Microservices Architecture

Didier Donsez, Vivien Quéma

Laboratoire d’Informatique de Grenoble

ERODS → DRAKKAR

Agenda

- Microservices Architecture
  - Motivations: Monolithic vs Microservice
  - Patterns for microservices
    - Data management
    - Transactional messaging
    - Inter-service communications
    - Service discovery
    - Security
    - Observability
    - Deployment
    - Etc.
  - Case studies : Netflix, Devoxx

- Practices with JHipster
  - Monolith generation
  - Code génération with OpenAPI (swagger).
  - Monolith deployment with Docker
  - Micro-services refactoring and generation
  - Micro-services deployment with Docker
  - Micro-services deployment with Kubernetes on GCP

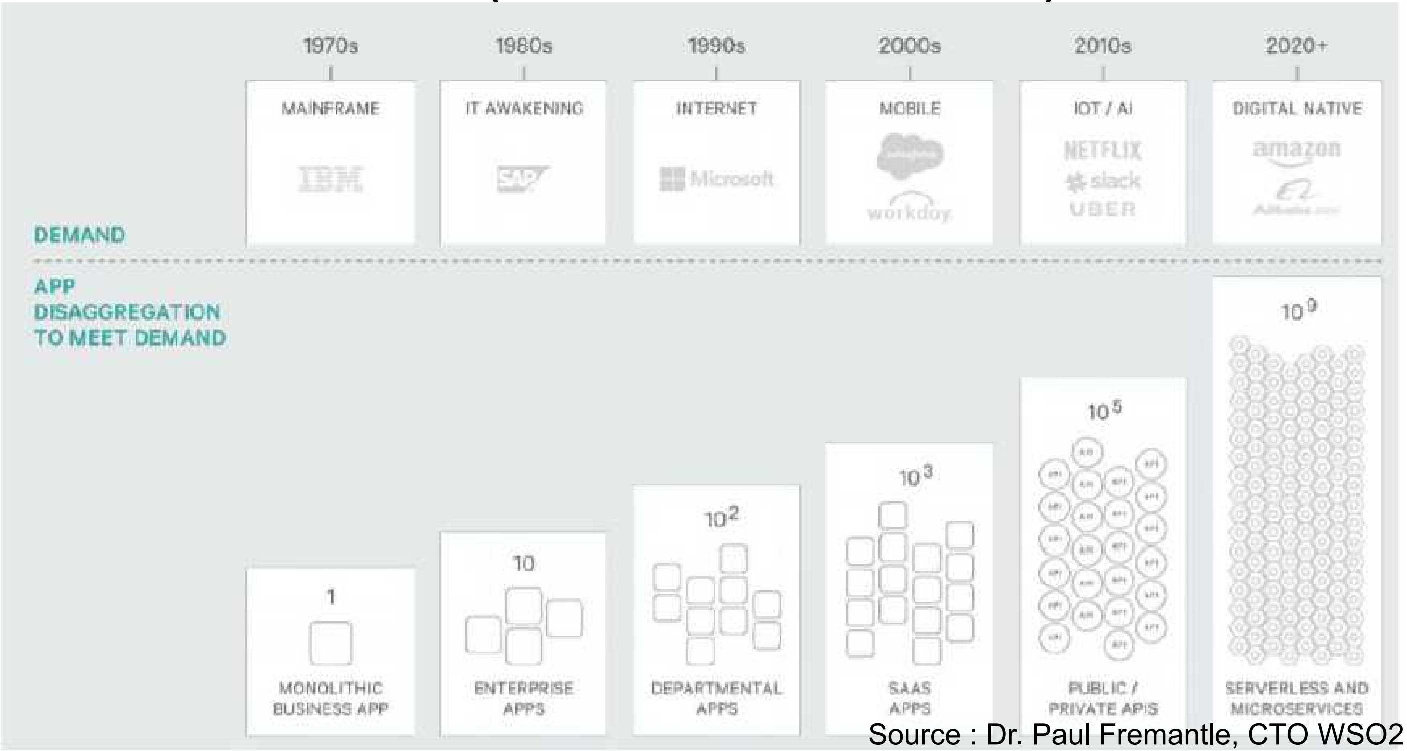
IT Architecture Trends

App “Desagregation” Evolution

FastIT

App “Desagregation” Evolution

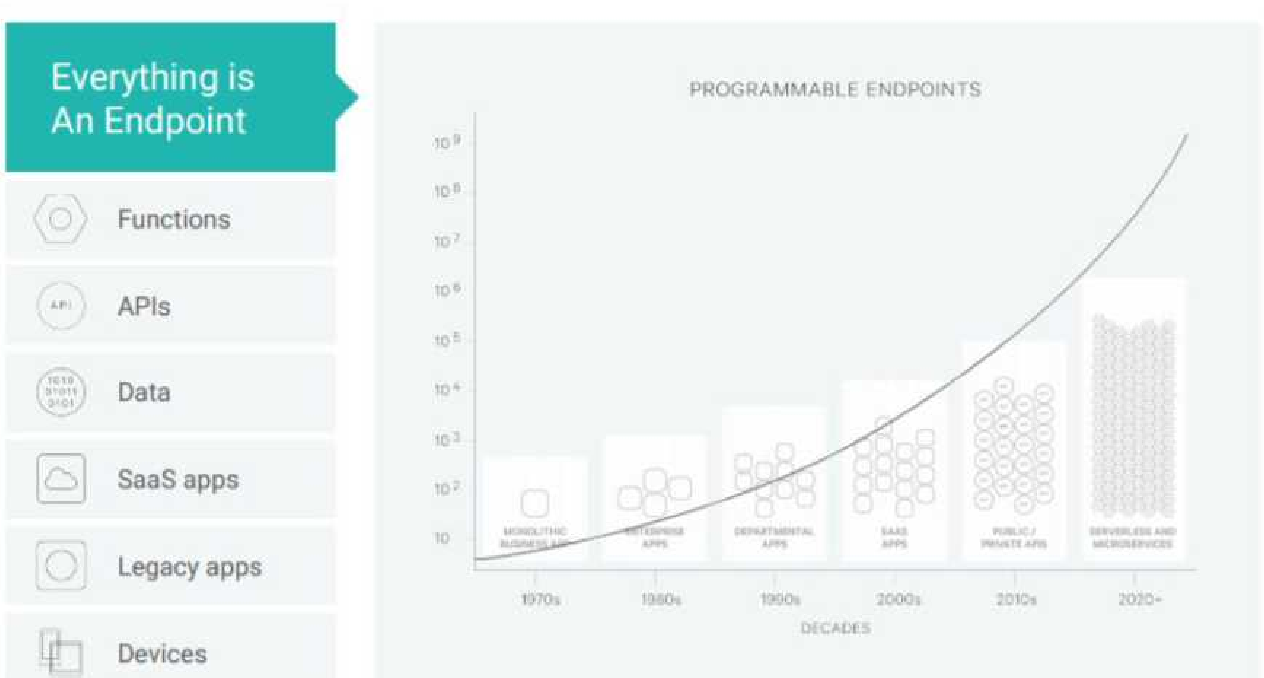
Monolith to Serverless (function as a service)



Source : Dr. Paul Fremantle, CTO WSO2

App “Desagregation” Evolution

More and more endpoints to integrate



Source : Dr. Paul Fremantle, CTO WSO2

FastIT: a motivation for Microservices

- Enterprise IT organization model for bringing the **agility** and the **innovation** required to produce (new) digital services

- Goals :
  - accelerate all phases prior to placing on the market
  - simplify the operational phase
  - opposes long-cycle projects and ITIL-type processes
- Medium: Reorganizing the methods around the product to be delivered
  - Design: lean startup, A/B testing, design thinking, user centric, hackathon, …
  - Development: mockup/prototype, code generation, agility, devops, …
  - Production: on-demand cloud architectures, cloud native applications, **Open API**, **microservices**, ...
- Expectations
  - Minimum Viable Product (MVP)
    - Answering to the functional and qualitative expectations of end-users

Exercice: The cost of the software

Use http://softwarecost.org/tools/COCOMO/ for estimating Effort, Price and Schedule of the development of

- (nominal) software of 20000 loc “from scratch”
- (nominal) software of 20000 loc for a generated boiler plate (1000 reused, 2000 added)
- reliable software such as Linux Kernel, Apache HTTPD, MySQL, Wordpress, Mattermost, Faveo Helpdesk …

Cost per Person-Month (Dollars): France (Paris, Grenoble), UK, Swizterland, India, Morroco, Hong Kong, Shenzen, Madagascar …

The software lifecycle of an artifact/API

Versioning schema (increment policy)

<major>.<mini>[.<micro>][-<qualifier>[-<buildnumber>]]

Major : major changes (except 0 to 1) : no retro-compatibility guarantee

Mini (or Micro): ajouts fonctionnels. retro-compatibility garantie

Micro (or Nano or Patch) : corrective maintenance (bug fix, perf fix)

Qualifiers

alpha1 : alpha version (very unstable and no completed) for dev team

beta1, b1, b2 : beta version (unstable). can be ea

rc1, rc2 : release candidate

m1, m2 : milestone

ea : early access (restricted to a set of volunteers/guinea pigs …)

rtm : release to marketing

lts : long term support (3 – 5 - 10 years)

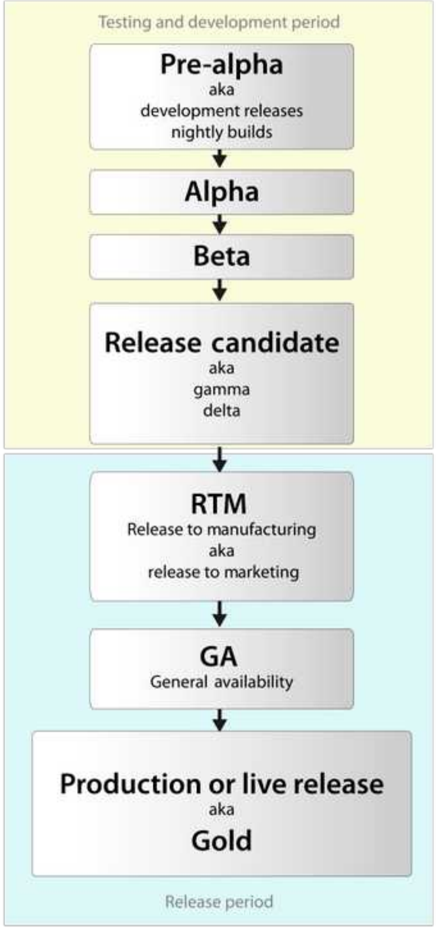
ga : general availability or general acceptance

sp : service pack

SNAPSHOT (Maven) : under construction (before rc1, rc 2 …)

RELEASE : frozen final

See http://en.wikipedia.org/wiki/Software_release_life_cycle et https://semver.org/



The software lifecycle of an artifact/API

Google Gmail

April 1, 2004 (limited beta release). exited the beta status on July 7, 2009.

Windows 10

July 29, 2015 (GA) - October 15, 2025 (official end)

Microsoft Popfly

May 18, 2007 (Beta) - July 16, 2009 (announced) - August 24, 2009 (discontinued)

Google PowerMeter

October 5, 2009 (Beta) - June 2011 (announced) - September 16, 2011 (discontinued)

What are microservices?

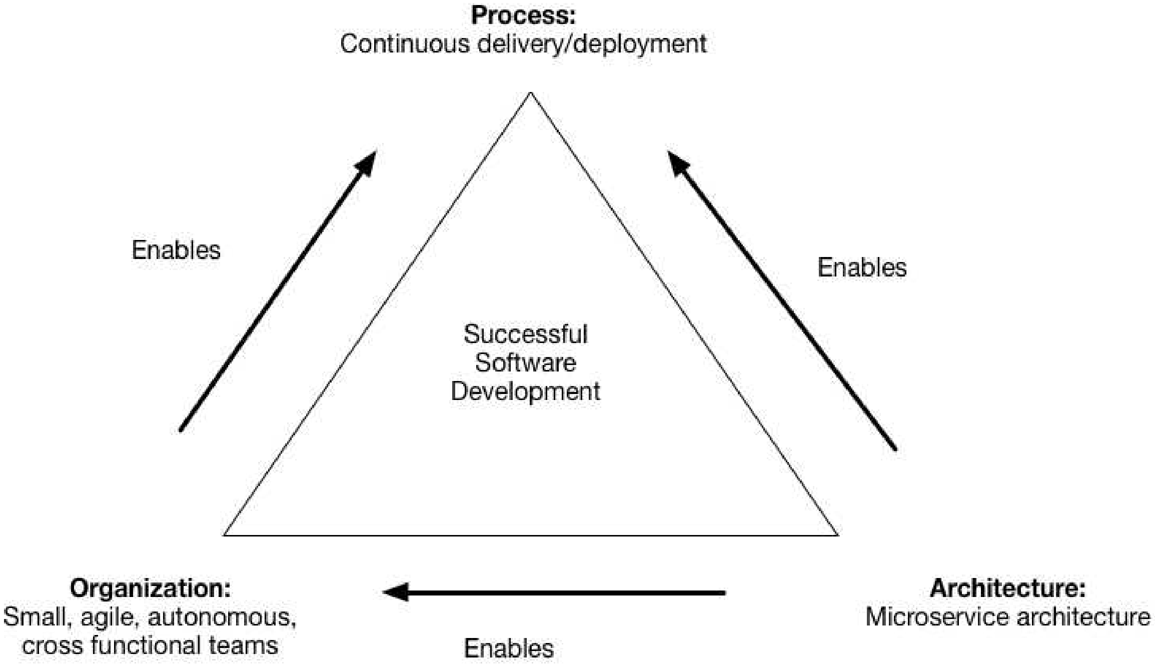
- An architectural style that structures an application as a collection of **loosely coupled services**, which implement **business capabilities**.

- The microservice architecture enables
  - the **continuous delivery/deployment** of **large, complex** applications.
  - an organization to evolve its technology stack

What are microservices?

The microservice architecture:

- Simplifies testing and enables components to be deployed **independently**
- Structures the engineering organization as a collection of **small** (6-10 members*), **autonomous teams**, each of which is **responsible** for one or more services


Process:
Continuous delivery/deployment
Enables
Enables
Successful Software Development
Organization:
Small, agile, autonomous, cross functional teams
Enables
Architecture:
Microservice architecture

aka Two-pizza team

Monolithic vs Microservice Architecture

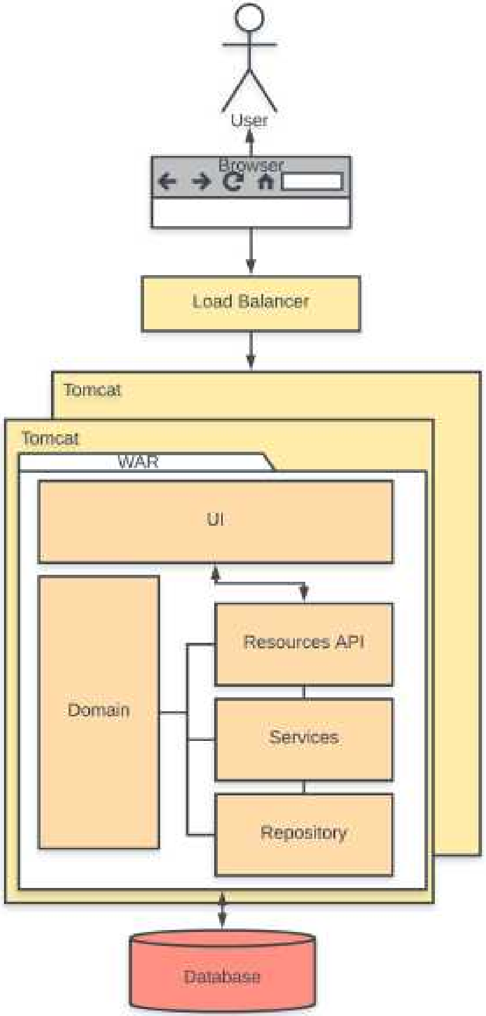
- Example of a server-side enterprise application:
  - Handles requests (HTTP requests and messages) by executing business logic;
  - Accesses a database;
  - Exchanges messages with other systems;
  - Returns a HTML/JSON/XML/Protobuf/FlatBuffers response
- The application:
  - Must support a variety of **different clients** including desktop browsers, mobile browsers and native mobile applications.
  - Might expose an **API for 3rd parties** to consume.
  - Might also integrate with other applications (internal or 3rd parties) via either web services or a message broker.

