12

# Domény, modely a mikroservisy

- Microservisy a mikroservisní patterny
- Konzistence
- DDD
- Event sourcing a CQRS

Next >

Objektové Modelování - podzim 2022

david.kadlecek@cvut.cz, www.czm.fel.cvut.cz

# MICROSERVISY

Microservisní architektura je o redukci komplexity pomocí dekompozice systému na moduly, které jsou dobře ohraničené z hlediska fungování a účelu.

V business termínech by microservice měla realizovat jednu ***business capabilitu*** *(schopnost či dovednost)*

Microservice nemá nic společného s velikostí, ale je o maximálním respektování **Single Responsibility Principle** (**S**OLID)

Microservisy fungují zcela samostatně a autonomně.

Microservice preferuje, aby měla vlastní databázi do které napřímo nepřistupuje žádný další systém (či microservice)

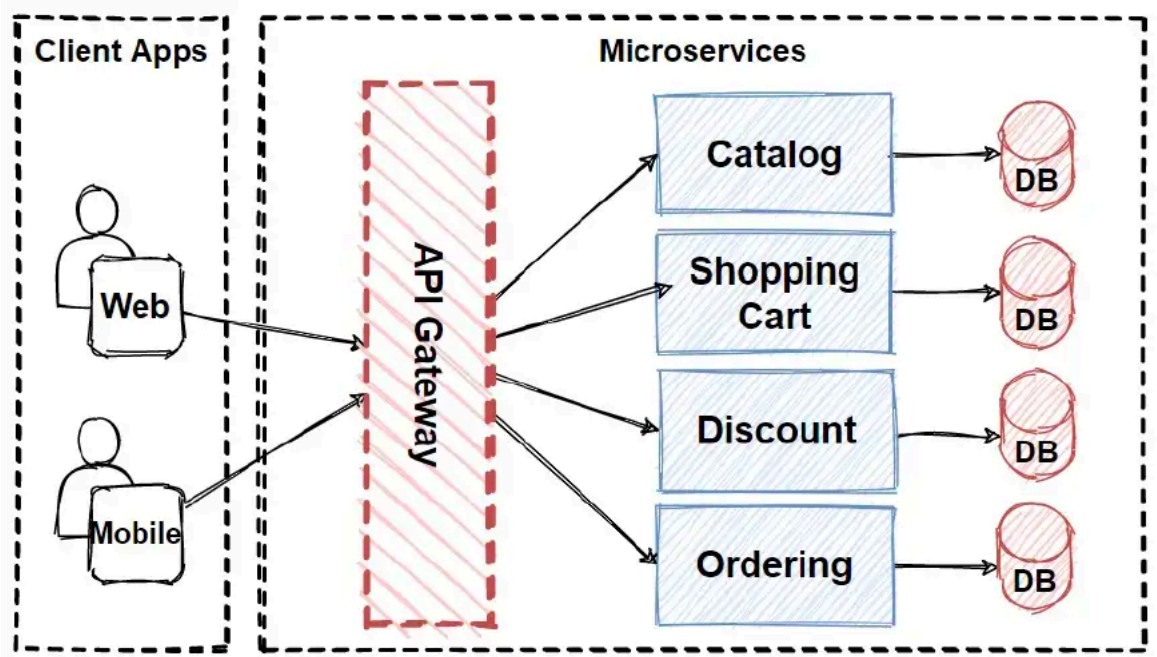
# MICROSERVISY

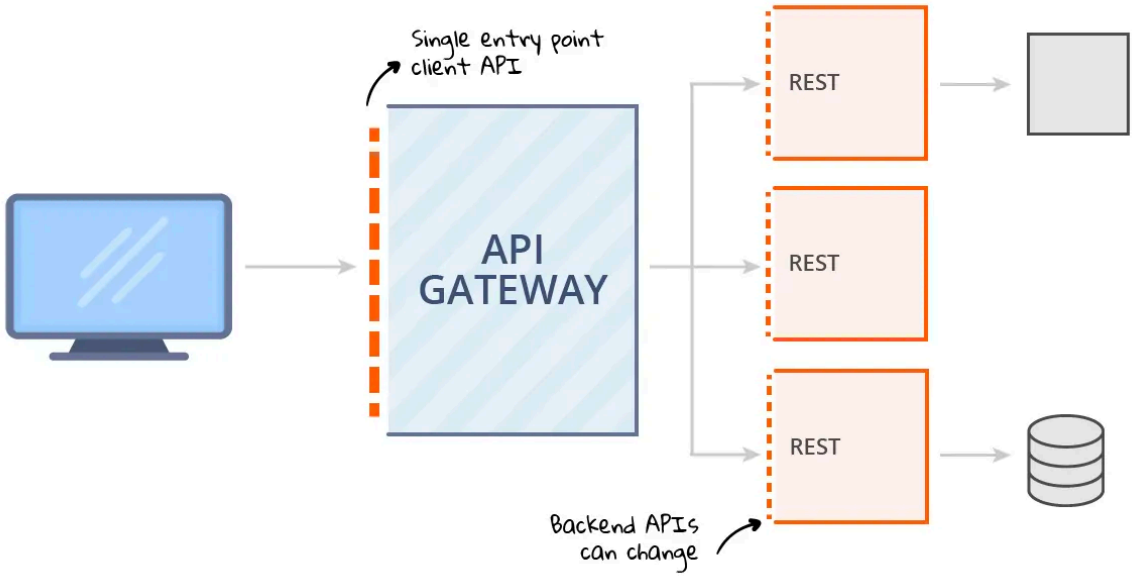
Dvě microservisy mohou sdílet data a odpovědnost. Děje se tak přes třetí subjekt. Závislost je však co nejmenší a nejvolnější.

Dvě autonomní microservisy a sdílená databáze  (TIE FIGHTERS a HVĚZDA SMRTI) =>

# 12 API GATEWAY

- Verzování služeb (a management API)
- Autentizace a autorizace
- Logování a caching
- Rozkládání zátěže (a throttling)
- Oddělení API pro frontendy ad API backendů



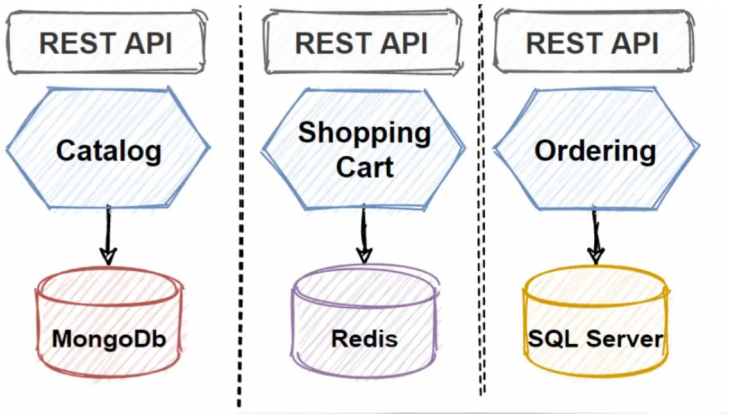
# 12 Transformation/ microservice

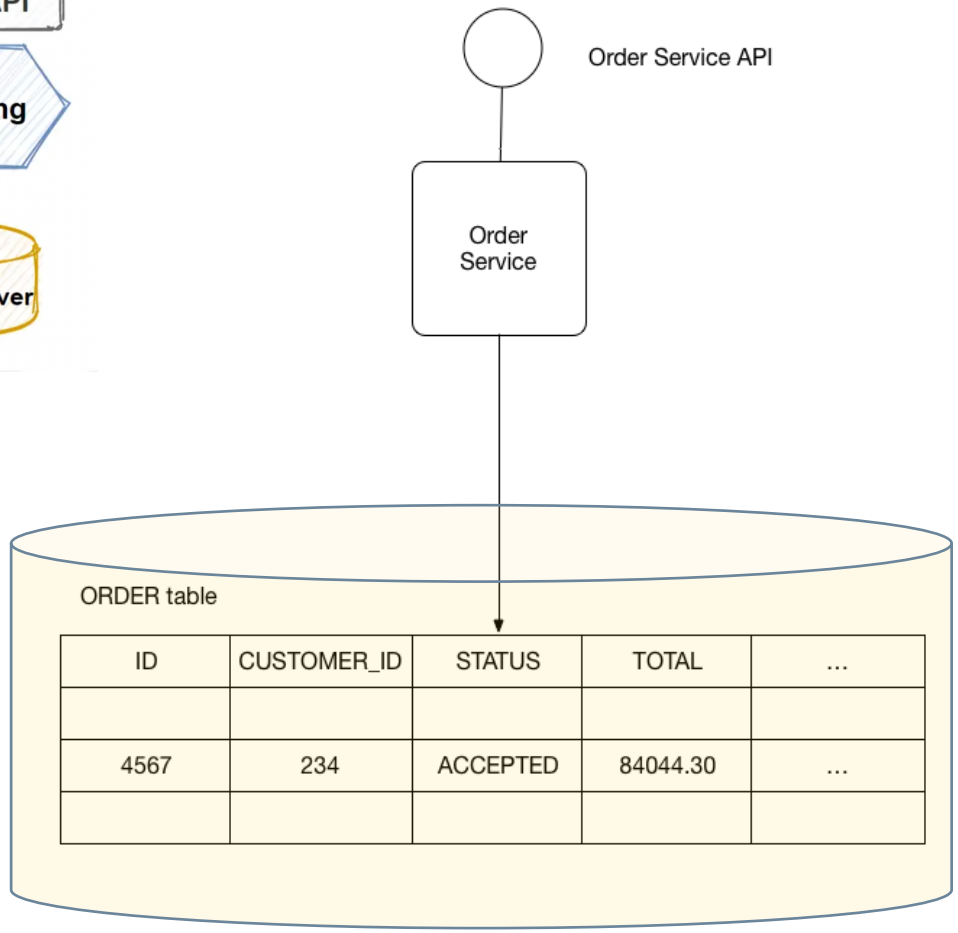
**PROBLÉM:** Potřebujeme provést transformaci dat nebo bezestavový výpočet

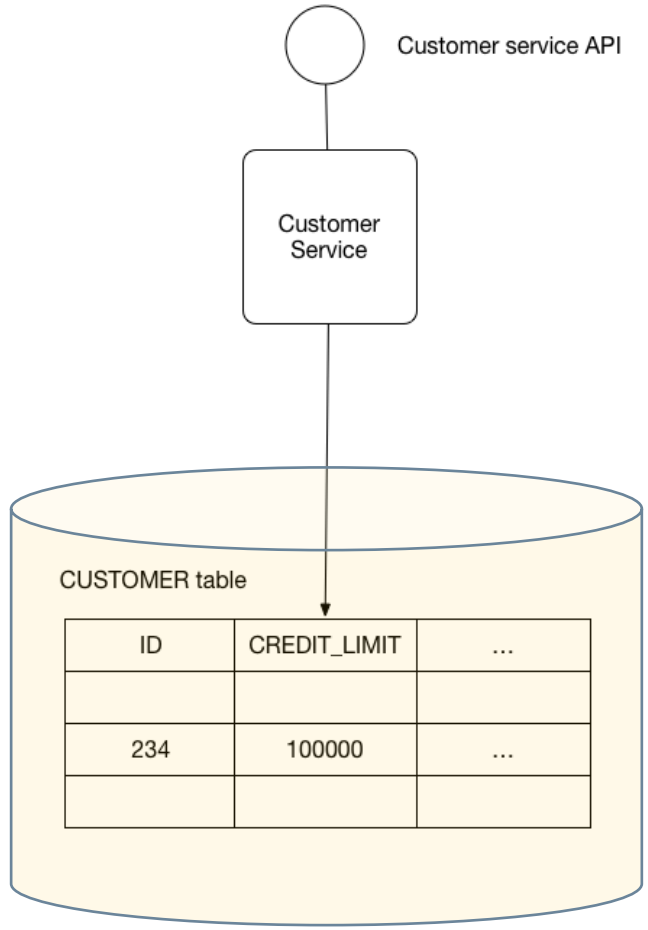
=> Microservice nemá napojení na žádnou databázi ani backend, je idempotentní a bezestavová, vrací finální výsledek nebo mezivýsledek.

# Database per service microservice

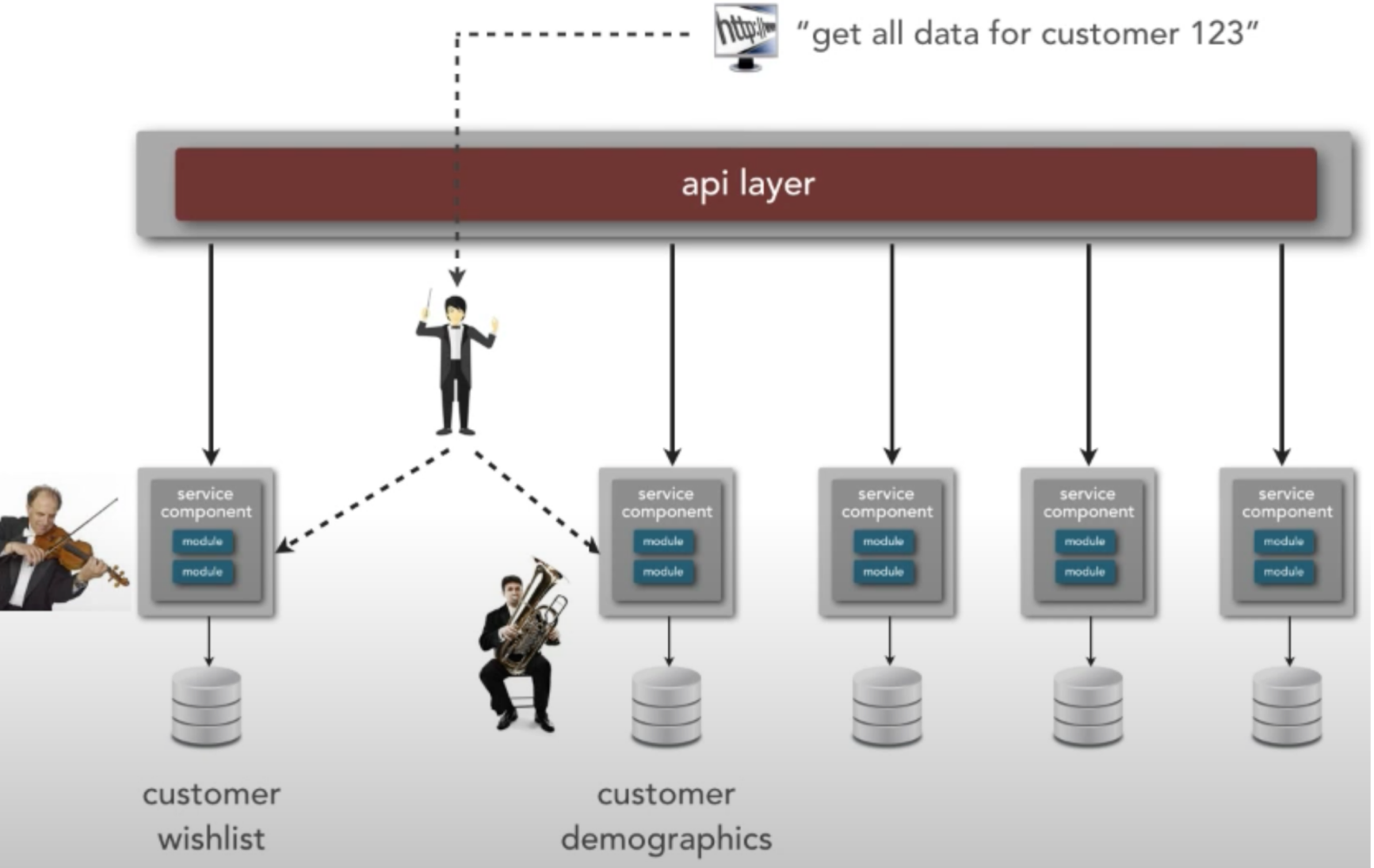
Nezávislé služby mohou provádět operace vedoucí k nekonzistentním datům (např. založím Order s ID zákazníka, který neexistuje)



