



UNIVERZA
V LJUBLJANI

FRI

Fakulteta za računalništvo
in informatiko

Laboratorij za integracijo informacijskih sistemov

Referenčna arhitektura aplikacijskih rešitev za DRO

Tehnološke zahteve, arhitekturni koncepti in
smernice domorodne oblačne arhitekture

Naročnik: Republika Slovenija, Ministrstvo za digitalno preobrazbo

29.11.2023

Vsebina:	Referenčna arhitektura aplikacijskih rešitev za DRO
Tema:	Tehnološke zahteve, arhitekturni koncepti in smernice domorodne oblačne arhitekture
Ključne besede:	Oblak, arhitektura, domorodna oblačna arhitektura, smernice
Datum nastanka:	29. 11. 2023, datum zadnje spremembe 31. 3. 2025
Verzija:	1.30
Status dokumenta:	Končna verzija
Avtor:	Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, Laboratorij za integracijo informacijskih sistemov Vodja projekta: prof. dr. Matjaž B. Jurič

KAZALO VSEBINE

1	UVOD	6
2	ARHITEKTURNI KONCEPTI IN SMERNICE DOMORODNIH OBLAČNIH APLIKACIJ	7
2.1	DOMORODNA OBLAČNA ARHITEKTURA	7
2.2	MIKROSTORITVE	7
2.3	KOMUNIKACIJA MED MIKROSTORITVAMI	9
2.3.1	<i>API-ji in komunikacija preko API prehodov</i>	9
2.3.2	<i>Sporočilni sistemi</i>	11
2.3.3	<i>Sistemi za pretočne dogodke</i>	11
2.4	VSEBNIKI IN OKOLJA ZA ORKESTRACIJO VSEBNIKOV	12
2.5	DEVOPS IN AVTOMATIZACIJA INFRASTRUKTURE	14
2.6	RAČUNALNIŠKI OBLAK S PODPORNIMI STORITVAMI	15
3	TIPI APLIKACIJ, ARHITEKTURA IN STORITVE V DRO	16
3.1	TIPI APLIKACIJ	16
3.1.1	<i>Aplikacije, ki v celoti izkoriščajo domorodno oblachno arhitekturo</i>	17
3.1.2	<i>Aplikacije, ki delno izkoriščajo domorodno oblachno arhitekturo</i>	17
3.1.3	<i>Monolitne aplikacije, ki ne sledijo domorodni oblachni arhitekturi</i>	18
3.1.4	<i>Aplikacije, ki jih ne moremo razvrstiti v te tri tipe</i>	18
3.2	VISOKONIVOJSKA ARHITEKTURNA DRO IZ VIDIKA RAZVIJALCEV APLIKACIJ	18
3.3	ARHITEKTURNA SLIKA DRO ZA DOMORODNE OBLAČNE APLIKACIJE	20
3.3.1	<i>Izvajalno okolje Kubernetes</i>	21
3.3.2	<i>Komunikacija med mikrostoritvami</i>	22
3.3.3	<i>Varnost, avtentikacija in avtorizacija</i>	22
3.3.4	<i>Uporaba platformskih storitev oz. gradnikov</i>	23
3.3.4.1	Centralni sistem za zbiranje dnevniških zapisov	23
3.3.4.2	Centralni konfiguracijski strežnik, etcd	23
3.3.4.3	Sistem za zbiranje metrik Prometheus in Grafana	24
3.3.5	<i>DevOps in CI/CD</i>	24
3.3.5.1	Usmeritve za uporabo repozitorija git	24
3.3.5.2	Avtomatizacija CI/CD	24
3.3.5.3	Avtomatizirana vzpostavitev okolja z uporabo IaC	25
3.3.6	<i>Uporaba centralnih podatkovnih baze in shramb podatkov</i>	25
3.4	OKOLJA, KI JIH PONUJA DRO	26
4	ARHITEKTURNO TEHNOLOŠKE ZAHTEVE ZA DOMORODNE OBLAČNE APLIKACIJE	27
4.1	ZAHTEVE ZA APLIKACIJE, KI V CELOTI IZKORIŠČAJO DOMORODNO OBLAČNO ARHITEKTURO	27
4.1.1	<i>Arhitekturna zasnova in sloji</i>	27
4.1.2	<i>Mikrostoritvena arhitektura</i>	28
4.1.3	<i>Komunikacija med storitvami in sloji</i>	29
4.1.4	<i>Tehnologije API-jev</i>	30
4.1.5	<i>API prehod in sistem za upravljanje API-jev</i>	30
4.1.6	<i>Verzioriranje</i>	31
4.1.7	<i>Uporaba principov 12 faktorskih aplikacij</i>	31
4.1.8	<i>Preverjanje vitalnosti (Health Check)</i>	33
4.1.9	<i>Zbiranje metrik</i>	33
4.1.10	<i>Odprta telemetrija (Open Telemetry)</i>	34
4.1.11	<i>Odpornost na napake (Fault Tolerance)</i>	35

4.1.12	Uporaba storitvenega omrežja (Service Mesh)	36
4.1.13	Vzorci uporabe podatkovnih baz	36
4.1.14	Orodja za migracijo shem	37
4.1.15	Centralni avtentikacijski in avtorizacijski (IAM) strežnik	37
4.1.16	Spletni uporabniški vmesniki	38
4.1.17	Centralni konfiguracijski strežnik	38
4.1.18	Izvajanje na okolju Kubernetes	40
4.1.19	Avtomatizirana vzpostavitev okolja z uporabo IaC	40
4.2	ZAHTEV ZA APLIKACIJE, KI DELNO IZKORIŠČAJO DOMORODNO OBLAČNO ARHITEKTURO	41
4.2.1	Arhitekturna zasnova in sloji	41
4.2.2	Komunikacija med storitvami in sloji	41
4.2.3	API prehod in sistem za upravljanje API-jev	42
4.2.4	Verzioriranje	42
4.2.5	Uporaba principov 12 faktorskih aplikacij	42
4.2.6	Vzorci uporabe podatkovnih baz	42
4.2.7	Spletni uporabniški vmesniki	42
4.2.8	Izvajanje na okolju Kubernetes	42
4.3	SKUPNE ZAHTEV ZA DOMORODNE OBLAČNE APLIKACIJE	43
4.3.1	Avtomatiziran postopek gradnje aplikacije (build)	43
4.3.1.1	Avtomatizirani testi enot in integracijski testi	43
4.3.1.2	Generiranje tehnične dokumentacije	44
4.3.2	Pakiranje v slike vsebnikov	45
4.3.3	Namestitev na Kubernetes	46
4.3.4	Centralni sistem za beleženje dnevnških zapisov	47
4.3.5	Smernice za uporabo repozitorija git	48
4.3.6	Avtomatiziran CI/CD in gradnja slik vsebnikov	49
4.3.7	Centralni register slik vsebnikov	50
4.3.8	Centralni repozitorij programskih artefaktov	50
4.3.9	Pravila za podatkovne baze (vzeto iz obstoječega GTZ)	50
4.3.10	Ostala priporočila	51
4.3.10.1	Varnostni vidiki	51
4.3.10.2	Varnostno preverjanje kode	52
4.3.10.3	Obremenilni testi	52
4.3.10.4	Preverjanje kakovosti kode	52
4.4	ZAHTEV ZA MONOLITNE APLIKACIJE, KI NE SLEDIJO DOMORODNI OBLAČNI ARHITEKTURI	52
4.5	ZBIRNA TABELA ZAHTEV PO TIPIH APLIKACIJ	52
5	TEHNOLOŠKE ZAHTEV, PROGRAMSKI JEZIKI, ORODJA IN OGRODJA	55
5.1	PROGRAMSKI JEZIKI IN OGRODJA ZA RAZVOJ	55
5.1.1	Podprti programski jeziki za zaledni del	55
5.1.2	Podprta programska ogrodja za spletne aplikacije (spletne uporabniške vmesnike)	56
5.1.3	Podprta programska ogrodja za mobilne aplikacije	56
5.1.4	Odlaganje artefaktov	57
5.2	PODATKOVNE BAZE IN TRAJNO STANJE PODATKOV	57
5.2.1	Licenčni relacijski sistemi za upravljanje podatkovnih baz	57
5.2.2	Odpri tokodni relacijski sistemi za upravljanje podatkovnih baz	57
5.2.3	Nerelacijski sistemi za upravljanje podatkov in objektne zbirke	57
5.2.4	Ostale podatkovne baze in podatkovne zbirke	57
5.3	TEHNOLOGIJE ZA APLIKACIJSKE PROGRAMSKE VMESNIKE - API	58
5.4	SPOROČILNI IN DOGODKOVNI SISTEMI	58

5.5	IZVAJALNA OKOLJA ZA APLIKACIJE	59
5.5.1	<i>Izvajalna okolja</i>	59
5.5.2	<i>Tehnologije vsebnikov in priprava slik vsebnikov</i>	60
5.5.3	<i>Nameščanje aplikacij</i>	60
5.5.4	<i>Vzpostavitev okolja – IaC in GitOps</i>	60
5.6	REPOZITORIJ IZVORNE KODE, REPOZITORIJ ARTEFAKTOV, DEVOPS IN AVTOMATIZACIJA CI/CD	60
5.6.1	<i>Repozitorij izvirne kode</i>	60
5.6.2	<i>Centralni repozitorij vsebnikov/slik in artefaktov</i>	61
5.6.3	<i>Avtomatizacija CI/CD in DevOps</i>	61
5.7	PLATFORMSKE STORITVE DRO	62
5.7.1	<i>Centralni avtentikacijski in avtorizacijski (IAM) strežnik</i>	62
5.7.2	<i>Centralni sistem za beleženje dnevniških zapisov</i>	62
5.7.3	<i>API prehod in sistem za upravljanje API-jev</i>	62
5.7.4	<i>Centralni konfiguracijski strežnik</i>	63
6	SMERNICE ZA MIGRACIJO MONOLITNIH APLIKACIJ V DOMORODNO OBLAČNO ARHITEKTURO	64
6.1	RAZUMEVANJE OMEJITEV OBSTOJEČE APLIKACIJE IN NAČRTOVANJE MIGRACIJE	64
6.2	SPREJEM ODLOČITVE, ALI SE PREDELUJE OBSTOJEČO APLIKACIJO ALI GRE V NOV RAZVOJ	64
6.3	PREDELAVA OBSTOJEČE APLIKACIJE	64
6.3.1	<i>Postopna predelava</i>	64
6.3.2	<i>Celostna tehnološka posodobitev</i>	65
7	ZAKLJUČEK	66
	VIRI IN REFERENCE	67

KAZALO SLIK

SLIKA 1:	PRIMERJAVA KLASIČNE VEČSLOJNE IN MIKROSTORITVENE ARHITEKTURE	9
SLIKA 2:	DIREKтна KOMUNIKACIJA DO MIKROSTORITEV	10
SLIKA 3:	UPORABA API PREHODA MED ZUNANJIMI ODJEMALCI IN ZALEDNIMI MIKROSTORITVAMI	11
SLIKA 4:	TIPI APLIKACIJ V DRO	16
SLIKA 5:	VISOKONIVOJSKA ARHITEKTURNA DRO IZ VIDIKA RAZVIJALCEV APLIKACIJ	19
SLIKA 6:	SHEMATSKI PRIKAZ IZVAJALNIH OKOLIJ IN NEKATERIH STORITEV V DRO KUBERNETES	21
SLIKA 7:	POENOSTAVLJENA ARHITEKTURNA ZASNOVA TIPIČNE APLIKACIJE	28
SLIKA 8:	TESTIRANJE ENOT, MOCKANJE IN INTEGRACIJSKO TESTIRANJE	43
SLIKA 9:	ZBIRNA TABELA ZAHTEV PO TIPIH APLIKACIJ	52

1 Uvod

Dokument podaja tehnološke zahteve, arhitekturne koncepte in smernice referenčne arhitekture aplikacijskih rešitev za obstoječi Državni računalniški oblak (DRO) ter njegove naslednike (DRO-NEXT), ki temeljijo na sodobnih principih oblačnega računalništva.

Dokument je namenjen razvijalcem aplikacijskih rešitev oz. programskih sistemov, ki se bodo izvajali v DRO oz. DRO-NEXT. Dokument podaja tehnološke zahteve, arhitekturne koncepte in smernice ter specificira nabor podprtih tehnologij, ki jih je potrebno upoštevati pri razvoju aplikacijskih rešitev.

Dokument naslavlja tri različne nivoje arhitekture aplikacijskih rešitev, glede na njihovo skladnost z domorodno oblačno arhitekturo (cloud-native arhitekturo):

- Aplikacije, ki v celoti izkoriščajo domorodno oblačno arhitekturo (DRO-NEXT),
- Aplikacije, ki delno izkoriščajo domorodno oblačno arhitekturo (in le-te aplicirajo samo najpomembnejše vzorce),
- Monolitne aplikacije, torej aplikacije, ki ne sledijo domorodni oblačni arhitekturi.

Ta dokument podaja ključne arhitekturne in tehnološke smernice in zahteve za naslednjo generacijo domorodnih oblačnih aplikacij za DRO. Dokument se osredotoča predvsem na smernice domorodne oblačne arhitekture in računalništva v oblaku in ne podaja ostalih, splošnih smernic, vzorcev in dobrih praks programskega inženirstva ter načrtovanja aplikacijskih sistemov. Predpostavlja se, da so bralci s temi koncepti seznanjeni. Dokument prav tako ne naslavlja varnostnih konceptov, kakor tudi ne projektnih in organizacijskih vidikov ter ostalih aspektov, ki so naslovljeni v ločnih dokumentih.

2 Arhitekturni koncepti in smernice domorodnih oblčnih aplikacij

To poglavje podaja povzetek arhitekturnih konceptov in smernic domorodnih oblčnih aplikacij za potrebe razvoja aplikacijskih rešitev. Poglavje se osredotoča na domorodne oblčne arhitekture, koncepte in najboljše prakse. Pričakuje se, da bodo razvijalci aplikacijskih rešitev seznanjeni s splošno sprejetimi praksami programskega inženirstva, vzorci in dobrimi praksami. Za podrobno seznanjanje z domorodno oblčno arhitekturo in koncepti naj bralci posežejo po dodatnih virih in literaturi.

2.1 Domorodna oblčna arhitektura

Definicija domorodne oblčne arhitekture:

Domorodna oblčna arhitektura in tehnologije so pristop k načrtovanju, gradnji in delovanju aplikacijskih rešitev, ki so specifično zgrajene za delovanje v oblaku in v celoti izkoriščajo model računalništva v oblaku.

Cloud Native Computing Foundation ponuja uradno definicijo:

Domorodna oblčna arhitektura omogočajo gradnjo in izvajanje skalabilnih aplikacij v sodobnih, dinamičnih okoljih, kot so javni, zasebni in hibridni oblaki. Vsebniki, storitvena omrežja (service mesh), mikrostoritve, nespremenljiva infrastruktura in deklarativni API-ji ponazarjajo ta pristop. Te tehnike omogočajo razvoj šibko sklopljenih sistemov, ki so prožni, obvladljivi, odporni na napake in sledljivi. V kombinaciji z robustno avtomatizacijo inženirjem omogočajo pogosto in predvidljivo izvajanje sprememb z minimalnim naporom.

»Cloud Native« arhitektura naslavlja hitrosti in agilnost razvoja in vzdrževanja aplikacijskih rešitev.

Ključni gradniki domorodne oblčne arhitekture so:

- mikrostoritve (in brezstrežniške funkcije¹),
- API-ji, sporočila in pretočni dogodki,
- vsebniki in okolja za orkestracijo vsebnikov,
- DevOps in avtomatizacija infrastrukture,
- računalniški oblak s podpornimi storitvami.

2.2 Mikrostoritve

Mikrostoritve so arhitekturni koncept oblikovanja in razvoja programske opreme, ki temelji na razdelitvi aplikacije na majhne, neodvisne in sodelujoče storitve. Vsaka storitev je odgovorna za določeno funkcionalnost ali poslovni proces in ima svoj lasten življenjski cikel, tehnološki sklad, svojo

¹ DRO trenutno ne podpira brezstrežniških funkcij. Podporo bo ponudil kasneje.

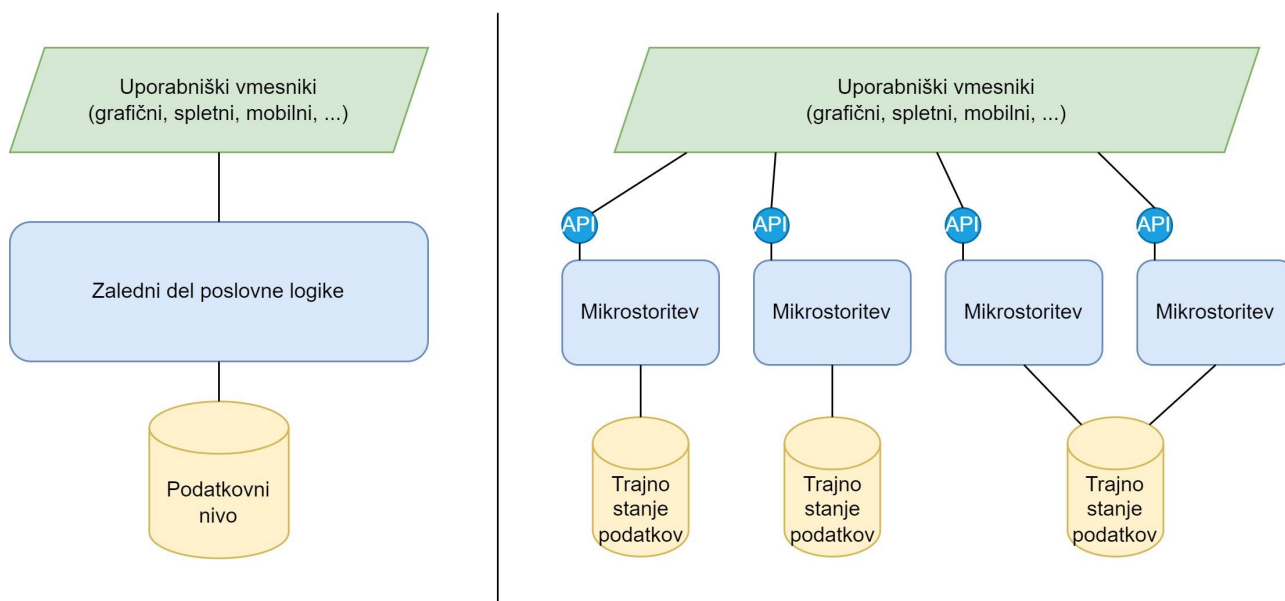
ali deljeno podatkovno shemo, uporablja API-je, sporočila ali pretočne dogodke za komunikacijo. Pomemben izziv pri načrtovanju in implementaciji mikrorstitev je kompleksnost, ki izhaja iz povezav in odvisnosti med mikrorstitevami. Slednje moramo skrbno načrtovati, zmanjšati število odvisnosti ter avtomatizirati postopke vzdrževanja odvisnosti.

Mikrorstitevna arhitektura je *arhitekturni pristop, ki poudarja razgradnjo aplikacij na šibko sklopljene specializirane storitve, katere lahko obvladljivo razvijajo in posodablajo multifunkcijske razvojne skupine, za zagotavljanje kakovosti in kratkih razvojnih ciklov razvoja in vzdrževanja kompleksnih sistemov v skladu z zahtevami današnjega digitalnega poslovanja.*

Mikrorstitevna arhitektura ima naslednje ključne značilnosti:

- Vsaka mikrorstitev implementira specifično funkcionalnost konteksta širše domene.
- Vsaka mikrorstitev je razvita samostojno, ima svoj življenjski cikel in jo je mogoče namestiti neodvisno.
- Vsaka mikrorstitev se izvaja v svojem procesu in komunicira z drugimi z uporabo standardnih komunikacijskih protokolov:
 - vmesnikov API (kot npr. REST API, gRPC, GraphQL),
 - pretočnih dogodkov, kot npr. Kafka,
 - sporočilnih sistemov, kot npr. AMQP ali NATS.
- Mikrorstitev skupaj sestavljajo in oblikujejo aplikacijo oz. aplikacijski sistem.
- Storitve so zasnovane s čim manj medsebojnimi odvisnosti in čim bolj samostojne.
- Storitve so zasnovane s ciljem zanesljivega in robustnega delovanja in implementirajo vzorce odpornosti na napake.
- Sledijo konceptom visoke kohezije in nizke sklopljenosti.
- Temeljijo na konceptu delovanja brez stanja (stateless) in podpirajo koncept horizontalnega skaliranja.
- Kompleksnost povezovanja in interakcije mikrorstitev naslavljam z avtomatizacijo gradnje in avtomatiziranim nameščanjem (CI/CD) ter opsijsko uporabo storitvenih omrežij (service mesh, kot npr. Istio).

Slika 1 prikazuje poenostavljeno primerjavo klasične večslojne in mikrorstitevne arhitekture.



Slika 1: Primerjava klasične večslojne in mikrororitvene arhitekture

2.3 Komunikacija med mikrororitvami

Komunikacija med mikrororitvami in med odjemalci in mikrororitvami poteka preko vmesnikov API, sporočil ali dogodkov. Uporablja različne vzorce izmenjave sporočil, kot npr. sinhrono in asinhrono, zahteva/odgovor, enosmerno, spodbujene odgovore, pretočno (streaming) komunikacijo in podobno. Tipično sledi različnim interakcijskim vzorcem, predvsem:

- Zahteva/odgovor ali poizvedba – mikrororitev, ki kliče drugo mikrororitev, zahteva odgovor klicane mikrororitve, na primer »vrni podatke o računu«.
- Ukazi brez odgovora ali enosmerna komunikacija – mikrororitev kliče drugo mikrororitev za izvedbo aktivnosti, vendar ne zahteva odgovora, na primer »pošlji osebni dokument po pošti«.
- Dogodki – mikrororitev sproži dogodek, katerega stanje se je spremenilo ali je prišlo do dejanja. Druge mikrororitve, naročniki, ki jih to zanima, se na dogodek ustrezno odzovejo. Sprožitelji dogodkov in naročniki na dogodke se ne zavedajo drug drugega.

