Технології створення Web-застосунків

Модуль 3.

Доц. Попівщий В.І. ЗНУ каф.ПЗАС 2020

ОСНОВНІ ДЖЕРЕЛА

1. Хорсдал К. Микросервисы на платформе .NET. – СПб. : Питер, 2018. 352 с.
2. Ньюмен С. Создание микросервисов. – СПб.: Питер, 2016. – 304 с.
3. Фаулер М. Шаблоны корпоративных приложений. – М.: Вильямс, 2016. – 544 с.
4. Ричардсон К. Микросервисы. Паттерны разработки и рефакторинга. – СПб.: Питер, 2019. 544 с.
5. Cole Matt R. Hands-On Microservices with C#. – Packt Publishing, 2018. – 234 p.
6. Newman Sam Monolith to Microservices. – O’Reilly Media, 2020. – 256 p.
7. Gaurav Aroraa, Ed Price Hands-On Microservices with C# 8 and .NET Core 3 Third Edition. – Packt Publishing, 2020. – 451 p.

8. Morgan Bruce, Paulo A. Pereira **Microservices in Action**. – Manning Publications, 2019. – 366 p.
9. Essentials of Microservices Architecture. Paradigms, Applications, and Techniques. - CRC Press: 2020. - 293 p.
10. Tayo Koleoso Beginning Quarkus Framework: Build Cloud-Native Enterprise Java Applications and Microservices. – Apress, 2020. – 297p. https://scanlibs.com/beginning-quarkus-framework-cloud-native-apps/
11. Binildas Christudas Practical Microservices Architectural Patterns. – Apress, 2019. – 902 p.

ТРЕНІНГИ

- Тренінг «Мікросервіси для Java розробників» https://itcluster.lviv.ua/paged/training-microservices-for-java-developers/
- Курси Microservices https://www.nobleprog.com.ua/kursy-microservices

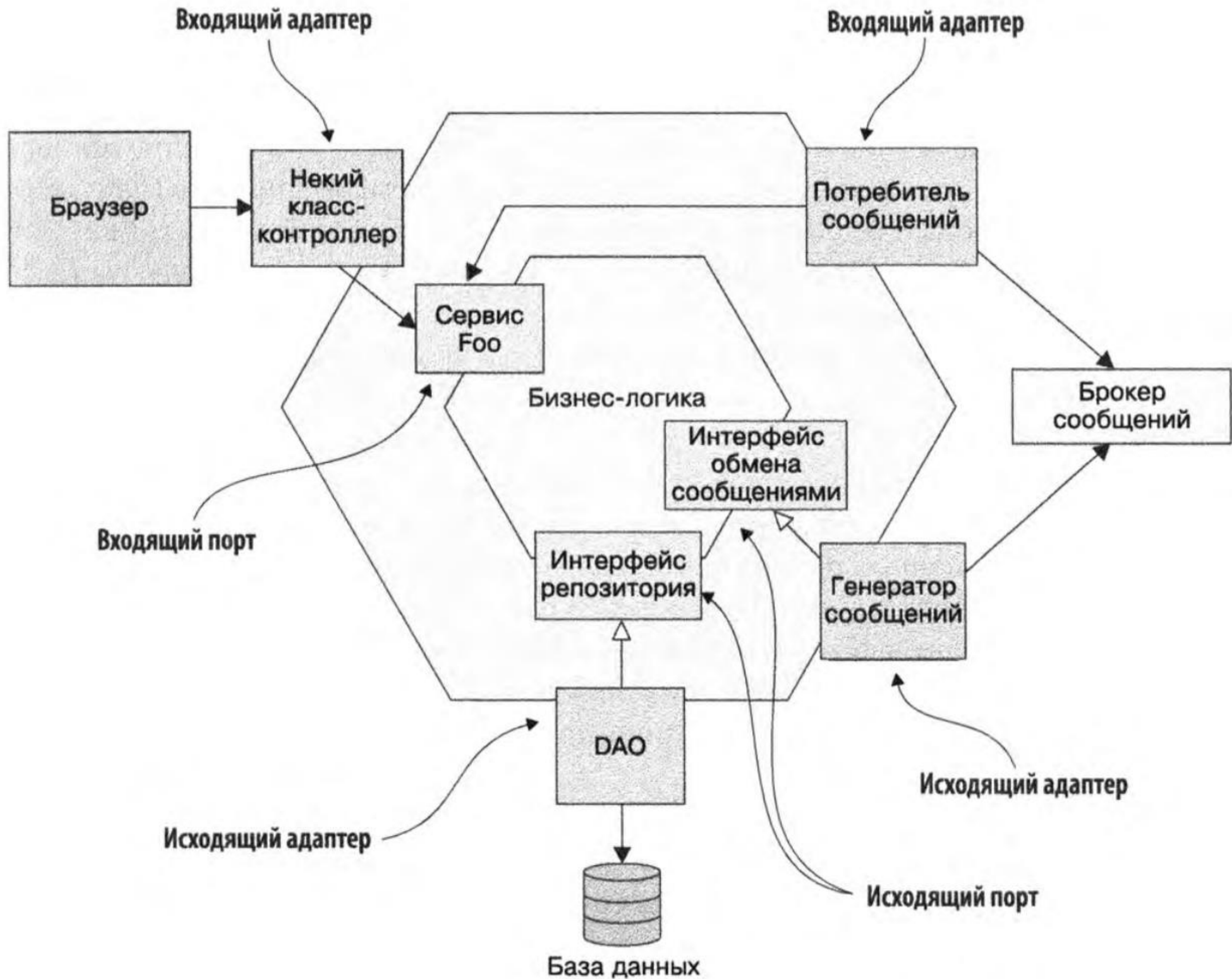
Огляд відеокурсів

1. Linkedin - Learning Creating Your First Spring Boot Microservice 2019
2. **Lynda - Azure Microservices with .NET Core for Developers 2020**
3. Pluralsight - Building Microservices (2019)
4. Pluralsight – Microservices The Big Picture (2018)
5. ASP.NET Core 2.0 E-commerce Web Site Based on Microservices (2019)
6. Lynda - Microservices Design Patterns (2020)
7. Packt – A Beginner's Guide to a Microservices Architecture
8. O'Reilly - Building Microservice Systems with Docker and Kubernetes

Модуль 3. Проектування бізнес-логіки в мікросервісах.

Шестигранна архітектура

- Шестигранна архітектура - це альтернатива багаторівневому стилю проектування.
- Вона організовує логічне уявлення таким чином, що бізнеслогіка виявляється в центрі
- Замість рівня уявлення у додатка є один або кілька **вхідних адаптерів**, які обробляють зовнішні запити шляхом виклику бізнес-логіки.
- Аналогічно замість рівня зберігання даних використовуються один або кілька **вихідних адаптерів**, які викликаються бізнес-логікою і звертаються до зовнішніх додатків.


Входящий адаптер
Входящий адаптер
Браузер
Некий класс-контроллер
Потребитель сообщений
Сервис Foo
Бизнес-логика
Брокер сообщений
Интерфейс обмена сообщениями
Входящий порт
Интерфейс репозитория
Генератор сообщений
DAO
Исходящий адаптер
Исходящий адаптер
Исходящий порт
База данных

- Ключовою характеристикою і перевагою даної архітектури є те, що бізнес-логіка не залежить від адаптерів.
- Все навпаки: адаптери залежать від неї.
- У бізнес-логіки є один або кілька портів. Порт визначає набір операцій і то, як і в чому бізнес-логіка взаємодіє із зовнішнім кодом.
- В Java, наприклад, порт часто є Java-інтерфейсом.
- Існує два види портів: вхідні та вихідні.
- Вхідний порт - це API, який виставляється назовні бізнес-логікою і доступний для виклику зовнішніми додатками. Як приклад вхідного порту можна привести інтерфейс сервісу, який описує його публічні методи.
- Вихідний порт - це те, як бізнес-логіка звертається до зовнішніх систем. Прикладом може служити інтерфейс сховища, яке визначає набір операцій для доступу до даних

- Навколо бізнес-логіки розміщуються адаптери.
- Як і порти, вони бувають двох типів: вхідні та вихідні.
- Вхідний адаптер обробляє запити з зовнішнього світу, звертаючись до вхідного порту. Прикладом вхідного адаптера може служити контролер Spring MVC, який реалізує або набір кінцевих точок формату REST, або колекцію веб-сторінок.
- Вихідний адаптер реалізує вихідний порт і обробляє запити бізнес-логіки, звертаючись до зовнішнього додатка або сервісу.
- Як приклад вихідного адаптера можна привести клас об'єкта доступу до даних (data access object, DAO), який реалізує операції для роботи з базою даних.

Проектування бізнес-логіки в мікросервісній архітектурі

- Серцем промислових додатків є бізнес-логіка, яка реалізує бізнес-правила.
- Розробка складної бізнес-логіки завжди пов'язана з певними труднощами.
- У мікросервісній архітектурі розробляти складну бізнес-логіку виявляється ще важче, тому що вона розподілена між різними мікросервісами.

Дві ключові проблеми

- Вам необхідно вирішити дві ключові проблеми.
- Типова доменна модель виглядає як павутина з пов'язаних між собою класів. У монолітних додатках в цьому немає нічого поганого, але в мікросервісній архітектурі, де класи розкидані по різних сервісів, потрібно **позбутися від посилань на об'єкти**, які перетинають кордони сервісів.
- Ще одна проблема полягає в проектуванні бізнес-логіки, яка працює в рамках обмежень, що накладаються **роботою з транзакціями** в мікросервісній архітектурі.

Подолання труднощів

- На щастя, для подолання цих труднощів можна скористатися шаблоном «Агрегат» зі складу DDD.
- Він структурує бізнес-логіку програми у вигляді набору агрегатів.
- Агрегат - це кластер об'єктів, до яких можна звертатися як до одного цілого.

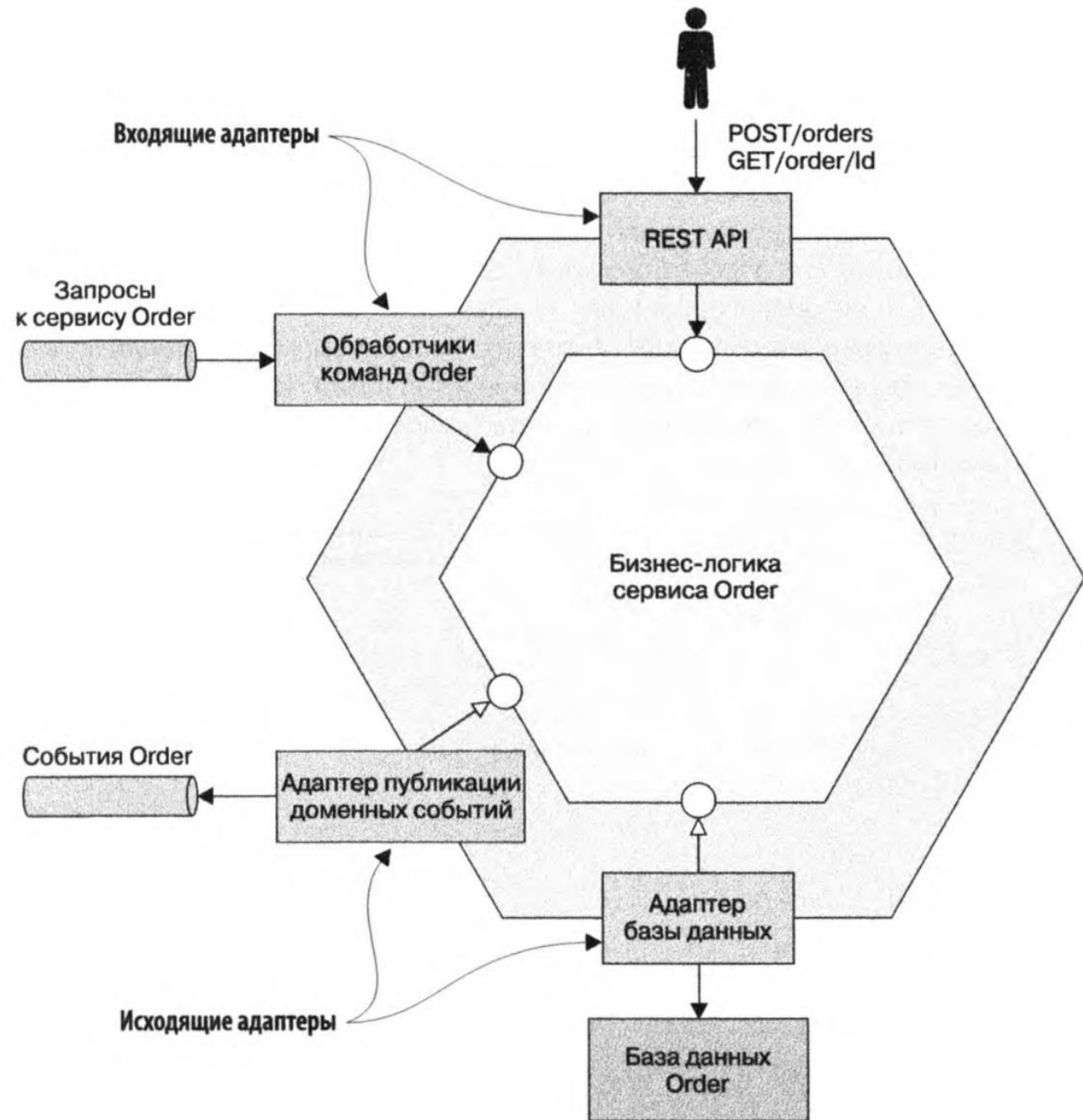
Агрегати

Є дві причини, чому агрегати можуть стати в нагоді при розробці бізнес-логіки в мікросервісній архітектурі.

- Агрегати виключають будь-яку можливість того, що посилання на об'єкти можуть вийти за рамки одного сервісу, тому що міжагрегатне посилання - це скоріше значення первинного ключа, а не об'єктне посилання.
- Транзакція може створити або оновити лише один агрегат, тому агрегати відповідають обмеженням транзакційної моделі мікросервісів.

Шаблони організації бізнес-логіки

- На наступному слайді показана архітектура типового сервісу.
- Як говорилося раніше, бізнес-логіка є ядром шестигранної архітектури.
- Її оточують вхідні та вихідні адаптери.
- Вхідний адаптер обробляє запити від клієнтів і викликає бізнес-логіку.
- Вихідний адаптер, який сам викликається бізнес-логікою, звертається до інших сервісів і додатків.


Входящие адаптеры
POST/orders
GET/order/Id
REST API
Запросы
к сервису Order
Обработчики
команд Order
Бизнес-логика
сервиса Order
События Order
Адаптер публикации
доменных событий
Адаптер
базы данных
Исходящие адаптеры
База данных
Order

Шаблони для Microservices-орієнтованих додатків

(Essentials of Microservices Architecture. Paradigms, Applications, and Techniques. - CRC Press: 2020, p.221-250)

- Архітектура мікросервісів вважається найефективнішим архітектурним зразком і стилем для створення та підтримки програм **корпоративного рівня**, які вимагають частого розгортання та постійної доставки.
- Бізнес-додатки та IT-послуги будуються з використанням унікальних можливостей MSA.
- Не тільки нові програми, але й існуючі та застарілі програми ретельно переробляються та виправляються як програми, орієнтовані на мікросервіси.

- Таким чином, MSA проголошується підходом до прориву та навіть **проривом для програмного забезпечення наступного покоління**, яке підтримується разом із апаратною інженерією.
- Давні цілі
  - **гнучкості** (agility)**,**
  - **надійності** (reliability)**,**
  - **масштабованості** (scalability)**,**
  - **доступності** (availability, accessibility)**,**
  - **стійкості** (sustainability) **та**
  - **безпеки** (security)

програмного забезпечення можна легко досягти за допомогою розумного використання архітектури мікросервісів.

В цьому розділі ми розглянемо детальний опис всіх видів корисних та придатних для використання шаблонів (patterns), спрямованих на мікросервісне

- **проектування**,
- **розробку**,
- **складання** та
- **розгортання**

додатків.

Макрорівневий погляд на архітектуру мікросервісів

- Для простоти та заохочення безризикового використання архітектури мікросервісів рекомендується перевірений часом рівневий (layered) підхід.
- Експерти запропонували різні типи сервісів, які можуть бути розміщені у декількох різних рівнях. У поданні на макрорівні можна реалізувати шари різних типів мікросервісів (https://medium.com/@kasunindrasiri/microservices-apis-and-integration-7661448e8a86), як показано на малюнку на наступному слайді

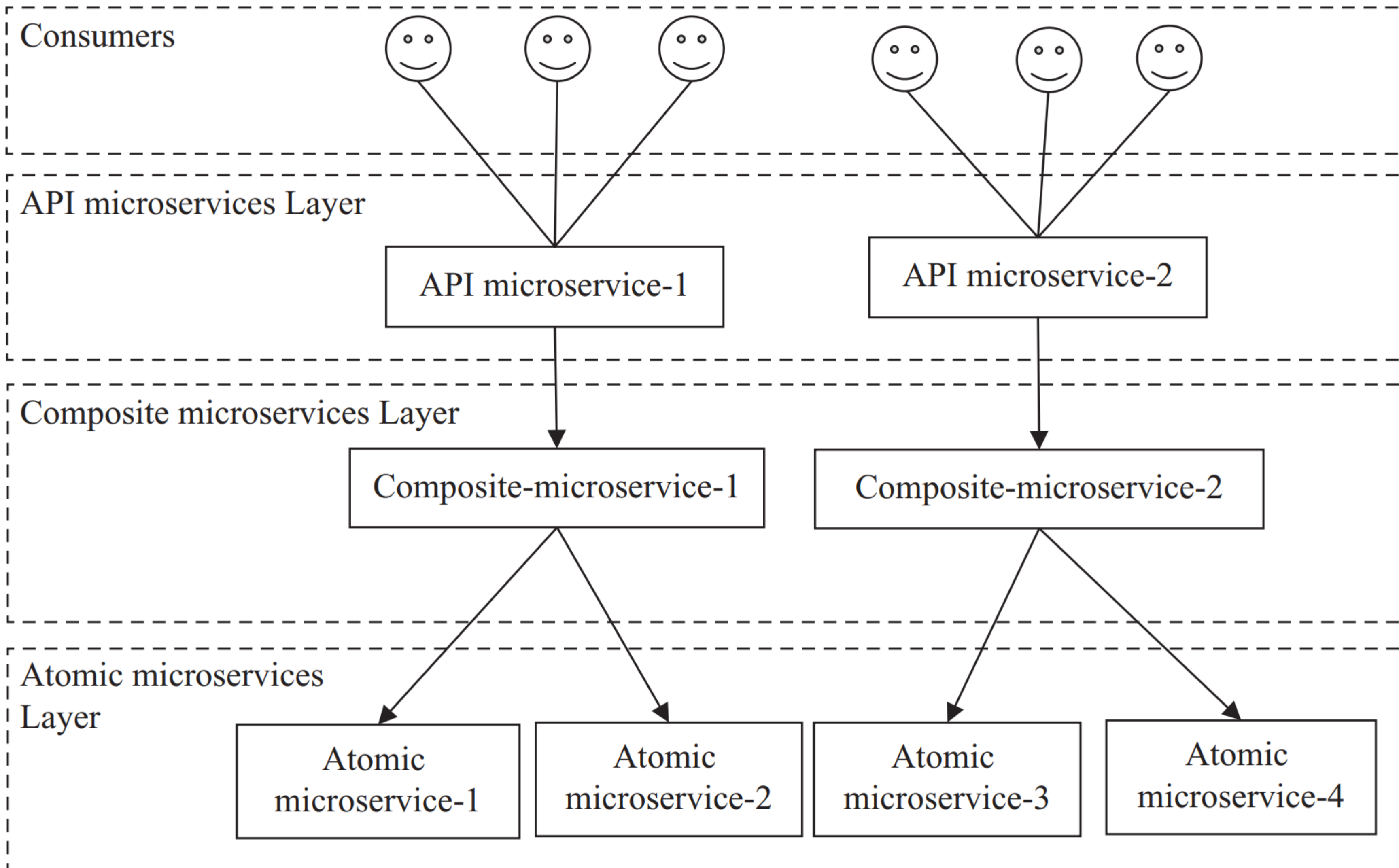


Figure 8.1 Macro-level view of Microservices Architecture.