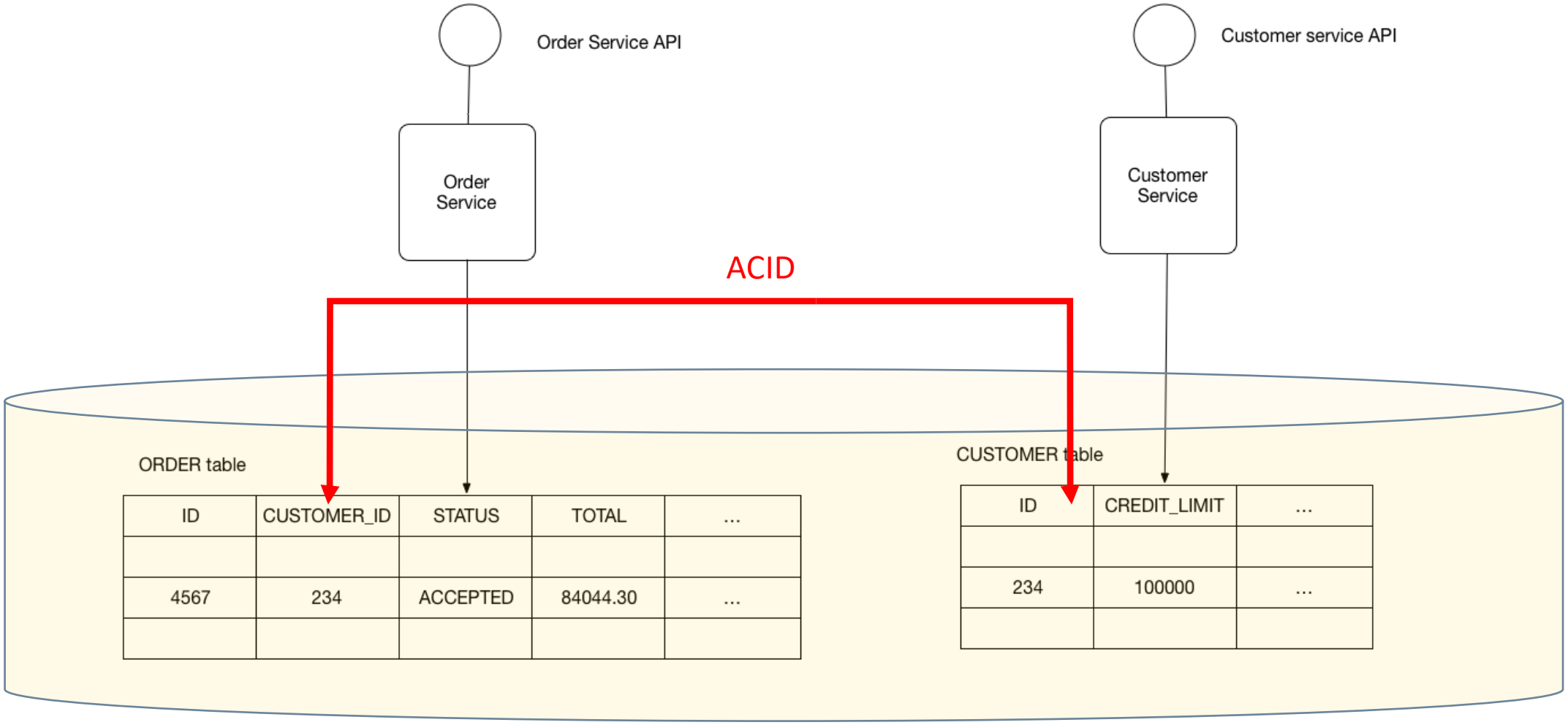
# 12 Orchestrace mikroservis



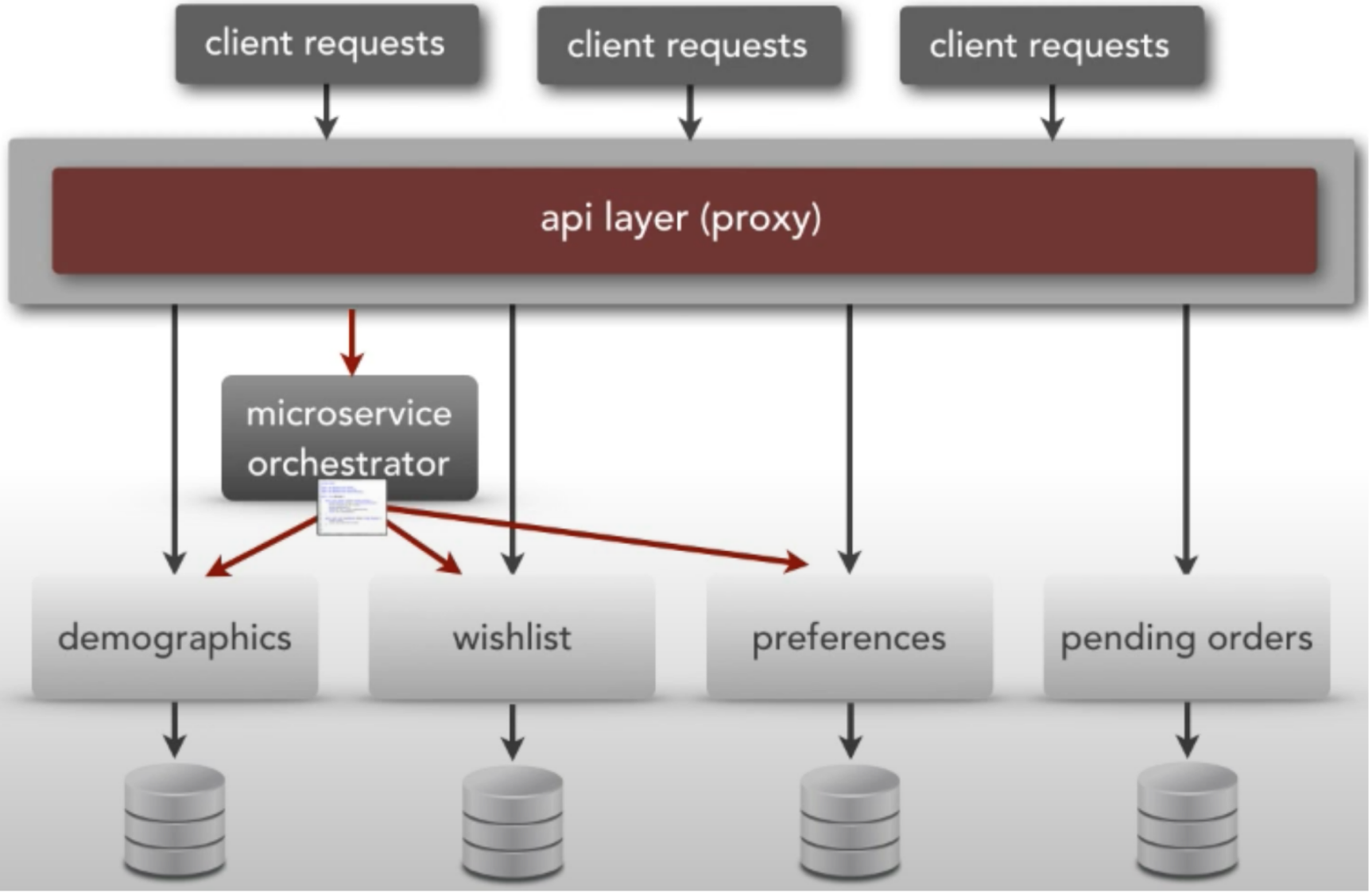

**PROBLÉM:** Potřebujeme službu, která provede volání více služeb (může si např. předávat mezivýsledek jednotlivých volání) a vrátí výsledek, kterým je výsledkem těchto více volání

# Shared database microservice

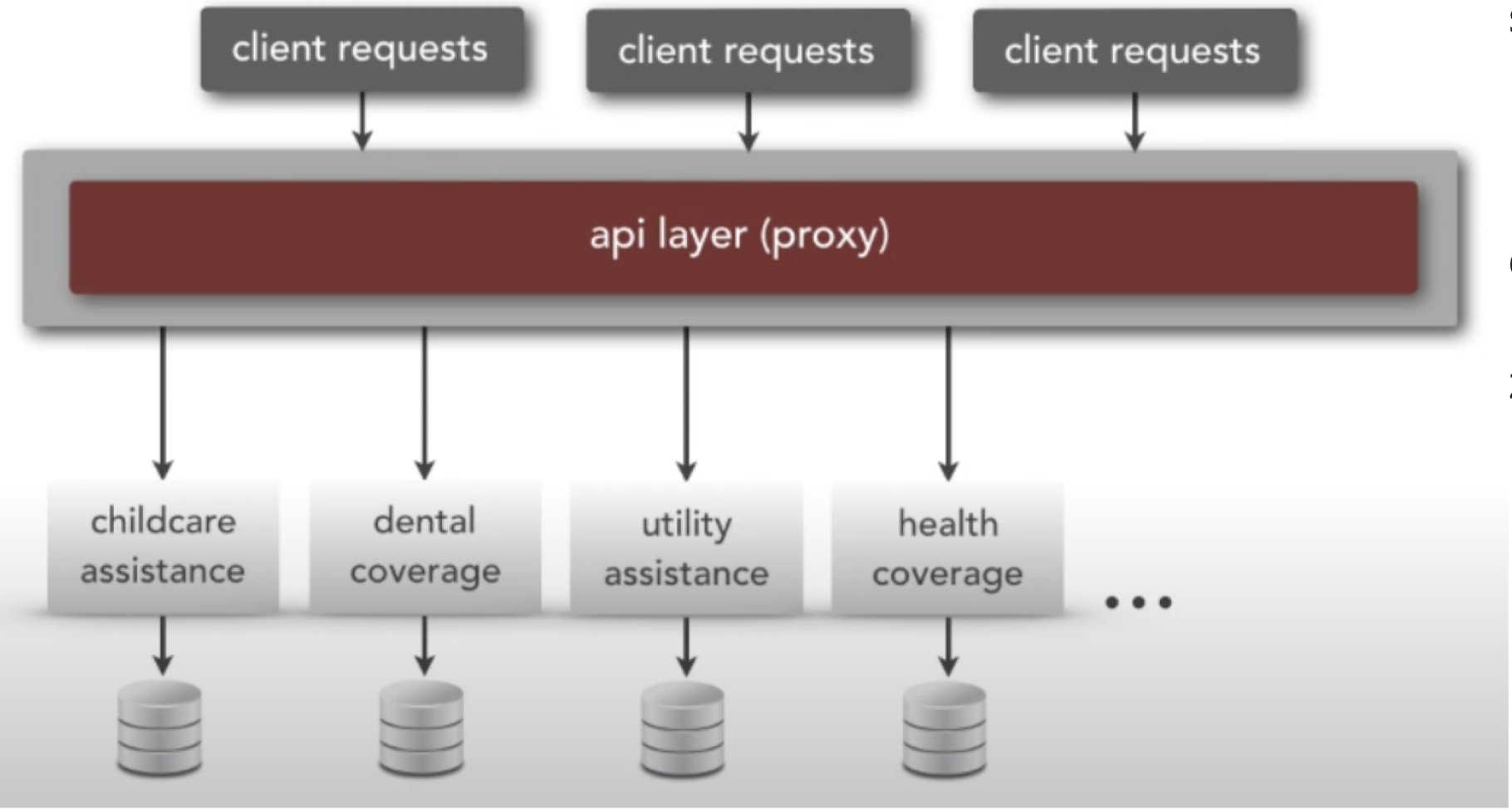
Operace prováděné více nezávislými službami nemohou vytvářet nekonzistentní data



# 12 Orchestrace mikroservis



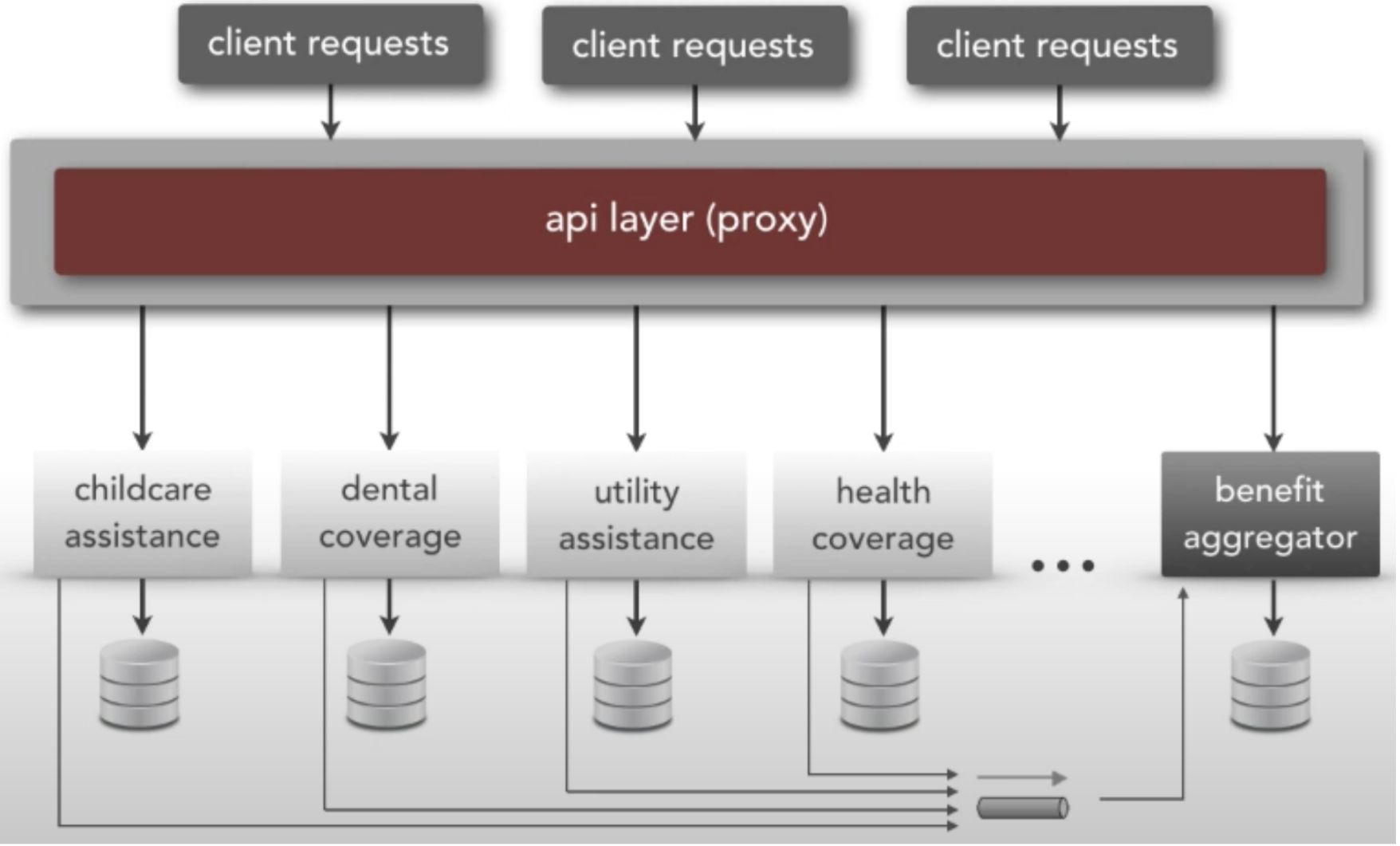
# 12 Agregace mikroservis




**PROBLÉM:** Potřebujeme držet data, která rse dvíjí od toho jak interagují existující služby.

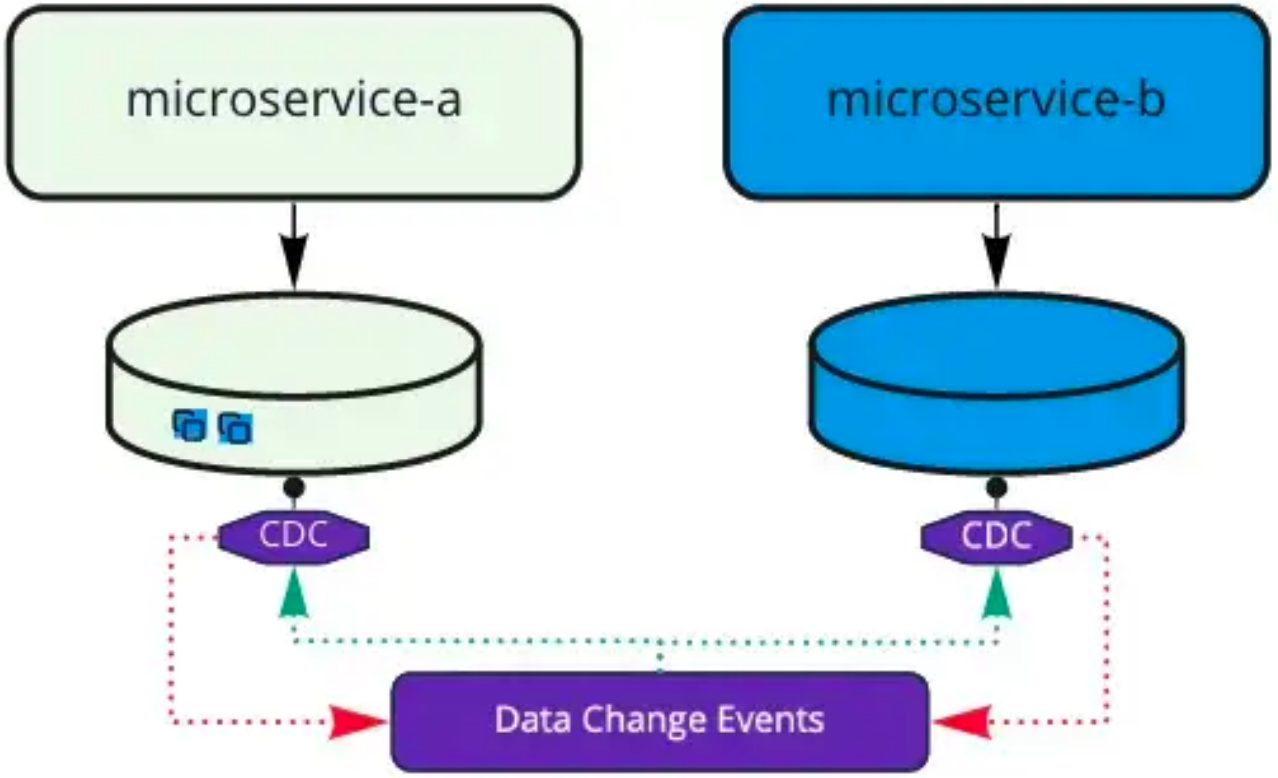
Např. Potřebujeme službu, která nám vrátí data odpovídající celkovým benefitům poskytovaným zákazníkovi. Benefity přitom přibývají na základě aktivity klienta (čerpání produktů poskytovaných firmou)