Komunikacija med mikrororitvami poteka izključno preko:

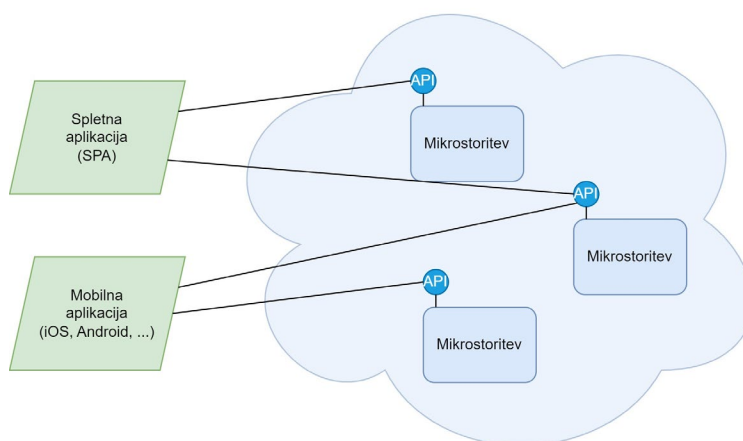
- API-jev,
- sporočilnih sistemov ali
- pretočnih dogodkov.

2.3.1 API-ji in komunikacija preko API prehodov

API-ji oz. aplikacijski programski vmesniki (application programming interfaces) so prevladujoč način komunikacije med mikrororitvami in med aplikacijskimi sistemi širše. API definira skupek operacij in podatkovnih tipov, ki omogoča dostop do funkcionalnosti mikrororitve. Predstavlja neke vrste

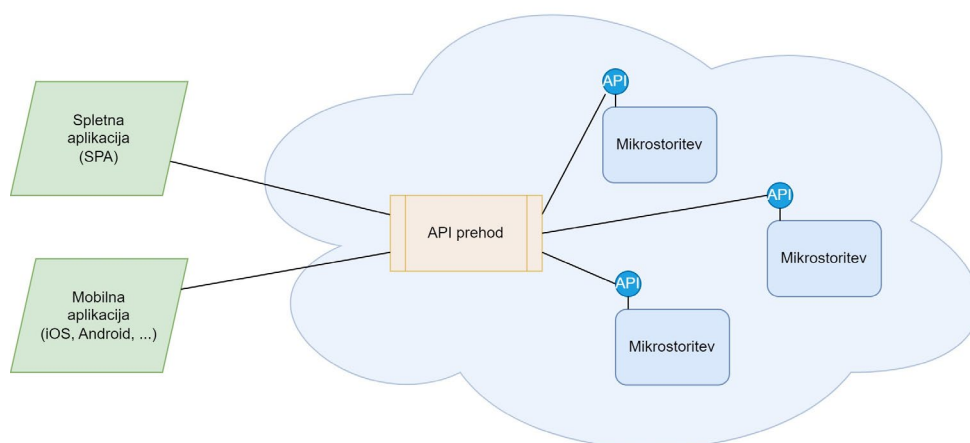
dogovor med mikrostoritvijo in njenimi klicatelji (odjemalci). API-ji poenostavljajo razvoj programske opreme, saj omogočajo enostavno in varno izmenjavo podatkov in funkcionalnosti med aplikacijami. Uporabljajo se za komunikacijo med storitvami, ki delujejo na različnih platformah, lokacijah, napravah itd. API-ji morajo biti dokumentirani in opisani, da lahko razvijalci in uporabniki razumejo, kako jih uporabljati in kaj pričakovati od njih. Razširjene tehnologije API-jev so REST, GraphQL in gRPC ter iz preteklosti še SOAP/WSDL. Tipično API-je uporabljamo za realizacijo vzorca izmenjave zahteva/odgovor, pogosto na sinhroni način. API-ji sicer omogočajo tudi realizacijo asinhronih in pretočnih vzorcev izmenjave.

V domorodnih oblačnih aplikacij potrebujejo odjemalci (uporabniški vmesniki kot npr. spletne aplikacije, mobilne aplikacije ali drugi sistemi) komunikacijski kanal do mikrorstitev. Direktna komunikacija do mikrorstitev je sicer možna rešitev, vendar odsvetovana, saj so na ta način mikrorstitevne direktno izpostavljene odjemalcem, kar zmanjšuje fleksibilnost sistema in odpira določene varnostne pomanjkljivosti, kot npr. izpostavljenost notranje arhitekture sistema, slabša zmožnost zaščite pred preobremenitvijo in omejevanje prometa (Rate Limiting), otežena vzpostavitev enotne varnostne politike, sledenja uporabi in centralnega nadzora uporabe API-jev. Poenostavljeno shemo prikazuje naslednja slika 2:



Slika 2: Direktna komunikacija do mikrorstitev

Splošno sprejeti komunikacijski vzorec je uporaba API prehoda med zunanji odjemalci (uporabniškimi vmesniki ali drugimi aplikacijami) in zalednimi mikrorstivami. To prikazuje naslednja slika 3:



Slika 3: Uporaba API prehoda med zunanji odjemalci in zalednimi mikrostoritvami

Storitev API prehoda abstrahira zaledne mikrostoritve in usmerja dohodni in izhodni promet. API prehod izolira odjemalca pred spremembami na zaledju. Izvaja tudi določene storitve, kot so avtentikacija, predpomnjenje, omejevanje dostopa ali prometa, beleženje dnevniških zapisov, porazdeljevanje obremenitve ipd.

2.3.2 Sporočilni sistemi

Sporočilni sistemi (message queue) omogočajo realizacijo asinhrono, šibko sklopljene, na sporočilih temelječe komunikacije med mikrostoritvami. Ena mikrostoritev pošlje sporočilo, ki vsebuje informacije ali ukaze drugi mikrostoritvi, ki sporočilo sprejme in obdela, ko je na voljo za izvedbo obdelave (ne nujno takoj, obdelava se lahko izvede odloženo). Sporočila se shranjujejo v vrsti (queue ali topic), dokler niso obdelana in izbrisana. Vsako sporočilo se obdela samo enkrat (queue) ali pa sporočilo obdela večje število sprejemnikov (topic).

Primeri razširjenih sporočilnih sistemov so AMQP (RabbitMQ ali ActiveMQ) in NATS.

2.3.3 Sistemi za pretočne dogodke

Sistemi za pretočne dogodke (event streaming) omogočajo asinhrono, šibko sklopljeno komunikacijo z uporabo pretočnih dogodkov. Omogočajo oblikovanje interakcij mikrostoritev, ki temeljijo na oddajanju in sprejemanju dogodkov, kar vodi v šibko sklopljeno arhitekturo z malo odvisnostmi. Sistemi za pretočne dogodke so sestavljeni iz treh glavnih komponent: proizvajalcev (producers), ki ustvarjajo in pošiljajo dogodke, potrošnikov (consumers), ki prejemajo in obdelujejo dogodke in posrednikov (brokers), ki shranjujejo in usmerjajo dogodke. Na prvi pogled so podobni sporočilnim sistemom, vendar izkazujejo nekaj bistvenih razlik, kot npr. ohranjanje dogodkov v dogodkovnem kanalu, skalabilnost delovanja in sposobnost obdelave velikega števila sočasnih dogodkov.

Kafka je eden izmed najbolj razširjenih in zmogljivih sistemov pretočnih dogodkov, ki je v uporabi na DRO (DRO-next). Je odprtokodna platforma, ki temelji na distribuiranem in skalabilnem modelu. Kafka omogoča visoko prepustnost, nizko latenco in visoko razpoložljivost dogodkov. Kafka organizira dogodke v teme (topics), ki so logične kategorije, ki združujejo sorodne dogodke. Vsaka tema je razdeljena na particije (partitions), ki so fizične enote, ki shranjujejo dogodke. Vsaka particija ima svoj zaporedni identifikator (offset), ki označuje položaj dogodka v particiji. Vsaka tema lahko ima več proizvajalcev in potrošnikov, ki lahko pošiljajo in prejemajo dogodke iz različnih particij.

Kafka zagotavlja odpornost in zanesljivost podatkov in dogodkov z replikacijo particij na več strežnikov. Vsaka particija ima en vodilni strežnik (leader), ki je odgovoren za branje in pisanje dogodkov in več sledilnih strežnikov (followers), ki kopirajo dogodke iz vodilnega strežnika. Če vodilni strežnik odpove, lahko eden izmed sledilnih strežnikov prevzame njegovo vlogo. Replikacija zagotavlja tudi visoko razpoložljivost in zmogljivost sistema, saj lahko potrošniki berejo dogodke iz različnih replik.

Kafka ohranja dogodke v particijah za določen čas ali velikost, ki jo lahko nastavimo za vsako temo. To pomeni, da lahko potrošniki dostopajo do zgodovinskih dogodkov, ki so bili poslani pred časom, in ne samo do trenutnih dogodkov. To omogoča večjo fleksibilnost in ponovljivost procesiranja dogodkov, saj lahko potrošniki začnejo, končajo ali ponovijo branje dogodkov iz kateregakoli položaja v particiji.

Kafka omogoča, da proizvajalci in potrošniki potrdijo uspešno pošiljanje ali prejemanje dogodkov. To pomeni, da lahko proizvajalci in potrošniki vedo, ali so dogodki pravilno dostavljeni ali obdelani, in lahko ukrepajo v primeru napak ali izgub. Potrjevanje lahko nastavimo na različne ravni, odvisno od potrebe po zanesljivosti in zmogljivosti.

2.4 Vsebniki in okolja za orkestracijo vsebnikov

Vsebniki so delno izolirana okolja, v katerih se izvajajo mikrostoritve (lahko pa tudi cele aplikacije). Za razliko od navideznih računalnikov, ki poganjajo popolnoma ločene operacijske sisteme, vsebniki neposredno delijo vire z operacijskim sistemom strežnika, ki gosti vsebnike. Zaradi tega so vsebniki učinkovitejši od navideznih strežnikov in ne zahteva popolnega gostujočega operacijskega sistema.

Poleg tega so vsebniki izolirani na ravni procesa od drugih vsebnikov, pa tudi od procesov brez vsebnikov, ki se izvajajo na strežniku. Vsak vsebnik definira lastne parametre okolja. Zaradi te izolacije so vsebniki varen način izvajanja mikrostoritev.

Vsebniki predstavljajo učinkovit način za pakiranje, distribucijo in izvajanje mikrostoritev. So veliko bolj učinkoviti od klasičnih virtualnih strežnikov v smislu porabe virov, hitrosti zagona in hitrosti ustavljanja. Zato so zelo primerni za zaganjanje mikrostoritev in posebej za njihovo skaliranje v sistemih, ki potrebujejo horizontalno skalabilnost.

Vsebniki omogočajo:

- Pakiranje mikrororitv v samostojne enote, ki lahko delujejo kjerkoli, ne glede na osnovno infrastrukturo.
- Vsebniki zagotavljajo dosledno in ponovljivo okolje za vaše mikroritve, kar zagotavlja, da delujejo na enak način pri razvoju, testiranju in produkciji.
- Vsebniki olajšajo komunikacijo in integracijo mikroritv, saj izpostavljajo standardne vmesnike in protokole, do katerih je mogoče dostopati preko API-jev.
- Vsebniki povečujejo varnost in zanesljivost mikroritv, saj jih izolirajo drug od drugega in omejujejo njihov dostop do gostiteljskega sistema.

Pri načrtovanju vsebnikov za mikroritve upoštevamo vzorce in najboljše prakse, med njimi:

- Vsebniki naj bodo čim manjši in enostavni, tako da vključimo samo bistvene komponente in odvisnosti za določeno mikroritev. To bo zmanjšalo velikost, porabo virov in ter čas gradnje in zagona.
- Vsebnike gradimo po principu brez stanja (stateless). Vse podatke hranimo zunaj vsebnikov, na primer v podatkovne baze, centralne sisteme za beleženje dnevniških zapisov ipd.
- Vsebniki so nespremenljivi in ne omogočajo sprememb po tistem, ko so zgrajene/generirane njihove slike (images).
- Slike vsebnikov hranimo v registru slik vsebnikov, ki predstavlja centralno shrambo slik.
- Pri poimenovanju vsebnikov uporabljamo uveljavljene in konsistentne imenske konvencije.
- Vsebniki so prilagojeni za horizontalno skaliranje in imajo kratke zagonske čase in omogočajo hitre izklope.
- Slike vsebnikov gradimo po nivojih v plasteh, plast za plastjo. Skrbimo za ponovno uporabo plasti in za ustrezno in smiselno strukturo aplikacij in datotek, ki jih nameščamo v posamezne plasti.

Na DRO je v uporabi vsebniška tehnologija Docker.

Poleg tehnologije vsebnikov potrebujemo še okolje za orkestracijo vsebnikov. Okolje za orkestracijo vsebnikov je sistem, ki avtomatizira in poenostavlja namestitve, upravljanje, skaliranje in komunikacije med vsebniki. Prav tako skrbi za njihovo zanesljivo delovanje. Okolje za orkestracijo vsebnikov ima naslednje ključne karakteristike:

- Avtomatizacija nameščanja, zagona, nadzora, posodabljanja in ostalih aktivnosti brez ročnega posredovanja. Običajno vključuje tudi mehanizme posodabljanja, kot so tekoče posodobitve (rolling updates), modro/zeleno posodabljanje ali vzporedno nameščanje (canary release).
- Skalabilnost oz. avtomatizirano prilagajanje števila delujočih vsebnikov oz. njihovih instanc glede na potrebe in obremenitve.
- Visoko razpoložljivost oz. odpornost na napake v smislu zagotavljanja dosegljivosti in delovanja vsebnikov, tudi če pride do napak ali izpadov. Z uporabo preverjanja vitalnosti, nadzora metrik in telemetrije se lahko vsebniki samodejno obnovijo, premestijo ali nadomestijo, če pride do težav. To izboljšuje zanesljivost, varnost in kakovost storitve.

- Okolje za orkestracijo vsebnikov omogoča odkrivanje storitev oz. vsebnikov (service discovery), povezovanje in komunikacijo prek omrežja z veliko fleksibilnostjo upravljanja omrežnih dostopov, usmerjanja prometa in ostalih storitev.

Na DRO je v uporabi okolje za orkestracijo vsebnikov Kubernetes.

2.5 DevOps in avtomatizacija infrastrukture

DevOps je način razvoja in dostave programske opreme, ki temelji na tesnem sodelovanju, komunikaciji in avtomatizaciji med razvojnimi in operativnimi ekipami. DevOps omogoča hitrejša, kakovostnejša in bolj inovativna rezultata, ki ustrezajo potrebam in pričakovanjem uporabnikov. Nekatere ključne značilnosti DevOps so:

- Sodelovanje: razvojne in operativne ekipe delujejo kot enotna funkcionalna ekipa, ki si izmenjuje povratne informacije, ideje in odgovornosti skozi celoten življenjski cikel programske opreme. DevOps vključuje tudi druge deležnike, kot so varnost, skladnost, upravljanje, tveganje in poslovne ekipe.
- Avtomatizacija: avtomatizacijo faz razvojnega in dostavnega procesa, kot so kodiranje, testiranje, integracija, namestitve, nadzor in posodabljanje. Avtomatizacija omogoča zmanjšanje ročnih napak, povečanje produktivnosti ekipe, izboljšanje kakovosti programske opreme in skrajšanje časa razvoja in dostave.
- Zvezno izboljševanje: temelji na eksperimentiranju, minimiziranju napak in optimiziranju za hitrost, stroške in enostavnost dostave. Zvezna dostava omogoča pogosto in dosledno pošiljanje posodobitev, ki izboljšujejo učinkovitost in funkcionalnost aplikacijskega sistema. Neprekinjen tok novih izdaj pomeni, da ekipe nenehno uvajajo spremembe kode, ki odpravljajo napake, izboljšujejo razvojno učinkovitost in prinašajo večjo vrednost za uporabnike.

Ključni gradniki DevOps, pomembni za domorodne oblačne aplikacije, so:

- uporaba repozitorija izvorne kode git in sledenje git delovnim tokovom,
- avtomatizacija gradnje (build) aplikacij,
 - običajno uporabljena build orodja so odvisna od uporabljenega programskega jezika, primeri so Maven, Gradle, npm, itd.
- avtomatsko izvajanje testov enot in integracijskih testov,
- avtomatsko generiranje tehnične dokumentacije,
- avtomatsko pakiranje v vsebnike (Docker),
- avtomatsko objavljanje v registrih slik,
- avtomatsko nameščanje.

Poseben pomen ima v tem kontekstu tudi avtomatizacija vzpostavitve okolja z uporabo konceptov IaC (Infrastructure as Code).

Podrobno so ti koncepti, opisani z vidika DRO, predstavljeni v nadaljevanju tega dokumenta.

2.6 Računalniški oblak s podpornimi storitvami

Domorodne oblačne aplikacije v celoti izkoriščajo računalniški oblak v smislu infrastrukture kot storitve (IaaS) in model storitev v oblaku v smislu platforme kot storitve (PaaS). Infrastrukturo v oblaku z vidika razvoja aplikacijskih rešitev obravnavamo kot dostopno, samopostrežno storitev, ki nam omogoča prosto razpolaganje z infrastrukturo (v okviru omejitev porabe virov). Infrastruktura po modelu IaaS nam omogoča samopostrežbo in ustvarjanje virov, ki so zagotovljeni v kratkem času (nekaj minutah) ter njihovo spreminjanje na avtomatiziran način brez ročnega (človeškega) posega.

Še bolj pomembne so z vidika razvoja aplikacij platformske storitve PaaS, ki jih ponuja računalniški oblak. Platformske storitve ponujajo nabor programskih storitev, kot so podatkovne baze, sporočilni sistemi, sistemi za pretočne dogodke, API prehodi, sistemi za centralizirano beleženje dnevniških zapisov, centralni konfiguracijski strežniki in druge, v obliki upravljanih storitev (managed services). To pomeni, da lahko te storitve uporabljajo vse aplikacije, ki se izvajajo na oblaku kot storitev, brez, da bi jih morale nameščati, konfigurirati, vzdrževati, nadgrajevati in podobno.

Platformske storitve so izredno pomembna komponenta vsakega oblaka, saj:

- Vzpostavljajo skupno osnovo storitev, ki so na voljo vsem aplikacijam.
- Standardizirajo določene storitve, kot npr. podatkovne baze, sporočilne sisteme, sisteme pretočnih dogodkov, sisteme za zbiranje dnevniških zapisov in podobno.
- Omogočajo interoperabilnost aplikacij in vpogled v delovanje posameznih rešitev.

Zato je priporočljivo, da aplikacije, zgrajene za izvajanje v oblaku, uporabljajo platformske storitve, ki jih oblak ponuja. V nadaljevanju tega dokumenta bodo specificirane storitve, ki jih ponuja DRO.

3 Tipi aplikacij, arhitektura in storitve v DRO

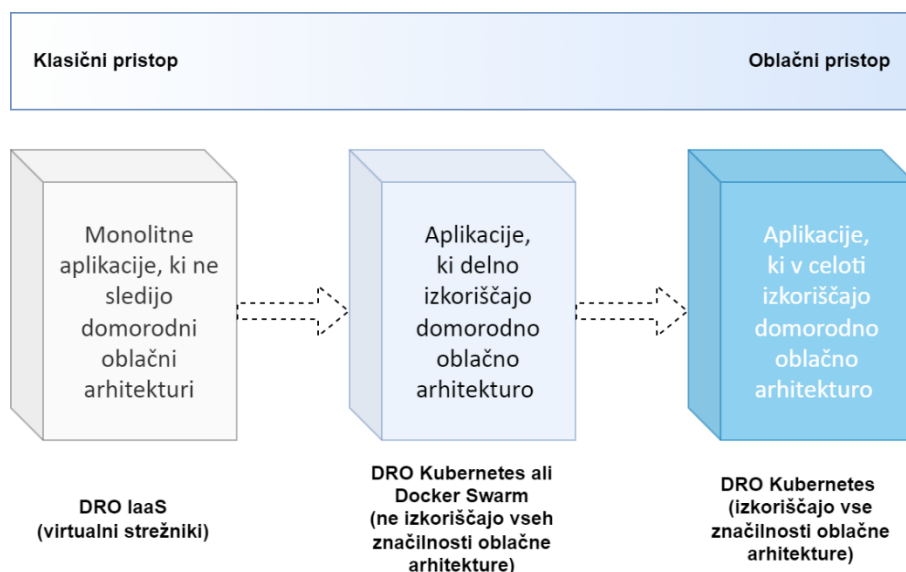
V tem poglavju so opisani tipi aplikacij, ki se izvajajo na DRO. Opisana so ponujena izvajala okolja in podan je nabor platformskih storitev (gradnikov), ki jih ponuja DRO posameznim tipom aplikacij oz. aplikacijskih sistemov.

3.1 Tipi aplikacij

Aplikacije oz. aplikacijski sistemi, ki se izvajajo v DRO, so kategorizirani v tri skupine, za katere DRO nudi natančno definirana pravila in ustrezna izvajalna okolja ter platformske storitve oz. gradnike:

- Aplikacije, ki v celoti izkoriščajo domorodno oblačno arhitekturo (DRO-NEXT),
- Aplikacije, ki delno izkoriščajo domorodno oblačno arhitekturo (in le-te aplicirajo samo najpomembnejše vzorce),
- Monolitne aplikacije, ki ne sledijo domorodni oblačni arhitekturi.

Naslednja slika 4 prikazuje tri tipe aplikacij:



Slika 4: Tipi aplikacij v DRO

Aplikacije je glede na njihove značilnosti potrebno kategorizirati v eno od naštetih skupin, pri čemer je poudarek na prvi skupini, to so aplikacijah, ki v celoti izkoriščajo domorodno oblačno arhitekturo. Predvideva se, da bo večina novo razvitih aplikacij takih, ki v celoti izkoriščajo domorodno oblačno arhitekturo. Obstoječe aplikacije pa bodo postopno migrirane v domorodno oblačno arhitekturo.

V nadaljevanju podajamo kratek opis lastnosti posameznih tipov aplikacij. Podrobneje bodo ti tipi aplikacij in njihove značilnosti opredeljene v naslednjih poglavjih.