- **Atomic/core microservices layer**— Ці служби нижнього шару зазвичай виконують ділову логіку додатків. Більшість існуючих технологій, таких як Spring Boot, Dropwizard, ASP.NET Core тощо, зосереджені на створенні цих послуг.
- **Composite microservices layer**— Оскільки атомарні послуги пропонують обмежену функціональність, вони поєднуються *для забезпечення корисних бізнес-процесів*. Отже, composite послуги лежать вище атомарних послуг. Ці композиційні послуги формуються або за допомогою **оркестровки**, або за допомогою **хореографії**. Ці послуги є грубозернистими. Поряд із складеними послугами *для ділової логіки* існують також складені служби, які виконують ряд послуг, таких як *безпека, надійність, масштабованість, підтримка транзакцій, стійкість та стабільність* тощо. Різні технології, такі як Ballerina - мова програмування, призначена для обслуговування композиції та мережевої взаємодії та service mesh frameworks, які можуть розвантажувати мережеві комунікації, використовуються для розробки композитних послуг

- **API/edge microservices layer**— Рівень API/edge служб лежить вище шару композиційних сервісів. Ці послуги надають споживачам API для доступу як до композитних, так і до атомарних послуг. Це спеціальні композитні сервіси, які виконують основні функції *маршрутизації, встановлення версій API, шаблони безпеки API, регулювання, застосовують монетизацію, створюють композиції API* тощо.

Потреба в шаблонах для архітектури мікросервісів

- Шаблони проектування використовуються для представлення кращих практик, адаптованих досвідченими об’єктно-орієнтованими розробниками програмного забезпечення.
- Шаблон проектування систематично називає, мотивує та пояснює загальний дизайн, який розглядає постійно виникаючі проблеми проектування в об'єктно-орієнтованих системах.
- Він описує проблему, рішення, коли застосовувати рішення та його наслідки.
- Шаблони проектування мають дві основні переваги. По-перше, вони **забезпечують спосіб вирішення питань, пов’язаних із розробкою програмного забезпечення, за допомогою перевіреного рішення**.

- Рішення сприяє розробці **високозв'язних модулів з мінімальним зв'язком**. Вони виділяють мінливість (variability), яка може існувати в системних вимогах, що робить загальну систему простішою для розуміння та обслуговування.
- По-друге, шаблони дизайну **роблять спілкування між дизайнерами більш ефективним.** Професіонали програмного забезпечення можуть одразу уявити собі висококласний дизайн у своїх головах, коли вони звертаються до назви шаблону, який використовується для вирішення певної проблеми під час обговорення дизайну системи.
- Оскільки зразки еволюціонували на основі найкращих практик, їх можна успішно застосовувати до подібних повторюваних проблем.

- У контексті мікросервісів, оскільки **мікросервіси спрямовані на забезпечення швидкого та безперервного розгортання**, використання шаблонів дозволить швидше та ефективніше безперервну доставку (continuous delivery).
- Мета мікросервісів - **збільшити швидкість випусків додатків**; шаблони можуть допомогти пом'якшити проблеми, пов'язані з проектуванням, розробкою та розгортанням програм на базі мікросервісів.
- Хоча MSA вирішує певні проблеми, вона має кілька недоліків, і, отже, роль шаблонів підвищується, щоб вигідно використовувати Архітектуру мікросервісів для створення критично важливих додатків.
- Типові принципи MSA можна зрозуміти, використовуючи рисунок 8.2

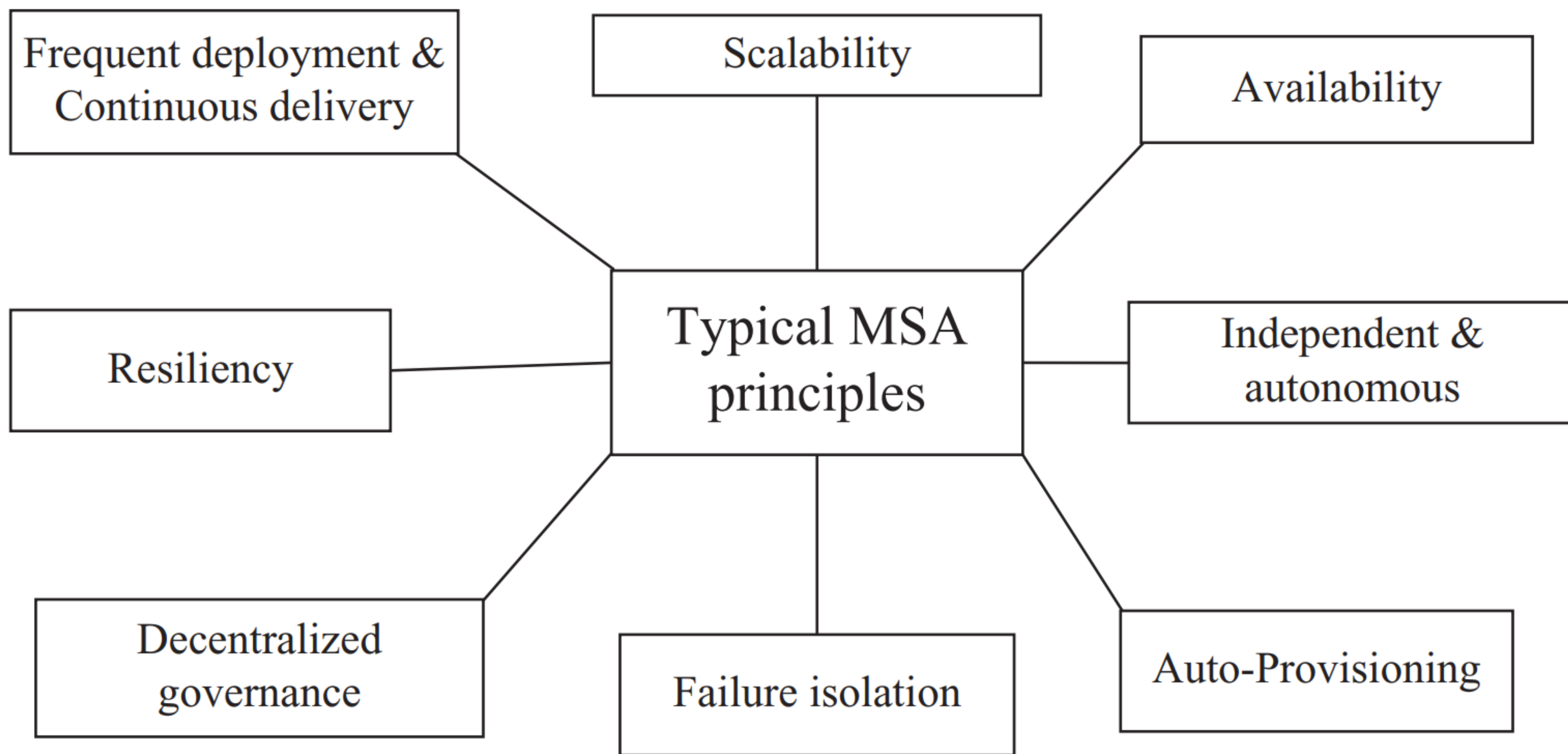


Figure 8.2 MSA principles.

Принципи MSA

Принципи MSA включають

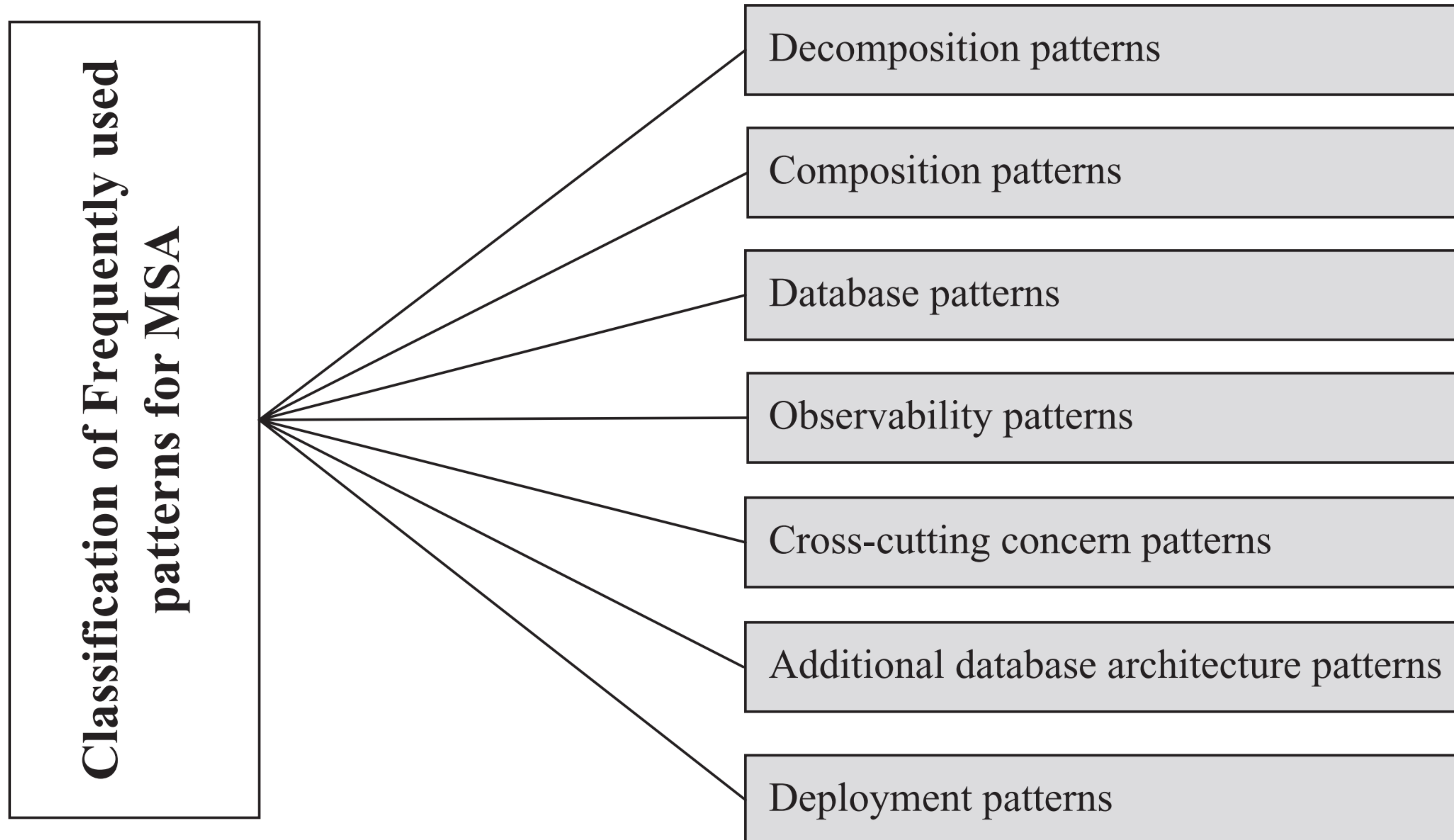
- (i) часте розгортання та безперервну доставку,
- (ii) масштабованість,
- (iii) доступність,
- (iv) стійкість,
- (v) незалежне та автономне,
- (vi) децентралізоване управління,
- (vii) ізоляція відмов та
- (viii) автоматичне забезпечення.

- Застосування всіх цих принципів спричиняє ряд проблем та їх рішень. Тут можна зрозуміти різні проблеми та те, як вирішувати їх, використовуючи різні схеми проектування

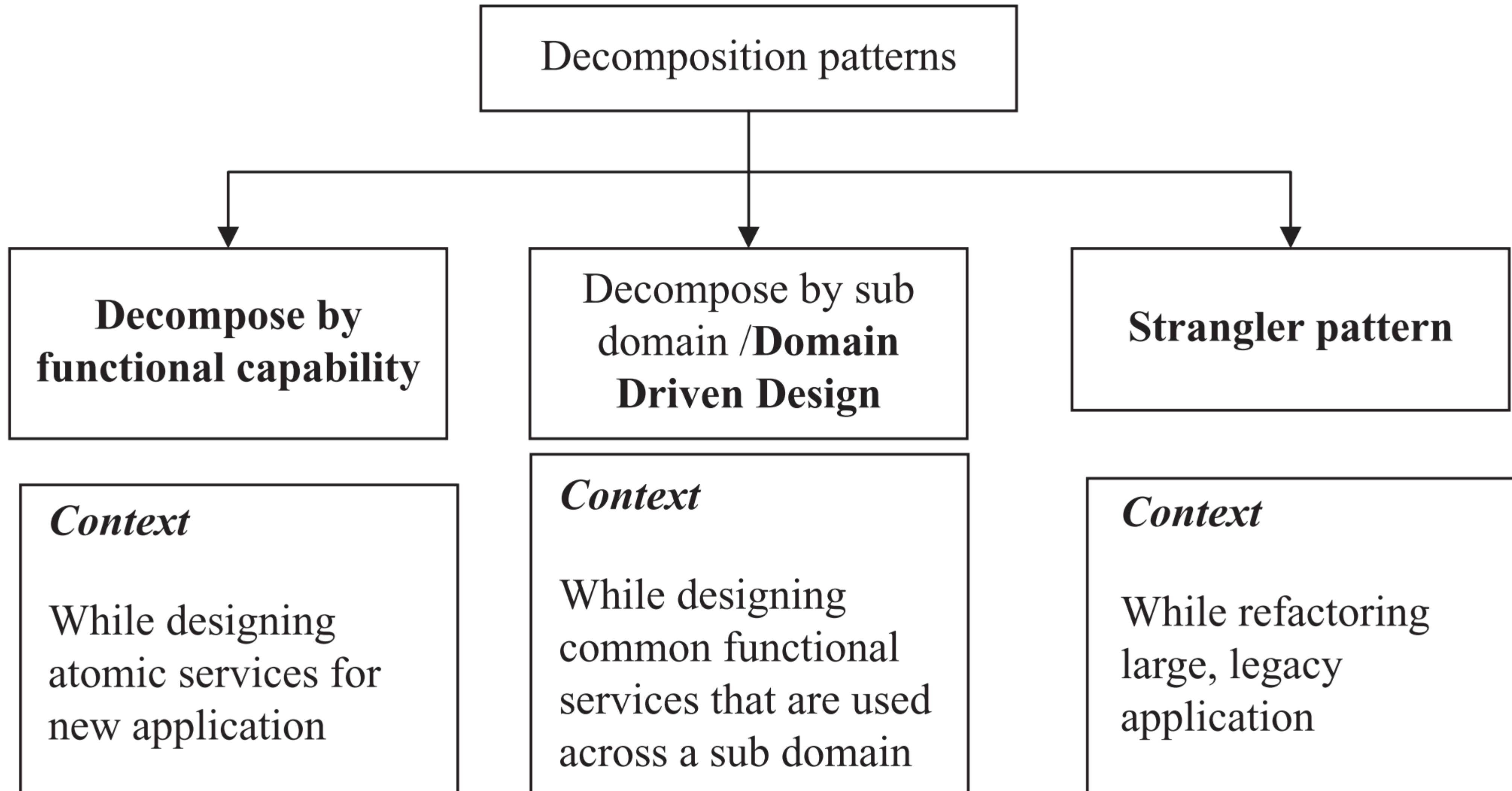
[Rajesh Bhojwani. Design Patterns for Microservices. DZone. 2018. Available at: https://dzone.com/articles/design-patterns-for-microservices .]

- Для реалізації вищезазначених принципів визначені різні моделі, і вони класифіковані, як показано на малюнку (наступний слайд)

Classification of frequently used patterns for microservices



Decomposition Patterns



Decomposition Patterns

У MSA додаток розбивається на незалежні мікросервіси. Величезні програми розділені на безліч інтерактивних та інтелектуальних мікросервісів. У цьому розділі розглядаються три моделі, як показано на попередньому малюнку.

(i) Decompose by business capability pattern

- **Problem** — Як розбити великі програми на менші шматочки? Розкладання великих додатків представляє значну проблему.
- **Context** — При розробці нових додатків, які вимагають частого розгортання та постійної доставки, використовується цей шаблон.
- **Solution—Decompose by business capability** — Одним з життєздатних підходів є декомпозиція програми за діловими можливостями (продовження далі).

Бізнес-здатність - це бізнес-функціонал, який гарантує ділову цінність. Наприклад, можливості страхової організації, як правило, включають *продажі*, *маркетинг*, *андеррайтинг*, *обробку претензій*, *виставлення рахунків*, *дотримання вимог* і т. д. Кожна з цих ділових можливостей розглядається як мікросервіс, який має бути описаний та поданий зовнішньому світу для пошуку та використання.

(ii) Decompose by subdomain/domain driven design pattern

- **Problem** — Як ми розкладаємо певні спеціальні бізнес-можливості, які потрібно використовувати в різних функціях / послугах?
- **Context** — Цей шаблон використовується при розробці нових додатків, які вимагають частого розгортання та постійної доставки. Крім того, при виділенні певних служб як спеціальних функціональних служб, які будуть використовуватися багатьма службами в межах субдомену, використовується цей шаблон (далі буде).

Існують певні інші ситуації з певними класами послуг, і вони використовуються в багатьох службах. Тому декомпозиція програми з використанням бізнес-можливостей недостатня. Наприклад, клас замовлень буде використовуватися для *управління замовленнями, прийому замовлень, доставки замовлень* тощо. Завдання полягає в тому, як ми розкладаємо ці спеціальні бізнес-можливості?