# 12 Agregace mikroservis



# 12 CDC - Change Data Capture

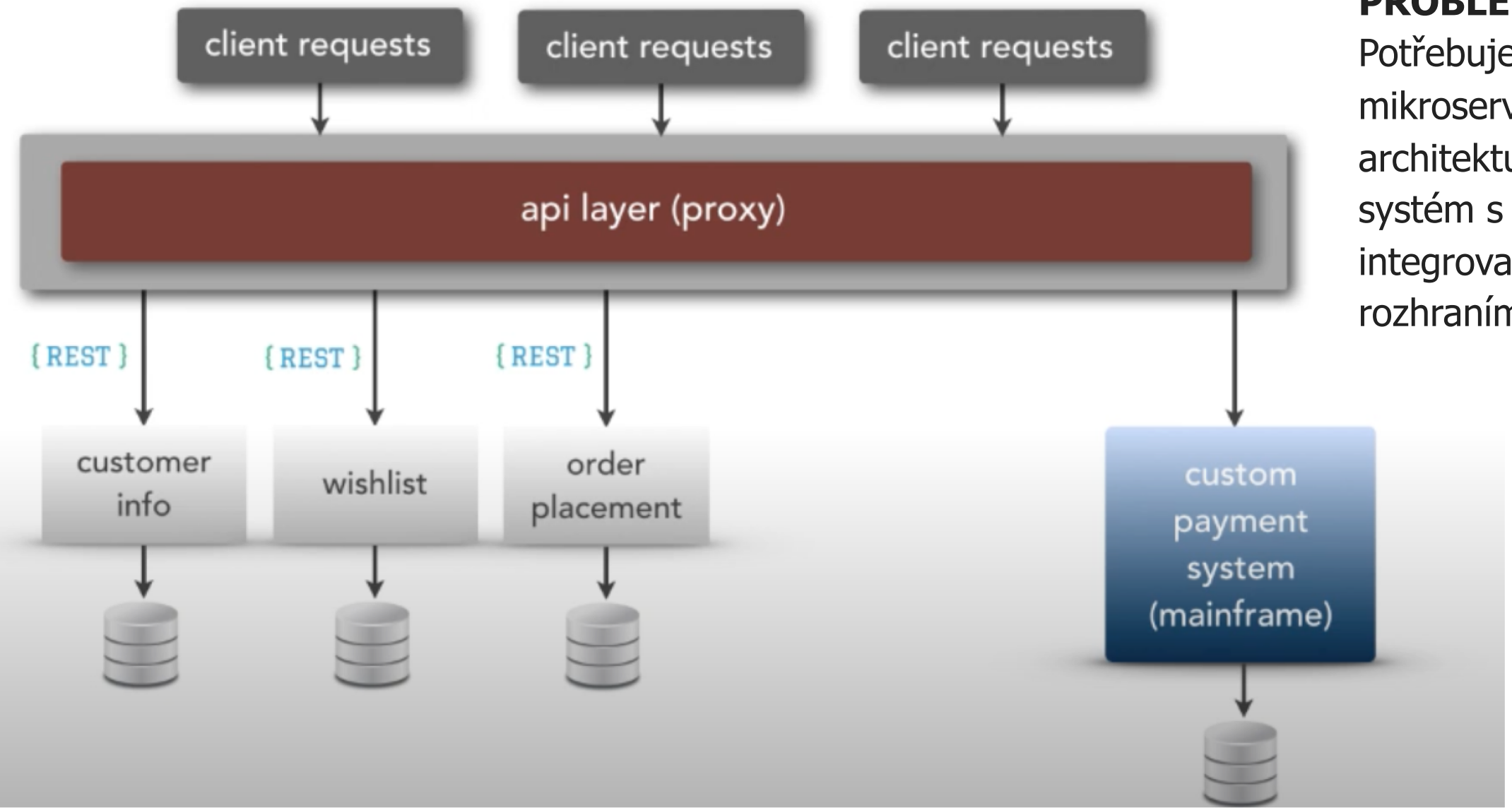
CDC nástroj je napojen na úložiště dat, konkrétně na jeho transakční logy, aby výkonově nezatěžoval datové úložiště a přetváří změny v datech v eventy. Ty je pak možné aplikovat jinde – např. Provoláním operace na jiné mikroservise nebo změnou dat v jiném datovém úložišti.




**PROBLÉM:** V jedné microservise chci reagovat na změny v datech spravovaných druhou mikroservisou nebo chci jako eakci přímo měnit obsah svých dat.

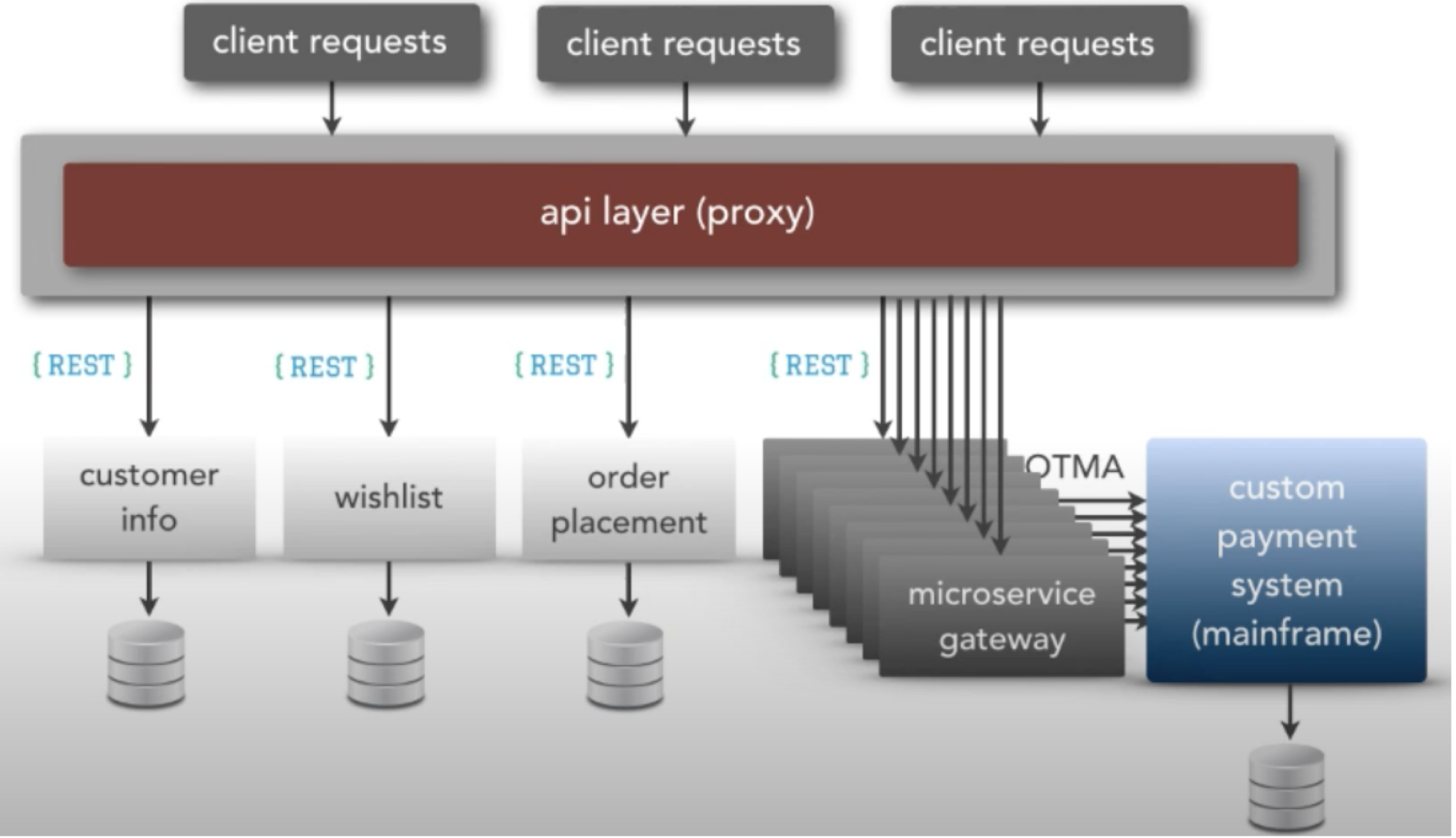
*Příkladem technologie umožňující CDC je Kafka Debezium nebo Oracle Golden Gate*

# 12 Microservisní Gateway (Brownfield)




**PROBLÉM:**
Potřebujeme do mikroservisní architektury zapojit starý systém s těžko integrovatelným rozhraním

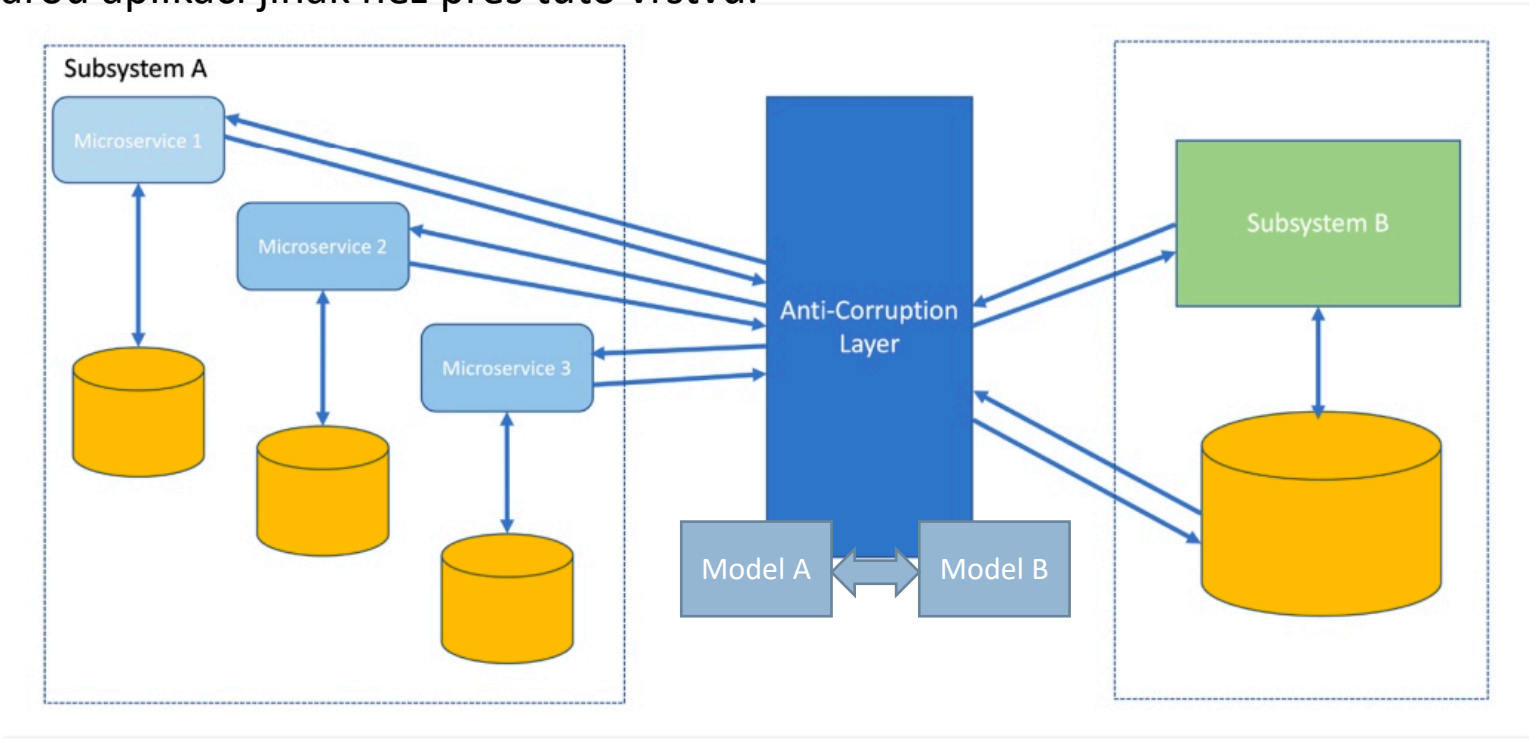
# 12 Microservisní Gateway (Brownfield)



# 12 BROWN FIELD PATTERNY - ANTI CORRUPTION LAYER

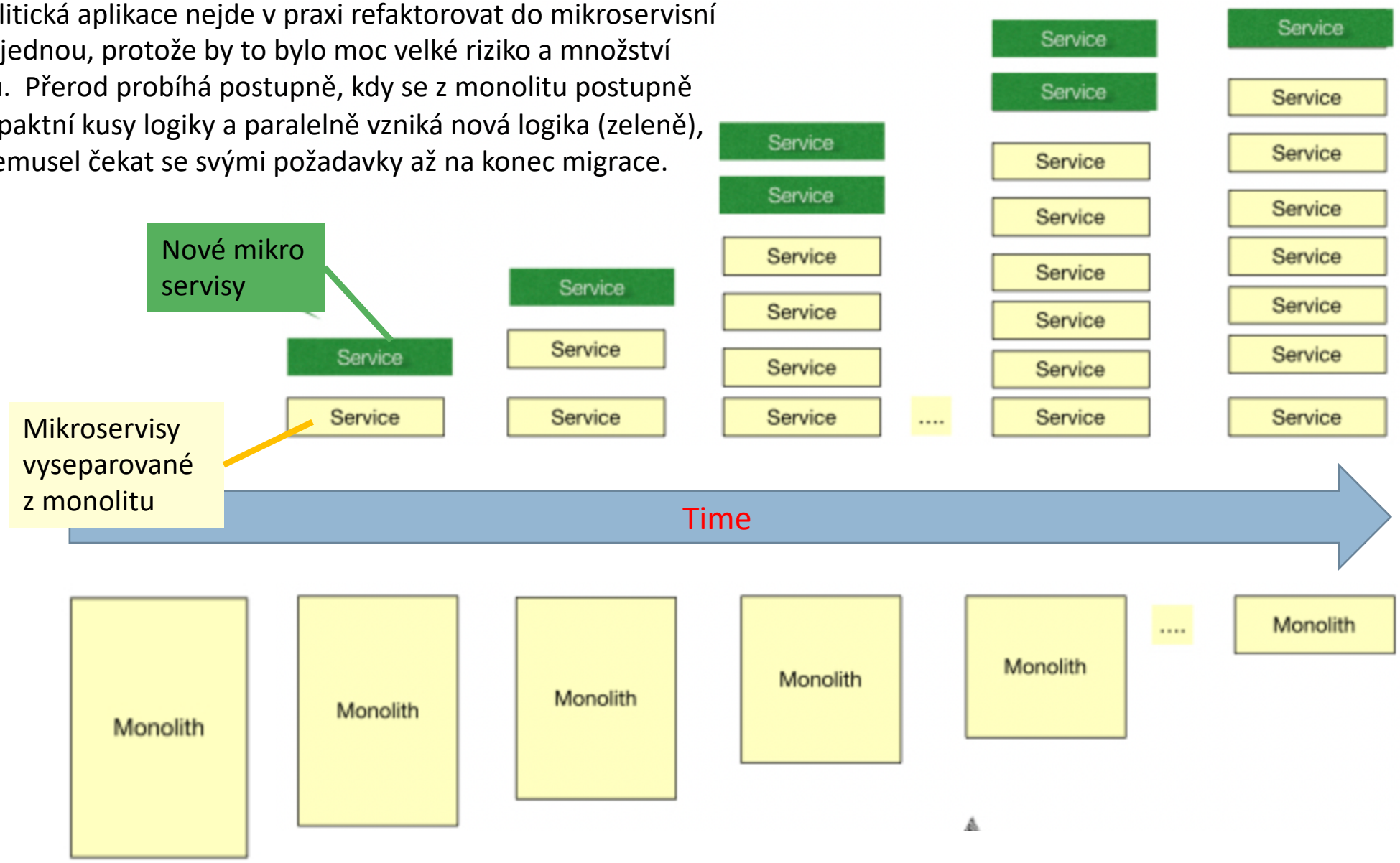
Když se moderní aplikace potřebuje integrovat se starší aplikací, je náročné spolupracovat se zastaralými infrastrukturními protokoly, API a datovými modely. Lpění na starých modelech a sémantice může poškodit nový systém.