3.1.1 Aplikacije, ki v celoti izkoriščajo domorodno oblačno arhitekturo

Aplikacije, ki v celoti izkoriščajo domorodno oblačno arhitekturo so priporočen način načrtovanja in razvoja aplikacijskih sistemov za DRO. Novo razvite aplikacij naj sledijo konceptom domorodne oblačne arhitekture. Take aplikacije morajo izpolnjevati smernice, ki so podane v tem dokumentu. Ključna arhitekturna izhodišča, na katerih temeljijo take aplikacije, so:

- Aplikacije sledijo vsem ali večini vzorcev domorodne oblačne arhitekture.
- Zasnova aplikacij je mikrororitvena (kasneje bodo podprte tudi brezstrežniške funkcije).
- Komunikacija med mikrororitvami poteka preko API, sporočilnih ali dogodkovnih sistemov.
- Mikrororitve in ostale komponente se pakirajo v vsebnike.
- Izvajalno okolje za aplikacije je Kubernetes.
- Aplikacije uporabljajo platformске storitve, ki jih ponuja DRO, kot npr. centralni sistem za zbiranje dnevniških zapisov (OpenSearch), protokol OAuth2 in sistem za avtentikacijo in avtorizacijo (Keycloak), API prehod, Kafka, konfiguracijski strežnik etcd in ostale storitve, opisane v nadaljevanju.
- Aplikacije imajo avtomatiziran cikel gradnje in CI/CD cikel, ki temelji na GitLab CI/CD.
- V okviru cikla gradnje (build) so vključeni avtomatski testi, varnostno preverjanje, preverjanje kakovosti kode in generiranje dokumentacije.
- Celotna postavitev infrastrukture in platforme, potrebne za izvajanje aplikacije, je zapisana po konceptih IaC (Infrastructure-as-Code) in GitOps.

Ciljno izvajalno okolje za ta tip aplikacije je DRO Kubernetes.

3.1.2 Aplikacije, ki delno izkoriščajo domorodno oblačno arhitekturo

Aplikacije, ki delno izkoriščajo domorodno oblačno arhitekturo so aplikacije, ki ne upoštevajo vseh značilnosti in arhitekturnih pristopov domorodnih oblačnih aplikacij, vseeno pa izkazujejo naslednje ključne lastnosti:

- Aplikacije sledijo troslojni ali storitveni arhitekturi in imajo strogo ločen uporabniški vmesnik, nivo poslovne logike in podatkovne shrambe.
- Aplikacije (delno) izpostavljajo svoje funkcionalnosti preko API (REST ali SOAP spletne storitve) ali sporočilnih sistemov.
- Lahko jih zapakiramo v vsebnike (Docker).
- Lahko jih izvajamo v okolju Kubernetes.

Ciljno izvajalno okolje za ta tip aplikacije je DRO Kubernetes. Predvideva se, da se bo ob nadgradnjah te aplikacije postopno migriralo v smer domorodne oblačne arhitekture na način, da bodo implementirale čim več značilnosti slednjih (čeprav vseh značilnosti morda ne bo mogoče doseči). Zato je ob nadgradnjah potrebno predvideti tudi tiste smiselne posege v arhitekturo aplikacij tega tipa, da se bodo čim bolj približale domorodni oblačni arhitekturi, kot je za DRO specificirana v tem dokumentu.

V prehodnem obdobju se aplikacije tega tipa lahko izvajajo tudi v DRO izvajalnem okolju Docker Swarm. Podobno, kot je v prejšnjem odstavku opisano za arhitekturo aplikacij, se tudi za izvajalna okolja predvideva, da se bodo aplikacije tega tipa postopoma selile v okolje DRO Kubernetes.

3.1.3 Monolitne aplikacije, ki ne sledijo domorodni oblačni arhitekturi

Monolitne aplikacije, ki ne sledijo domorodni oblačni arhitekturi so aplikacije, ki ne sledijo konceptom in vzorcem domorodnih oblačnih arhitektur (in jih v tem dokumentu označujemo z izrazom monolitne aplikacije). Ključna značilnost takih aplikacije je, da jih:

- Ni smiselno pakirati v vsebnike (Docker), v določenih primerih to niti ni možno.
- Ni smiselno izvajati v okolju Kubernetes.

Verjetno pa take aplikacije izkazujejo še ostale lastnosti:

- Njihova arhitektura ni troslojna oz. večslojna, ampak monolitna in ne ločuje v celoti treh ključnih slojev (uporabniškega vmesnika, poslovne logike in trajnega stanja podatkov).
- Aplikacije ne izpostavljajo programskih vmesnikov (API ali spletnih storitev) oz. drugih mehanizmov za komunikacijo.

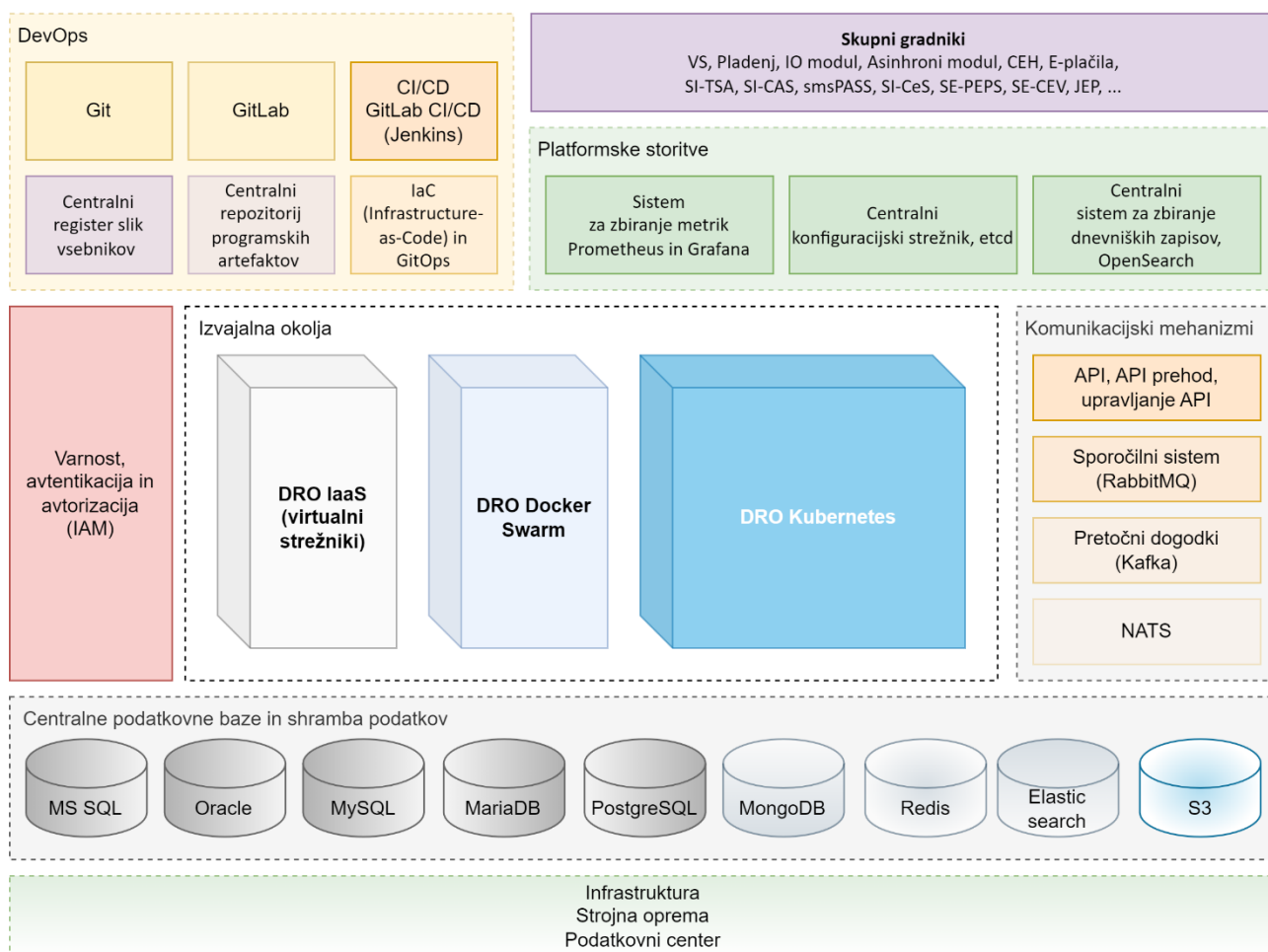
Ciljno izvajalno okolje za tak tip aplikacij so DRO virtualni strežniki (VM) oz. oblačna infrastruktura (IaaS – Infrastructure as a Service), ki jih bo DRO ohranjal za potrebe kompatibilnosti s tem tipom aplikacij. Kljub vsemu se priporoča, da se za vsako tako aplikacijo, ki deluje v DRO, izvede analiza možnosti nadgradnje oz. migracije na domorodno oblačno arhitekturo. V tem smislu se naj presodi preostala življenjska doba aplikacije, možnosti za nadgradnjo arhitekture obstoječe aplikacije in s tem povezani stroški in se na osnovi tega sprejme odločitev o nadaljnjih smernicah razvoja.

3.1.4 Aplikacije, ki jih ne moremo razvrstiti v te tri tipe

DRO dopušča tudi možnost izvajanja aplikacij, ki jih iz različnih razlogov ne moremo razvrstiti v te tri tipe. To bodo lahko aplikacije, ki izkoriščajo tehnologije ali arhitekturne stile, ki so nastali po pisanju tega dokumenta. DRO ne zapira vrat takim aplikacijam in jih bo omogočal oz. ponudil ustrezno izvajalno okolje v skladu s vsakokratno presojo arhitekture aplikacije, uporabljenih tehnologij, zmožnosti podpore v DRO in strateških ciljev, ki jih taka aplikacija naslavlja.

3.2 Visokonivojska arhitekturna DRO iz vidika razvijalcev aplikacij

Visokonivojska arhitekturna DRO iz vidika razvijalcev aplikacij je prikazana na naslednji sliki 5.



Slika 5: Visokonivojska arhitekturna DRO iz vidika razvijalcev aplikacij

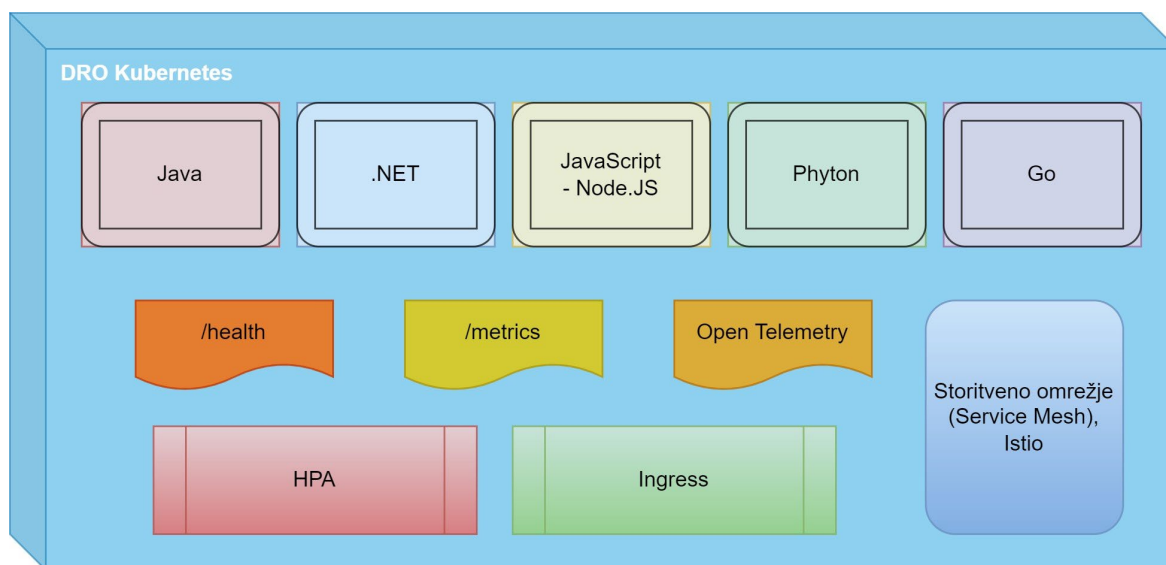
Obsega naslednje sklope:

- Izvajalna okolja:
 - DRO Kubernetes,
 - DRO Docker Swarm,
 - Virtualni strežniki oz. IaaS:
- Varnost, avtentikacijo in avtorizacijo:
 - OAuth2 in OpenID Connect,
 - Keycloak kot strežnik IAM (Identity and Access Management).
- Komunikacijske mehanizme:
 - Podporo za API-je REST, SOAP, gRPC in GraphQL,
 - Podporo za sporočilne in dogodkovne sisteme AMQP (RabbitMQ), Kafka, NATS,
 - Centralni gradnik API prehod in okolje za upravljanje API-jev,
 - Centralni gradnik Kafka, na voljo kot storitev,
 - Centralni gradnik RabbitMQ, na voljo kot storitev.
- Platformске storitve oz. gradnike:
 - Centralni sistem za zbiranje dnevniških zapisov, OpenSearch,

- Centralni konfiguracijski strežnik, etcd,
- Sistem za zbiranje metrik Prometheus in Grafana.
- DevOps in CI/CD:
 - Repozitorij izvirne kode Git in okolje GitLab,
 - *Za potrebe obstoječih aplikacij do nadaljnjega ostaja tudi podpora za SVN,*
 - CI/CD avtomatizacijo na osnovi GitLab CI/CD,
 - *Za potrebe obstoječih aplikacij do nadaljnjega ostaja tudi podpora za Jenkins,*
 - Podporo za vzpostavitev okolja (infrastrukture in platforme) z uporabo konceptov IaC (Infrastructure-as-Code) in GitOps,
 - Centralni register slik vsebnikov na osnovi Docker Harbor,
 - Centralni repozitorij programskih artefaktov na osnovi Sonatype Nexus.
- Centralne podatkovne baze in shramba podatkov:
 - Relacijske podatkovne baze:
 - Oracle RDBMS,
 - MS SQL,
 - Odprtokodni relacijski sistemi za upravljanje podatkovnih baz:
 - MySQL,
 - MariaDB,
 - PostgreSQL,
 - Nerelacijski sistemi za upravljanje podatkov in objektne zbirke, odprtokodne nerelacijske zbirke:
 - MongoDB,
 - Redis,
 - Elasticsearch,
 - S3 (Minio).
- Skupne gradnike:
 - VS, Pladenj, IO modul, Asinhroni modul, CEH, E-plačila, SI-TSA, SI-CAS, smsPASS, SI-CeS, SE-PEPS, SE-CEV, JEP in ostali gradniki.
 - Skupni gradniki so podrobno specificirani in opisani v dokumentu [Smernice MDP za razvoj informacijskih rešitev | Izdelki | Portal NIO](#)

3.3 Arhitekturna slika DRO za domorodne oblačne aplikacije

Ciljno izvajalno okolje za domorodne oblačne aplikacije v DRO je Kubernetes, ki ga shematsko prikazuje naslednja slika 6:



Slika 6: Shematski prikaz izvajalnih okolij in nekaterih storitev v DRO Kubernetes

V nadaljevanju so opisani posamezni sklopi.

3.3.1 Izvajalno okolje Kubernetes

Vse domorodne oblačne aplikacije morajo biti pripravljene za namestitev v Kubernetes. Kubernetes v DRO ponuja in podpira standardizirana izvajalna okolja za naslednje programske jezike:

- Java,
- .NET (vključuje C#, F# in VisualBasic),
- JavaScript - Node.JS,
- Phyton,
- Go.

Podrobneje so izvajalna okolja specificirana v poglavju o tehnoloških zahtevah v nadaljevanju tega dokumenta.

Za uporabo standardiziranih izvajalnih okolij ponuja DRO izhodiščne vsebniške (Docker) slike, ki jih je potrebno uporabiti kot osnovo (base image, FROM) pri generiranju slik Docker posameznih mikrostoritev in ostalih komponent aplikacije. Običajno sledijo načelu, da se v eno sliko Docker pakira ena mikrostoritev. Specifikacija (koda, Dockerfile) za generiranje slik vsebnikov mora biti del projekta izvirne kode.

Za namestitev na okolje Kubernetes mora aplikacija uporabiti namestitvene deskriptorje. Namestitvenimi deskriptorji so datoteke v formatu YAML ali JSON, ki definirajo lastnosti, vire, odvisnosti in pravila za namestitev in upravljanje aplikacije v Kubernetes. Med drugim mora specificirati način oblikovanja strokov (podov), običajno po načelu, da se ena mikrostoritev pakira v en strok. Uporabiti je potrebno imenska področja (namespaces) za ločevanje in organiziranje virov aplikacije v logične skupine.

Priporočeno je, da so aplikacije organizirane v Helm chartih, ki so paketi namestitvenih deskriptorjev, ki omogočajo modularno, parametrizirano in ponovno uporabno namestitvev aplikacije v Kubernetes. Helm charti morajo upoštevati najboljše prakse in smernice za strukturo, imenovanje, predloge, spremenljivke, funkcije, odvisnosti, testiranje in dokumentacijo.

Deskriptorji oz. Helm charti morajo biti del projekta izvorne kode. Aplikacije se nameščajo v svoja, unikatna imenska področja. Vsa konfiguracija mora biti eksternalizirana in dokumentirana. Občutljivi podatki v konfiguraciji se shranjujejo v obliki skrivnosti (secrets).

3.3.2 Komunikacija med mikrostoritvami

Komunikacija med mikrostoritvami mora potekati preko API-jev, sporočilnega ali dogodkovnega sistema. Komunikacijske porte za komunikacijo preko API mora aplikacija ustrezno izpostaviti na nivoju vsebnika in na nivoju stroka (poda). Pri tem mora uporabiti ustrezen Kubernetes mehanizem (Ingress ali Service).

Vsa komunikacija s strani odjemalcev, ki prihaja od zunaj (npr. iz uporabniškega vmesnika ali drugih zunanjih odjemalcev, ki kličejo API-je), mora biti obvezno usmerjena preko API prehoda. Vsi REST API-ji morajo biti obvezno specificirani z uporabo OpenAPI 3.x. API-je je potrebno registrirati v centralnem sistemu za upravljanje API-jev.

Za komunikacijo preko sporočilnih sistemov se predvideva uporaba centralnega gradnika RabbitMQ kot storitve.

Za komunikacijo preko dogodkovnih kanalov se predvideva uporaba centralnega gradnika Kafka kot storitve.

3.3.3 Varnost, avtentikacija in avtorizacija

Aplikacija mora uporabljati protokola OAuth2 in OpenID Connect v povezavi s strežnikom Keycloak, ki ga DRO ponuja kot centralni avtentikacijski in avtorizacijski strežnik. Aplikacija delegira prijavne in registracijske tokove avtentikacijskemu in avtorizacijskemu strežniku Keycloak. To pomeni, da se vsi ekrani za prijavo (login) in če je potrebno tudi za registracijo, implementirajo kot tokovi (flows) na strežniku Keycloak (in ne direktno v aplikaciji).

Keycloak se uporabi tudi za avtentikacijo in avtorizacijo dostopa do zaledni storitev (npr. mikrostoritev in njihovih API-jev). Priporočljiva je uporaba JWT žetonov. Par žetonov je sestavljen iz dostopnega žetona (access token) in osvežilnega žetona (refresh token). Dostopni žeton ima kratek čas trajanja in se uporablja za avtorizacijo zahtevkov do zaščitenih virov in storitev. Osvežilni žeton ima dolg čas trajanja (npr. 30 dni) in se uporablja za pridobivanje novih dostopnih žetonov, ko ti potekajo.

Aplikacija mora definirati ustrezne varnostne realme. Uporablja lahko avtorizacijo dostopa na osnovi vlog (role based) ali na osnovi virov (resource based).

3.3.4 Uporaba platformskih storitev oz. gradnikov

3.3.4.1 Centralni sistem za zbiranje dnevniških zapisov

Aplikacije morajo dnevniške zapise (loge) zbirati v predpisanem formatu DRO in jih pošiljati na centralni strežnik za zbiranje dnevniških zapisov OpenSearch, ki ga ponuja DRO. Zbiranje dnevniških zapisov v centralni sistem za zbiranje dnevniških zapisov je obvezna zahteva. Pošiljanje dnevniških zapisov na centralni strežnik OpenSearch lahko poteka na dva načina:

- Aplikacija dnevniške zapise piše v datoteke, skonfigurira se ustrezen kolektor (npr. Filebeat, ki dnevniške zapise jemlje iz datoteke in jih pošilja na OpenSearch.
 - Aplikacije, ki se izvajajo v vsebnikih, morajo poskrbeti, da se v centralni sistem za beleženje dnevniških zapisov dejansko pošljejo vsi dnevniški zapisi oz. da se v primeru avtomatskega ponovnega zagona vsebnika (ki bi ga naredil npr. Kubernetes ali Swarm) del dnevniških zapisov ne izgubi.
- Aplikacija pošilja dnevniške zapise na OpenSearch preko ustreznega mehanizma, kot npr. Logstash ali Fluentd.
 - V tem primeru mora aplikacija uporabljati takšno ogrodje za beleženje dnevniških zapisov, ki dnevniške zapise pošilja asinhrono in s tem ne vpliva pomembno na delovanje aplikacije.
 - Smiselno je tudi, da se vgradi mehanizem za odpornost v primeru, če sistem za dnevniške zapise začasno ni dosegljiv.

Uporabiti je potrebno standardni format dnevniških zapisov, ki je predpisan s strani DRO. Prav tako je potrebno zagotoviti sledljivost dnevniških zapisov skozi različne mikrostoritve oz. komponente aplikacije (z uporabo enoličnih identifikatorjev ali Open Telemetry).

3.3.4.2 Centralni konfiguracijski strežnik, etcd

Aplikacije lahko (opcijsko) uporabijo centralni konfiguracijski strežnik etcd, ki ga ponuja DRO. V centralni konfiguracijski strežnik shranjujejo nastavitve aplikacije. S tem je vzpostavljen bolj učinkovit način konfiguracije od uporabe okoljskih spremenljivk (env) oz. konfiguracijskih datotek (ki so v vsebnikih Docker samo bralne oz. read-only).

Za potrebe vpisovanja nastavitve v konfiguracijski strežnik mora aplikacija predvideti ustrezno strukturo konfiguracijskih nastavitvev (polj in vrednosti) na način, da se konfiguracija v centralni konfiguracijski strežnik zapiše enolično glede na aplikacijo oz. aplikacijski sistem. Pri tem je potrebno konfiguracijo zapisati tako, da se razlikujejo vrednosti, ki veljajo za vse instance določene storitve in vrednosti, ki naslavljajo konfiguracije specifične instance storitve (v primeru horizontalnega skaliranja).