- **Solution — Decompose by subdomain** —Широко відомий Domain Driven Design (DDD) є життєздатною відповіддю. Він цікаво використовує *субдомени* та *обмежені контекстні* концепції для вирішення цієї унікальної проблеми. DDD розбиває всю модель домену на ряд субдоменів. Тепер кожен субдомен матиме власну модель, і область дії цієї моделі називається *обмеженим контекстом*. Мікропослуги розробляються навколо обмеженого контексту. Однак ідентифікація субдоменів також супроводжується проблемами. Глибоке розуміння бізнесу та його організаційної структури стає в нагоді для подолання цих викликів.

(iii) Strangler pattern

- **Problem** — Як переформатувати або переписати застарілі системи в сучасні програми? В рамках старої модернізації аналізуються та модернізуються масивні та монолітні програми. Однак для цього є два варіанти: переписати або рефакторизувати. Переписування чи рефакторинг великомасштабної програми з нуля, безумовно, вимагає багато часу та ризикує, оскільки немає хороших документів для застарілих програм. Отже, рефакторинг або переписування застарілих систем на сучасні програми є складною проблемою.
- **Context** — Під час модернізації великих, застарілих програм шляхом рефакторингу використовується *шаблон давитель*.

- **Solution — The Strangler pattern** — Шаблон Задушника зменшує вищезазначений ризик. Замість того, щоб переписувати / переробляти всю програму, за допомогою цієї схеми пропонується *поетапна заміна функціональних можливостей програми*. Бізнес-цінність нової функціональності реалізується та надається швидше. Шаблон Strangler наголошує на поступовому перетворенні монолітних програм у мікросервіси, замінюючи певну функціональність новою службою як послугою, що розгортається в індивідуальному порядку, як показано на Рис. 8.5. Коли нова функціональність готова, старий компонент *задушується*, нова служба вводиться в експлуатацію, а старий компонент взагалі виводиться з експлуатації.

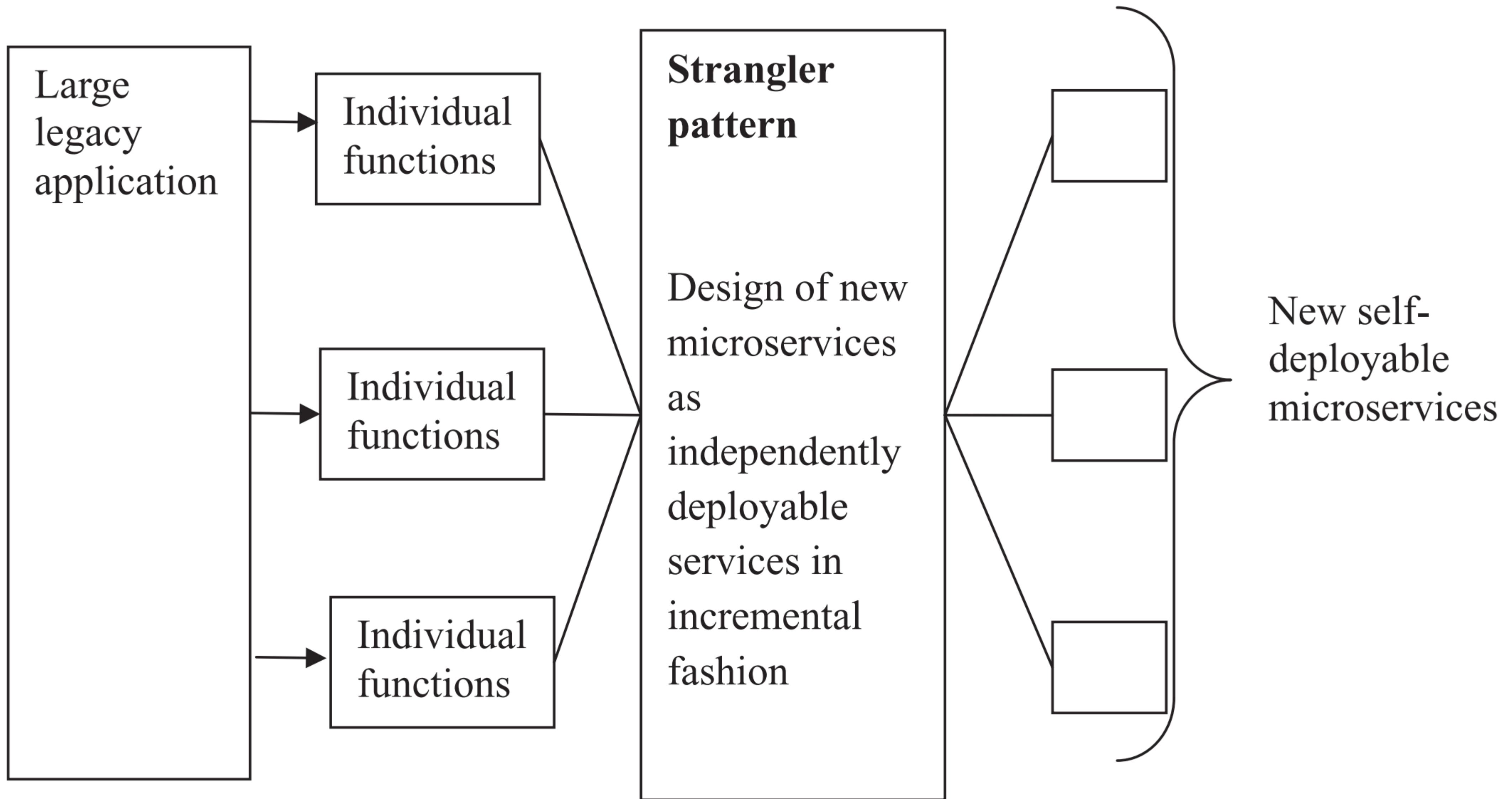


Figure 8.5 The Strangler pattern for refactoring a large legacy application

Composition Patterns

Часто використовувані композиційні шаблони для MSA включають:

(i) Aggregator patterns
(ii) Proxy patterns
(iii) Chained Patterns
(iv) Branch Microservice Patterns
(v) Shared resource patterns
(vi) API gateway patterns
(vii) The client-side UI composition patterns

(i) Aggregator pattern

- **Problem** — Потрібно поєднувати функції, що надаються більш ніж однією службою.
- **Context** — Під час розробки композитних служб, а також під час реалізації загальних функцій, таких як балансування навантаження, маршрутизація, виявлення та перетворення даних / перетворення протоколів, існує необхідність поєднувати функції, що надаються більш ніж однією службою. У таких ситуаціях використовується шаблон Агрегатор.

Розбиваючи будь-яку ділову функціональність на кілька менших логічних частин коду (мікропослуги), важливо продумати шляхи та *засоби надання їм можливості взаємодіяти одна з одною*. Крім того, дані, надіслані однією службою, повинні однозначно розуміти інші служби (дивись далі).

Якщо одна служба надсилає повідомлення, воно повинно бути повністю зрозуміле та оброблене, щоб виконати те, що наказано. Оскільки клієнти не знають, як реалізується додаток постачальника, цим *перекладом даних і перетворенням протоколів* повинна займатися горизонтальна служба, така як служба Агрегатора.

- **Solution — Aggregator pattern** — Як шаблон Агрегатора використовується для поєднання багатьох функцій, пояснюється на типовому прикладі, скажімо, *проектуванні веб-сторінки*, яка складається з багатьох елементів інтерфейсу, і кожен з них відповідає функції. У найпростішій формі Агрегатор - це проста веб-сторінка, яка викликає кілька служб, що реалізують ділову функцію. Це зазвичай використовується у веб-програмі на базі мікропослуг (див. Рис. 8.6). «Агрегатор» відповідає за виклик різних служб по одній.

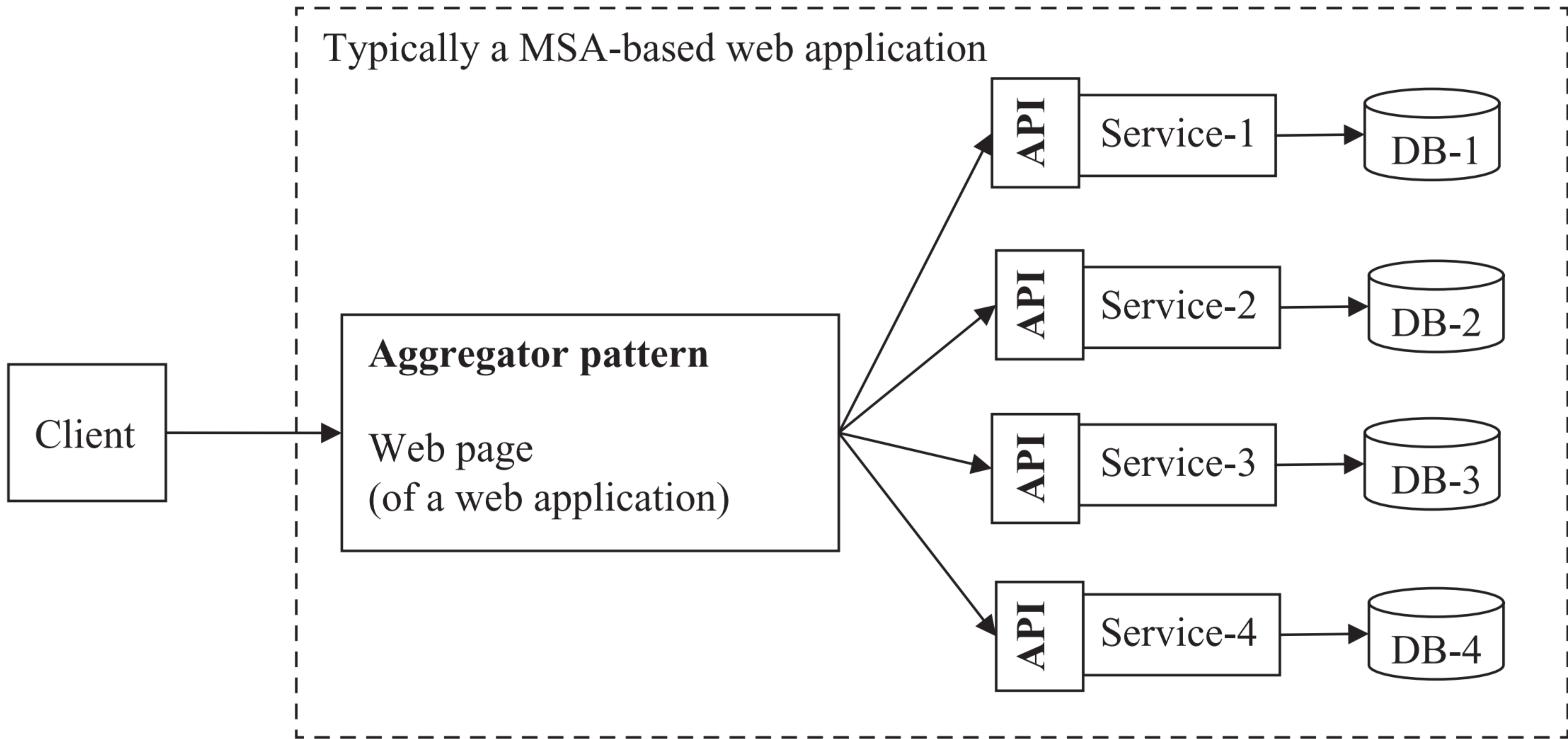


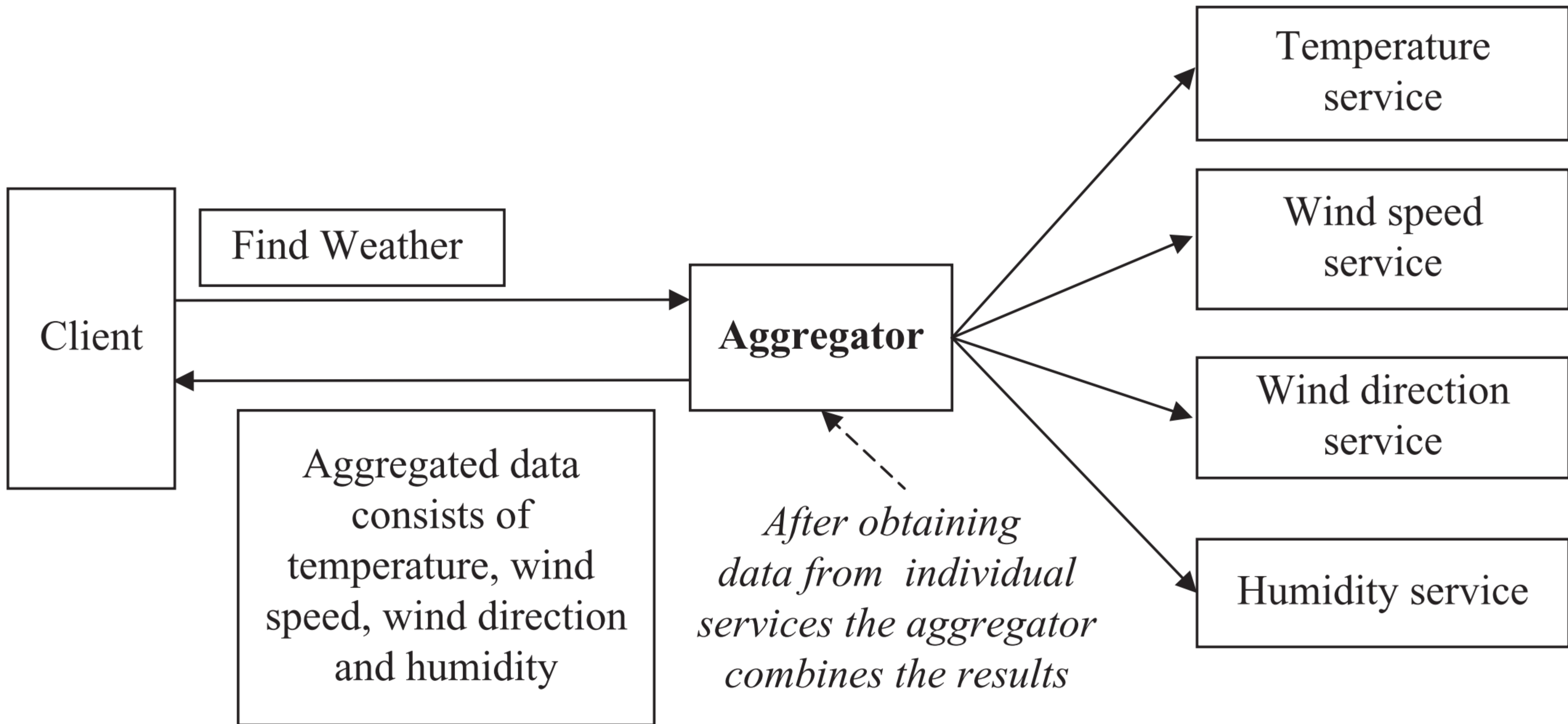
Figure 8.6 A simple Aggregator pattern

Інший варіант для агрегатора - це *композитна мікрослужба вищого рівня, яку можуть використовувати інші служби*. У цьому випадку Агрегатор просто *збирає дані з кожного окремого мікросервісу, застосовує до нього бізнес-логіку та надалі публікує їх як кінцеву точку REST*. Потім це можуть споживати інші служби, які цього потребують.

Наприклад, існує чотири різні послуги, такі як:

- Сервіс Температури - визначає температуру місця.
- Сервіс швидкості вітру - дає швидкість вітру місця.
- Сервіс напрямку вітру - дає напрямок, в якому дме вітер.
- Служба вологості - дає відносну вологість місця.

Тепер припустимо, що клієнт робить **один запит**, знайти погоду. Внутрішня служба Aggregator призначена для використання вищезазначених окремих послуг та поєднання результатів окремих служб. Це показано на Рис. 8.7.



Шаблон Агрегатор, що виконує агрегування даних.

Proxy pattern

- **Problem** — Як запровадити контрольований доступ до мікросервісу? Або як приховати прямий вплив мікросервісу на клієнтів?
- **Context** — Можуть бути ситуації, коли розробник хоче приховати прямий вплив API певних мікросервісів на клієнтів. Тобто доступ до мікросервісу потрібно контролювати. У деяких інших випадках під час доступу до об’єкта може знадобитися додаткова функціональність. У цих ситуаціях використовується шаблон проксі.
- **Solution — Proxy pattern** — Проксі - це мікросервіс, який функціонує як інтерфейс до іншого мікросервісу, для якого слід заборонити прямий доступ. Це показано на Рис. 8.8.

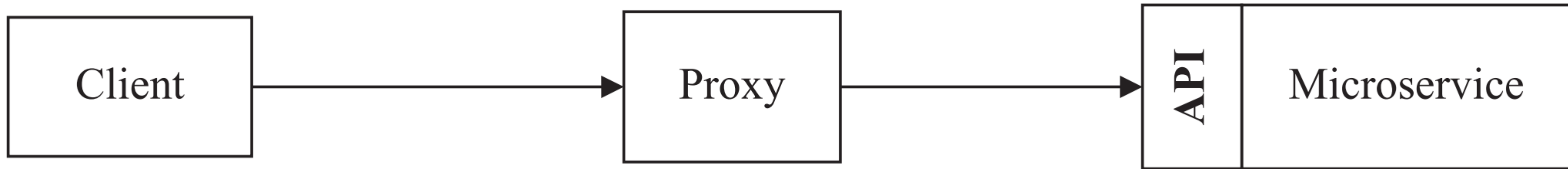


Figure 8.8 A Proxy pattern

- Проксі - це обгортка або об'єкт агента, який викликається клієнтом для доступу до реального обслуговуваного мікросервісу.
- Це може надати додаткову логіку. Наприклад, проксі-сервер може перевірити передумови до того, як будуть викликані операції на реальних мікросервісах. Він може перевірити, чи мають клієнти необхідні права доступу.

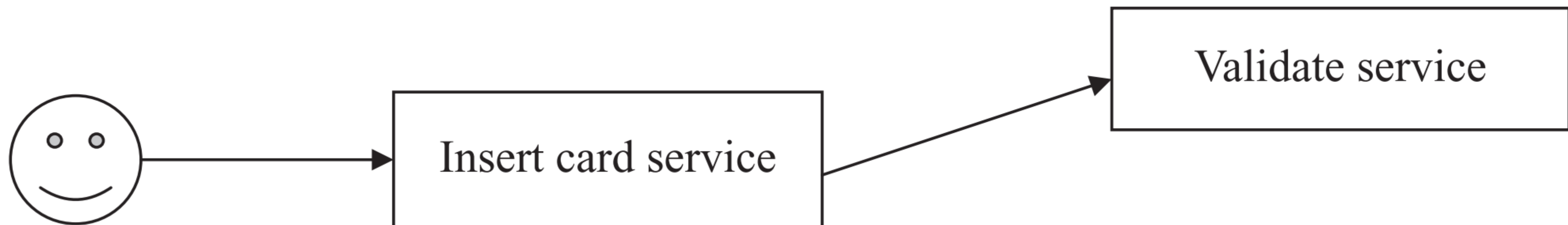
Chained pattern (Ланцюговий патерн)

- Цей тип композиційного шаблону буде дотримуватися *ланцюгоподібної структури*.
- Тут клієнт спілкується безпосередньо зі службами, і всі служби будуть прив’язані до мережі таким чином, що вихід однієї послуги буде входом для наступної послуги. Ланцюговий патерн зображений на Рис. 8.9. Зверніть увагу, що ділова логіка пов’язана з окремими послугами (хореографія).

