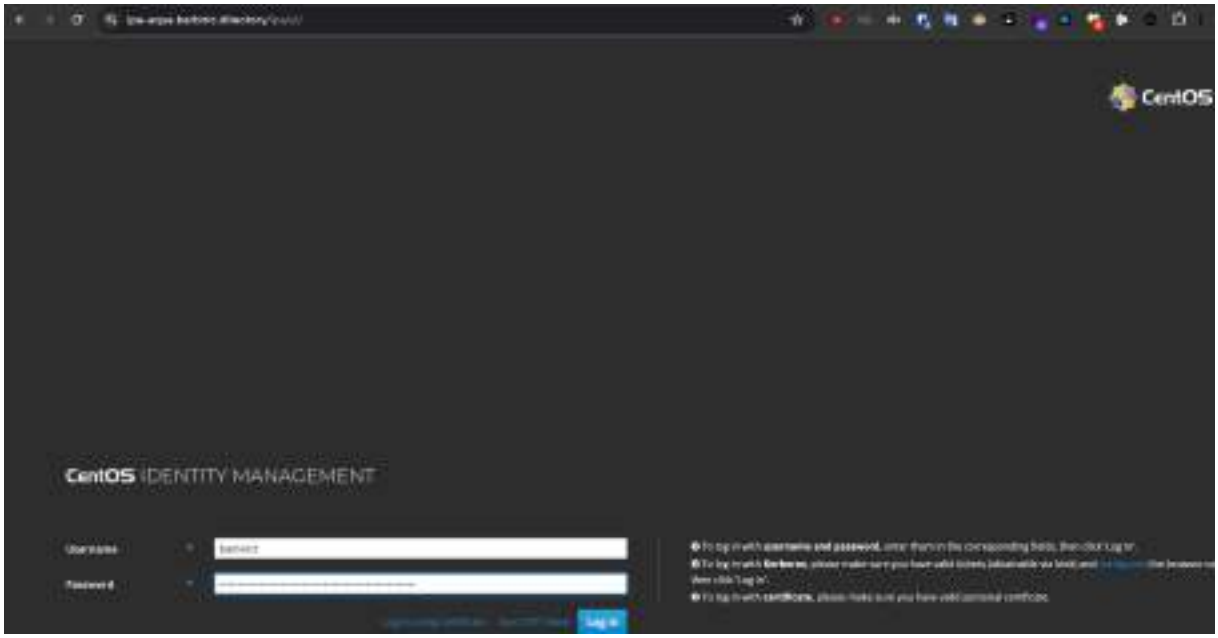
Primarno se uporablja za centralno administracijo UNIX sistemov in ima podobne funkcije kot Microsoftov *Active Directory* za sisteme Windows. Podpira tudi način delovanja z več strežniki, kar omogoča dobro redundanco ter visoko razpoložljivost. Sistem je tudi izredno varen in podpira mnogo različnih načinov avtentikacije – med drugim tudi avtentikacijo v dveh korakih.

Sistem sem izbral, ker imam z njim že predhodne izkušnje , je že bil postavljen pred pisanjem same diplomske naloge, zaradi vgrajenih funkcij LDAP in DNS ter zaradi preproste administracije.

4.5.8.1 Primer dodajanja uporabnika testDD

Postopek za dodajanje uporabnika *testDD*:

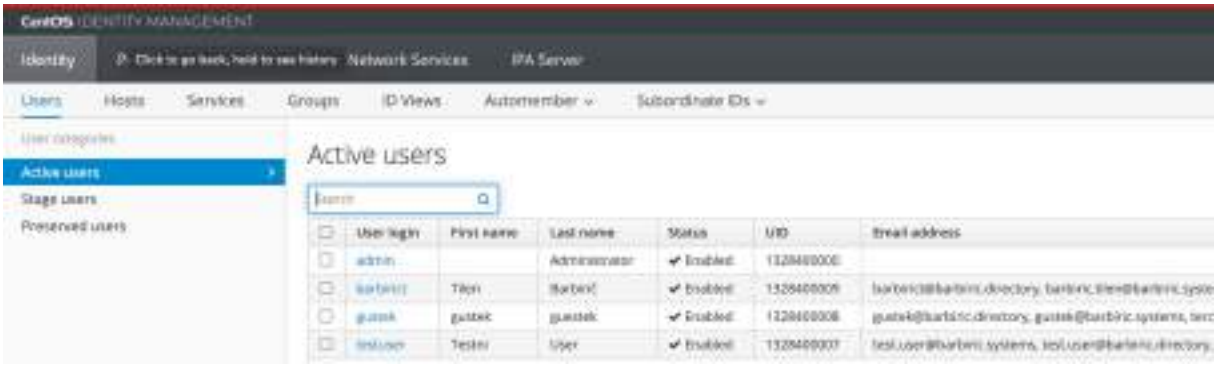
1. Prijava v sistem preko spletnega vmesnika:



Slika 27 Prijavna stran sistema FreeIPA

Vir: Lasten vir

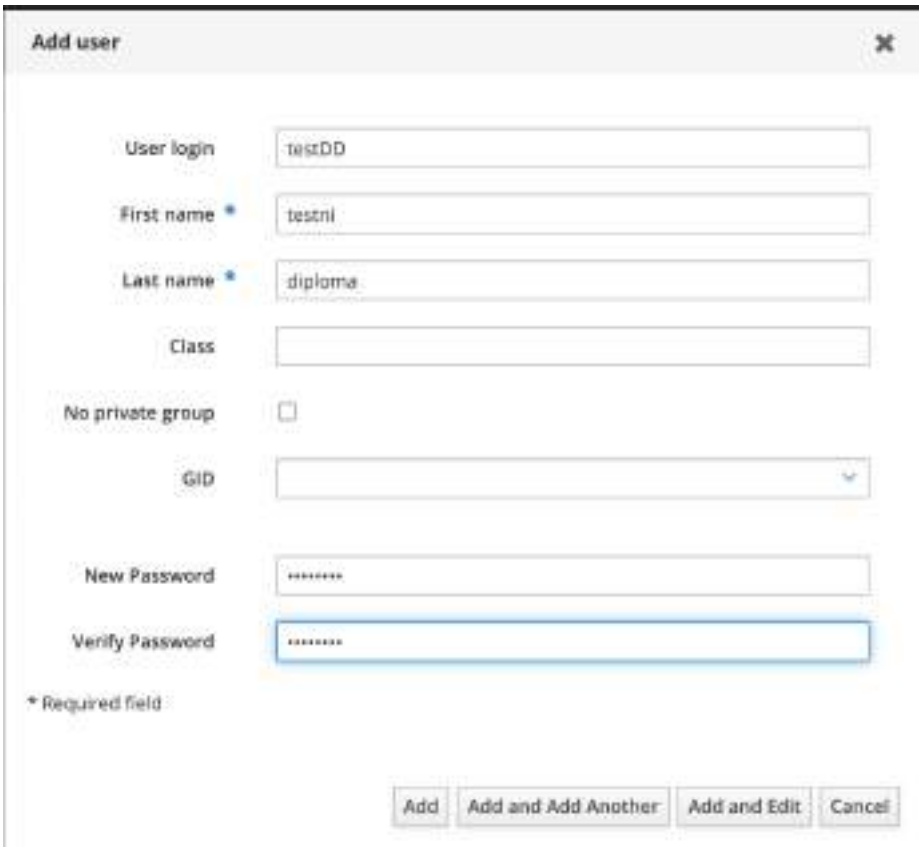
2. Po prijavi nas pričaka stran z uporabniki v sistemu. Z klikom na gumb »Add« pričnemo z dodajanjem novega uporabnika.



Slika 28 Seznam uporabnikov v sistemu *FreeIPA*

Vir: Lasten vir

3. Po kliku se nam odpre okno za vnos podatkov o uporabniku. V našem primeru bomo vnesli uporabniško ime kot *testDD*, ime (testni), priimek (diploma) ter nastavili geslo. Ko to naredimo stisnemo gumb »Add end Edit« za dodatno urejanje uporabnika.



Slika 29 Okno za dodajanje uporabnika v sistem FreeIPA

Vir: Lasten vir

4. Odpre se nam okno z vsemi možnimi konfiguracijskimi lastnostmi uporabnika. Naš poštni strežnik bo upravljal z dvema domenama – *barbiric.directory* bo interna domena, za interno pošto, *barbiric.systems* pa domena, ki bo uporabljena tudi za zunanjo pošto. E-naslov za interno pošto nam je sistem FreeIPA pripravil že sam, mi pa dodamo le še *alias* za zunanjo pošto – lahko bi dodali tudi več različnih e-poštnih naslovov in uporabnik bi lahko pošiljal/prejemal pošto iz vseh vnesenih.



Slika 30 Dodajanje dodatnega e-poštnega naslova za uporabnika v sistemu FreeIPA

Vir: Lasten vir

Konfiguracija in dodajanje uporabnika v sistemu *FreeIPA* je zelo preprosto in ponuja mnogo možnosti, ki jih lahko popolnoma prilagodimo lastnim potrebam.

4.6 Postopek vzpostavitve in konfiguracija

4.6.1 Z Docker Swarm

Za vzpostavitev poštnega strežnika in pomožnih sistemov smo uporabili kombinacijo datotek *docker compose* in ukazov *docker*, vse skupaj pa je bilo vzpostavljeno z orodjem *Ansible*. Poštni strežnik in vsi odvisni servisi (*minIO*, *YugabyteDB*) so bili vzpostavljeni z datoteko *mailStack.yaml*, obzervacijsko-zbirateljska orodja *Grafana*, *Loki*, *Promtail*, *Prometheus* in *node_exporter* pa z datoteko *monitoring,yaml*.

4.6.1.1 Omrežje mail-network

Omrežje *mail-network* je interno *docker* omrežje, po katerem med sabo komunicirajo naši servisi. Kontejnerjem v gruči omogoča medsebojno komunikacijo, izolirano od širšega lokalnega omrežja, prav tako pa omogoča tudi DNS resolucijo.



Slika 31 *Ansible* naloga za ustvarjanje omrežja *mail-network*

Vir: lasten vir

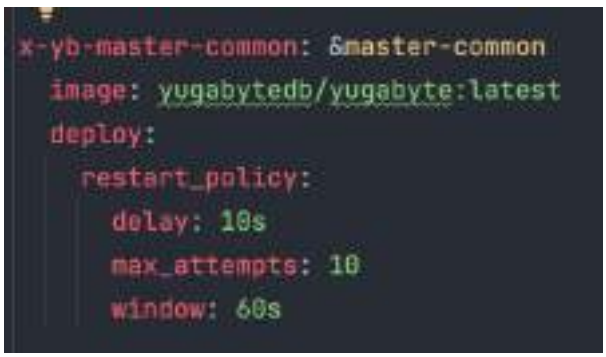
Za podomrežje smo uporabili masko/25, torej 128 uporabnih naslovov, kar je več kot dovolj za naše servise. Lahko bi uporabili tudi manjšo masko/26, ampak smo raje zaradi možnosti povečanja števila vozlišč v prihodnosti izbrali večje omrežje.

Gonilnik »overlay« smo izbrali, ker omogoča *load balancing* – če npr. servis A teče na vozlišču 1 in pride zahtevek na vozlišče 2, ga bo avtomatsko posredovalo vozlišču 1.

4.6.1.2 Servisi

4.6.1.2.1 X-Common

Ker imamo v naši *compose* datoteki veliko servisov z relativno podobno konfiguracijo (v veliko primerih gre za replike z minimalnimi razlikami), bi bilo nesmiselno večkrat pisati iste konfiguracije. Na srečo obstaja atribut »razširi« (angl. *extend*), ki omogoča, da skupno konfiguracijo napišemo le enkrat in jo uporabimo za eno ali več replik.



Slika 32 Primer »skupne« konfiguracije za *yb-master* kontejner

Vir: Lasten vir

4.6.1.2.2 Podatkovna baza yugabyteDB

Za podatkovno bazo *YugabyteDB* bomo uporabili 3 kontejnerje v vlogi vladarjev (angl. *masters*) in tri v vlogi delavcev (angl. *workers*). Vladarji so imenovani *yb-masters*, delavci pa *yb-tservers (transaction servers).* Ker ima vsak kontejner rahlo drugačno konfiguracijo, žal ne moremo uporabiti funkcije replik sistema *swarm*.

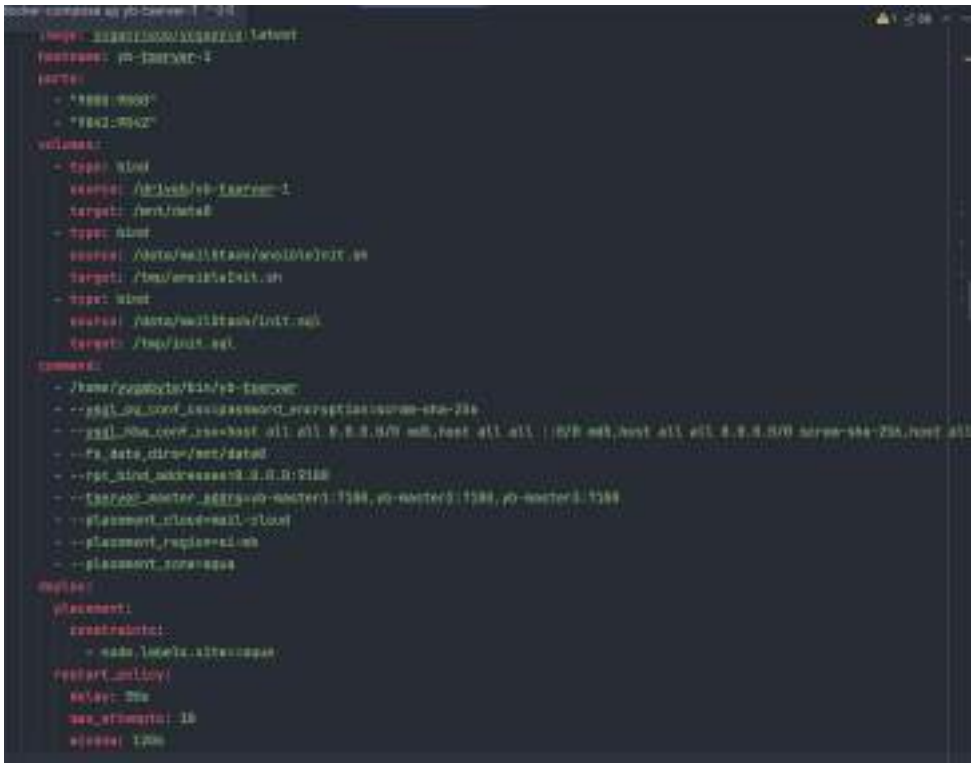


Slika 33 Konfiguracija za vladarja yb-master1

Vir: lasten vir

Pri vzpostavitvi vladarjev je bil najpomembnejši del pravilna konfiguracij polja *command*; pri zagonskem ukazu je bilo treba vnesti:

- Naslove vseh vladarjev v gruči (*master_addresses*) – uporabili smo imena gostiteljev (angl. *hostname*) in vrata 7100.
- IP naslovi, na katerih naj vladar posluša (*rpc_bind_addresses*), v našem primeru smo nastavili naj posluša na vseh možnih IP naslovih z vrati 7100 (0.0.0.0:7100).
- Podatki o poziciji (*placement_cloud*, *placement_region* in *placement_zone*) – ker želimo enakomerno replikacijo čez vsa tri vozlišča. Pri vseh treh smo nastavili *placement_cloud* na *mail-cloud*, *placement_region* na *si-mb*, razlikujejo se le po vrednosti v polju *placement_zone* (vrednosti so: *aqua*, *benneton* in *dawn*).