Konfiguracijski strežnik omogoča tudi spreminjanje konfiguracijskih vrednosti med delovanjem aplikacije oz. posamezne storitve, zato je smiselno, da razvijalec aplikacije predvidi možnost spreminjanja konfiguracijskih nastavitvev med delovanjem aplikacije (in ne zgolj pri zagonu).

V prihodnosti bo centralni konfiguracijski strežnik DRO razširjen še s funkcionalnostjo označevanja funkcionalnosti (Feature Flags).

3.3.4.3 Sistem za zbiranje metrik Prometheus in Grafana

Aplikacije lahko izpostavljajo podatke o metrikah delovanja, ki se nato pošiljajo v sistem Prometheus, ki ga ponuja DRO. Predvideno je, da bodo metrike izpostavljale predvsem domorodne oblačne aplikacije v okolju Kubernetes. Izpostavljanje metrik je opsijsko.

Aplikacija mora za potrebe izpostavljanja metrik imeti HTTP končno točko za metrike, ki jo bo klical Prometheus. Predviden je način pridobivanja »potegni« (pull). V kolikor takega načina ni možno realizirati, ponuja DRO tudi pristop z uporabo komponente push-gateway.

Aplikacija mora uporabljati standardni format, tipe in orodja za metrike, ki jih podpira Prometheus. Prav tako mora aplikacija upoštevati varnostne in zmogljivostne smernice za pošiljanje metrik.

3.3.5 DevOps in CI/CD

3.3.5.1 Usmeritve za uporabo repozitorija git

Predvidena je uporaba repozitorija izvirne kode git. Priporočeno je, da so posamezne komponente/mikrostoritve aplikacije v ločenih repozitorijih. Tudi poimenovanje repozitorijev naj sledi dogovorjenim imenskim konvencijam in pravilom.

DRO ne predpisuje uporabe določenega git delovnega toka (workflow), tako da ga lahko izberejo razvijalci aplikacij po lastni izbiri. Kljub temu se priporoča uporaba delovnega toka *Trunk-Based Development*.

Kot del izvirne kode je potrebno predati tudi:

- Kodo za gradnjo slik vsebnikov (Docker) – npr. Dockerfile,
- Namestitvene deskriptorje za Kubernetes in Helm charte.

Opomba: Uporaba obstoječega repozitorija SVN je namenjena izključno za obstoječe (legacy) aplikacije, za katere pa se priporoča, da se jih postopno migrira na git.

3.3.5.2 Avtomatizacija CI/CD

Vsi projekti aplikacij morajo imeti definiran avtomatiziran cevovod za CI/CD, v okviru katerega se zgradi (build) aplikacija, artefakti se odložijo na repozitorij artefaktov, izvedejo se testi enot, integracijski testi in generiranje dokumentacije, slike vsebnikov se odložijo na register slik vsebnikov ter pripravijo vsi ostali potrebni elementi, ki so potrebni za namestitev (namestitveni deskriptorji, konfiguracije ipd.).

Opcijsko lahko CI/CD cevovod vključuje tudi (avtomatizirano) namestitev na testno okolje, ustrezne posodobitve okolja, konfiguracijo v centralnem konfiguracijskem strežniku (če se uporablja), registracija virov, kot npr. registracija API-jev na API prehodu in podobno, objava tehnične dokumentacije in ostale korake.

Aplikacije definirajo cevovod v.gitlab-ci.yml datoteki, s katero specificirajo celoten postopek CI/CD. Pri tem naj sledijo dobrim praksam in smernicam okolja DRO, uporabijo pa lahko tudi predloge in okolja, ki jih ponuja DRO.

3.3.5.3 Avtomatizirana vzpostavitev okolja z uporabo IaC

Za avtomatizacijo vzpostavitve okolja (infrastrukture), ki ga potrebuje aplikacija, ponuja DRO podporo za IaC (Infrastructure-as-Code). IaC je postopek, kjer uporabljamo programsko kodo za definicijo in vzpostavitev okolja aplikacije. Za avtomatizacijo postopkov IaC se uporablja Terraform/OpenTofu ter Ansible. Prvi je bolj namenjen sami vzpostavitvi okolja, drugi pa konfiguraciji aplikacijskih rešitev. Vzpostavitev infrastrukture DRO izvaja sistemska ekipa upravljalca infrastrukture DRO.

3.3.6 Uporaba centralnih podatkovnih baze in shramb podatkov

Aplikacije morajo prvenstveno uporabljati centralne podatkovne baze in shrambe podatkov, ki jih ponuja DRO in sicer:

- Relacijske podatkovne baze:
 - Oracle RDBMS,
 - MS SQL.
- Odprtokodni relacijski sistemi za upravljanje podatkovnih baz:
 - MySQL,
 - MariaDB,
 - PostgreSQL.
- Nerelacijski sistemi za upravljanje podatkov in objektne zbirke, odprtokodne nerelacijske zbirke:
 - MongoDB,
 - Redis,
 - Elasticsearch,
 - S3 (Minio).

Vzpostavljanje lastnih instanc podatkovnih baz (v okviru aplikacije oz. aplikacijskega sistema) in uvajanje dodatnih sistemov podatkovnih baz ali shramb podatkov ni priporočeno in naj bo izjema, ki mora biti utemeljena in jo mora potrditi DRO.

Pri uporabi relacijskih podatkovnih baz je obvezna uporaba orodij za migracijo shem, ki se jih lahko vključi v avtomatiziran postopek gradnje (builda), včasih imenovan tudi Database-as-code. Primeri takih orodij so Liquibase, Flyway, Sqitch ali Atlas.

Pri oblikovanju dostopov do baze je zahteva upravljalca infrastrukture DRO, da se aplikacija/modul na bazo prijavlja z računom (account/uporabnik) s tistim minimalnim naborom pravic, ki aplikaciji še omogoča delovanje oziroma izvrševanje poslovnih funkcij, ki jih aplikacija implementira. Direktne

povezav do baz v kodi ni dovoljeno vzpostavljati, vedno je potrebno iti preko koncepta bazena povezav (connection pool).

3.4 Okolja, ki jih ponuja DRO

DRO ponuja naslednja okolja:

- Testno okolje,
 - Testno okolje je namenjeno potrditvenemu testiranju pred prehodom v produkcijo (torej preverjanju, ali je bila naročena aplikacija oz. popravek izveden v skladu z željami naročnika (samo regresijsko testiranje pravilnosti kode se izvaja na strani izvajalca).
- UAT okolje,
 - UAT je šolsko (uvajalno) okolje, namenjeno šolanju uporabnikov in naj bi bilo po verzijah aplikacij izenačeno s produkcijskim.
- Produkcijsko okolje,
 - Produkcijsko okolje je namenjeno produkcijskemu izvajanju aplikacij in rešitev. Sem se nameščajo samo popravki, katerih prehod iz testa na produkcijo je bil po predpisanem protokolu odobren. Je visokorazpoložljivo okolje z dejanskimi, živimi podatki in aplikacijami.

Vsa tri okolja so arhitekturno enaka, razlikujejo se v kapacitetah in visoki razpoložljivosti. Razvijalci aplikacij lahko predpostavljajo, da delovanje aplikacije v testnem okolju implicira tudi delovanje aplikacije v ostalih okoljih.

4 Arhitekturno tehnološke zahteve za domorodne oblačne aplikacije

V tem poglavju so specificirane splošne tehnološke zahteve za domorodne oblačne aplikacije. Poglavje je razdeljeno v podpoglavja, ki opisujejo:

- Zahteve za aplikacije, ki v celoti izkoriščajo domorodno oblačno arhitekturo,
 - V tem poglavju so definirane zahteve za aplikacije, ki jih označujemo kot domorodne oblačne aplikacije in izpolnjujejo vse ključne zahteve takih aplikacij.
- Zahteve za aplikacije, ki delno izkoriščajo domorodno oblačno arhitekturo,
 - V tem poglavju so definirane zahteve za aplikacije, ki sicer ne sledijo vsem smernicam domorodne oblačne arhitekture, vseeno pa udeležujejo najpomembnejše vidike.
- Skupne zahteve,
 - V tem poglavju so definirane skupne zahteve, ki veljajo tako za aplikacije, ki v celoti izkoriščajo domorodno oblačno arhitekturo, kakor tudi za aplikacije, ki delno izkoriščajo domorodno oblačno arhitekturo. Do določene mere te zahteve lahko veljajo tudi za monolitne aplikacije, ki ne sledijo konceptom domorodne oblačne arhitekture.

4.1 Zahteve za aplikacije, ki v celoti izkoriščajo domorodno oblačno arhitekturo

4.1.1 Arhitekturna zasnova in sloji

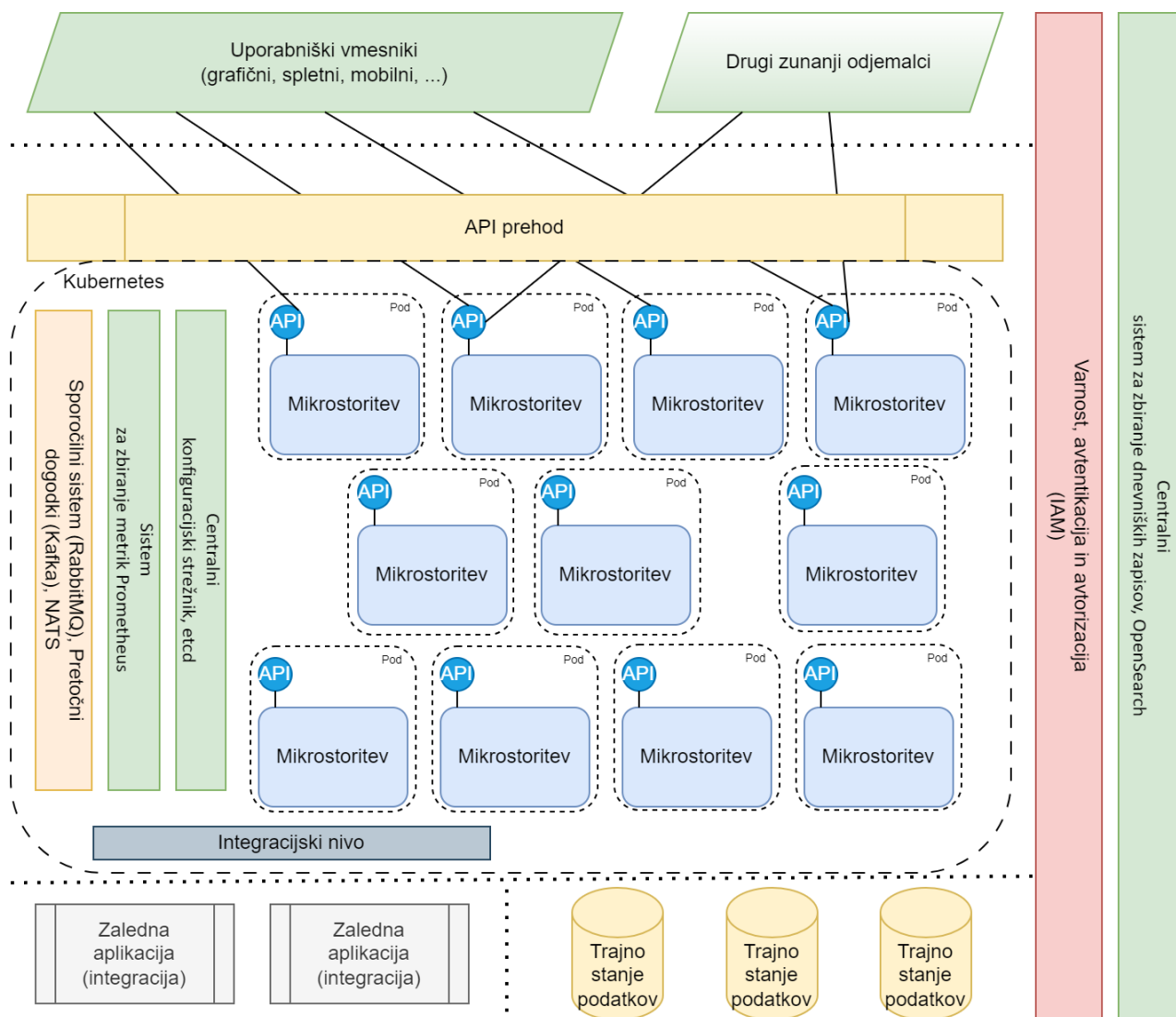
Arhitekturna zasnova aplikacij, ki v celoti izkoriščajo domorodno oblačno arhitekturo, mora slediti konceptom domorodne oblačne arhitekture. Aplikacije morajo imeti strogo ločene vsaj naslednje sloje:

- uporabniški vmesnik,
- API (aplikacijski programski vmesniki),
- zaledni del poslovne logike, implementiran kot mikrororitve,
- integracijski nivo (če ga aplikacija potrebuje),
- nivo zagotavljanja trajnosti podatkov.

Shematsko (poenostavljeno) arhitekturno zasnovo tipične aplikacije prikazuje naslednja slika. Na sliki je prikazan uporabniški vmesnik in morebitni ostali zunanji odjemalci in zaledni del. V okviru zalednega dela se v okolju Kubernetes izvaja množica mikrororitv. Vsaka mikrororitev se izvaja v svoje Kubernetes stroku (podu). Mikrororitve izpostavljajo API-je, npr. REST API. Komunikacija od zunanjih odjemalcev do API-jev poteka preko API prehoda. Storitve uporabljajo platformske funkcionalnosti:

- IAM sistem za avtentikacijo in avtorizacijo.
- Centralni sistem za beleženje dnevniških zapisov,
- Centralni konfiguracijski strežnik,
- Sistem za zbiranje metrik,
- Sporočilni in/ali sistem za pretočne dogodke.

Slika prikazuje tudi nivo trajnega stanja podatkov in zaledne aplikacije, ki se integrirajo preko integracijskega nivoja.



Slika 7: Poenostavljena arhitekturna zasnova tipične aplikacije

4.1.2 Mikrostoritvena arhitektura

Arhitektura aplikacije:

- Uporabljen mora biti arhitektura mikrorstitev².
- Zasnova mikrorstitev naj temelji na splošno sprejetih najboljših praksah, priporoča se uporaba domensko vodenega načrtovanja (domain-driven design) za identifikacijo in modeliranje mikrorstitev.
- Vsaka mikrorstitev naj pokriva določeno funkcionalno področje aplikacije.
- Odvisnosti med mikrorstitevami naj bodo minimizirane.
- Sklopljenost med storitvami naj bo šibka.
- Mikrorstitev naj bodo zasnovane brez stanja (stateless).
- Podprto mora biti horizontalno skaliranje mikrorstitev.

Mikrorstitev imajo vsaka svoj, neodvisen življenjski cikel. Graditi in nameščati je mogoče vsako mikrorstitev posebej, ločeno oz. neodvisno od drugih mikrorstitev.

Omogočeno naj bo, da več verzij iste mikrorstitev deluje vzporedno na okolju.

Za nadgradnje verzij iste mikrorstitev v izvajalnem okolju se lahko uporabljajo:

- Tekoče posodobitve (Rolling Updates).
- Modro/rdeče posodabljanje (Blue/Green Deployment).
- Vzporedno nameščanje (Canary Release).

4.1.3 Komunikacija med storitvami in sloji

Mikrorstitev morajo izpostavljati vsaj enega od naslednjih mehanizmov komunikacije:

- API-je (REST, gRPC, GraphQL ali SOAP).
- Komunikacija preko sporočilnih sistemov.
- Komunikacija preko dogodkov (pretočni dogodki).

Komunikacija med uporabniškim vmesnikom in zalednimi mikrorstitevami praviloma poteka preko API-jev. Spodbuja se uporaba različnih tipov API-jev, glede na namen:

- API, namenjen komunikaciji z zunanjimi in/ali internimi odjemalci (velikokrat poimenovan client-api),
- API, namenjen komunikaciji med storitvami (service-api),
- API, namenjen integraciji z drugimi sistemi (integration-api).

Podana klasifikacijska je zgolj informativne narave, mikrorstitev lahko uporabijo lastno klasifikacijo API-jev, glede na problemsko področje in zahteve.

² V prihodnosti bo DRO-NEXT predvidoma podpiral tudi brezstrežniške (serverless) funkcije, ki pa trenutno še niso podprte.

Mikrostoritve niso direktno izpostavljene uporabniškemu vmesniku. Med uporabniškim vmesnikom in API-jem je praviloma API prehod, ki upravlja povezave.

Integracije do drugih sistemov lahko potekajo preko API, sporočilnih sistemov ali pretočnih dogodkov. Običajno bodo integracije do zunanjih (tretjih) sistemov potekale preko API-jev. Integracije z internimi sistemi pa lahko potekajo preko API, sporočilnih sistemov ali pretočnih dogodkov. Kot integracijski API-ji so lahko uporabljeni isti API-ji kot za uporabniški vmesnik, če to ne vpliva na varnost delovanja integracij. Sicer je možno za potrebe integracij razviti ločen nabor API-jev. Tudi pri integracijah preko API-jev gre promet preko API prehoda.

4.1.4 Tehnologije API-jev

Pri uporabi tehnologij za API-je morajo aplikacije upoštevati:

- REST API,
 - REST API je priporočena tehnologija aplikacijskih programski vmesnikov za DRO.
 - Obvezna je uporaba specifikacije REST API z uporabo OpenAPI 3.x ali višje (bivši Swagger).
- gRPC,
 - Podprta je uporaba gRPC.
 - Uporaba .proto datotek za specifikacijo storitev.
 - Obvezna specifikacija portov, ki jih uporablja gRPC.
- GraphQL,
 - Podprta je uporaba GraphQL.
 - Priporoča se dosledno upoštevanje pravil poimenovanja, uporaba camelCase za polja in parametre in PascalCase za tipe in enumeracije.
- SOAP in WSDL,
 - Uporaba storitev SOAP je podprta predvsem za obstoječe (legacy) rešitve in se ne priporoča za domorodne oblačne rešitve.
 - Obvezna uporaba WSDL opisa storitve.
 - Priporočena uporaba document/literal oblike ter obvezna definicija ustreznih XML Shem (XSD) ter imenskih področij.

4.1.5 API prehod in sistem za upravljanje API-jev

DRO ponuja API prehod (API gateway) in sistem za upravljanje API-jev. API prehod je komponenta, ki omogoča enotno točko dostopa do različnih API-jev, ki jih izpostavljajo aplikacije in storitve. API prehod opravlja različne funkcije, kot so usmerjanje zahtev, avtentikacija in avtorizacija uporabnikov, validacija in transformacija podatkov, omejevanje uporabe, varovanje in šifriranje komunikacije, spremljanje in beleženje prometa, itd. API prehod lahko izboljša zmogljivost, varnost, in uporabniško izkušnjo API-jev.

Sistem za upravljanje API-jev je platforma, ki omogoča celovito upravljanje življenjskega cikla API-jev, od načrtovanja, razvoja, testiranja, objave, dokumentiranja, promocije, do vzdrževanja in upokojitve. Sistem za upravljanje API-jev pomaga pri oblikovanju in standardizaciji API-jev, zagotavljanju kakovosti in skladnosti API-jev, upravljanju verzij in sprememb API-jev, upravljanju odnosov in pogodb z uporabniki API-jev, analizi in optimizaciji delovanja in uporabe API-jev. Primeri rešitev za upravljanje z API-ji so MuleSoft Anypoint, Kong API Gateway, Tyk API Gateway, WSO2 API Manager, Red Hat 3scale API Management, Kumuluz API in ostali, v javnih oblakih pa še Azure API Management, AWS API Gateway, Google Cloud Apigee API Management in drugi.

Aplikacije morajo:

- Vse API komunikacije z drugimi sistemi opravljati preko API prehodov. Direktna komunikacija mimo API prehoda ni dovoljena. Pri tem uporabijo ustrezen prehod glede na tip komunikacije:
 - med aplikacijskimi sistemi znotraj DRO,
 - med aplikacijami v sistemu javne uprave (omrežje HKOM),
 - do zunanjih sistemov.
- Aplikacije morajo pripraviti seznam API-jev za eksterno komunikacijo in specificirati njihovo namembnost.
- Vse te API-je morajo registrirati na ustrezen API prehod.
- Vsi API-ji morajo biti opremljeni z ustrežno dokumentacijo (minimalno OpenAPI 3.x za REST API-je).
- Vse API-je morajo tudi objaviti v sistemu za upravljanje API-jev, skladno s smernicami DRO.

4.1.6 Verzioniranje

Zagotoviti je potrebno sistematično, dosledno in konsistentno verzioniranje:

- izvirne kode,
- API-jev,
- artefaktov, ki nastanejo v procesu gradnje (build) in se odlagajo na repozitorij,
- izdaj verzij aplikacije oz. posameznih mikrorstitev (če aplikacija podpira neodvisni nameščanje mikrorstitev).

Shema verzioniranja za posamezne sklope bo določena in predpisana s strani DRO.

