Mikroservisna arhitektura

Principi softverskog inženjerstva, *Elektrotehnički fakultet Univerziteta u Beogradu*

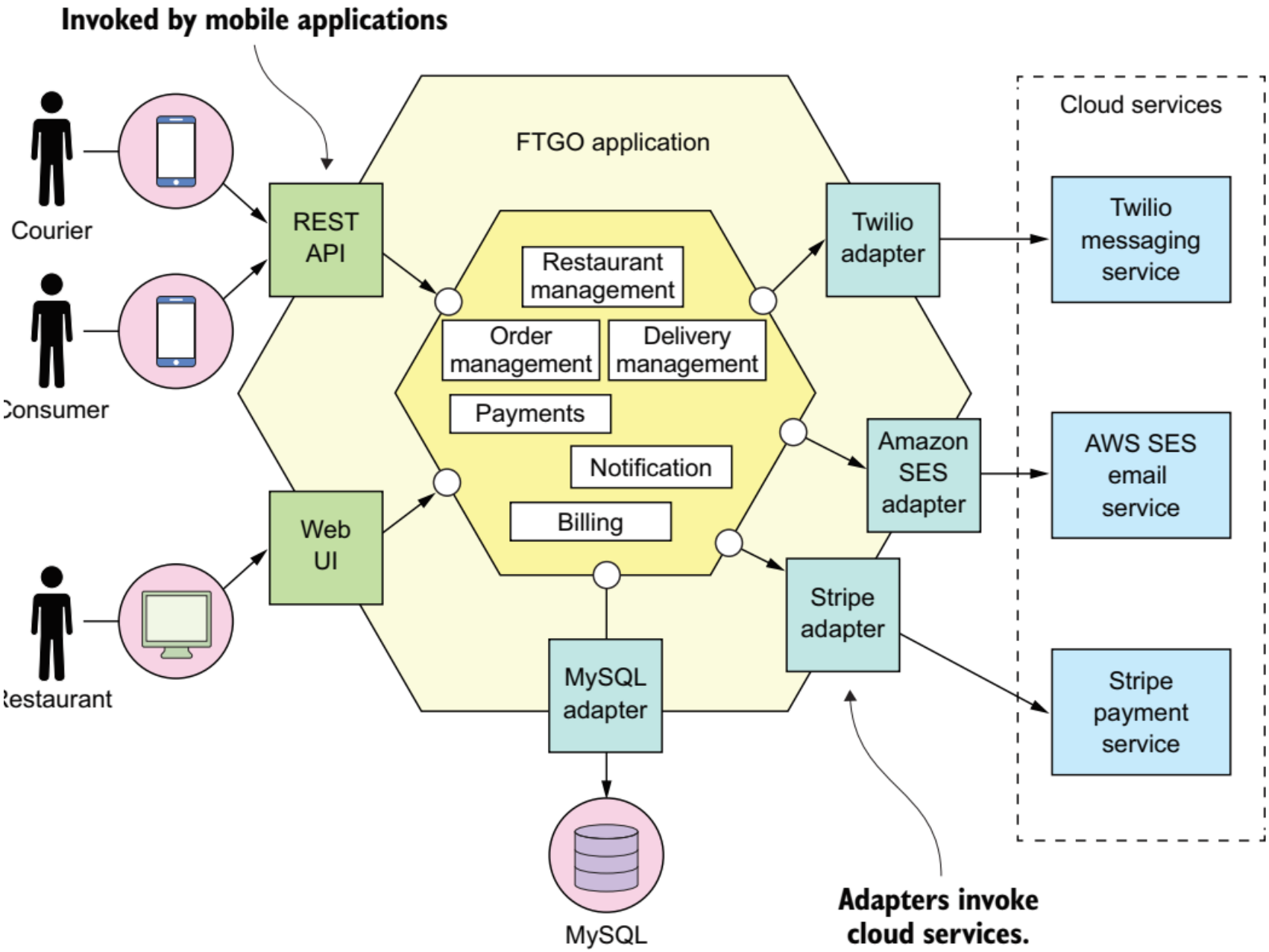
Definicija

- Mikroservisna arhitektura je pristup razvoju jedne aplikacije kao skupa malih usluga, svaka se pokreće u svom procesu i komunicira jednostavnim mehanizmima, često HTTP.
- Svaki servis implementira zaseban poslovni zahtev i može se nezavisno pustiti u rad potpuno automatizovano.
- Postoji minimum centralizovanog upravljanja ovim uslugama, koje mogu biti napisane na različitim programskim jezicima i koristiti različite tehnologije za skladištenje podataka.

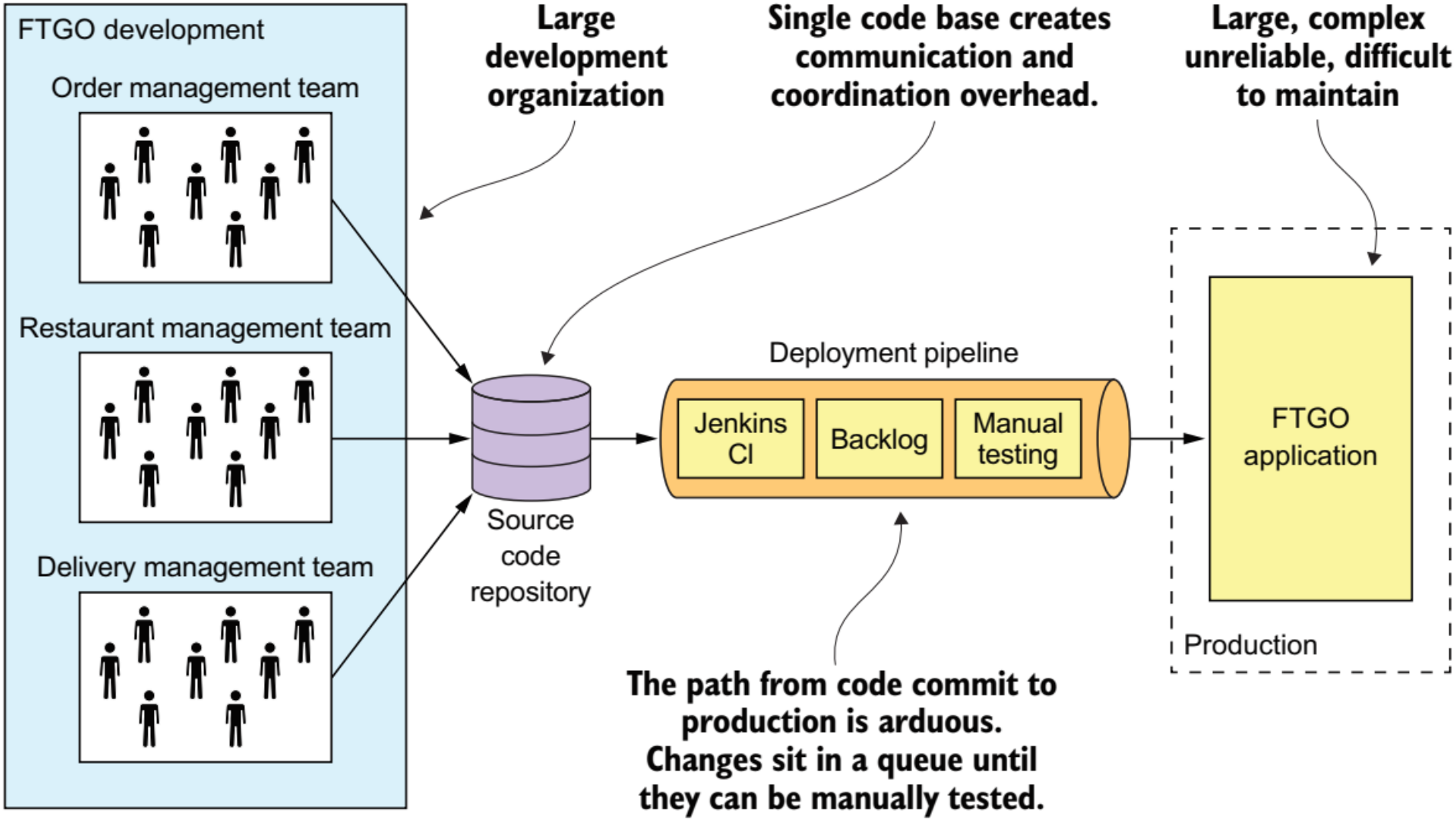
Poređenje sa monolitnom arhitekturom

- U ranijim lekcijama smo obrađivali arhitekturu aplikacije izgrađene u tri glavna dela: korisnički interfejs na strani klijenta (koji se sastoji od HTML stranica i javascripta koji se izvršavaju u pregledaču na korisnikovom računaru) baza podataka (koja se sastoji od mnogih tabela umetnutih u zajedničko i obično relaciono upravljanje bazama podataka) i aplikacije na serveru.
- Aplikacija na strani servera obrađuje HTTP zahteve, izvršava logiku domena, preuzima i ažurira podatke iz baze podataka i bira i popunjava HTML poglede koji se šalju pregledaču.
- Ova aplikacija na strani servera je jedna logička celina. Sve promene sistema uključuju izgradnju i primenu nove verzije aplikacije na serveru.