Slika 34 Konfiguracija za delavca *yb-tserver-1*

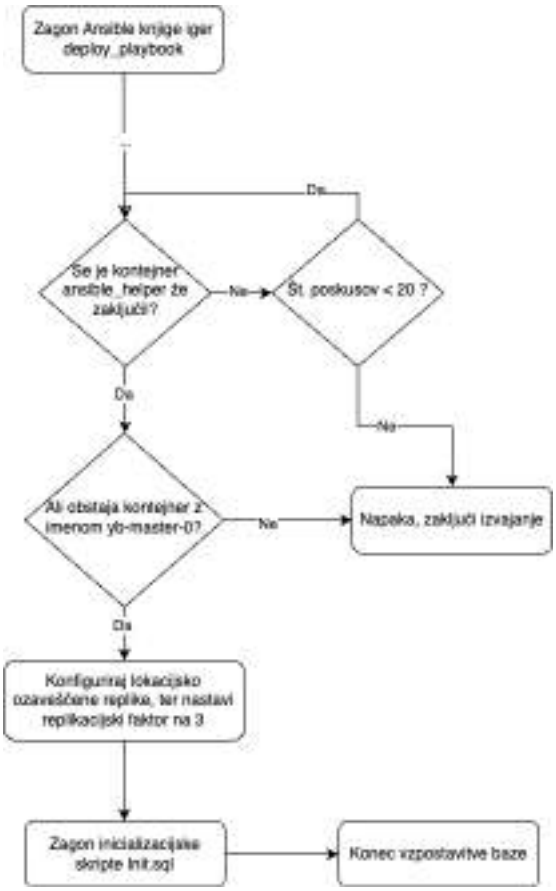
Vir: lasten vir

Delavcem je bilo potrebno vpeti datoteke za inicializacijo baz in uporabnikov, ki jih bomo potrebovali za ostale servise (*ansibleInit.sh* in *init.sql*).

Ponovno pa smo morali biti pozorni na polja in vrednosti v zagonskem ukazu. Izpostavil bi naslednja polja:

- Način šifriranja gesel (*ysql_pg_conf_csv*) – tukaj smo nastavili algoritem za šifriranje gesel (*scram-sha-256*).
- IP naslovi, na katerih naj delavec posluša (*rpc_bind_addresses*), v našem primeru smo nastavili naj posluša na vseh možnih IP naslovih z vrati 9100 (0.0.0.0:9100).
- Podatki o naslovih vladarjev *tserver_master_addrs* (identično kot pri vladarjih za polje *master_addresses*).
- Podatki o poziciji (identično kot pri vladarjih).

4.6.1.2.2.1 Postopek vzpostavitve



Slika 35 Diagram poteka vzpostavitve *yugabyteDB* v sistemu *Docker Swarm*

Vir: Lasten vir

Vzpostavitev se začne z zagonom knjige iger *deploy_playbook*. V procesu vzpostavitve se med drugim ustvari tudi kontejner *ansible_helper*, ki preverja, če so servisi (med njimi tudi *yugabyteDB* vladarji in delavci) pripravljeni (v knjigi je definiranih 20 poizkusov s 30 sekundnimi razmaki.) Če se *ansible_helper* uspešno zaključi, poiščemo na vozlišču Aqua kontejner z imenom »yb-master-0« in če je le-ta najden, izvedemo ukaz za nastavitev lokacijsko ozaveščenih replik ter nastavimo replikacijski faktor na število vozlišč (v našem primeru 3). V primeru, da se *ansible_helper* ne zaključi ali kontejner »yb-master-0« ni najden, pa knjiga prekine delovanje in vrže napako.

```
ALTER USER yugabyte WITH ENCRYPTED PASSWORD 'password';
CREATE DATABASE stalwartdb;
CREATE USER stalwartu WITH ENCRYPTED PASSWORD 'password';
GRANT ALL PRIVILEGES ON DATABASE stalwartdb TO stalwartu;

CREATE DATABASE grafanadb;
CREATE USER grafanau WITH ENCRYPTED PASSWORD 'password';
GRANT ALL PRIVILEGES ON DATABASE grafanadb to grafanau;
```

Slika 36 Skripta *Init.sql*

Vir: Lasten vir

Izziv pri vzpostavitvi je bil ravno zagon skripte *Init.sh* v kontejnerju, saj Ansible privzeto ne more vnašati gesel pred izvedbo skripte, zato smo morali napisati skripto *ansibleInit.sh*, ki v kontejnerju namesti modul *expect*. Vloga modula *expect* je, da pričakuje zahtevo za vnos gesla in ko se le-to zgodi, vnese geslo, kar omogoča, da *ansible* potem zažene konfiguracijski ukaz.

```
#!/usr/bin/env expect
set timeout -1
set password [lindex $argv 0]
set script [lindex $argv 1]

spawn ysqlsh -U yugabyte -f $script
expect "Password for user yugabyte:"
send "$password\r"
expect eof
```

Slika 37 Skripta *ansibleInit.sh*

Vir: Lasten vir

4.6.1.2.3 Sistem za shranjevanje podatkov minIO

```
minio1:
  <<: *minio-common
  hostname: minio1
  configs:
    - source: minio-stalwart-policy
      target: /tmp/sBucketPolicy.json
    - source: minio-loki-policy
      target: /tmp/lBucketPolicy.json
  deploy:
    placement:
      constraints:
        - node.labels.site==aqua
  volumes:
    - /driveb/minio1:/data
```

Slika 38 Skupna konfiguracija kontejnerjev *minIO*

Vir: Lasten vir

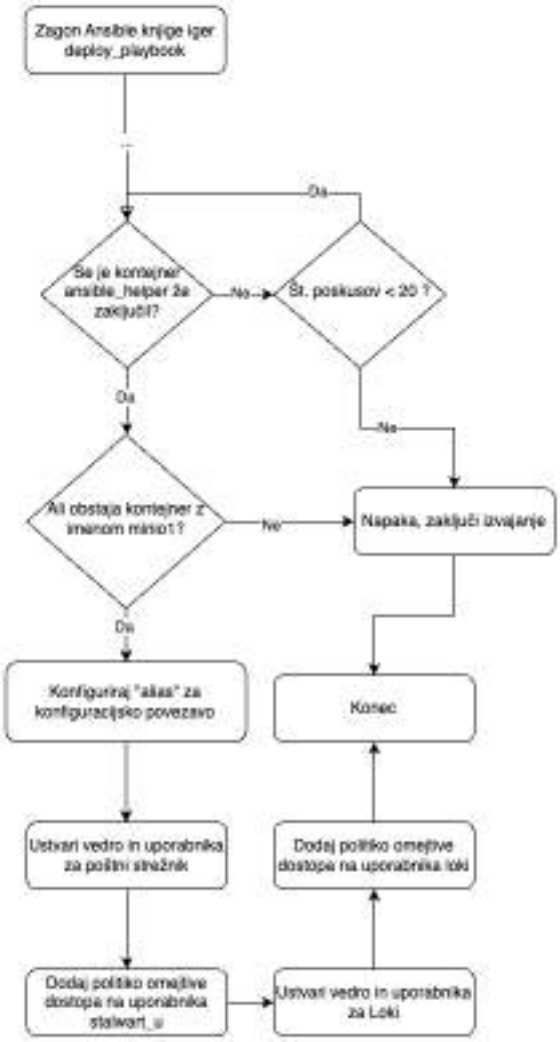
Za vzpostavitev shrambe *minIO* smo uporabili 3 kontejnerje. Pri konfiguraciji je bilo ponovno potrebno paziti pri polju *command*, s katerim instanci povemo, na katerem naslovu naj posluša konzola za administracijo – v našem primeru smo kot naslov nastavili ime gostitelja *minIO{1…3}:9001. MinIO* v *docker* kontejnerju omogoča tudi, da podatke za administratorja prebere iz datoteke, kar je v našem primeru odlično, saj uporabljamo skrivnosti *Swarm*, ki skrivnosti v kontejner priklopijo kot datoteke. Nastavili smo tudi preverjanje zdravja (angl. *health check*), ki preverja, kdaj je kontejner uspešno vzpostavljen.



Slika 39 Konfiguracija kontejnerja *minIO*

Vir: Lasten vir

4.6.1.2.3.1 Postopek vzpostavitve



Slika 40 Diagram poteka vzpostavitve servisa *MinIO* v sistemu *Docker Swarm*

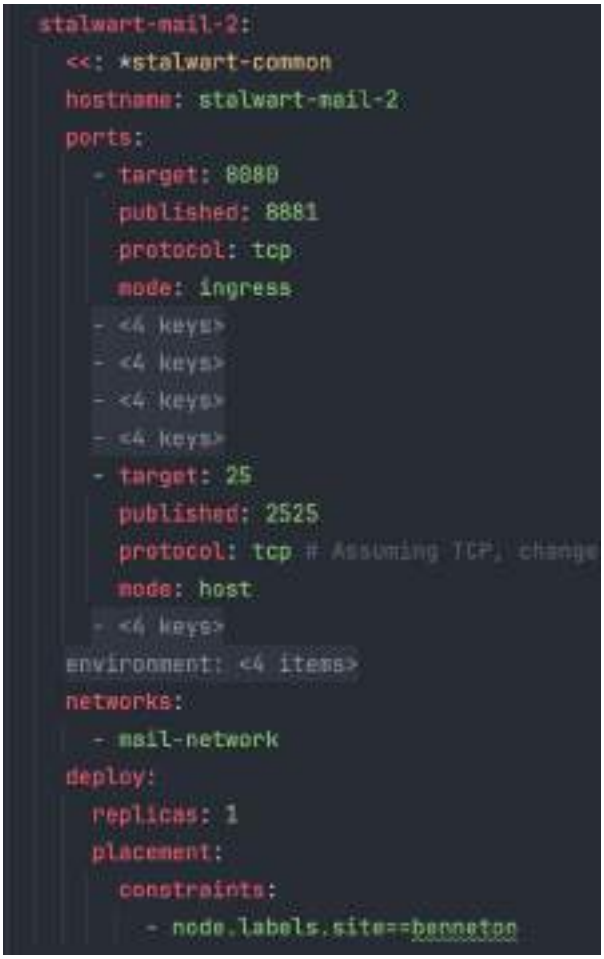
Vir: Lasten vir

Vzpostavitev se začne z zagonom knjige iger *deploy_playbook*. V procesu vzpostavitve se med drugim ustvari tudi kontejner *ansible_helper*, ki preverja, če so servisi (med njimi tudi *minIO*) pripravljeni (v knjigi je definiranih 20 poizkusov z 30 sekundnimi razmaki.). Če je izvedba uspešna, preverimo obstoj kontejnerja z imenom »minio1«, sledi:

1. konfiguracija vzdevka (angl. *alias)* s komando *mc admin user add local {{ minIO.username }} {{ minIO.password }}* kjer sta *minIO.username* in *minIO.password* podatka za dostop administratorskega uporabnika. *Alias* moramo nastaviti zaradi potreb CLI orodja »mc«, ki se uporablja za konzolno upravljanje sistema *minIO*.

2. Ustvarjanje vedra in uporabnika za poštni strežnik *Stalwart*, z ukazoma *mc mb local/stalwart-bucket* in *mc admin user add local {{ minIO.username }} {{ minIO.password }}.*
3. Ustvarjanje in dodajanje politike omejitve dostopa na uporabnika *stalwart* z ukazoma *mc admin policy create local stalwart-bucket /tmp/sBucketPolicy.json* in *mc admin policy attach local stalwart-bucket --user {{ minIO.username }}.*
4. Ustvarjanje vedra in uporabnika za zbiralnik dnevniških datotek *Loki* z ukazoma *mc mb local/stalwart-bucket* in *mc admin user add local {{ loki.username }} {{ loki.password }}.*
5. Ustvarjanje in dodajanje politike omejitve dostopa na uporabnika *stalwart* z ukazoma *mc admin policy create local stalwart-bucket /tmp/lBucketPolicy.json* in *mc admin policy attach local stalwart-bucket --user {{ loki.username }}.*

4.6.1.2.4 Poštni strežnik Stalwart



Slika 41 Konfiguracija za enega izmed kontejnerjev poštnega strežnika *Stalwart*

Vir: lasten vir

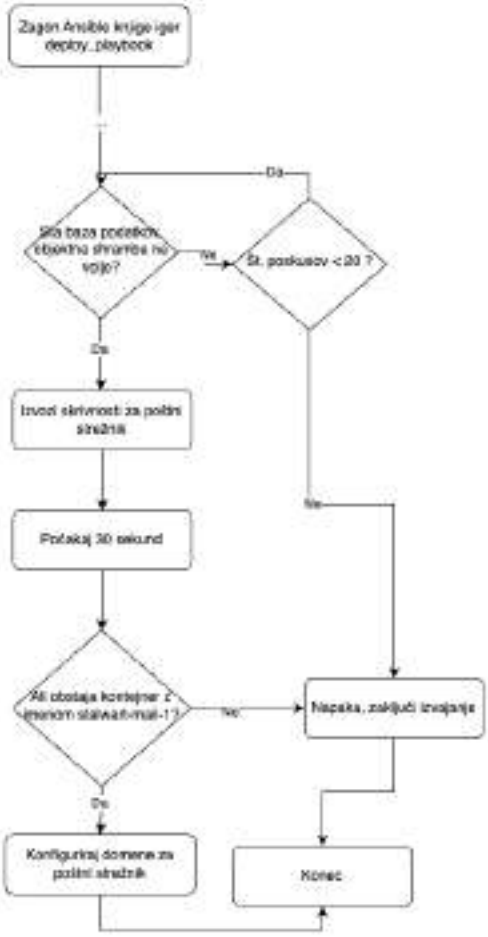


Slika 42 Skupna konfiguracija za kontejnerje poštnega strežnika *Stalwart*

Vir: lasten vir

Poštni strežnik *Stalwart* ima v gruči *swarm* tri kontejnerje. Ker smo v kontejnerje vpeli skrivnosti, je bilo potrebno uporabiti skripto za izvoz skrivnosti. Vsak kontejner izpostavlja vrata za SMTP, SMTPS, IMAP, IMAPS, HTTP in HTTPS protokole. HTTP in HTTPS uporabljata omrežni način gruče, ki avtomatsko porazdeli obremenitev med vse tri kontejnerje, ostali protokoli pa uporabljajo gostiteljski omrežni način zaradi potrebe delovanja *proxy* protokola (*proxy* protokol omogoča, da kontejner vidi izvorni IP naslov vsake povezave, čeprav je ta posredovana preko zunanjega izenačevalca obremenitve).

4.6.1.2.4.1 Postopek vzpostavitve



Slika 43 Diagram poteka za vzpostavitev poštnega strežnika v *Docker Swarm*

Vir: lasten vir

Preden se kontejner uspešno zažene, skripta preveri, ali sta shramba *MinIO* in baza *YugabyteDB* že uspešno zagnani. To funkcionalnost sem dodal, da se kontejner ne bi prevečkrat in nepotrebno ponovno zaganjal. Ko skripta zazna, da so vse odvisnosti pripravljene, izvozi skrivnosti iz datotek v okoljske spremenljivke. Knjiga iger na to počaka 30 sekund (približen čas, ki ga kontejner porabi za prvo konfiguracijo), na to pa še ustvari domene, ki jih bomo uporabljali.