Můžeme se tím vyhnout, že implementujeme vrstvu, která překládá komunikaci mezi dvěma systémy. ***Anticorruption Layer*** odpovídá datovému modelu staršího nebo moderního systému v závislosti na tom, se kterým systémem komunikuje, aby data získala. Zajišťuje, že se starý systém nemusí měnit a moderní systém nedělá kompromisy ve svém designu a technologii. Žádoucí je naopak zcela znemožnit, aby nové systémy mohly komunikovat se starou aplikací jinak než přes tuto vrstvu.



# 12 BROWN FIELD PATTERNY – STRANGLER (ŠKRTIČ)

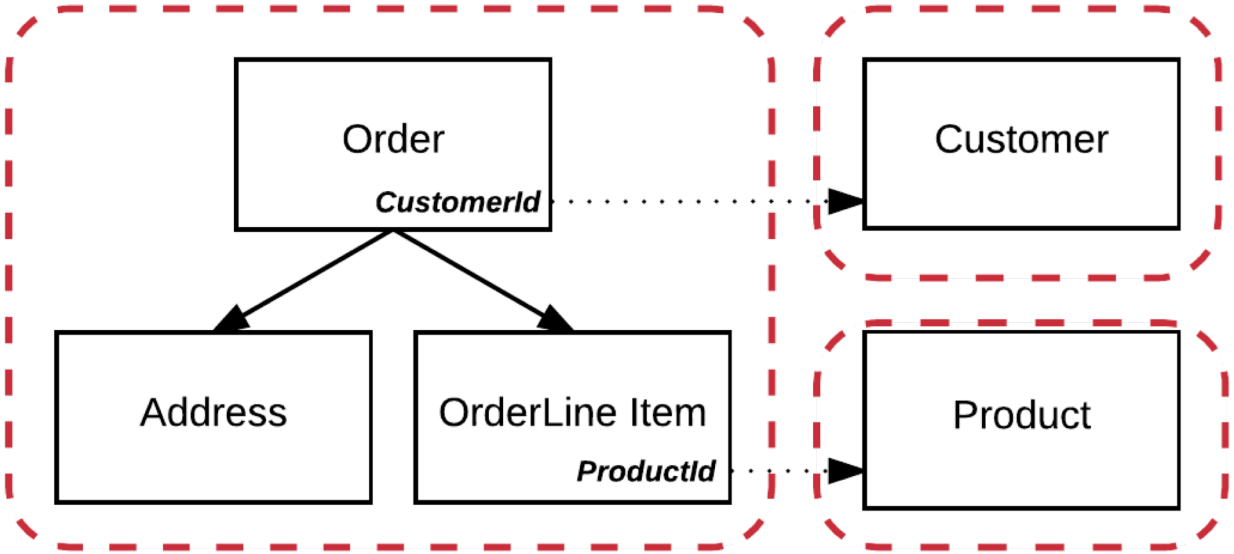
Rozsáhlá monolitická aplikace nejde v praxi refaktorovat do mikroservisní architektury nejednou, protože by to bylo moc velké riziko a množství změn najednou. Přerod probíhá postupně, kdy se z monolitu postupně odřezávají kompaktní kusy logiky a paralelně vzniká nová logika (zeleně), aby business nemusel čekat se svými požadavky až na konec migrace.



# 12 Mikroservisy - problém s konzistencí

V klasickém monolitické systému jsme mohli najednou měnit Order, Customer i Product, protože byly ve stejné databázi a přistupovali jsme k nim přes stejnou vrstvu.

**Jak zařídíme konzistentnost v systému dekomponovaném na microservisy?**




Nejdříve je nutné pochopit co je transakce ve smyslu klasické relační databáze ...

# ACID

Relační databáze splňují ACID při operacích (transakcích), které jsou prováděny s daty v databázi:

- *Atomicity* - operace provede z hlediska uživatele buď celá, nebo vůbec – tedy výsledkem nebude žádný nedefinovaný mezistav.
- *Consistency* - transakce nesmí narušit databázovou integritu. To nevylučuje ztrátu konzistence na vyšší, aplikační úrovni, požadavek se týká pouze zachování podmínek, které byly definovány na databázové úrovni.
- *Isolation* - vícero současně probíhajících transakcí se nesmí ovlivnit.
- *Durability - j*akmile je již transakce dokončena, znamená to, že je zaznamenána trvalým způsobem, takže ji nevymaže například následný výpadek napájení

ACID je splněn pokud jsou data v jedné databázi a nebo ve více databázích, které podporují dvoufázový commit (viz backup)

# 12 Sága a eventuální konzistence

Dočasně nekonzistentní systémy = **Eventual consistency**

Z ***technických*** nebo ***business důvodů*** nejsme schopni po celou dobu zaručit konzistentnost

=> navrhneme systém a business proces tak, aby byl použitelný i v nekonzistentních stavech (Inconsistent by design)

*Je použitelné i když není úplně funkční - tady je to spíš náhoda než záměr ;-)  =>*

Systém je konzistentní jen v některých stavech, kterými prochází. Jsou i systémy, které se do plně konzistentního stavu nedostanou nikdy

*Např. hypotéka je konzistentní těsně před schválením. V tu chvíli ke všem nemovitostem existuje výpis z katastru, ke všem osobám na hypotéce existuje doklad o příjmech atd. Předtím byla hypotéka uložená, ale např. k některým osobám nebyl uložen příjem.*

# Sága a eventuální konzistence

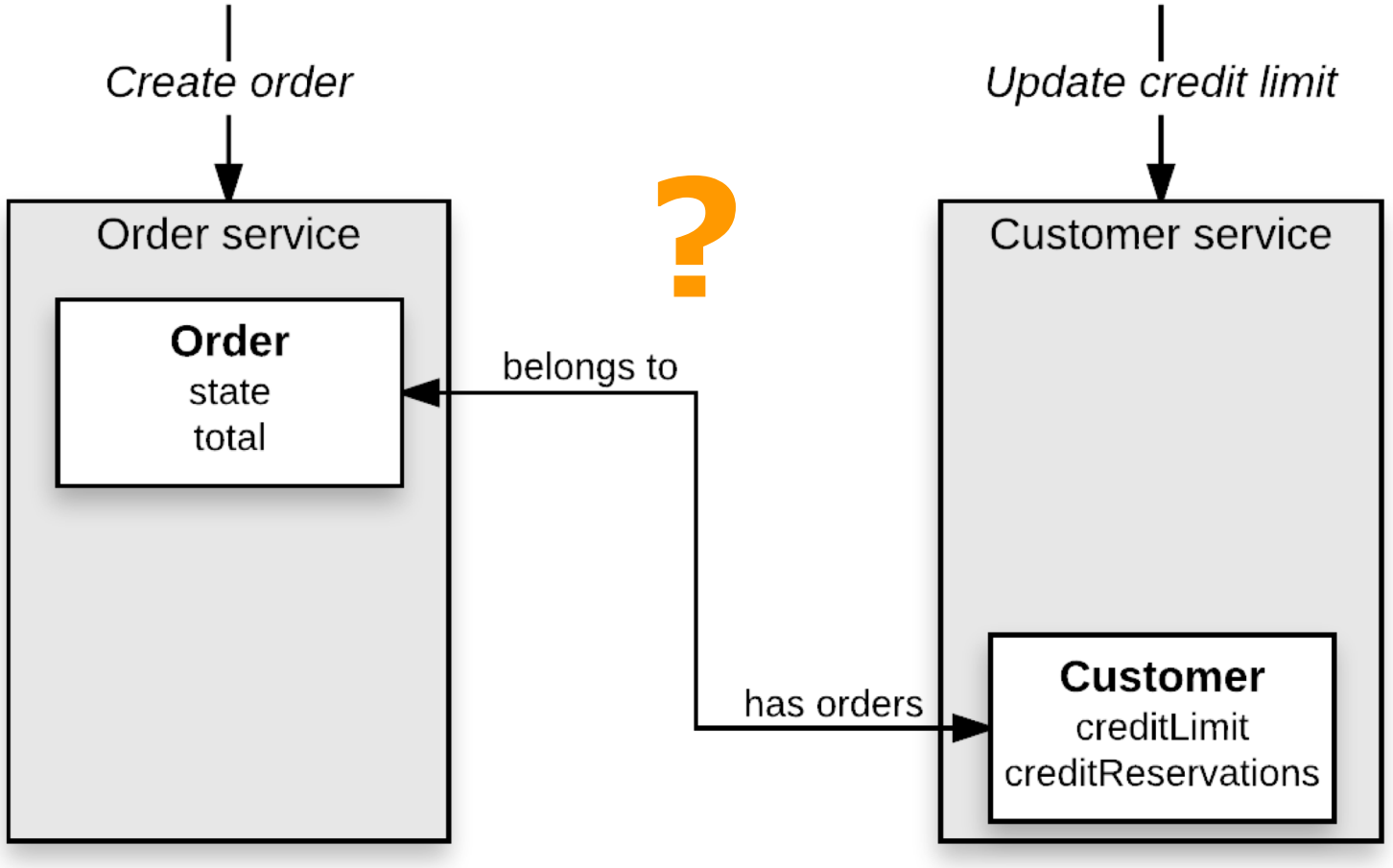
Sága řeší konsensus mezi microservisami bez transakcí a to pomocí správné dekompozice business procesu a eventual consistency.

Sága rozdělí komplexní business proces na menší akce a protiakce (counter actions), které koordinuje a řídí pomocí zpráv a timeoutů.

**Business process => Sága, která obsahuje akce, protiakce, zprávy, timeouty**
Operace v každém kroku business procesu má definovanou kompenzační operaci

# Sága a eventuální konzistence

Zakládám objednávky a potřebuji zajistit, že suma otevřených objednávek není vyšší než kreditní limit zákazníka