Chained pattern for implementing Travel plan service

Branch microservice pattern

- Це розширена версія шаблону Агрегатор та Ланцюжок. Клієнт може безпосередньо спілкуватися зі службою. Крім того, одна служба може спілкуватися одночасно з кількома службами. Шаблон мікросервісу Branch показаний на Рис. 8.10. Тут клієнт викликає послугу, вставляючи смарт-карту із послугою вставки картки. Внутрішньо служба вставки картки викликає службу перевірки, яка перевіряє тип смарт-картки.



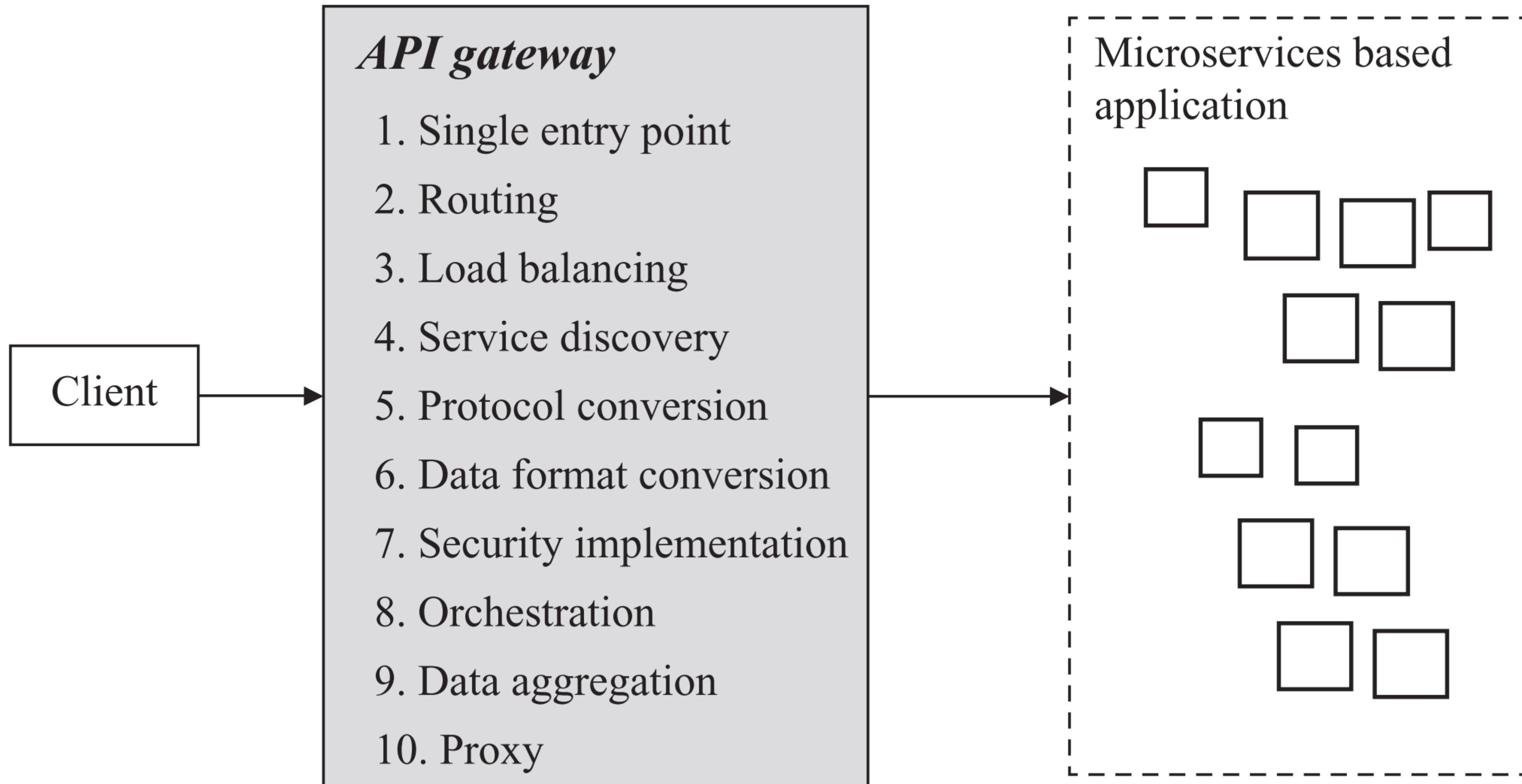
Shared resource pattern

- Це насправді конгломерат усіх типів зразків, згаданих раніше. За такою схемою клієнт або **балансир навантаження** безпосередньо спілкуватимуться з кожною службою, коли це необхідно.

API gateway pattern

- **Problem** — Коли програма розбивається на атомарні сервіси, як клієнт викликає та реалізує корисний бізнес-процес?
- **Context** — Коли програма розбивається на ряд мікросервісів, існує кілька проблем, які потрібно вирішити.
  - Виклик декількох мікросервісів та отримання інформації про виробника (producer).
  - Існує кілька категорій пристроїв та каналів введення / виводу. Служби додатків повинні отримувати та реагувати по-різному.
  - Перетворення даних та протоколів необхідні для того, щоб різні та розподілені мікросервіси могли співпрацювати між собою. Існує декілька типів клієнтів, які взаємодіють із серверними сервісами та базами даних.

- **Solution—API gateway pattern** допомагає вирішити вищезазначені проблеми. Це показано на Рис. 8.11.
  - Шлюз API - це єдиний пункт входу для будь-якого виклику мікросервісу.
  - Він також може працювати як проксі-сервіс для маршрутизації запитів до мікросервісів та їх екземплярів.
  - Він може розіслати запит до декількох служб і згрупувати результати, щоб відправити їх назад клієнту.
  - Універсальні API не можуть відповідати різним вимогам споживача. Шлюзи API можуть створювати детальний API для кожного конкретного типу клієнта.
  - Він також може конвертувати запит протоколу (наприклад, AMQP) в інший протокол (наприклад, HTTP) і навпаки.
  - Це також може розвантажити відповідальність за аутентифікацію / авторизацію мікросервісу.



An API gateway pattern

Client-side UI composition pattern

- **Problem** — Коли функціональні можливості програми пов’язані між кількома службами, як ви надаєте клієнтам інтегровану інформацію, коли дані / інформація повинні збиратися та складатися з декількох служб?
- **Context** — Надаючи інформацію користувачам, існує потреба у поєднанні результатів, щоб забезпечити інтегрований погляд для користувачів.
- **Solution — Client-side UI composition pattern** — Як зазначено вище, мікросервіси розробляються шляхом декомпозиції ділових можливостей / субдоменів. Тобто існує декілька служб, що працюють разом для реалізації бізнес-систем. Тепер, щоб повернути споживачам інтегровану та проникливу інформацію, дані / інформація повинні збиратися та складатися з безлічі служб.

- В епоху мікросервісів користувальницький інтерфейс (UI) повинен бути спроектований як скелет з декількома розділами/регіонами екрану/сторінки.
- Кожен розділ повинен здійснити виклик до окремої внутрішньої мікросервісної служби для отримання даних.
- Існує кілька втілюючих фреймворків, таких як AngularJS та ReactJS, які допомагають легко зробити цю композицію інтерфейсу.
- Ці додатки відомі як Single Page Applications (SPA). Цей тип налаштування дозволяє програмі оновлювати певну область екрана/сторінки замість цілої сторінки.

Database Patterns

До часто використовуваних шаблонів бази даних належать:

- **Database per service pattern**
- **Shared database per service pattern**
- **Command Query Responsibility Segregator (CQRS) pattern**
- **Saga pattern**

Database per service pattern

- **Problem** — Як досягти незалежності щодо баз даних?

У MSA кожна служба повинна бути незалежно розроблена, налагоджена, доставлена, розгорнута та масштабована.

Крім того, вони також повинні бути вільно з'єднані (loosely coupled). Одна послуга не повинна залежати від інших служб.

Як служба може зберігати власні дані?

Як досягти незалежності щодо баз даних?

- **Context** — До операцій та функцій баз даних може бути багато вимог, таких як наступні.
  - Бізнес-операції можуть застосовувати інваріанти, що включають кілька служб.
  - Деякі комерційні операції можуть наполягати на запиті даних, які належать декільком службам.
  - Бази даних повинні бути відтворені, щоб забезпечити високу та горизонтальну масштабованість.
  - Різні служби мають різні вимоги до моделі даних/зберігання.

У вищезазначених ситуаціях використовуються різні моделі баз даних.

- **Solution — Database per service pattern** — Розуміючи різні вимоги, рекомендується мати екземпляр бази даних для кожного мікросервісу. Доступ до бази даних здійснюється лише через API відповідного сервісу. Тобто, до бази даних не можуть отримати доступ інші служби безпосередньо. Наприклад, для реляційних баз даних доступні такі варіанти:
  - Private-tables-per-service
  - Schema-per-service
  - Database-server-per-service

Кожний мікросервіс повинен мати окремий ідентифікатор бази даних, і одна служба не може отримати доступ до таблиць іншої служби. Таким чином, безпека даних реалізується за допомогою такого розділення.

Shared database per service pattern

- **Problem** — Як поділитися базами даних між кількома службами?
- **Context** — Переважним аспектом є надання можливості кожному мікросервісу мати власну базу даних. Однак існують певні виклики, особливо коли ми використовуємо додатки, що базуються на перевіреній техніці DDD. Крім того, застарілі програми отримують централізовані та спільні бази даних. Модернізація застарілих додатків за допомогою MSA породжує інші проблеми, що стосуються архітектури баз даних.

- **Solution — Shared database per service** — Надзвичайний успіх мікросервісів обумовлений окремою базою даних для кожної служби. Для додаткових програм (маючи на увазі програми, які модифіковані або модернізовані), наявність окремої бази даних є ідеальним та стійким. Однак для додатків, що працюють на гринфілді - for greenfield applications - (маючи на увазі програми, розроблені з нуля), одна база даних може бути пов’язана та узгоджена з декількома мікросервісами. Число має бути в межах двох-трьох. В іншому випадку помітні функції (масштабування, автономність тощо) MSA втрачаються.

Command Query Responsibility Segregation pattern

- **Problem** — Як досягти високої доступності сучасних додатків із великими веб-сайтами? Виділення однієї бази даних для кожного мікросервісу ставить питання про те, *як можна підключити чи опитати кілька служб для певних запитів до бази даних*.
- **Context** — Коли програма електронної комерції має великі веб-сайти, завжди існує необхідність забезпечити достатню доступність веб-сайтів величезній кількості клієнтів. У таких ситуаціях використовується шаблон CQRS.

- **Solution—Command Query Responsibility Segregation pattern**— Цей підхід пропонує розділити програму на *дві частини: сторону команди та сторону запиту*. Команда обробляє запити на створення, оновлення та видалення, тоді як сторона запиту обробляє частину запиту, використовуючи materialized views. Іншими словами, CQRS - це спосіб роз'єднати записи (команда) та читання (запит). Тобто ми можемо мати одну базу даних для управління частиною запису. Частина зчитування (materialized views або проекція) є похідною від частини запису і може керуватися однією або декількома базами даних. Це показано на Рис. 8.12.

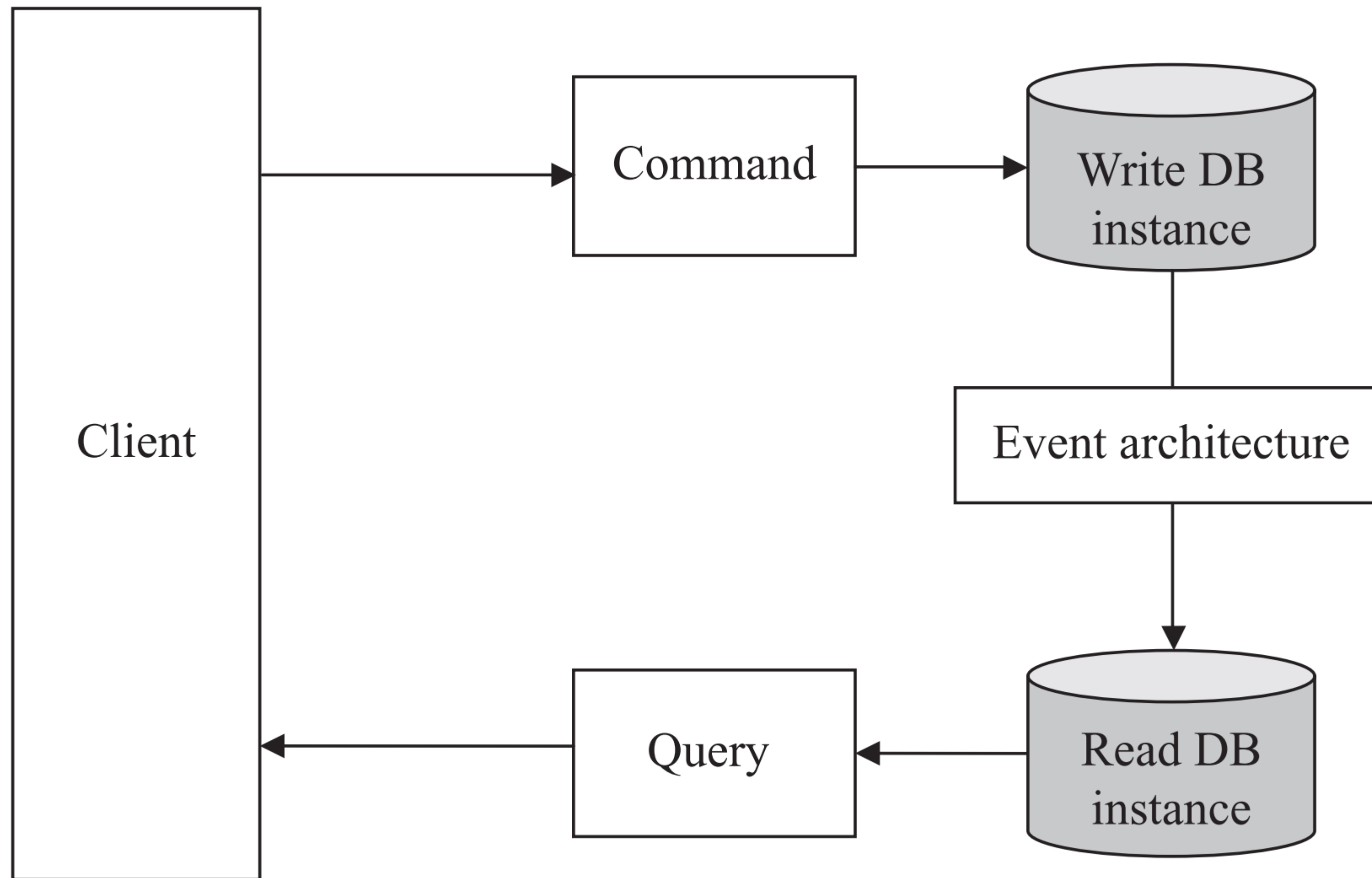


Рис. 8.12 Segregating read and write operations using a CQRS pattern

- Як правило, частина <u>зчитування</u> обчислюється асинхронно, а отже, обидві частини не є суворо узгодженими.
- <u>Шаблон пошуку джерел подій</u> зазвичай використовується для створення подій для будь-якої зміни даних. <u>Частина зчитування (materialized views) постійно оновлюється шляхом *підписки на потік подій*</u>.
- Шаблон пошуку подій гарантує, що <u>*всі зміни стану програми зберігаються як послідовність подій*</u>. Це означає, що <u>стан об'єкта не зберігається. Але всі події, що впливають на стан об'єкта, зберігаються</u>. Потім, щоб отримати стан об'єкта, потрібно прочитати різні події, пов'язані з цим об'єктом, і застосовувати їх по одному відповідно до позначки часу.

- **CQRS + Event sourcing** — Обидва шаблони часто групуються разом. Застосування джерела подій на вершині CQRS означає, що кожна подія зберігається в частині запису програми. Тоді прочитана частина виводиться з послідовності подій з архітектури подій (див. Рис. 8.12).

- Примітка. Більш детально про CQRS дивись [11]

  Practical Microservices Architectural Patterns. – Apress, 2019, Ch. 5

Saga pattern

- **Problem** — Як реалізувати транзакцію бази даних, коли вона включає операції з базами даних у декількох мікросервісах?
- **Context** — У MSA кожна служба пропонує обмежену функціональність із власною базою даних. Ця тенденція мікропослуг означає, що ***ділова транзакція, як правило, охоплює кілька служб*** (оскільки дані пов'язані з різними службами). Цей вид розподілених транзакцій, забезпечуючи узгодженість даних у всіх базах даних, є справжньою проблемою. Наприклад, для програми електронної комерції, де клієнти мають кредитний ліміт, програма повинна забезпечити, щоб нове замовлення не перевищувало кредитний ліміт клієнта. Оскільки замовлення та клієнти знаходяться в різних базах даних, програма не може просто використовувати локальну транзакцію ACID. Потреба в розподіленій транзакції зростає. Для прикладу цієї вічної проблеми розглянемо наступну архітектуру мікропослуг високого рівня системи електронної комерції.