4.6.1.2.5 Grafana in zbiralniki

Vzpostavitev servisov za opazovanje našega sistema je v primerjavi z ostalimi servisi relativno preprosta, saj ne potrebujejo dodatne konfiguracije po koncu vzpostavitve.

4.6.1.2.5.1 Grafana



Slika 44 Konfiguracijska datoteka za kontejner *Grafana* v sistemu *Docker Swarm*

Vir: lasten vir

Pri vzpostavitvi *Grafane* smo zaradi že prej omenjenih omejitev uporabili le en kontejner in izklopili zaklepanje migracije z okoljsko spremenljivko *GF_DATABASE_MIGRATION_LOCKING=FALSE*. Kontejnerska slika podpira branje skrivnosti iz datotek, kar nam zelo olajša sam proces inicializacije (skripta za izvoz skrivnosti v okoljske spremenljivke ni potrebna). Skrivnosti, ki smo jih vpeli, so podatki za administratorskega uporabnika ter podatki uporabnika za povezavo z podatkovno bazo, saj bo *Grafana* v naši gruči tam hranila vso konfiguracijo. Do *Grafane* lahko na lokalnem omrežju dostopamo preko kateregakoli vozlišča na vratih 3000.

4.6.1.2.5.2 Loki

```
  loki:
    image: grafana/loki:latest
    ports:
      - "3100:3100"
    deploy:
      restart_policy:
        delay: 30s
        max_attempts: 10
        window: 120s
      mode: global

    configs:
      - source: loki-config
        target: /etc/loki/local-config.yaml
    secrets:
        - loki-s3-bucket-url
    volumes:
      - /data/monitoring/secretsToENV.sh:/tmp/secretsToENV.sh
      - /drive0/monitoring/loki:/loki
    entrypoint: /bin/sh -c "/tmp/secretsToENV.sh --service=Loki"
    networks:
      - mail-network
```

Slika 45 Konfiguracijska datoteka za kontejner *Loki* v sistemu *Docker Swarm*

Vir: lasten vir

Zbiralniku dnevniških datotek *Loki* vpnemo skrivnost za povezavo in dostop do objektne shrambe: *http://<UPORABNIK>:<GESLO>@minIO:<PORT>/<IME_VEDRA>.* Ker pa kontejnerska slika za *Loki* ne podpira branja skrivnosti iz datotek, je bilo potrebno napisati skripto, ki datoteko izvozi v okoljsko spremenljivko.

```
#!/bin/sh

for arg in "$@"
do
  case $arg in
    --services=*)
      SERVICE="${arg#*=}"
      shift
      ;;
  esac
done


if [ "$SERVICE" == 'loki' ]; then
      export LOKI_S3_BUCKET_URL

      LOKI_S3_BUCKET_URL=$(cat /run/secrets/loki-s3-bucket-url)

      if [ "${#LOKI_S3_BUCKET_URL}" -gt 0 ]; then
        echo "Successfully exported LOKI_S3_BUCKET_URL"
        echo "$LOKI_S3_BUCKET_URL"
      else
        echo "Failed to export LOKI_S3_BUCKET_URL"
        exit 125
      fi
    exec /usr/bin/loki -config.file=/etc/loki/local-config.yaml -config.expand-env
else
    echo "Invalid service"
    exit 125
fi
```

Slika 46 Skripta za izvoz skrivnosti v okoljske spremenljivke

Vir: lasten vir

Ta skripta se zažene pred samim servisom, tako da je spremenljivka na voljo takoj po zagonu.

```
auth_enabled: false

server:
  http_listen_port: 3100

frontend:
  address: 127.0.0.1

common:
  replication_factor: 3

schema_config:
  configs:
    - from: "2024-01-01"
      store: tsdb
      index:
        prefix: loki_index_
        period: 24h
      object_store: s3
      schema: v13


storage_config:
  s3:
    s3: ${LOKI_S3_BUCKET_URL}
    s3ForcePathStyle: true
```

Slika 47 Konfiguracijska datoteka *config.yaml* za *Loki* (*Docker Swarm*)

Vir: lasten vir

V konfiguracijski datoteki *local-config.yaml* smo nastavili tudi čas ohranitve – 24 h.

4.6.1.2.5.3 Node Exporter in Prometheus



Slika 48 Konfiguracijska datoteka za kontejner *Prometheus* v sistemu *Docker Swarm*

Vir: lasten vir



Slika 49 Konfiguracijska datoteka za kontejnerje *node_exporter* v sistemu *Docker Swarm*

Vir: lasten vir

Do *Prometheusa* lahko iz lokalnega omrežja dostopamo na vratih 9090. Na vsakem vozlišču za zagotavljanje visoke razpoložljivosti teče ena replika. V konfiguraciji smo nastavili, da »strga« podatke vsakih 15 sekund.

Node_exporter teče na vsakem vozlišču gruče, dostopen je pa tudi preko lokalnega omrežja na vratih 9100 za potrebe *Prometheus* statične konfiguracije. V kontejner smo vpeli tudi poti

našega operacijskega sistema, kjer se nahajajo metrike o delovanju. Če te poti ne bi vpeli, bi *node_exporter* bral le svoje kontejnerske metrike, ki pa niso enako reprezentativne kot metrike celotnega sistema.

4.6.2 Z Kubernetes

V gruči *Kubernetes* sem želel pri vzpostavitvi poštnega strežnika in ostalih servisov prikazati čim več različnih načinov. Za sam proces postavitve servisov sem uporabil tako *Kubernetes API* in orodje Helm. Prav tako sem uporabil več načinov za ustvarjanje konfiguracij *(ConfigMap)* in skrivnosti – pri določenih servisih sem skrivnosti in konfiguracije ustvaril pred samo postavitvijo servisa, pri ostalih pa sem skrivnosti definiral v Helm datoteki *values.yaml* in pustil, da skrivnosti ustvari orodje Helm.

4.6.2.1 Longhorn



Slika 50 Ukazi za vzpostavitev shrambe *Longhorn*

Vir: lasten vir

Zgornja slika prikazuje ukaze, ki smo jih uporabili za namestitev in konfiguracijo. Vzpostavitev se je pričela z prenosom konfiguracijske datoteke na vozlišče, nato smo z ukazom *kubectl apply* namestili *Longhorn* v gručo. Pri tem se samodejno ustvari novi imenski prostor *longhorn-system*, vsi potrebni servisi, stroki, kontejnerji in konfiguracije za trajne nosilce. Privzete nastavitve ob ustvarjanju trajnega nosilca ustvarijo tri replike, ki pa ni nujno, da so na istem vozlišču kot strok, ki ga uporablja. Želena konfiguracija v izdelku ima le eno repliko za vsak strok, hramba lete pa mora biti na istem vozlišču. Konfiguracijo smo izvedli z ukazom *kubectl edit configmap*, ki nam v urejevalniku besedila *vim* omogoči vizualno urejanje konfiguracije.



Slika 51 Urejanje konfiguracije *ConfigMap* za *Longhorn*

Vir: lasten vir

Vrednosti, ki smo jih v konfiguracijski datoteki spremenili, sta:

- *numberOfReplicas: 1* – željeno število replik smo nastavili na 1.
- *dataLocality: »best-effort«* – trajni nosilec bo ustvarjen na istem vozlišču kot strok, ki ga bo uporabljal.

4.6.2.2 Imenski prostori

Vsi servisi v gruči *Kubernetes* tečejo v ločenih imenskih prostorih. Izjema so le *Grafana, Loki, Promtail, Prometheus* in *Node_exporter.*



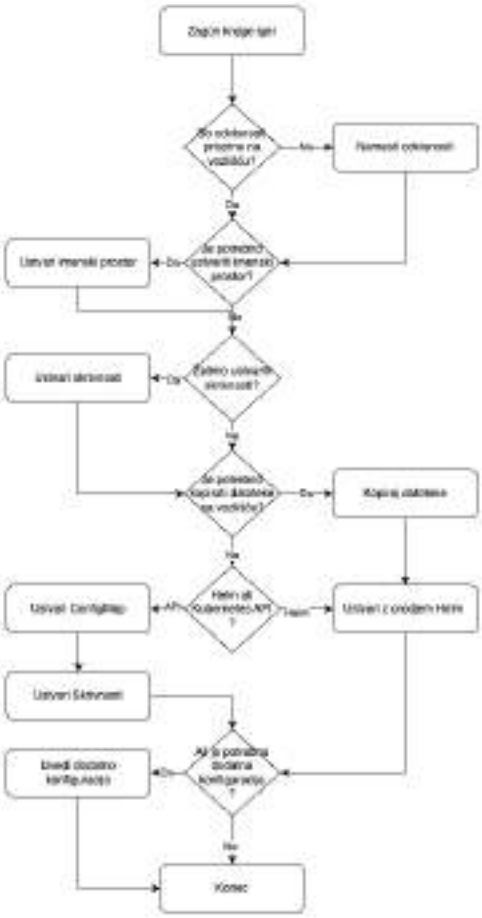
Slika 52 Primer ustvarjanja imenskega prostora v sistemu *Kubernetes*

Vir: lasten vir

Zgoraj je prikazana *Ansible* naloga za ustvarjanje imenskega prostora. Naloga iz spremenljivke prebere željeno ime imenskega prostora in ga ustvari v gruči. Tak način je uporabljen za ustvarjanje vseh imenskih prostorov v gruči.

4.6.2.3 Servisi

Postopek postavitve vseh servisov v gruči *Kubernetes* lahko opišemo s spodnjim diagramom:



Slika 53 Diagram poteka, ki prikazuje vzpostavitev servisov v sistemu *Kubernetes*

Vir: lasten vir

4.6.2.3.1 Podatkovna baza yugabyteDB

Za vzpostavitev podatkovne baze *yugabyteDB* smo uporabili tri imenske prostore z ločenimi konfiguracijami. V vsakem imenskem prostoru teče en *vladar* in en *delavec*. Gre za enako implementacijo kot v gruči *Swarm*, ker če bi v eno datoteko z vrednostmi vnesli, da želimo 3 vladarje in 3 delavce, bi bili prepoznani, kot da tečejo na isti lokaciji, kar bi onemogočilo konfiguracijo lokacijsko ozaveščenih replik.



Slika 54 Diagram poteka za vzpostavitev in konfiguracijo *YugabyteDB* v sistemu *Kubernetes*

Vir: lasten vir

Knjiga iger najprej ustvari vse tri potrebne imenske prostore, nato v njih ustvari skrivnosti in nato v vsakem imenskem prostoru naredi namestitev podatkovne baze. Ko je namestitev končana, knjiga iger počaka, da je strok *yb-master-0* pripravljen, nakar sledi konfiguracija lokacijsko ozaveščenih replik in ustvarjanje potrebnih uporabnikov in zbirk podatkov, enako kot pri gruči *Swarm*.

4.6.2.3.2 Minio S3 sharmba objektov

Kljub temu da za shrambo v gruči *Kubernetes* uporabljamo Longhorn, je bilo še vedno treba za potrebe poštnega strežnika *Stalwart* in zbiratelja dnevniških datotek *Loki* vzpostaviti objektno shrambo *MinIO*. Na vsakem vozlišču v gruči teče ena *MinIO* instanca, do uporabniške vmesnika pa lahko iz lokalnega omrežja preko vseh vozlišč dostopamo na vratih, ki jih določi interni izenačevalnik obremenitve.

Vzpostavitev smo tudi tukaj izvedli z orodjem Helm. Izpostavil bi rad naslednje vrednosti v datoteki z vrednostmi:



Slika 55 Datoteka z vrednostmi za *MinIO* v sistemu *Kubernetes*

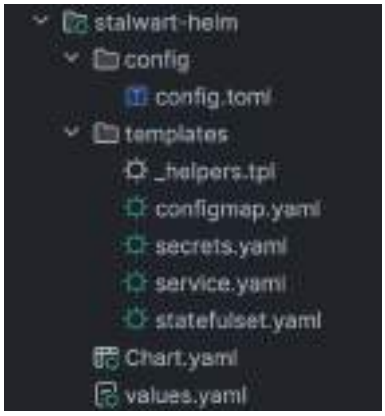
Vir: lasten vir

Ko vzpostavljamo *MinIO* s Helm diagramom, je vzpostavitev zelo preprosta, saj nam omogoča, da že vnaprej definiramo vsa vedra, uporabnike in politike dostopa, zato kasnejša konfiguracija ni potrebna. Orodje *H*Helm to omogoči z uporabo *Kubernetes* nalog (angl. *jobs*) *makeUserJob*, *makePolicyJob* in *makeBucketJob*. Edina skrivnost, ki je definirana pred vzpostavitvijo s Helm, je skrivnost z *minio-root-secret*, ki vsebuje podatke o administratorju (uporabniško ime in geslo). Naša knjiga iger more tako ustvariti le imenski prostor, eno skrivnost in inštalirati *MinIO* HHelm diagram z našo datoteko vrednosti.

4.6.2.3.3 Poštni strežnik Stalwart

Posebnost pri postavitvi poštnega Strežnika *Stalwart* v *Kubernetes* gruči je, da sem napisal svoj Helm diagram, saj sem želel inštalacijo popolnoma prilagoditi potrebam izdelka, prav tako pa izkoristiti polni potencial sistema *Kubernetes*.

4.6.2.3.3.1 Struktura HHelm diagrama



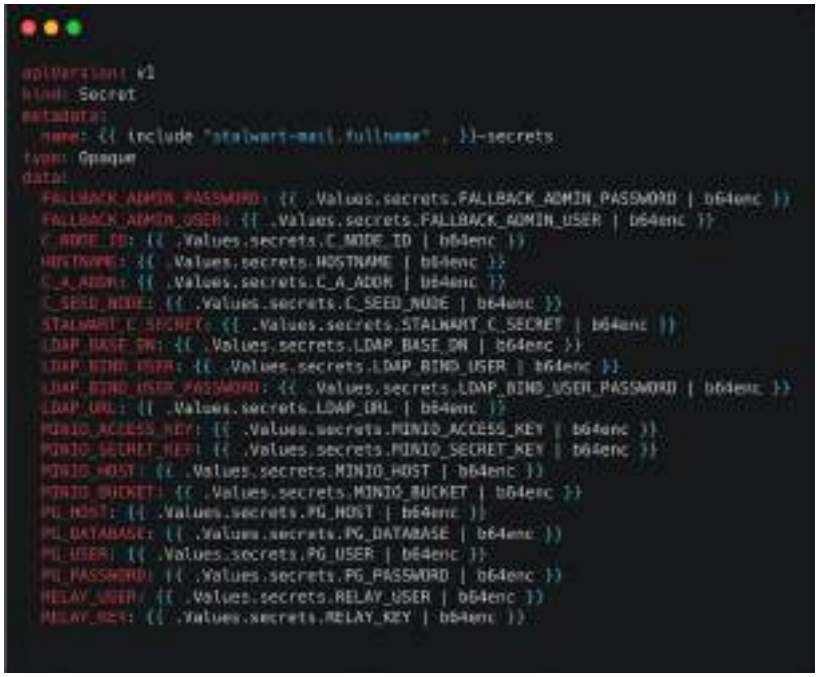
Slika 56 Struktura HHelm diagrama za poštni strežnik Stalwart

Vir: lasten vir

Kot je vidno iz zgornje slike, je naš diagram sestavljen iz naslednjih datotek:

- *config.toml* – konfiguracijska datoteka za poštni strežnik *Stalwart* (enaka datoteka kot pri sistemu *Docker Swarm*).
- *_helpers.tpl* – ta datoteka se uporablja za definiranje vrednosti, ki jih bomo v ostalih datotekah večkrat ponovno uporabili, kar nam prihrani veliko časa pri pisanju samega diagrama.
- *configmap.*yaml – *Kubernetes* konfiguracija, ki bo v stroke vpela konfiguracijsko datoteko *config.toml.*
- *secrets.yaml* – s to datoteko bomo ustvarili vse skrivnosti, ki so potrebne za delovanje našega poštnega strežnika. Vrednosti za skrivnosti so definirane v datoteki *values.yaml*.
- *service.*yaml – *Kubernetes* servis, preko katerega bomo dostopali do poštnega strežnika. V datoteki smo definirali dva servisa; prvi je za protokola HTTP in HTTPS, ki uporabljata omrežni način gruče (izvorni IP naslov v tem primeru pridobimo preko glave *X-FORWARDED-FOR)*, drugega pa za SMTP, SMTPS, IMAP in IMAPS – pri drugem servisu uporabljamo omrežni način gostitelja, saj želimo v kontejnerjih dostop do izvornih IP naslovov, ki jih pridobimo s *proxy* protokolom.