4.1.7 Uporaba principov 12 faktorskih aplikacij

Od aplikacije se pričakuje skladnost z vsemi dvanajstimi principi 12 faktorskih aplikacij:

1. Upravljanje z izvirno kodo (code base),
 - Vsaka mikrorstitev oz. komponenta ima svojo kodno osnovo (code base).

- Shranjena je v repozitoriju in uporablja sistem za nadzor in sledenje verzij (version control).
 - Iz skupne kodne osnove generiramo namestitvev (deployment), ki jo nameščamo na različna okolja (npr. preko slik vsebnikov).
2. Upravljanje z odvisnostmi,
- Vsaka mikrostoritev izrecno (eksplicitno) deklarira in izolira svoje odvisnosti, s čimer se zmanjšuje vpliv sprememb in implikacije na druge dele sistema.
 - Mikrostoritev naj se ne zanaša na sistemske pakete.
 - Vsaka mikrostoritev vidi samo odvisnosti, specificirane v manifestu.
3. Konfiguracija,
- Celotna konfiguracija je eksternalizirana.
 - Zahtevano je strogo ločevanje konfiguracije in kode.
 - Podprti so naslednji konfiguracijski viri:
 - Obvezno okoljske spremenljivke in konfiguracijska datoteka.
 - Opcijsko centralni konfiguracijski strežnik.
 - Namestitev generiramo enkrat in jo nameščamo na različna okolja tako, da nastavimo ustrezno konfiguracijo (brez ponovne gradnje (build)).
4. Zaledne storitve,
- Vse zaledne storitve obravnavamo kot zamenljive vire.
 - Zaledne storitve in vire, kot npr. podatkovne baze, predpomnilnike, sporočilne sisteme, dogodkovne kanale itd., naslavljamo z uporabo URL ali drugih konfigurabilnih povezav.
 - S tem razvežemo storitve in vire od aplikacij na način, da jih lahko preprosto zamenjamo.
5. Gradnja, izdaja, izvajanje,
- Strogo ločimo med fazami gradnje, izdaje in izvajanja.
 - Vsaka izdaja mora biti označena z edinstvenim ID-jem in podpirati možnost povrnitve (rollback).
6. Procesi,
- Vsaka mikrostoritev se izvaja v svojem, ločenem procesu.
 - Mikrostoritve oz. procesi, v katerih se le-te izvajajo, ne hranijo stanja (stateless).
 - Stanje se shranjuje zunaj mikrostoritev oz. procesov, npr. v podatkovni bazi, shrambi, predpomnilniku ipd.
7. Zahtevano je strogo ločevanje konfiguracije in kode (port binding),
- Vsaka mikrostoritev je samozadostna enota.
 - Svoje funkcionalnosti, ki so na voljo preko vmesnikov, izpostavlja preko svojih, lastnih vrat (port).
 - S tem se zagotavlja izolacija od ostalih mikrostoritev.
8. Sočasnost procesov,
- Potrebe po povečanem obsegu zahtev naslavljamo s horizontalnim skaliranjem tako, da zaženemo več identičnih procesov mikrostoritve (oz. več instanc).

- Koncept horizontalnega skaliranja je preferiran način. Vertikalnemu skaliranju, kjer povečujemo kapacitete enega procesa, se izogibamo.
- Mikrostoritve morajo podpirati in biti prilagojene za horizontalno skaliranje.
- 9. Zahtevano je strogo ločevanje konfiguracije in kode,
 - Zagotoviti je potrebno, da se mikrostoritve oz. njihovi vsebniki zaganjajo hitro.
 - V vsakem trenutku jih mora biti mogoče izklopiti brez negativnih posledic.
 - To je pomembno pri skaliranju in hitrem nameščanju novih verzij.
- 10. Enakopravnost okolij,
 - Različna izvajalna okolja so med seboj čim bolj podobna.
 - DRO zagotavlja podobnost testnega, UAT in produkcijskega okolja.
 - Razvijalci naj zagotovijo podobnost razvojnega okolja DRO okolju.
- 11. Beleženje dnevniških zapisov,
 - Na beleženje dnevniških zapisov gledamo kot na tokove dogodkov.
 - Dnevniški (log) zapisi naj se izpisujejo na standarden izhod.
 - V tem primeru bo izvajalno okolje dnevniške zapise pošiljalo v centralen strežnik za beleženje dnevniških zapisov.
 - Dnevniški (log) zapisi naj se pošiljajo direktno na centralni strežnik za beleženje dnevniških zapisov.
- 12. Skrbniški procesi,
 - Skrbniški (administrativni) posegi v aplikacijo naj se izvajajo v enkratnih procesih, enako kot aplikacijski procesi.
 - Skrbniški procesi naj se izvajajo v enakem okolju kot ostali procesi aplikacije.

4.1.8 Preverjanje vitalnosti (Health Check)

Mikrostoritve morajo obvezno implementirati preverjanje vitalnosti:

- Zahteva se preverjanje liveness in readiness.
- Mikrostoritev izpostavlja obe preverjanji na ustrezni končni točki (npr. /health/live in /health/ready).
- Pri tem mora uporabljati standarden nabor statusnih kod in podatkovno strukturo.
- Preverjanje vitalnosti je potrebno konfigurirati tudi v namestitvenem deskriptorju za Kubernetes.

4.1.9 Zbiranje metrik

DRO v okolju Kubernetes ponuja možnost zbiranja metrik z uporabo Prometheus. Poleg tega ponuja dostop do Grafane, na kateri je možno spremljati metrike izvajanja aplikacij. Izpostavljanje metrik je za mikrostoritve opcijsko.

Predvsem za domorodne oblačne aplikacije je priporočljivo, da izpostavijo podatke o metrikah delovanja na nivoju vsake mikrostoritve. Mikrostoritev mora za potrebe izpostavljanja metrik

implementirati HTTP končno točko za metrike, ki jo bo klical Prometheus. Tipično se metrike izpostavijo na končni točki /metrics, vendar lahko aplikacija uporabi tudi drugačno poimenovanje metrik. Na končni točki se vsebina metrik izpostavi v formatu, ki je skladen s Prometheus. Format je tekstovni in temelji na vrsticah, ki vsebujejo ime metrike, oznake (labels), vrednost in čas. Ime metrike mora biti opisno in dosledno slediti priporočilom za poimenovanja.

Predviden in privzeti je način pridobivanja »potegni« (pull), kjer bo Prometheus v določenih časovnih intervalih pobiral metrike. V kolikor takega načina ni možno realizirati, npr. zaradi omejitev dostopnosti omrežja ali drugih razlogov, ponuja DRO tudi pristop z uporabo komponente push-gateway.

Mikrostoritve naj izpostavljajo nabor metrik, ki ga je glede na aplikacijo smiselno spremljati, kar vključuje naslednje skupine:

- Metrike porabe virov (CPU, pomnilnik, podatkovna shramba, omrežje, ...).
- Metrike glede delovanja mikrostoritve (velikost kopice (heap), število niti, število povezav na bazo, število odprtih http povezav ipd.).
- Aplikacijske metrike (čas izvedbe oz. RTT (round-trip-time) posameznih metod, operacij, metod na API-jih ipd.; število sočasnih uporabnikov, količine prenesenih podatkov ipd.).

Pri zbiranju in izpostavljanju metrik je potrebno uporabljati ustrezne tipe metrik, ki jih ponuja Prometheus. Tipi metrik so: Counter, Gauge, Histogram in Summary. Counter je metrika, ki se samo povečuje in predstavlja število dogodkov ali zahtevkov. Gauge je metrika, ki se lahko povečuje ali zmanjšuje in predstavlja trenutno stanje ali vrednost. Histogram je metrika, ki zbira porazdelitev vrednosti in predstavlja število opazovanj, vsoto opazovanj in število opazovanj v določenih razredih. Summary je metrika, ki zbira porazdelitev vrednosti in predstavlja število opazovanj, vsoto opazovanj in kvantile opazovanj.

Mikrostoritev naj zbira in izpostavlja metrike tako, da to ne bo pomembno vplivalo na njeno delovanje. Običajno se za te namene lahko uporabi kakšna knjižnica mikrostoritvenega ogrodja, ki omogoča avtomatizirano spremljanje in objavljanje metrik, npr. Prometheus client libraries, Micrometer, Spring Boot Actuator, Dropwizard metrics ipd.

4.1.10 Odprta telemetrija (Open Telemetry)

Implementacija odprte telemetrije (Open Telemetry) je opsijska, vendar priporočena lastnost mikrostoritev. Odprta telemetrija, pogosto imenovana Otel, je odprtokodni nabor orodij, API-jev in SDK-jev, ki omogočajo instrumentacijo storitev in pošiljanje podatkov v zaledne sisteme. Gre za naslednika Open Census in Open Tracing, ki sta bila dva ločena projekta. Sedaj sta združena v Open Telemetry. Odprta telemetrija podpira štiri tipe signalov:

- sledi izvajanja (traces)
- metrike,
- dnevniške zapise (loge),

- prtljago (baggage).

Določeni programski jeziki, kot npr. Go, ponujajo podporo za odprto telemetrijo že na nivoju programskega jezika. Za druge jezike obstajajo knjižnice, ki avtomatizirajo vpeljavo podpore. Tretja možnost je ročno dodajanje kode za instrumentacijo. Vse te opcije so podprte za DRO.

4.1.11 Odpornost na napake (Fault Tolerance)

Priporoča se, da mikrororitve implementirajo vzorce odpornosti na napake (fault tolerance). Implementacija vzorcev odpornosti na napake je opsijska.

Priporoča se uporaba oz. implementacija naslednjih vzorcev odpornosti na napake:

- Časovnik (timeout), ki ga uporabimo za:
 - Omejitev časa za izvedbo klica na zunanjo komponento (npr. mikrororitev preko API).
 - Zajezitev čakalnega časa in sprostitve virov ob izteku časa.
- Ponovno poskušanje (retry), ki ga uporabimo za:
 - Ponovno poskušanje poskrbi za ponovno izvedbo iste akcije, npr. klica API-ja v primeru, ko klic prvič ni bil uspešen.
 - Pred ponovno izvedbo akcije se doda dodaten konstanten časovni zamik (delay), ki mu velikokrat prištejemo še nekaj naključnega odstopanja (jitter).
 - Največkrat uporablja skupaj s časovnikom.
- Nadomestni mehanizmi (fallback):
 - Namesto vračanja izjem in napak se ob neuspehu omogoči izvedba nadomestnega mehanizma, ki poskuša vrniti odgovor.
 - To je lahko klic nadomestne storitve ali vira.
 - Lahko pa tudi samo generiranje preprostega statičnega odgovora.
- Hitri neuspeh (fail fast):
 - Počasni odzivi na zahtevo so slabši od neodzivnosti, najslabše pa je dobiti počasen odziv z napako.
 - Hitra napaka se vrne v primerih, ko sistem vnaprej pričakuje napako pri določeni operaciji.
- Pregrade (bulkheads):
 - Omejuje akcije s ciljem prepreči, da bi neuspeli klici preplavili določen vir (npr. mikrororitev).
- Prekinjevalec toka oz. varovalke (circuit breakers):
 - Ob zaznani napaki oz. nedelujoči klicani komponenti (npr. mikrororitvi), se blokira izvajanje klicev te komponente za določen časovni interval in s tem prepreči nekontrolirano širjenje napake.

Poleg omenjenih vzorcev se priporoča, da se v mikrororitvah implementirajo mehanizmi za izboljšanje robustnosti delovanja, kot npr. ustrezna obravnava izjem z izvedbo korektivnih akcij.

4.1.12 Uporaba storitvenega omrežja (Service Mesh)

Uporaba storitvenih omrežij (Service Mesh) je opsijska zahteva, vendar podprta s strani DRO. Storitveno omrežje je konfigurabilni nivo, ki omogoča obvladovanje in upravljanje komunikacije med storitvami. Uporablja posrednik, ki se imenuje stranski vsebnik (sidecar) ali okvir (mesh), s čemer zagotovi izolacijo kode in logike za nadzorne in sistemske funkcije od kode za poslovno logiko. S tem lahko aplikacija del zahtev delegira posredniku.

Storitveno omrežje podpira naslednje funkcije:

- Usmerjanje prometa med mikrostoritvami usmerja na podlagi pravil in politik, ki jih lahko dinamično spreminjamo. To omogoča, da se lahko izvajajo funkcije, kot so preusmeritev, ponovni poskus, prekinjevalci toka, modro/zeleno nameščanje, vzporedno nameščanje (canary release) in drugo.
- Nadzor dostopa do mikrostoritev se lahko omeji na podlagi identitete, vloge, zahtev ali drugih meril.
- Sledenje in merjenje oz. zbiranje podatkov o prometu, zmogljivosti, napakah in drugih metrikah med mikrostoritvami.
- Vstavljanje dodatne logike v komunikacijo med mikrostoritvami, kot so preverjanje pristnosti, preverjanje pravic, preoblikovanje sporočil, filtriranje vsebine in drugo.

Priporoča se uporaba razširjenih orodij za storitvena omrežja kot npr. Istio, Linkerd, Consul in drugi.

4.1.13 Vzorci uporabe podatkovnih baz

Iz vidika uporabe podatkovnih baz DRO omogoča oba vzorca:

- Uporaba skupne podatkovne baze, ki si jo deli več mikrostoritev.
- Vsaka mikrostoritev uporablja svojo ločeno podatkovno bazo (database per microservice).

Opomba: posamezne podatkovne baze, ki jih uporabljajo mikrostoritve, gostijo centralni sistemi za upravljanje podatkovnih baz, ki jih ponuja DRO in so specifikirani v ločenem poglavju.

Način, kjer vsaka mikrostoritev uporablja svojo ločeno podatkovno bazo, temelji na načelu, da ima vsaka mikrostoritev svojo lastno podatkovno zbirko, ki je samo njej dostopna. To pomeni, da ni deljenih ali skupnih podatkovnih zbirk med mikrostoritvami in da se podatki med mikrostoritvami izmenjujejo samo prek API-jev oz. sporočil/dogodkov. Tak pristop ima naslednje prednosti:

- Večja kohezija in nizka sklopljenost: Vsaka mikrostoritev je odgovorna za svoje podatke in logiko, in ni odvisna od drugih mikrostoritev za dostop do podatkov.
- Večja skalabilnost in zmogljivost: Vsaka mikrostoritev lahko prilagodi svojo podatkovno zbirko glede na svoje potrebe in zahteve, in lahko uporablja različne tehnologije, kot so relacijske, nerelacijske podatkovne zbirke. To omogoča boljšo optimizacijo in izkoriščanje podatkovnih virov.

- Večja varnost in skladnost: Vsaka mikrostoritev lahko uporablja svoje mehanizme za zaščito, šifriranje, varnostno kopiranje in obnavljanje svojih podatkov.

Slabosti:

- Večja kompleksnost in stroški: Vsaka mikrostoritev mora vzdrževati svojo podatkovno zbirko, kar pomeni, da je potrebno več infrastrukture, orodij, licenc, osebja in drugih virov.
- Težavnejša konsistentnost in integracija: Vsaka mikrostoritev ima svoj pogled na podatke, ki se lahko razlikuje od drugih mikrostoritev. To lahko povzroči neskladnosti, zastarelost ali podvajanje podatkov. Prav tako je težje zagotoviti transakcijsko konsistentnost ali izvesti kompleksne poizvedbe med različnimi podatkovnimi zbirkami.

V primeru uporabe ločenih podatkovnih baz, lahko mikrostoritve uporabijo tudi mehanizme, kot so event sourcing in CQRS.

4.1.14 Orodja za migracijo shem

Pri uporabi relacijskih podatkovnih baz je obvezna uporaba orodij za migracijo shem, ki se jih lahko vključi v avtomatiziran postopek gradnje (builda), včasih imenovan tudi Database-as-code. Primeri takih orodij so Liquibase, Flyway, Sqitch ali Atlas.

4.1.15 Centralni avtentikacijski in avtorizacijski (IAM) strežnik

Avtentikacijski in avtorizacijski IAM strežnik v DRO je Keycloak in nudi podporo za registracijske in prijavne tokove za uporabnike in za upravljanje identitet ter nadzora dostopa. IAM strežnik Keycloak bo integriran s skupnim aplikacijskim gradnikom Varnostna shema. Uporaba centralnega avtentikacijskega in avtorizacijskega IAM strežnika je obvezna za naslednje funkcionalnosti:

- Prijavne in registracijske tokove:
 - Hranjenje uporabniških prijavnih podatkov v sami aplikaciji ni dovoljeno.
- Avtentikacijo in avtorizacijo dostopa do zalednih storitev in API-jev:
 - Omogočen je nadzor dostopa na osnovi vlog (Role based access control) in na osnovi virov (Resource based access control).

Aplikacije, ki uporabljajo centralni Keycloak, morajo specificirati realme, uporabnike, vloge, prijavno/registracijske tokove in ostale podrobnosti. Pri tem je obvezna uporaba standarda OAuth2 in protokola OpenID Connect. Aplikacija mora podpirati enega ali več tokov (flows) OAuth2 in OpenID Connect, ki so primerni za njeno vrsto in namen. Aplikacija naj uporablja konfigurabilne končne točke IAM strežnika in ustrezne parametre glave (Header) za pošiljanje in prejemanje podatkov s strežnikom Keycloak.

Obvezna uporaba JWT žetonov, lahko v kombinaciji z API ključi na vseh nivojih komunikacije. Aplikacija (mikrostoritve, uporabniški vmesnik in ostale komponente) mora preverjati veljavnost in

podpis JWT žetona, ki ga prejme od IAM strežnika. Žeton je sestavljen iz treh delov: glava (header), telo (payload) in podpis (signature).

Aplikacija mora upoštevati varnostne in zmogljivostne smernice pri uporabi Keycloak strežnika. Na primer, aplikacija mora uporabljati HTTPS protokol za komunikacijo z Keycloak strežnikom, aplikacija mora shranjevati žetone na varnem mestu in jih ne izpostavljati tretjim osebam, aplikacija mora osveževati žetone pred njihovim potekom z uporabo žetona za osvežitev (refresh token), aplikacija mora preklicati žetone, ko niso več potrebni. Smiselno je, da aplikacija uporablja predpomnjenje (caching) za zmanjšanje števila klicev do strežnika IAM.

Izjemoma v utemeljenih primerih lahko aplikacija, po predhodni odobritvi upravljalca infrastrukture DRO, uporabi tudi ločeno namestitev Keycloaka, integrirano v informacijsko rešitev in konfigurirano v skladu s smernicami upravljalca infrastrukture DRO. Pri tem mora upoštevati vse smernice, navedene v tem dokumentu.

Izključno za namene obstoječih (legacy) aplikacij so podprti tudi nekateri drugi mehanizmi avtentikacije in avtorizacije:

- SAML,
- certifikati,
- uporabniško ime in geslo oz. basic avtentikacija.

4.1.16 Spletni uporabniški vmesniki

Na nivoju spletnih uporabniških vmesnikov je obvezna uporaba konceptov:

- Enostranskih aplikacij (SPA – Single Page Applications) za spletne aplikacije oz.
- PWA (Progressive Web Applications) za uporabniške vmesnike, ki morajo delovati na spletnih in mobilnih napravah.
- Prav tako je obvezna uporaba odzivnega (responsive) designa.

Za zahteve glede avtentikacije in avtorizacije ter prijavnih tokov glej prejšnje poglavje.

4.1.17 Centralni konfiguracijski strežnik

DRO (opsijsko) ponuja centralni konfiguracijski strežnik etcd. S tem omogoča aplikacijam (oz. njihovim mikrostoritvam), da konfiguracijo shranjujejo v centralni konfiguracijski strežnik (in ne zgolj v okoljske spremenljivke ali konfiguracijske datoteke).

Vloga in namen centralnega konfiguracijskega strežnika je zagotoviti zanesljivo, visoko razpoložljivo, in porazdeljeno shranjevanje ključ-vrednost parov, ki se uporabljajo za konfiguracijo in koordinacijo aplikacij in storitev. Priporoča se, da aplikacije in njihovi sestani deli (storitve, mikrostoritve) uporabljajo centralni konfiguracijski strežnik za upravljanje konfiguracije. Centralni konfiguracijski strežnik omogoča:

- Enostavno in dinamično spreminjanje konfiguracije aplikacij in storitev, brez potrebe po ponovnem zagonu ali prenosu podatkov.
- Odkrivanje in registracijo storitev, ki se lahko samodejno prilagajajo spremembam v razpoložljivosti in lokaciji.
- Sinhronizacijo in usklajevanje stanja in akcij med različnimi aplikacijami in storitvami, ki delujejo na različnih vozliščih.
- Varno in šifrirano komunikacijo in shranjevanje podatkov, ki preprečuje nepooblaščen dostop in manipulacijo.

DRO postavitve etcd podpira protokola v2 in v3. Etcd protokola v2 in v3 sta dve različici API-ja, ki jih etcd podpira za komunikacijo. Etcd protokol v2 je starejši in preprostejši protokol, ki temelji na HTTP in JSON in omogoča osnovne operacije nad ključ-vrednost podatki. Etcd protokol v3 je novejši in naprednejši protokol, ki temelji na gRPC in protobuf in omogoča bolj učinkovite in zmogljive operacije nad ključ-vrednost podatki, kot so transakcije, zasebni ključi, zaklepanje, časovne omejitve, itd.

Priporočila za uporabo etcd v DRO aplikacijah:

- Aplikacije naj ustrezno strukturirajo konfiguracijske nastavitve v sklope.
- Uporabljajte hierarhično strukturo ključev, ki odraža logično organizacijo konfiguracije. Na primer, lahko uporabite obliko `/app/env/section/key`, kjer je `app` ime aplikacije, `env` okolje (npr. `dev`, `test`, `prod`), `section` področje konfiguracije (npr. `database`, `server`, `security`), in `key` posamezna nastavev (npr. `host`, `port`, `password`).
- Uporabljajte opisna in smiselna imena ključev, ki jasno povejo, kaj predstavljajo. Izogibajte se uporabi kratkih ali nejasnih imen, ki lahko povzročijo zmedo ali napake. Uporabljajte enotno konvencijo poimenovanja, ki je skladna z vašim programskim jezikom ali okvirjem.
- Uporabljajte ustrezne tipe vrednosti, ki ustrezajo tipom podatkov v vaši aplikaciji. Etcd podpira shranjevanje vrednosti kot nize, bajte, ali JSON objekte.
- Uporabljajte funkcije etcd, ki olajšajo upravljanje konfiguracije, kot so `TTL`, `watch`, `lease`, `lock`, `transaction`, itd. Te funkcije omogočajo, da nastavite časovne omejitve, spremljate spremembe, zaklenete vire, izvedete atomične operacije, itd.
- Če je možno, uporabljajte etcd protokol v3, saj je bolj stabilen, varen in združljiv z novimi funkcijami in orodji.
- Starejše aplikacije ali storitve lahko uporabljajo etcd protokol v2, posebej v primeru, če etcd v3 ni podprt.
- Aplikacija naj ne meša etcd protokolov v2 in v3, saj sta ločena in izolirana in ne delita istega prostora za shranjevanje podatkov.

4.1.18 Izvajanje na okolju Kubernetes

Aplikacija mora biti v celoti prilagojena izvajanju v okolju Kubernetes. Mikrostoritve morajo biti ustrezno razporejene v stroke (pod). Namestitveni deskriptorji morajo konfigurirati napredne mehanizme, ki jih podpirajo mikrostoritve aplikacije (tiste, ki so za aplikacijo smiselni):

- preverjanje vitalnost (health check),
- skalabilnost na nivoju posameznega stroka (pod) in aplikacije kot celote (statično ali z uporabo HPA – Horizontal Pod Autoscaler),
- ustrezno izpostavljanje končnih točk,
- strategija posodabljanja (npr. Rolling Updates),
- zbiranje metrik oz. odprte telemetrije,
- ostale nastavitve.