Monolithic vs Microservice Architecture - Requirements

- New team members must quickly become productive
- The application must be easy to understand and modify
- Practice continuous deployment of the application
- Run multiple copies of the application on multiple machines in order to satisfy scalability and availability requirements
- Take advantage of emerging technologies (frameworks, programming languages, etc)

Monolithic architecture

- Examples of monolithic architectures:
  - a single Java WAR file
  - a single directory hierarchy of Rails xor NodeJS code
  - +
  - a relational database (Postgres, MySQL) xor
    a NoSQL database (MongoDB)

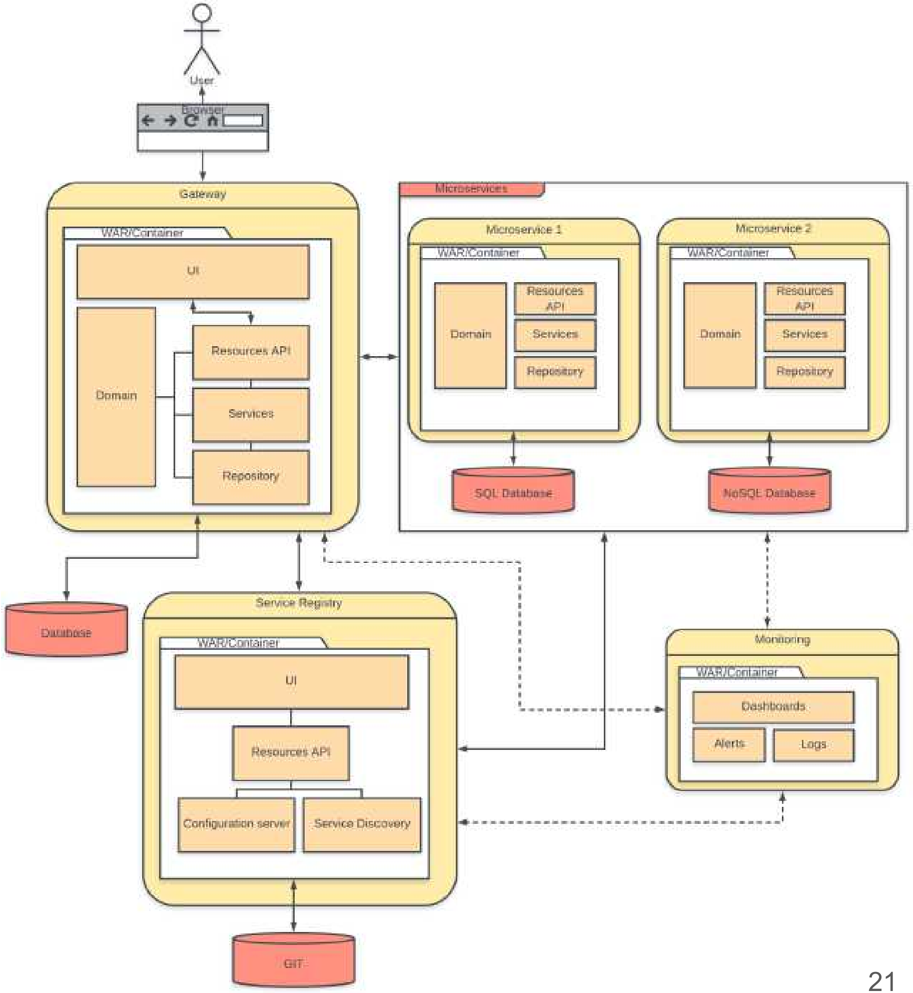


Monolithic architecture - Drawbacks

- Large monolithic code base
- Continuous deployment is difficult
- Scaling the application can be difficult
- Slow web container startup
- Obstacle to scaling development
- Requires a **long-term commitment** to a technology stack

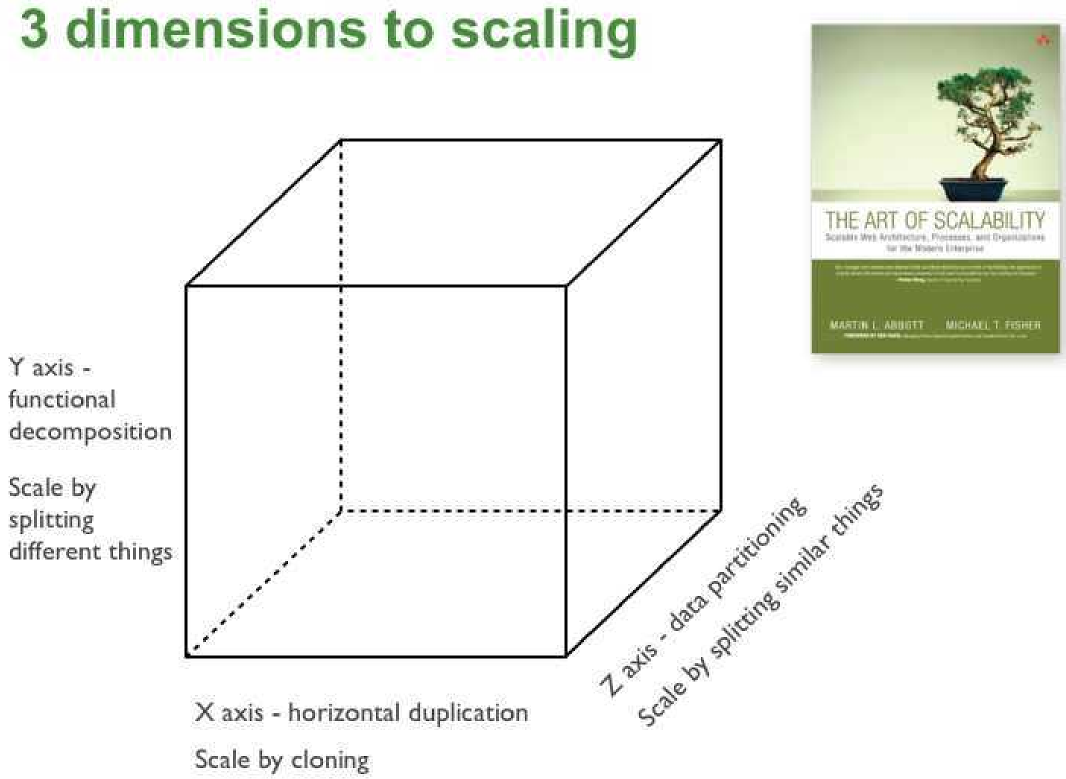
Microservice architecture

- The application is structured as a set of **loosely coupled, collaborating services**
- Each service implements a set of **narrowly, related functions**
- Services communicate using either:
  - **synchronous** protocols such as HTTP/REST
  - or **asynchronous** protocols such as AMQP.
- Services can be developed and deployed independently of one another
- Each service has its own database in order to be decoupled from other services



The Scale Cube

Three dimension scalability model


3 dimensions to scaling
Y axis - functional decomposition
Scale by splitting different things
X axis - horizontal duplication
Scale by cloning
Z axis - data partitioning
Scale by splitting similar things
THE ART OF SCALABILITY
Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise
MARTIN L. ABBOTT
MICHAEL T. FISHER

Microservice architecture: benefits

- Enables the continuous delivery and deployment of large, complex applications
  - Better testability
  - Better deployability
  - Autonomous teams
- Each microservice is (relatively) small
  - Easier to understand
  - The application starts faster
  - Improved fault isolation.
- Eliminates any long-term commitment to a technology stack

Microservice architecture: drawbacks

- Additional complexity of creating a distributed system.
  - Testing
  - Inter-service communication mechanism
  - Distributed transactions
  - Data redundancy
- Deployment complexity
- Increased memory consumption

When to use the microservice architecture?

Depends on

- application scope
- team size
- team skill
- time to market
- infrastructure manpower
- user base

Choosing a monolithic architecture

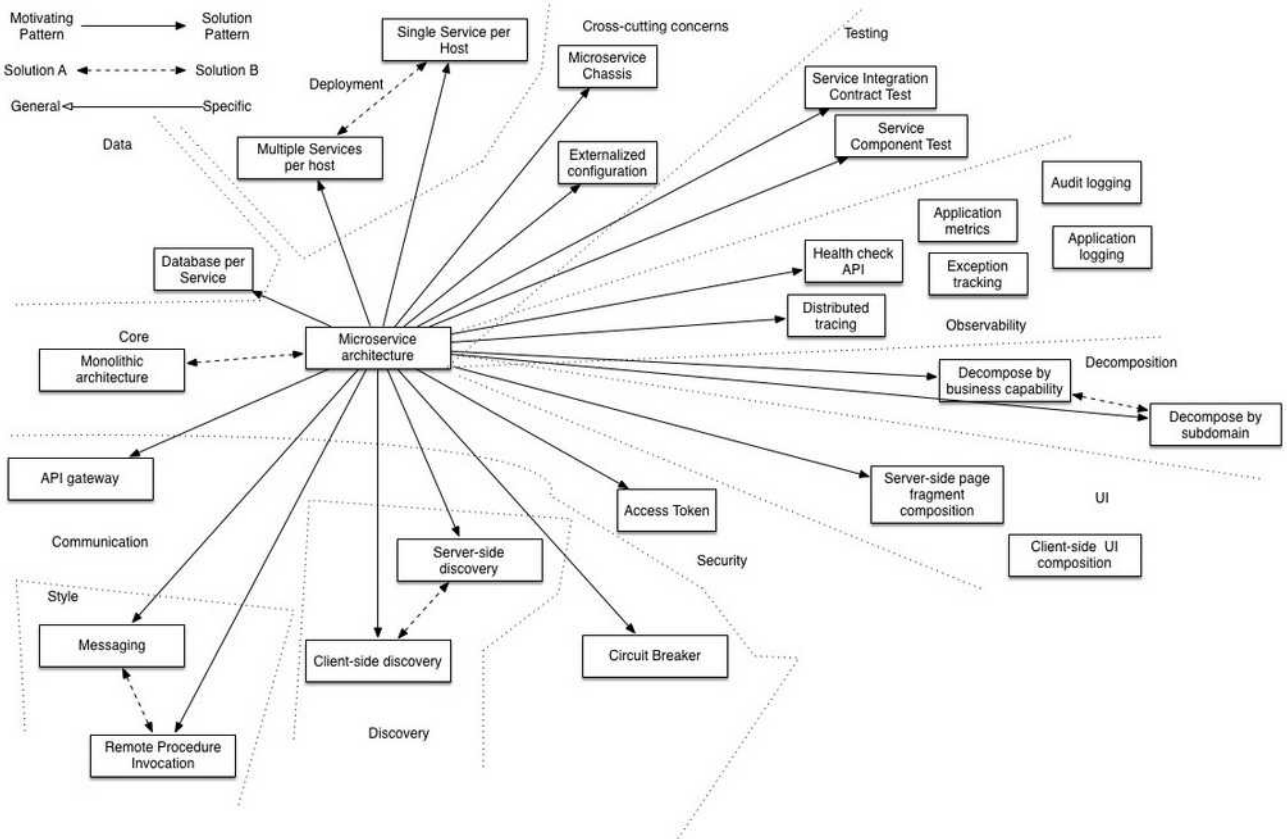
- application scope : small and well-defined and remains simple
- team size : small (up to 8 peoples)
- team skill : novice and intermediate
- time to market : critical
- infrastructure manpower : do not want to spend time
- user base : small or specific set of users in the enterprise app

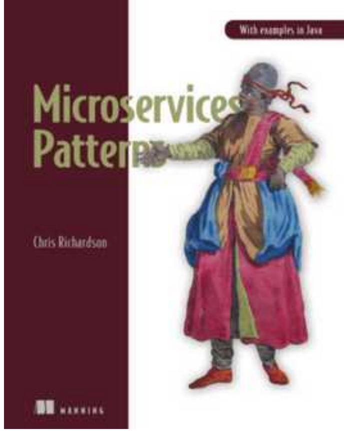
Choosing a microservice architecture

- application scope : large and well defined
- team size : large
- team skill : good and confident in advanced MS patterns
- time to market : not critical, long-term vision
- infrastructure manpower : spend time on infra and in monitoring
- user base : huge or growing