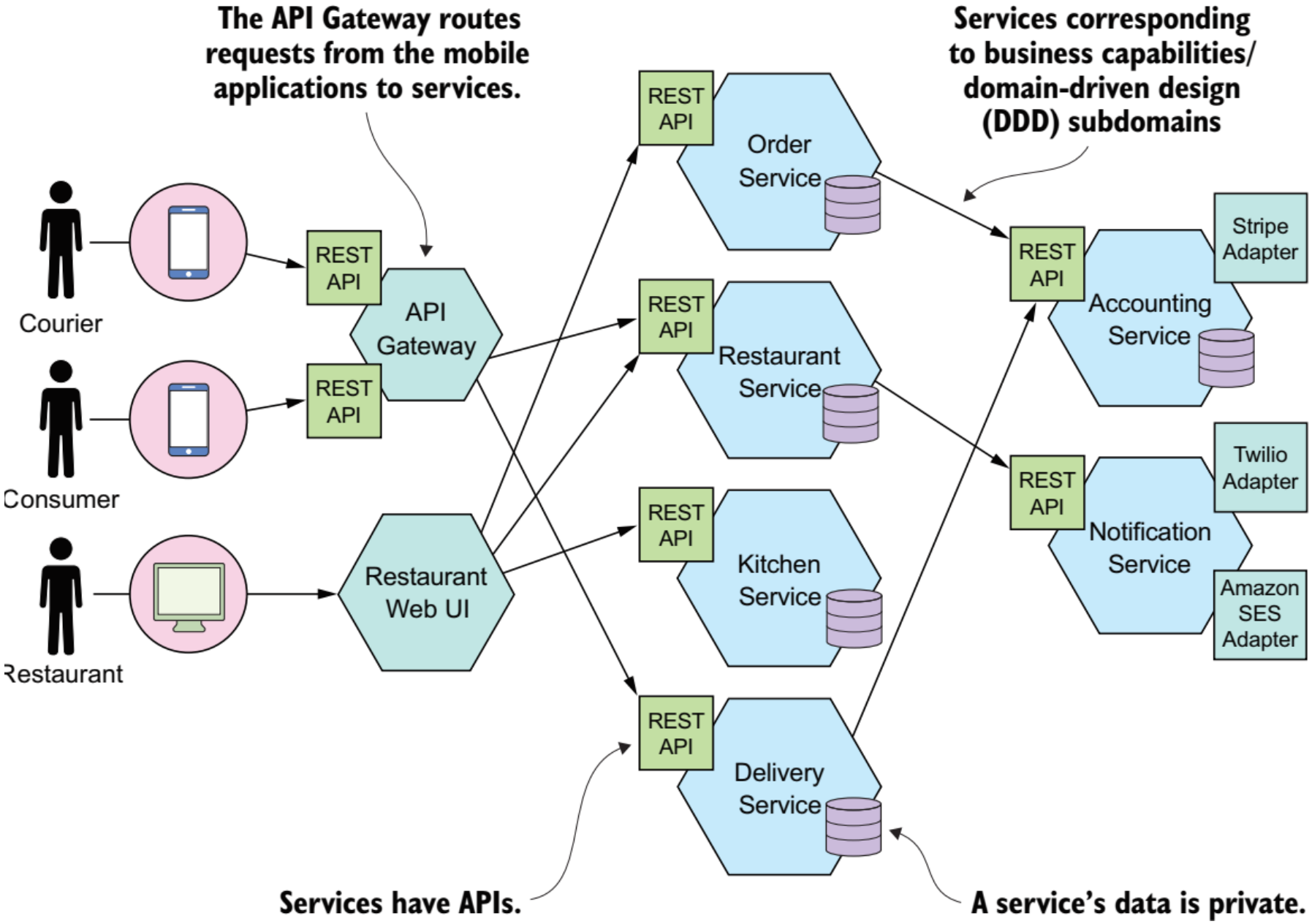
Primer monolitne aplikacije



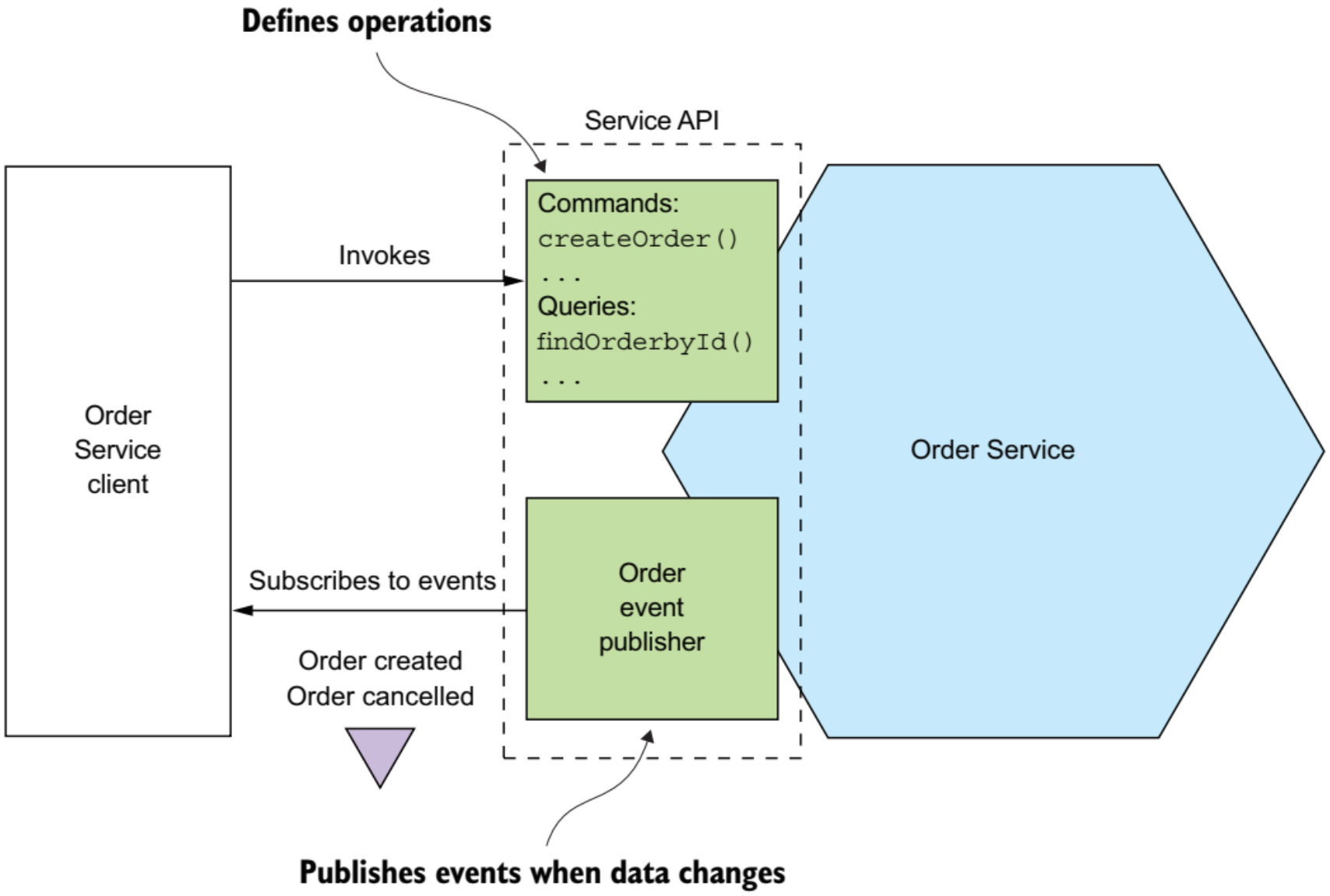
Održavanje monolitne aplikacije



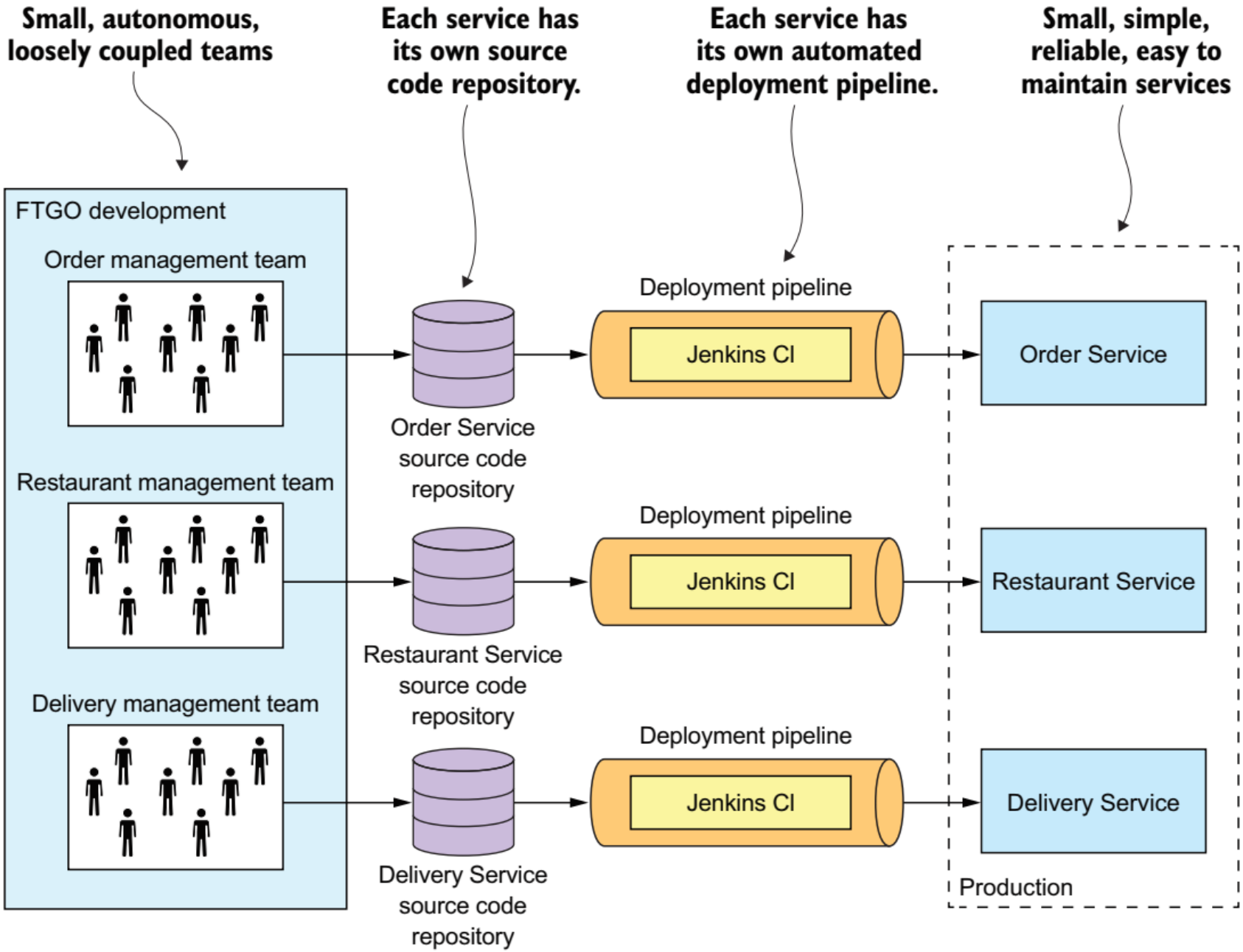
Primer iste aplikacije kao mikroservisa



Šta je to mikroservis



Održavanje mikroservisa



Mikroservisi naspram monolitne arhitekture

Mikroservisi imaju prednosti...

- Jasne granice modula: Mikroservisi ojačavaju modularnu strukturu, što je posebno važno za veće timove.
- Nezavisno puštanje u rad: Jednostavne servise je lakše pustiti u rad, a pošto su autonomni, ređe će prouzrokovati kvarove sistema kada otkažu.
- Tehnološka raznolikost: U razvoju mikroservisa se može kombinovati više jezika, razvojnih okvira i tehnologija skladištenja podataka.

...ali to dolazi uz troškove

- Distribuiranost: Distribuirane sisteme je teže programirati, jer su udaljeni pozivi spori i uvek su u opasnosti od otkaza.
- Konačna (eventualna) konzistencija: Održavanje čvrste konzistencije izuzetno je teško za distribuirani sistem, što znači da svi moraju da savladaju eventualnu konzistenciju.
- Kompleksnost rada: Potreban je zreo operativni tim za upravljanje mnoštvom servisa koji se redovno puštaju u rad i zamenjuju starije verzije.

Poređenje mikroservisa sa (ranijim) servisno orijentisanim arhitekturama

Karakteristika	SOA	Mikroservisi
Inter-servisna komunikacija	Pametne veze, na primer Enterprise Service Bus, koji koriste protokole teške kategorije, kao što je SOAP i drugi WS* standardi.	“Dumb pipes”, kao što je posrednik za poruke, ili direktna komunikacija servis-servis, koristeći lagane protokole kao što su REST ili gRPC
Podaci	Globalni model podataka i deljena baza podataka	Lokalni model i baza podataka za svaki servis
Veličina servisa	Veća monolitna aplikacija	Manji servis

KOMUNIKACIJA MEĐU SERVISIMA

Komunikacija među servisima

- Servisi moraju sarađivati da bi ispunili neki zahtev klijenta.
- Pitanja komunikacije uključuju stilove interakcije kao što su sinhroni/asinhroni), direktno ili putem posrednika, verzioniranje APIja, standarde komunikacije (REST, gRPC, GraphQL,...), pronalaženje servisa, transakcije, oporavak pri otkazu itd.
- U nastavku ćemo ukratko obraditi ova pitanja i dati neke primere konkretnih projektnih šablona, protokola i biblioteka za realizaciju.

Stilovi komunikacije među servisima

	Jedan-na-jedan	Jedan-na-više
Sinhrona	Zahtev/odgovor	—
Asinhrona	Asinh. zahtev/odgovor Jednosmerna notifikacija	Objava/pretplata Objava/asinh. odgovori

Tipovi jedan na jedan komunikacije

- **Zahtev/odgovor (request/response)**- Klijent servisa podnosi zahtev servisu i čeka na odgovor. Klijent očekuje da odgovor stigne pravovremeno. Može i se blokira tokom čekanja. Ovo rezultuje u čvrsto povezanim servisima.
- **Asinhroni zahtev/odgovor** - Klijent šalje servisu zahtev, koji odgovara asinhrono. Klijent se ne blokira dok čeka, jer servis možda dogu neće poslati odgovor.
- **Jednosmerno obaveštenje** - Klijent servisa šalje zahtev, ali ne očekuje odgovor.

Tipovi komunikacije jedan na više

- **Objava/pretplata (publish/subscribe)** - Klijent šalje poruku obaveštenja koju prima nula ili više zainteresovanih servisa.
- **Objava/asinhroni odgovori** - Klijent objavi poruku sa zahtevom, a zatim čeka određeno vreme na odgovore zainteresovanih servisa.

Sinhroni zahtev-odgovor

- Ova vrsta komunikacije često se realizuje putem REST ili gRPC protokola.