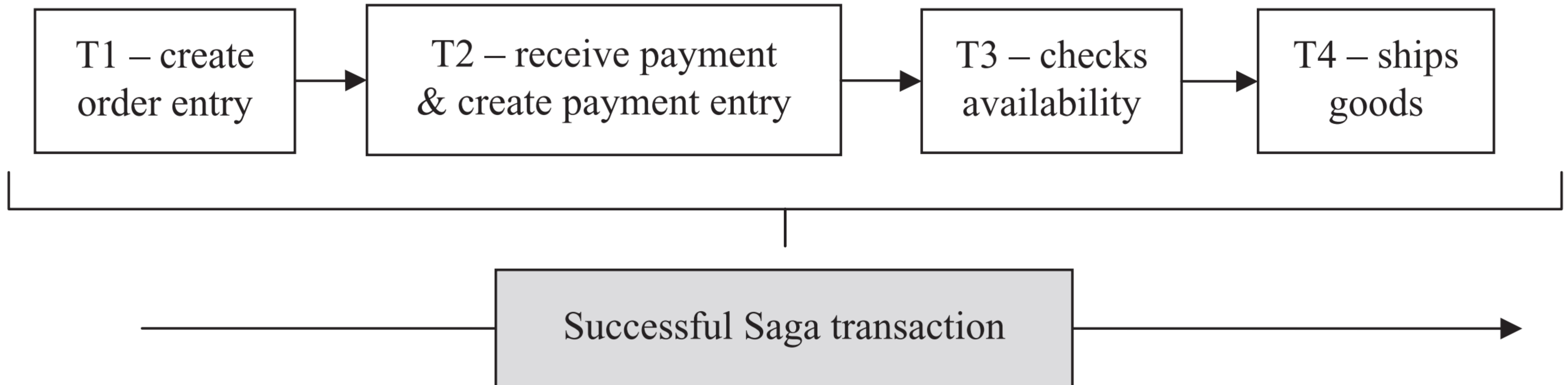
У наведеному вище прикладі не можна просто оформити замовлення, стягнути плату з клієнта, оновити запас і відправити його на доставку. Усі ці аспекти не можна врахувати в одній транзакції. Щоб послідовно виконувати весь цей потік, важливо створити розподілену транзакцію. Робота з перехідними станами, можлива узгодженість послуг, ізоляція та відкат - це сценарії, які слід враховувати на етапі проектування.

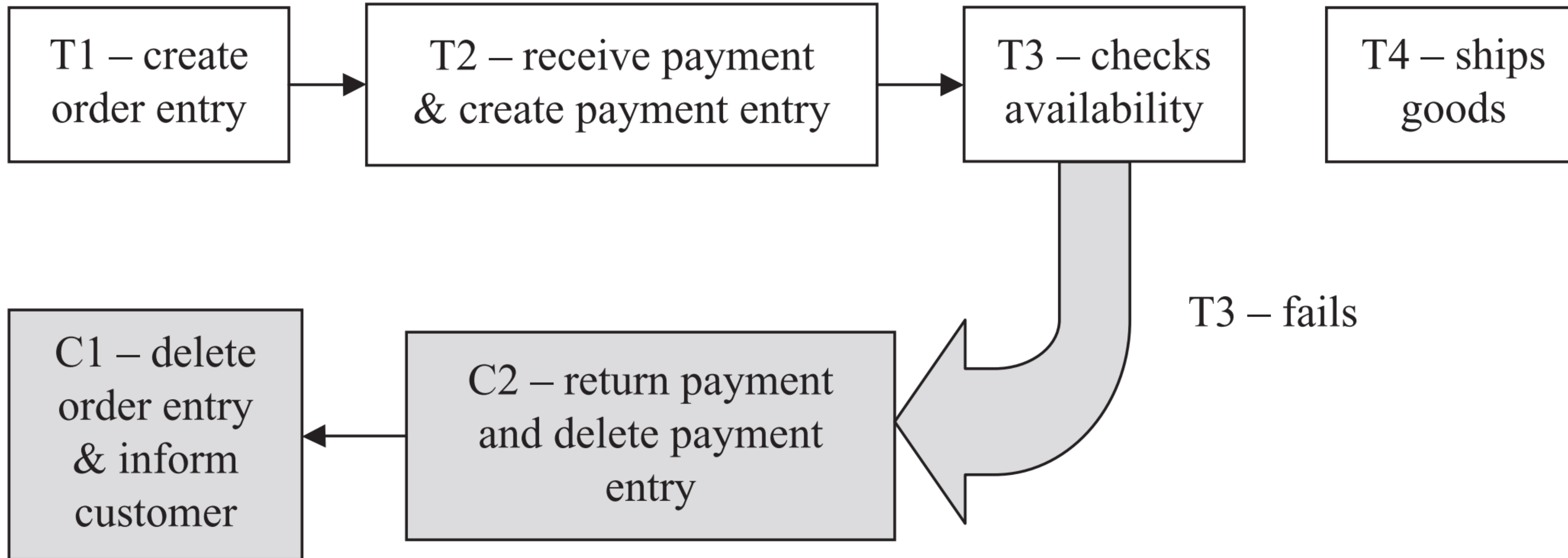
Solution—Saga pattern — *Сага - це послідовність локальних транзакцій, де кожна транзакція оновлює дані в межах однієї служби.* Перша транзакція ініціюється зовнішнім запитом, що відповідає операції системи, а потім кожен наступний крок ініціюється завершенням попереднього. (Дивись далі)

Кожна локальна транзакція має компенсаційну транзакцію, яка виконується, коли запит не вдається. Нехай T1, T2, T3, ... Tn позначають набір локальних транзакцій, що відбуваються в послідовності. Кожна локальна транзакція повинна мати свою компенсаційну транзакцію. Нехай C1, C2, C3, ... Cn позначають компенсуючі операції. Шаблон Saga використовується для обробки помилок у розподіленій транзакції.

У транзакції Saga кожна операція з базою даних у відповідному мікросервісі відбувається як локальна транзакція. Якщо всі служби успішно завершують свої локальні транзакції, тоді транзакція Saga завершується успішно. Якщо будь-яка з локальних транзакцій зустрічається з помилкою, тоді всі локальні транзакції, що відбулися раніше, відкочуються, виконуючи компенсаційні транзакції в зворотному порядку.

- Наприклад, розглянемо чотири послуги, службу замовлення (order service), платіжну службу (payment service), послугу перевірки наявності (check availability service) та послугу доставки (shipment service).





Існує кілька способів реалізації транзакції Saga.

- **Events/Choreography** — Коли центральної координації немає, кожна служба виробляє та слухає події іншої служби та вирішує, чи слід вживати заходів чи ні.
- **Command/Orchestration** — Коли сервіс координатора відповідає за централізацію прийняття рішень Saga та послідовність бізнес-логіки.

Observability Patterns

(Шаблони спостережуваності)

- Мікросервіси - це не що інше, як звичайні розподілені системи. Ключовим відмінником є те, що кількість мікросервісів, що беруть участь та вносять внесок у будь-який розподілений додаток, досить велика. Тому вони загострюють добре відомі проблеми, з якими стикається будь-яка розподілена система, такі як *відсутність видимості ділової транзакції через межі процесу*.
- Кажуть, що систему можна спостерігати, коли можна зрозуміти її стан *на основі метрик, журналів та слідів*, які вона видає. Оскільки існує багато служб та їх екземплярів, *важливо агрегувати метрики для всіх екземплярів даної служби*, можливо згрупованих за версіями.

Рішення метрик відіграють вирішальну роль у вирішенні проблеми спостережуваності. ***Існують деякі шаблони спостережуваності***, які часто використовуються для мікросервісів. Вони є:

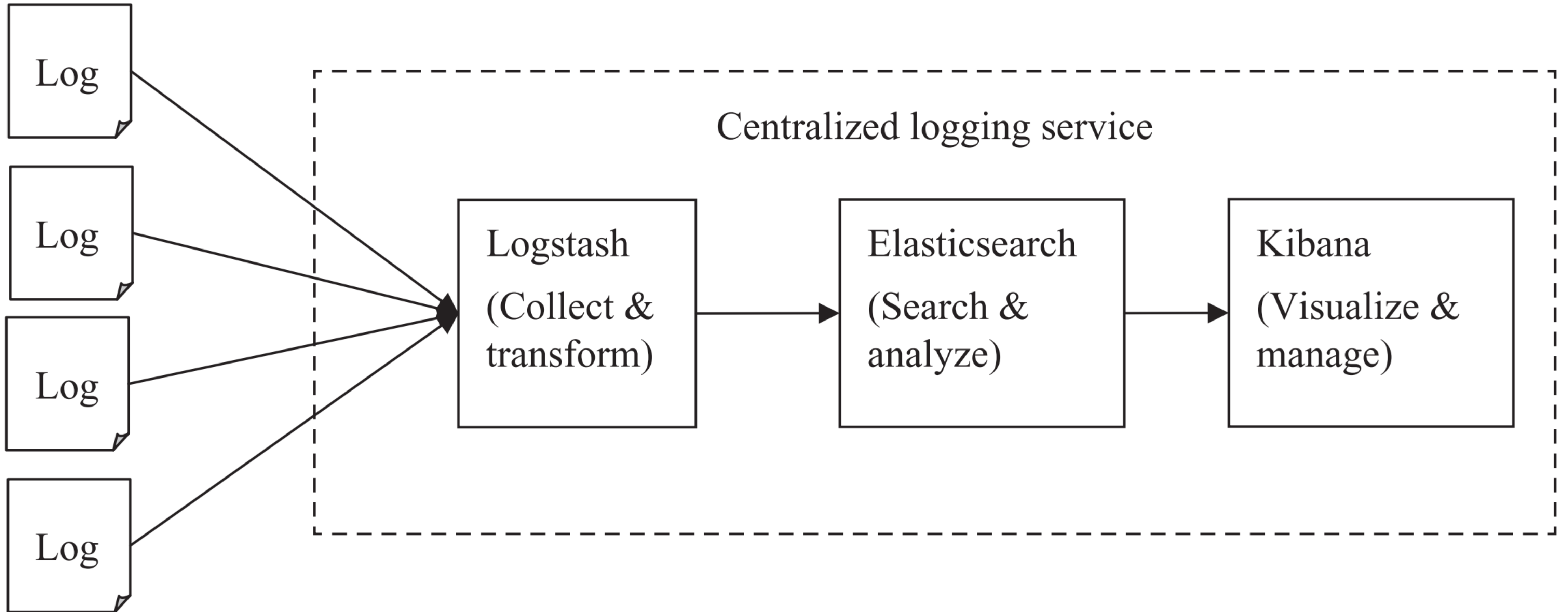
- **Centralized logging service pattern** (Централізований шаблон служби ведення журналу)
- **Application performance metrics pattern** (Шаблон метрики продуктивності програми)
- **Distributed tracing pattern** (Розподілений шаблон трасування)
- **Health check pattern** (Шаблон перевірки стану здоров'я)

Centralized logging service pattern

- **Problem** — Як зрозуміти поведінку програми через журнали для конкретного запиту?
- **Context** — У світі мікросервісів додаток складається з безлічі екземплярів служб, що працюють на декількох машинах. Тобто запити на обслуговування часто охоплюють кілька екземплярів служби. Кожен екземпляр служби генерує файл журналу у стандартизованому форматі. Проблема тут полягає в тому, як зрозуміти поведінку програми через журнали для конкретного запиту?

- **Solution—Centralized logging service pattern** — Централізована служба журналювання збирає журнали з кожного екземпляра служби. Користувачі можуть шукати та аналізувати журнали. Вони можуть налаштовувати попередження, які запускаються, коли в журналах з’являються певні повідомлення.

Наприклад, існують рішення для агрегування журналів, які збирають журнали з кожного програмного та апаратного компонента. Журнали потрібно зберігати в центральному місці, оскільки неможливо проаналізувати журнали з окремих екземплярів кожної служби. Інструменти та технології з відкритим кодом можуть бути використані для реалізації централізованої служби ведення журналу. Наприклад, може використовуватися один із *технологічних стеків з відкритим кодом, ELK, що є абревіатурою для трьох проектів з відкритим кодом: **Elasticsearch**, **Logstash** та **Kibana***.


Log
Log
Log
Log
Centralized logging service
Logstash
(Collect & transform)
Elasticsearch
(Search & analyze)
Kibana
(Visualize & manage)

- **Elasticsearch** - це високо масштабований повнотекстовий механізм пошуку та аналітики з відкритим кодом. Він може зберігати, шукати та аналізувати великі обсяги даних швидко і майже в реальному часі.
- **Logstash** - це інструмент з відкритим кодом для збору, аналізу та зберігання журналів для подальшого використання.
- **Kibana** допомагає у візуалізації даних за допомогою діаграм та графіків. Таким чином, Logstash використовується в поєднанні з Elasticsearch та Kibana для реалізації функції централізованого сервісу реєстрації. Це показано на Рис. 8.15.

Application performance metrics pattern (Шаблон метрики продуктивності програми)

- **Problem** — Як збирати метрики для моніторингу продуктивності програми?
- **Context** — Після моніторингу різних метрик з окремих мікросервісів виникає потреба перетворити зібрані метрики у бізнес-значення або інші значення, тобто потрібно збирати ті метрики, які дійсно передають значущі ідеї. Отже, це справжній виклик знайти метрики, які відображають ефективність програми? У цій ситуації використовується шаблон метрики продуктивності програми.

- **Solution—Application performance metrics pattern** — Призначення APM - виявляти та діагностувати складні проблеми роботи продуктивності, щоб підтримувати очікуваний рівень сервісу. APM - це переклад IT-показників у ділове значення або цінність. Існують показники, пов’язані з продуктивністю, та автоматизовані рішення для захоплення всіх видів метрик. Крім того, існують аналітичні платформи та продукти, які дозволяють отримати ефективне розуміння даних про ефективність, щоб виправити будь-який вид погіршення продуктивності.

Як правило, наступні показники ретельно збираються та піддаються різноманітним дослідженням для підвищення продуктивності додатків. APM може допомогти зробити кілька ключових речей, таких як:

- *Measure and monitor application performance* (Вимірювати та контролювати ефективність програми).
- *Find the root causes of application problems* (Знайти основні причини проблем із додатками).
- *Identify ways to optimize application performance* (Визначити шляхи оптимізації продуктивності програми).

APM - це один або декілька програмних (та / або апаратних) компонентів, які полегшують моніторинг для досягнення таких функціональних замірів.

- **End-user experience monitoring** (Моніторинг взаємодії з кінцевими користувачами) — Це є збирання даних про ефективність на основі користувачів, щоб оцінити, наскільки добре працює програма, та виявити потенційні проблеми з продуктивністю.

- **Application topology discovery and visualization** (Виявлення та візуалізація топології додатка) — Це забезпечує візуальне вираження програми на схемі, щоб встановити всі різні компоненти програми та те, як вони взаємодіють між собою.
- **User-defined transaction profiling** (Користувальницьке профілювання транзакцій) — Це вивчає конкретні взаємодії для відтворення умов, які призводять до проблем з продуктивністю для цілей тестування.
- **IT operations analytics** (Аналітика IT-операцій) — Це допомагає виявити схеми використання, виявити проблеми з продуктивністю та передбачити потенційні проблеми до їх виникнення.

Основними показниками (metrics) продуктивності додатків є:

- **Задоволеність користувачів / оцінки Apdex** (оцінка Apdex - це значення співвідношення кількості задоволених та допустимих запитів до загальної кількості надісланих запитів)
- Середній час відгуку
- Частота помилок
- Кількість екземплярів додатків
- Частота запитів
- Прикладний та серверний процесор (Application and server CPU)
- Доступність програми

Distributed tracing pattern (Розподілений шаблон трасування)

- **Problem** — Як простежити запит наскрізно (end-to-end), щоб усунути проблему?
- **Context** — У MSA запити клієнтів часто надходять до декількох служб. Кожна служба зазвичай обробляє запит, виконуючи одну або декілька операцій у кількох службах. Надзвичайно часто одночасно працює кілька версій однієї служби. Це помітно в сценарії тестування A / B або коли ми випускаємо новий реліз, застосовуючи техніку випуску Canary. Коли існують сотні мікросервісів, практично неможливо скласти карту взаємозалежностей та зрозуміти шлях ділової транзакції між службами та їхніми різними версіями. Проблема тут полягає в тому, як відстежити запит наскрізним способом для усунення проблеми?

Важливо мати сервіс, який:

- Кожному зовнішньому запиту присвоює унікальний ідентифікатор зовнішнього запиту.
- Передає ідентифікатор зовнішнього запиту для всіх служб.
- Включає зовнішній ідентифікатор запиту у всіх повідомленнях журналу.
- Записує інформацію (наприклад, час початку, час закінчення) про запити та операції, виконані під час обробки зовнішнього запиту в централізованій службі.

- **Solution—Distributed tracing pattern** — Цей шаблон (альтернативно шаблон розподіленого трасування запитів) використовується для профілювання та моніторингу програм, особливо тих, що побудовані за допомогою архітектури мікросервісів. Розподілене трасування допомагає визначити, де трапляються збої та що спричиняє низьку продуктивність. Відстеження дає уявлення про код, що виконується в усіх службах.

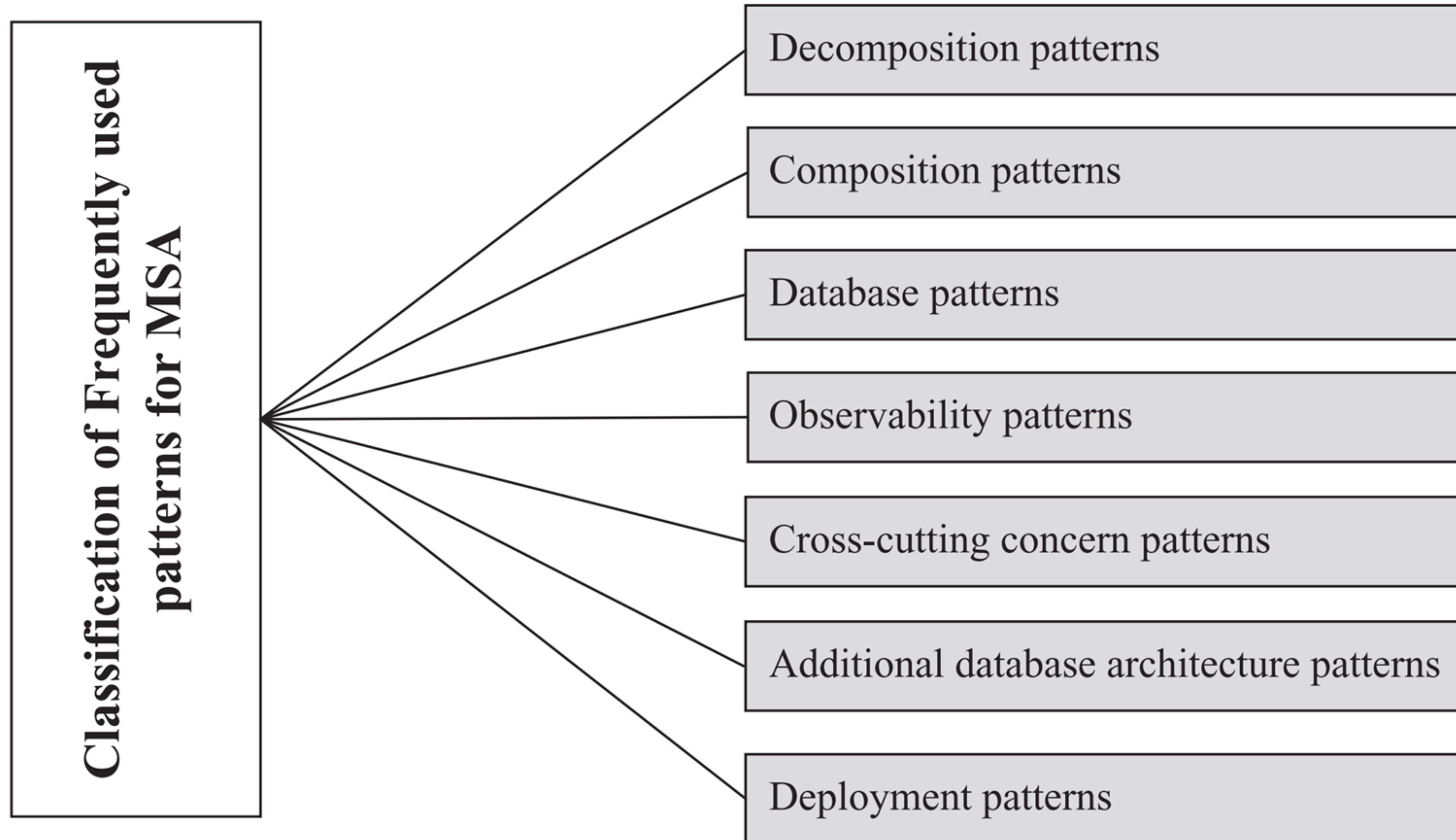
Health check pattern (Шаблон перевірки стану здоров'я)

- **Problem** — Чи функціонує мікросервіс нормально?
- **Context** — Очевидною вартістю використання мікросервісів є збільшення складності. Завдяки більш динамічним деталям у системі, орієнтованій на мікросервіси, постійною проблемою є повне розуміння та усунення проблем із продуктивністю. Поглиблений та рішучий моніторинг стану мікросервісів допомагає зрозуміти загальну роботу системи та виявити окремі точки несправності.
- **Solution—Health check pattern** — Мікросервіси надають кінцеву точку здоров’я (health endpoint) з інформацією про стан послуги. Додаток монітора відстежує ці дані, агрегує їх та представляє результати на інформаційній панелі.