Microservice architecture - 101 patterns

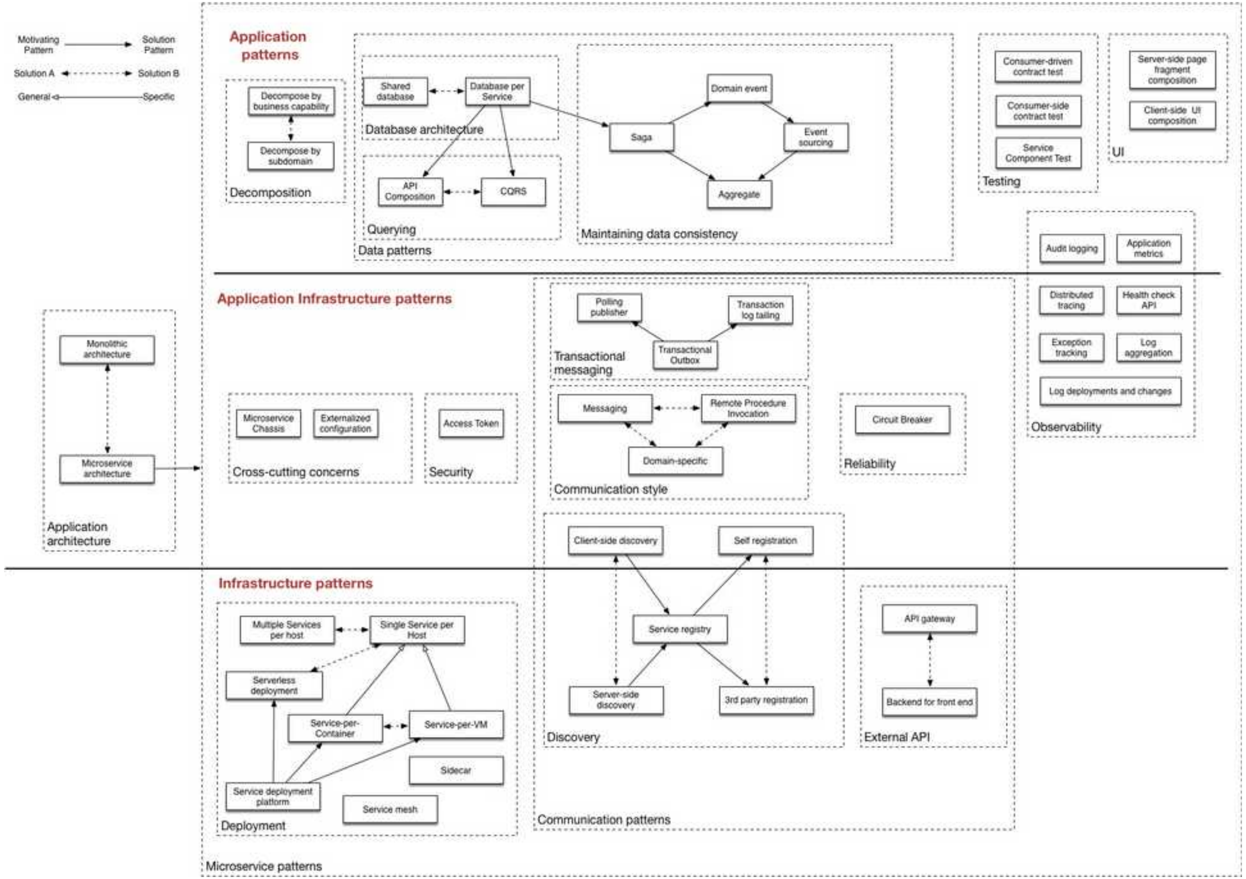
https://microservices.io/patterns

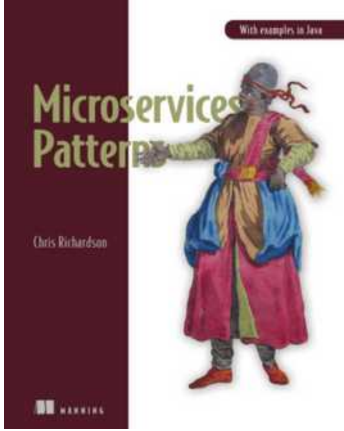




Microservice architecture - 101 patterns

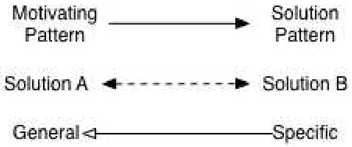
https://microservices.io/patterns

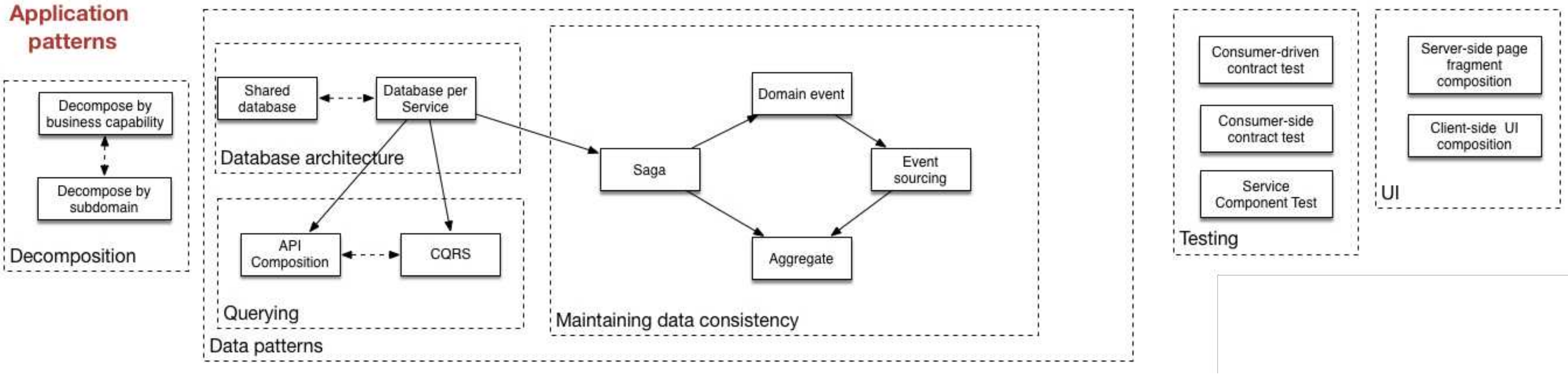




Microservices Patterns

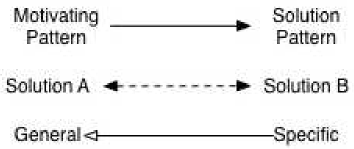
Application Patterns





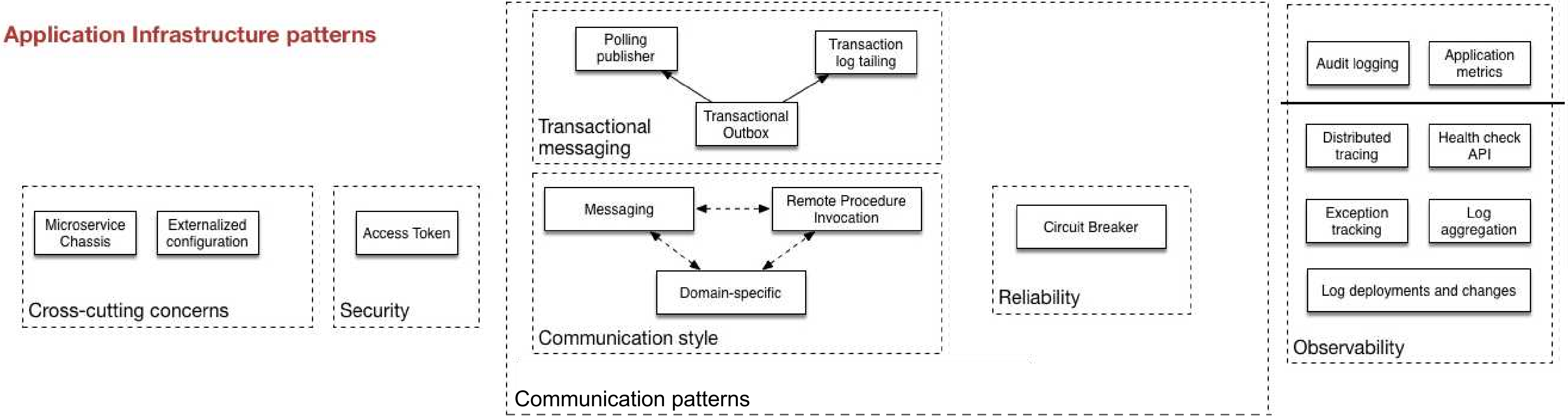
Microservices Patterns

Application Infrastructure Patterns


Motivating Pattern → Solution Pattern

Solution A ↔ Solution B

General ◁ Specific


Application Infrastructure patterns

Microservice Chassis

Externalized configuration

Cross-cutting concerns

Access Token

Security

Polling publisher

Transaction log tailing

Transactional Outbox

Transactional messaging

Messaging

Remote Procedure Invocation

Domain-specific

Communication style

Circuit Breaker

Reliability

Communication patterns

Audit logging

Application metrics

Distributed tracing

Health check API

Exception tracking

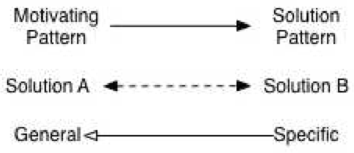
Log aggregation

Log deployments and changes

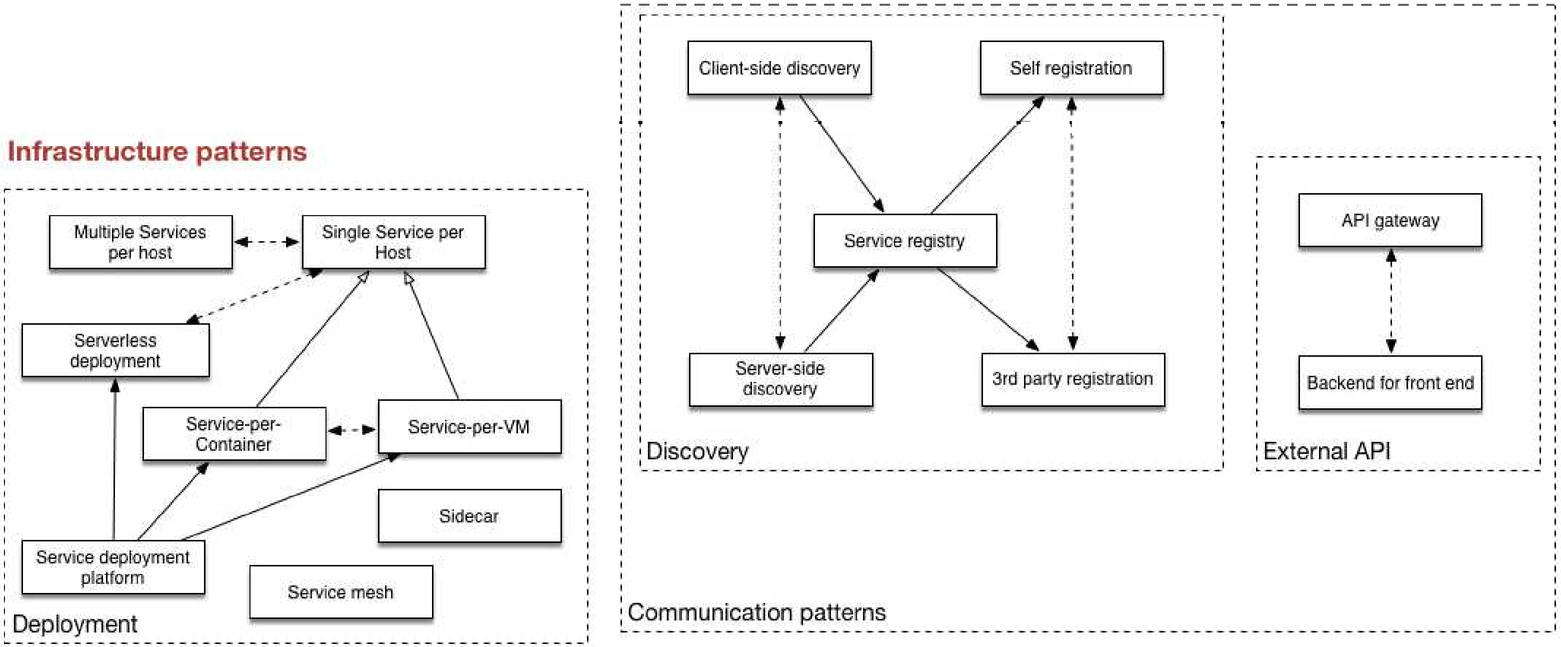
Observability

Microservices Patterns

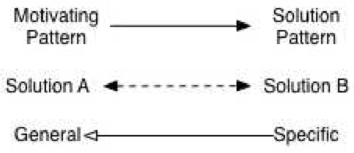
Infrastructure Patterns

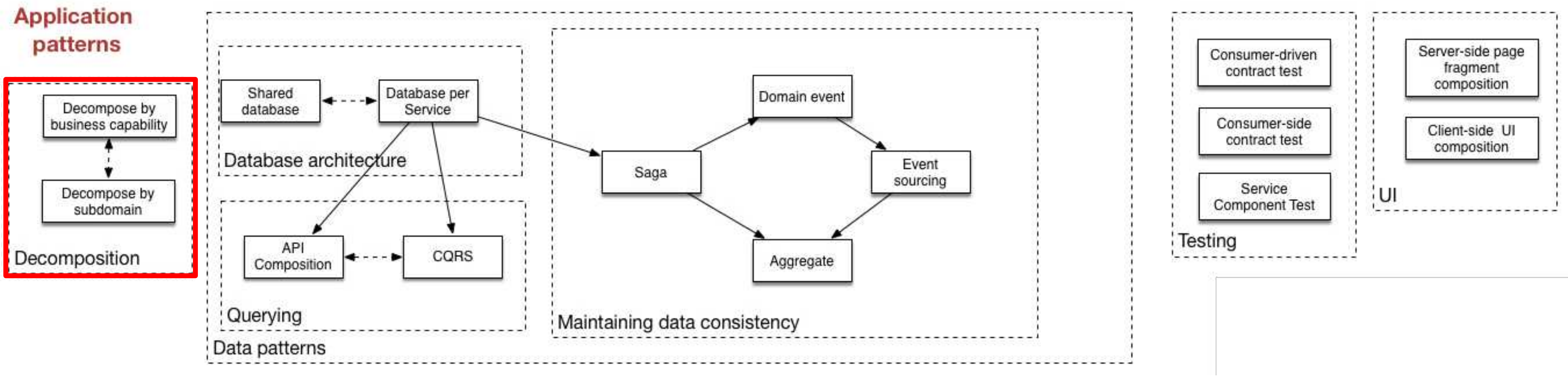


Infrastructure patterns



Microservices Patterns Decomposition





How to decompose the application into services?

- Requirements:
  - The architecture must be stable
  - Services must be cohesive
  - Services must conform to the Common Closure Principle
  - Services must be loosely coupled
  - Services should be testable
  - Services should be small
  - Development teams should be autonomous

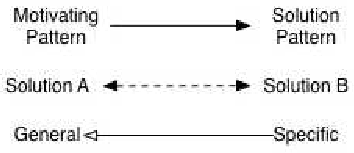
How to decompose the application into services?

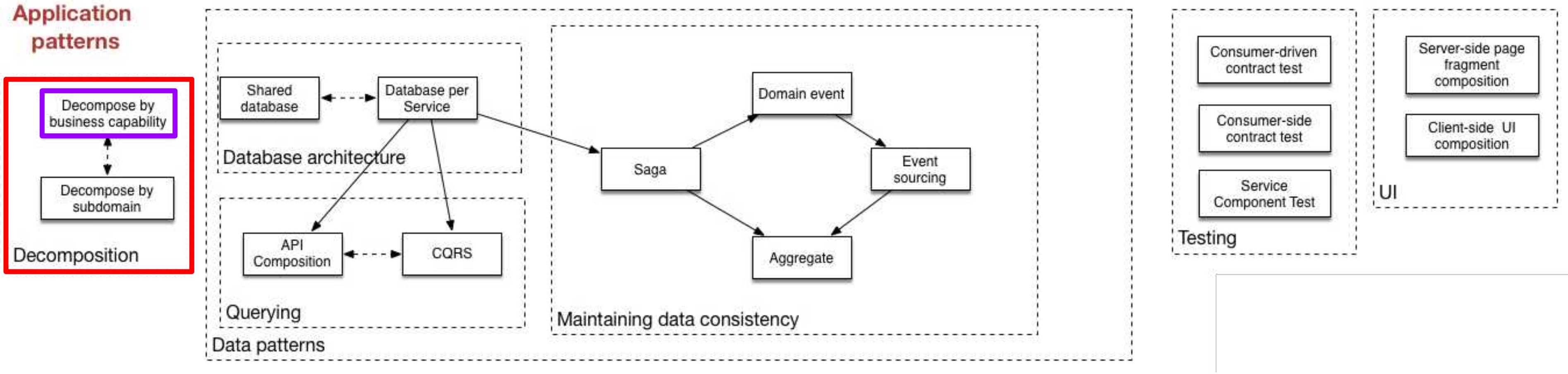
Two strategies exist:

- Decompose by business capability
- Decompose by domain-driven design subdomain

Enforce the SRP (Single Responsibility Principle) pattern

Microservices Patterns Decomposition





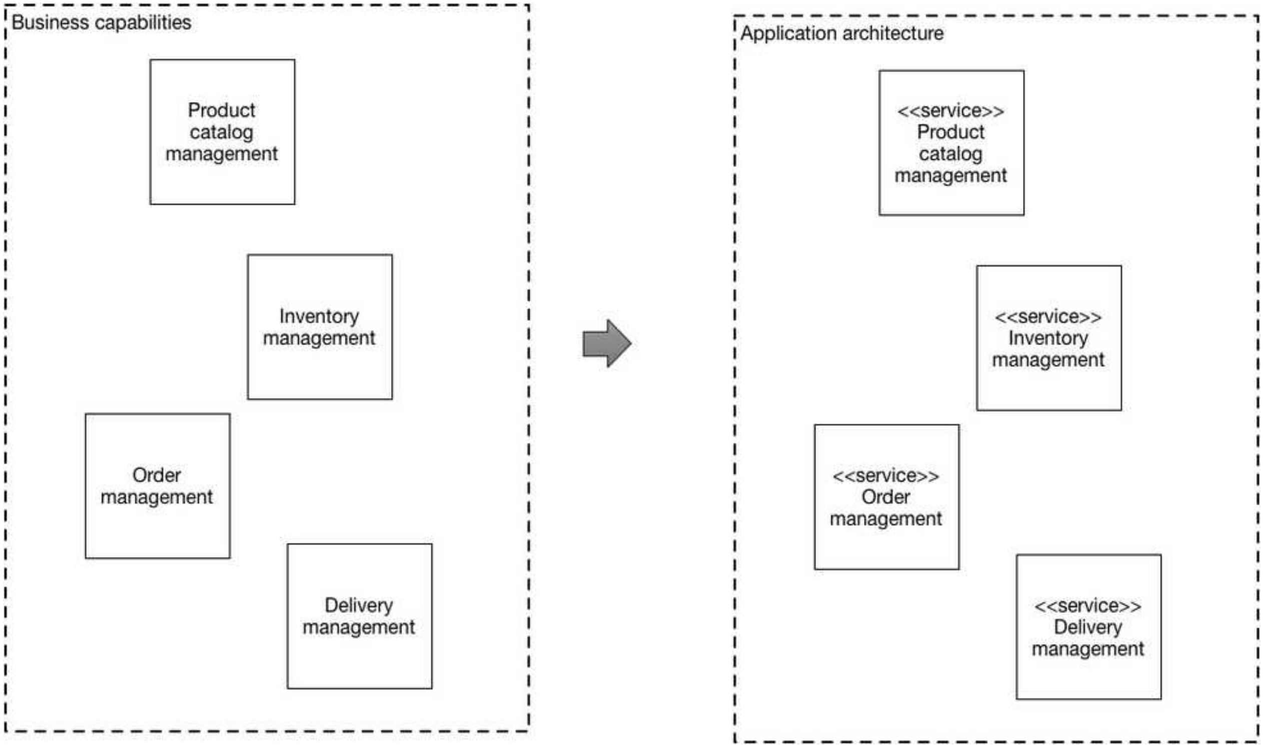
Decompose by business capability

Definition: *A business capability is a concept from business architecture modeling. It is something that a business does in order to generate value.*

Example:

- *Order Management* is responsible for orders
- *Customer Management* is responsible for customers