  Representational State Transfer (REST)
- koristi HTTP glagole za manipulisanje resursima na koje se upućuje pomoću URLa.
- GET zahtev vraća predstavu resursa u obliku JSON objekta, mada ima i drugih formata.
- Zahtev POST kreira novi resurs a PUT zahteva ažuriranje resursa, DELETE brisanje resursa.
- Na primer, da bi se ažurirao status fakture mikroservisu Invoice moguće je uputiti sledeći zahtev:

  PUT localhost:8081/.../api/v1/invoices/{invoice_id}

  sa telom zahteva: ["status": "waiting”]

Representational State Transfer (REST)

- Klijentski zahtev:

```
import requests
def updateInvoice(order_id, order_data):
    # define the url for the REST API endpoint
    url = "https://example.com/api/orders/" + order_id
    # define the headers for the request
    headers = {"Content-Type": "application/json"}
    # make a PUT request with the order data as the json payload
    response = requests.put(url, data=order_data, headers=headers)
    # check the status code of the response
    if response.status_code == 200:
        # the request was successful, return the response json
        return response.json()
    else:
        # the request failed, raise an exception with the response text
        raise Exception(response.text)
```

Representational State Transfer (REST)

Odgovor na zahtev u flask-u.

```
from flask import Flask, request, jsonify
# create a flask app
app = Flask(__name__)

# define a route for updating an order
@app.route("/api/orders/<order_id>", methods=["PUT"])
def update_order(order_id):
    # get the order data from the request json
    order_data = request.get_json()
    # validate the order data
    if order_data and "items" in order_data and "total" in order_data:
        # update the order in the database (this is just a mock example)
        db.update(order_id, order_data)
        # return a success message with the updated order data
        return jsonify({"message": "Order updated successfully", "order": order_data})
    else:
        # return an error message with a bad request status code
        return jsonify({"message": "Invalid order data"}), 400

# run the app
if __name__ == "__main__":
    app.run()
```

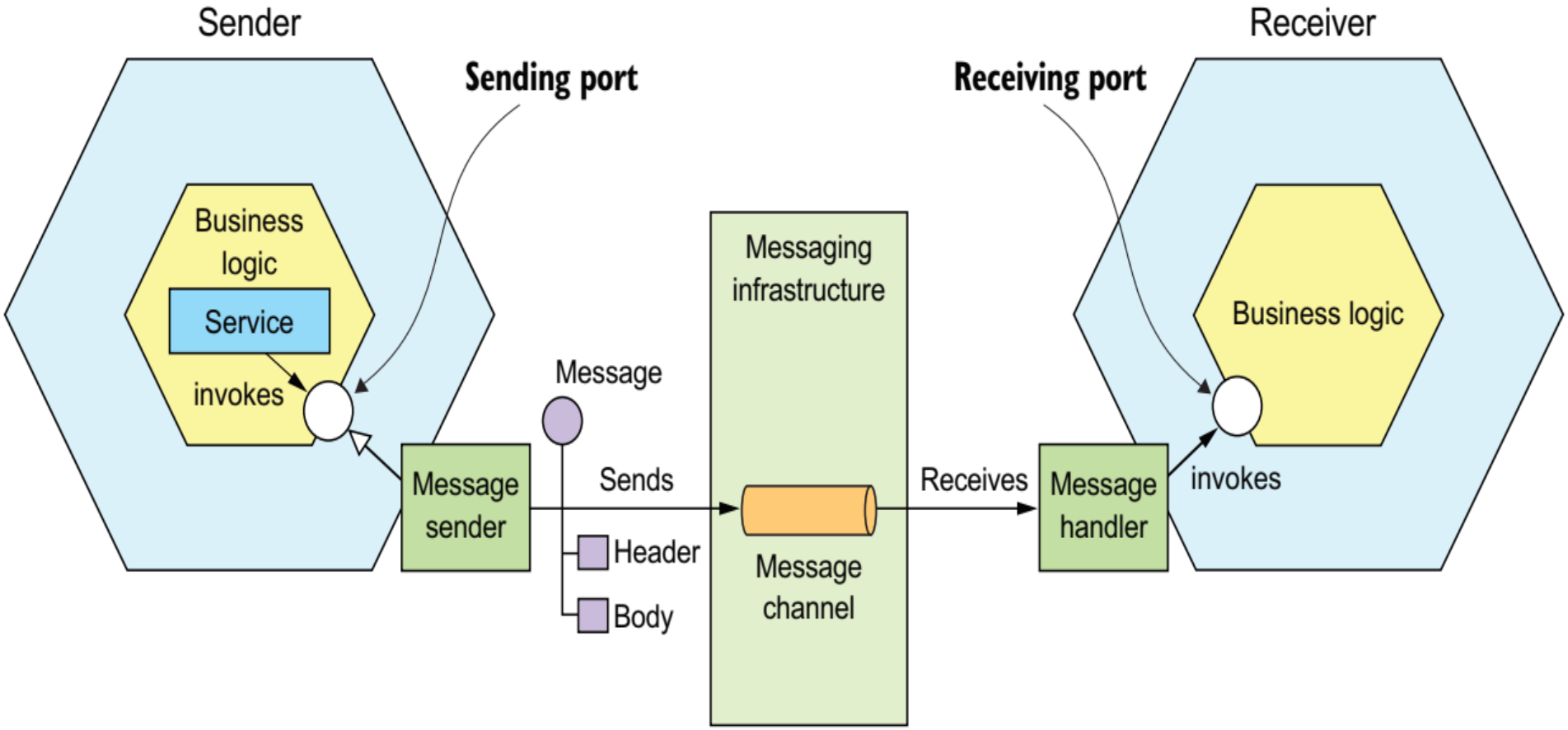
gRPC – alternativa RESTu

- U gRPC-u (remote procedure call protokol razvijen u Googleu), klijentska aplikacija može direktno pozvati metodu na serverskoj aplikaciji na drugoj mašini kao da je lokalni objekat, što olakšava stvaranje distribuiranih aplikacija i usluga.
- gRPC razmena podataka zasnovana je na Protokol Baferima, binarnom formatu za razmenu podataka koji je jezički neutralan.
- Pošto komprimuje podatke efikasniji je nego REST, ali zahteva HTTP/2 što ga čini pogodnim npr. za sinhronu komunikaciju između mikroservisa unutar oblaka, ali ne toliko između npr. mobilnog klijenta i servisa jer neki firewall-ovi ne podržavaju HTTP/2.
- Takođe je kompleksniji za implementaciju.

Asinhrona komunikacija slanjem poruka

- Servisi mogu da komuniciraju asinhronom razmenom poruka.
- Obično se za razmenu poruka koristi posrednik poruka (message broker), koji deluje kao posrednik između servisa, mada je druga opcija korišćenje brokerless arhitekture, gde servisi međusobno komuniciraju direktno.
- Klijent uputi zahtev servisu slanjem poruke. Ako se očekuje da instanca servisa odgovori na poruku, učiniće to tako što će klijentu poslati zasebnu poruku.
- Zbog toga što je komunikacija asinhrona, klijent se ne blokira čekanjem odgovora, nego može da radi nešto drugo.

Asinhona komunikacija slanjem poruka



Asinhrona komunikacija slanjem poruka

- Posrednik poruka je posrednik kroz koji prolaze sve poruke. Pošiljalac piše poruku posredniku poruka, a posrednik poruke je dostavlja primaocu.
- Važna prednost upotrebe posrednika poruka je ta što pošiljalac ne mora znati mrežnu lokaciju primaoca. Još jedna prednost je što posrednik poruka baferuje poruke dok primalac ne bude u stanju da ih obradi.
- Primeri popularnih posrednika: RabbitMQ, ActiveMQ, Apache Kafka.
- Svaki broker pravi različite kompromise. Na primer, broker sa malom latencijom može da ne čuvaj redosled, ne daje garancije za isporuku poruka i samo ih čuva u memoriji.
- Posrednik za razmenu poruka koji garantuje isporuku i pouzdano skladišti poruke na disku će verovatno imati veće kašnjenje.
- Izbor posrednika poruka zavisi od zahteva aplikacije. Moguće je čak i da različiti delovi aplikacije imaju različite zahteve za razmenu poruka.

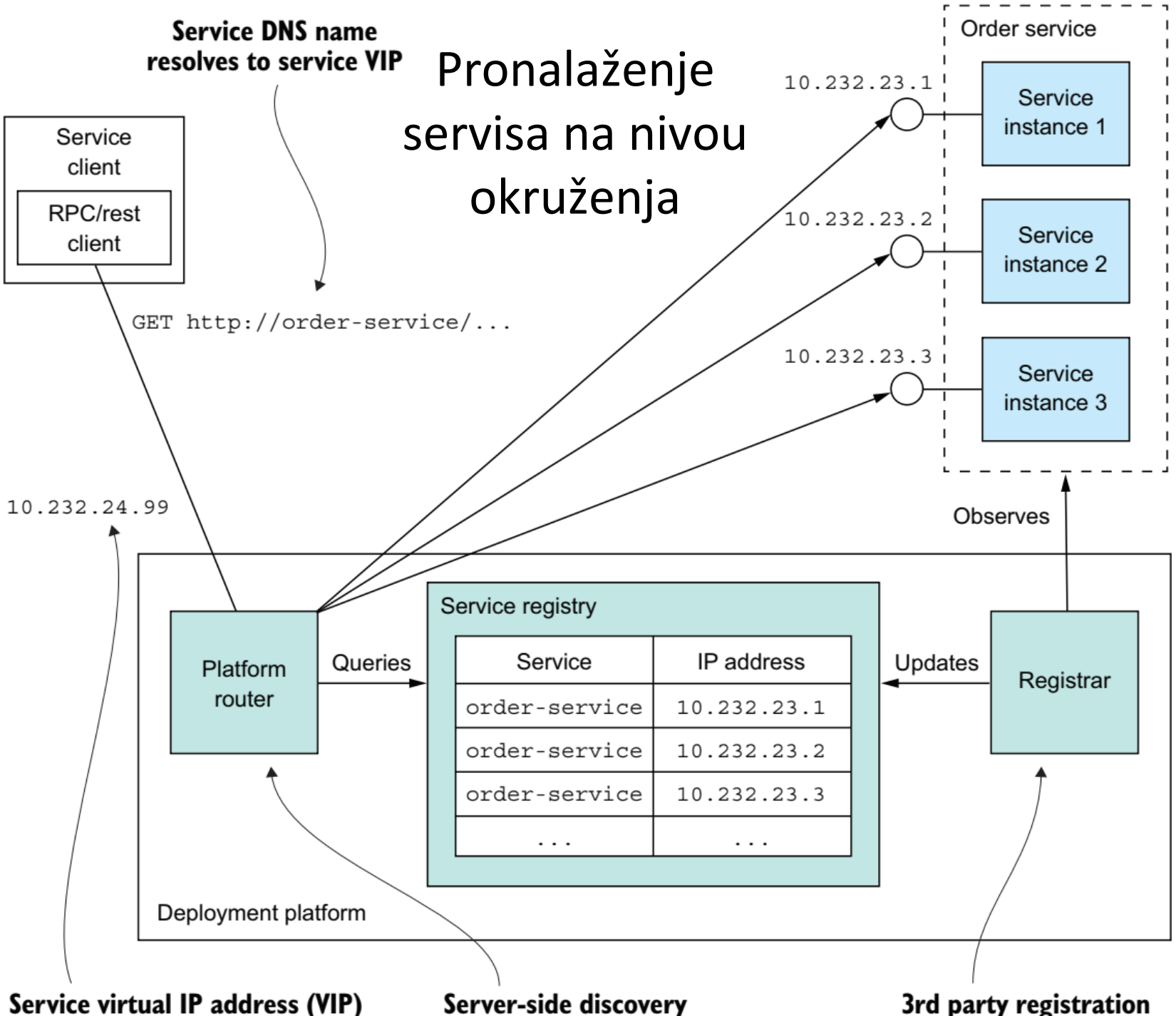
Pronalaženje servisa

- Da bi podneo zahtev, klijentski kod mora da zna mrežnu lokaciju (IP adresa i port) instance servisa.
- U tradicionalnoj aplikaciji koja radi na fizičkom hardveru, mrežne lokacije instanci servisa su obično statične. Na primer, klijentski kod može pročitati mrežne lokacije iz konfiguracione datoteke koja se povremeno ažurira.
- Ali u modernom okruženju zasnovanom na oblaku to obično nije tako jednostavno.
- Servisne instance imaju dinamički dodeljene mrežne lokacije. Štaviše, skup instanci servisa se dinamički menja zbog automatskog skaliranja, kvarova i nadogradnji.
- Zbog toga klijentski kod mora da koristi pronalaženje servisa.

Pronalaženje servisa

- Postoje dva načina obezbeđivanja pronalaženja servisa:
  - Na nivou aplikacije gde sama aplikacija obezbeđuje pronalaženje svojih servisa. Ovo se realizuje pristupom servisnom registru koji pamti skup aktivnih instanci servisa i njihovih adresa. Servisne instance moraju da se autoregistruju u registar, takođe registar mora periodično da proverava zdravlje svake instance.
  - Na nivou produkcionog okruženja. Okruženje ima servisni registar koji prati adrese instanci servisa.

Pronalaženje servisa na nivou okruženja



RAD SA PODACIMA U MIKROSERVISIMA

RAD SA PODACIMA U MIKROSERVISIMA

- Većina servisa ima potrebu da sačuva podatke u nekoj vrsti baze podataka. Postoje dva različita pristupa:
- **Deljena baza podataka** - Jedno rešenje je upotreba jedinstvene deljenje baze podataka za više servisa. Svaki servis slobodno pristupa svojim i podacima drugih servisa koristeći lokalne ACID transakcije.
- **Posebna baza za svaki servis** - Alternativno rešenje je da svaki servis poseduje sopstvenu bazu podataka. Podaci su privatni za servis. Transakcije koje radi servis odnose se samo na njegovu lokalnu bazu. Podaci koji nisu lokalni, dostupni su isključivo putem APIja servisa.

OSOBINE TRANSAKCIJA U BAZAMA

- *ACID* je skraćenica od Atomicity, Consistency, Isolation, Durability, to jest, Atomičnost, Konzistentnost, Izolovanost, Trajnost.
- Atomičnost znači da se transakcija tretira kao nedeljiva celina, ili uspeva u celosti, ili nema nikakvog efekta.
- Konzistentnost znači da transakcija pri promeni sadržaja baza očuvava sva ograničenja i referencijalni integritet definisanih relacija među tabelama.
- Izolovanost se odnosi na konkurentno izvršavanje transakcija i garantuje da će krajnji efekat takvog izvršavanja biti isti kao da su se transakcije izvršavale sekvencijalno jedna po jedna.
- Trajnost označava da se rezultati jednom uspešno izvršene transkacije (commited) trajno čuvaju čak i u slučaju kvarova sistema, nestanka struje itd.