- *statefulset.yaml – Kubernetes statefulSet*. V njem smo definirali, da želimo 3 replike, ki pa naj bodo razporejene glede na območje posameznega vozlišča, s tem dosežemo enakomerno razporeditev strokov v gruči.
- *Chart.yaml* – datoteka vsebuje osnovne podatke o našem diagramu.
- *values.yaml* – privzete vrednosti za vzpostavitev HHelm diagrama.



Slika 57 Datoteka *secrets.yaml*

Vir: lasten vir



Slika 58 Datoteka *StatefulSet* za poštni strežnik *Stalwart*

Vir: lasten vir



Slika 59 Datoteka z vrednostmi za HHelm diagram poštnega strežnika *Stalwart*

Vir: lasten vir

Posebnost pri postopku vzpostavitve je, da čeprav uporabljamo orodje *H*Helm, je kljub temu potrebno na vozlišče prekopirati datoteke za HHelm diagram. Razlog za tem je, da naš HHelm diagram ni objavljen v nobenem repozitoriju in ga orodje HHelm ne bo našlo. Postopek torej pričnemo s kopiranjem datotek našega HHelm diagrama na eno vozlišče. Nato ustvarimo imenski prostor, sledi pa klasični način inštalacije s Helmin *Ansible*, le da v ukaz za inštalacijo namesto repozitorija in imena diagrama dodamo pot do mape z našim diagramom.

4.6.2.3.4 Grafana, loki in Prometheus

Tako *Grafana*, *Loki* in *Prometheus* so bili vsi vzpostavljeni z uporabo orodja Helm. Pri *Grafani* smo najprej vzpostavili skrivnosti za administratorske prijavne podatke ter za skrivnost s podatki za dostop do podatkovne baze. Enako kot pri vzpostavitvi v sistemu *Docker Swarm* smo definirali vire podatkov *datasources.yaml* in nastavili skrivnosti v konfiguracijsko datoteko *grafana.ini*, le da tega nismo delali v ločenih datotekah, ki bi jih bilo potrebno vpeti v kontejnerje, ampak smo vse to napisali v Helm datoteki z vrednostmi *values.yaml*

Zaradi boljše dokumentacije dnevniškega zbiralnika *Loki* za *Kubernetes* kot za sistem *Docker Swarm* mi je le uspelo vzpostaviti visoko razpoložljivo verzijo. V datoteki z vrednostmi smo definirali željeno število replik, pomembno pa je tudi bilo dodati skrivnost za povezavo s shrambo objektov *MinIO*. Razlika pri zbiratelju dnevnikov *Loki* je, da v sistemu *Kubernetes* ne obstaja gonilnika za beleženje kot pri sistemu *Docker Swarm*, zato je potrebno tukaj zraven vzpostaviti še pomočniški servis *promtail*. *Promtail* smo namestili ročno, saj nam je to vzelo res le nekaj sekund časa. *Promtail* smo namestili z ukazom Helm *install grafana/promtail promtail -n monitoring*. *Promtail* ne potrebuje nobene dodatne konfiguracije, sam zazna *Loki* v istem imenskem prostoru in mu začne avtomatsko pošiljati dnevniške datoteke vseh elementov gruče.

Tudi vzpostavitev *Prometheusa* in *node_exporterja* je bila izredno preprosta. V datoteki z vrednostmi smo omogočili *node_exporter* in v konfiguraciji *Prometheusa* nastavili le eno nalogo za *node_exporter*. *Prometheus* tudi tukaj metrike »strga« vsakih 15 sekund.

4.6.3 Konfiguracija poštnega strežnika

Oba sistema imata identično konfiguracijsko datoteko ter iste naloge, ki se izvedejo po vzpostavitvi. Konfiguracijo strežnika lahko izvedemo preko spletnega administracijskega vmesnika ali z konfiguracijsko datoteko. V mojem izdelku sem zaradi avtomatizacije in večje preglednosti izbral možnost z konfiguracijsko datoteko.

4.6.3.1 Konfiguracijska datoteka

Konfiguracijska datoteka poštnega strežnika Stalwart je v obliki zapisa *TOML (Tom's Obvious Minimal Language)*. Pomembnejše vrednosti v konfiguracijski datoteki (ključ – opis):

- *authentication.fallback-admin.** – Prijavni podatki administratorskega uporabnika za dostop do spletnega administracijskega vmesnika.
  - *username*: Uporabniško ime administratorja.
  - *secret:* Geslo administratorja.
- Samopodpisani TLS certifikat
  - Javni ključ
  - Privatni ključ
- *Directory.<ime>.** – Podatki o imeniškem strežniku, za avtentikacije, verifikacijo in iskanje uporabnikov in skupin.
  - *attributes:* Večina imeniških strežnikov uporablja različne vrste zapisov in ključev. Poštni strežnik potrebuje podatke o imenih določenih ključev, da ve, kako in kje poiskati uporabnike.
    - *e-mail:* Primarni e-mail naslov uporabnika.
    - *e-mail-alias:* Ostali uporabnikovi e-naslovi, iz katerih bo dovoljeno pošiljanje in prejemanje e-pošte. Vsi e-poštni naslovi uporabnika bodo imeli skupni poštni nabiralnik.
    - *groups:* Podatki o uporabnikovem članstvu v uporabniških skupinah.
    - *name:* Ključ, v katerem se nahaja uporabnikovo uporabniško ime.
    - *secret:* Ključ, ki hrani uporabnikovo geslo.

- *base-dn:* Začetna pot v imeniku, po kateri naj poštni strežnik začne iskanje uporabnikov v imeniku.
- *bind-dn:* Uporabnik, ki se naj uporablja za iskanje po imeniku.
- *url:* Domensko ime ali IP naslov imeniškega strežnika.

- *storage.*:*
- *blob:* Kje bodo hranjeni veliki binarni objekti?
- *data:* Kje bo hranjena konfiguracija?
- *directory:* Kje se bodo hranili uporabniki?

- *store.*:*
- *minio: Podatki za povezavo z objektno shrambo MinIO.*
- *postgres: Podatki za povezavo z podatkovno bazo.*

Občutljive vrednosti, ki jih hranimo v skrivnostih in se ob zagonu kontejnerja vpnejo kot okoljske spremenljivke v konfiguracijsko datoteko, vnesemo kot vrednost z imenom: *%{env:IME_OKOLJSKE_SPREMENLJIVKE}*. To omogoča večjo varnost pri hranjenju konfiguracijske datoteke.

Konfiguracijska datoteka je najbolj pomembna le ob prvi vzpostavitvi strežnika, saj ko strežnik zazna, da ima na voljo možnost shrambe podatkov v podatkovni bazi, konfiguracijo avtomatsko hrani v njej, razen za vrednosti definirane pod ključem *config.local-keys*. To pomeni, da če naredimo spremembo na enem vozlišču, se bo ta uveljavila tudi na ostalih.

4.6.3.2 Ustvarjanje lokalnih domen

Poštni strežnik v naši konfiguraciji na imeniškem strežniku avtomatsko poišče domene za namene avtentikacije in prevajanja, če je domena lokalna, vendar je za potrebe pravilne konfiguracije po končanem procesu vzpostavitve potrebno vseeno na strežniku ustvariti vnose za e-poštne domene, ki bodo v uporabi. Namen tega je, da nam strežnik ustvari DNS zapise ter certifikate za podpisovanje poštnih sporočil. To lahko naredimo na dva načina:

1. Preko spletnega uporabniškega vmesnika – ta način je najpreprostejši, saj vse, kar je potrebno narediti je, da kliknemo na gumb »Add domain« in vnesemo ime domene. Strežnik nam potem pripravi vse zapise in certifikate.



Slika 60 Dodajanje domene preko spletnega vmesnika

Vir: lasten vir

2. Z orodjem *stalwart-cli* preko ukazne vrstice – z ukazom *stalwart-cli -u http://localhost:8080 -c <uporabnik:geslo> domain create <ime domene>*. Orodje v ukazni vrstici nam v našem primeru omogoča, da ta korak avtomatiziramo z *Ansible* knjigo iger, ampak žal se pri dodajanju nove domene preko ukazne vrstice ne ustvarijo certifikati za podpisovanje, tako da je ta način primeren le za testiranje in ne za normalno uporabo.

4.6.3.3 Zagotavljanje varnosti poštnega strežnika

Pri gostovanju poštnega strežnika je izredno pomembno, da ga primerno zavarujemo, saj obstaja velik potencial za zlorabo.

4.6.3.3.1 DMARC, DKIM in SPF

DMARC *(Domain-based Message Authentication, Reporting & Conformance)*, DKIM *(DomainKeys Identified Mail)* in SPF *(Sender Policy Framework)* so e-poštne avtentikacije metode, s katerimi domene zaščitimo pred nepooblaščenim pošiljanjem e-poštnih sporočil. (Cloudflare, Inc., n.d.)

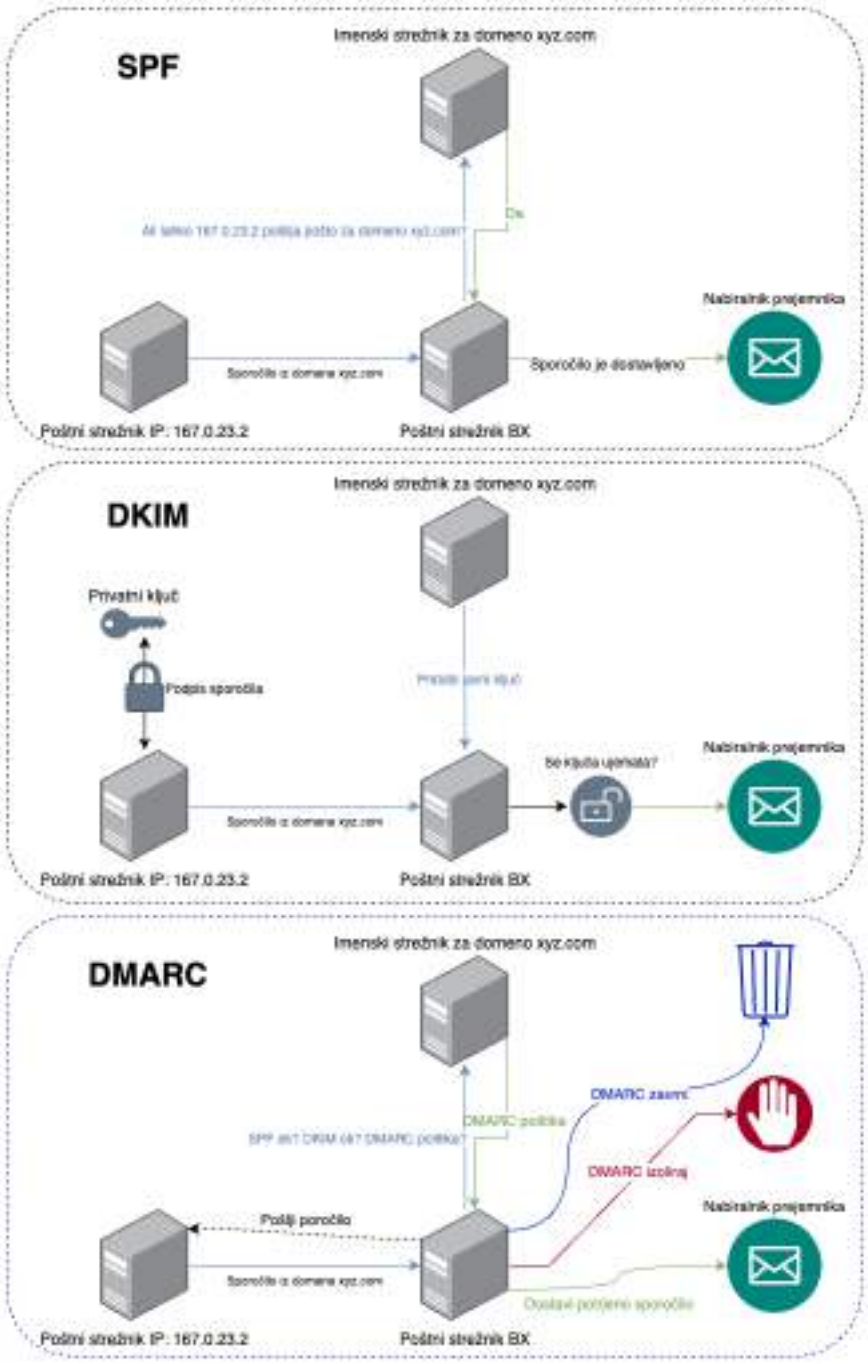
Vse tri načine avtentikacije vzpostavimo tako, da dodamo v imenski strežnik pravilne DNS zapise tipa TXT:

- *SPF* – vnesemo seznam IP naslovov, ki jim dovolimo da pošiljajo pošto za našo domeno.
- *DKIM* – zapis vsebuje javni ključ, s katerim bodo naša sporočila podpisana.
- *DMARC* – v tem zapisu definiramo naslov za pošiljanje poročil in kaj želimo, da poštni strežniki naredijo, ko so SPF in/ali DKIM pregledi neuspešni:

a) *Quarantine* – sporočila bodo označena kot neželena pošta;

b) *Reject* – sporočila bodo zavrnjena in ne bodo dostavljena;

c) *Deliver* – sporočila bodo vseeno dostavljena.