Aplikacija kot celota mora tudi smiselno definirati zmogljivosti virov, ki jih bo uporabljala (CPU, shramba, omrežje, število povezav (npr. povezav do podatkovne baze, drugih zunanjih virov ipd.)).

4.1.19 Avtomatizirana vzpostavitev okolja z uporabo IaC

DRO omogoča avtomatizirano vzpostavitev okolja (infrastrukture), ki ga potrebuje aplikacija, po konceptih IaC (Infrastructure-as-Code). IaC je postopek, kjer uporabljamo programsko kodo za definicijo in vzpostavitev okolja aplikacije. V preteklosti je bila vzpostavitev okolja (infrastrukture) večinoma ročni postopek - ročno ste ustvarili VM, namestili programsko opremo ter konfigurirali nastavitve in datoteke. IaC to zamenjuje s programsko kodo in na ta način te postopke avtomatizira, pohitri in jih naredi ponovljive.

Za avtomatizacijo postopkov IaC se uporablja Terraform/OpenTofu ter Ansible. Prvi je bolj namenjen sami vzpostavitvi okolja, drugi pa konfiguraciji aplikacijskih rešitev. Razvijalci aplikacij se odločijo, katero od obeh (ali obe) orodij bodo uporabili glede na zahteve in potrebe aplikacije, ki jo razvijajo. V ta namen pripravijo delujoče verzije skript za namestitev in konfiguracijo aplikacij rešitev (oz. posameznih storitev in komponent) ter jih predajo v sklopu projekta upravljalcu infrastrukture DRO, ki bo omenjene skripte dodelal in integriral za potrebe namestitve in konfiguracije.

Avtomatizirana vzpostavitev okolja v DRO je opsijska in ni obvezno zahtevana od aplikacij. V DRO zasledujemo naslednje cilje z vpeljavo IaC:

- Postopek vzpostavitve okolja, zapisan v kodi, je ponovljiv.
- Zapis okolja v kodi olajša ustvarjanje novih okolij in predvsem hitro nameščanje in skaliranje rešitve v oblaku.
- Spremembe in dopolnitve okolja so sledljive.
- Zagotavlja se enotnost in konsistentnost vseh okolij ter identificira morebitne odklone med njimi.
- IaC koda je zapisana idempotentno, da se lahko konfiguracija aplicira večkrat.
- Z uporabo IaC se v celoti izognemo ročni konfiguraciji.

- IaC koda se shranjuje v repozitoriju git, ki predstavlja edini izvor resnice (single source of truth).
- IaC uporablja postopke CI/CD avtomatizacije, podobno kot je to definirano za izvirno kodo.

Smernice priprave IaC:

- Koda za konfiguracijo okolja aplikacije je ločena in nima povezav do izvirne kode same aplikacije.
- Če je možno, uporabimo deklarativni opis infrastrukture in aplikacij, ki določa želeno stanje v ciljnem okolju. S tem se izognemo zapletenim skriptom in ukazom, ki lahko povzročijo napake ali neskladnosti.
- Uporabimo git kot enotni vir resnice za svojo infrastrukturo in aplikacije. Vse spremembe naj bodo sledljive, pregledne in verzionirane v gitu. Pri tem sledimo izbranemu git delovnemu toku (workflow).
- Uporabite avtomatizirane procese za uporabo sprememb v ciljnem okolju. Najboljša praksa je uporabiti »pull« namestitvev, kjer agent ali operater potegne in uporabi spremembe iz git-a. To povečuje varnost in zanesljivost infrastrukture in aplikacij.
- IaC kodo razdelimo v module in se izogibamo podvajanju. Zapletene predloge razdelimo na več lažje obvladljivih.

4.2 Zahteve za aplikacije, ki delno izkoriščajo domorodno oblačno arhitekturo

4.2.1 Arhitekturna zasnova in sloji

Arhitekturna zasnova aplikacij, ki delno izkoriščajo domorodno oblačno arhitekturo, mora slediti tro ali večslojni arhitekturi z naslednjimi sloji:

- Uporabniški vmesnik,
- Poslovna logika (lahko delno izpostavljena skozi API-je ali spletne storitve),
- Podatkovni nivo.

V to kategorijo se uvršča večina klasičnih troslojnih aplikacij in aplikacij, ki temeljijo na storitveno usmerjeni arhitekturi (SOA).

4.2.2 Komunikacija med storitvami in sloji

Komunikacija poteka delno preko API-jev ali spletnih storitev, običajno so uporabljeni naslednji mehanizmi:

- API-ji oz. spletne storitve (REST ali SOAP),
- Komunikacija preko sporočilnih sistemov.

Komunikacija med uporabniškim vmesnikom in zalednimi storitvami poteka preko REST API ali spletnih storitev SOAP/WSDL. Če pa aplikacija uporablja strežniški koncept spletnih aplikacij (JSP, PHP, ASP in podobno), pa se lahko uporabljajo tudi drugi zaledni mehanizmi in protokoli.

4.2.3 API prehod in sistem za upravljanje API-jev

Aplikacije tega tipa običajno ne uporabljajo komunikacije preko API prehodov. Pri nadgradnjah teh aplikacij pa je smiselno premisliti o vključitvi API prehoda tam, kjer je to smiselno.

4.2.4 Verzioniranje

Za ta tipa aplikacij je zahtevano dosledno verzioniranje izvirne kode. Verzioniranje ostalih artefaktov je opcijsko.

4.2.5 Uporaba principov 12 faktorskih aplikacij

Aplikacije so skladne samo z naslednjimi principi 12 faktorskih aplikacij:

- Upravljanje z izvirno kodo (code base),
- Konfiguracija,
- Zaledne storitve,
- Beleženje dnevniških zapisov.

4.2.6 Vzorci uporabe podatkovnih baz

Ta tip aplikacij uporablja princip skupne podatkovne baze za celotno aplikacijo.

4.2.7 Spletni uporabniški vmesniki

Na nivoju spletnih uporabniških vmesnikov je možna uporaba:

- Strežniških tehnologij za spletne strani (JSP, Servleti, PHP, ASP in podobne),
- Odjemalnih tehnologij za spletne strani (SPA, PWA, Angular, React, Vue.js, itd.),
- Kombiniranega pristopa.

4.2.8 Izvajanje na okolju Kubernetes

Aplikacija omogoča izvajanje v okolju Kubernetes, vendar je običajno pakirana v en ali nekaj strokov (podov). Prav tako ne uporablja naprednih mehanizmov, kot so preverjanje vitalnost (health check), strategija posodabljanja (npr. Rolling Updates), zbiranje metrik oz. odprte telemetrije in ostalih nastavitev.

V odvisnosti od tega, kako aplikacija obravnava stanje (je stateless ali stateful), je možna uporaba horizontalne skalabilnosti.

4.3 Skupne zahteve za domorodne oblačne aplikacije

V tem poglavju so definirane skupne zahteve, ki veljajo tako za aplikacije, ki v celoti izkoriščajo domorodno oblačno arhitekturo, kakor tudi za aplikacije, ki delno izkoriščajo domorodno oblačno arhitekturo. Do določene mere te zahteve lahko veljajo tudi za monolitne aplikacije, ki ne sledijo konceptom domorodne oblačne arhitekture.

4.3.1 Avtomatiziran postopek gradnje aplikacije (build)

Postopek gradnje aplikacije mora biti avtomatiziran z ustreznim orodjem za gradnjo (build tool) izbranega programskega jezika (npr. Maven, Gradle, NPM itd.). Pri tem je potrebno upoštevati pravila verzioniranja izvirne kode in artefaktov ter ustreznega poimenovanja.

Postopek gradnje mora vključevati minimalno naslednje sklope:

- Prevajanje,
- Pakiranje,
- Izvedba testov (testi enot in integracijski testi),
- Generiranje dokumentacije,
- Podprto naj bo tudi čiščenje artefaktov,
- Dodatno je priporočeno še, da so podprti avtomatski varnostni in obremenilni testi ter preverjanje kakovosti kode.

Artefakti, ki nastanejo v postopku gradnje aplikacije, se morajo odlagati na repozitorij artefaktov DRO Sonatype Nexus, skladno s strukturo repozitorija in pravili poimenovanja ter ustreznim verzioniranjem.

4.3.1.1 Avtomatizirani testi enot in integracijski testi

Aplikacije morajo kot del postopka gradnje izvesti tudi:

- Avtomatizirane teste enot z mockanimi odvisnostmi,
- Avtomatizirane integracijske teste.

Teste enot in integracijske teste mora implementirati razvijalec aplikacije in jih vključiti v repozitorij izvirne kode.

DRO ne predpisuje testnih ogrodij in jih razvijalec lahko poljubno izbere glede na izbran programski jezik. V nadaljevanju podajamo nekatere možnosti za posamezne programske jezike, vendar to ni zavezujoč seznam in lahko razvijalci izberejo tudi kakšno drugo ogrodje (slika 7).

Slika 8: Testiranje enot, mockanje in integracijsko testiranje

Testiranje enot	Mockanje	Integracijsko testiranje
-----------------	----------	--------------------------

<i>Java</i>	JUnit, TestNG, Mockito, PowerMock, itd	Mockito, PowerMock, EasyMock, JMockit, itd	Spring Boot Test, Arquillian, Selenium, RestAssured, itd
<i>C#</i>	Spring Boot Test, Arquillian, Selenium, RestAssured, itd	Moq, NSubstitute, FakeItEasy, Rhino Mocks, itd	SpecFlow, Selenium, RestSharp, FluentAssertions, itd
<i>JavaScript / Node.js</i>	Jest, Mocha, Chai, Sinon, Enzyme, itd	Sinon, Jest, Nock, proxyquire, itd	Cypress, SuperTest, Mocha, Chai, itd
<i>Phyton</i>	unittest, pytest, mock, responses, itd	mock, responses, unittest.mock, pytest-mock, itd	behave, selenium, requests, pytest, itd
<i>Go</i>	esting, testify, ginkgo, gocheck, itd	gomock, mockery, testify/mock, pegomock, itd	httpexpect, go-resty, resty, gomega, itd

Teste enot in integracijske teste je potrebno vključiti v avtomatiziran cikel gradnje aplikacije (build).

4.3.1.2 Generiranje tehnične dokumentacije

Kot del cikla gradnje aplikacije naj bo vključeno tudi generiranje tehnične dokumentacije aplikacije. Dokumentacija naj bo avtomatsko generirana iz izvorne kode, komentarjev, testov, konfiguracij in drugih datotek, ki so del aplikacije. To zagotavlja, da je dokumentacija vedno ažurna, dosledna in sinhronizirana z aplikacijo.

Dokumentacija naj vključuje specifikacijo API, ki opisuje, kako se lahko aplikacija poveže in komunicira z drugimi aplikacijami ali storitvami. Specifikacija API naj vsebuje informacije o končnih točkah, parametrih, glavah, telesih, odgovorih, napakah, avtentikaciji, avtorizaciji in drugih podrobnostih. Specifikacija API naj uporablja standardni format OpenAPI 3.x (bivši Swagger).

Dokumentacija naj vključuje dokumentacijo izvorne kode, npr. razredov, ki opisuje, kako je aplikacija strukturirana in organizirana v smiselne enote, ki imajo svoje lastnosti, metode, odvisnosti in odgovornosti. Dokumentacija razredov naj vsebuje informacije o imenih, tipih, opisih, argumentih, vrednostih, izjemah, dedovanju, implementaciji in drugih podrobnostih. Dokumentacija razredov naj uporablja standardni format, kot je npr. Javadoc za Javo, ki omogoča enostavno generiranje, pregledovanje in povezovanje dokumentacije.

Dokumentacija naj vključuje tehnično dokumentacijo namestitve, ki opisuje posamezne komponente, kako se lahko aplikacija namesti, zgradi, zažene, testira, posodobi in odstrani v različnih okoljih, kot so test, UAT in produkcija. Tehnična dokumentacija namestitve naj vsebuje informacije o potrebnih orodjih, knjižnicah, platformah, sistemskih zahtevah, konfiguracijah, parametrih, ukazih, skriptah, korakih in drugih podrobnostih. Tehnična dokumentacija namestitve naj uporablja standardni format, kot je Markdown.

Dokumentacija naj uporablja ustrezna orodja, knjižnice, okvirje ali vtičnike za generiranje, testiranje, analizo in objavo dokumentacije. DRO teh orodij ne predpisuje. Orodja, knjižnice, okvirje ali vtičnike

je treba izbrati glede na programski jezik in platformo. Nekatera možna orodja, knjižnice, okvirje ali vtičnike so: Doxygen, Sphinx, SwaggerUI, Redoc, JSDoc, MkDocs, itd.

4.3.2 Pakiranje v slike vsebnikov

Aplikacije se morajo za izvajanje na DRO Kubernetes (in za DRO Docker Swarm) zapakirati v obliki slike vsebnikov (Docker images).

Pri pripravi slik vsebnikov (Docker images) je potrebno slediti naslednjim konceptom:

- Slike morajo biti pripravljene tako, da so iste slike uporabne v vseh okoljih (test, UAT, produkcija).
 - Priporoča se uporaba večstopenjske gradnje (multi-stage build) slik.
- Eksternalizirana konfiguracija mora biti dokumentirana preko nastavitve okoljskih spremenljivk.
 - Preko okoljskih spremenljivk se nastavljajo vse odvisnosti, povezave, URLji, dostopi do baze, API ključi in ostalo.
 - Vse nastavitve z občutljivimi podatki (uporabniška imena, gesla ipd.) se zapišejo v obliki skrivnosti (secret).
 - Opcijsko se lahko uporablja centralni konfiguracijski strežnik.
- Specificirani morajo biti porti, ki jih vsebnik izpostavlja.
- Določena mora biti oznaka slike (tag). Priporočeno je, da se preko oznake tudi verzionira slike.
- Priporočena je uporaba zgostitvene (hash) vrednosti.
- Uporabljene naj bodo labele, s katerimi se sliki dodajo metapodatki.
- Specifikacijska datoteka za gradnjo slike (npr. DOCKERFILE) mora vsebovati vse zahtevane elemente.

Slike vsebnikov je dovoljeno odlagati izključno na DRO-jev repozitorij vsebnikov/slik (glej ustrezno poglavje v tem dokumentu).

Priprava slik vsebnikov:

- Pri pripravi slik vsebnikov je potrebno obvezno uporabiti izhodišče slike (base images, FROM) iz nabora za DRO certificiranih slik, ki so podana za vsako izvajalno okolje oz. podprt operacijski sistem.
- Podroben seznam DRO certificiranih izhodiščnih slik je specificiran v ločenem dokumentu.
- Uporaba ostalih slik iz javnih repozitorijev (npr. Docker Hub) ni dovoljena.
- Prav tako ni dovoljeno odlaganje slik vsebnikov aplikacij na javne repozitorije vsebnikov/slik (npr. na Docker Hub).

4.3.3 Namestitev na Kubernetes

Nameščanje aplikacij na okolje Kubernetes:

- Zahtevana je uporaba namestitvenih deskriptorjev za Kubernetes.
- Priporočena je uporaba Helm Chartov.

Aplikacija mora specificirati stroke (pode), ki so osnovne namestitvene enote. Običajno se uporablja strategije ene mikrostoritve na en strok. V primeru, da aplikacija ne uporablja mikrostoritev, se v strok pakira ena monolitna celota. Najpomembnejša priporočila pri oblikovanju strokov (podov):

- Za zagotavljanje visoke razpoložljivosti in skalabilnosti uporabimo Replicaset ali Deployment.
- Za izpostavljanje vaših podov drugim podom ali zunanjemu svetu uporabimo Service ali Ingress krmilnik.
- Za označevanje in povezovanje strokov z drugimi objekti uporabimo Label in Selector. Label je ključ-vrednost par, ki se doda stroku ali drugemu objektu, da ga opiše, identificira ali kategorizira. Selector je izraz, ki se uporablja za izbiro strokov ali drugih objektov, ki ustrezajo določenim labelam.
- Za omejevanje porabe virov za stroke uporabimo ResourceQuota in LimitRange. ResourceQuota je objekt, ki določa, koliko CPU, pomnilnika, diska, omrežja, objektov in drugih virov lahko uporabi vsak namespace ali strok. LimitRange je objekt, ki določa, koliko CPU, pomnilnika, diska, omrežja, objektov in drugih virov lahko zahteva ali porabi vsak strok ali njegov vsebnik.
- Za preverjanje vitalnosti delovanja (health checks) in pripravljenosti strokov uporabimo LivenessProbe in ReadinessProbe. LivenessProbe je mehanizem, ki preverja, ali strok še vedno deluje pravilno in ga po potrebi ponovno zažene. ReadinessProbe je mehanizem, ki preverja, ali strok lahko sprejema zahteve (promet) in ga po potrebi odstrani.

Ustrezno je potrebno uporabiti imenska področja (namespaces) in se pri tem držati naslednjih priporočil:

- Uporabiti je potrebno imenska področja (namespaces) za ločevanje in organiziranje virov v logične skupine, ki imajo skupen namen, kontekst ali lastnika.
- Uporabiti je potrebno smiselna in dosledna imena za imenska področja. Imena naj odražajo njihovo funkcijo, vsebino in obseg.
- Ne uporablja se privzetih imenskih področij.
- Priporoča se uporaba kvot virov (Resource Quotas) za omejevanje porabe virov na nivoju CPU, pomnilnika, diska, omrežja, objektov in drugih virov.
- Priporoča se uporaba label in anotacij za dodajanje metapodatkov in informacij.

Imena namespace-ov projektom (aplikacijami) dodeljuje upravitelj Kubernetes gruče, v skladu z predifiniranimi imenskimi standardi.

Ustrezno je potrebno načrtovati stroke (pode) in komunikacijo med njimi. Možnosti komunikacije strokov (podov) v Kubernetes gruči so naslednje:

- Znotraj vozlišča: Stroki, ki se nahajajo na istem vozlišču, lahko komunicirajo med seboj neposredno preko svojih IP naslovov in izpostavljenih vrat (ports).
- Med vozlišči: Stroki, ki se nahajajo na različnih vozliščih, lahko komunicirajo med seboj preko podomrežja (pod network), ki je virtualno omrežje, ki povezuje vse stroke v gruči.
- Navzven: Stroki, ki želijo komunicirati z zunanjim svetom, lahko uporabijo različne mehanizme, ki so na voljo v Kubernetes. Najpomembnejši od teh mehanizmov so:
 - Ingress: Ingress je priporočen način za upravljanje pravil za usmerjanje prometa do strokov na podlagi URL, domene, glave, telesa ali drugih kriterijev. Ingress deluje kot vmesnik med zunanjim in notranjim prometom in omogoča, da se stroki izpostavijo na fleksibilen in granularen način.
 - Service: Service je abstrakcija, ki definira logično skupino strokov, ki izvajajo isto funkcionalnost in politiko za dostop do njih. Service omogoča, da stroke naslavljamo z imenom in vrsto, ki sta neodvisna od strokov, ki jih izvajajo. Service tudi omogoča odkrivanje in uravnoteženje obremenitve znotraj in zunaj gruč. Service lahko izpostavi stroke na različne načine, kot so ClusterIP, NodePort, LoadBalancer ali ExternalName. V DRO se priporoča uporaba LoadBalancer.
 - NetworkPolicy: NetworkPolicy je objekt, ki določa, kako se lahko stroki povezujejo z drugimi podi ali zunanjimi entitetami.
 - Egress: način za upravljanje pravic klicanja zunanjih (predvsem internetnih) storitev.

Pri uporabi Helm Chartov je potrebno upoštevati

- Pravila poimenovanja in imenske konvencije.
- Ustrezno shemo verzioniranja.
- Uporaba subchartov za upravljanje odvisnosti.
- Uporaba label za naslavljanje virov.
- Ustrezno dokumentiranje chartov.
- Testiranje namestitve iz chartov.
- Uporaba funkcij za predloge (template functions).
- Posodobitev namestitev v primeru spremembe ConfigMap ali skrivnosti.
- Uporaba funkcije lookup za preprečevanje ponovnega generiranja skrivnosti.
- Priporoča se uporaba Helm v3 ali višje.

V vseh primerih je obvezno varno shranjevanje zaupnih podatkov, kot npr. uporabniška imena in gesla. V ta namen je potrebno uporabiti koncept *secret*-ov.

4.3.4 Centralni sistem za beleženje dnevniških zapisov

DRO ponuja centralni sistem za beleženje dnevniških zapisov OpenSearch kot centralni gradnik, ki ga morajo uporabljati vse aplikacije. Vloga in namen centralnega sistema za beleženje dnevniških

zapisov je zagotoviti enotno, varno, in učinkovito rešitev za zbiranje, shranjevanje, analizo, in prikaz dnevniških zapisov različnih aplikacij in storitev. Dnevniški zapisi so zapisi o dogodkih, ki se zgodijo med delovanjem aplikacij in storitev ter lahko vsebujejo pomembne informacije o njihovem stanju, delovanju, napakah, varnosti in uporabi.

Aplikacije zapisujejo dnevniške zapise na naslednji način:

- Aplikacije, ki v celoti ali delno izkoriščajo domorodno oblačno arhitekturo in se izvajajo v vsebnikih na okolju Kubernetes, zapisujejo dnevniške zapise na standardni izhod (stdout za splošne in informativne zapise in stderr za napake in opozorila).
- Starejše monolitne aplikacije lahko pišejo dnevniške zapise v datoteke.

DRO bo zajel dnevniške zapise in jih prenesel v centralno zbirko na OpenSearch.

Aplikacijske rešitve morajo uporabljati standardni nabor podatkov v logu, kot je specificiran v DRO dokumentaciji o logiranju. Za zapisovanje dnevniških zapisov morajo aplikacije uporabiti ustrezno ogrodje za beleženje dnevniških zapisov v izbranem programskem jeziku oz. ogrodju (npr. log4j2 v Javi).