OSOBINE TRANSAKCIJA U BAZAMA

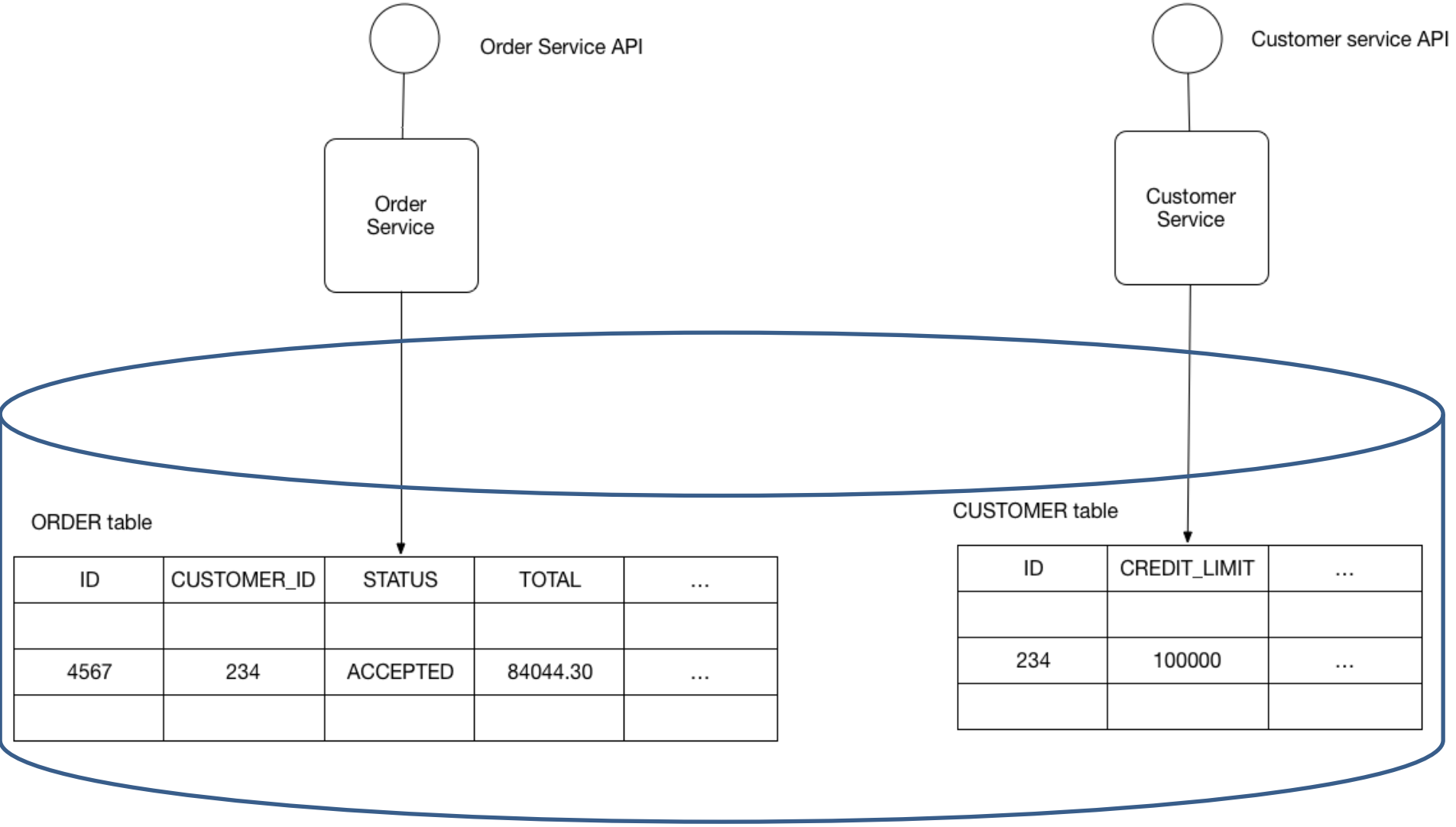
- Međutim u mikroservisnim i generalno distribuiranim arhitekturama (npr. noSQL baze) najčešće nije moguće obezbediti ACID semantiku transakcija.

- Ovo je posledica takozvane CAP teoreme, koja tvrdi da je nemoguće istovremeno postići i konzistenciju i raspoloživost u distriburianom sistemu u kome su podaci distribuirani na čvorove, a poruke između čvorova mogu da se izgube ili zakasne.
- Ako se desi otkaz mreže sistem može:
  - Vratiti grešku ili time out u situaciji kada ne može garantovati ažurne podatke, u situaciji kada se preferira konzistentnost
  - Uvek vratiti neki podatak (čak i ako nije garantovano najažurniji) ako se preferira raspoloživost.
- U noSQL bazama preferira se visoka raspoloživost nauštrb konzistencije.

OSOBINE TRANSAKCIJA U BAZAMA

- noSQL baze poseduju takozvani *BASE transakcioni model*, to jest Basically Available, Soft state, Eventual Consistency. To znači sledeće:
- U osnovi raspoloživ – čitanja i upisi maksimalno uspevaju ali bez garancija konzistentnosti,
- "Meko" stanje – vrednosti podataka mogu se menjati tokom vremena dok se ne razreše svi konflikti.
- Konačno konzistentan – posle nekog vremena u odsustvu novih upisa, sistem će postati konzistentan, to jest čitanja će vračati najažurnije podatke.

DELJENA BAZA PODATAKA



ID	CUSTOMER_ID	STATUS	TOTAL	...
4567	234	ACCEPTED	84044.30	...

ID	CREDIT_LIMIT	...
234	100000	...

PRIMER KORIŠĆENJA

- OrderService i CustomerService slobodno pristupaju međusobnim tabelama. Na primer, OrderService može da koristi sledeću ACID transakciju kako bi osigurao da nova porudžbina neće prekršiti kreditno ograničenje kupca.

```
BEGIN TRANSACTION
SELECT ORDER_TOTAL
 FROM ORDERS WHERE CUSTOMER_ID = ?
SELECT CREDIT_LIMIT
FROM CUSTOMERS WHERE CUSTOMER_ID = ?
INSERT INTO ORDERS …
COMMIT TRANSACTION
```

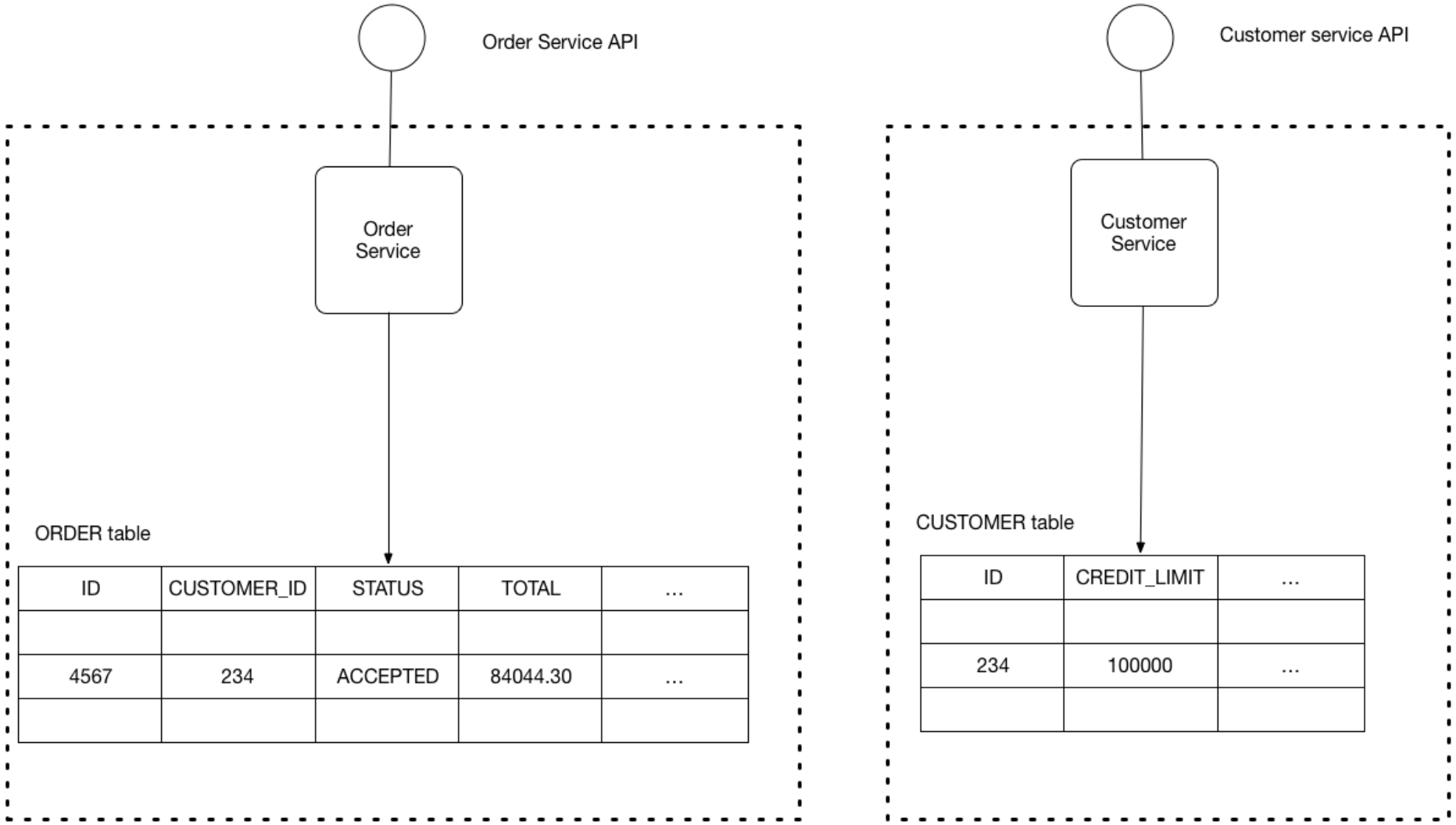
- Baza podataka garantuje da kreditni limit neće biti prekoračen čak i kada istovremene transakcije pokušavaju da kreiraju naloge za istog kupca.

NEDOSTACI DELJENE BAZE PODATAKA

Mane ovog pristupa su:

- Konflikti tokom razvoja - programer koji radi, na primer, na usluzi OrderService, moraće da koordinira promene šeme sa programerima drugih usluga koje pristupaju istim tabelama. Ova sprega i dodatna koordinacija usporiće razvoj.
- Neželjena sprega tokom rada sistema - svi servisi pristupaju istoj bazi podataka i mogu potencijalno da ometaju jedan drugi. Na primer, ako dugotrajna transakcija CustomerService drži zaključavanje na tabeli ORDER, tada će OrderService biti blokiran.
- Pojedinačna baza podataka možda neće kapacitetom i peformansama zadovoljiti zahteve za skladištenjem podataka i pristupom svih servisa.
- Zbog toga se često ovaj pristup naziva antipaternom u mikroservisnoj arhitekturi.

POSEBNA BAZA PODATAKA PO SERVISU



PREDNOSTI ODVOJENIH BAZA PODATAKA

Korišćenje posebne baze podataka po servisu ima sledeće prednosti:

- Pomaže da se osigura da su servisi labavo povezani. Promene u bazi podataka jednog servisa ne utiču na bilo koji drugi servis.

- Svaki servis može koristiti tip baze podataka koji najviše odgovara njegovim potrebama. Na primer, servis koji vrši pretragu teksta može da koristi ElasticSearch. Servis koji manipuliše socijalnom mrežom mogao bi da koristi Neo4j.

NEDOSTACI ODVOJENIH BAZA PODATAKA

Korišćenje separatnih baza podataka po servisu ima sledeće nedostatke:

- Implementacija poslovnih transakcija koje obuhvataju više servisa nije jednostavna. Štaviše, mnoge moderne (NoSql) baze podataka ne podržavaju klasične transakcije.

- Primena upita koji spajaju podatke koji su sada u više baza podataka je izazov.