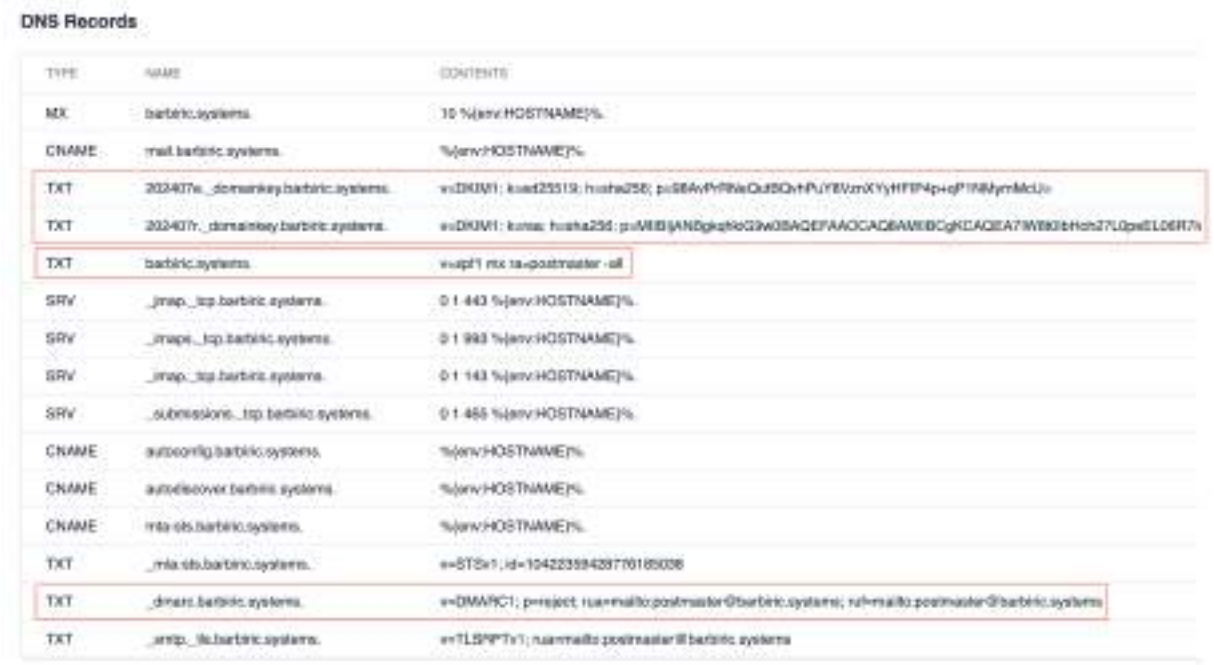
Slika 61 Diagram SPF, DKIM in DMARC

Vir: lasten vir

Iz zgornje slike lahko razberemo delovanje avtentikacijskih metod:

- *SPF* – poštni strežnik, ki prejme sporočilo preveri, če je pošiljateljev IP naslov med dovoljenimi za to domeno in če preverjanje uspe dostavi sporočilo.
- *DKIM* – pošiljatelj sporočilo podpiše z zasebnim ključem. Strežnik, ki sporočilo prejme v DNS zapisu, prebere javni ključ za domeno in če se ujemata, se sporočilo dostavi.

- *DMARC* – strežnik, ki prejme sporočilo, naredi pregleda *SPF* in *DKIM* ter iz DNS zapisa prebere *DMARC* politiko. Če sta uspešna, sporočilo dostavi, če sta ne uspešna, pa sporočilo zavrne ali izolira (označi kot neželeno), odvisno od določene *DMARC* politike. Če je v politiki za domeno določen tudi naslov za pošiljanje poročil, pa strežnik na ta naslov pošlje še poročilo.



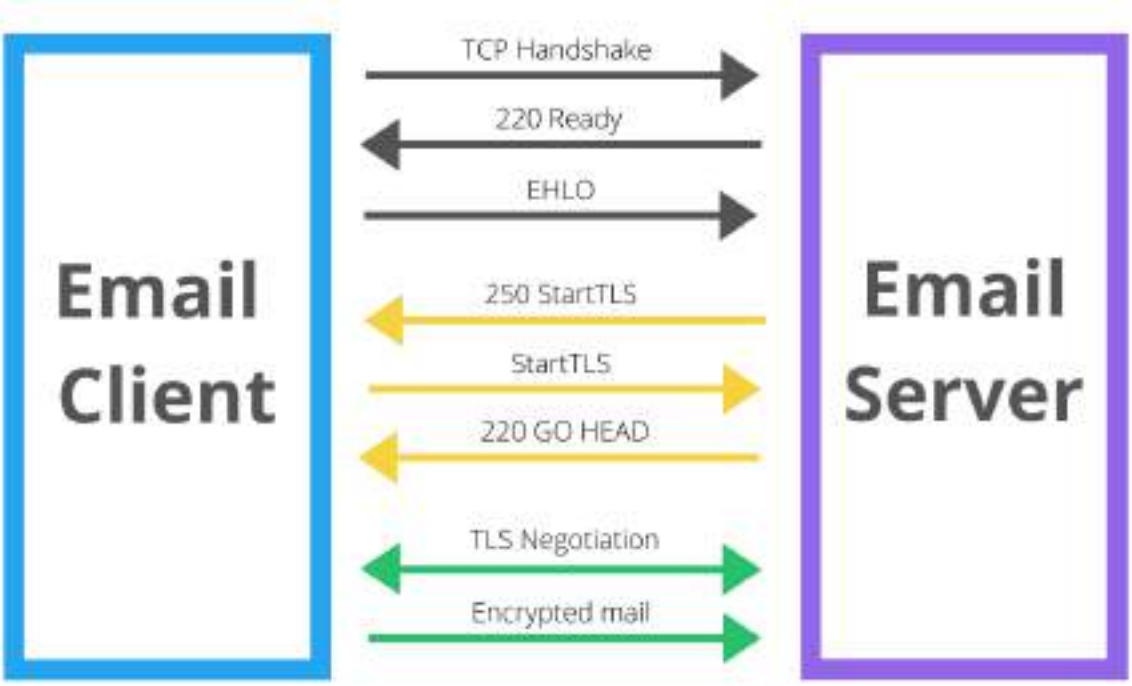
Slika 62 DKIM, SPF, DMARC zapisi, ki jih ustvari *Stalwart*

Vir: lasten vir

Kot je že bilo omenjeno v prejšnji točki, nam ob dodajanju domene *Stalwart* sam ustvari vse potrebne zapise, ki jih nato le dodamo v naš DNS strežnik.

4.6.3.3.2 Varnost povezave

Varnost povezave našega poštnega strežnika zagotavlja protokol TLS. Protokol TLS (*Transport Layer Security*) je standardizirana verzija protokola SSL (*Secure Socket Layer*). »TLS protokol uporablja šifriranje, ki temelji na kombinaciji asimetričnih in simetričnih postopkov ter uporablja pretočni ali blokovni način šifriranja.« (Rožman, 2008)



Slika 63 Primer komunikacije z implicitnim TLS

Vir: (Griffin, 2023)

V mojem izdelku TLS deluje na dva načina: implicitno in eksplicitno. Primer eksplicitnega načina je protokol HTTPS, kar pomeni, da ob povezavi, na primer na spletno administracijo našega sistema, tako brskalnik kot strežnik že od začetka pričakujeta varno, šifrirano povezavo. Pri implicitnem načinu pa se povezava sprva vzpostavi nešifrirano, šifriranje se prične šele po izvedbi ukaza STARTTLS.

Protokol	Vrata	TLS
SMTP	25	Brez/ekspliciten
STMPS (SMTP OVER SSL)	465	Impliciten
IMAP	143	Brez/ekspliciten
IMAPS	993	Impliciten

Tabela 3 E-poštni protokoli in TLS

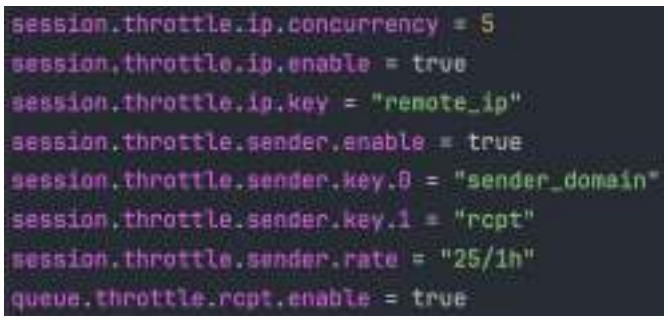
Vir: lasten vir

Naš izdelek uporablja samopodpisan SSL/TLS certifikat. V praksi to pomeni, da bo uporabnik pri direktni povezavi na eksplicitna vrata naletel na opozorilo o neveljavnem certifikatu, kar

lahko negativno vpliva na zaupanje v sistem. Običajna praksa v takšnih primerih je izvajanje SSL terminacije.

SSL terminacija pomeni, da se varna povezava vzpostavi med uporabnikom in izenačevalnikom obremenitve, medtem ko se promet med izenačevalnikom obremenitve in strežnikom nadaljuje nešifrirano. To nam omogoča uporabo veljavnega certifikata na izenačevalniku obremenitve, kar izboljša varnost in uporabniško izkušnjo, omogoča pa tudi bolj centraliziran pristop k šifriranju povezave. (HAProxy Technologies, LLC, 2024)

4.6.3.3.3 Dušenje in stopenjsko omejevanje

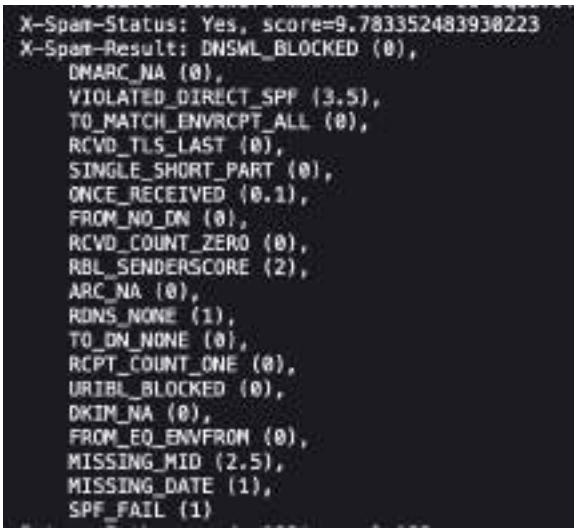


Slika 64 Privzete nastavitve dušenja v konfiguracijski datoteki

Vir: lasten vir

Dušenje *(*angl. *throttling)* in stopenjsko omejevanje *(*angl. *rate-limiting)* sta metodi za preprečevanje zlorabe strežnika. Dušenje deluje tako, da omejuje npr. število hkratnih povezav, stopenjsko omejevanje pa blokira povezavo po določenem številu zahtev iz istega vira. (Srivastava, 2023)

4.6.3.3.4 Spam filter in SMTP omejitve



Slika 65 Primer glave sporočila, ki je bilo označeno kot e-slama

Vir: lasten vir

Poštni strežnik *Stalwart* ima vgrajeno funkcijo za prepoznavno in označevanje *e-slame* pri dohodnih sporočilih. *Spam* filter, ki ga uporablja, je vzdrževan s strani razvijalcev in je deležen pogostih rednih posodobitev. Vključuje sezname brezplačnih domen, domen ki so znane po tem, da pošiljajo neželeno pošto, domen za enkratno uporabo in pa verificiranih (varnih) domen. Filtre lahko po želji tudi spreminjamo, lahko jim dodamo tudi verificirane domene (npr. če imamo testne strežnike z različnimi domenami in se zaradi lažjega testiranja ne želimo ukvarjati z e-poštnimi avtentikacijskimi metodami). Funkcija deluje tako, da za vsako sporočilo izračuna njegov *spam score*. Večji kot je ta rezultat, večja je možnost, da gre za neželeno pošto, mejo pa lahko tudi sami spreminjamo – privzeta meja je 5. Po tem, ko se sporočilu izračuna njegov rezultat, se le-ta vnese v glavo sporočila *X-Spam-Status* in *X-Spam-Result*.



Slika 66 SMTP avtentikacija

Vir: lasten vir

Da pa preprečimo, da bi naš strežnik bil uporabljen za neavtorizirano pošiljanje e-sporočil za domene, ki jih ne upravljamo, poskrbi SMTP avtentikacija. Ta na poštnem strežniku *Stalwart* deluje tako, da najprej preveri, ali je domena, iz katere želi uporabnik pošiljati, sploh lokalna. Če ni, sporočilo avtomatsko zavrne. Poleg tega SMTP avtentikacija omogoča preverjanje, ali je sicer avtenticiran uporabnik dejansko pooblaščen za pošiljanje iz določenega e-poštnega naslova. *Stalwart* v imeniškem strežniku preveri vse dovoljene e-poštne naslove; če naslova ni na seznamu, uporabnik ne more pošiljati sporočil iz tega e-naslova.

4.7 Testiranje

4.7.1 Vizualizacija delovanja (Grafana)

Za vizualizacijo delovanja, smo z sistemom *Grafana* pripravili dve nadzorni plošči, preko katerih lahko opazujemo podatke o delovanju obeh gruč. Vidimo lahko podatke o porabi procesorja, spomina, zasedenosti diska, hitrost branja in pisanja na disk.



Slika 67 Nadzorna plošča za sistem *Kubernetes*

Vir: lasten vir

Dodal pa sem še tudi del, ki se nanaša specifično na delovanje poštnega strežnika *Stalwart*. V tem razdelku najdemo dva tortna diagrama in števca (en prikazuje število prejetih sporočil iz mreže, drug pa dejansko število dostavljenih sporočil za vsako vozlišče) in pa okno, v katerem so prikazani zapisi dnevniških datotek vseh treh instanc poštnega strežnika. Število dostavljenih in prejetih sporočil izračunamo tako, da v dnevniških datotekah preštejemo vnose, ki vključujejo dogodek = *rcpt* in status *delivered*, nato pa vrednosti združimo glede na njihov izvor (torej kateremu vozlišču pripadajo) in dobimo tortni diagram.



Slika 68 Sekcija za nadzor poštnega strežnika v *Grafani*

Vir: lasten vir

4.7.2 Prilagoditve konfiguracije poštnega strežnika pred testiranjem

Preden pričnem z opisom obeh testov bi rad poudaril, da sem za namene testiranja izklopil določene varnostne nastavitve na strežnikih, saj so ti testi efektivno »Spam napadi«, s katerimi strežnike zasujemo z večjo količino ne verificiranih e-poštnih sporočil za namene preverjanja delovanja.

```
session.throttle.ip.concurrency = 5
session.throttle.ip.enable = false
session.throttle.ip.key = "remote_ip"
session.throttle.sender.enable = false
session.throttle.sender.key.0 = "sender_domain"
session.throttle.sender.key.1 = "rcpt"
session.throttle.sender.rate = "25/1h"
queue.throttle.rcpt.enable = false
```

Slika 69 Izsek iz konfiguracije z onemogočenimi varnostnimi nastavitvami

Vir: lasten vir

Iz zgornjega odseka konfiguracijske datoteke lahko razberemo, da sem izklopil:

- stopenjsko omejevanje pošiljanja iz istega IP naslova;
- dušenje števila e-poštnih sporočil v vrsti;
- dušenje pošiljanja iz iste domene;
- dušenje pošiljanja istemu prejemniku.

4.7.3 Skripta za izvajanje testov



```
#!/bin/bash
echo $(date) + " ZAČETEK TESTA" >> test.log

smtp_server="10.10.0.2" # Naslov našega poračunalniškega
smtp_port=2525 # 465 2526 za Sauro

# Maximum number of parallel sends
max_jobs=27


send_email() {
    local from=$1
    local to=$2
    local subject=$3
    local body=$4

    {
    echo "open $smtp_server $smtp_port"
    sleep 2
    echo "EHLO $smtp_server"
    sleep 2
    echo "MAIL FROM: <$from>"
    sleep 2
    echo "RCPT TO: <$to>"
    sleep 2
    echo "DATA"
    sleep 2
    echo "Subject: $subject"
    echo "From: $from"
    echo "To: $to"
    echo
    echo "$body"
    echo "."
    sleep 2
    echo "QUIT"
    sleep 1  # Ensure the QUIT command is processed
    } | telnet
}

i=0

while [ $i -lt 2 ]
do

while IFS=';' read -r from to subject body
do
    send_email "$from" "$to" "$subject" "$body" &

    while [ $(jobs -p | wc -l) -ge $max_jobs ]; do
        sleep 1
    done
done < "email_list.txt"

    i=$((i+1))
done

wait

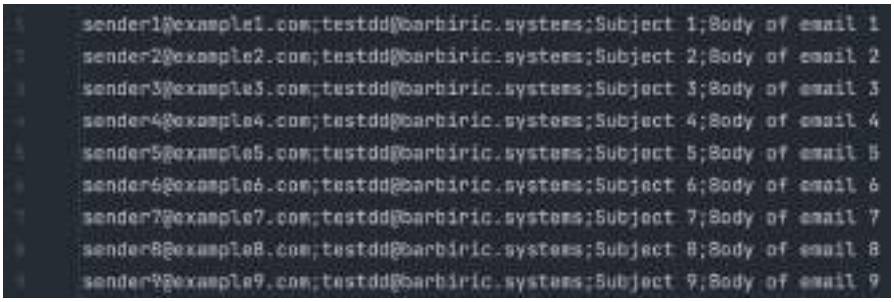
echo $(date) + " KONEC TESTA" >> test.log
```

Slika 70 Testna skripta spam.sh

Vir: lasten vir

Za oba testa smo uporabili isto testno skripto, ki oba sistema za kratek čas obremeni s sporočili. Testna skripta z programom *telnet* pošilja sporočila. To sem dosegel tako, da s pavzami v

program pošiljamo SMTP ukaze za pošiljanje sporočil. Sporočila in podatke pridobi iz tekstovne datoteke, v kateri smo definirali 100 različnih pošiljateljev s 100 različnimi sporočili, prejemnik pa ostaja vedno enak.