Aplikacijske rešitve morajo zagotoviti sledljivost dnevniških zapisov skozi nivoje. To še posebej velja za aplikacije, ki uporabljajo mikrostoritve. To zagotovijo z vključitvijo enoličnega identifikatorja v dnevniške zapise, ki se uporabi v sledi klica posameznih storitev. Enolični identifikator lahko generirajo same, ali uporabijo Open Telemetry.

Aplikacija mora imeti možnost konfiguriranja cilja, formata, ravni in frekvence pošiljanja dnevniških zapisov.

4.3.5 Smernice za uporabo repozitorija git

Predvidena je uporaba repozitorija izvirne kode git. Razvijalec aplikacije prenese celotno izvirno kodo aplikacije na DRO git. Pri tem se je potrebno držati priporočil oblikovanja repozitorijev, poimenovanja, uporabe vej (branches), oblikovanja zahtevkov (pull request), označevanja (tags) in verzioniranja.

Priporočeno je, da so posamezne komponente/mikrostoritve aplikacije v ločenih repozitorijih. Tudi poimenovanje repozitorijev naj sledi dogovorjenim imenskim konvencijam in pravilom.

DRO ne predpisuje uporabe določenega git delovnega toka (workflow), tako da ga lahko izberejo razvijalci aplikacij po lastni izbiri. Kljub temu se priporoča uporaba delovnega toka *Trunk-Based Development*. *Trunk-Based Development* je sodoben delovni tok, ki vključuje uporabo enotne veje (trunk ali master branch) in kratkotrajnih vej (feature branches) za različne funkcionalnosti. Pristop je primeren za uporabo v navezi z avtomatiziranim ciklom CI/CD. Osnovne usmeritve za *Trunk-Based Development* so:

- Uporablja ene glavne veje (trunk ali master), kjer vsi razvijalci sodelujejo in pogosto potrjujejo (commit) svoje spremembe s ciljem hitre in pogoste dostave kode v produkcijo, zmanjševanja konfliktov in izboljšanje kakovosti kode.
- Uporaba kratkotrajnih vej (branches) za razvoj posameznih funkcionalnosti, popravkov ali eksperimentov, ki se nato združujejo (merge) v glavno vejo s pomočjo zahtevkov za poteg (pull requests) in pregledov kode (code reviews). To omogoča ločevanje in izolacijo dela posameznih razvijalcev in preverjanje spremembe, preden so vključene v glavno vejo.
- Uporaba oznak (tags) za označevanje določenih točk v zgodovini kode, ki predstavljajo pomembne mejnike, kot so izdaje (releases), različice (versions), stabilna stanja (snapshots) ali kandidati za izdajo (release candidates). To omogoča enostavno lociranje, referenciranje in primerjavo kode na teh točkah ter sledenje napredka in sprememb.

Kot del izvorne kode je potrebno predati tudi:

- Kodo za gradnjo slik vsebnikov (Docker) – npr. Dockerfile,
- Namestitvene deskriptorje za Kubernetes in Helm charte.

Cilj je, da se prenesena izvorna koda na git v okolju DRO zgradi (builda), prav tako se v okolju DRO generirajo slike vsebnikov in odložijo artefakti gradnje. Na osnovi tega se izvede namestitev.

Opomba: Uporaba obstoječega repozitorija SVN je namenjena izključno za obstoječe (legacy) aplikacije, za katere pa se priporoča, da se jih postopno migrira na git.

4.3.6 Avtomatiziran CI/CD in gradnja slik vsebnikov

Vsi projekti aplikacij morajo imeti definiran avtomatiziran cevovod za CI/CD, v okviru katerega se:

Obvezni del:

- zgradi (build) aplikacija,
- artefakti se odložijo na repozitorij artefaktov,
- v okviru gradnje aplikacije se izvedejo tudi:
 - testi enot,
 - integracijski testi in
 - generiranje dokumentacije
- generirajo slike vsebnikov (Docker),
- slike vsebnikov se odložijo na register slik vsebnikov,
- pripravijo vsi ostali potrebni elementi, ki so potrebni za namestitev (namestitveni deskriptorji, konfiguracije ipd.).

Opcijski del:

- izvede se (avtomatizirana) namestitev na testno okolje,

- v okviru namestitve se naredijo tudi ustrezne posodobitve (npr. posodobitve strukture podatkovne baze z uporabo orodij za migracijo shem ipd.),
- konfiguracija v centralnem konfiguracijskem strežniku (če se uporablja),
- registracija virov, kot npr. registracija API-jev na API prehodu in podobno,
- objava tehnične dokumentacije.

Aplikacije naj imajo vnaprej pripravljeno `.gitlab-ci.yml` datoteko, s katero specificirajo celoten postopek CI/CD. Pri tem naj sledijo dobrim praksam in smernicam okolja DRO, uporabijo pa lahko tudi predloge in okolja, ki jih ponuja DRO.

Pri tem je potrebno upoštevati smernico, da se **slike vsebnikov (Docker) generirajo samo enkrat** in se iste slike uporabijo za vsa okolja (test, UAT, produkcijo, ...).

4.3.7 Centralni register slik vsebnikov

Vse slike vsebnikov se shranjujejo na centralni register slik vsebnikov, ki ga ponuja DRO. Pri tem se uporabi konsistentna shema poimenovanja, kot npr. `<projekt>/<storitev>:<verzija>`. Shranjevanje slik vsebnikov na javne repozitorije (npr. Docker Hub) ni dovoljeno. Za podrobnejše smernice gradnje slik glej poglavje o gradnji slik vsebnikov v tem dokumentu.

4.3.8 Centralni repozitorij programskih artefaktov

Vse programske artefakte, ki nastanejo tekom gradnje (build) aplikacije, vse zasebne knjižnice in ostale odvisnosti, ki niso javno dostopne, se shranjujejo na centralni repozitorij programskih artefaktov. Glede na uporabljen programski jezik in shemo verzioniranja se naj uporabi konsistentno poimenovanje artefaktov. Shranjevanje artefaktov na javne repozitorije (npr. Maven Central ipd.) ni dovoljeno.

4.3.9 Pravila za podatkovne baze (vzeto iz obstoječega GTZ)

Pri oblikovanju dostopov do baze je zahteva upravljalca infrastrukture DRO, da se aplikacija/modul na bazo prijavlja z računom (account/uporabnik) s tistim minimalnim naborom pravic, ki aplikaciji še omogoča delovanje oziroma izvrševanje poslovnih funkcij, ki jih aplikacija implementira. Direktnih povezav do baz v kodi ni dovoljeno vzpostavljati, vedno je potrebno iti preko koncepta bazena povezav (connection pool).

Do podatkovne baze lahko dostopajo samo storitve iz nivoja poslovne logike ali integracijskega nivoja. Kakršnikoli dostop iz nivojev uporabniških vmesnikov direktno do baze niso dovoljeni.

Storitev uporablja za dostop do podatkovne baze povezave iz bazena povezav, ki imajo minimalnim nabor pravic, ki storitvi/aplikaciji še omogoča delovanje oziroma izvrševanje poslovnih funkcij, ki jih storitev implementira.

4.3.10 Ostala priporočila

4.3.10.1 Varnostni vidiki

Ta dokument podaja nekatere smernice za razvijalce aplikacij, ki jih je smiselno nasloviti za potrebe zagotavljanja varnosti razvitih aplikacij. Opomba: ta dokument je namenjen arhitekturnim smernicam domorodnih oblačnih aplikacij in zato ne vsebuje vseh smernic in vidikov za zagotavljanje varnosti. Le-ti so specificirani v ločenih dokumentih DRO.

Priporočeno je, da aplikacije oz. posamezne mikrostoritve, upoštevajo naslednje smernice zagotavljanja varnosti:

- Za vsako mikrostoritev je pripravljena analiza tveganj ob varnostnih incidentih.
- Vgrajevanje varnostnih vidikov se uporablja na vseh korakih načrtovanja in implementacije aplikacije kot celote in posameznih mikrostoritev. Tak princip je poznan tudi pod imenom Security by design.
- Na vseh nivojih se uporablja princip najnižjih možnih pravic (Principle of Least Privilege (POLP)). To velja tako za izvajanje mikrostoritev, kakor tudi za pravice pri klicih odvisnosti (drugih storitev, baz podatkov, sporočilnih in dogodkovnih sistemov ipd.).
- Pri razvoju je potrebno uporabljati in upoštevati smernice za razvoj varne izvorne kode vsakega uporabljenega programskega jezika.
- Vse komunikacije med mikrostoritvami znotraj aplikacije in vse komunikacije navzven se morajo realizirati preko varnih kriptiranih komunikacijskih kanalov in protokolov (npr. HTTPS) z ustrezno avtentikacijo in avtorizacijo.
- Vse API-ji in ostale končne točke morajo uporabljati avtentikacijo in avtorizacijo uporabnikov z uporabo vlog ali dodeljevanja pravic na osnovi virov.
- Občutljive podatke je potrebno hraniti v kriptirani obliki v podatkovni bazi ali drugih storitvah za hrambo. Kriptiranje prevzame aplikacija. V takem primeru mora aplikacija posebej specificirati mehanizme za upravljanje ključev (generiranje, shranjevanje ključev, njihovo varnostno kopiranje in obnavljanje), podati podrobne podatke o uporabljenih enkripcijskih algoritmi in njihovih parametrih, specifikat, kateri podatki so kriptirani ter sam proces enkripcije/dekripcije, obravnavo napak in morebiten vpliv na zmogljivosti delovanja ter kakšne so zahteve za izdelavo varnostnih kopij.
- Uporabljati je potrebno ustrezno varne enkripcijske algoritme, mehanizme za generiranje naključnih števil in pravilno izbrane dolžine ključev.
- Upoštevati je potrebno splošno sprejete smernice, kot npr. OWASP.
- Potrebno je povzeti vse ukrepe, ki so potrebni, da se ustrezno zaščiti postopek gradnje (build).
- Vse uporabniška imena, gesla in druge občutljive podatke je potrebno hraniti izključno v varni shrambi, npr. z uporabo koncepta skrivnosti (secret).
- Aplikacijo kot celoto in posamezne mikrostoritve je potrebno podvreči rigoroznemu varnostnemu testiranju.

- V okviru spremljanja izvajanja delovanja aplikacije in posameznih mikrostoritev je potrebno spremljati in zaznavati varnostne incidente.

4.3.10.2 Varnostno preverjanje kode

Priporoča se, da razvijalci aplikacij le-te pred predajo samostojno varnostno preverijo. Varnostno preverjanje se bo izvedlo tudi pred namestitvijo aplikacije na DRO. Podrobneje so zahteve glede varnostnega preverjanja zapisane v dokumentu *Navodilo projektnim vodjem za varnostna preverjanja – Zahteve*.

4.3.10.3 Obremenilni testi

Priporoča se, da razvijalec aplikacije pred predajo izvede obremenilne teste. Pri tem se priporoča uporaba orodij, kot sta JMeter in LoadRunner.

4.3.10.4 Preverjanje kakovosti kode

Za preverjanje kakovosti kode se priporoča uporaba orodij, kot so Syft Bill of material, SonarCube Code quality check in Dependency check. Priporočilo je, da razvijalci aplikacij izvedejo ustrezno preverjanje kakovosti kode kot del build cikla že pri sebi. Ko bo koda prenesena v DRO, se bo to preverjanje opravilo tudi s strani DRO.

4.4 Zahteve za monolitne aplikacije, ki ne sledijo domorodni oblačni arhitekturi

Monolitne aplikacije ne izpolnjujejo zahtev domorodne oblačne arhitekture, niti niso prilagojene za izvajanje v okolju Kubernetes (ali Docker Swarm). V naslednjem poglavju podajamo zbirno tabelo zahtev po tipih aplikacij, iz katere je razvidno, kakšne karakteristike imajo monolitne aplikacije.

Monolitne aplikacije razumemo kot obstoječe (legacy) aplikacije, ki jih bo v prihodnosti potrebno predelati in ali nadgraditi ali na novo razviti po konceptih domorodnih oblačnih aplikacij.

4.5 Zbirna tabela zahtev po tipih aplikacij

Tabela v nadaljevanju podaja pregled zahtev po tipih aplikacij (slika 8).

Slika 9: Zbirna tabela zahtev po tipih aplikacij

Zahteva	Aplikacije, ki v celoti izkoriščajo domorodno oblačno arhitekturo	Aplikacije, ki delno izkoriščajo domorodno oblačno arhitekturo	Monolitne aplikacije, ki ne sledijo domorodni oblačni arhitekturi
Arhitekturna zasnova	V celoti mikrostoritvena, večslojna	Storitvena, troslojna	Monolitna
Uporaba API-jev	Da, v celoti (REST z OpenAPI 3, gRPC, GraphQL)	Delno (REST in SOAP)	Ne ali v zelo majhnem obsegu

Zahteva	Aplikacije, ki v celoti izkoriščajo domorodno oblačno arhitekturo	Aplikacije, ki delno izkoriščajo domorodno oblačno arhitekturo	Monolitne aplikacije, ki ne sledijo domorodni oblačni arhitekturi
Uporaba sporočilnih sistemov	Opcijsko	Opcijsko	Opcijsko
Uporaba pretočnih dogodkov	Da, opsijsko	Opcijsko	Ne
Uporaba API prehodov	Da	Ne	Ne
Verzioriranje izvirne kode, API-jev, artefaktov in izdaj	V celoti skladno s smernicami	Delno skladno s smernicami	Vsaj verzioriranje izvirne kode
Uporaba principov 12 faktorskih aplikacij	Da, v celoti	Delno	Ne
Preverjanje vitalnosti (Health Check)	Da	Ne ali opsijsko	Ne
Zbiranje metrik	Da, opsijsko	Ne	Ne
Odprta telemetrija	Da, opsijsko	Ne	Ne
Odpornost na napake	Da, opsijsko	Ne	Ne
Uporaba storitvenega omrežja	Da, opsijsko	Ne	Ne
Vzorci uporabe podatkovnih baz	Skupna baza Ločena baza po storitvi	Skupna baza	Skupna baza
Pravila dostopa do podatkovnih baz	Da	Da	Da
Orodja za migracijo shem	Da	Opcijsko	Ne
Centralni avtentikacijski in avtorizacijski strežnik (IAM)	Da	Delno ali ne	Ne
Spletni uporabniški vmesniki	Odzivni, SPA ali PWA	Oboji	Klasični
Konfiguracija	Okoljske spremenljivke ali konfiguracijski strežnik	Okoljske spremenljivke	Datoteka ali opsijsko okoljske spremenljivke
Avtomatiziran postopek gradnje (build)	Da, podprte vse zahtevane faze	Ne ali delno	Ne ali delno
Testi enot in integracijski testi	Da, v celoti	Delno	Ne ali opsijsko

Zahteva	Aplikacije, ki v celoti izkoriščajo domorodno oblachno arhitekturo	Aplikacije, ki delno izkoriščajo domorodno oblachno arhitekturo	Monolitne aplikacije, ki ne sledijo domorodni oblachni arhitekturi
Generiranje tehnične dokumentacije	Da, v celoti	Ne ali opcijsko	Ne
Pakiranje v slike vsebnikov	Da, skladno s smernicami	Da	Ne
Namestitev na Kubernetes	Namestitveni deskriptorji ali Helm charti	Namestitveni deskriptorji	Ne
Izvajanje v okolju Kubernetes	Da, izkoriščajo vse ali večino funkcionalnosti	Da, omejeno	Ne
Centralni sistem za beleženje dnevniških zapisov	Da, direktno pošiljanje ali pridobivanje iz stdout	Da, pisanje v datoteko	Da, pisanje v datoteko
Uporaba repozitorija git	Da, skladno s smernicami	Da, delno skladno	Ne (uporaba SVN)
Avtomatiziran CI/CD	Da, v celoti	Da, delno	Ne
Centralni register slik vsebnikov	Da	Da	Ne
Centralni repozitorij artefaktov	Da	Opcijsko	Ne
Avtomatizirana vzpostavitev okolja z uporabo IaC	Da	Ne ali delno	Ne

5 Tehnološke zahteve, programski jeziki, orodja in ogrodja

V tem poglavju so zbrane tehnološke zahteve, ki specificirajo uporabo tehnologij, programskih jezikov, ogrodij in orodij ter njihovih verzij.

Tehnološke zahteve veljajo smiselno za vse tipe aplikacij, pri čemer za posamezni tip aplikacij upoštevamo tiste sklope, ki jih aplikacija dejansko uporablja.

5.1 Programski jeziki in ogrodja za razvoj

V tem poglavju podajamo informativni pregled podprtih programski jezikov in ogrodij za razvoj. Podrobna specifikacija trenutno podprtih tehnologij in verzij je podana v dokumentu Podprta programska oprema DRO, ki je priloga GTZ in je dostopen na Portalu NIO (<https://nio.gov.si>).

5.1.1 Podprti programski jeziki za zaledni del

Za razvoj zalednega dela poslovne logike so podprti naslednji programski jeziki in ogrodja:

- Java:
 - Java Standard Edition s podporo za vsakokratno tekočo in eno verzijo starejšo LTS različico Java na OpenJDK izvajalnem okolju (<https://openjdk.java.net/>)
 - Podpora za mikrostoritvena ogrodja
 - Spring Boot, Quarkus,
 - Eclipse Microprofile.
 - Podpora za odprtokodna Java in Jakarta EE izvajalna okolja:
 - Wildfly, <https://www.wildfly.org/>
 - Tomcat, <https://tomcat.apache.org/>
 - Podpora za licenčna Java in Jakarta EE izvajalna okolja oz. aplikacijske strežnike:
 - Red Hat JBoss EAP, <https://www.redhat.com/en/technologies/jboss-middleware/application-platform>
 - Oracle Weblogic, <https://www.oracle.com/java/weblogic/>
 - IBM WebSphere, <https://www.ibm.com/products/websphere-application-server>
 - Build orodja
 - Maven,
 - Gradle
- .NET (vključuje C#, F# in VisualBasic)
 - Podpora za aplikacije na .NET ogrodju tekoče in eno verzijo starejše LTS različice
- JavaScript - Node.JS
 - Podpora za aplikacije v Node.JS za tekoče in eno verzijo starejše LTS različice
- Python
 - Podpora za zadnji dve stabilni verziji

- Go
 - Podpora za zadnji dve stabilni verziji

Za monolitne aplikacijske rešitve so podprti tudi:

- PHP s strežniškimi izvajalnimi okolji
 - Apache: <https://www.php.net/manual/en/book.apache.php>
 - Nginx: <https://www.nginx.com/resources/wiki/start/topics/examples/phpfcgi/>
- PL/SQL z naslednjimi izvajalnimi okolji
 - Oracle RDBMS
 - Oracle APEX, zadnja LTS verzija <https://apex.oracle.com/en/>

5.1.2 Podprta programska ogrodja za spletne aplikacije (spletne uporabniške vmesnike)

Za razvoj spletnih aplikacij oz. spletnih uporabniških vmesnikov so podprta naslednja ogrodja:

- Angular
 - Zadnji dve LTS verziji
- React
 - Zadnji dve LTS verziji
- Vue.js
 - Zadnji dve LTS verziji

V vseh primerih se uporablja ECMAScript/JavaScript oz. TypeScript.

Zahteve glede podpore spletnih brskalnikov so specificirane v GTZ.

5.1.3 Podprta programska ogrodja za mobilne aplikacije

Za razvoj mobilnih aplikacij so podprta naslednja programske ogrodja za native razvoj aplikacij. V tem primeru se vzdržuje dva ločena repozitorija izvirne kode, enega za Android, enega za iOS:

- Android z uporabo Kotlin
- iOS z uporabo Swift (pogojno Objective C)

Za razvoj mobilnih aplikacij so podprta tudi naslednja programske ogrodja za več-platformski razvoj mobilnih aplikacij iz enotne izvirne kode:

- Flutter
- React Native

V slednjem primeru se iz skupne izvirne kode zgradijo mobilne aplikacije za iOS in Android.

5.1.4 Odlaganje artefaktov

Vsi artefakti, ki nastanejo kot del postopka gradnje aplikacije (build) se morajo odlagati na interni repozitorij artefaktov DRO. Odlaganje na javne repozitorije (npr. Maven Central, Gradle Central Repository, npm registry, ipd.) ni dovoljeno.

5.2 Podatkovne baze in trajno stanje podatkov

DRO podpira naslednje sisteme za upravljanje s podatkovnimi bazami (DBMS) kot centralne gradnike, ki jih lahko uporabljajo aplikacijske rešitve brez potrebe po namestitvi lastne instance baze, saj navedene zbirke upravlja DRO in jih aplikacijam ponuja kot storitev (as a Service).

5.2.1 Licenčni relacijski sistemi za upravljanje podatkovnih baz

- MS SQL, centralni infrastrukturni gradnik, v2019, v2022
Centralna zbirka je nameščena v visoko-razpoložljivem režimu, podatki porazdeljeni preko treh instanc nameščenih na primarni in nadomestni lokaciji;
- Oracle RDBMS, zadnja LTS verzija,
Centralna zbirka je upravljana v visoko-razpoložljivem načinu, z dataguard zrcalno asinhrono zbirko na nadomestni lokaciji, z možnostjo vzpostavitve sinhrono replikacije in instance v zmogljivi namenski strojni opremi (Exadata);

5.2.2 Odprtokodni relacijski sistemi za upravljanje podatkovnih baz

- PostgreSQL, v14.1 ali višje, <https://www.postgresql.org/> (priporočljivo)
- MySQL, v8.0 ali višje, <https://www.mysql.com/>
- MariaDB, v10.8 ali višje, <https://mariadb.org/>

5.2.3 Nerelacijski sistemi za upravljanje podatkov in objektne zbirke

Odprtokodne nerelacijske zbirke:

- MongoDB, <https://www.mongodb.com/>
- Redis, <https://redis.io/>
- Elasticsearch, <https://www.elastic.co/elastic-stack/>
- S3 (Minio), <https://min.io/>

5.2.4 Ostale podatkovne baze in podatkovne zbirke

Aplikacije lahko uporabljajo tudi druge podatkovne baze ali podatkovne zbirke, ki niso našteje v tem seznamu. Uporaba takih rešitev je stvar predhodne odobritve s strani naročnika.

5.3 Tehnologije za aplikacijske programske vmesnike - API

Podprta je uporaba naslednjih tehnologij za aplikacijske programske vmesnike:

- REST API
 - REST API je priporočena tehnologija aplikacijskih programski vmesnikov za DRO.
- SOAP in WSDL
 - Uporaba storitev SOAP je podprta predvsem za obstoječe (legacy) rešitve.
- gRPC
- GraphQL

V povezavi z API tehnologijami je potrebno obvezno uporabiti API prehod (glejte poglavje o API prehodih).