- Složenost upravljanja velikim brojem SQL i NoSQL baza podataka

PROJEKTOVANJE ODVOJENIH BAZA PODATAKA

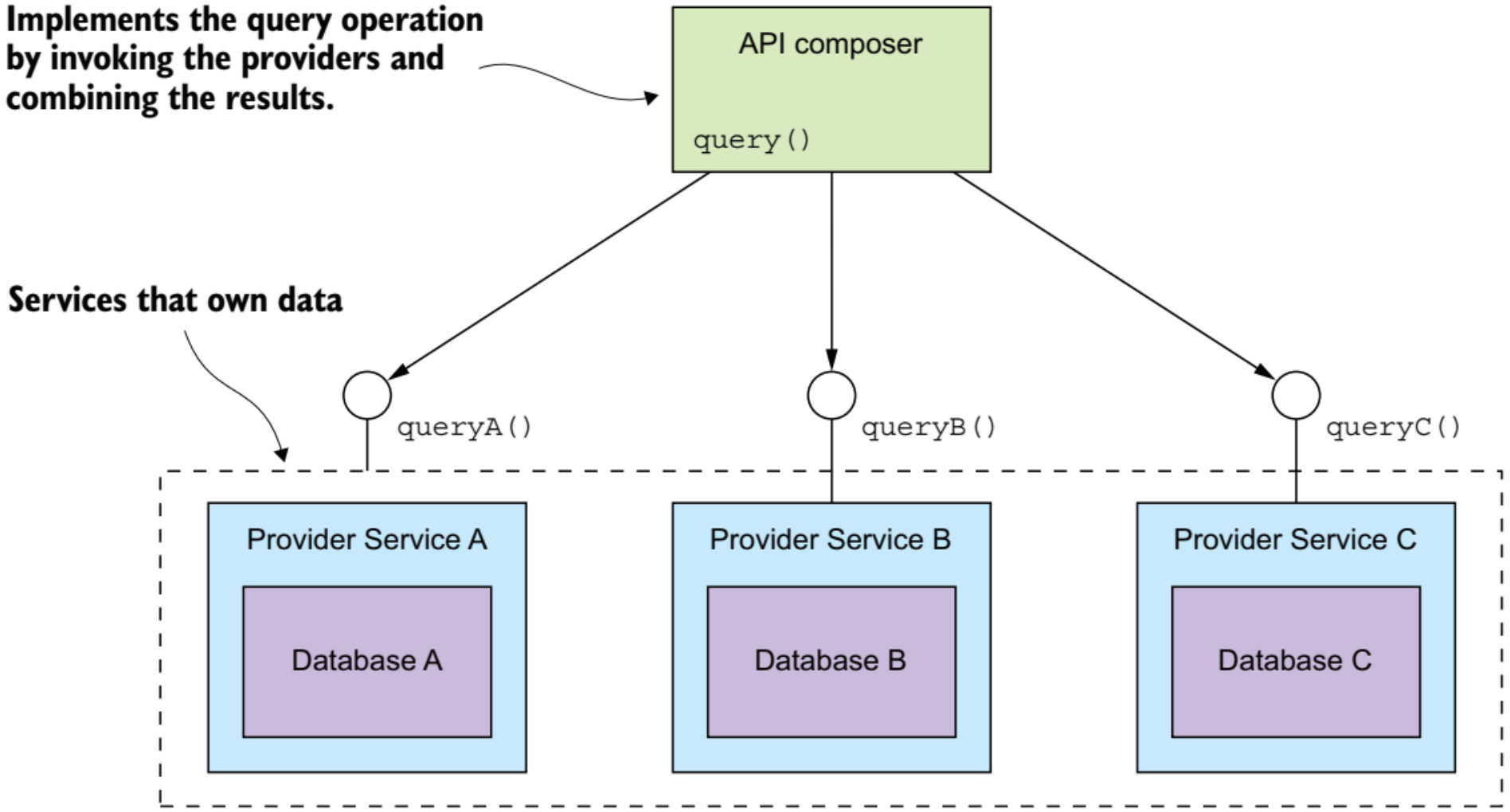
Postoje različiti projektni obrasci za realizaciju transakcija i upita koji obuhvataju više servisa servisa:

- Implementacija transakcija koje obuhvataju više servisa – koristi se *Saga* obrazac.

Implementacija upita koji se prostiru na više servisa:

- *API Kompozicija* – kombinovanje podataka se vrši u programskom kodu servisa (ili api prolazu), a ne u bazi podataka.
- *Razdvajanje odgovornosti za naredbe i upite (CQRS)* - održava jedan ili više materijalizovanih prikaza koji sadrže podatke iz više servisa. Prikaze čuvaju servisi koji se pretplaćuju na događaje koje svaki servis objavljuje kada ažurira svoje podatke.

Projektni obrazac API Composition

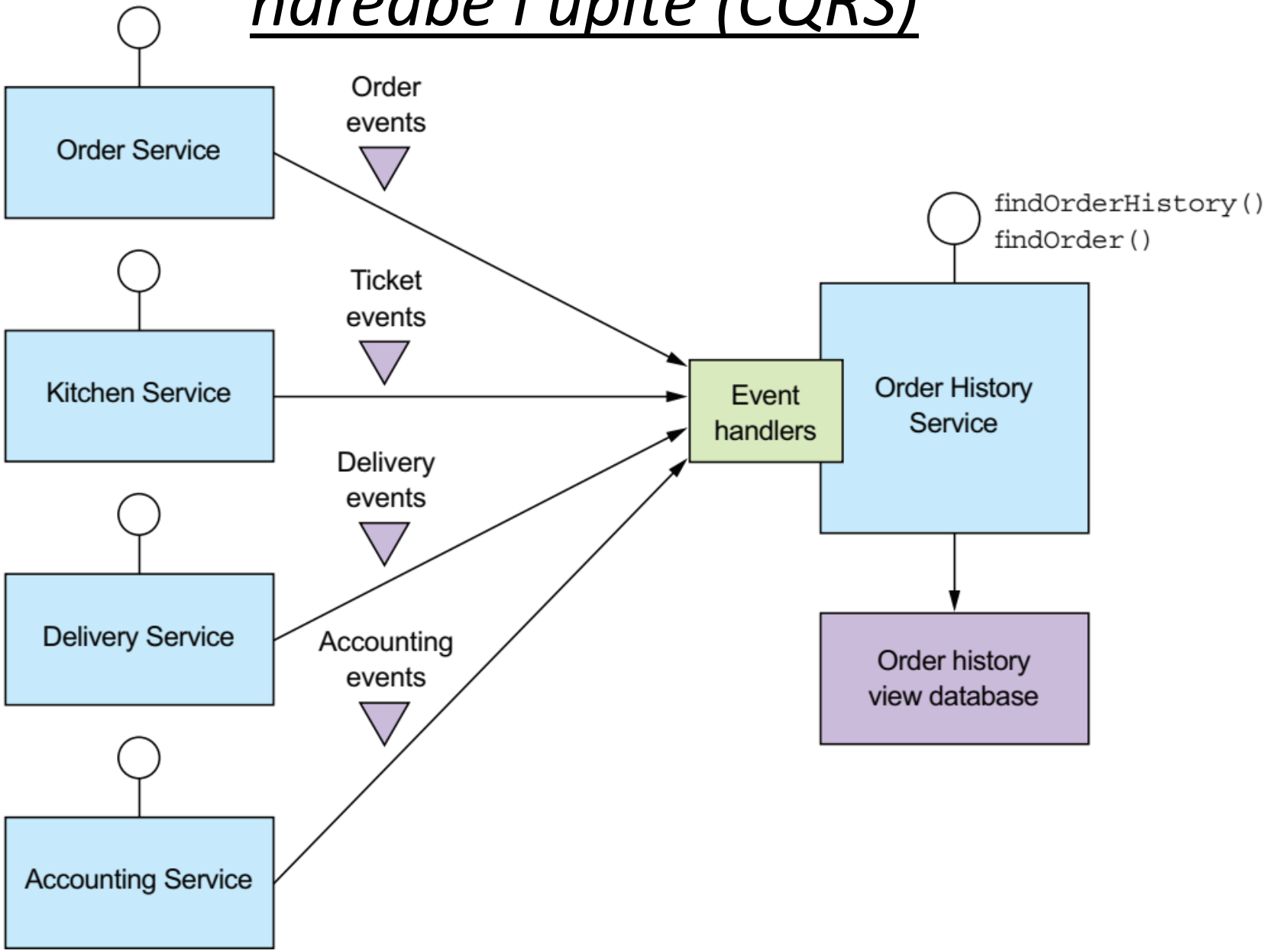


Projektni obrazac API Composition

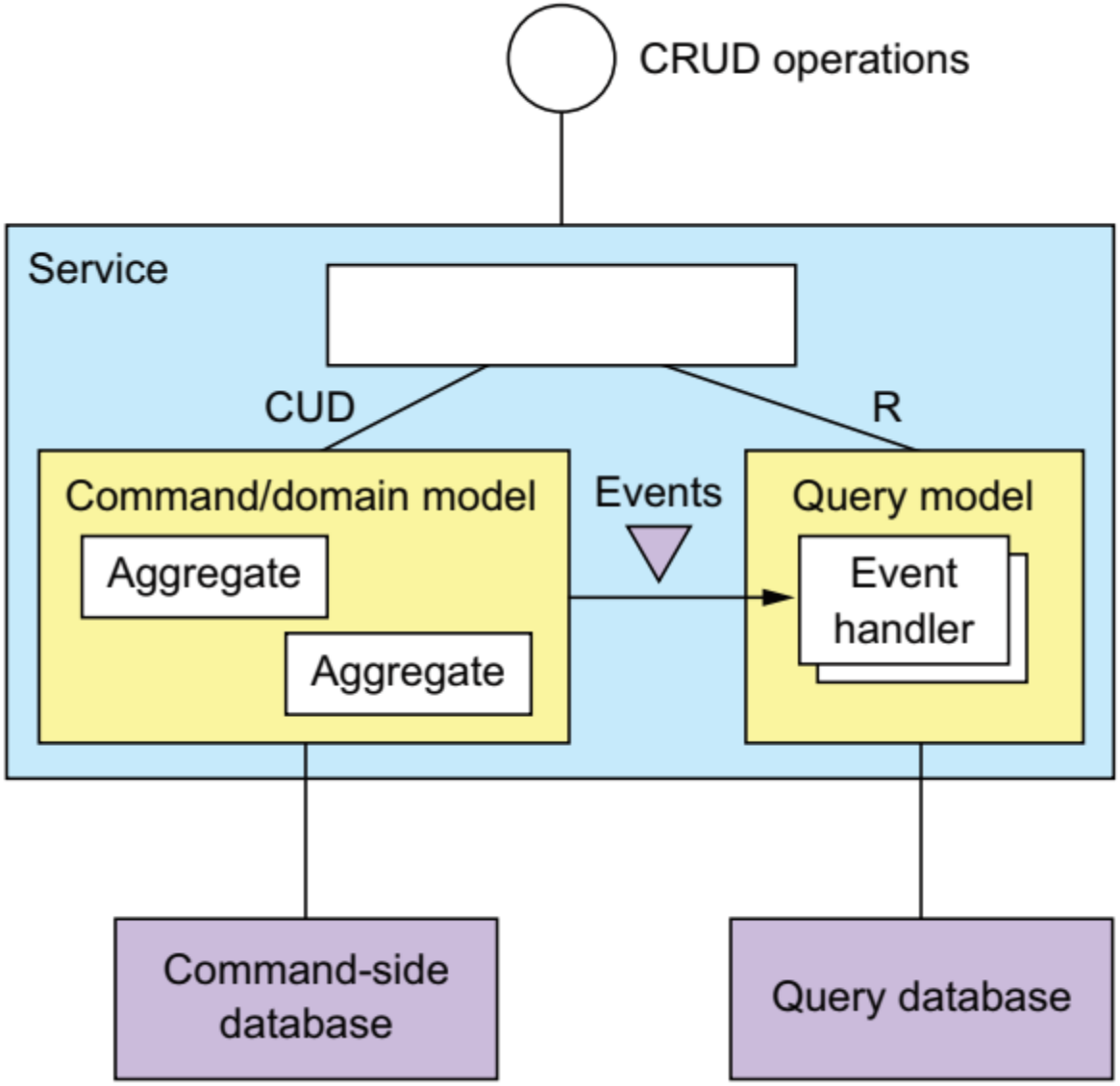
Stvari o kojima treba voditi računa prilikom realizacije ovog uzorka:

- Ko igra ulogu sastavljača? U slučaju brze LAN konekcije to može biti klijent aplikacije. Alternativno to može biti API prolaz ili poseban mikroservis aplikacije.
- Poboljšanje performansi postiže se ako sastavljač šalje upite dobavljačima u paraleli, ne blokirajući se.
- Postoji rizik smanjene raspoloživosti servisa (pošto je u igri više servisa). Ako neki od servisa dobaljača otkaže, sastavljač može biti tako napravljen da vrati ili nekompletne podatke ostalih dobavljača, ili ranije keširane podatke od trenutno nedostupnog dobavljača (ovo može čak i ako je dobavljač dostupan radi povećanja performansi).
- Nema garancija za konzistenciju podataka s obzirom da se izvršava veći broj upita nad većim brojem baza. Ako je konzistencija neophodna, koristiti CQRS uzorak umesto ovoga.