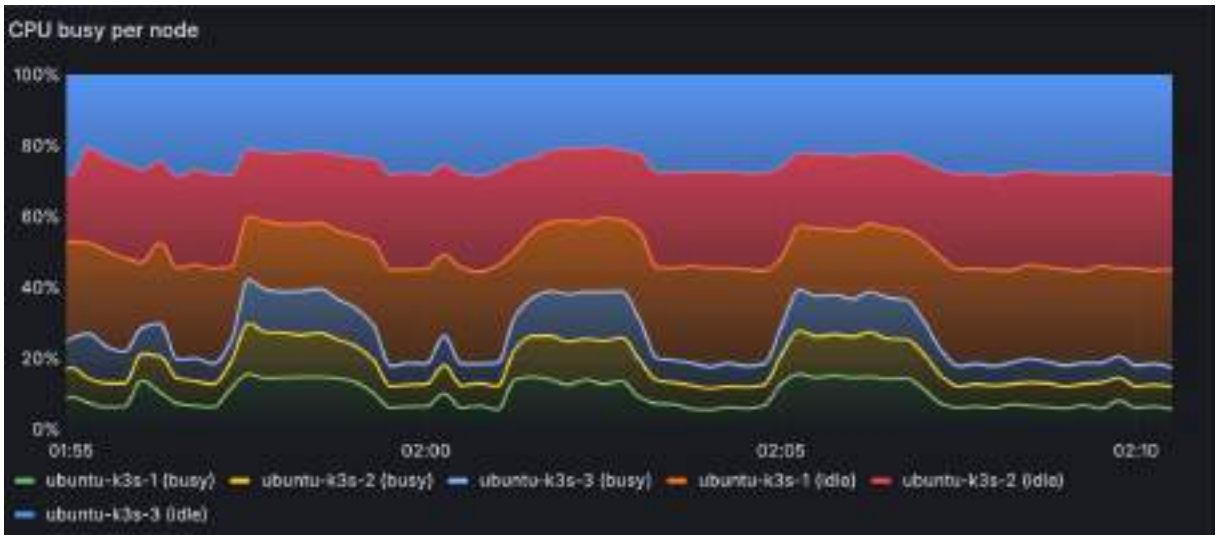


Slika 71 Izsek iz datoteke z pošiljatelji

Vir: lasten vir

Hkrati se paralelno izvaja 27 instanc (to specifično številko sem izbral, ker je deljiva z 3, kar teoretično pomeni, da bo vsako vozlišče v gruči hkrati prejelo 9 sporočil) programa *telnet,* ki skupaj pošljejo 100 sporočil, proces pa se dvakrat ponovi, torej je z vsakim izvajanjem skripte poslanih skupno 200 sporočil. Skripto smo zagnali z ukazom *./spam.sh && sleep 120 && ./spam.sh && sleep 120 && ./spam.sh,* torej smo jo za oba testna scenarija pognali 3-krat v razmaku 2 minut. 2minutne razmake smo izbrali zaradi boljše preglednosti na grafih, da lažje vidimo, kdaj so se testi izvajali. Vsaka izvedba skripte na mojem prenosnem računalniku (Apple MacBook Pro M1, povezan preko brezžične povezave) je trajala natanko 1 minuto in 46 sekund.

4.7.4 Test 1: Obremenitev sistema ter učinkovitost porazdeljevanja



Slika 72 Zasedenost procesorja pri testu 1 (*Kubernetes*)

Vir: lasten vir



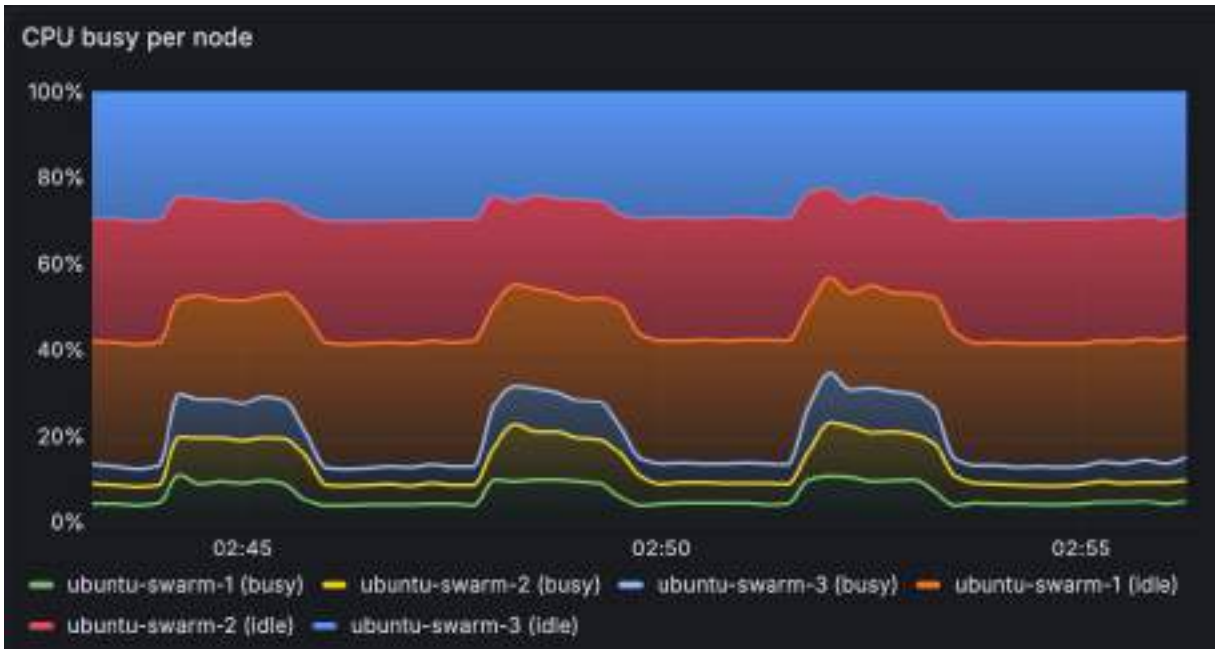
Slika 73 Utilizacija pisanja na disk pri testu 1 (*Kubernetes*)

Vir: lasten vir

Tabela 4 Rezultati testa 1 (*Kubernetes*)

Št. testa	**Vozlišče**	**CPU**	**Disk**	**Prejeta sporočila**	**Dostavljena sporočila**
1	K3s-1	51,5 %	1,41MB/s	67	63
1	K3s-2	47,8 %	1,24MB/s	67	88
1	K3s-3	44,4 %	1,34MB/s	66	49
2	K3s-1	46,3 %	1,53MB/s	66	61
2	K3s-2	48,8 %	1,34MB/s	67	72
2	K3s-3	48,3 %	1,43 MB/s	67	67
3	K3s-1	50,4 %	1,44 MB/s	67	59
3	K3s-2	48,6 %	1,29 MB/s	67	85
3	K3s-3	43,3 %	1,37 MB/s	66	56
POVPREČJE	K3s-1	**49,4 %**	1,46 MB/s	66,67	61,0
POVPREČJE	K3s-2	48,40 %	1,29 MB/s	**67,0**	**81,67**
POVPREČJE	K3s-3	45,33 %	**1,38 MB/s**	66,33	57,33
POVPREČJE	SKUPAJ	47,71 %	1,38 MB/s	66,67	66,67
ODSTOPANJE	K3s-1	1,69 %	0,08 MB/s	0	5,67
ODSTOPANJE	K3s-2	0,69 %	**–0,09 MB/s**	0,33	**15**
ODSTOPANJE	K3s-3	**2,38 %**	0 MB/s	**–0,34**	–9,34
ODSTOPANJE	SKUPAJ	1,73 %	0,069 MB/s	0,27	10,71

Vir: lasten vir



Slika 74 Zasedenost procesorja pri testu 1 (*Docker Swarm*)

Vir: lasten vir



Slika 75 Utilizacija pisanja na disk pri testu 1 (*Docker Swarm*)

Vir: lasten vir

Tabela 5 Rezultati testa 1 (*Docker Swarm*)

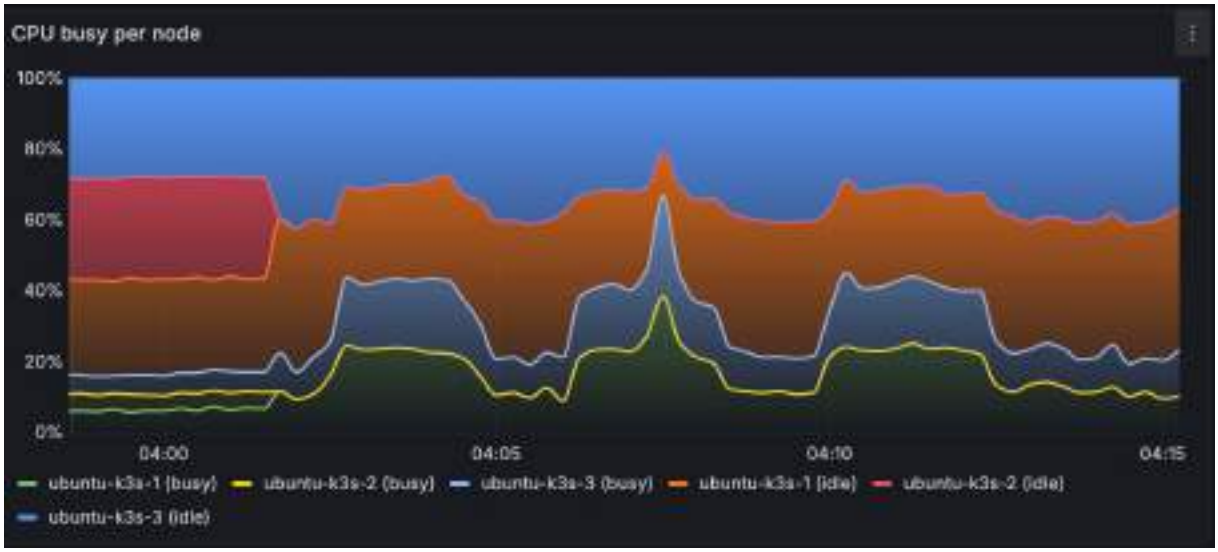
Št. testa	**Vozlišče**	**CPU**	**Disk**	**Prejeta sporočila**	**Dostavljena sporočila**
1	Swarm 1	34,9 %	1,43 MB/s	66	63
1	Swarm 2	41,1 %	1,11 MB/s	67	63
1	Swarm 3	33,2 %	1,27 MB/s	67	74
2	Swarm 1	35,8 %	1,34 MB/s	67	73
2	Swarm 2	40,2 %	1,10 MB/s	67	57
2	Swarm 3	32,4 %	1,27 MB/s	66	70
3	Swarm 1	35,8 %	1,29 MB/s	67	73
3	Swarm 2	42,0 %	1,14 MB/s	66	53
3	Swarm 3	34,9 %	1,13 MB/s	67	74
POVPREČJE	Swarm 1	35,5 %	**1,35 MB/s**	66,67	**69,67**
POVPREČJE	Swarm 2	**41,1 %**	1,12 MB/s	66,67	57,67
POVPREČJE	Swarm 3	33,5 %	1,22 MB/s	66,67	72,67
POVPREČJE	SKUPAJ	36,7 %	1,23 MB/s	66,67	66,67
ODSTOPANJE	Swarm 1	–1,2 %	**0,12 MB/s**	0	3,0
ODSTOPANJE	Swarm 2	**4,4 %**	–0,11 MB/s	0	**–9,0**
ODSTOPANJE	Swarm 3	–3,2 %	–0,01 MB/s	0	6,0
ODSTOPANJE	SKUPAJ	3,22 %	0,094 MB/s	0	6,48

Vir: lasten vir

Iz rezultatov prvega testa je razvidno, da čeprav pri obeh sistemih vozlišča enakomerno prejemajo sporočila, se dejansko število dostavljenih sporočil na posameznem vozlišču razlikuje. Do variacij prihaja, ker se vrsta sporočil shranjuje v podatkovni bazi, nato pa vsak strežnik naključno dostavi določeno število sporočil, kar pomeni, da ne *Kubernetes* ne *Docker Swarm* ne vplivata neposredno na to distribucijo. Povprečna poraba procesorja in količina pisanja na disk sta v sistemu *Kubernetes* sicer višji kot v sistemu *Docker Swarm* (47,71 %, 1,38

MB/s proti 36,7 %, 1,23 MB/s), vendar pa je odstopanje med posameznimi vozlišči v sistemu *Kubernetes* manjše (1,73 %, 0,069 MB/s proti 3,22 %, 0,094 MB/s), kar nakazuje, da je delovna obremenitev v sistemu *Kubernetes* nekoliko bolj enakomerno razporejena.