- Рекомендується створити кілька конкретних перевірок стану здоров’я для кожної послуги. Це призводить до кількох гідних переваг.

1. Конкретні перевірки стану дозволяють користувачеві визначити несправність.
2. Вимірювання затримки під час конкретних перевірок стану здоров'я можуть бути використані для прогнозування відключень.
3. Перевірки стану здоров’я можуть мати різний ступінь тяжкості.
   - Critical: сервіс не реагує на будь-які запити.
   - High priority: нижча послуга недоступна.
   - Low priority: нижча послуга недоступна; кешована версія може бути подана.

Classification of frequently used patterns for microservices



Cross-Cutting Concern Patterns

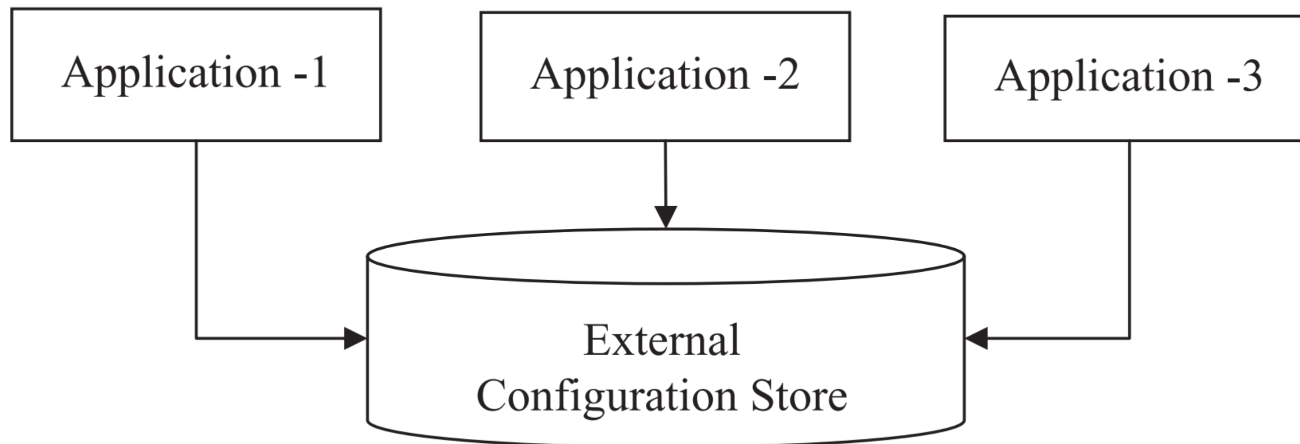
Деякі Cross-cutting concern patterns, що використовуються в MSA, включають:

1. **External configuration store pattern** (Шаблон зовнішнього сховища конфігурацій)
2. **Service registry pattern** (Шаблон реєстру служби)
3. **Circuit breaker pattern** (Шаблон вимикача)
4. **Blue-green deployment** (Синьо-зелене розгортання)

1. External configuration store pattern

- **Problem** — Як уникнути модифікації коду для всіх видів змін конфігурації?
- **Context** — Сервіс зазвичай викликає також інші служби та бази даних. Для кожного середовища, такого як розробка, забезпечення якості (QA), приймальне тестування (UAT) або виробництво, URL-адреса кінцевої точки або деякі властивості конфігурації можуть змінюватися. Для конфігурації розробник зазвичай використовує файл конфігурації (.properties, .xml або інший) всередині програми. Однак цей метод поєднує пакет програми з базовим середовищем, і розробнику потрібно створити окремий пакет для кожного середовища. *Будь-яка зміна будь-якої з цих властивостей автоматично вимагає повторної побудови та перерозгортання служби.* Постійне питання тут полягає в тому, як уникнути модифікації коду для всіх видів змін конфігурації?

- **Solution—External configuration store pattern** — Одним із рішень є *наявність зовнішнього сховища конфігурацій*. Це операційний шаблон, який відокремлює деталі конфігурації від програми. Зовнішня конфігурація, як правило, не знає значення властивостей конфігурації і знає лише властивості, які слід прочитати з сховища конфігурацій. Програми із зовнішнім шаблоном сховища конфігурацій показані на Рис. 8.16.



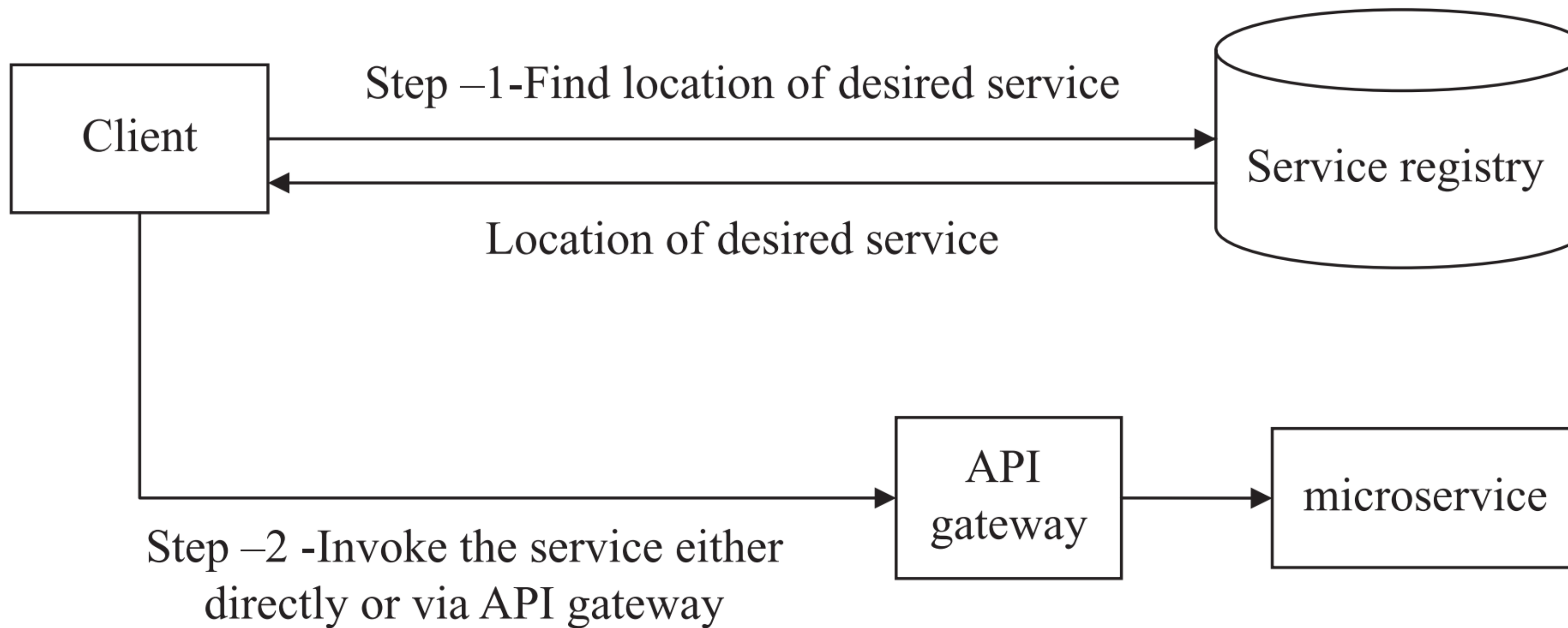
- *Однією з основних переваг цього шаблону є оновлення значень конфігурації без відновлення програми.*
- Після створення пакета він може працювати в будь-якому середовищі без жодних перешкод. Будь-яка команда (інфраструктура чи проміжне програмне забезпечення) може керувати конфігурацією без допомоги розробника, оскільки пакет програм не потребує оновлення.
- Крім того, він централізує всі конфігурації, а різні програми можуть зчитувати властивості конфігурації з того самого місця.

2. Service registry pattern

- **Problem** — Як дозволити споживачам послуг або шлюзам API (API gateways) знати всі доступні екземпляри та розташування послуг?
- **Context** — Мікросервіси та їх екземпляри повинні бути ідентифіковані до їх використання. Оскільки контейнери наразі розглядають як найефективніший інструмент часу виконання мікросервісів, *IP-адреси динамічно призначаються мікросервісам*. Якщо відбувається будь-яке переміщення мікросервісів, то змінюється IP-адреса. *Це дуже ускладнює пошук клієнтом послуг мікросервісів*. Отже, основне питання тут полягає в тому, як дозволити споживачам послуг або шлюзам API знати всі доступні екземпляри та місця розташування послуг?

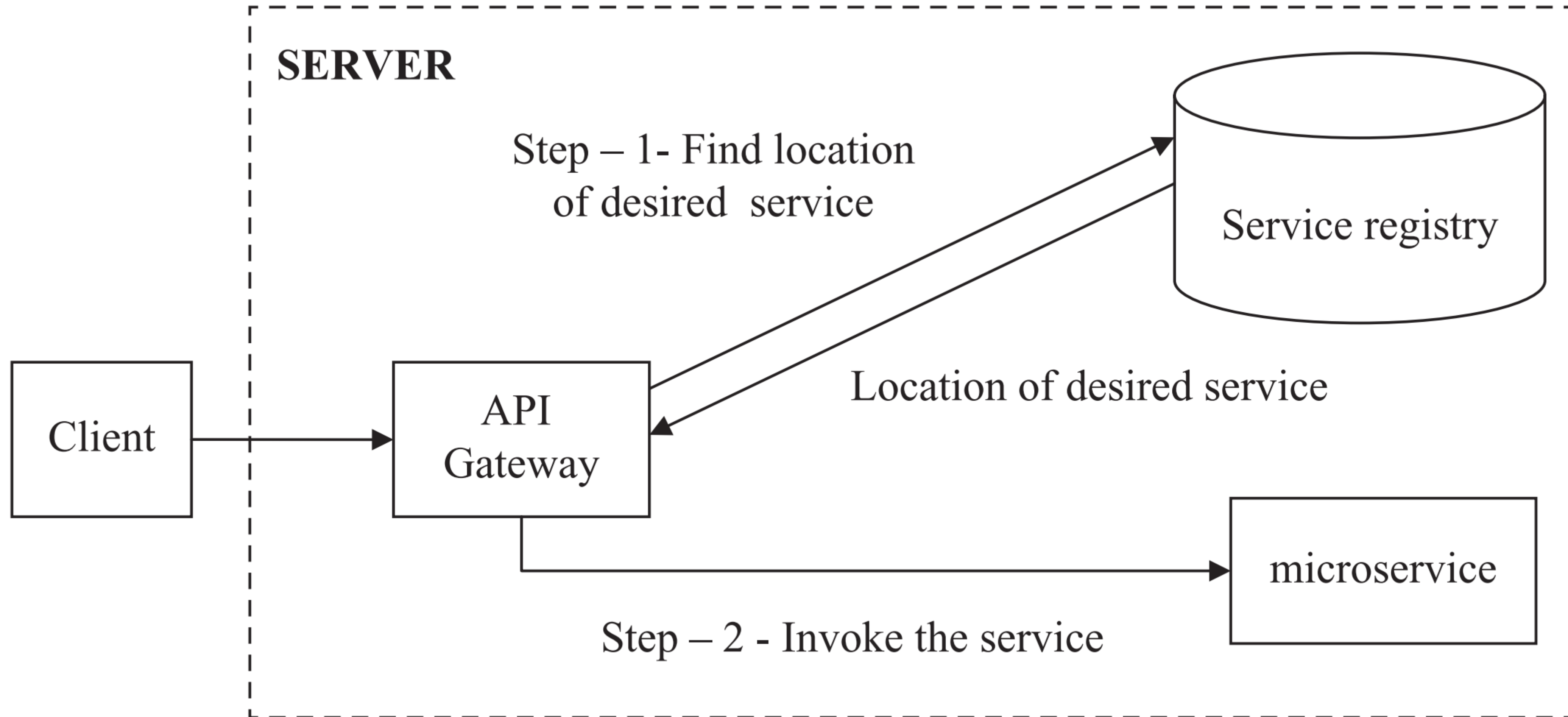
- **Solution—Service registry pattern** — Реєстр сервісів може зберігати метадані кожної служби, що бере участь у взаємодії. *Екземпляр служби повинен зареєструватися в реєстрі під час запуску, а також скасувати реєстрацію під час вимкнення* або переміщення в інше місце. Споживач або API-шлюз / маршрутизатор повинен запитати реєстр, щоб з'ясувати точне місце розташування послуги. Крім того, існує два прийоми, а саме виявлення на стороні клієнта та виявлення на стороні сервера, щоб виявити місцезнаходження бажаної послуги з реєстру. Вони є:

1. **Client-side discovery** — Споживачі послуг зберігають усі місця провайдерів та балансують навантаження між запитами. Потреба полягає у впровадженні процесу виявлення для різних мов / фреймворків, які підтримуються додатком. Виявлення на стороні клієнта показано на Рис. 8.17



Як правило, є два етапи. На кроці 1 клієнт взаємодіє з реєстром служб і запитує в реєстрі розташування потрібної послуги. Реєстр служб повертає клієнту розташування та інші метадані. На кроці 2 клієнт або надсилає запит безпосередньо кінцевій точці служби (якщо в дизайні програми немає шлюзу API), або він подає запит шлюзу API

2. **Server-side discovery**—Споживач надсилає запити на шлюз API, який, у свою чергу, запитує реєстр і вирішує, куди постачальники відправляти. Це показано на Рис. 8.18.



3. Circuit breaker pattern (Шаблон вимикача)

- **Problem** — Як уникнути збоїв у каскаді сервісів та грамотно обробляти їх?
- **Context** — Служба, як правило, викликає інші служби для отримання даних, і існує ймовірність того, що нижча служба може не працювати. З цим є дві проблеми: по-перше, запит буде продовжувати надходити до відключеної служби, виснажуючи мережеві ресурси та сповільнюючи продуктивність. По-друге, взаємодія з користувачем буде поганою та непередбачуваною. Завдання полягає в тому, *як нам уникнути каскадних збоїв сервісу та грамотно обробляти їх*?

- **Solution—Circuit breaker pattern** — Споживач повинен викликати віддалену послугу через проксі-сервер, який працює так само, як електричний вимикач. Коли кількість послідовних збоїв переходить поріг, вимикач спрацьовує (див. Рис. 8.19), і протягом періоду очікування всі спроби викликати віддалену службу негайно зазнають невдачі. Шаблон автоматичного вимикача - це механізм швидкої відмови запитів на unhealthy послуги, а отже, запобігає непотрібному трафіку та каскадним збоям. Після закінчення часу очікування вимикач дозволяє пройти обмежену кількість тестових запитів. Якщо ці запити успішні, автоматичний вимикач відновлює нормальну роботу. В іншому випадку, якщо сталася помилка, період очікування починається знову.

Netflix Hystrix - це хороша реалізація шаблону автоматичного вимикача. Це також допомагає визначити резервний механізм, який можна використовувати під час спрацьовування автоматичного вимикача. Це забезпечує кращу взаємодію з користувачем.