Projektni obrazac *Razdvajanje odgovornosti za naredbe i upite (CQRS)*



Projektni obrazac *Razdvajanje odgovornosti za naredbe i upite (CQRS)*

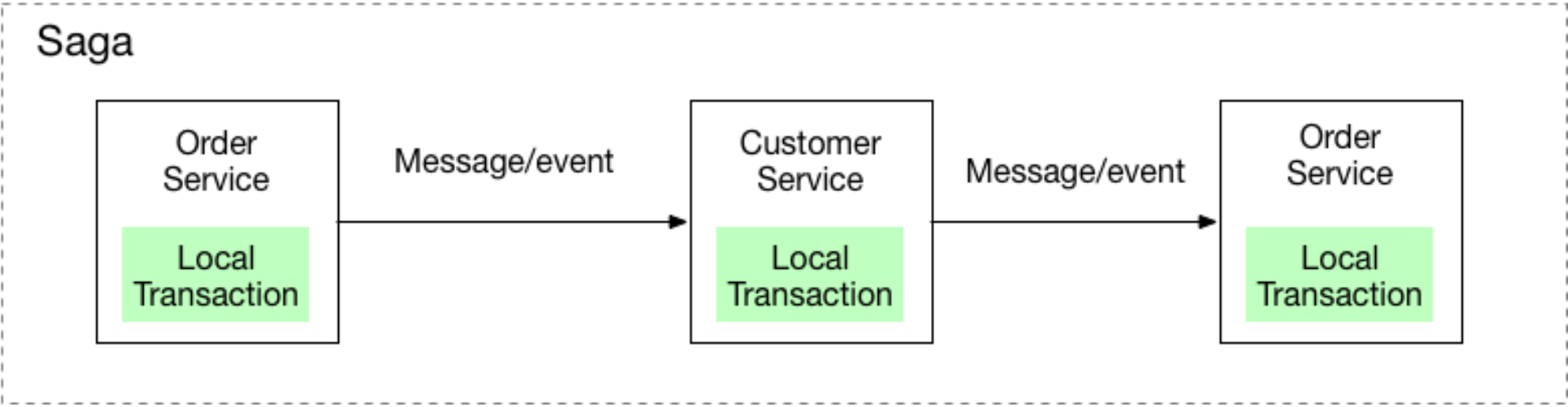


Projektni obrazac *Razdvajanje odgovornosti za naredbe i upite (CQRS)*

- Dobre strane CQRS šablona su mogućnost postizanja dobrih performansi upita i skaliranja sistema.
- Takođe je pogodan u sistemima zasnovanim na događajima (ako je to osnovna projektna karakteristika sistema).
- Loša strana je povećana kompleksnost sistema.
- Takođe u situaciji kada postoji kašnjenje pri ažuriranju replika podataka, sistem može da ponudi samo konačnu konzistenciju.

Projektni obrazac *Saga*

- Rešava problem realizacije transakcija koje se prostiru na više servisa sa lokalnim bazama podataka kada se zahteva konzistencija podataka.



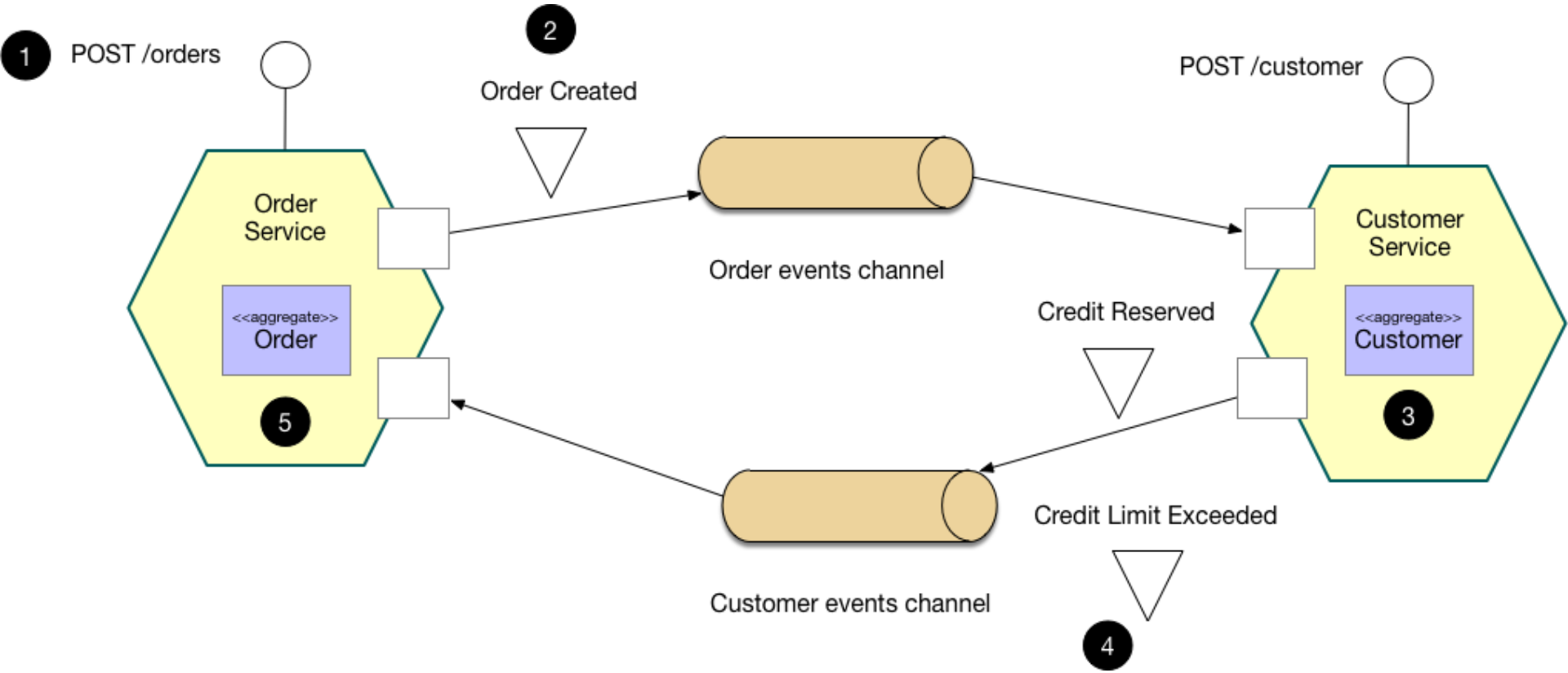
Projektni obrazac *Saga*

- Postoje dva načina koordinacije saga:

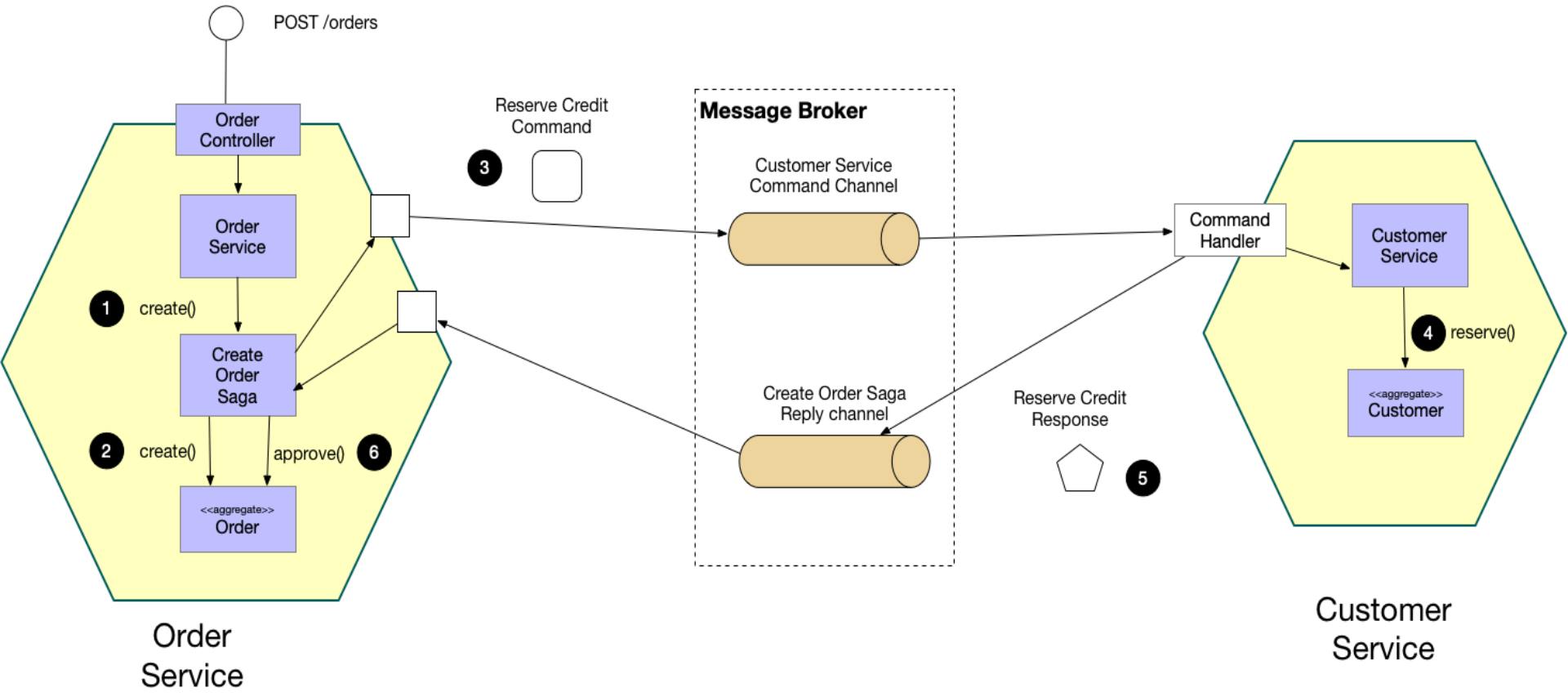
- **Koreografija** - svaka lokalna transakcija objavljuje događaje koji pokreću lokalne transakcije u drugim uslugama.

- **Orkestracija** - orkestrator (jedan servis) govori drugim servisima, učesnicima sage, koje lokalne transakcije treba izvršiti.