Decompose by business capability


Business capabilities
Product catalog management
Inventory management
Order management
Delivery management
Application architecture
<<service>> Product catalog management
<<service>> Inventory management
<<service>> Order management
<<service>> Delivery management

Decompose by business capability

- Products management
- Cart management
- Shipping Management
- Order Management
- Payment management
- Shipping Management
- Marketing Content Management
- Notifications (Email/SMS) Management

Decompose by business capability

Subcapabilities

Order management

- Order processing
- Invoice Management

Shipping Management

- Order Tracking
- Fulfillment

Marketing Content Management

- Content Management
- Campaign Management
- Discount Coupons Management
- Email/SMS Management

Decompose by business capability

- Result in
    - Products Service
    - Inventory Service
    - Shopping Cart Service
    - Ordering Service
    - Shipping Service
    - Payment Service
    - Invoice Service
    - Communication Service
    - Shipment Tracking & fulfillment Service
    - Content Service
    - Coupon Management Service

Decompose by business capability

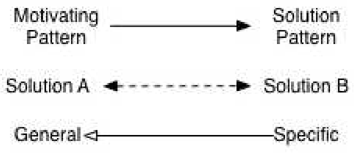
Advantages:

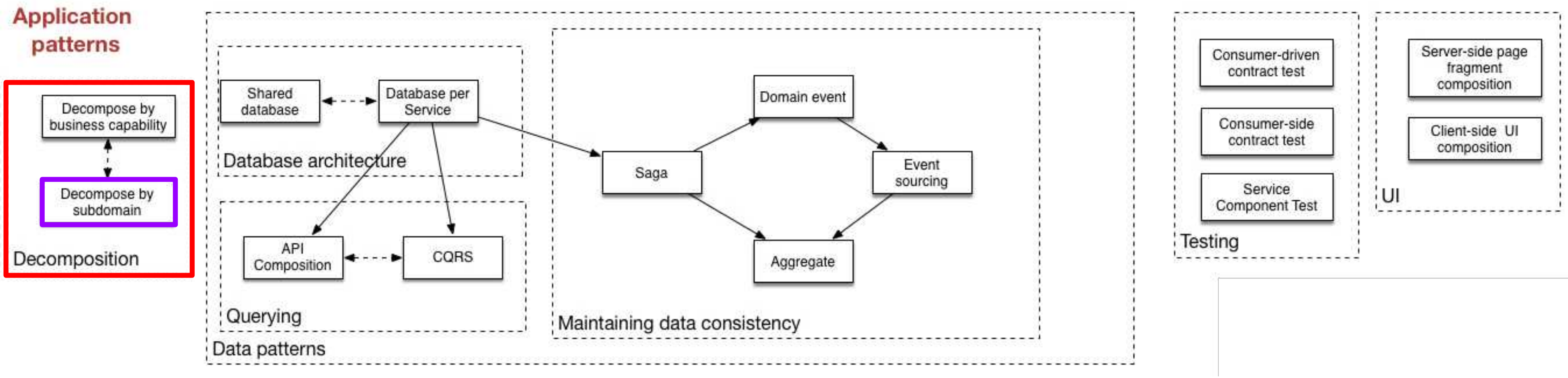
- Stable architecture since the business capabilities are relatively stable
- Development teams are organized around delivering business value rather than technical features
- Services are cohesive and loosely coupled

Issues:

- Identifying business capabilities is sometimes difficult

Microservices Patterns Decomposition





Decompose by subdomain

Define services corresponding to Domain-Driven Design (DDD) subdomains. Subdomains can be classified as follows:

- Core - key differentiator for the business and the most valuable part of the application
- Supporting - related to what the business does but not a differentiator.
- Generic - not specific to the business

Decompose by subdomain

Advantages:

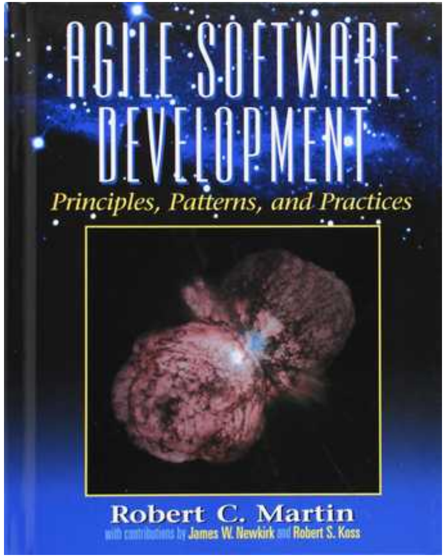
- Stable architecture since the subdomains are relatively stable
- Development teams are cross-functional, autonomous, and organized around delivering business value rather than technical features
- Services are cohesive and loosely coupled

Issues

- Identifying subdomains can be difficult

Single responsibility pattern

Defined in 2006 by Robert C. Marting, a.k.a. Uncle Bob, in the book
Agile Principles, Patterns, And Practices in C#

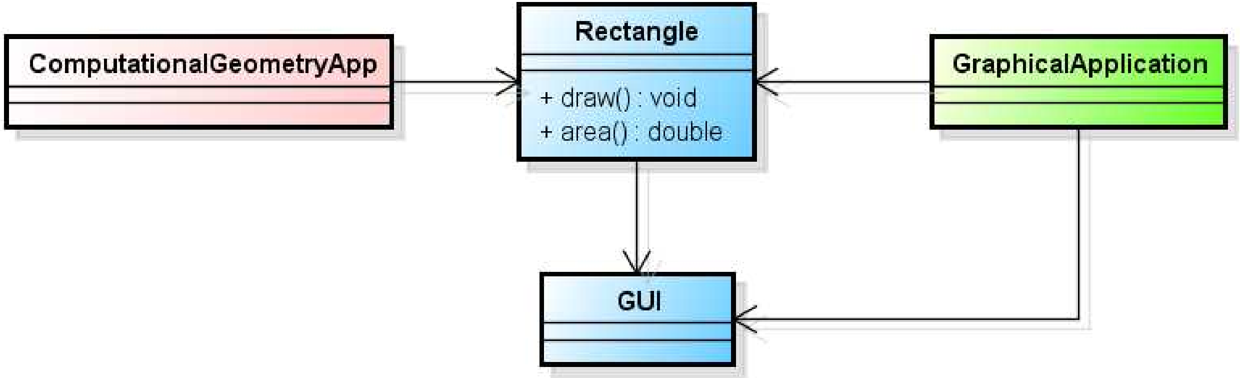

AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss

Single responsibility pattern

Every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class.

SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

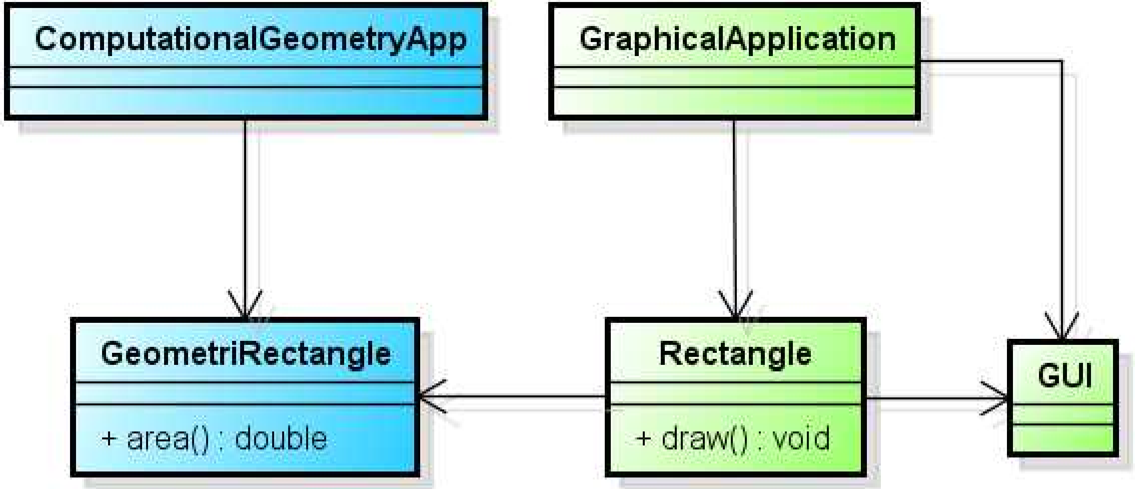
Single responsibility pattern



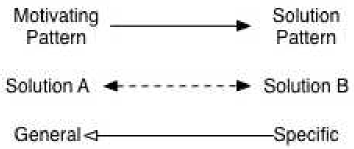
at least *two responsibilities*:

1) drawing a rectangle on a GUI
2) calculating the area of that rectangle.

Single responsibility pattern



Microservices Patterns
Data patterns



Motivating Pattern

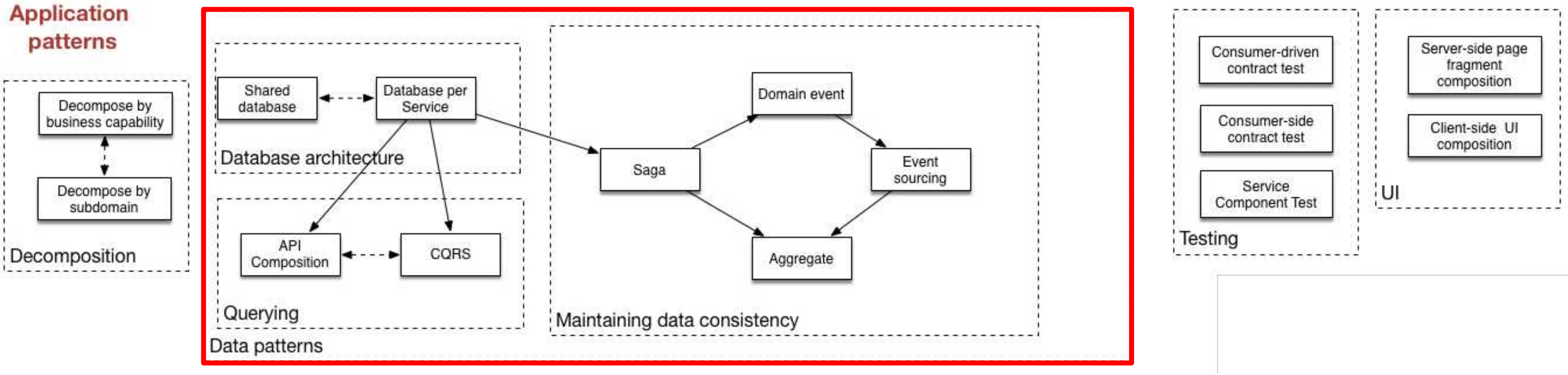
Solution Pattern

Solution A

Solution B

General

Specific



Application patterns

Decompose by business capability

Decompose by subdomain

Decomposition

Shared database

Database per Service

Database architecture

API Composition

CQRS

Querying

Domain event

Saga

Event sourcing

Aggregate

Maintaining data consistency

Data patterns

Consumer-driven contract test

Consumer-side contract test

Service Component Test

Testing

Server-side page fragment composition

Client-side UI composition

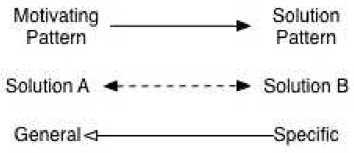
UI

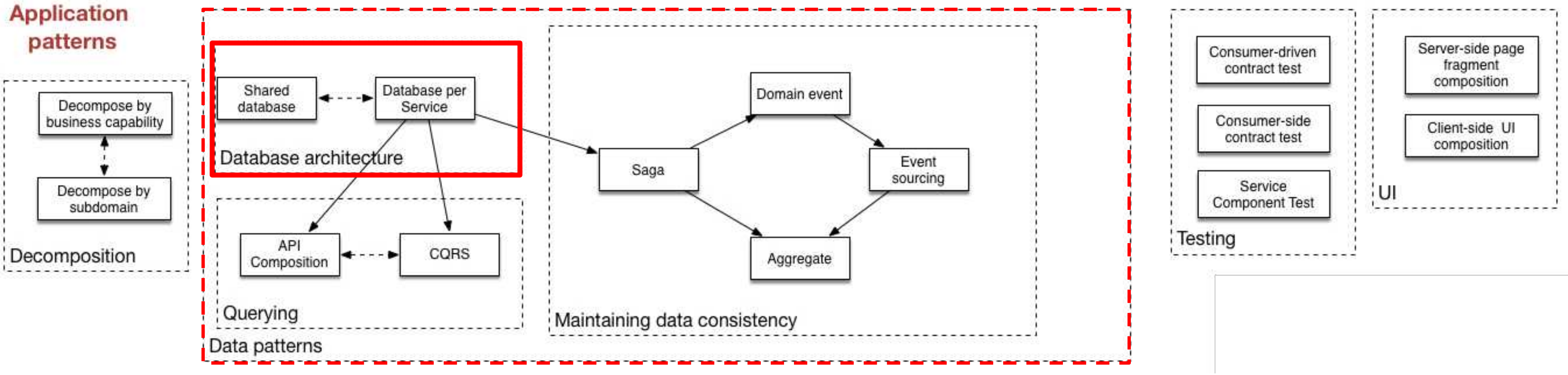
How to maintain data consistency?

- Database per Service pattern
- Shared Database (anti) pattern

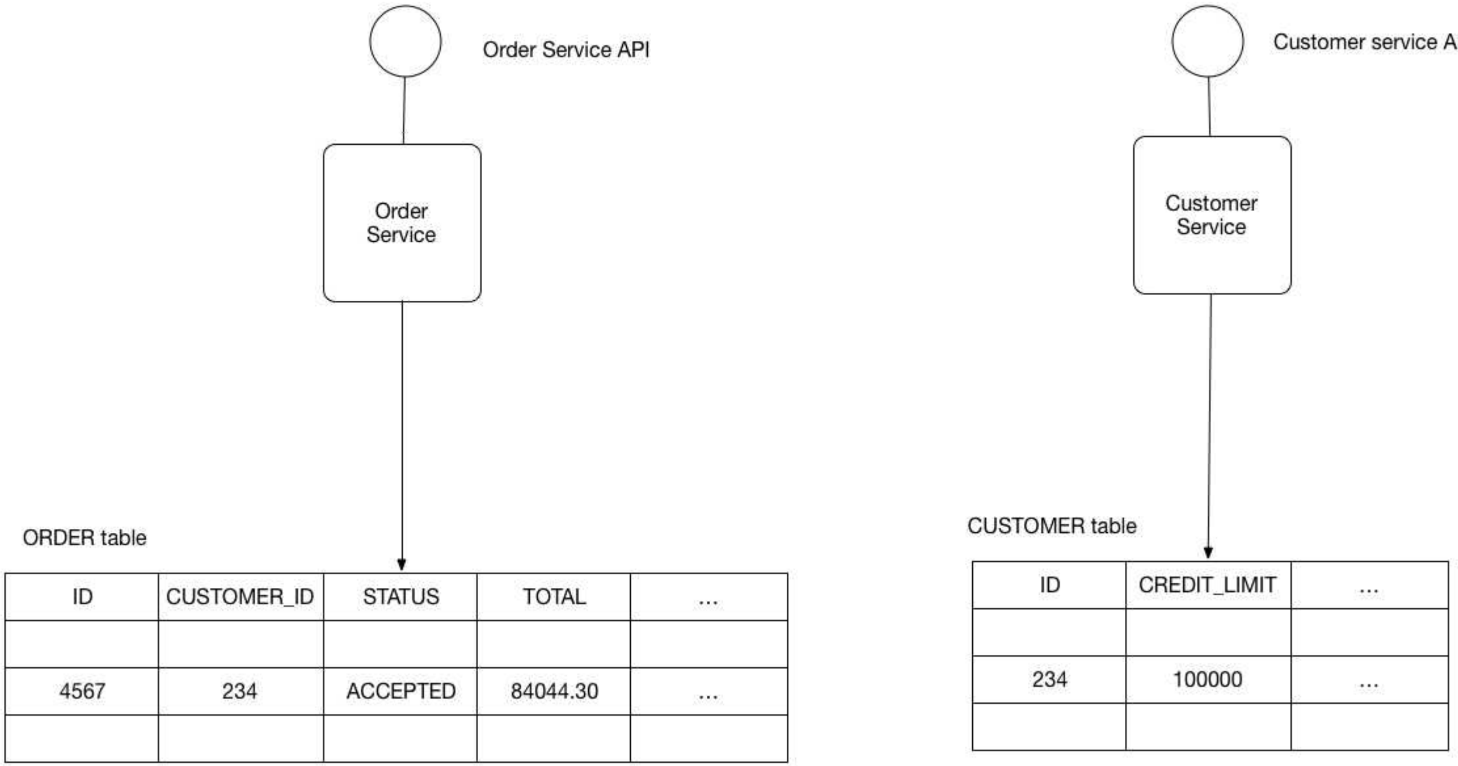
Several patterns exist to maintain **data consistency** and perform **queries**

Microservices Patterns
Database architecture





Database architecture



ORDER table

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

CUSTOMER table

ID	CREDIT_LIMIT	...
234	100000	...