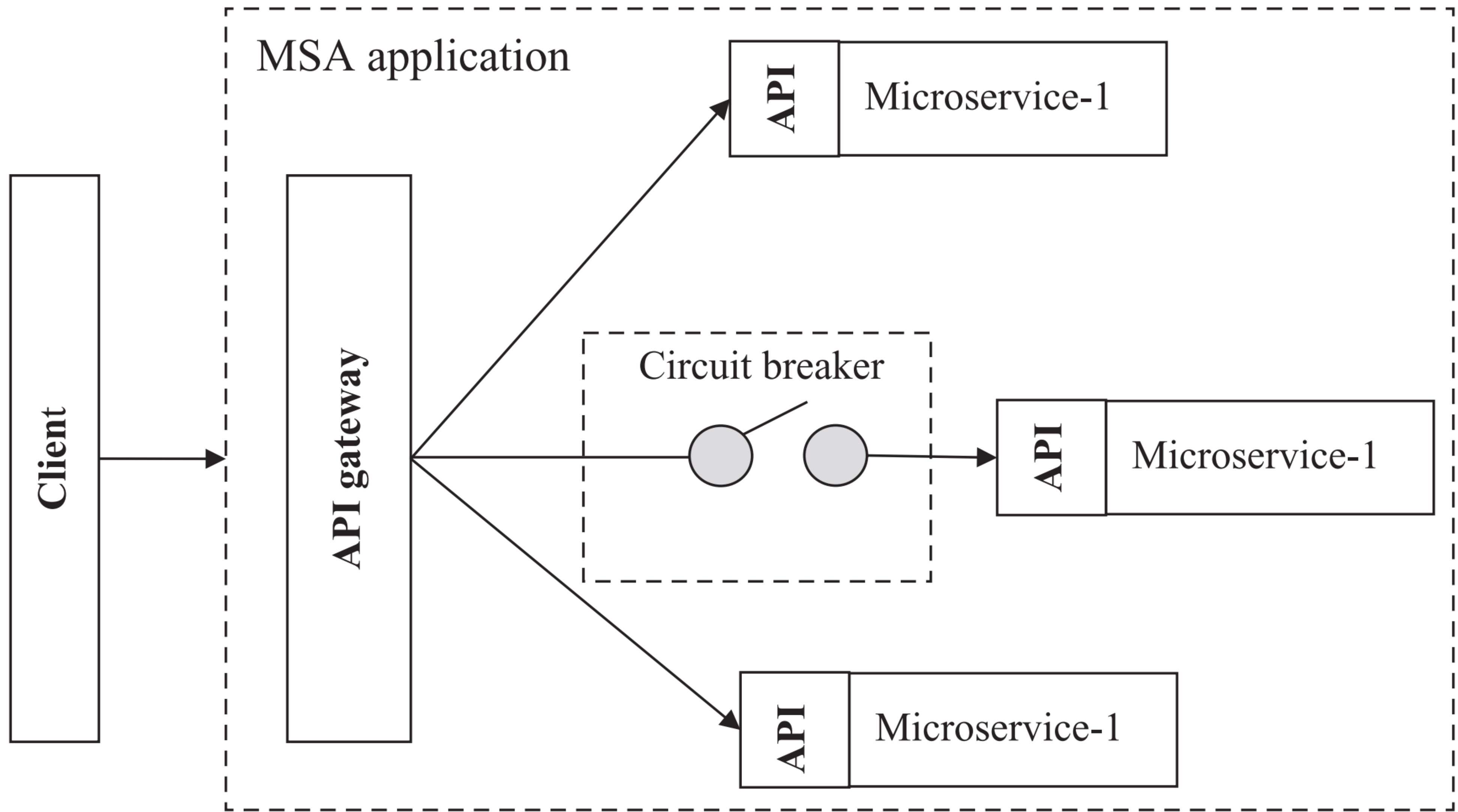
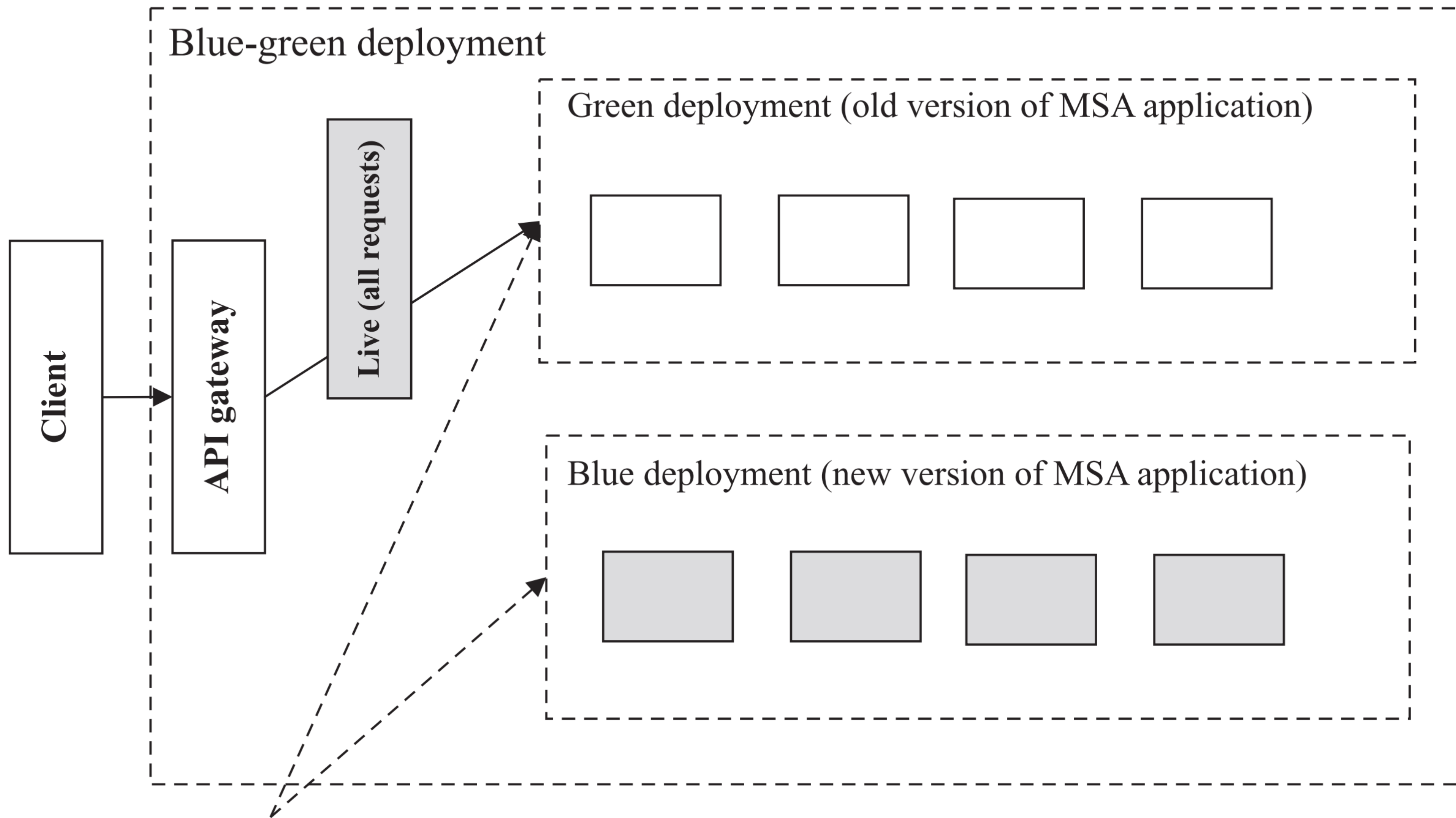


Рис. 8.19 Circuit breaker pattern

4. Blue-green deployment pattern

- **Problem** — Як нам уникнути або зменшити час простою служб під час розгортання?
- **Context** — Завдяки архітектурі мікросервісів одна програма може мати багато мікросервісів. Коли виникає необхідність передислокувати (redeploy) розширену / нову версію мікросервісу, якщо ми припинимо всі служби для передислокації однієї служби, простої будуть величезними і можуть вплинути на бізнес. Проблема полягає в тому, як нам уникнути або скоротити час простою служб під час розгортання?

- **Solution—Blue-green deployment pattern** — Ця стратегія розгортання може бути реалізована для зменшення або усунення простоїв. Ця стратегія розгортання зменшує час простою та ризик завдяки використанню двох однакових виробничих середовищ, іменованих Синім та Зеленим. Припустимо, що Green є існуючим екземпляром, а Blue - новою версією програми. У будь-який час лише одне із середовищ працює в прямому ефірі, причому живе середовище обслуговує весь виробничий трафік. Усі хмарні платформи надають варіанти реалізації синьо-зеленого розгортання. Це показано на Рис. 8.20.


Blue-green deployment
Green deployment (old version of MSA application)
Client
API gateway
Live (all requests)
Blue deployment (new version of MSA application)
Two deployments to
reduce downtime

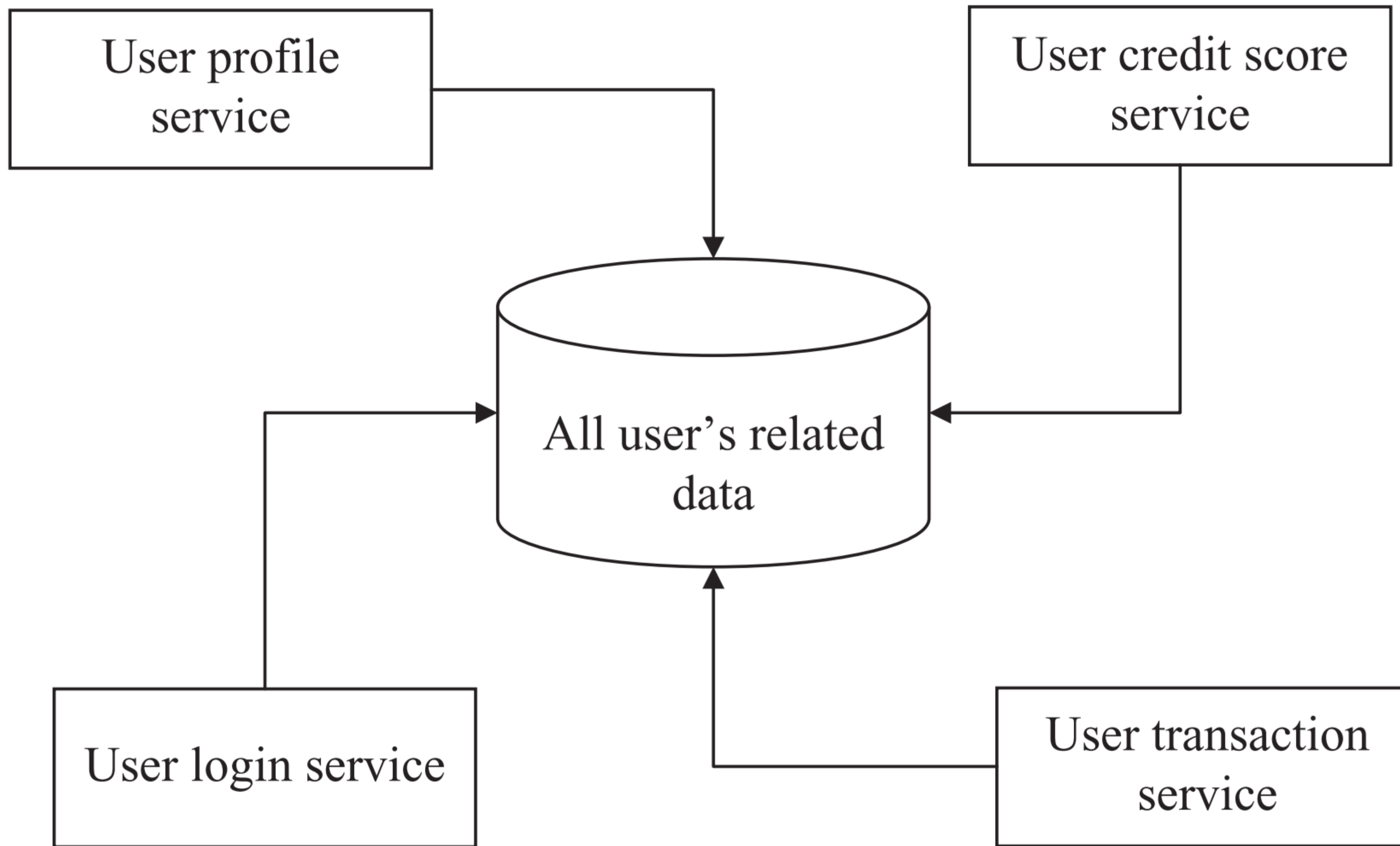
Additional Database Architecture Patterns

Деякі додаткові схеми патерни бази даних для MSA включають:

1. Database Integration pattern
2. Synchronous API calls pattern
3. Asynchronous messaging pattern

1. Database integration pattern

- **Problem** — Як дві або більше служб можуть виконувати операції читання / запису в одній базі даних?
- **Context** — Можуть бути ситуації, коли багатьом службам потрібно отримати доступ до даних із центрального сховища. Наприклад, врахуйте, що дані про логін користувача, профіль користувача, кредитний рейтинг користувача та транзакції користувача можуть використовуватися різними службами. Якщо всі дані пов'язані з інформацією користувача, як різні служби можуть обмінюватися даними користувача?
- **Solution—Database integration pattern** — За цією схемою дві або більше служб зчитують і записують дані з одного центрального сховища даних. Усі служби переходять до цього центрального сховища даних, як на Рис. 8.21.


User profile service
User credit score service
All user's related data
User login service
User transaction service

- Однією з істотних переваг інтеграції баз даних є простота. Управління транзакціями є більш простим у порівнянні з іншими моделями. Це, мабуть, найбільш широко використовувана схема інтеграції.
- Цей шаблон поєднує послуги, що ускладнює управління та обслуговування архітектури мікросервісів. Визначення права власності на дані та оновлення схеми може стати дещо складним. Кожна зміна однієї служби вимагає перекомпіляції та розгортання всіх служб. Це підтримує масштабування вгору / вертикально. Для масштабування ємності бази даних потрібно більше апаратних ресурсів.

2. Synchronous API calls pattern

- **Problem** — Як мікросервіс мже зробити запит або отримати доступ до даних, доступних для інших мікросервісів?
- **Context** — У MSA кожна служба має свої дані. Оскільки атомарні служби пропонують лише обмежену функцію, для надання корисної функції завжди потрібен доступ до даних, пов'язаних із послугою, або запит на них, тобто мікросервіси повинні отримувати доступ до даних, пов'язаних з іншими службами. Оскільки в MSA не дозволяється прямий обмін даними кількома службами, як запитувати або отримувати доступ до даних, доступних для інших служб?

- **Solution—Synchronous API calls pattern** — За цією схемою служби взаємодіють синхронно через API. Весь доступ до даних один одного координується через API за допомогою способу запит-відповідь, і *служба чекає, поки дані з API виконають свою дію*. Розглянемо, наприклад, дві служби, а саме послугу профілю користувача та послугу оцінки кредитних даних користувачів. Служба кредитного рейтингу користувача повинна зчитувати дані профілю користувача. Отже, він викликає службу профілю користувача через свій API і отримує те, що йому потрібно. Це показано на Рис. 8.22.

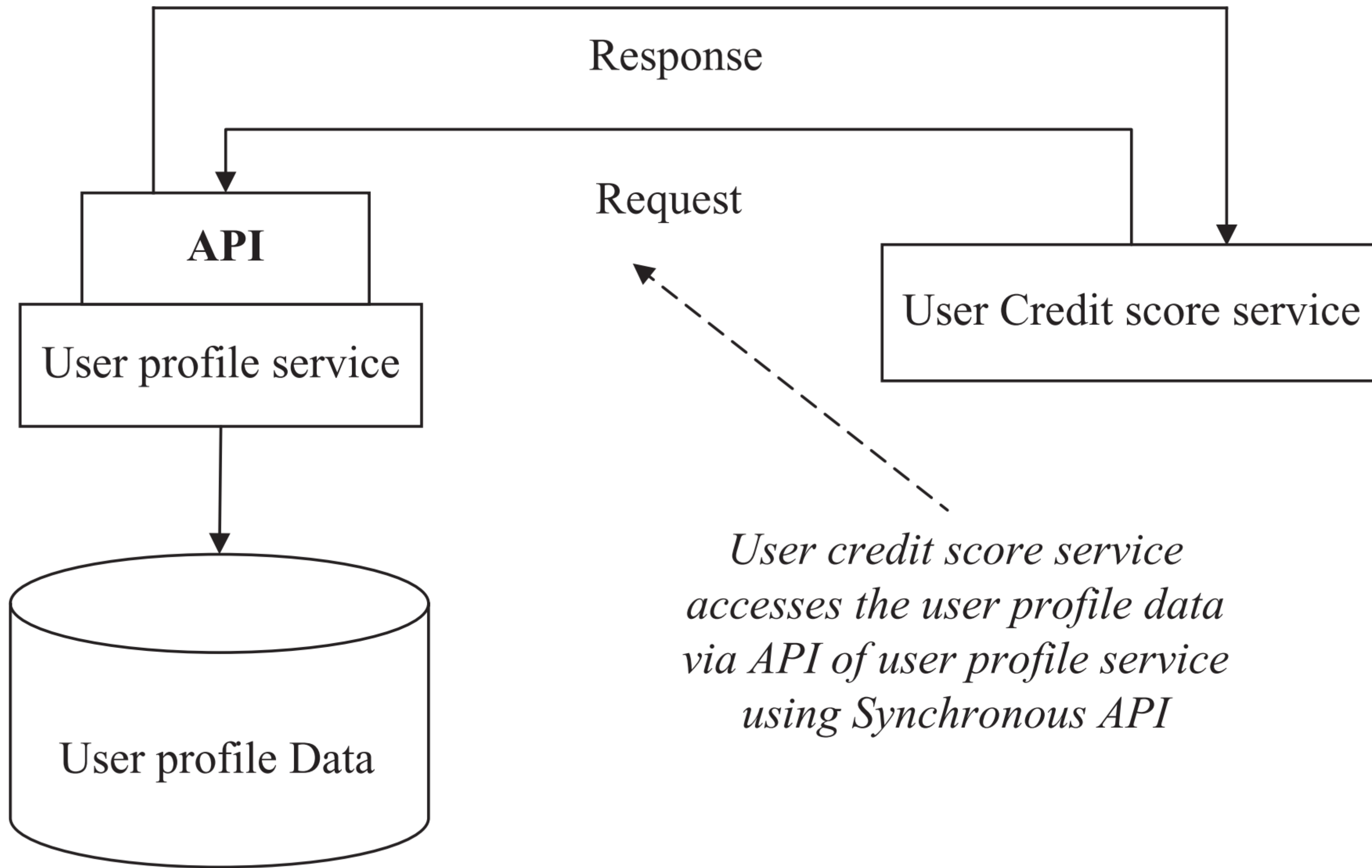


Рис. 8.22 Synchronous API call pattern

- Це забезпечує перевагу приховування багатьох деталей реалізації: абстракція надає розробникам свободу впроваджувати необхідні зміни, а також технології, не зачіпаючи один одного та клієнтів. Наприклад, служба профілю користувача може використовувати Java і MySQL, тоді як служба кредитної оцінки користувачів може використовувати SQL-сервер і .NET, і вони все ще можуть легко спілкуватися між собою за допомогою API.

3. Asynchronous messaging patterns

- **Problem** — Як поділитися подіями, пов’язаними з базою даних, які можуть спричинити дії в інших службах, не блокуючи та не впливаючи на незалежний характер послуг?
- **Context** — У будь-яких додатках події використовуються, щоб передати, які помітні зміни відбулись у програмі. Ці події також викликають дії в інших службах. Як служби інформують про події? Як воно може шукати свої зацікавлені події? Як поділитися подіями, пов’язаними з базою даних, які можуть спричинити дії в інших службах, не блокуючи та не впливаючи на незалежний характер послуг?

- **Solution—Asynchronous messaging patterns** — За цією схемою *служби обмінюються значущими повідомленнями між собою за допомогою так званих команд або подій інтеграції*. Вони надсилаються через ***асинхронних посередників повідомлень***, таких як RabbitMQ, тощо. Асинхронний зв’язок між службами за допомогою асинхронного посередника повідомлень показаний на Рис. 8.23.