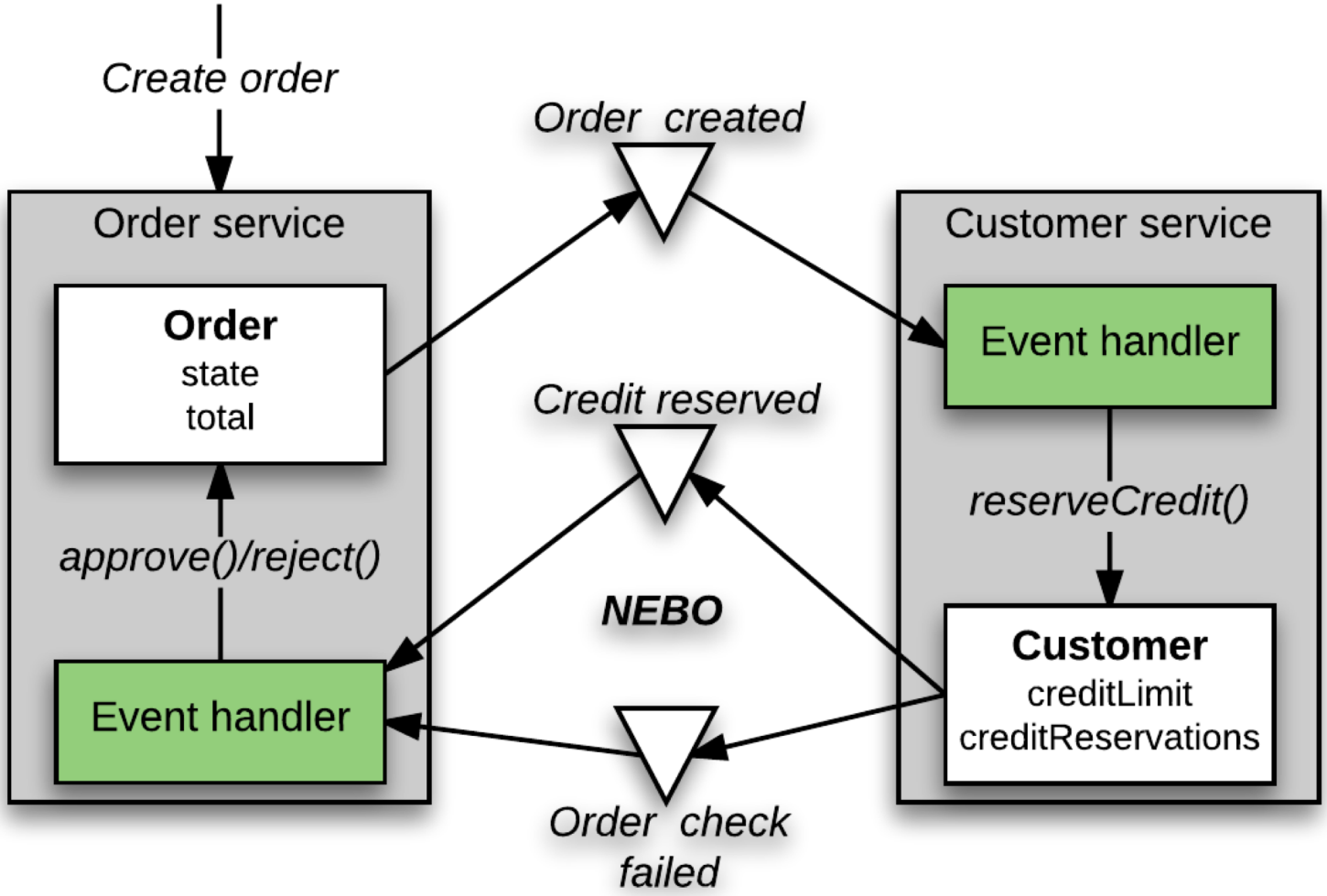


# Sága a eventuální konzistence

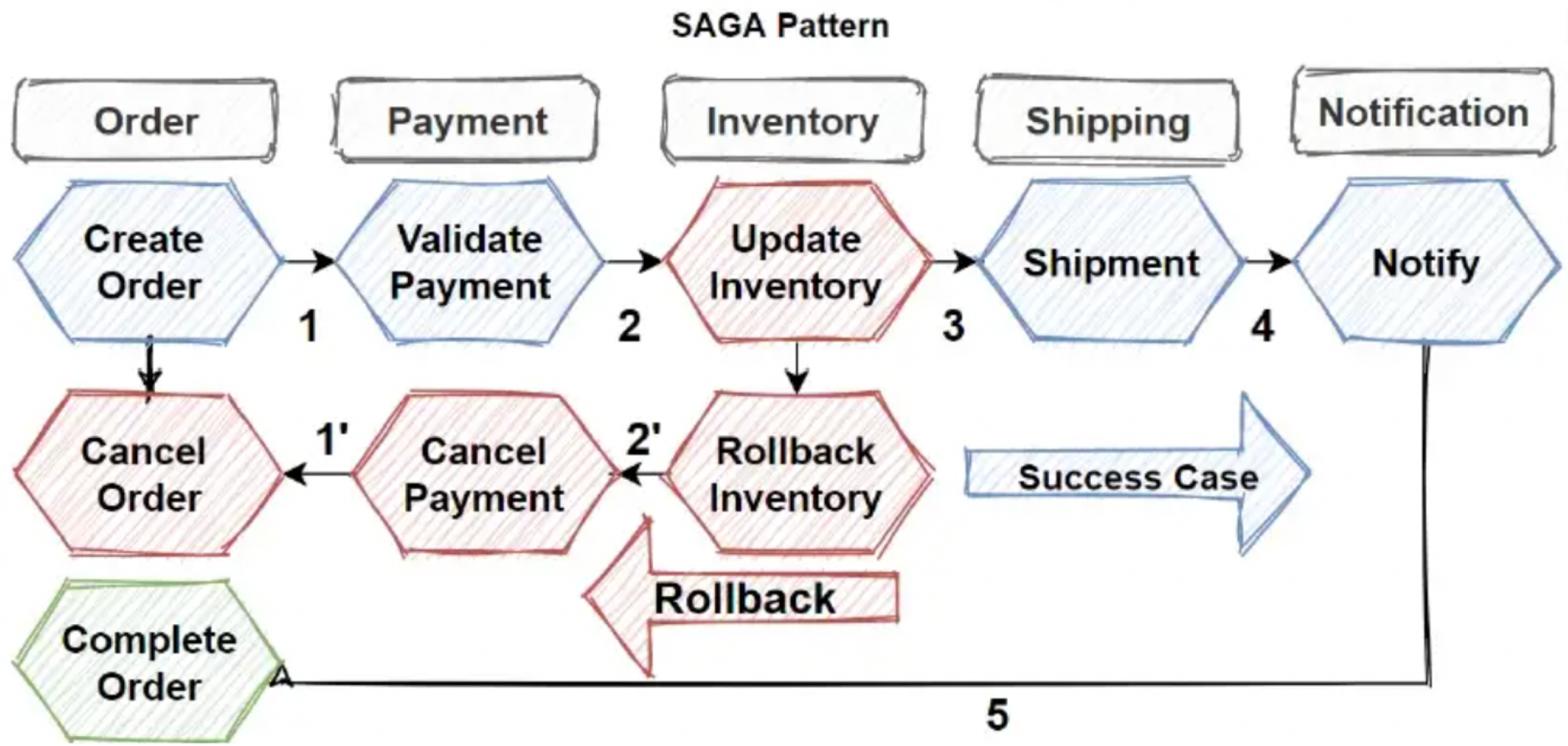
Po založení objednávky je systém po nějakou dobu částečně nekonzistentní - mohl jsem eventuálně založit objednávku, která překračuje kreditní limit zákazníka.

Toto řešení je možné, jelikož jsem **upravil business process a částečně degradoval user experience** => Zákazník čeká na potvrzení objednávky.

Obecně platí, že od jistého stavu systému (paretovsky optimální řešení) nemůžu zlepšit některé jeho vlastnosti aniž bych degradoval jiné.



# Sága a eventuální konzistence

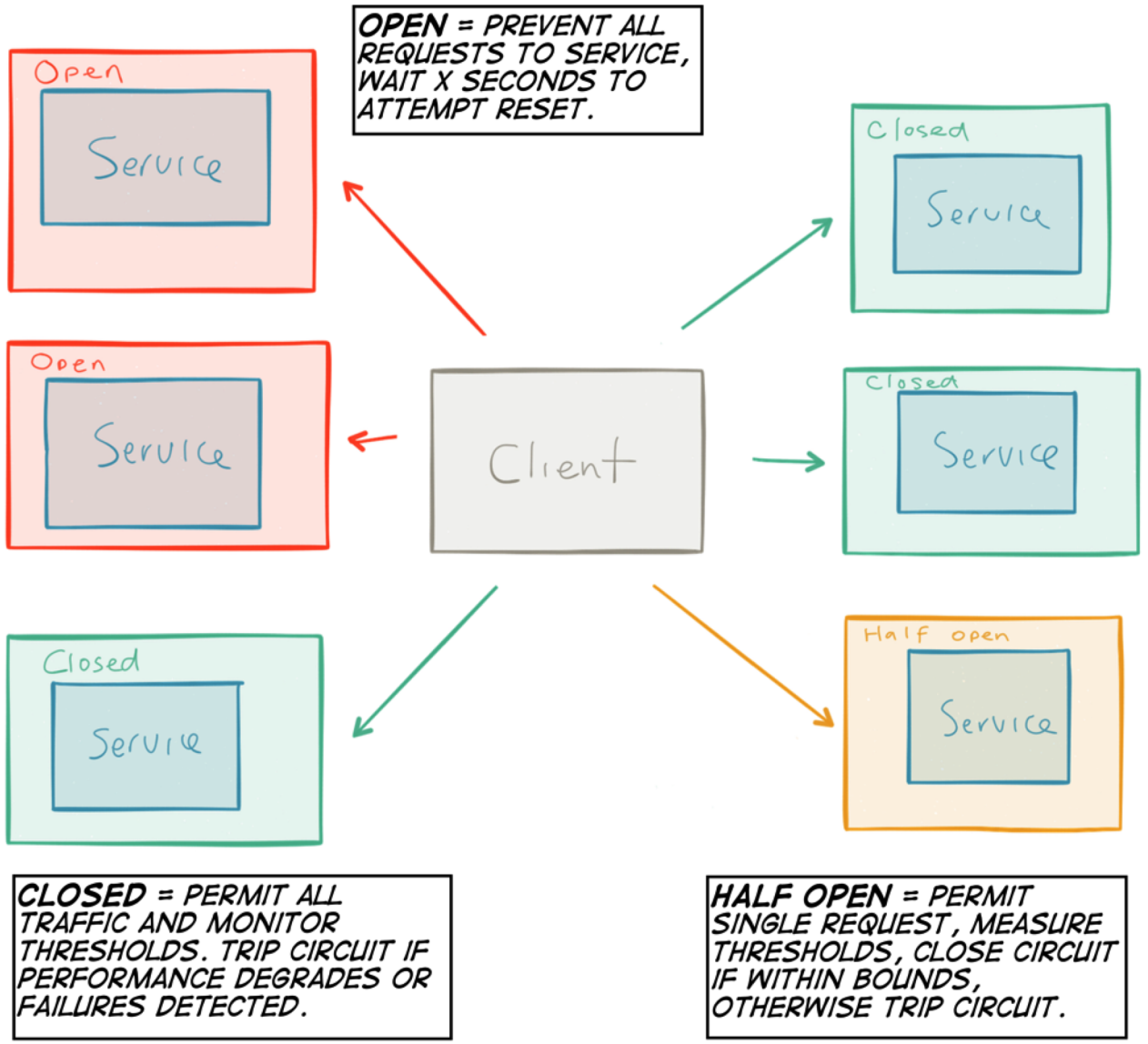


# 12 Dostupnost microservis - CIRCUIT BREAKER

Circuit breaker = **jistič**

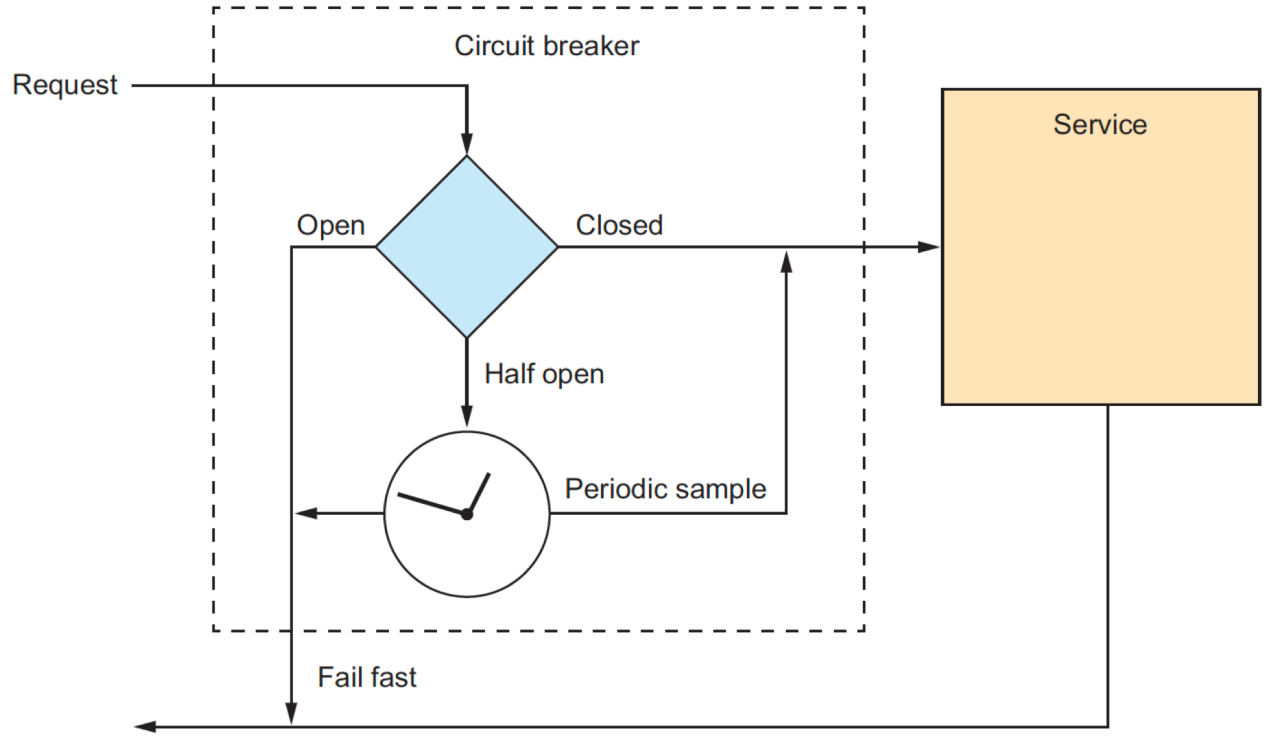
Wrapper obalující službu, který zajistí, že v případě, že je služba přetížená, tak na ni nechodí další požadavky a vrací uživateli okamžitou odpověď o nedostupnosti (uživatel nečeká až mu služba např. Až po minutě vrátí timeout)

*Circuit Breaker pracuje pomocí měření odezev obalované služby nebo přímo instrumentací, kdy dostává monitoring data např. o utilizaci CPU a paměti komponenty ve které běží služba*



# 12 Dostupnost microservis - CIRCUIT BREAKER

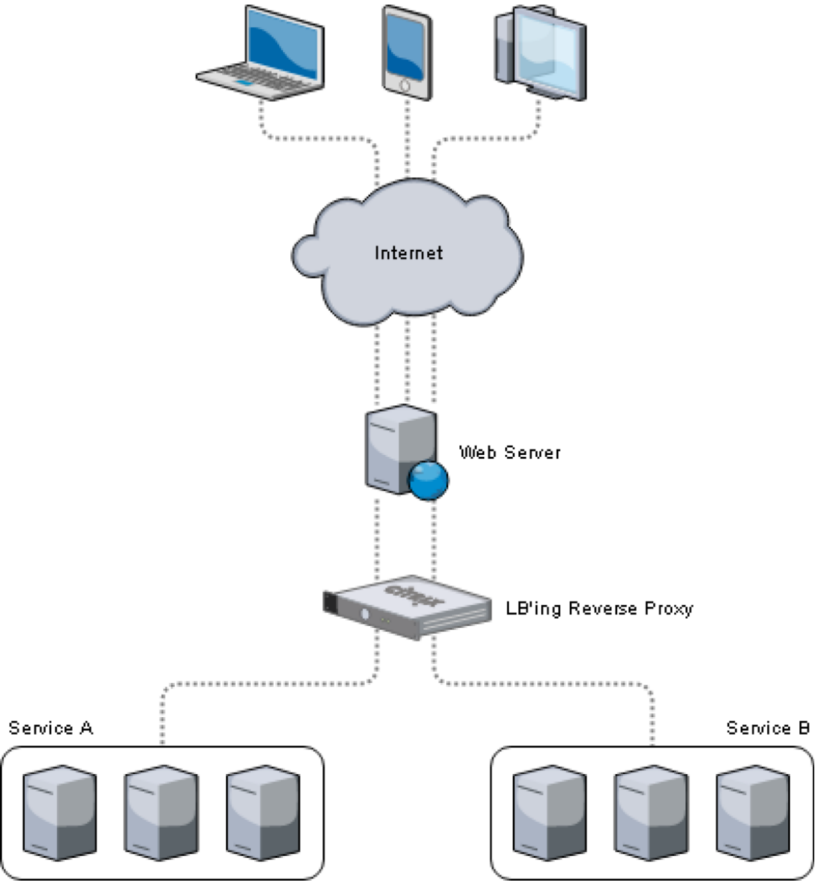
Speciální stav - napůl otevřený, kdy circuit breaker propouští jen omezený počet požadavků



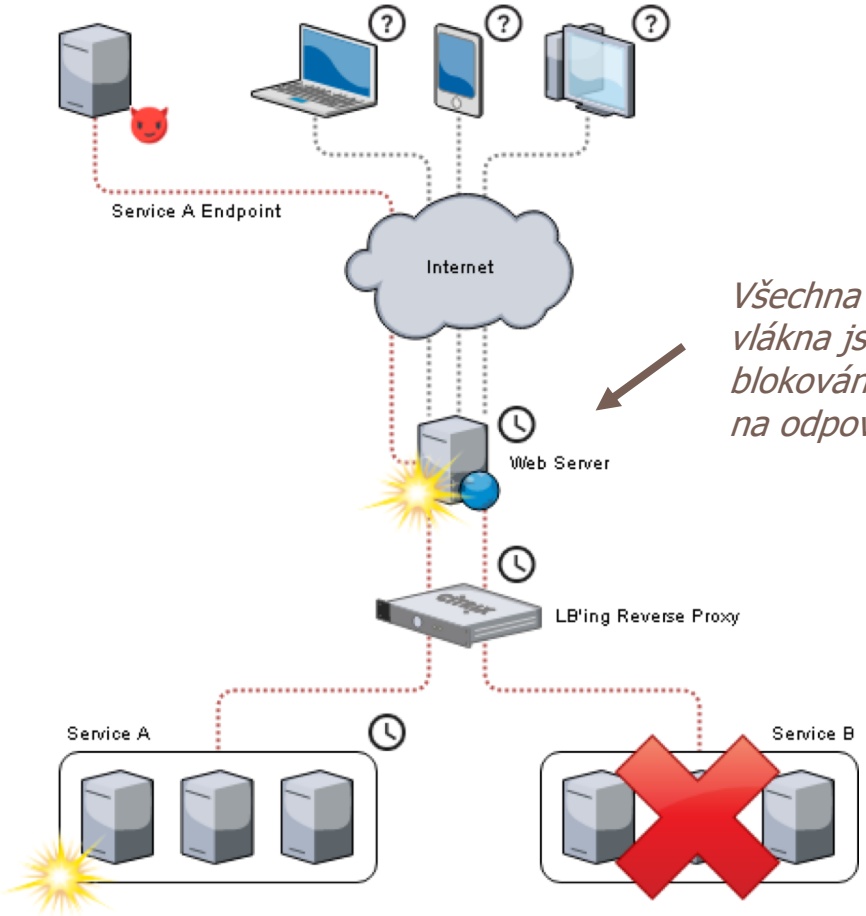

*Příkladem je např. Hystrix od Netflixu, který je embedovatelný ve Spring frameworku*
*Nebo Istio*