Remark: CUSTOMER_ID is a foreign key referencing CUSTOMER

Database architecture

Requirements:

- Services must be loosely coupled
- Transactions must **enforce invariants** that span multiple services
- Transactions need to **query data** that is owned by multiple services
- Transactions may be **long-running**
- Some queries must **join data** that is owned by multiple services
- Different services have different **data storage** requirements

Reminder: Transaction processing

DATABASE TRANSACTION MODELS

For Advanced Applications

Edited by

AHMED K. ELMAGARMID

http://www.tpc.org/

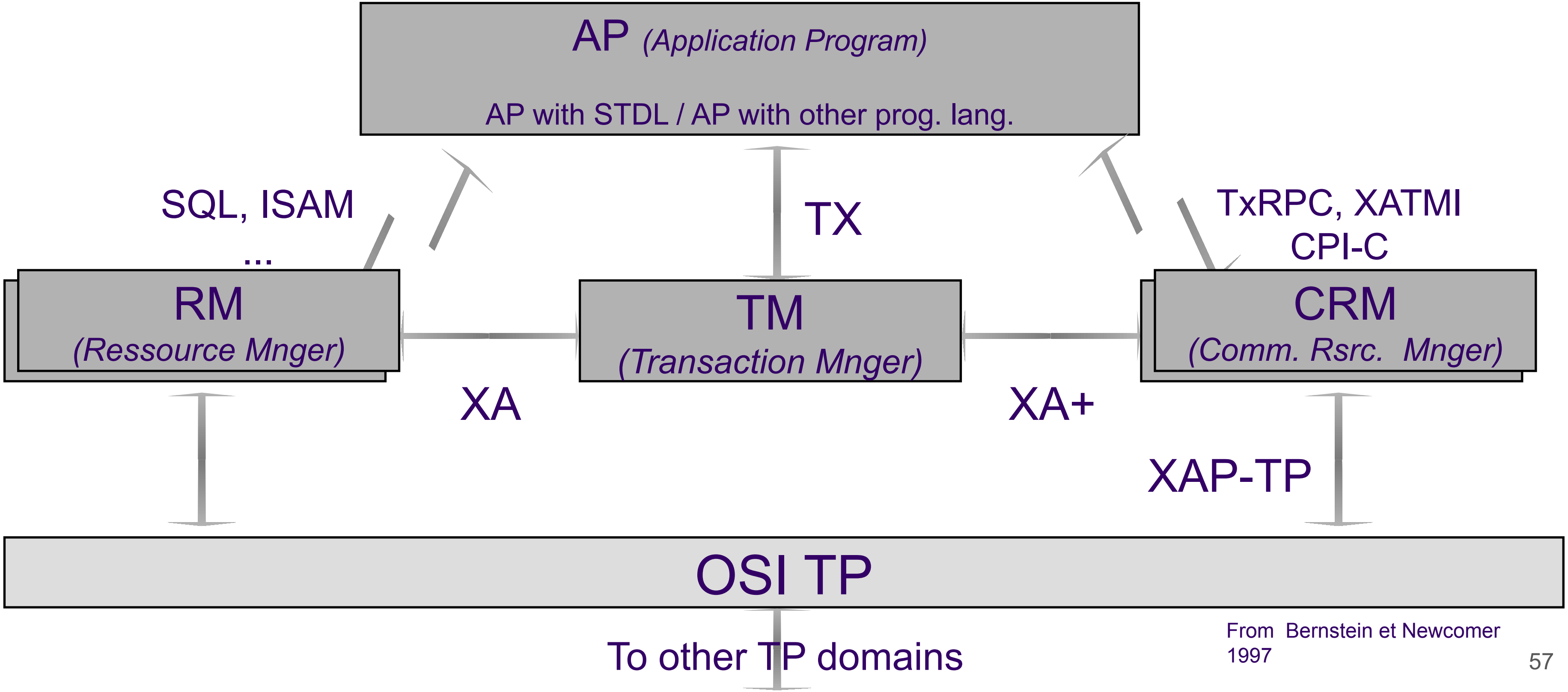
Short-running : several milliseconds to several minutes

- OLTP systems
  - Debit-Credit (TPC-A), Order (TPC-C) …
- Standards (Xopen DTP, OSI/TP…) for Two Phase Commit protocol
- Robust and scalable Transaction Monitors (Sabre, …)

Long-running : several hours to several days

- B2B usecases (next slides)
- No standard, several research works (Sagas, Contract, Flex, ACTA, …)

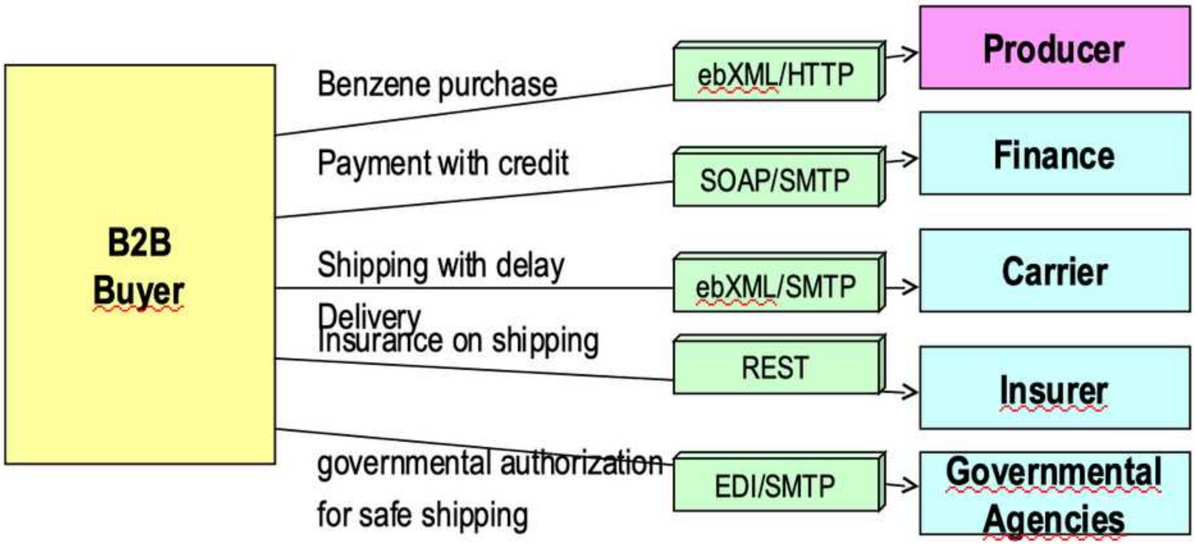
Reminder: Transaction processing The X/Open DTP Model



From Bernstein et Newcomer 1997

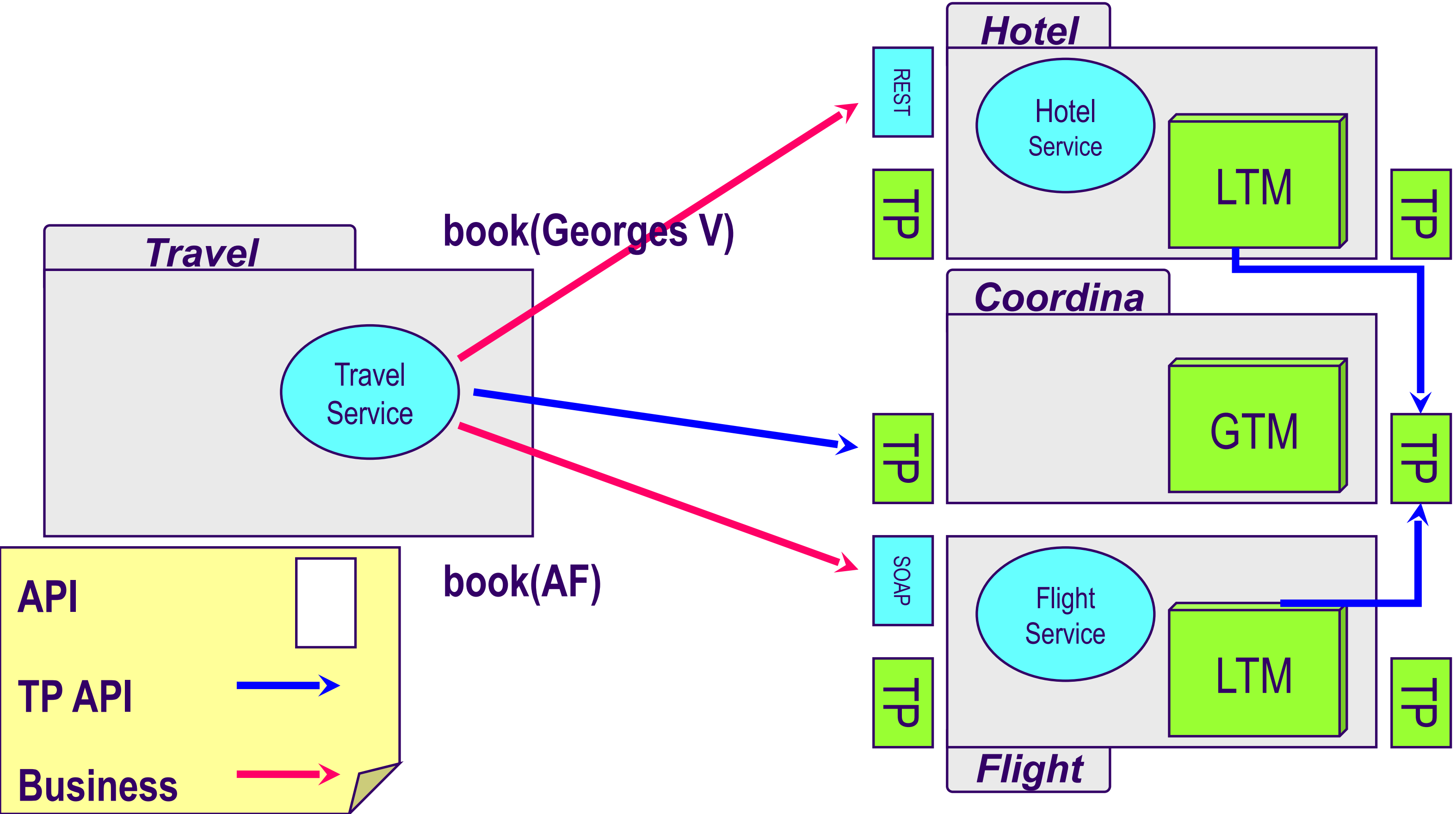
Long-running transactions

- Benzene purchase by a producer on the Web
- + Requires additional services provided by third parties

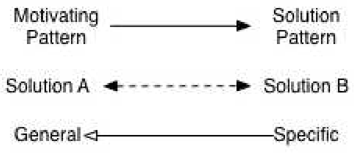


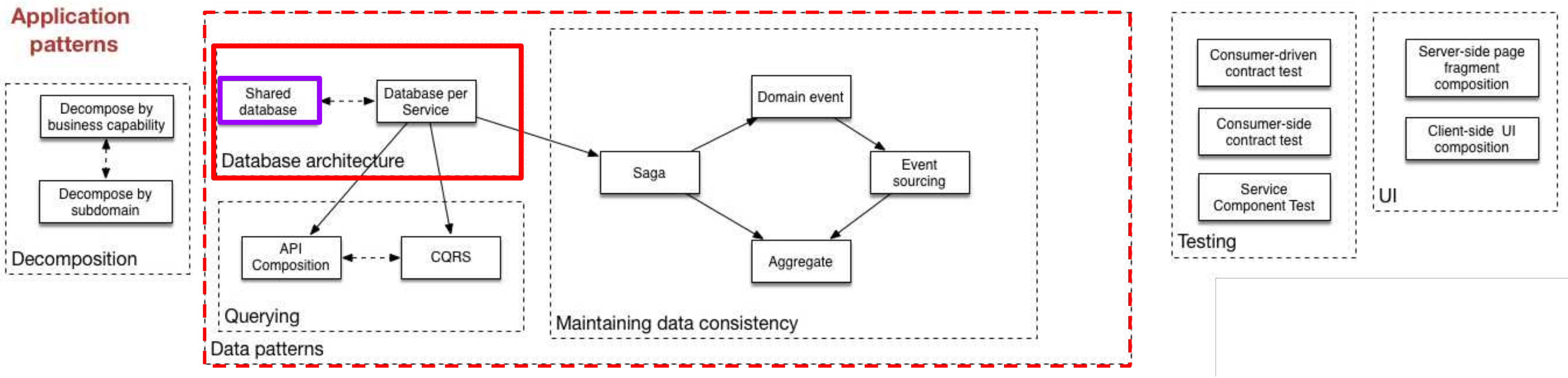
Several standards use compensation transactions : BTP, BWTP, XTML, ...

Long-running transactions : The travel agency



Microservice Patterns
Database architecture





Shared Database

Use a (single) database that is shared by multiple services

Each service freely accesses data owned by other services using local ACID transactions

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

Shared Databases

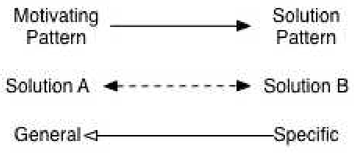
Advantages:

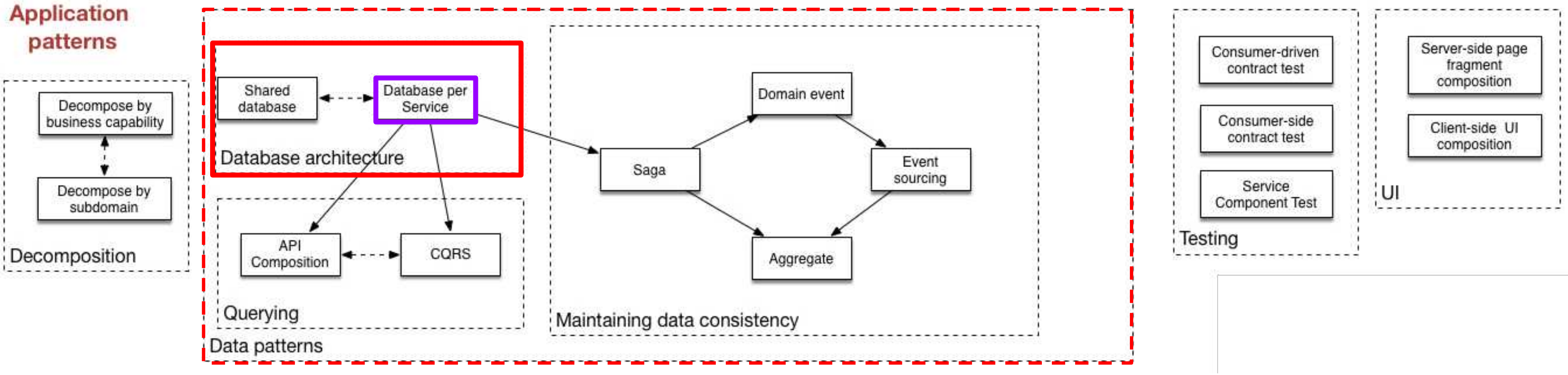
- Familiar and straightforward ACID transactions to enforce data consistency
- A single database is simpler to operate

Drawbacks:

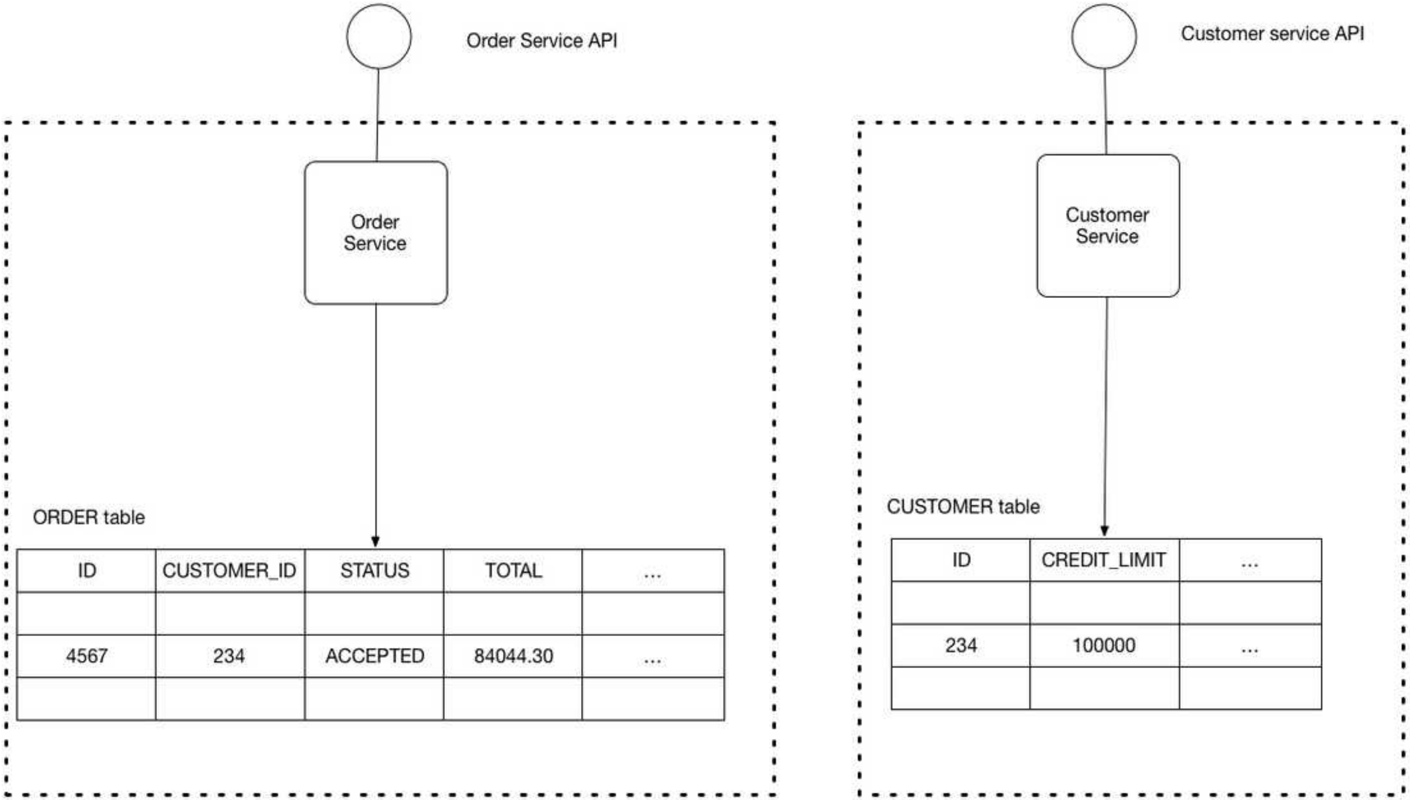
- Development time coupling
- Runtime coupling
- Inadequate for **long-running** or **long-lived** transactions
- One single database might not fit all requirements

Decomposition Patterns
Database architecture





Database per Service



ORDER table

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

CUSTOMER table

ID	CREDIT_LIMIT	...
234	100000	...

Database per Service

The service’s database is effectively part of the implementation of that service.

Different possibilities:

- Private-tables-per-service (same DBMS for all µS)
- Schema-per-service (same DBMS for all µS)
- Database-server-per-service

Database per Service

Advantages:

- Services are loosely coupled
- Different databases can be used (e.g. key-value store, document database, time series database, graph database)

Drawbacks:

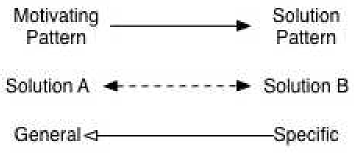
- Difficult to implement transactions that span multiple services
- Difficult to implement queries that join data in multiple databases
- Complexity of managing multiple SQL and NoSQL databases

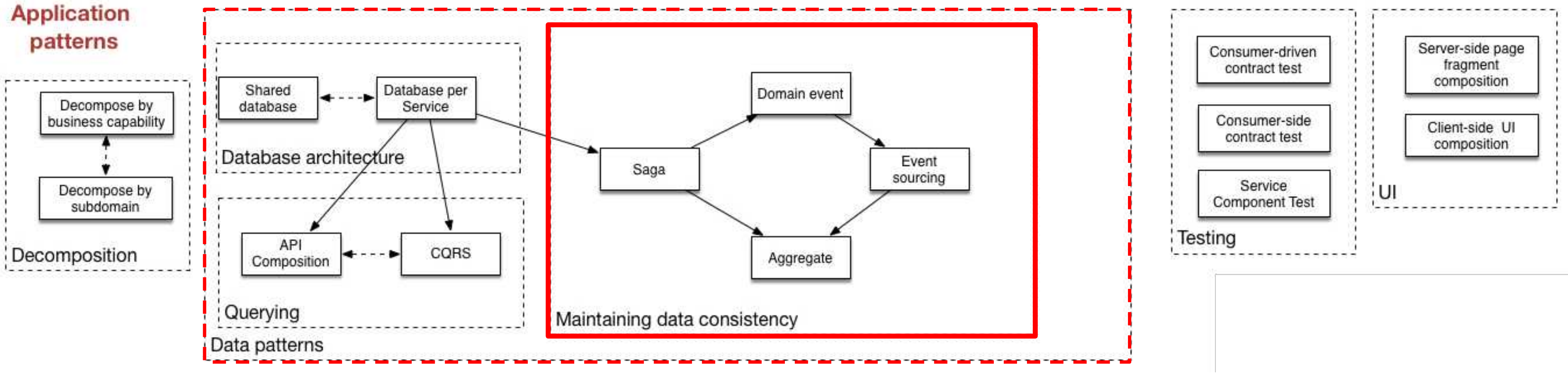
Database per Service

Some patterns provide solutions to the previously mentioned drawbacks

- **API Composition** - the application performs the join rather than the database
- **Command Query Responsibility Segregation** (CQRS) - maintain one or more materialized views that contain data from multiple services

Microservices Patterns
Data consistency





Maintaining consistency

- What is consistency ?
    - Paolo Viotti, Marko Vukolic: Consistency in Non-Transactional Distributed Storage Systems. ACM Comput. Surv. 49(1): 19:1-19:34 (2016)
- The “old way”: 2 phase commit
- The microservice way: transactions Saga

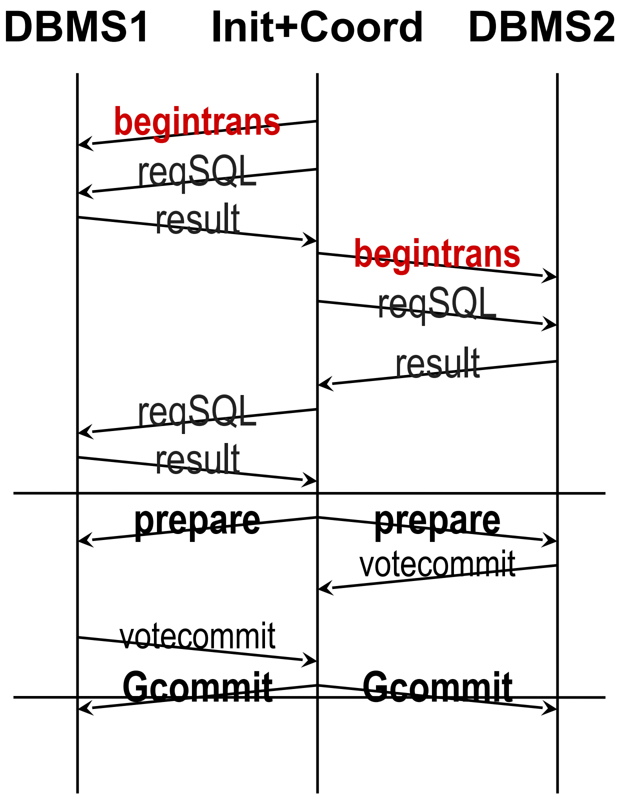
The Two-Phase Commit Protocol

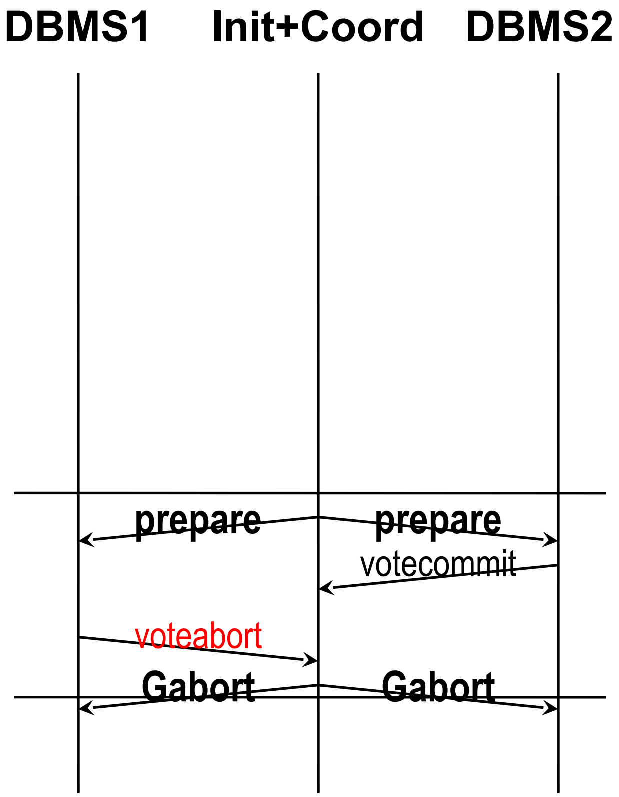
Achieve ACID properties over distributed X/Open ressources (MOM, RDBMS)

ACID for Atomicity, Consistency, Isolation, Durability

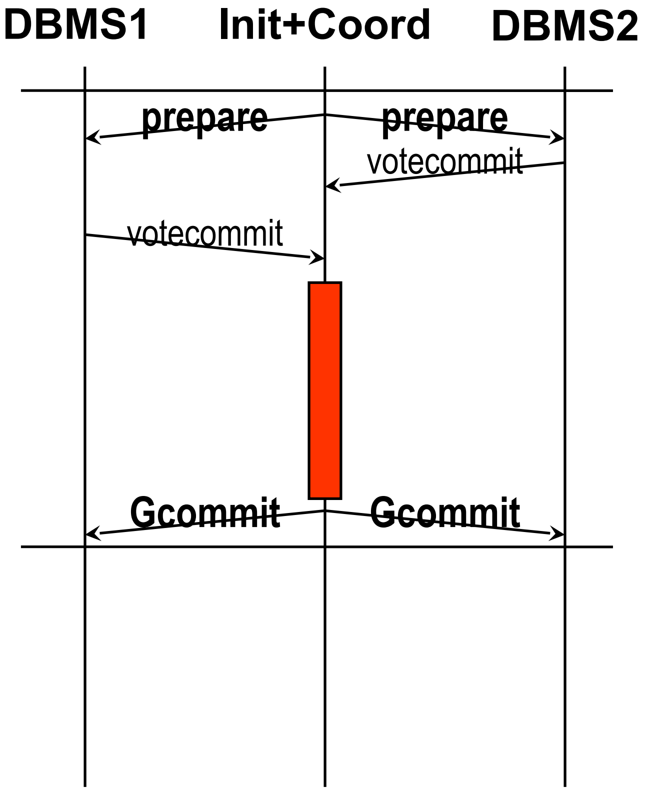
- 2PC Actors
  - Initiator (the application)
  - Coordinator (ie transaction monitor)
  - Ressources (aka participants, slaves)

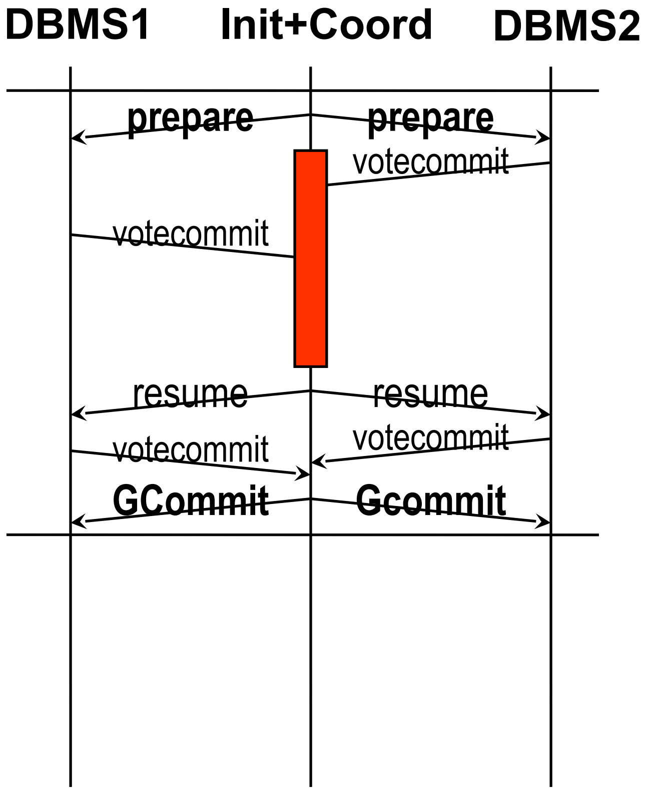
The Two-Phase Commit Protocol : without failure