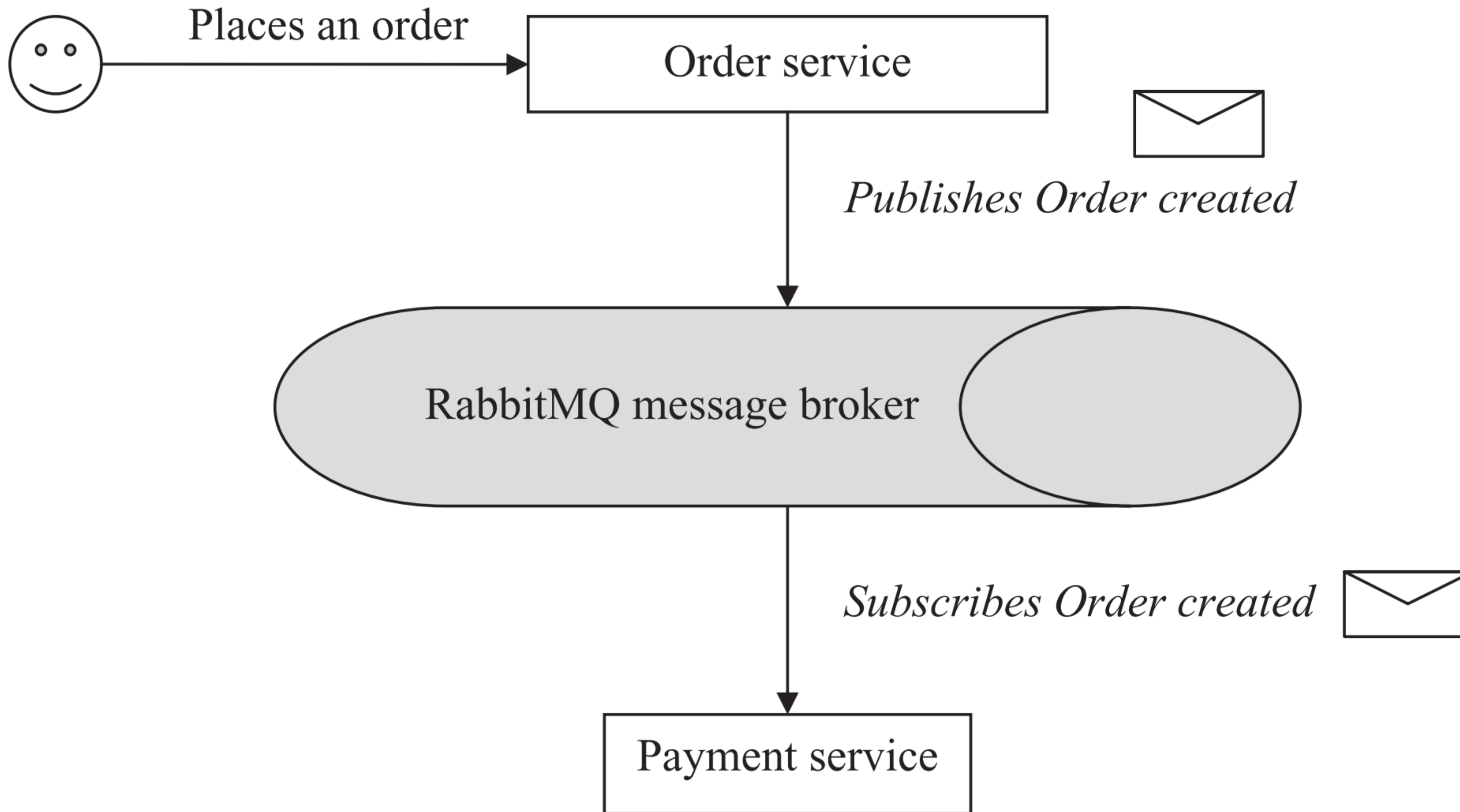


Рис. 8.23 Asynchronous message pattern via a message broker

- Як тільки клієнт розміщує замовлення в службі замовлення, служба замовлення створює замовлення та публікує зміну стану, тобто публікує повідомлення, створене замовленням, за допомогою асинхронного брокера повідомлень RabbitMQ.
- Брокер повідомлень надсилає повідомлення тим службам, які підписалися на це повідомлення. Скажімо, наприклад, платіжна служба передплачує повідомлення, і як тільки вона отримує повідомлення, вона починає дію отримання платежу від клієнта.

- **Two categories of messaging** — Типові рішення обміну повідомленнями будуються на додаток до властивостей транспорту. На високому рівні ці транспорти можна розділити на дві категорії: чергування повідомлень (message queuing) та потокове передавання повідомлень (message streaming).
- Рішення в черзі (queuing solution) *передбачає роботу з реальними даними*. Після успішної обробки повідомлення воно виходить з черги. Поки обробка не відстає, черги не накопичуються і не вимагають занадто багато місця. Однак *порядок надсилання повідомлень не може бути гарантованим* у випадку масштабованих підписників.

- У рішенні для потокового передавання (streaming solution) *повідомлення зберігаються в потоці* у міру їх упорядкування. Це відбувається на самому транспорті повідомлень. Позиція абонента на потоці зберігається на транспорті. За необхідності він може рухатися вперед по потоку. Це дуже вигідно для відмов та нових сценаріїв передплати. Однак це залежить від того, наскільки довгий потік. Це вимагає набагато більше конфігурації з точки зору сховища, що вимагає архівування потоків.

Deployment Patterns

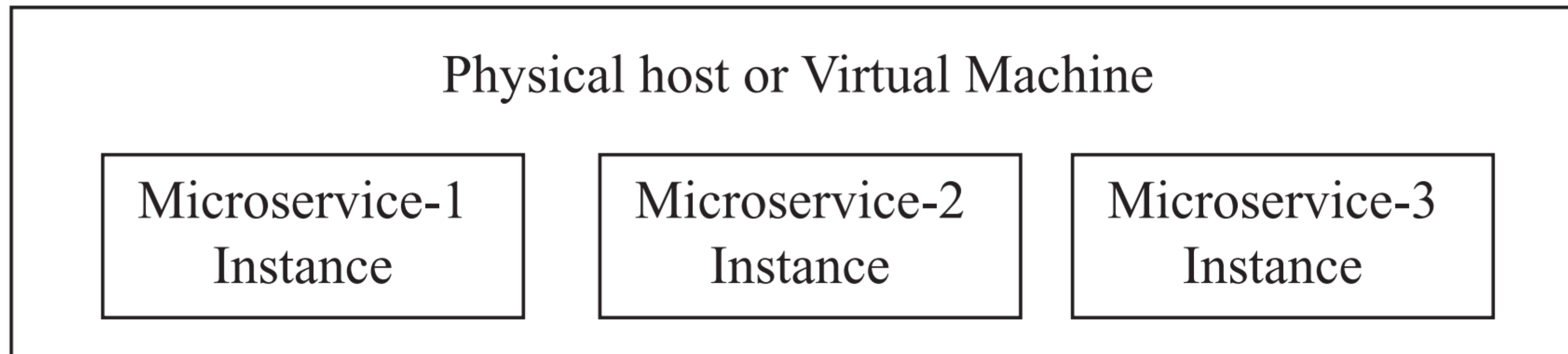
- **Problem** — Як швидко розгорнути мікросервіс з усіма його залежностями?
- **Context** — У монолітну епоху розгортання або масштабування програмної системи - це просто запуск декількох однакових копій програмного забезпечення на фізичні сервери або віртуальний сервер. У випадку застосування мікросервісів існує багато служб, і послуги можуть бути написані різними мовами. Кожна мікрослужба володіє певним ресурсом, даними, а також стратегією моніторингу та розгортання, а також конфігурацією мережі. Отже, проблема полягає в тому: Як розгорнути мікросервіс з усіма його залежностями та вимогами?

- **Solution** — Деякі з моделей розгортання мікросервісів включають:

1. Multiple services instances per host pattern (Шаблон Кілька екземплярів служб на хост).
2. Service instance per host pattern (Екземпляр служби на хост)
   - Service instance per Virtual Machine pattern
   - Service instance per Container pattern
3. Serverless deployment (Безсерверне розгортання).

1. Multiple service instances per host pattern

- За цим шаблоном кілька екземплярів однієї і тієї ж служби розгортаються на одному (або такому ж) хості. Хост може бути фізичним або віртуальним. Це показано на Рис. 8.24.



- Цей шаблон має як переваги, так і недоліки. Однією з головних переваг є *відносно ефективне використання ресурсів*. Кілька екземплярів служб мають спільний доступ до сервера та його операційної системи. Ще однією перевагою цього шаблону є те, що розгортання екземпляра служби відбувається порівняно швидко (зводиться до простого копіювання послуги на хост і запуску її).
- Одним із головних недоліків є *слабка ізольованість сервісів*, якщо кожен екземпляр служби не є окремим процесом. Хоча ви можете точно контролювати використання ресурсів кожного екземпляра служби, не можна обмежувати ресурси, які використовує кожен екземпляр. Неправильний екземпляр служби може споживати всю пам’ять або центральний процесор хоста.

2. Service instance per host pattern

- Інший спосіб розгортання мікросервісів - використання одного екземпляра служби на хосту. За цим шаблоном кожен екземпляр служби працює ізольовано на своєму хості. Існує дві різні спеціалізації цього шаблону, а саме: (i) екземпляр служби на віртуальній машині та (ii) екземпляр служби на контейнер.
- **Service instance per Virtual Machine pattern** — Цей шаблон показаний на Рис. 8.25.

За цим шаблоном послуга упаковується як Virtual Machine image (образ віртуальної машини, VM image). Це означає, що кожен екземпляр служби розгортається як VM, яка запускається за допомогою цього образу VM image.

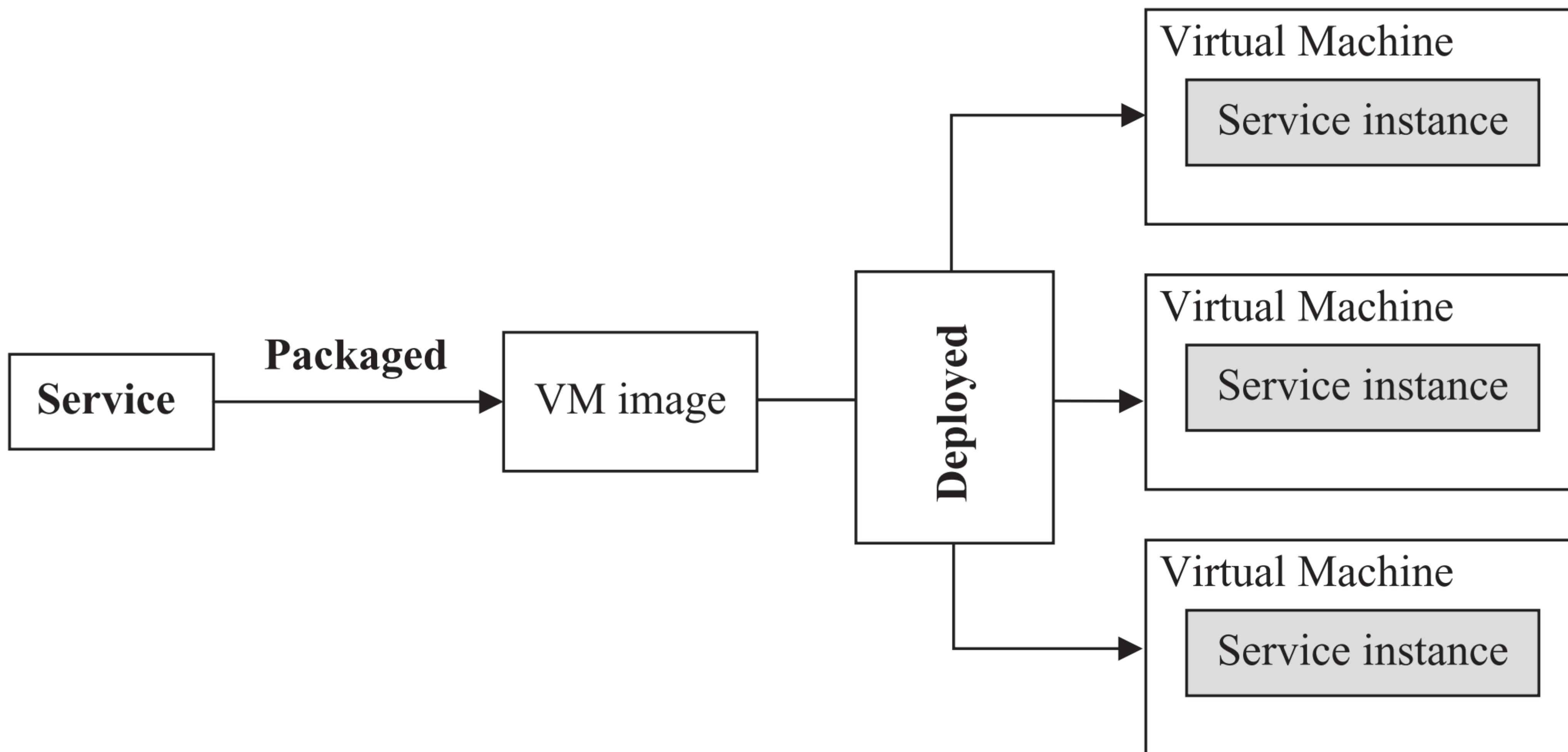


Рис. 8.25 Single instance per Virtual Machine

- **Solution—Service instance per container pattern** — При використанні цього шаблону кожен екземпляр служби працює в окремому контейнері. Контейнери позиціонуються як оптимальне середовище виконання для мікросервісів. Контейнери дають необхідну ізоляцію. Контейнери можуть ходити скрізь. Цей вид розгортання показано на Рис. 8.26.

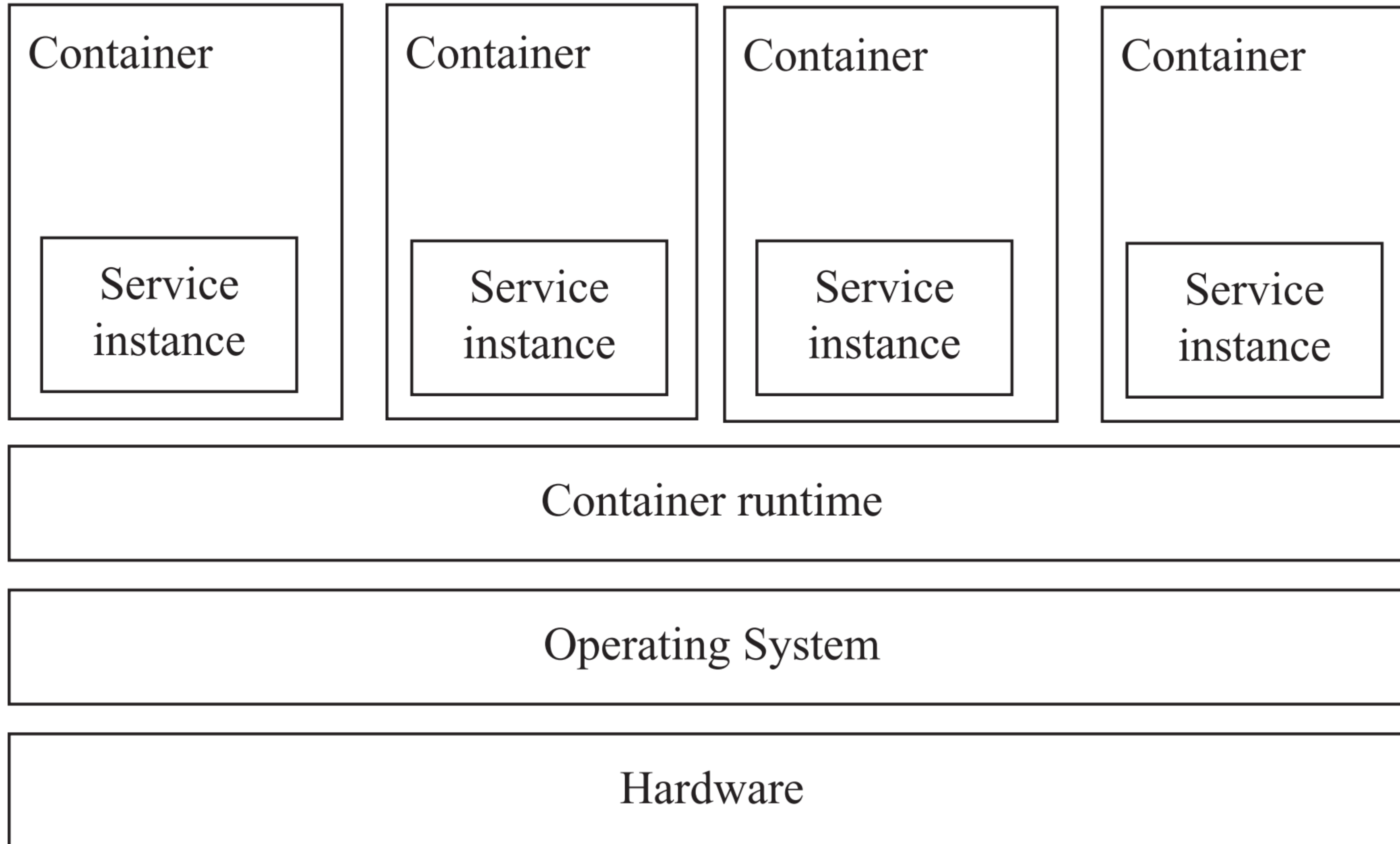


Рис. 8.26 Service instance per container pattern

3. Serverless deployment (Безсерверне розгортання).

- Безсерверне розгортання приховує базову інфраструктуру, і для її запуску знадобиться код мікросервісу. Він стягується відповідно до використання, наприклад (i) скільки запитів він обробив та (ii) скільки ресурсів використовується для обробки кожного запиту. Щоб використовувати цей шаблон, розробникам потрібно упакувати свій код і вибрати бажаний рівень продуктивності. Різні загальнодоступні хмарні провайдери пропонують цю послугу, де для ізоляції служб використовують контейнери або віртуальні машини. У цій системі користувачі не мають права керувати будь-якою інфраструктурою низького рівня, такою як сервери, операційна система, віртуальні машини або контейнери.

Приклади безсерверного середовища розгортання включають:

- AWS Lambda
- Google cloud functions
- Azure functions

Можна викликати безсерверну функцію безпосередньо, використовуючи запит на послугу, або у відповідь на подію, або через шлюз API, або ви можете запускати їх періодично відповідно до cron-подібного розкладу. Деякі переваги безсерверного розгортання:

- Хмарні споживачі можуть зосередитись на своєму коді та застосуванні, і їм не потрібно турбуватися про базову інфраструктуру.

- Споживачам хмар не потрібно турбуватися про масштабування, оскільки воно автоматично масштабується у разі навантаження.
- Споживачі хмарних послуг повинні платити лише за те, що вони використовують, а не за ресурси, надані їм.

Деякі недоліки безсерверного розгортання:

- Багато обмежень, як-от підтримка обмежених мов, більше підходять для служб без збереження стану (stateless).
- Може відповідати лише на запити з обмеженого набору джерел вхідних даних.
- Підходить лише для програм, які можуть швидко запускатися.
- Шанс високої затримки у випадку раптового стрибка навантаження.