4.7.5 Test 2: Obremenitev sistema ter učinkovitost porazdeljevanja pri izpadu enega vozlišča



Slika 76 Zasedenost procesorja pri testu 2 (*Kubernetes*)

Vir: lasten vir



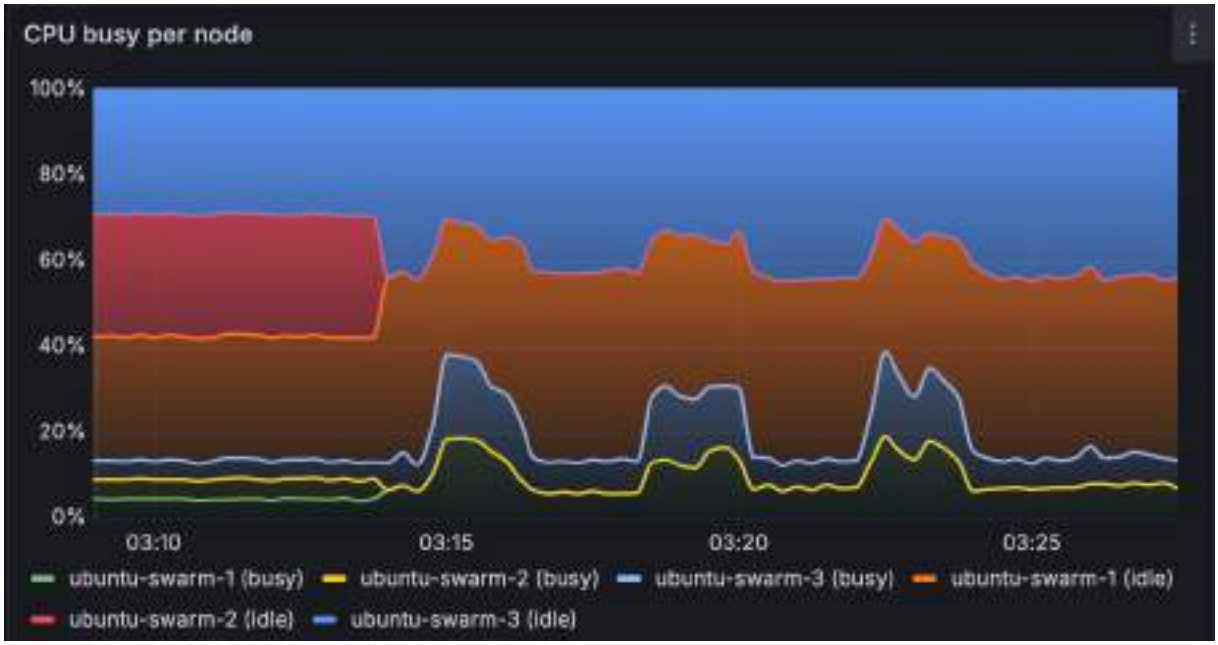
Slika 77 Utilizacija pisanja na disk pri testu 2 (*Kubernetes*)

Vir: lasten vir

Tabela 6 Rezultati testa 2 (*Kubernetes*)

Št. testa	Vozlišče	CPU	Disk	Prejeta sporočila	Dostavljena sporočila
1	K3s-1	52,3 %	1,67 MB/s	100	92
1	K3s-3	47,3 %	1,57 MB/s	100	108
2	K3s-1	78,4 %	1,82 MB/s	100	89
2	K3s-3	63,2 %	1,67 MB/s	100	111
3	K3s-1	55,4 %	1,77 MB/s	100	78
3	K3s-3	46,4 %	1,71 MB/s	100	122
POVPREČJE	K3s-1	**62,03 %**	**1,75 MB/s**	100	86,33
POVPREČJE	K3s-3	52,30 %	1,65 MB/s	100	**113,67**
POVPREČJE	SKUPAJ	57,17 %	1,70 MB/s	100	100,00

Vir: lasten vir



Slika 78 Zasedenost procesorja pri testu 2 (*Docker Swarm*)

Vir: lasten vir



Slika 79 Utilizacija pisanja na disk pri testu 2 (*Docker Swarm*)

Vir: lasten vir

Tabela 7 Rezultati testa 2 (*Docker Swarm*)

Št. testa	**Vozlišče**	**CPU**	**Disk**	**Prejeta sporočila**	**Dostavljena sporočila**
1	Swarm 1	38,7 %	1,32 MB/s	100	98
1	Swarm 3	32,6 %	1,06 MB/s	100	102
2	Swarm 1	34,2 %	1,25 MB/s	100	92
2	Swarm 3	37,4 %	0,93 MB/s	100	108
3	Swarm 1	37,5 %	1,25 MB/s	100	100
3	Swarm 3	38,8 %	0,94 MB/s	100	100
POVPREČJE	Swarm 1	**36,8 %**	**1,27 MB/s**	100	96,67
POVPREČJE	Swarm 3	36,27 %	0,98 MB/s	100	**103,33**
POVPREČJE	SKUPAJ	36,53 %	1,13 MB/s	100	100,00

Vir: lasten vir

Pri rezultatih drugega testa je razvidno, da v sistemu *Swarm* obremenitev ostane praktično ista kljub večjemu številu prejetih sporočil na posameznem vozlišču. Pri sistemu *Kubernetes* pa lahko razberemo, da je prišlo pri drugi ponovitvi testa do nenadnega povečanja porabe procesorja. *Kubernetes* preverja stanje vozlišč vsakih 30 sekund, medtem ko se metrike, ki jih spremljamo, posodabljajo vsakih 15 sekund. Po analizi grafa sem ugotovil, da je do izpada

vozlišča k3s-2 prišlo ob 04:01:15. Prav tako vem, da imajo stroki za servisa *YugabyteDB* in *Prometheus* 5-minutno obdobje prizanašanja *(*angl. *grace period)* pred terminacijo. Izredno povečana zasedenost procesorja v gruči se je pojavila med drugo ponovitvijo testa, ob okoli 04:07:15. Torej je časovna razlika med izpadom vozlišča in povečano zasedenostjo procesorja dolga šest minut. Če upoštevamo minimalno zamudo zaradi posodabljanja podatkov, približno 15 sekund, in *Kubernets*ovo preverjanje na 30 sekund, dobimo skoraj isto številko. To dokazuje, da je izredna povečanost zasedenosti procesorja dejansko posledica tega, da vozlišča prejmejo informacije o začetku terminacije nedostopnih strokov v gruči.

4.7.6 Primerjava rezultatov obeh testov in kritična ocena

Če primerjamo rezultate obeh testov, lahko vidimo, da je povprečna poraba virov pri sistemu *Docker Swarm* v obeh primerih praktično identična (36,7 %, 1,23 MB/s proti 36,8 %, 1,27 MB/s), kar se ujema tudi z izračunanim odstopanjem v prvem testu (47,71 %, 1,38 MB/s proti 57,17 %, 1,70 MB/s), kar je mnogo več kot izračunano odstopanje iz prvega testa. Vendar je, kot sem že razložil, to posledica pričetka terminacij nedostopnih strokov v gruči. 100 % prejetje in dostava sporočil v obeh testih pa nakazuje tudi na to, da poštni strežnik in ostali servisi kljub izpadu enega vozlišča v obeh gručah še naprej deluje nemoteno. Iz teh podatkov lahko sklepamo, da je sposobnost povečanja obsega v obeh gručah zelo dobra, hkrati pa to potrjuje, da sem za izdelek res izbral primerne servise.

4.8 Primerjava obeh sistemov in načinov vzpostavitve

Moj pristop do vzpostavitve v sistemu *Docker Swarm* je bil monoliten, saj sem poštni strežnik in vse odvisnosti vzpostavil z le dvema *Docker Compose* datotekama. S tem ni nič narobe, saj sem z uporabo dodatnih skript servisom omogočil »zavedanje« stanja ostalih servisov v gruči. Slabost takšnega načina vzpostavitve je, da je že za eno spremembo potrebno ponovno vzpostaviti čisto vse servise v gruči, ampak hkrati je to tudi prednost, saj lahko vsako spremembo ovrednotimo v »svežem« okolju. Monoliten pristop je hkrati imel tudi velik vpliv pri *Ansible* vrednostnih datotekah, saj so te hitro postale zelo nepregledne.

Pri vzpostavitvi v sistemu *Kubernetes* pa sem uporabil bolj distribuiran, porazdeljen pristop – vsak servis je bil vzpostavljen z ločeno *Ansible* nalogo, posledično tudi z ločenimi knjigami, kar je omogočalo, da sem lahko (če je bilo to potrebno) vsak servis vzpostavil neodvisno od ostalih. Tak pristop omogoča večjo fleksibilnost pri testiranju različnih konfiguracij, saj ni potrebe po nepotrebnih ponovnih vzpostavitvah preostalih servisov.

Menim, da je porazdeljen pristop najprimernejši, ker omogoča večjo fleksibilnost in lažje spremembe po sami vzpostavitvi. Če bi izdelek delal še enkrat, bi tak pristop uporabil pri obeh gručah.

Pri vzpostavitvi servisov z *Ansible* knjigami iger bi rad dodal tudi, da za sistem *Kubernetes* ni potrebno dejansko kopirati datotek na vozlišče (z izjemo Helm diagrama za poštni strežnik *Stalwart*), kot je to potrebno pri sistemu *Docker Swarm*. To pomeni, da je dejansko sama vzpostavitev z mojim načinom avtomatizacije lažja in varnejša.

Velika razlika med sistemoma so tudi možnosti za trajno shranjevanje in ustvarjanja trajnih nosilcev. Rešitev, kot je *Longhorn* v sistemu *Kubernetes*, zelo olajša delo s trajnimi nosilci. Ko sem enkrat končal s konfiguracijo *Longhorn*, o trajnih nosilcih sploh nisem več razmišljal, saj so bili ustvarjeni skupaj z inštalacijo aplikacij in ob izbrisu imenskih okolij tudi odstranjeni. Pri sistemu *Docker Swarm* pa so bile moje možnosti zelo omejene. *Docker Swarm* sicer podpira avtomatsko ustvarjanje trajnih nosilcev, ampak problem je, da bodo ti vedno shranjeni na primarnem disku, v izdelku pa sem želel doseči, da so vsi aplikacijski podatki na ločenem disku, saj je tako lažje spremljati zasedenost in uporabo. Če bi hranil vse na primarnem disku, bi zelo težje videl, koliko prostora aplikacije zasedajo, pa tudi težje bi bilo razločiti, kdaj bere in piše gruča in kdaj ostali deli vozlišča. Prav tako nisem želel uporabiti rešitev, kot so npr. *sshfs*, ker sem želel, da vsak del izdelka teče v gruči, alternative pri sistemu *Docker Swarm* pa zahtevajo konfiguracijo datotečnih zahtev izven le-te.

V obeh sistemih obstajajo različne abstrakcije pri definiciji aplikacije. Kot primer poglejmo sistem *Kubernetes*, kjer bo aplikacija X tekla kot kontejner v stroku, katerega tip je določen iz strani načina postavitve (*Stateful Set, Deployment,* …). Do aplikacije, ki teče v kontejnerjih stroka, pa dostopamo preko servisov. Pri sistemu *Docker Swarm* bo ista aplikacija tekla kot kontejner v servisu, ki pa združuje vso konfiguracijo ter načine postavitve – kot da bi združili *Kubernetes* strok in *Kubernetes* servis. Večja kompleksnost pri sistemu *Kubernetes* omogoča veliko boljši nadzor nad aplikacijami, načini postavitve in tudi porabo virov.

Dober primer razlik med sistemoma je tudi način dostopa do odvisnosti v gruči. Pri sistemu *Docker Swarm* za dostop uporabljamo imena gostiteljev *<ime_kontejnerja>:<vrata>*, v sistemu *Kubernetes* pa do posameznega servisa dostopamo s kombinacijo imena servisa in imenskega prostora *<ime-servisa>.<imenski-prostor>.svc.cluster.local:<vrata>*, kar omogoča bolj predvidljiv in lažji dostop kot v gruči *Docker Swarm*. V obeh sistemih je sicer bilo potrebno zaradi želje po tem, da niso vsi deli servisov dostopni na lokalnem omrežju, vzpostaviti notranji izenačevalnik obremenitve s *HAProxy* – pri sistemu *Kubernetes* zaradi potreb podatkovne baze, pri *Docker Swarm* pa še zaradi potreb objektne shrambe *MinIO*. V nasprotnem primeru bi lahko to reševali tudi z zunanjim izenačevalnikom obremenitve, ki pa ga pri izdelku uporabljamo le za izenačevanje pri pošiljanju pošte na strežnike.

Sistem *Kubernetes* ima zaradi svoje popularnosti in pogostosti uporabe pod seboj zelo veliko skupnost ljudi. To v praksi pomeni, da je na voljo veliko več informacij in pomoči pri vzpostavitvi aplikacij kot za sistem *Docker Swarm*. Vsak servis, uporabljen v izdelku, je imel za sistem *Kubernetes* na voljo mnogo večjo dokumentacijo (z izjemo poštnega strežnika *Stalwart*), pa tudi število člankov in ljudi, ki so se že ukvarjali s postavitvijo servisov v danem sistemu je bilo pri sistemu *Kubernetes* mnogo več. Posledično je postavitev z sistemom *Kubernetes* dejansko lažja, saj zraven prej omenjenih razlogov obstaja praktično za vsak servis Helm diagram, ki je pripravljen za postavitev servisa za veliko razpoložljivost, med tem ko sem se pri sistemu *Docker Swarm* moral večkrat znajti sam in bil velikokrat primoran uporabljati manj učinkovite rešitve za iste servise.

Pri skrivnostih in konfiguracijah *(Kubernetes secret, Docker secret, Kubernetes ConfigMap, Docker config)*, obstaja veliko podobnosti. Tako *Docker Swarm* kot *Kubernetes* omogočata, da ko skrivnosti in konfiguracije definiramo na enem vozlišču, so potem na voljo v celotni gruči. Sistem *Kubernetes* omogoča tudi, da v eni skrivnosti hranimo več vrednosti različnih tipov, npr. datoteke in besedilo, pri sistemu *Docker Swarm* pa lahko v skrivnosti hranimo le eno vrednost. Prav tako pa je treba omeniti, da sistem *Kubernetes* omogoča, da vpnemo skrivnost ali

konfiguracijo v kontejner na različne načine – kot okoljsko spremenljivko ali datoteko, v sistemu *Docker Swarm* pa le v datoteko. Po ustvarjanju v sistemu *Docker Swarm* spremembe več niso mogoče, lahko jih le izbrišemo, ampak le, če jih ne uporablja noben aktivni servis, medtem ko sistem *Kubernetes* omogoča urejanje tako skrivnosti kot konfiguracij tudi, ko so v uporabi (da se sprememba v strokih dejansko uveljavi, jih je treba prej ponovno zagnati). To je zelo velika slabost sistema *Docker Swarm*, saj je za spremembe konfiguracij in skrivnosti potrebno aplikacijo čisto odstraniti.

Sistem *Docker Swarm* ponuja preprosto definiranje servisov z datotekami *compose*, vendar žal brez mnogih funkcij, ki delujejo v samostojnem načinu delovanja *Docker*. Primer take funkcije je *depends_on*, ki omogoča, da določimo vrstni red, v katerem se bodo servisi vzpostavili. Brez te funkcije sem bil primoran uporabljati lastne skripte, ki so reševale ta problem.

Kot sem omenil in ugotovil med testi (RE), pri delovanju porablja gruča *Swarm* manj sistemskih virov kot gruča *Kubernetes*, za to menim da je *Docker Swarm* primernejši v situacijah, ko imamo zelo omejene sistemske vire. V splošnem pa menim, da je sistem *Kubernetes* primernejša rešitev za kompleksnejše aplikacije, med katere bi štel tudi naš izdelek, saj četudi porablja več virov, je dejanska vzpostavitev aplikacij zaradi vseh omenjenih prednosti mnogo lažja in dosti bolje podprta s strani razvijalcev.

5 SKLEP

Z diplomskim delom sem želel podrobno preučiti prednosti, slabosti, razlike in podobnosti sistemov *Kubernetes* in Docker Swarm. Želel sem tudi izboljšati svoje znanje glede sistemov za upravljanje gruč in delovanja porazdeljenih visoko razpoložljivih aplikacij. Diplomsko delo se deli na dva dela; teoretični in praktični del. V teoretičnem sem želel raziskati in predstaviti teorijo za obema sistemoma in ker je predmet primerjave tudi vzpostavitev poštnega strežnika, sem pisal tudi o e-pošti. V praktičnem sklopu sem najprej predstavil vse servise, ki sem jih postavil v obeh sistemih, navedel sem tudi razloge za mojo odločitev za uporabo le-teh. Za tem sem podrobno opisal postopke vzpostavitve izdelka, torej od vzpostavitve samih gruč do vzpostavitve servisov in dodatnih specifik pri obeh sistemih. Kasneje sem opisal tudi kompleksnost konfiguracije poštnega strežnikas s poudarkom na varnostni konfiguraciji.