# 12 Dostupnost microservis – CIRCUIT BREAKER

1. Běžící systém

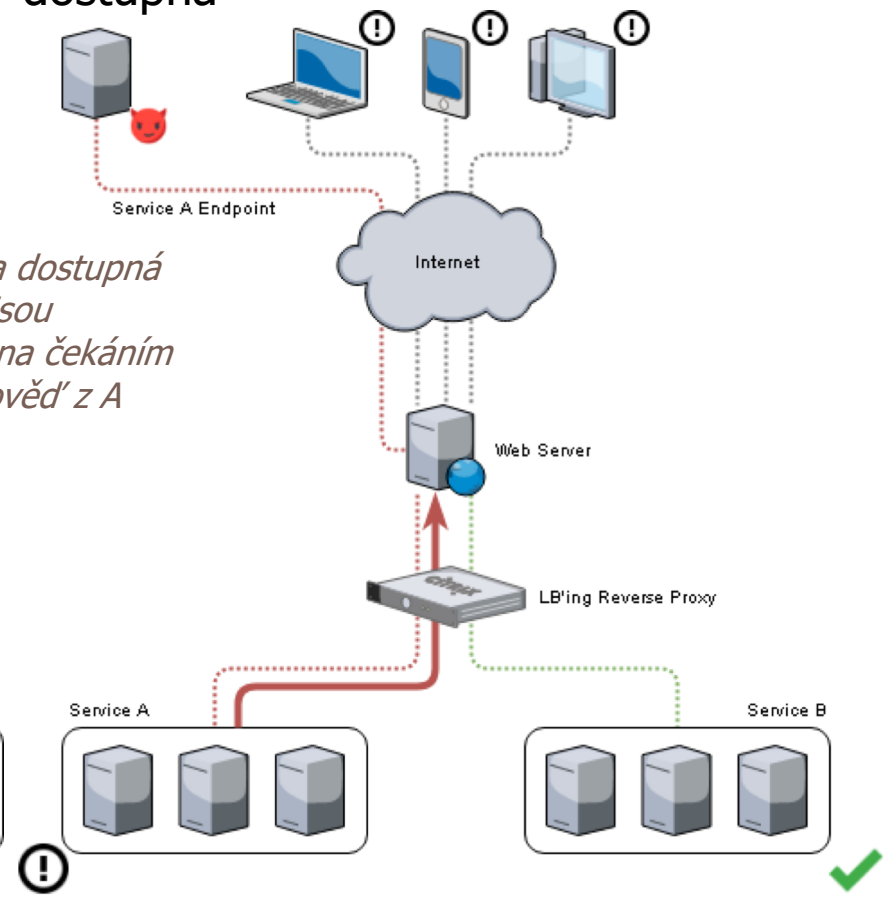



2. DoS útok




*Všechna dostupná vlákna jsou blokována čekáním na odpověď z A*

3. Zavřeli jsme možnost provolávání služby A => služba B je nadále dostupná



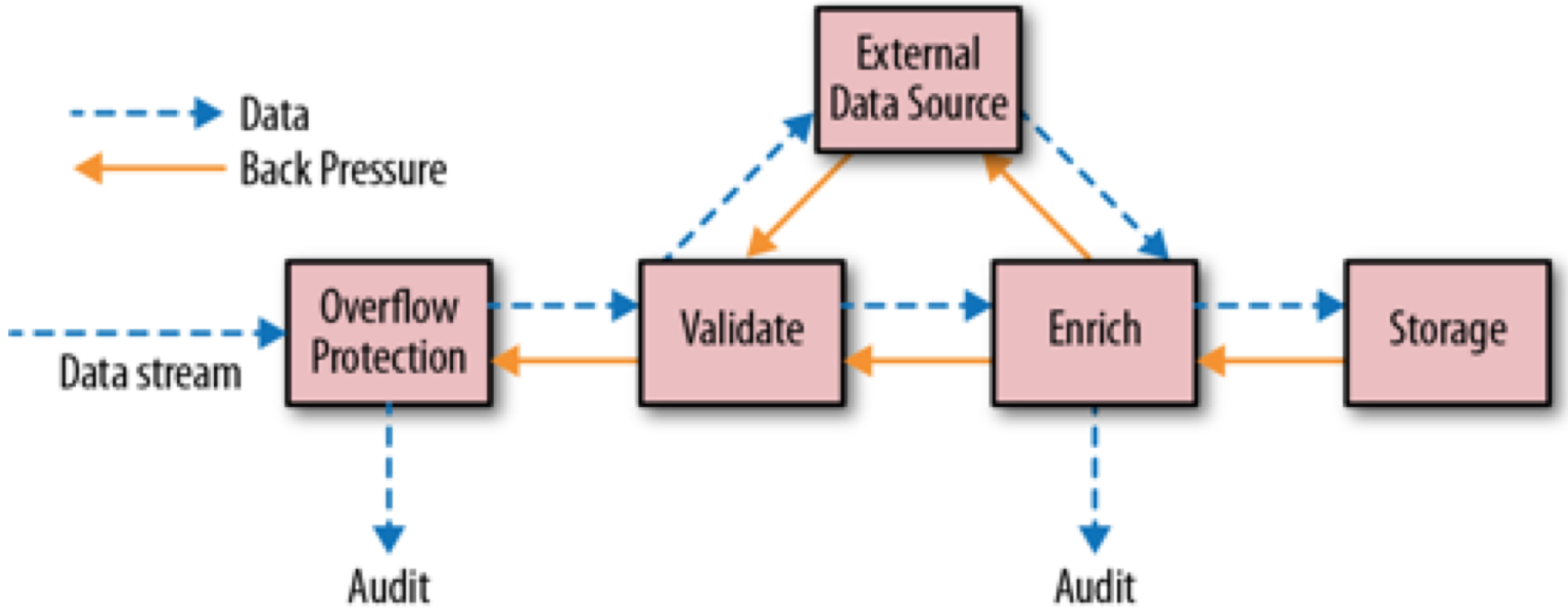
# 12 Dostupnost microservis - THROTLING

Mechanismus, kdy služby, které jsou volány mají nastavené bezpečnostní limity - např. 1000 volání za sekundu, 1GB za sekundu, 30000 volání za sekundu. Nastavení může být různé podle důležitosti služby.

Ostatní volání jsou zahozeny a klientovi se vrací chyba

# 12 Dostupnost microservis – BACK PRESSURE

Abychom zajistili co největší zdraví systému, tak se snažíme "přivřít kohout" co nejblíže zdroji loadu. Zavírání tedy propagujeme od komponenty, která nestíhá co nejblíže ke zdroji.



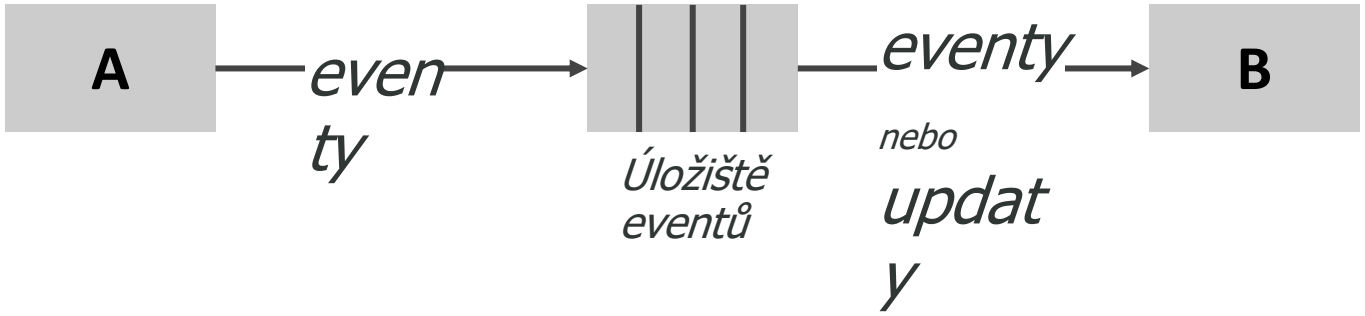
# Event sourcing

Klasický přístup při změně dat:

- první aplikace přímo updatuje stav druhé aplikace

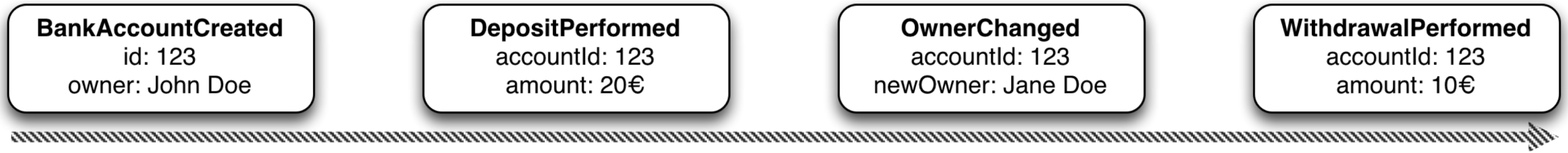

**Event Sourcing** je pattern při kterém:

- veškeré změny do stavu aplikace jsou ukládány jako eventy
- eventy jsou uloženy v sekvenci ve které byly provedeny
- mezi updatující aplikací a updatovanou aplikací je mezivrstva pro uložení eventů (databáze, fronta …)

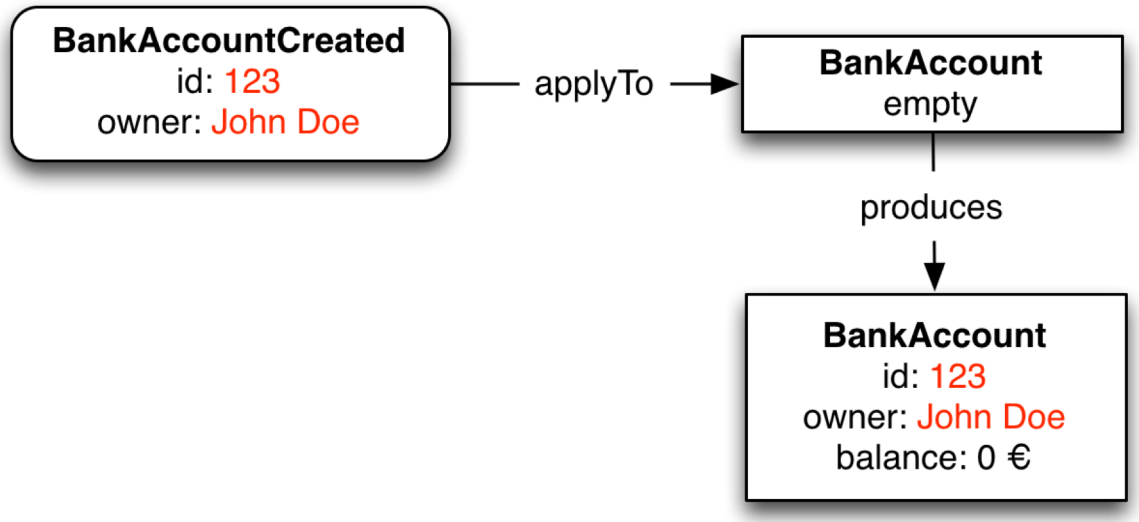


# Event sourcing

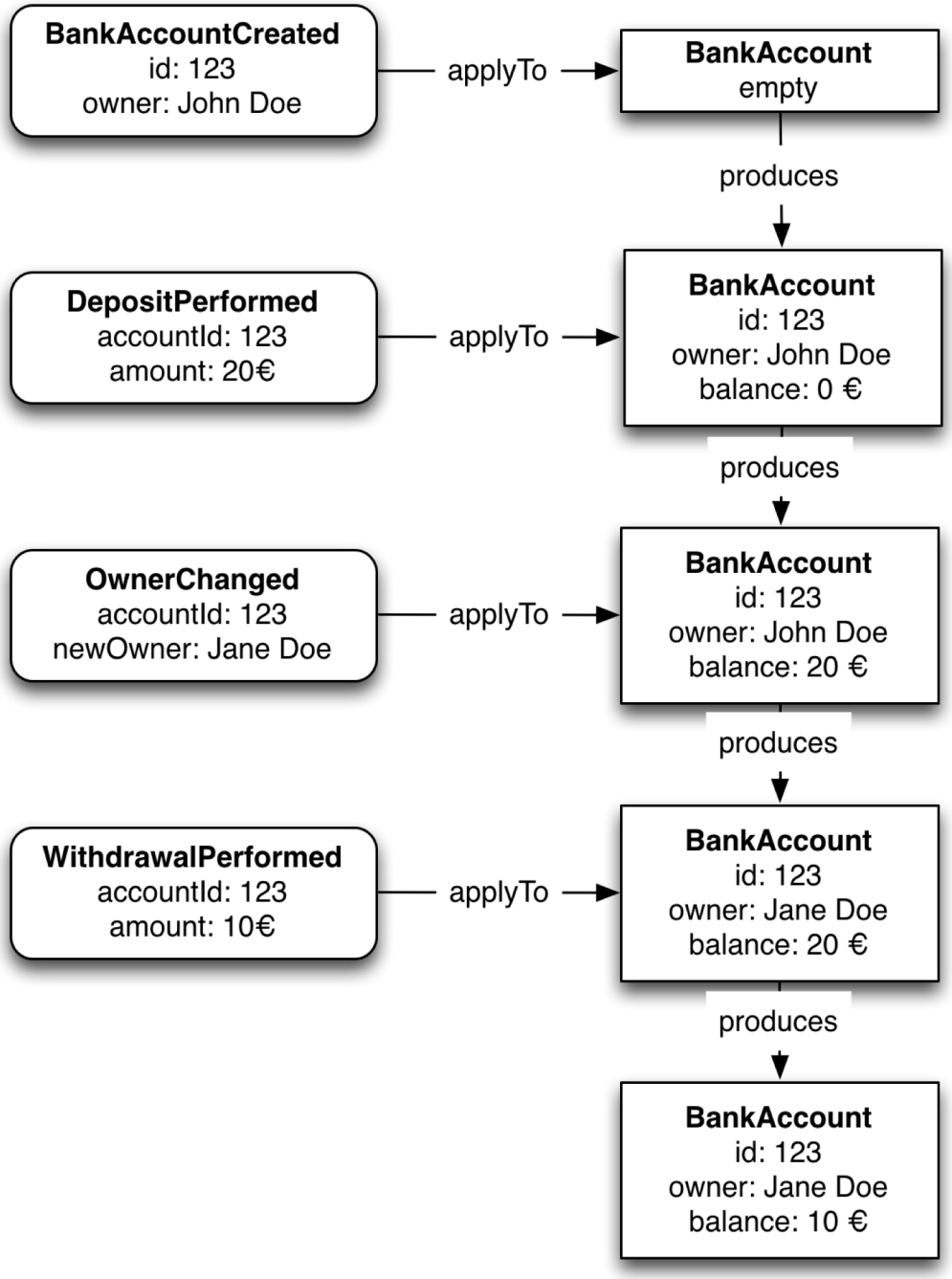
## Sekvence eventů >>>




## Aplikace eventu



# Event sourcing



# Výhody: sourcing

- ***Performance*** - aplikace zapisující změny není blokována aplikací do které se změny zapisují
- ***Rekonstrukce stavu aplikace*** - když bychom přišli o stav aplikace, tak ho jsme schopni zreplikovat od každého okamžiku novou aplikací eventů
- ***Vrácení se k jakémukoliv minulému stavu aplikace*** - vrátíme se jednoduše tím, že znovu aplikujeme event až do požadovaného okamžiku
- ***Rollback*** - když zjistíme, že chceme poslední updatu zrušit, tak provedeme inverzní operace k eventům

Pozor na:

- ***Side efekty*** - např. abychom neposílali klientovi 2x ten samý email => řešením je např. oddělení side efektů od stavových změn
- Úložiště eventů se např. ***nehodí na generování reportů a dotazování na data*** => pokud toto chceme, tak eventy aplikujeme do úložiště pro čtení a čteme data až z něj

# Materializovaný pohled (Materialized view)

Pro pochopení termínu Materializovaný pohled je třeba pochopit co je tzv. místo pravdy dat (**single source of truth**) - jediné místo v systému (nebo organizace), kde jsou 100% aktuální data

**Motivace**

- Data čteme výrazně častěji než zapisujeme
- Čtení dat je pomalejší než např. odezvy, které potřebujeme na uživatelském rozhraní.
- Dotaz který čte data je náročný na výkon