Primer sage zasnovane na koreografiji



Primer sage zasnovane na orkestraciji

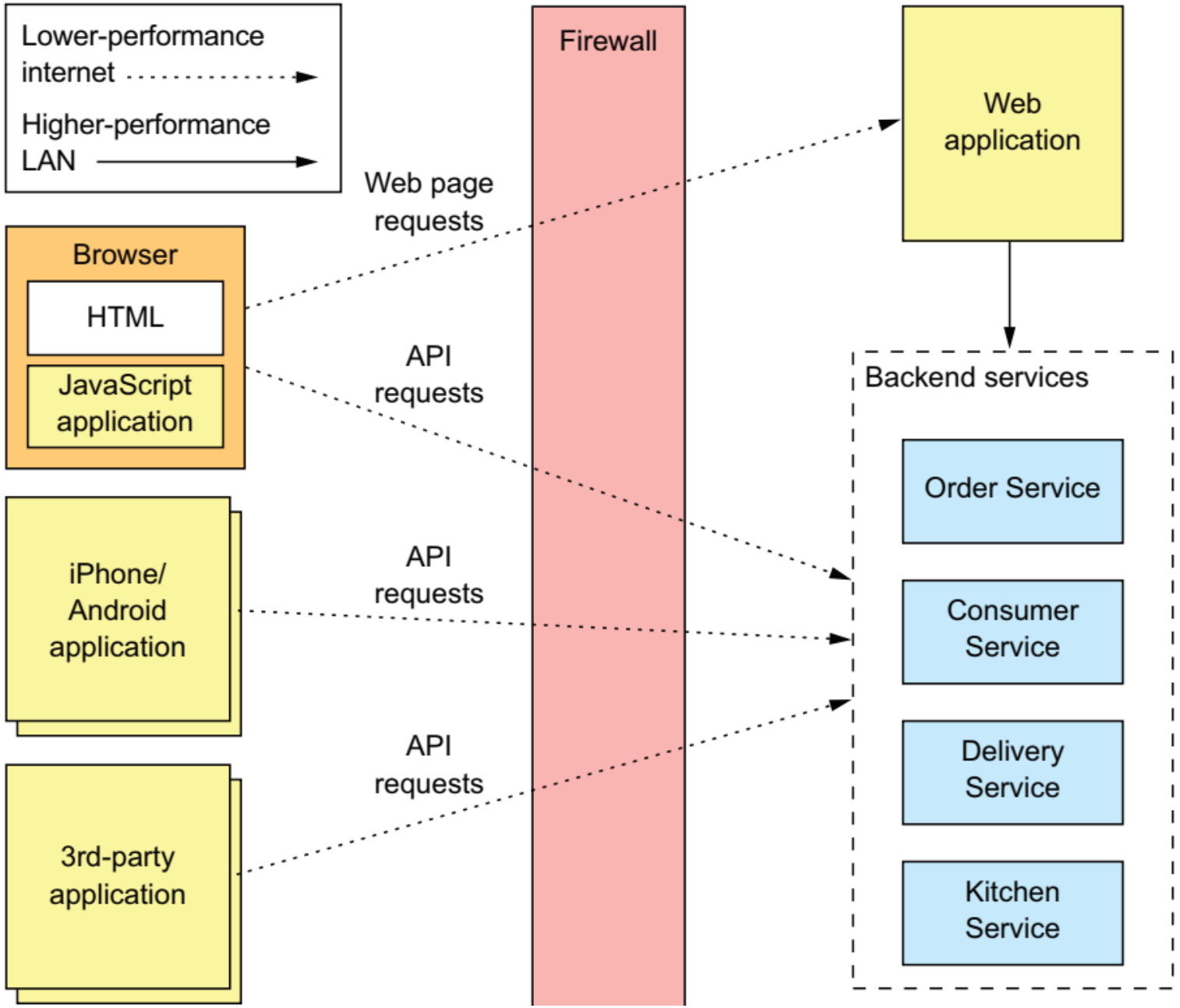


Projektni obrazac *Saga*

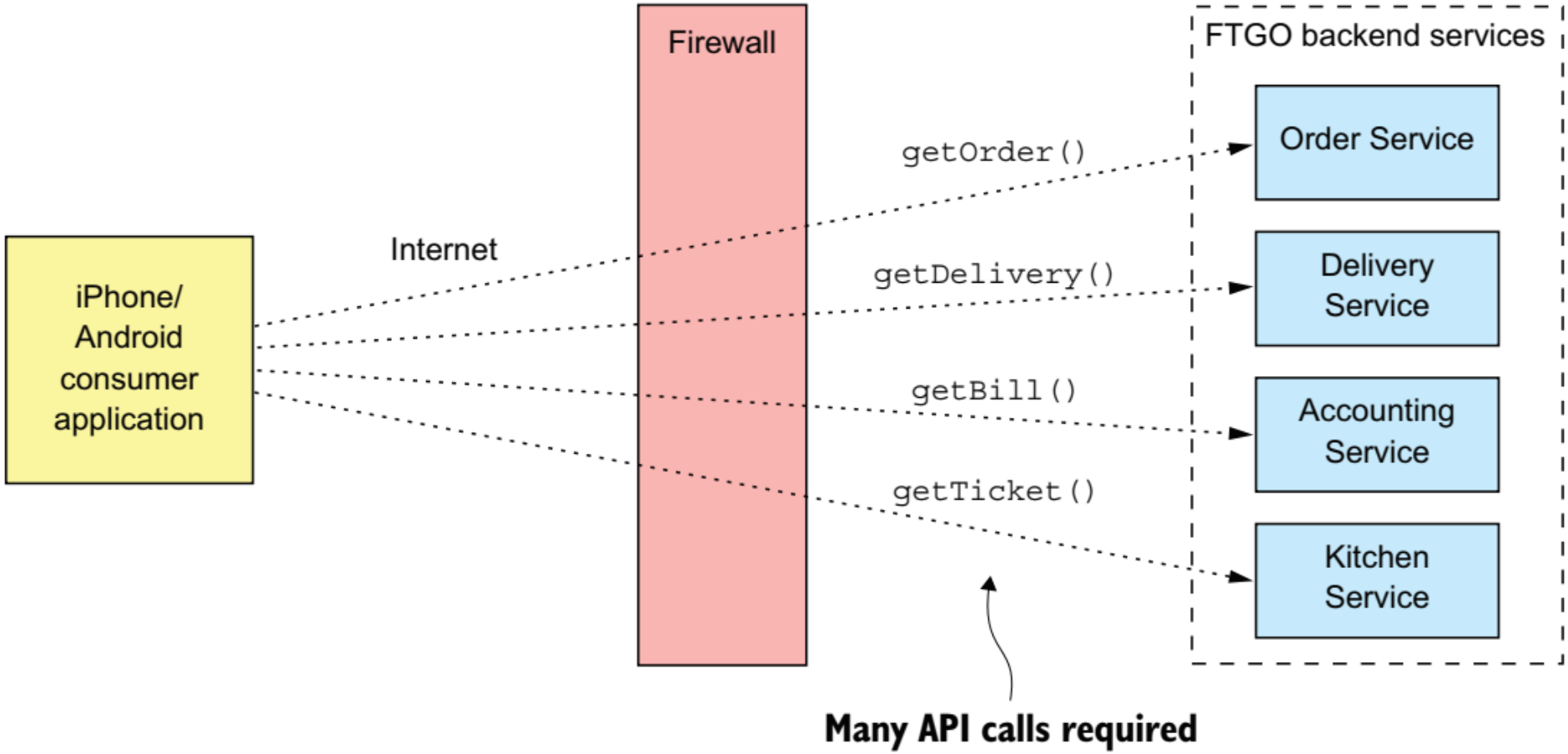
- Korist od ovoga šablona je što omogućava konzistentnost podataka koji obuhvataju više servisa sa lokalnim bazama
- Nedostatak saga je povećana kompleksnost. Na primer, moraju se implementirati kompenzacione transakcije koje eksplicitno poništavaju izmene koje su ranije urađene u okviru sage.
- Da bi radila pouzdano, servis mora atomično ažurirati svoju bazu i publikovati događaj. Za ovo postoje posebni obrasci (pogledati u knjizi navedenoj na kraju).
- Takođe ovaj obrazac ne obezbeđuje izolovanost konkurentnih transakcija tako da se mogu javiti anomalije (na primer jedna saga vrši upis a da nije pročitala izmene druge sage). Za ovo se moraju primeniti određene kontramere što nećemo obrađivati.

APLIKATIVNI INTERFEJS PREMA SPOLJNOM SVETU

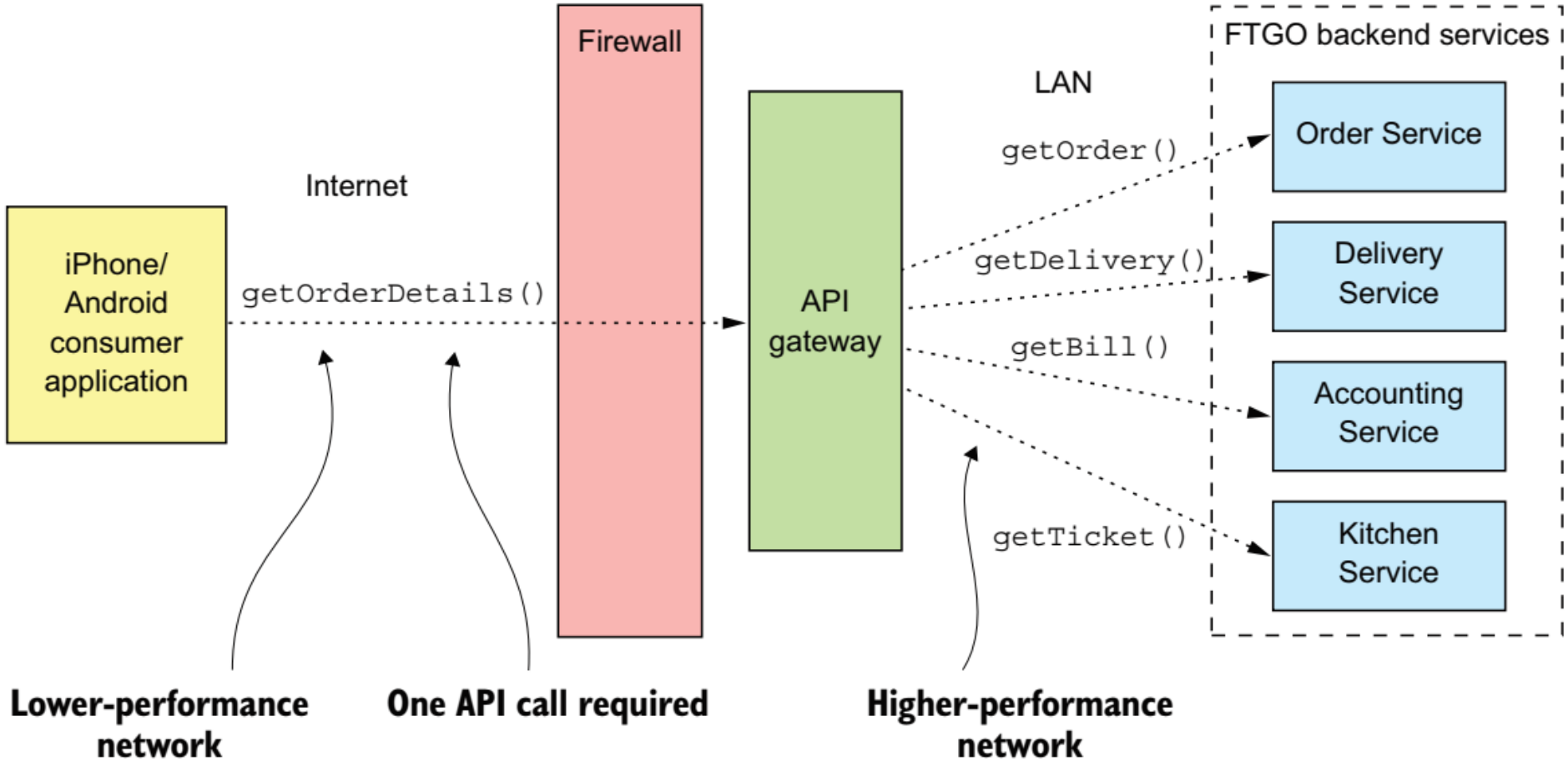
Klijenti mikroservisne aplikacije



Klijenti mikroservisne aplikacije



API Gateway (aplikativni interfejsni prolaz)



Zaduženja API prolaza

- **Usmeravanje (routing) zahteva na odgovarajući servis.** Kada primi zahtev, API prolaz konsultuje mapu usmeravanja koja specificira na koji servis da usmeri zahtev. Mapa usmeravanja može, na primer, mapirati HTTP metod i put do HTTP URL adrese usluge. Ova funkcija je identična funkcijama obrnutog proksiranja koje pružaju veb serveri poput NGINX.
- **Kompozicija APIja** - API mrežni prolaz obično radi više nego jednostavno obrnuto proksiranje. Takođe može pružiti aplikativni interfejs većeg nivoa granularnosti. Klijent upućuje jedan zahtev API prolazu, koji komponuje odgovor obraćajući se većem broju mikroservisa.

Zaduženja API prolaza

- **Translacija protokola** – prolaz može klijentima obezbediti RESTFul interfejs, iako mikroservisi interno koriste na primer gRPC.
- **Implementacija ivičnih (edge) funkcija –** Ivična funkcija je kao ime sugeriše, funkcija obrade zahteva implementirana na ivici aplikacije. Primeri ivičnih funkcija koje aplikacija može implementirati su sledeći:
  - Autentifikacija - provera identiteta klijenta koji podnosi zahtev.
  - Autorizacija - provera da li je klijent ovlašćen da izvrši određeni zahtev.
  - Ograničenje brzine - Ograničavanje broja zahteva u sekundi od određenog klijenta ili od svih klijenata.
  - Keširanje - Keširanje odgovora radi smanjenja broja zahteva upućenih servisima.
  - Prikupljanje metrika - prikupljajte pokazatelje o upotrebi API-ja u svrhu obračuna plaćanja.
  - Logovanje zahteva – pamćenje evidencije o zahtevima sa klijenata.

Implementacija API prolaza

Dve mogućnosti:

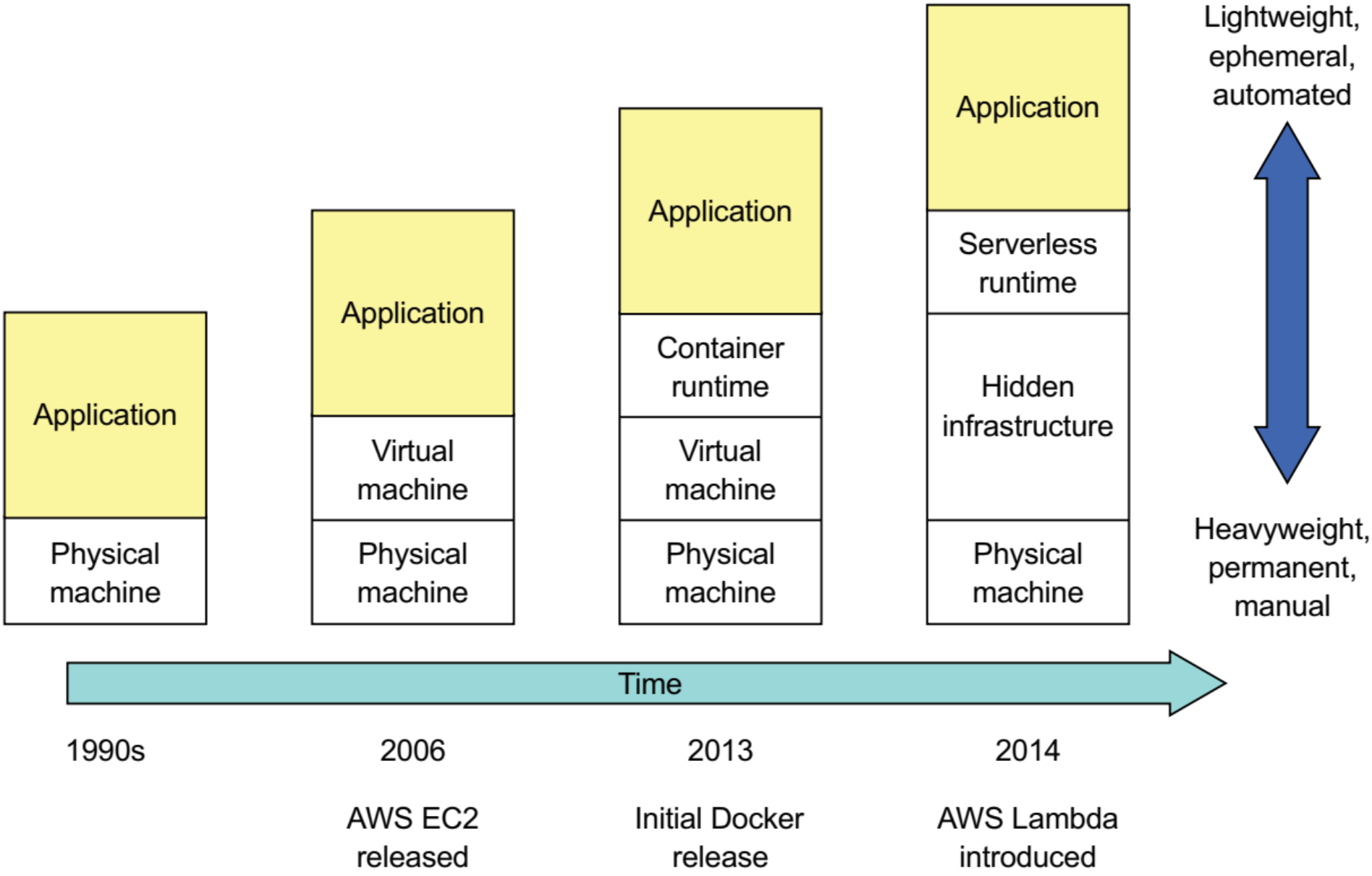
- Upotreba gotovog (off-the-shelf) proizvoda ili servisa na primer, AWS API Gateway, Kong, NGINX Plus itd. Obično je ovo pravi put.
- Razvoj softvenog zasnovanog na nekom gotovom frejvorku. Samo ako postoje neki jako specifični zahtevi koji standardni prolazi ne mogu da zadovolje, ali zahteva resurse za razvoj i održavanje a lako može da se dogodi da performanse ili skup funkcionalnosti budu smanjeni.