5.4 Sporočilni in dogodkovni sistemi

Podprti so naslednji sporočilni sistemi (Message Oriented Middleware):

- AMQP
 - Podprta je uporaba protokola AMQP.
 - DRO ponuja RabbitMQ v obliki platformske storitve. Aplikacijska rešitev lahko uporabi centralno postavitev RabbitMQ.
 - Podprta je tudi uporaba drugih sporočilnih sistemov, vendar mora v tem primeru ponudnik aplikacijske rešitve namestiti in vzdrževati svojo instanco sporočilnega sistema.
 - V vsakem primeru mora aplikacija, ki uporablja sporočilni sistem, definirati:
 - Vrste (queue, topic, ...), ki jih uporablja, vključno z natančnim poimenovanjem.
 - Strukturo in format sporočil, ki jih pošilja.
 - Pravila za obravnavo sporočil v DLQ (dead-letter queue).
- NATS (Neural Autonomic Transport System)
 - Podprta je uporaba NATS.
 - Aplikacijska rešitev, ki uporablja NATS, mora definirati:
 - Predmete (subject) in njihovo poimenovanje.
 - Strukturo in format sporočil, ki jih pošilja.

Podprt je naslednji sporočilni sistem (sistem za pretočne dogodke – event streaming):

- Kafka
 - Podprta je uporaba platforme Kafka.
 - DRO ponuja Kafka kot centralni gradnik v obliki platformske storitve. Aplikacijska rešitev lahko uporabi centralno postavitev Kafka.

- Aplikacija mora specificirati:
 - Topice, ki jih uporablja ter njihovo poimenovanje.
 - Particije za posamezne topice.
 - Strukturo in format sporočil, ki jih pošilja.
 - Časovno okno ohranjanja dogodkov/sporočil v topicu.
 - Morebitne ostale zahteve.

Za potrebe uporabe IoT (Internet of Things) naprav je priporočena uporaba protokola MQTT. DRO trenutno ne podpira centralnega gradnika v smislu MQTT sporočilnega sistema, zato mora za tak sistem poskrbeti vsaka aplikacija zase.

5.5 Izvajalna okolja za aplikacije

5.5.1 Izvajalna okolja

DRO podpira naslednja izvajalna okolja:

- Kubernetes
 - Kubernetes je preferirano izvajano okolje za domorodne oblačne aplikacije in vse ostale aplikacije, ki jih lahko pakiramo v obliki vsebnikov (Docker) in izvajamo v Kubernetesu.
 - Znotraj okolja Kubernetes so na voljo tudi naslednji gradniki:
 - Prometheus, CAdvisor in druga orodja
 - Grafana (ki je lahko nameščena centralno)
 - Opcijsko je omogočena uporaba service mesh Istio.

Za potrebe obstoječih (legacy) aplikacij oz. aplikacij, ki ne izkoriščajo v celoti domorodne oblačne arhitekture, sta podprta še:

- Docker Swarm in navadni Docker runtime
 - Namenjen je izvajanju obstoječih aplikacij, ki trenutno delujejo na Docker Swarm
 - Aplikacije iz Docker Swarm se bodo postopno selile v okolje Kubernetes
- Klasični navidezni strežniki – VM
 - Klasični navidezni strežniki so namenjeni izvajanju obstoječih aplikacij. Obstoječe aplikacije, ki delujejo na VM se bodo postopno selile v okolje Kubernetes.
 - Klasični navidezni strežniki so namenjeni tudi izvajanju specifične programske opreme, kot na primer gručam podatkovnih baz in ostalih sistemov, ki arhitekturno niso prilagojeni za izvajanje v vsebnikih in okolju Kubernetes.
 - Uporabljena virtualizacija je VMware.

5.5.2 Tehnologije vsebnikov in priprava slik vsebnikov

Podprta tehnologija vsebnikov (containers) je:

- CRI-O (Container Runtime Interface).

V prihodnosti bo lahko dodana še podpora za dodatne tehnologije vsebnikov.

5.5.3 Nameščanje aplikacij

Nameščanje aplikacij na okolje Kubernetes:

- Zahtevana je uporaba namestitvenih deskriptorjev za Kubernetes.
- Priporočena je uporaba Helm Chartov.

Za nameščanje ostalih aplikacij so podprte:

- Docker stack deploy za aplikacije, ki uporabljajo Docker in se nameščajo na SWARM.
- Skripte za nameščanje aplikacij na klasične virtualne strežnike.

5.5.4 Vzpostavitev okolja – IaC in GitOps

Za vzpostavljanje infrastrukturnih in platformskih gradnikov okolja ter za avtomatizacijo namestitve aplikacij je potrebno dosledno uporabljati koncepte Infrastructure-as-Code (IaC). Podprti orodji sta:

- Ansible in
- Terraform oz. OpenTofu.

Izvajalci aplikacijskih rešitev morajo pripraviti delujoče skripte za avtomatizacijo namestitve aplikacij in jih predati kot del projekta, skrbnik infrastrukture DRO pa jih bo integriral in uporabil. Priložena naj bo ustrezna dokumentacija. Z uporabo skript naj bo pokrita celotna vzpostavitev in namestitev vseh delov (komponent) aplikacijske rešitve, pri tem pa predpostavlja, da bo namestitev potekala v ločeno imensko področje na Kubernetes-u.

Obvezna je uporaba koncepta GitOps. Hranjenje vseh občutljivih podatkov (gesel in podobno) z uporabo skrivnosti (secretov).

5.6 Repozitorij izvirne kode, repozitorij artefaktov, DevOps in avtomatizacija CI/CD

5.6.1 Repozitorij izvirne kode

Preferiran repozitorij izvirne kode za DRO je:

- Git

Vse aplikacije in aplikacijski sistemi odlagajo izvorno kodo in ostale artefakte v git. Pri tem sledijo pravilom strukture, označevanja, vej in poimenovanja, ki je določen v ločenem dokumentu.

Za obstoječe (legacy) aplikacijske sisteme je na voljo tudi obstoječi sistem:

- SVN

SVN je namenjen samo obstoječim aplikacijam. Predvideva se, da se nove (major) verzije obstoječih aplikacij selijo iz SVN na git.

5.6.2 Centralni repozitorij vsebnikov/slik in artefaktov

Centralni repozitorij vsebnikov/slik:

- Docker Harbor

Vse slike vsebnikov je potrebno shranjevati na repozitorij DRO. Uporaba slik direktno iz javnih repozitorijev ni dovoljena. Za oblikovanje novih slik je potrebno uporabiti izhodišče slike (base images, FROM) iz nabora za DRO certificiranih slik, ki so podana za vsako izvajalno okolje oz. podprt operacijski sistem.

Centralni repozitorij programskih artefaktov

- Sonatype Nexus

Vsi artefakti, ki nastanejo kot del postopka gradnje aplikacije (build) se morajo odlagati na interni repozitorij artefaktov DRO. Odlaganje na javne repozitorije (npr. Maven Central, Gradle Central Repository, npr. registry, ipd.) ni dovoljeno.

5.6.3 Avtomatizacija CI/CD in DevOps

Preferiran sistem za avtomatizacijo CI/CD je:

- GitLab CI/CD

Aplikacije naj uporabljajo Gitlab CI/CD in vnaprej pripravijo .gitlab-ci.yml datoteko, s katero specificirajo celoten postopek CI/CD. Pri tem naj sledijo dobrim praksam in smernicam okolja DRO, uporabijo pa lahko tudi predloge in okolja, ki jih ponuja DRO.

Za potrebe obstoječih (legacy) aplikacijskih rešitev bo DRO še nekaj časa podpiral:

- Jenkins

Jenkins se naj uporablja samo za obstoječe rešitve. Ob selitvi na git in novih (major) verzijah se naj predela tudi CI/CD cevovod na uporabo GitLab CI/CD.

5.7 Platformske storitve DRO

DRO ponuja naslednje platformske storitve, za katere je priporočeno oz. obvezno, da jih uporabijo aplikacijske rešitve:

- Centralni avtentikacijski in avtorizacijski strežnik (IAM – Identity and Access Management)
- Centralni sistem za zbiranje dnevniških zapisov
- API prehod in sistem za upravljanje API-jev
- Centralni konfiguracijski strežnik

5.7.1 Centralni avtentikacijski in avtorizacijski (IAM) strežnik

Centralni avtentikacijski in avtorizacijski IAM strežnik v DRO nudi podporo za registracijske in prijavnne tokove za uporabnike in za upravljanje identitet ter nadzora dostopa. DRO kot avtentikacijski in avtorizacijski IAM strežnik uporablja:

- Keycloak - <https://www.keycloak.org/>

5.7.2 Centralni sistem za beleženje dnevniških zapisov

DRO ponuja centralni sistem za beleženje dnevniških zapisov kot centralni gradnik, ki ga morajo uporabljati vse aplikacije:

- OpenSearch

Aplikacije morajo pošiljati dnevniške zapise na centralni sistem za beleženje dnevniških zapisov:

- Z neposrednim pošiljanjem dnevniških zapisov v sistem OpenSearch.
- Z odlaganjem dnevniških zapisov v datoteko in pobiranjem dnevniških zapisov iz datoteke ter pošiljanjem v centralni sistem za beleženje dnevniških zapisov.

5.7.3 API prehod in sistem za upravljanje API-jev

DRO ponuja centralni API prehod (API gateway) in centralni sistem za upravljanje API-jev.

- API prehod:
 - API prehod ima tri instance in je vzpostavljen za naslednje komunikacijske poti:
 - Med aplikacijskimi sistemi znotraj DRO
 - Med aplikacijami v sistemu javne uprave znotraj HKOM
 - Do zunanjih sistemov (zunaj HKOM v javnem Internetu)
 - Fizično je lahko postavljen tako, da je API prehod implementiran znotraj posamezne Kubernetes gruče.
- Sistem za upravljanje API-jev

5.7.4 Centralni konfiguracijski strežnik

DRO ponuja naslednji centralni konfiguracijski strežnik:

- Etcd - <https://etcd.io/>

Etcd je odprtokodni projekt, ki temelji na algoritmu Raft, ki omogoča konsistentno in odporno replikacijo podatkov med več vozlišči. DRO ponuja namensko, visokorazpoložljivo namestitev strežnika etcd za namen konfiguracije aplikacij, ki delujejo v DRO.

Opomba: Etcd je tudi ključna komponenta okolja Kubernetes. DRO za namene konfiguracije aplikacij ne uporablja systemskega etcd okolja Kubernetes, ampak ločeno namensko postavitev.

DRO postavitve etcd podpira protokola v2 in v3.

6 Smernice za migracijo monolitnih aplikacij v domorodno oblačno arhitekturo

Obstoječe (legacy) aplikacije in aplikacijske sisteme, ki temeljijo na monolitni arhitekturi, bo potrebno postopno migrirati v domorodno oblačno arhitekturo. V nadaljevanju podajamo kratek povzetek ključnih smernic v zvezi z migracijo.

6.1 Razumevanje omejitev obstoječe aplikacije in načrtovanje migracije

Pred začetkom migracije je ključnega pomena razumevanje arhitekture, delovanja, uporabljenih tehnologij in vsebinskega delovanja obstoječe aplikacije ter njenih odvisnosti do drugih aplikacij in sistemov. Na osnovi poglobljenega razumevanja je potrebno napraviti visokonivojski načrt migracije z oceno obsega, truda in količino zahtevanih predelav. Prav tako je smiselno oceniti učinke migracije in razumeti omejitve.

6.2 Sprejem odločitve, ali se predeluje obstoječo aplikacijo ali gre v nov razvoj

Na osnovi visokonivojskega načrta migracije je potrebno sprejeti odločitev:

- Ali je smiselna predelava obstoječe aplikacije,
- Ali je bolj smiselno na novo razviti aplikacijo (in sočasno nasloviti tudi dodatne funkcionalne zahteve ali tudi vsebinsko in funkcionalno modificirati aplikacijo).

Odločitev je smiselno sprejeti glede na vsebinske zahteve, stanje obstoječe aplikacije, kritičnost obstoječe aplikacije, stopnjo podpore in vzdrževanja, finančne in časovne omejitve ter druge relevantne kriterije.

6.3 Predelava obstoječe aplikacije

V primeru sprejetja odločitve za predelavo obstoječe monolitne aplikacije je potrebno sprejeti nadaljnjo odločitev, kako se lotiti predelave:

- Predelavo lahko izpeljemo kot postopno predelavo, pri kateri postopno predelujemo posamezne sklope aplikacije in jih spreminjamo v mikrostoritveno arhitekturo.
- Predelavo izvedemo kot celostno tehnološko posodobitev.

6.3.1 Postopna predelava

Pri postopni predelavi postopoma predelujemo ključne funkcionalnosti v mikrostoritve. Prav tako razvoj vseh novih funkcionalnosti implementiramo kot mikrostoritve. Na ta način se v aplikaciji postopno povečuje delež funkcionalnosti, implementiranih v obliki mikrostoritev in zmanjšuje delež

obstoječe monolitne aplikacije. Končni cilj je, da se monolitni del v celoti izloči in dobimo celotno aplikacijo v obliki mikrororitev.

6.3.2 Celostna tehnološka posodobitev

Pri celostni tehnološki posodobitvi monolitno aplikacijo predelamo v celoti v arhitekturo mikrororitev. Po končani posodobitvi prenesemo (migriramo) podatke in ostale elemente obstoječe aplikacije, posodobimo integracije ter začnemo uporabljati novo razvito aplikacijo, obstoječo monolitno aplikacijo pa izklopimo.

V obeh primerih pri implementaciji nove, domorodno oblačne aplikacije, sledimo vsem smernicam mikrororitvene in domorodne oblačne arhitekture. Pri tem si lahko pomagamo tudi s tem dokumentom, ki v celoti specificira zahteve za aplikacije, ki v celoti izkoriščajo domorodno oblačno arhitekturo.

7 Zaključek

Ta dokument definira referenčno arhitekturo aplikacijskih rešitev za DRO in podaja tehnološke zahteve, arhitekturne koncepte in smernice domorodne oblačne arhitekture za razvoj nove generacije aplikacij za DRO. Dokument definira zahteve za tri različne nivoje arhitekture aplikacijskih rešitev, glede na njihovo skladnost z domorodno oblačno arhitekturo (cloud-native arhitekturo):

- Aplikacije, ki v celoti izkoriščajo domorodno oblačno arhitekturo (DRO-NEXT),
- Aplikacije, ki delno izkoriščajo domorodno oblačno arhitekturo (in le-te aplicirajo samo najpomembnejše vzorce),
- Monolitne aplikacije, torej aplikacije, ki ne sledijo domorodni oblačni arhitekturi.

Najprej so opisani arhitekturni koncepti in smernice domorodnih oblačnih aplikacij, vključno z koncepti domorodne oblačne arhitekture, arhitekturo mikrostoritev, komunikacijo med mikrostoritvami, API-ji in API prehodi, sporočilnimi sistemi, sistemi za pretočne dogodke, vsebniki in okolji za orkestracijo vsebnikov, DevOps in avtomatizacija infrastrukture ter računalniški oblak s podpornimi storitvami.

Nato so definirani osnovni koncepti za posamezne tri tipe aplikacij, arhitekturne smernice in storitve, ki jih ponuja DRO. Predstavljena je visokonivojska arhitekturna DRO iz vidika razvijalcev aplikacij in arhitekturna slika DRO za domorodne oblačne aplikacije s ključnimi elementi, od izvajalnega okolja Kubernetes, preko komunikacije, varnosti, avtentikacije in avtorizacije, uporabe platformskih storitev oz. gradnikov, DevOps in CI/CD, uporabe IaC ter opisa okolij, ki jih ponuja DRO.

Sledi podroben opis arhitekturno tehnoloških zahtev za domorodne oblačne aplikacije, ki so opisane za aplikacije, ki v celoti izkoriščajo domorodno oblačno arhitekturo, aplikacije, ki delno izkoriščajo domorodno oblačno arhitekturo, skupne zahteve za oba tipa aplikacij ter zahteve za monolitne aplikacije, ki ne sledijo domorodni oblačni arhitekturi. Zbirna tabela zahtev po tipih aplikacij podaja pregleden zapis ključnih zahtev.

Nato so opisane tehnološke zahteve, programski jeziki, orodja in ogrodja, kar vključuje programske jezike in ogrodja za razvoj, podatkovne baze in trajno stanje podatkov, tehnologije za aplikacijske programske vmesnike – API, sporočilne in dogodkovne sistemi, izvajalna okolja, repozitorij izvirne kode, repozitorij artefaktov, DevOps in avtomatizacija CI/CD in platformske storitve DRO, ki vključujejo centralni avtentikacijski in avtorizacijski (IAM) strežnik, centralni sistem za beleženje dnevninskih zapisov, API prehod in sistem za upravljanje API-jev in centralni konfiguracijski strežnik. Podan je še kratek povzetek smernic za migracijo monolitnih aplikacij v domorodno oblačno arhitekturo.

Viri in reference

10 Key Attributes of Cloud Native Applications, <https://thenewstack.io/cloud-native/10-key-attributes-of-cloud-native-applications/>

A lightweight open-source microservice framework, <https://ee.kumuluz.com/>

Cloud-native application, <https://www.techtarget.com/searchcloudcomputing/definition/cloud-native-application>

Espen Tønnessen Nordli, Sindre Grønstøl Haugeland, Phu H. Nguyen, Hui Song, Franck Chauvel, Migrating monoliths to cloud-native microservices for customizable SaaS, Information and Software Technology, Volume 160, 2023, 107230, ISSN 0950-5849, <https://doi.org/10.1016/j.infsof.2023.107230>

Giovanni Toffetti, Sandro Brunner, Martin Blöchliger, Josef Spillner, Thomas Michael Bohnert, Self-managing cloud-native applications: Design, implementation, and experience, Future Generation Computer Systems, Volume 72, 2017, Pages 165-179, ISSN 0167-739X, <https://doi.org/10.1016/j.future.2016.09.002>

How To Migrate From Monolithic To Microservices(Part-I), <https://medium.com/@puneet.vyas/how-to-migrate-from-monolithic-to-microservices-part-i-bd0645a7c529>

Hüseyin Ünlü, Dhia Eddine Kennouche, Görkem Kılınç Soylu, Onur Demirörs, Microservice-based projects in agile world: A structured interview, Information and Software Technology, Volume 165, 107334, ISSN 0950-5849, <https://doi.org/10.1016/j.infsof.2023.107334>

Java EE microservices: why start-up time and size matter, <https://www.linkedin.com/pulse/java-ee-microservices-why-start-up-time-size-matters-matjaz-b-juric/>

Ken Hamric. Tracing the History of Distributed Tracing and OTel. 2022. <https://tracetest.io/blog/tracing-the-history-ofdistributed-tracing-opentelemetry>

L. Giamattei, A. Guerriero, R. Pietrantuono, S. Russo, I. Malavolta, T. Islam, M. Dînga, A. Koziolk, S. Singh, M. Armbruster, J.M. Gutierrez-Martinez, S. Caro-Alvaro, D. Rodriguez, S. Weber, J. Henss, E. Fernandez Vogelín, F. Simon Panojo, Monitoring tools for DevOps and microservices: A systematic grey literature review, Journal of Systems and Software, Volume 208, 111906, ISSN 0164-1212, <https://doi.org/10.1016/j.jss.2023.111906>

Lei Wen, Hengshun Qian, Wenpan Liu, Research on Intelligent Cloud Native Architecture and Key Technologies Based on DevOps Concept, Procedia Computer Science, Volume 208, 2022, Pages 590-597, ISSN 1877-0509, <https://doi.org/10.1016/j.procs.2022.10.082>

Md. Delowar Hossain, Tangina Sultana, Sharmen Akhter, Md Imtiaz Hossain, Ngo Thien Thu, Luan N.T. Huynh, Ga-Won Lee, Eui-Nam Huh, The role of microservice approach in edge computing: Opportunities, challenges, and research directions, ICT Express, 2023, ISSN 2405-9595, <https://doi.org/10.1016/j.icte.2023.06.006>

Migrate a monolithic application to microservices using domain-driven design, <https://learn.microsoft.com/en-us/azure/architecture/microservices/migrate-monolith>

Nane Kratzke, Peter-Christian Quint, Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study, Journal of Systems and Software, Volume 126, 2017, Pages 1-16, ISSN 0164-1212, <https://doi.org/10.1016/j.jss.2017.01.001>

Open Telemetry authors. Open Telemetry Documentation. 2023. <https://opentelemetry.io/docs/>

Računalniške storitve v oblaku, gradivo predavanj, Fakulteta za računalništvo in informatiko, 2023

Rafik Tighilt, Manel Abdellatif, Imen Trabelsi, Loïc Madern, Naouel Moha, Yann-Gaël Guéhéneuc, On the maintenance support for microservice-based systems through the specification and the detection of microservice antipatterns, Journal of Systems and Software, Volume 204, 2023, 111755, ISSN 0164-1212, <https://doi.org/10.1016/j.jss.2023.111755>

Refactoring a monolith into microservices, <https://cloud.google.com/architecture/microservices-architecture-refactoring-monoliths>

Sören Henning, Wilhelm Hasselbring, Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud, Journal of Systems and Software, Volume 208, 111879, ISSN 0164-1212, <https://doi.org/10.1016/j.jss.2023.111879>

The Twelve Factor App, <https://12factor.net/>

Tomas Cerny, Amr S. Abdelfattah, Abdullah Al Maruf, Andrea Janes, Davide Taibi, Catalog and detection techniques of microservice anti-patterns and bad smells: A tertiary study, Journal of Systems and Software, Volume 206, 2023, 111829, ISSN 0164-1212, <https://doi.org/10.1016/j.jss.2023.111829>

Understanding cloud-native applications, <https://www.redhat.com/en/topics/cloud-native-apps>

Wesley K.G. Assunção, Jacob Krüger, Sébastien Mosser, Sofiane Selaoui, How do microservices evolve? An empirical analysis of changes in open-source microservice repositories, Journal of Systems and Software, Volume 204, 2023, 111788, ISSN 0164-1212, <https://doi.org/10.1016/j.jss.2023.111788>