**Řešení**

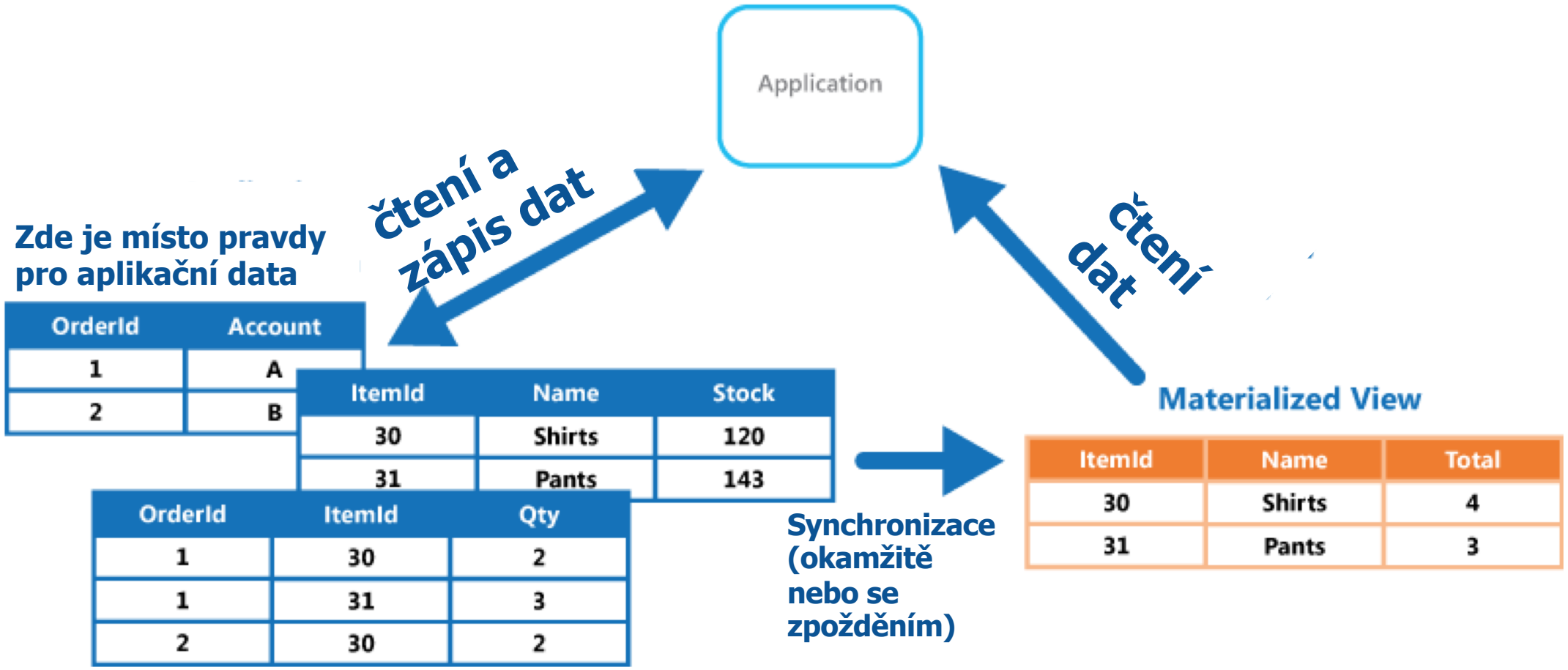
=> Z místa pravdy dat si dopředu vytvoříme pohled na data, který obsahuje pouze data které potřebujeme.

=> Data jsou předzpracována do modelu, ve kterém potřebujeme data číst.

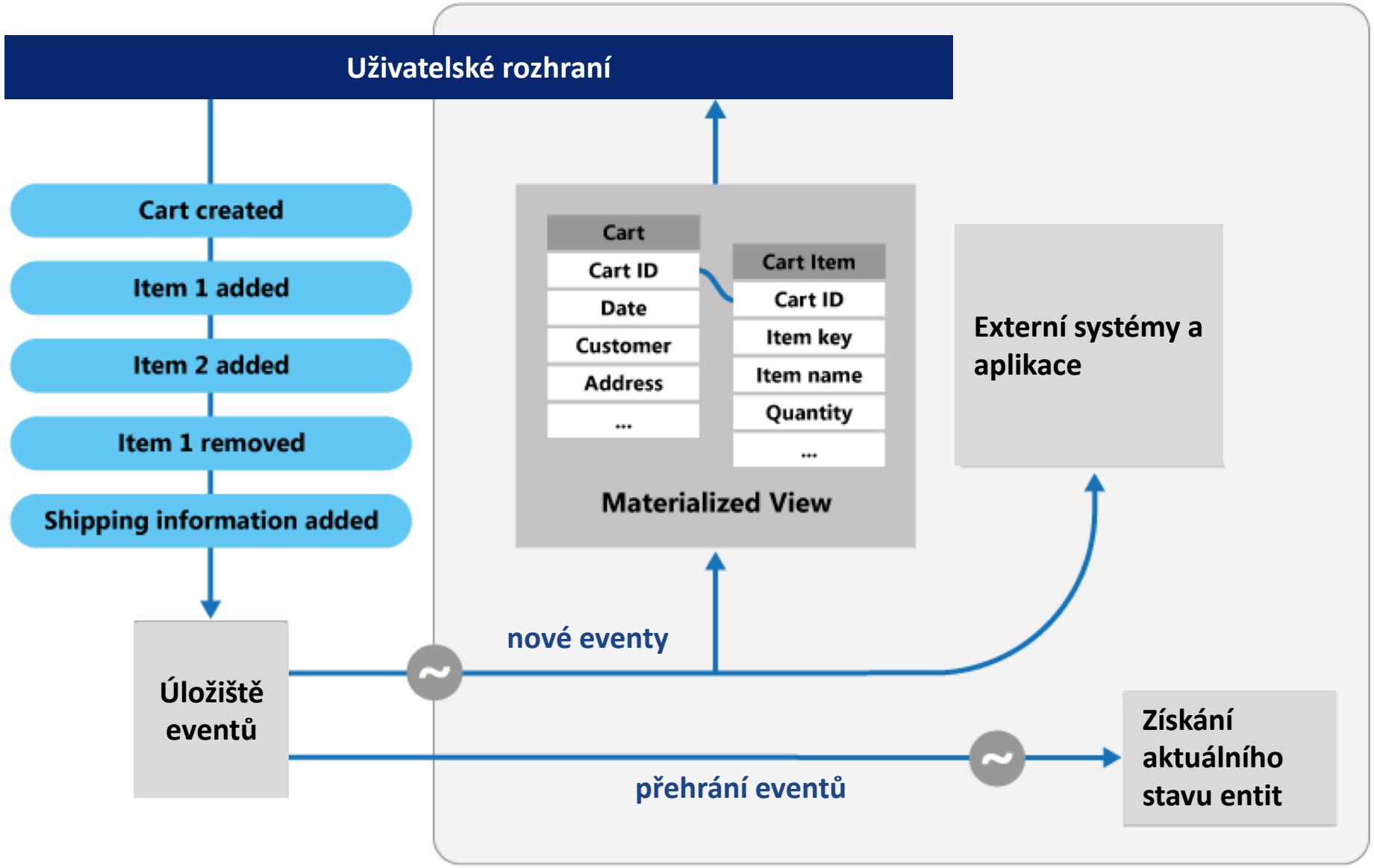
=> Tento pohled používáme pouze na čtení.

=> Do tohoto pohledu nikdy nezapisujeme přímo, měníme ho až po změně v hlavním místě pravdy - buď současně se zápisem do hlavního místa pravdy nebo se zpožděním (asynchronně nebo v dávce).

# Materializovaný pohled (Materialized view)



# Event sourcing a materializovaný View



# 12 CQRS – Command Query Responsibility Segregation

CQRS odděluje model pro zápis od modelu na čtení = **Command Query Responsibility Segregation**

Hlavní idea CQRS je:

Operace by měla buď změnit stav objektu nebo vrátit výsledek, ale ne obojí najednou - ***o**dpovídání na otázku nemění otázku*. Když provedeme takovou segregaci, tak budeme mít vždy dva typy operací:

- **Commands** - mění stav objektu nebo celého systému - tzv. ***mutátory***
- **Queries** - vrací výsledek a nemění stav objektu

Důvody pro CQRS:

- U složité aplikace vede spojení požadavků pro zápis a čtení do jednoho modelu k příliš komplikovanému modelu a komplikovaným dotazům
- Aplikace má zcela odlišné nefunkční požadavky na čtení a zápis - zápis a čtení se navzájem blokují a prodlužují odezvy

# 12 Jaký je rozdíl mezi eventem a commandem

***Event*** - co se stalo

***Command*** - co se má udělat

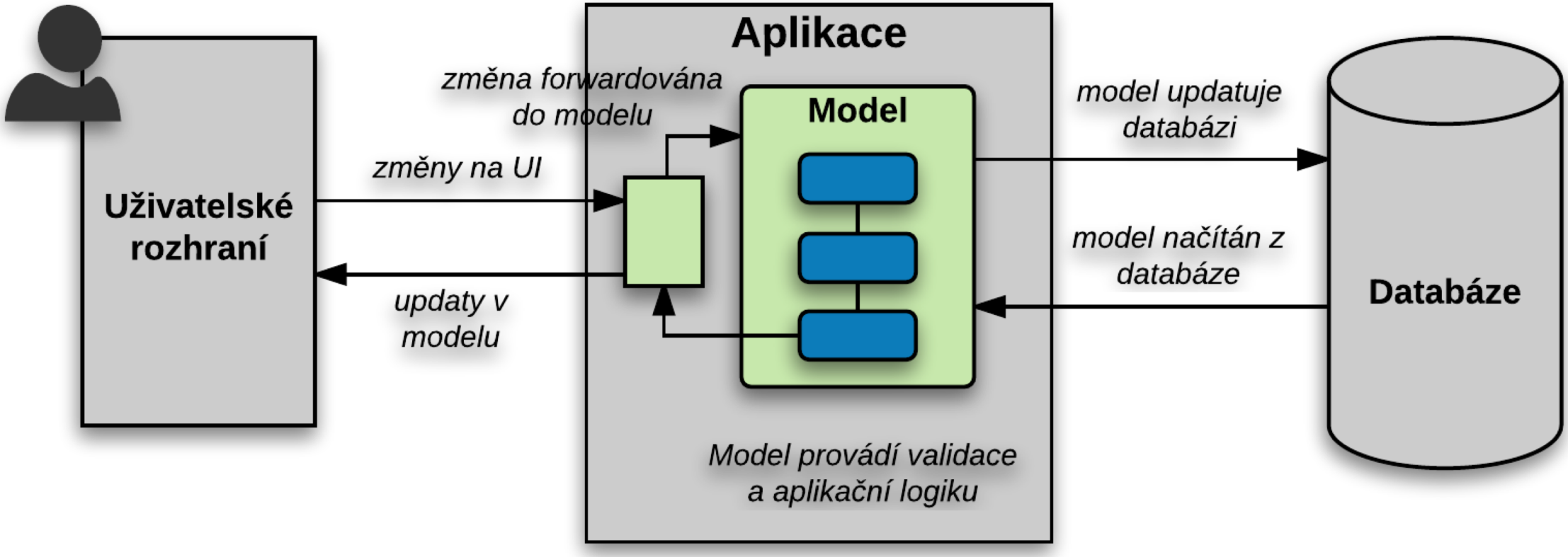
**Order system**

**Payment system**

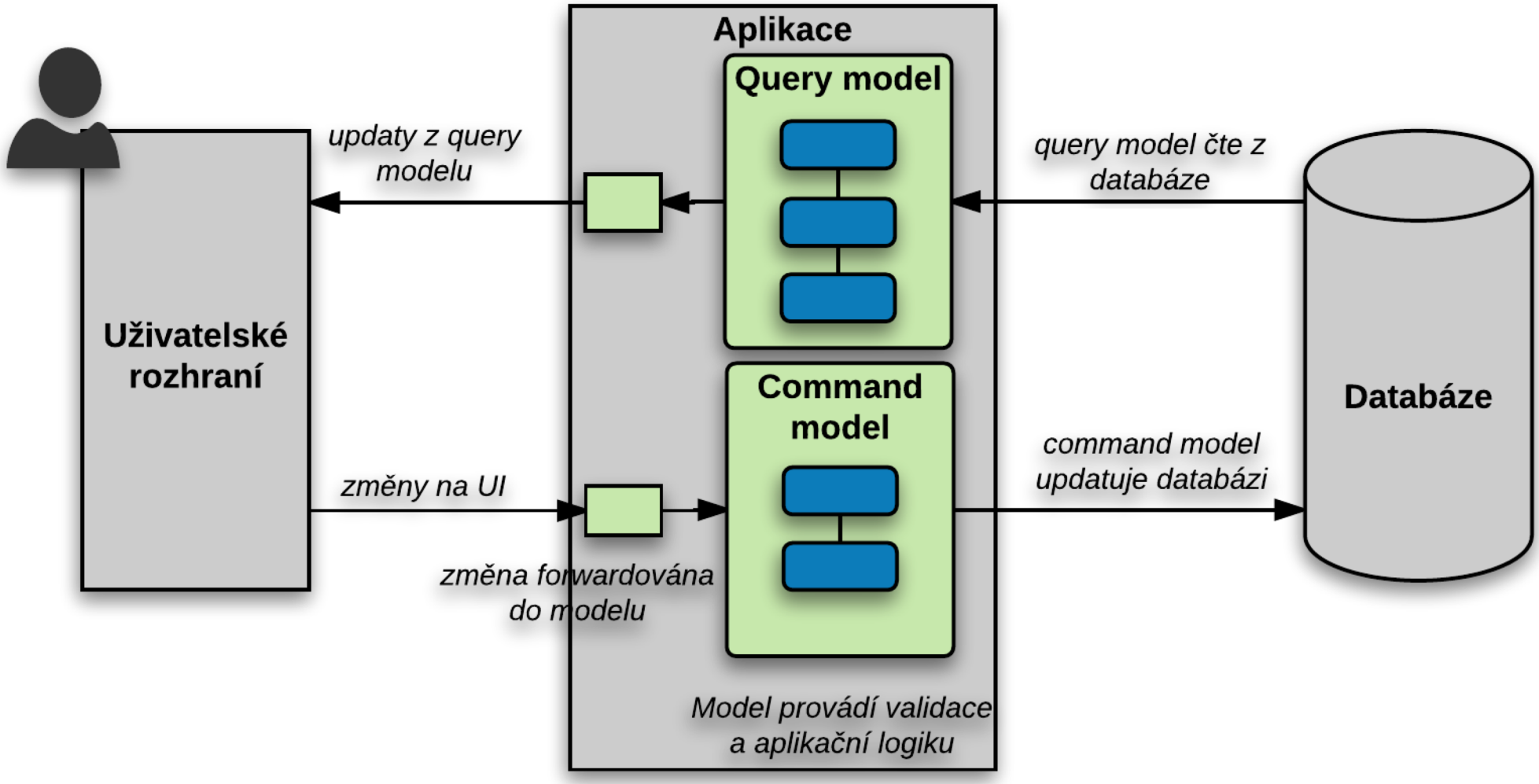
```
public class OrderPlacedEvent implements Event {
   UUID eventId;
   UUID srcId;
   List items;
   public OrderPlacedEvent(UUID SrcId,
List<Item> items){
       //set instance variables
   }
}
```

```
public class RetrievePaymentCommand implements Command{
   UUID commandId;
   String accountId;
   BigDecimal amount;
   public RetrievePaymentCommand(String accountId,
BigDecimal amount ){
       //set instance variables
   }
   public void execute(){
       //code to be executed
   }
}
```