Pred pričetkom pisanja sem si na podlagi predhodnega znanja in osnovnega raziskovanja za temo zastavil tudi hipoteze:

Prva hipoteza (H1) pravi: »Vzpostavitev Docker Swarm »gruče«, bo preprostejša kot vzpostavitev »gruče« *Kubernetes*.« Kot sem že napisal pri točkah o vzpostavitvi gruč, je bila vzpostavitev obeh gruč relativno preprosta, to pa zato, ker smo gručo *Kubernetes* vzpostavili z pomočjo avtomatizacije. Če gledamo enakovredno ročno vzpostavitev obeh gruč, je bila vzpostavitev gruče Docker Swarm dosti lažja, zato je ta hipoteza potrjena.

Druga hipoteza (H2) pravi: »Sistem *Kubernetes* omogoča naprednejše upravljanje s kontejnerji, kot sistem Docker Swarm«. Menim, da sem z obema deloma diplomskega dela to hipotezo potrdil, saj sistem *Kubernetes* podpira dosti bolj kompleksnejše vzpostavitve kontejnerjev kot sistem Docker Swarm, ki je po mojem mnenju primeren le za preproste aplikacije, ki ne potrebujejo kompleksnih konfiguracij.

Tretja hipoteza (H3) pravi: »Prilagajanje obsega poštnega strežnika bo pri *Kubernetes* »gruči« boljša kot pri »gruči« Docker Swarm.« S testiranjem smo ugotovili, da oba sistema omogočata zelo dobro prilagajanje obsega Vendar ima, kot smo ugotovili že pri teoretičnem delu, *Kubernetes* še vedno prednosti, kot so avtomatsko prilagajanje obsega, ki omogoča da servis prilagaja število kontejnerjev glede na obremenitev. Hipotezo glede na vse ugotovitve in faktorje potrjujem.

Četrta hipoteza (H4) je definirana kot: »Nadzor in sledenje dogodkov poštnega strežnika bo pri *Kubernetes* »gruči« boljše kot pri »gruči« Docker Swarm.« To hipotezo delno potrjujem zaradi

tega, ker za nadzor in slednje dogodkom v obeh gručah uporabljamo iste rešitve, edina prednost, ki jo ima poštni strežnik v gruči *Kubernetes* je da lahko z orodjem *kubectl* bolj podrobno analiziramo *Kubernetes* komponente poštnega strežnika.

Peta hipoteza (H5) trdi: »Sistem *Kubernetes* ima večjo uporabniško skupnost kot sistem Docker Swarm.« Glede na izkušnje in vso dokumentacijo lahko potrdim tudi to hipotezo. Docker Swarm je deležen, le frakcije popularnosti sistema *Kubernetes*.

Šesta hipoteza (H6), preverja naslednjo trditev: "Poraba sistemskih virov bo pri sistemu *Kubernetes* večja kot pri sistemu Docker Swarm". Kot sem že ugotovil pri testiranju in primerjavi, ima sistem *Kubernetes* zaradi kompleksnejše arhitekture večjo porabo sistemskih virov kot Docker Swarm.

Dodatna hipoteza BTEC (H7): »Big data will negatively impact the privacy of users in the long run.« Pri teoretičnem delu o e-pošti sem raziskoval tudi o velikih podatkih in kako bi lahko bili uporabljeni v kontekstu e-pošte. Ugotovil sem, da je analitika velikih podatkov v kontekstu neželene pošte zelo koristna za prepoznavanje vzorcev v takšnih sporočilih. Hkrati pa sem tudi ugotovil, da lahko procesiranje, hramba in netransparentnost podjetij vodi do hudih posledic za našo zasebnost. Tudi to hipotezo potrjujem, saj menim, da se bo število podatkov, ki jih podjetja o nas zbirajo, še zelo povečalo, kar bo direktno povečalo možnosti za njihovo zlorabo, pa četudi obstajajo pozitivne aplikacije procesiranja velikih podatkov.

Menim, da sem se z diplomsko nalogo zelo veliko naučil o sistemih za upravljanje z gručami, ter dobro predstavil delovanje kontejnerskih delovnih obremenitev.

6 KAZALO VIROV IN LITERATURE

AlMahmoud, A., Damiani, E., Hadi, O., & Al-Hammadi, Y. (2019, September 1). Spamdoop: A Privacy-Preserving Big Data Platform for Collaborative Spam Detection. *IEEE Transactions on Big Data, 5*(3), 293-303.

Amazon.com, Inc. (2006, March 14). *Amazon Web Services Launches*. Retrieved May 2024, from Amazon press center: https://press.aboutamazon.com/2006/3/amazon-web-services-launches

Awati, R., & Teravainen, T. (n.d.). *email spam*. Retrieved June 2024, from TechTarget Security: https://www.techtarget.com/searchsecurity/definition/spam

Bloom, Z. (2017, September 9). *The History of Email*. Retrieved April 2024, from The Cloudflare Blog: https://blog.cloudflare.com/the-history-of-email

Broadcom Inc. (n.d.). *What is a virtual machine?* Retrieved April 2024, from WMware: https://www.vmware.com/topics/glossary/content/virtual-machine.html

Cloud Native Storage with Longhorn: Overview. (n.d.). Retrieved April 2024, from Rancher Documentation: https://ranchermanager.docs.rancher.com/integrations-in-rancher/longhorn/overview

Cloudflare, Inc. (n.d.). *Learning center: What are DMARC, DKIM, and SPF?* Retrieved July 2024, from Cloudflare: https://www.cloudflare.com/learning/email-security/dmarc-dkim-spf/

Docker Compose – What is It, Example & Tutorial. (2023, September 15). Retrieved April 2024, from Spacelift: https://spacelift.io/blog/docker-compose

Docker Inc. (n.d.). *Guides: Docker overview*. Retrieved June 2024, from Docker docs: https://docs.docker.com/guides/docker-overview/

Docker Inc. (n.d.). *Manuals: Swarm mode*. Retrieved June 2024, from Docker docs: https://docs.docker.com/engine/swarm/#feature-highlights

Docker Inc. (n.d.). *Swarm mode: How Swarm nodes work?* Retrieved June 2024, from Docker Documentation: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/

Docker Inc. (n.d.). *What is a container?* Retrieved April 2024, from Docker: https://www.docker.com/resources/what-container/

Docker Mailserver Organization. (n.d.). *Documentation: About*. Retrieved June 2024, from docker-mailserver: https://docker-mailserver.github.io/docker-mailserver/latest/

Donenfeld, J. A. (2020, June 1). WireGuard: Next Generation Kernel Network Tunnel. Retrieved from https://www.wireguard.com/papers/wireguard.pdf

Fenollosa , C. (2022, September 4). *After self-hosting my email for twenty-three years I have thrown in the towel. The oligopoly has won.* Retrieved April 2024, from Carlos Fenollosa: https://cfenollosa.com/blog/after-self-hosting-my-email-for-twenty-three-years-i-have-thrown-in-the-towel-the-oligopoly-has-won.html

Fortinet, Inc. (n.d.). *19 Types of Phishing Attacks*. Retrieved June 2024, from Fortinet resources: https://www.fortinet.com/resources/cyberglossary/types-of-phishing-attacks

FreeIPA Team. (n.d.). *About: What is FreeIPA?* Retrieved June 2024, from FreeIPA: https://www.freeipa.org/About.html

Gadgets 360 Staff. (2022, August 9). *Google Search, Maps, Photos, YouTube Recover Following Outages in Some Parts of the World*. Retrieved 2024 March, from Gadgets 360: https://www.gadgets360.com/internet/news/google-search-maps-photos-drive-youtube-experiencing-outages-in-some-parts-of-the-world-3238156

Google Inc. (n.d.). *What is Kubernetes?* Retrieved April 2024, from Google Cloud: https://cloud.google.com/learn/what-is-kubernetes

Google LLC. (n.d.). *What is big data?* Retrieved June 2024, from Google Cloud: https://cloud.google.com/learn/what-is-big-data

Griffin, J. (2023, January 2). *What is StartTLS?* Retrieved July 2024, from Sendgrid blog: https://sendgrid.com/en-us/blog/what-is-starttls

Hailey, C. (2021, June 1). *An Introduction to Prometheus Monitoring (2021)*. Retrieved July 2024, from Sensu blog: https://sensu.io/blog/introduction-to-prometheus-monitoring

HAProxy Community. (n.d.). *Introduction to HAProxy*. Retrieved May 2024, from Haproxy Documentation: https://docs.haproxy.org/3.0/intro.html#3

HAProxy Technologies, LLC. (2024, June 3). *What is SSL/TLS termination?* Retrieved June 2024, from HAProxy glossary: https://www.haproxy.com/glossary/what-is-ssl-tls-termination

Hewage, R. (2020, March 23). *What is a Container?* . Retrieved from faun.pub: https://faun.pub/what-is-a-container-78f08ef884a8

IAC.ai. (2023, July 5). *The Implications of Big Data and Privacy Concerns*. Retrieved June 2024, from Linkedin: https://www.linkedin.com/pulse/implications-big-data-privacy-concerns-intelligentautomationcompany/

k3s-ansible authors. (n.d.). *k3s-ansible github page*. Retrieved April 2024, from Github: https://github.com/k3s-io/k3s-ansible

Kaspersky Lab. (n.d.). *Damage caused by spam*. Retrieved June 2024, from Kaspersky encyclopedia: https://encyclopedia.kaspersky.com/knowledge/damage-caused-by-spam/

LogicMonitor Inc. (2023, February 9). *What is HAProxy, and what is it used for?* Retrieved May 2024, from Logic Monitor: https://logicmonitor.com/blog/what-is-haproxy-and-what-is-it-used-for

Mailu authors. (n.d.). *Mailu*. Retrieved June 2024, from Mailu documentation: https://mailu.io/2024.06/

Markuson, D. (2021, April 15). *A BRIEF HISTORY OF EMAIL: FROM PROTOCOLS TO THE @ SYMBOL*. Retrieved May 2024, from Nord Security blog: https://nordsecurity.com/blog/history-of-email

Mihalcea, V. (2023, March 24). *What is YugabyteDB*. Retrieved April 2024, from vladmihalcea.com: https://vladmihalcea.com/yugabytedb/

MinIO, Inc. (n.d.). *Core Administration Concepts: What is object storage?* Retrieved from MinIO documentation: https://min.io/docs/minio/container/administration/concepts.html

MinIO, Inc. (n.d.). *MinIO Object Storage for Container: Data Encryption (SSE)*. Retrieved May 2024, from Minio documentation: https://min.io/docs/minio/container/operations/server-side-encryption.html

MinIO, Inc. (n.d.). *Architecture: The Foundation for Large Scale Data Infrastructure*. Retrieved May 2024, from min.io: https://min.io/product/enterprisearchitecture

Mucci, T., & Stryker, C. (2024, April 5). *What is big data analytics?* Retrieved June 2024, from IBM Think: https://www.ibm.com/topics/big-data-analytics

Petrosyan, A. (2024, March 7). *Global spam volume as percentage of total e-mail traffic from 2011 to 2023* . Retrieved from Statista: https://www.statista.com/statistics/420400/spam-email-traffic-share-annual/

Pratt, M. (2023, December 14). *Introduction to Grafana Loki*. Retrieved July 2024, from Baeldung: https://www.baeldung.com/ops/grafana-loki

Rahul, A., & Casey, K. (n.d.). *Containers and virtualization: Docker Swarm*. Retrieved June 2024, from TechTarget: https://www.techtarget.com/searchitoperations/definition/Docker-Swarm

Red Hat, Inc. (2022, May 13). *Data services: What is Grafana?* Retrieved June 2024, from Red Hat: https://www.redhat.com/en/topics/data-services/what-is-grafana

Rožman, S. (2008, May 3). *TLS/SSL (Transport Layer Security/Secure Socket Layer)*. Retrieved July 2024, from Abakus: https://www.abakus.si/sl/blog/2008_03_05_SR_TLS_SSL

Sanebox Inc. (n.d.). *The History of IMAP*. Retrieved April 2024, from Pop2Imap: https://www.pop2imap.com/

Sanebox Inc. (n.d.). *The History of POP*. Retrieved April 2024, from IMAP vs POP3: https://www.pop2imap.com/

Schmitt, J. (2023, March 16). *What is Helm? A complete guide*. Retrieved April 2024, from CircleCI Blog: https://circleci.com/blog/what-is-helm/

Sholly, E. (2023, October 4). *2023 Self-Host User Survey Results*. Retrieved April 2024, from selfhst.sh: Self-hosted news, content, updates, and more.: https://selfh.st/survey/2023-results/

Singh Gill, N. (2022, October 31). *Insights: Minio Distributed Object Storage Architecture and Performance* . Retrieved May 2024, from xenonstack.com: https://www.xenonstack.com/insights/minio

Slingerland, C. (2023, July 5). *What Are Kubernetes Nodes? A Beginner-Friendly Guide*. Retrieved April 2024, from Cloudzero: https://www.cloudzero.com/blog/kubernetes-nodes/

Srivastava, V. (2023, March 8). *API Rate Limiting vs. API Throttling: How Are They Different?* Retrieved July 2024, from Nordic APIs: https://nordicapis.com/api-rate-limiting-vs-api-throttling-how-are-they-different/

Stalwart Labs Ltd. (n.d.). *Getting started*. Retrieved June 2024, from Stalwart Documentation: https://stalw.art/docs/get-started

Susnjara, S., & Smalley, I. (2024, June 6). *Topics: What is docker?* Retrieved June 2024, from IBM: https://www.ibm.com/topics/docker

The Kubernetes Authors. (n.d.). *Concepts: Kubernetes Components*. Retrieved April 2024, from Kubernetes: https://kubernetes.io/docs/concepts/overview/components/

The Kubernetes Authors. (n.d.). *Concepts: Service*. Retrieved March 2024, from Kubernetes: https://kubernetes.io/docs/concepts/services-networking/service/

The Kubernetes Authors. (n.d.). *Concepts: Workloads*. Retrieved from Kubernetes: https://kubernetes.io/docs/concepts/workloads/

The Kubernetes Authors. (n.d.). *Kubernetes Components*. Retrieved April 2024, from Kubernetes Documentations.

The Kubernetes Authors. (n.d.). *Kubernetes Concepts Overview*. Retrieved April 2024, from Kubernetes Documentation: https://kubernetes.io/docs/concepts/overview/

The Kubernetes Authors. (n.d.). *Objects in Kubernetes: Namespaces*. Retrieved from Kubernetes: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

The Kubernetes Authors. (n.d.). *Storage: volumes*. Retrieved March 2024, from Kubernetes Documentation: https://kubernetes.io/docs/concepts/storage/volumes/

Tschabitscher, H. (2019, December 19). *What Is an Email Client?* Retrieved May 2024, from Lifewire: https://www.lifewire.com/what-is-an-email-client-1171116

Varghese, L., Jacob, K., & Supriya, M. (2017, April 30). Spam: A Big Data Challenge. *International Journal of Advanced Research in Computer Science, 8*(1), 195-198.

What is ansible? (n.d.). Retrieved March 2024, from opensource.com: https://opensource.com/resources/what-ansible

Yugabyte, Inc. (n.d.). *Key benefits*. Retrieved June 2024, from YugabyteDB documentation: https://docs.yugabyte.com/preview/features/

Yugabyte, Inc. (n.d.). *Comparisons / PostgreSQL*. Retrieved May 2024, from YugabyteDB documentation: https://docs.yugabyte.com/preview/faq/comparisons/postgresql/