RASPOREĐIVANJE (DEPLOYMENT) MIKROSERVISA

Raspoređivanje mikroservisa

- Raspoređivanje servisa u radno okruženje definisano je sa dva povezana pojma:
  - arhitekturom sistema i
  - procesom puštanja u rad.
- Produkciona arhitektura sistema definiše strukturu (hardversku i softversku) strukturu sistema na kome se servisi izvršavaju.
- Proces puštanja servisa u rad definiše korake koje se moraju sprovesti od strane ljudi (developera i operativaca) da bi se servis pokrenuo u produkcionom okruženju.

Evolucija produkcionih arhitektura



Evolucija produkcionih arhitektura

- U prvo vreme softver (web aplikacije) raspoređivao se na fizičke mašine servere koje nije bilo lako zameniti. Softver je bio u obliku kolekcije biblioteka i komponenata koje su zavisile od konkretnog programskog jezika (na primer u slučaju php to je bila kolekcija php skriptova i pomoćnih fajlova, u slučaju jave to je mogao biti war arhiva fajlova). Deployment je rađen od strane posebnog operacionog tima, tipično ručno (npr. u administrativnoj konzoli aplikativnog servera).
- Sredinom 2000tih, web aplikacije i servisi počeli su se raspoređivati na virtuelne mašine umesto fizičkih servera. Virtuelizacijom se resursi (procesor, memorija,...) jedne fizičke mašine dele između više virtuelnih softverskih mašina od kojih svaka ima izgled kompletne fizičke mašine sa kompletnim operativnim sistemom (samo sa manje fizičkih resursa).

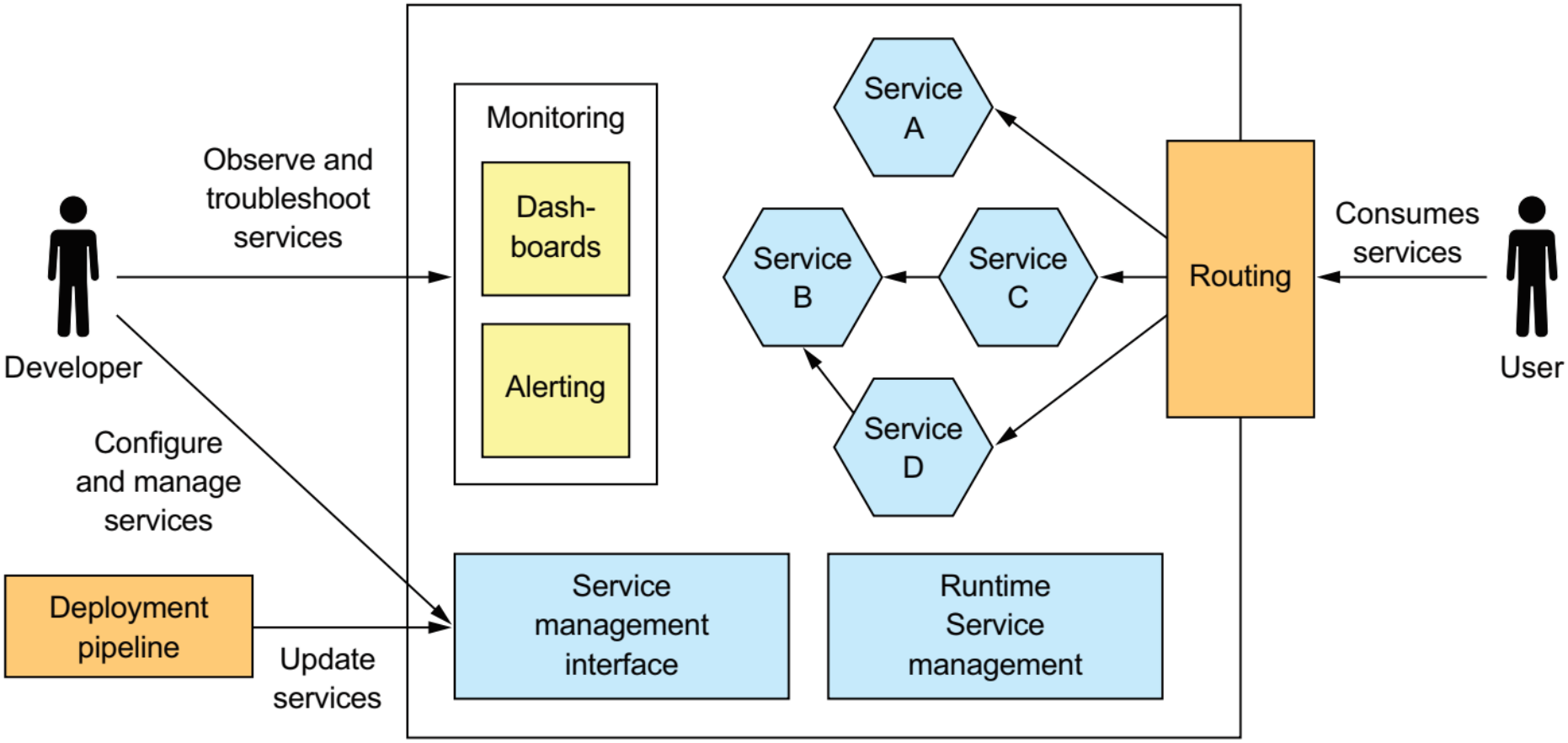
Evolucija produkcionih arhitektura

- Zbog posedovanja kompletnog sistemskog softvera virtuelne mašine spadaju u “tešku kategoriju” virtuelizacije operativnog okruženja.
- Servisi su se mogli pakovati kao slike (image) virtuelnih mašina, gde je sa aplikativnim kodom bio upakovan i ceo tehnološki stek (biblioteke, web serveri i slično) potreban za rad servisa. Proces pravljenja slike je još uvek bio najvećim delom manuelan. Takve slike mogle su se po potrebi lako instancirati i uklanjati sa fizičkih mašina.

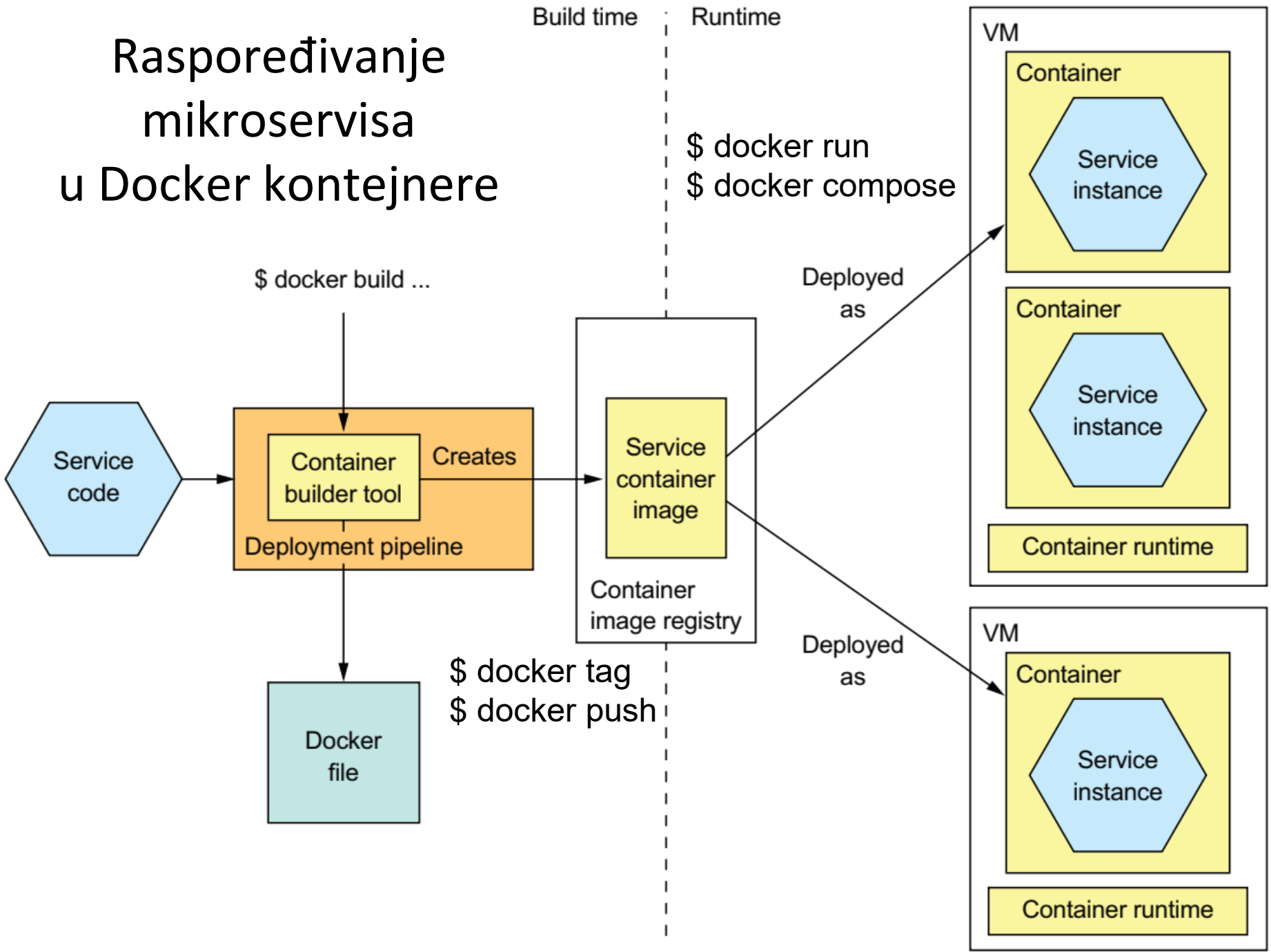
Evolucija produkcionih arhitektura

- Sredinom 2010tih popularizuje se Docker, tehnologija virtuelizacije na nivou operativnog sistema. Kontejnerska slika je samostalni izvršni softverski paket koji ima upakovan kod i sve njegove zavisnosti (podešavanja, sistemske biblioteke, sistemske alate), Pošto se više kontejnera (instanci kontejnerskih slika) izvršava u okviru istog operativnog sistema, njima treba manje resursa neko virtuelnim mašinama i zato spadaju u laku kategoriju virtuelizacije. Instance mikroservisa vrte su u kontejnerima.
- Još lakša kategorija produkcione arhitekture je serverless (ili FaaS, function as a service) gde se aplikativna logika na serveru izvršava u računarskim kontejnerima koji nemaju stanje, pokretani su događajima (event-triggered), kratkotrajni (mogu egzistirati samo jedno izvršavanje) i potpuno kontrolisani od strane provajdera. Serverles funkcije mogu se kombinovati sa tradicionalnim mikroservisima za manje troškove rada.

Savremeno produkciono okruženje



Raspoređivanje mikroservisa u Docker kontejnere



Literatura

- Chris Richardson, Microservices Patterns: With examples in Java, 2019.