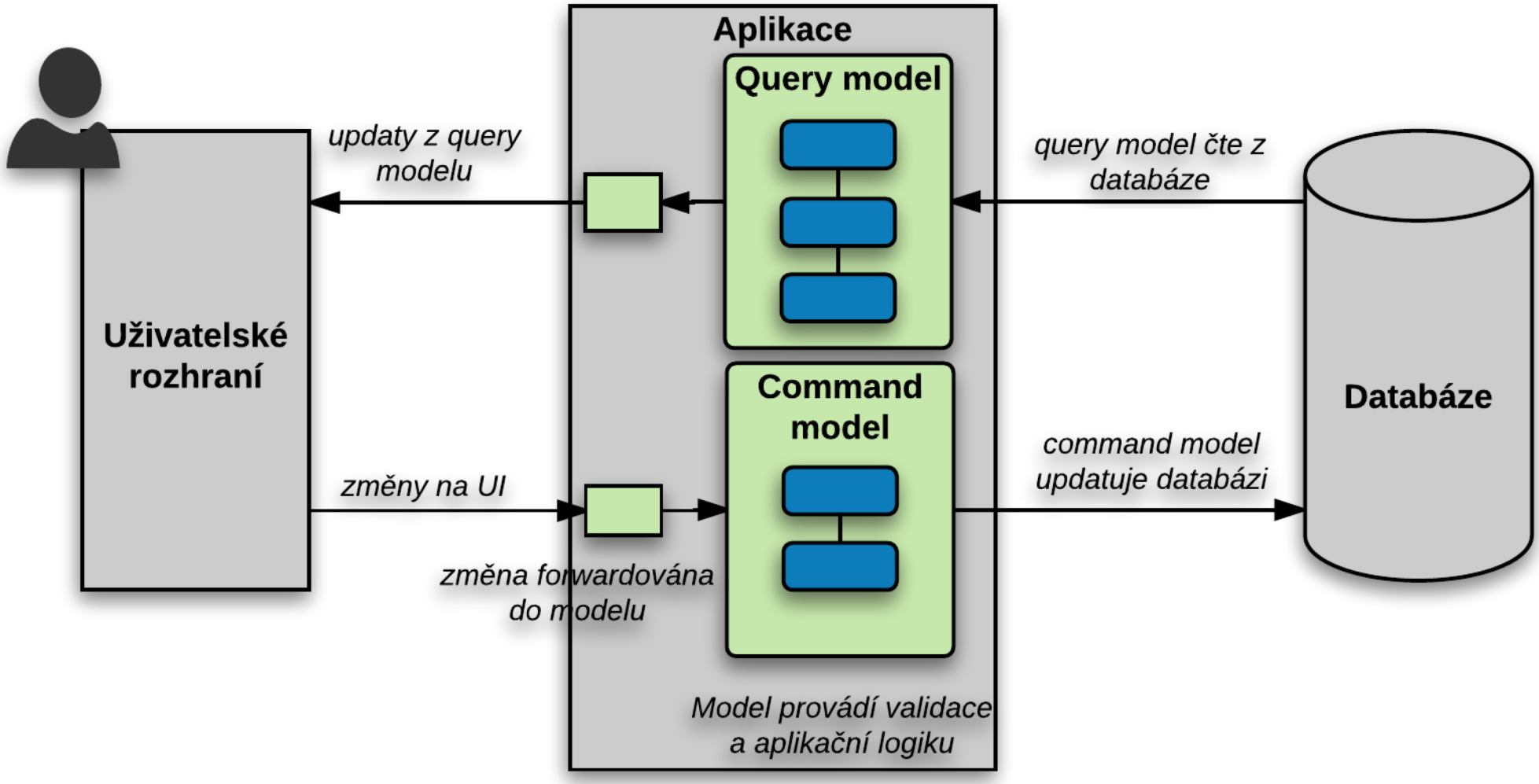
# 12 CQRS – Command Query Responsibility Segregation



# 12 CQRS – Command Query Responsibility Segregation

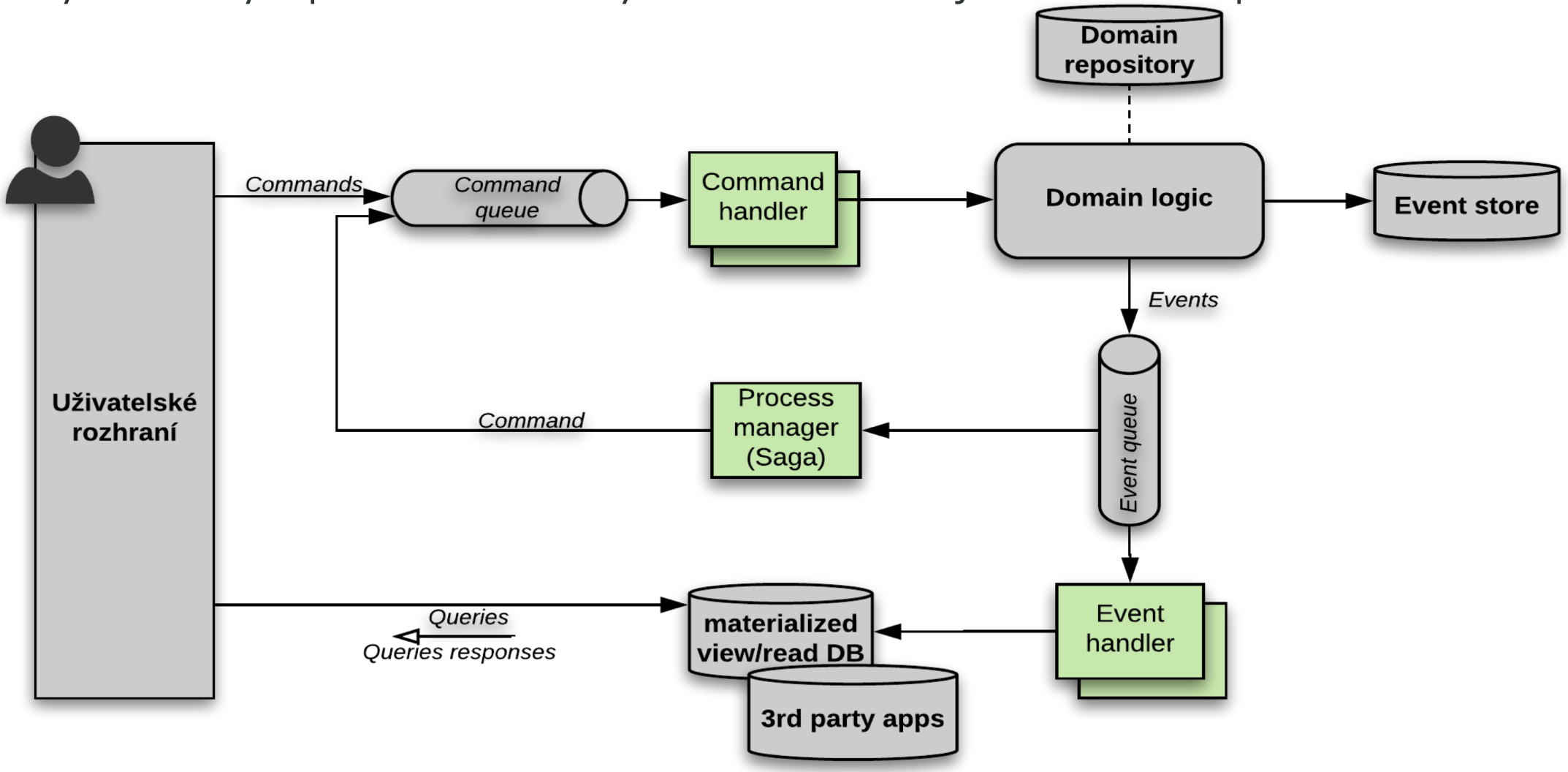


# 12 CQRS – Command Query Responsibility Segregation



# Materialized view + CQRS + Event sourcing + Saga

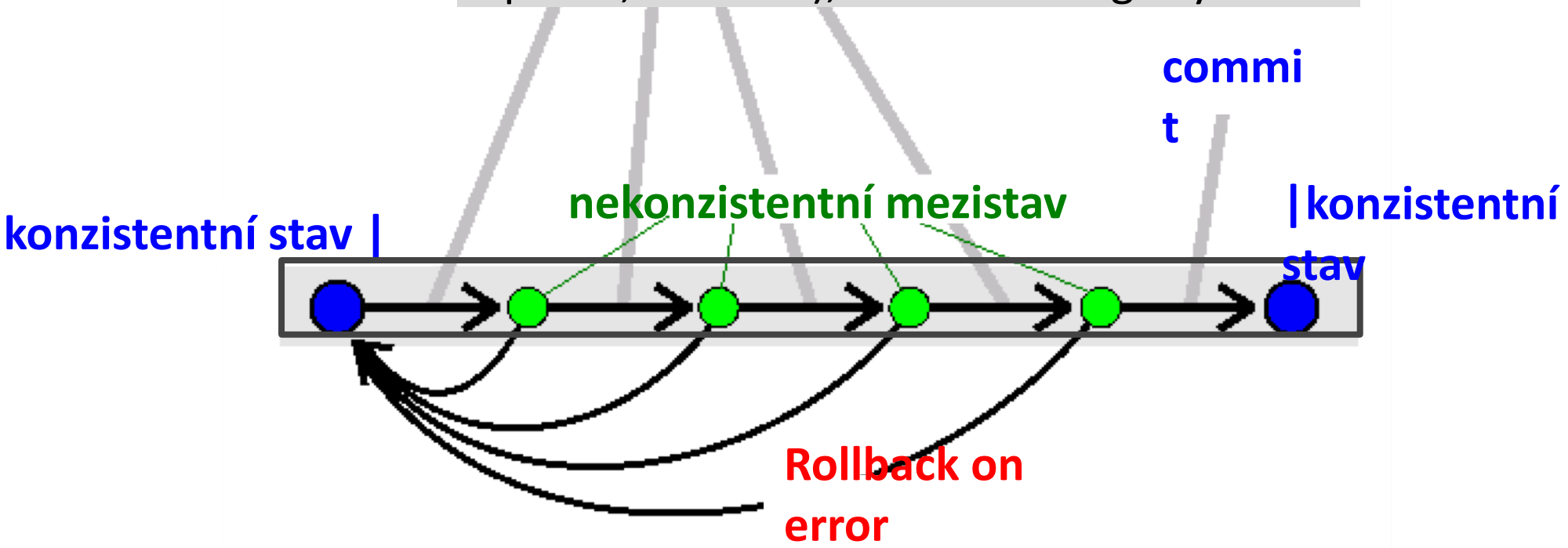
Po spojení výše uvedených patternů dohromady dostáváme následující architekturu aplikace



BACKUP SLIDY – Pro lepší pochopení, nebude u zkoušky

# Relační databáze a ACID

**Transakce** = sekvence operací (insert, update, delete...), které tvoří logický celek




**ACID**

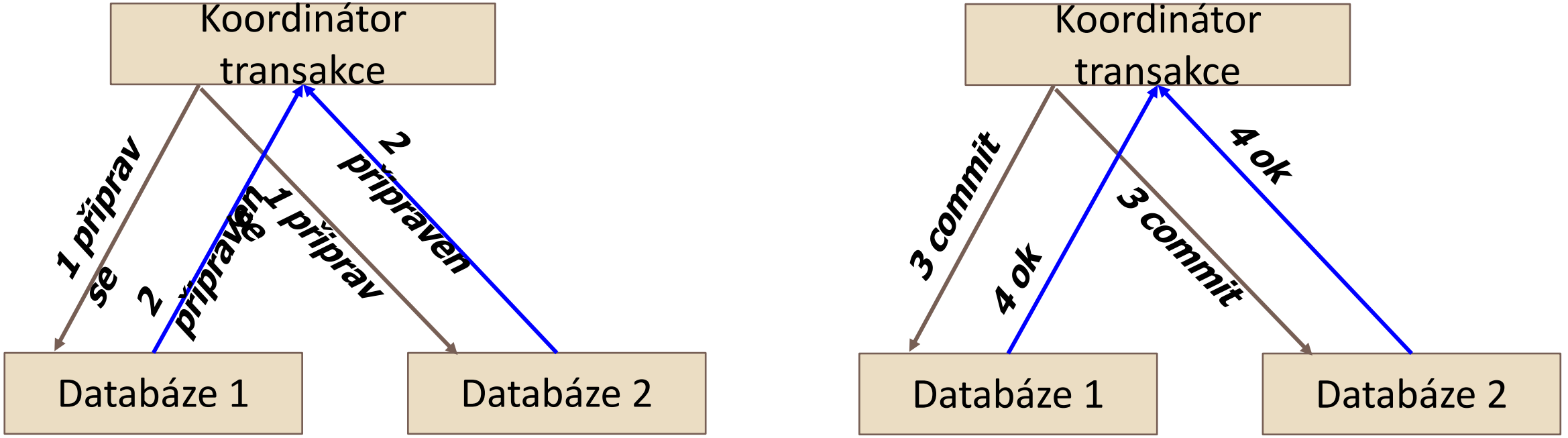
**A**tomicity - změny prováděné v transakci jsou buď realizovány jako celek nebo vůbec

**C**onsistency - databáze je konzistentní před a po transakci

**I**solation - během transakce jsou data se kterými pracuje izolovány od dalších transakcí

# Relační databáze a dvoufázový commit (2PC)

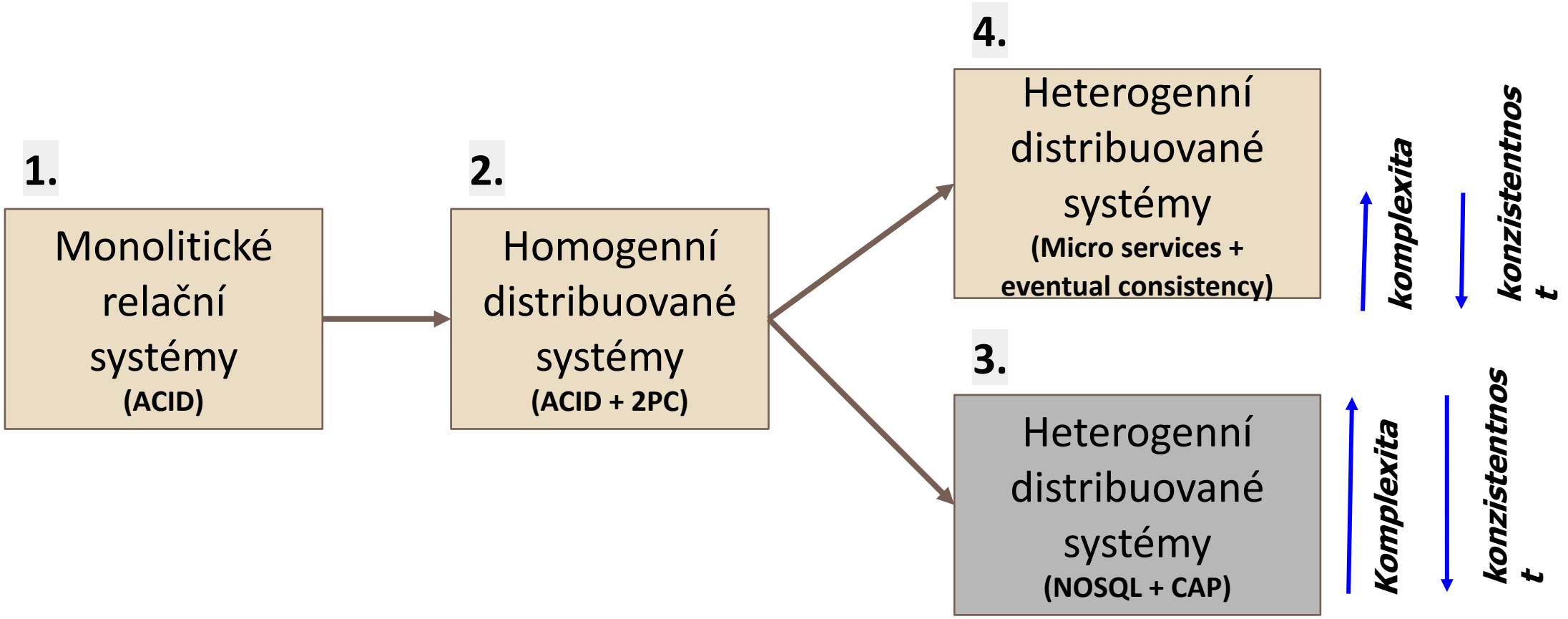
V případě, že všichni účastníci distribuované transakce potvrdí připravenost realizovat operace, tak se provádí *commit* na všech těchto systémech, jinak se neprovede na žádném




Proč 2PC není vždy použitelný:

- Existující aplikace většinou nepodporují transakce a dvoufázový commit - není to tzv. XA resource
- Koordinátor transakce zavádí *single point of failure* (při jeho výpadku nejde zapsat nikam)
- Protokol pro realizaci 2PC je velmi "upovídaný"
- Omezená propustnost, protože kvůli konzistentnosti musíme zamykat tabulky a serializovat operace

# Geneze architektury



# 12 DOMÉNY

*each module of our system should have only one reason to change.*

*keep our domain completely pure and without side effects, so we can easily test it and safely reuse its functions everywhere we need.*

*An interesting question is how to distinguish what is business logic from what is not? The main difference is at language level: business logic is something you can discuss with the business people, without using any technical term like for example serialization formats, network protocols etc. - tedy pokud není cíle vyrobit technický produkt jako např. cloud*

Rozhraní, které obsahuje přesně jednu abstraktní metodu, se nazývá *Functional Interface*. Může mít libovolný počet výchozích statických metod, ale může obsahovat pouze jednu abstraktní metodu. Může také deklarovat metody třídy objektů.

Funkční rozhraní je také známé jako Single Abstract Method Interfaces nebo SAM Interfaces.