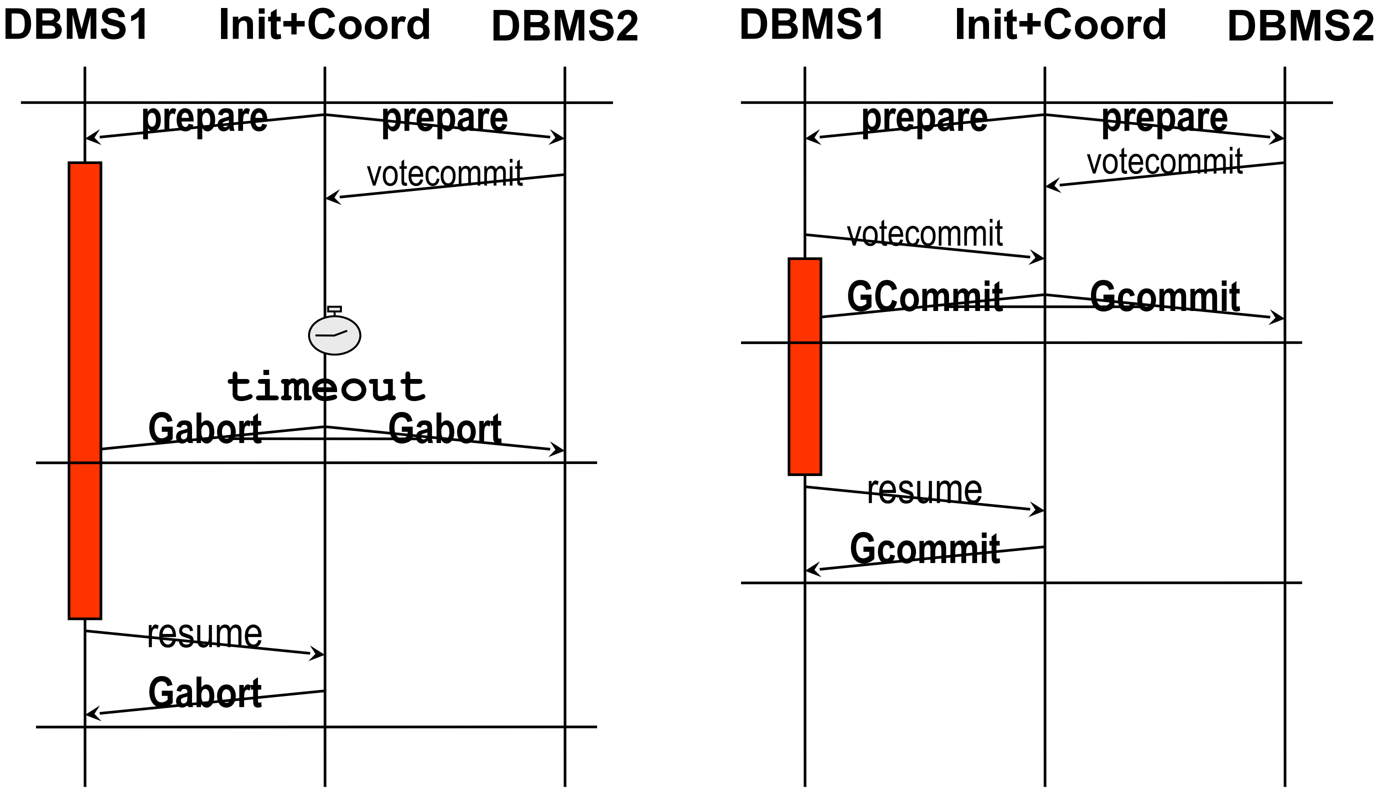


The Two-Phase Commit Protocol : with failure

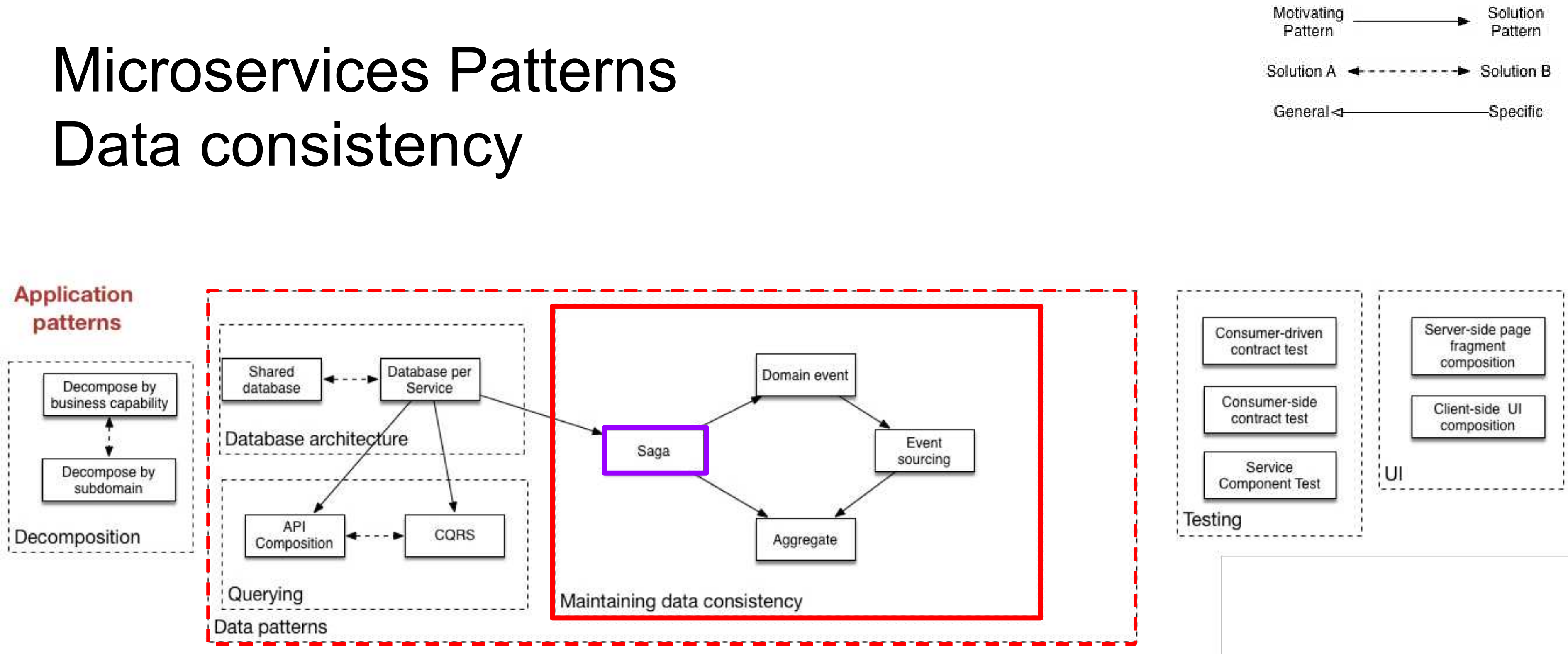




The Two-Phase Commit Protocol : with failure



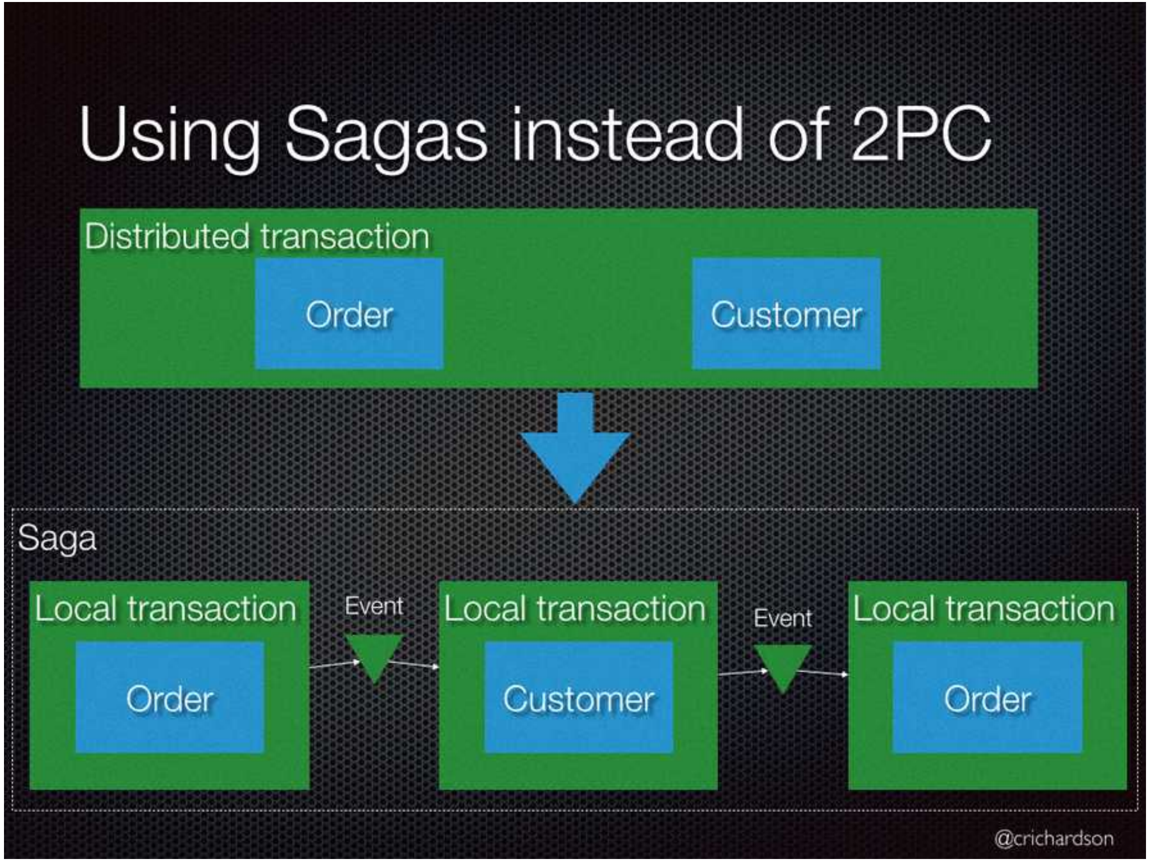
Microservices Patterns
Data consistency



Saga Transactions

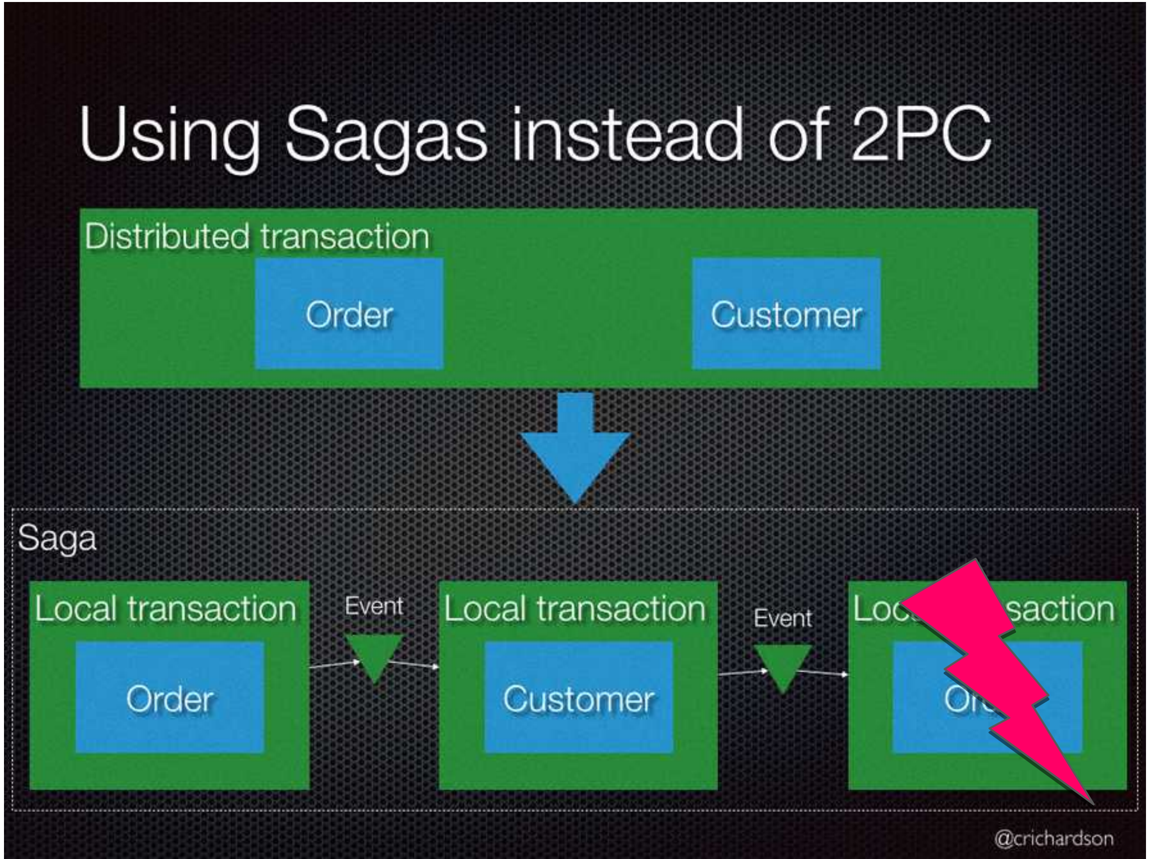
- Business transaction that spans multiple services are implemented as a **saga**
- A saga is a **sequence of local transactions**
- Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga
- If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions

Saga Transactions



Using Sagas instead of 2PC

Saga Transactions : without failure



1) Compensate Customer

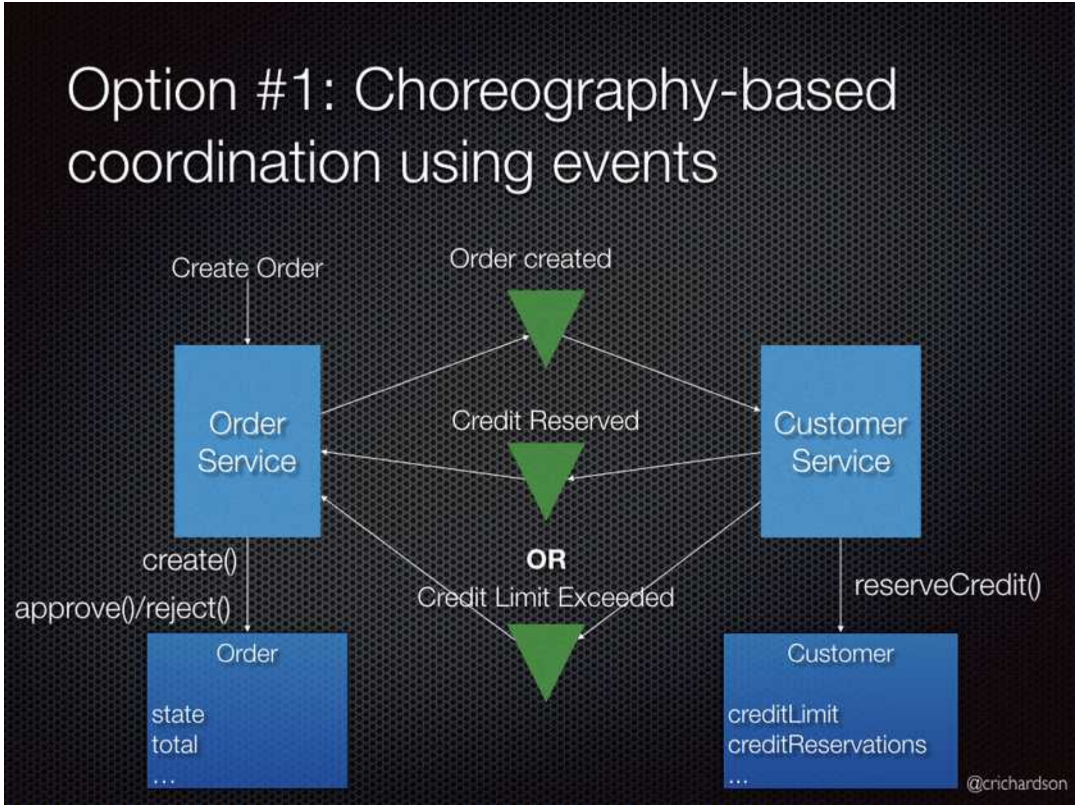
2) Compensate Order 1

Saga Transactions Coordination

Two ways for coordinating sagas:

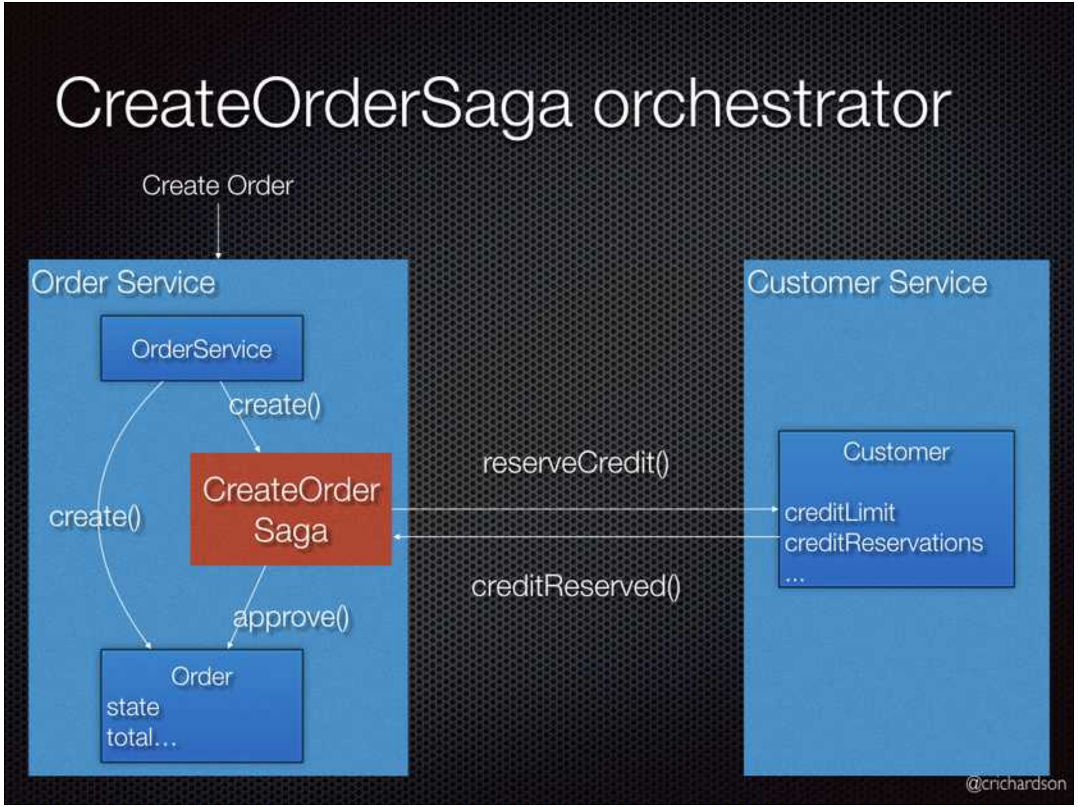
- Choreography
- Orchestration

Choregraphy-based Saga

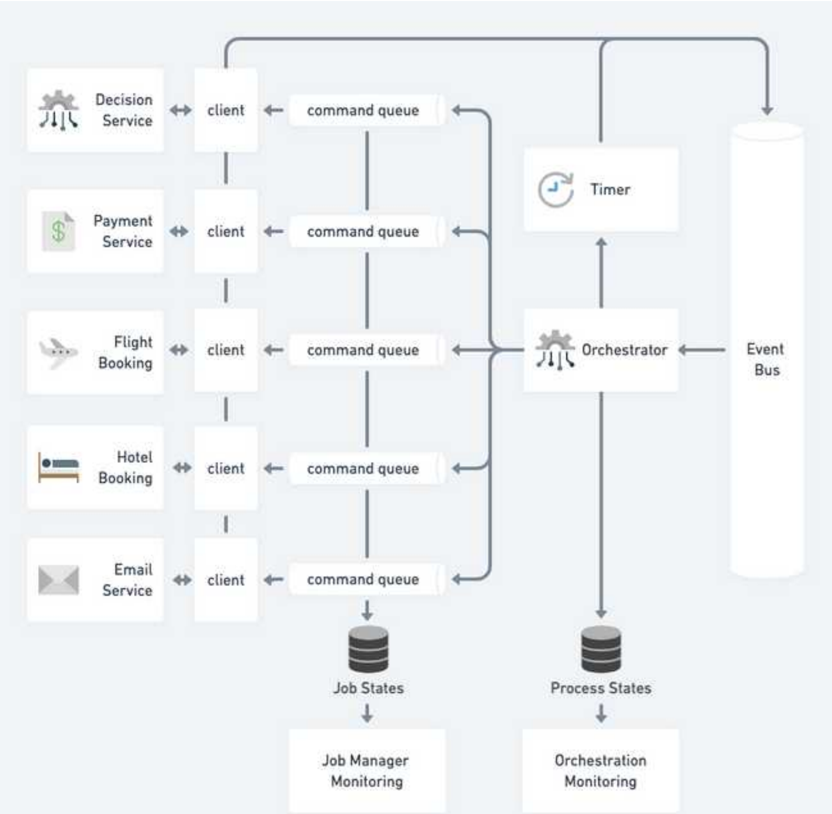


Option #1: Choreography-based coordination using events

Orchestration-based Saga



Example : Choregraphy-based Saga @ Zenaton



Event broker:
RabbitMQ then
Apache Pulsar

https://gillesbarbier.medium.com/building-an-event-driven-orchestration-engine-bf62d45aef5d

Saga Transactions

Advantages:

- Allows maintaining data consistency across multiple services without using distributed transactions

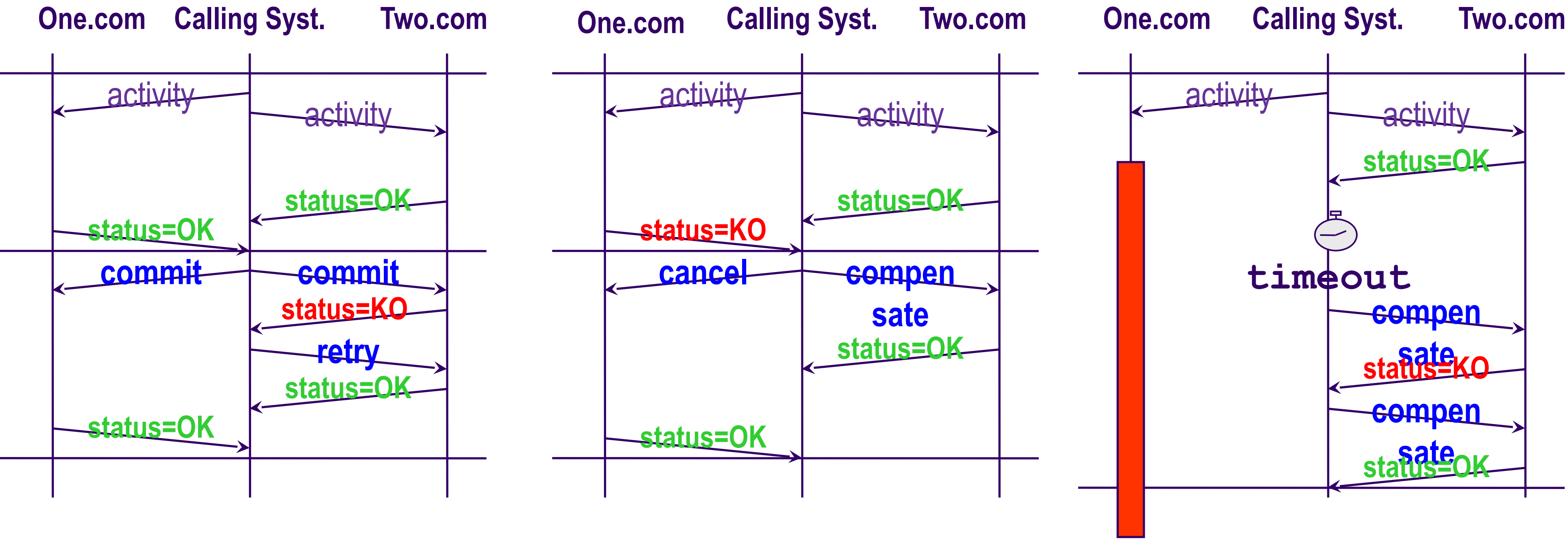
Drawbacks

- Complex programming model (workflow oriented)
- **Complex design of compensating transactions** that explicitly undo changes made earlier in a saga
- Compensation is not always possible
- Compensation can fail

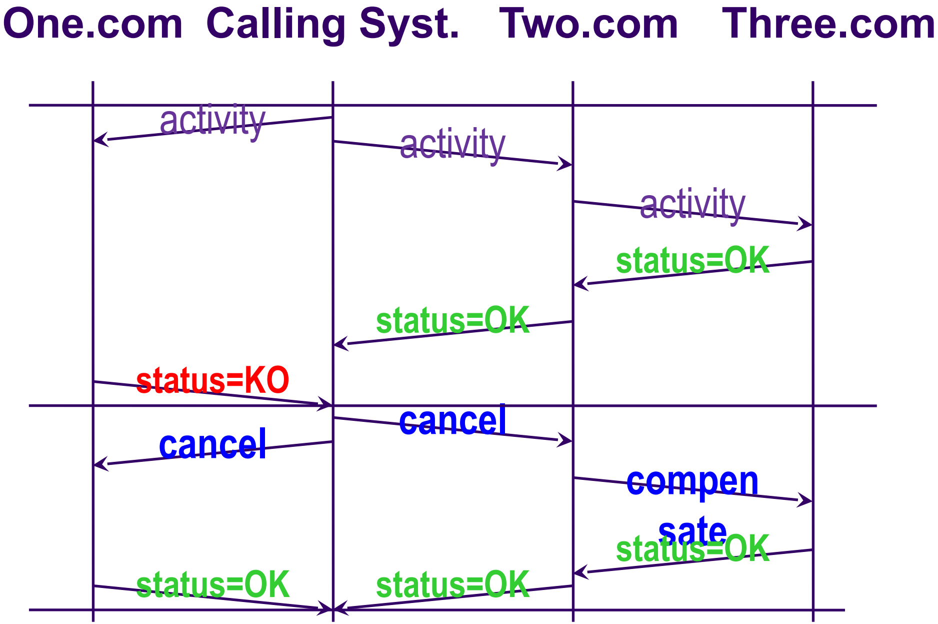
Issues : event/message broking

- A service must be able to atomically update its database
  and publish a message/event

Saga vs BWTP
BWTP transaction completion



Saga vs BWTP
BWTP compensation cascade



One.com Calling Syst. Two.com Three.com

activity

activity

activity

status=OK

status=OK

status=KO

cancel

cancel

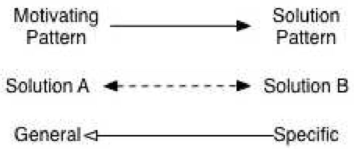
compen
sate

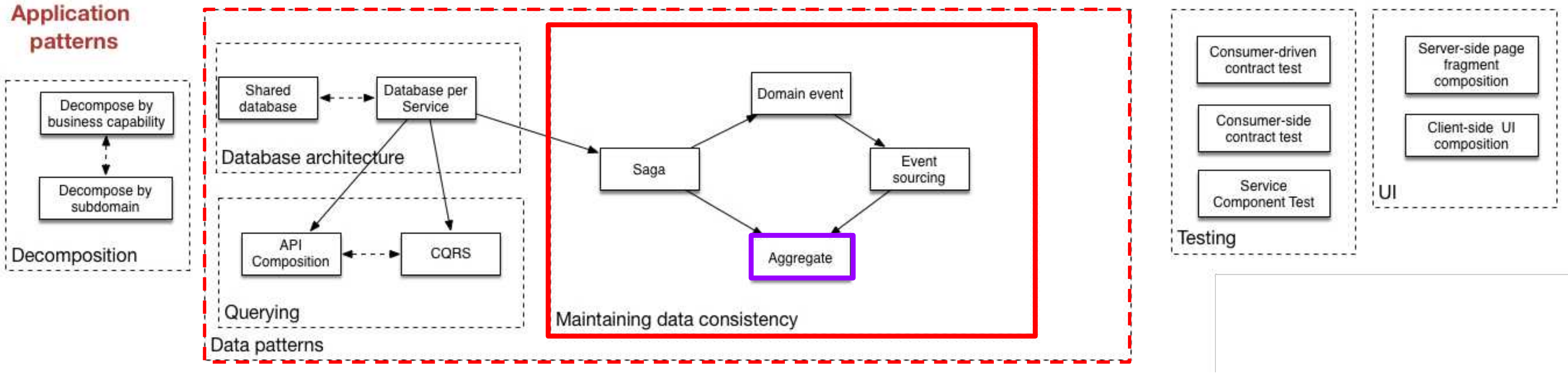
status=OK

status=OK

status=OK

Microservices Patterns
Data consistency





Aggregate

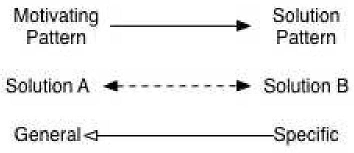
An aggregate is a graph of objects that can be treated as a unit

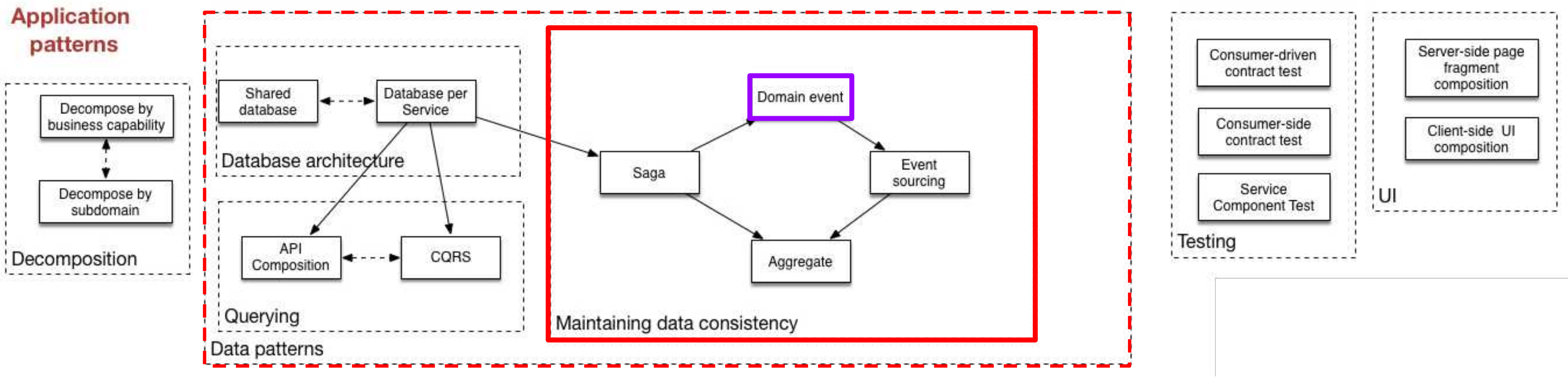
> *Example: “When you drive a car, you do not have to worry about moving the wheels forward, making the engine combust with spark and fuel, etc.; you are simply driving the car. In this context, the car is an aggregate of several other objects and serves as the aggregate root to all of the other systems.” (Wikipedia)*

A concept from DDD (Domain-Driven Design)

Aggregates produce **Domain events**.

Microservices Patterns
Data consistency





Domain event

A service often needs to publish events when it updates its data.

Used by transaction Saga and CQRS.

A concept from DDD (Domain-Driven Design)

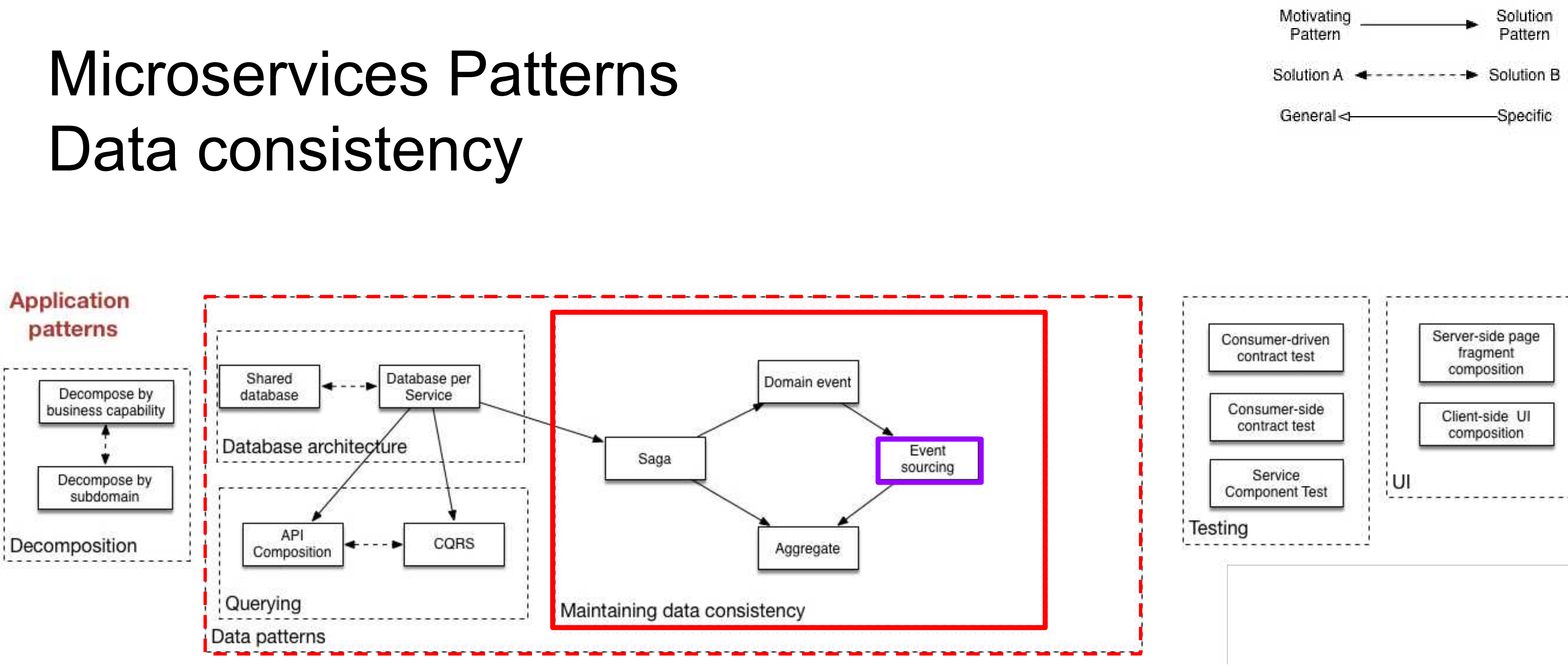
Domain events are emitted by Aggregates

ie OrderCreated, CreditReserved, CreditLimitExceeded ...

https://paucls.wordpress.com/2018/05/31/ddd-aggregate-roots-and-domain-events-publication/

https://en.wikipedia.org/wiki/Domain-driven_design

Microservices Patterns
Data consistency



Event sourcing

How to reliably/atomically update the database and publish messages/events?

2PC is not an option!

A concept from DDD (Domain-Driven Design)

Event sourcing

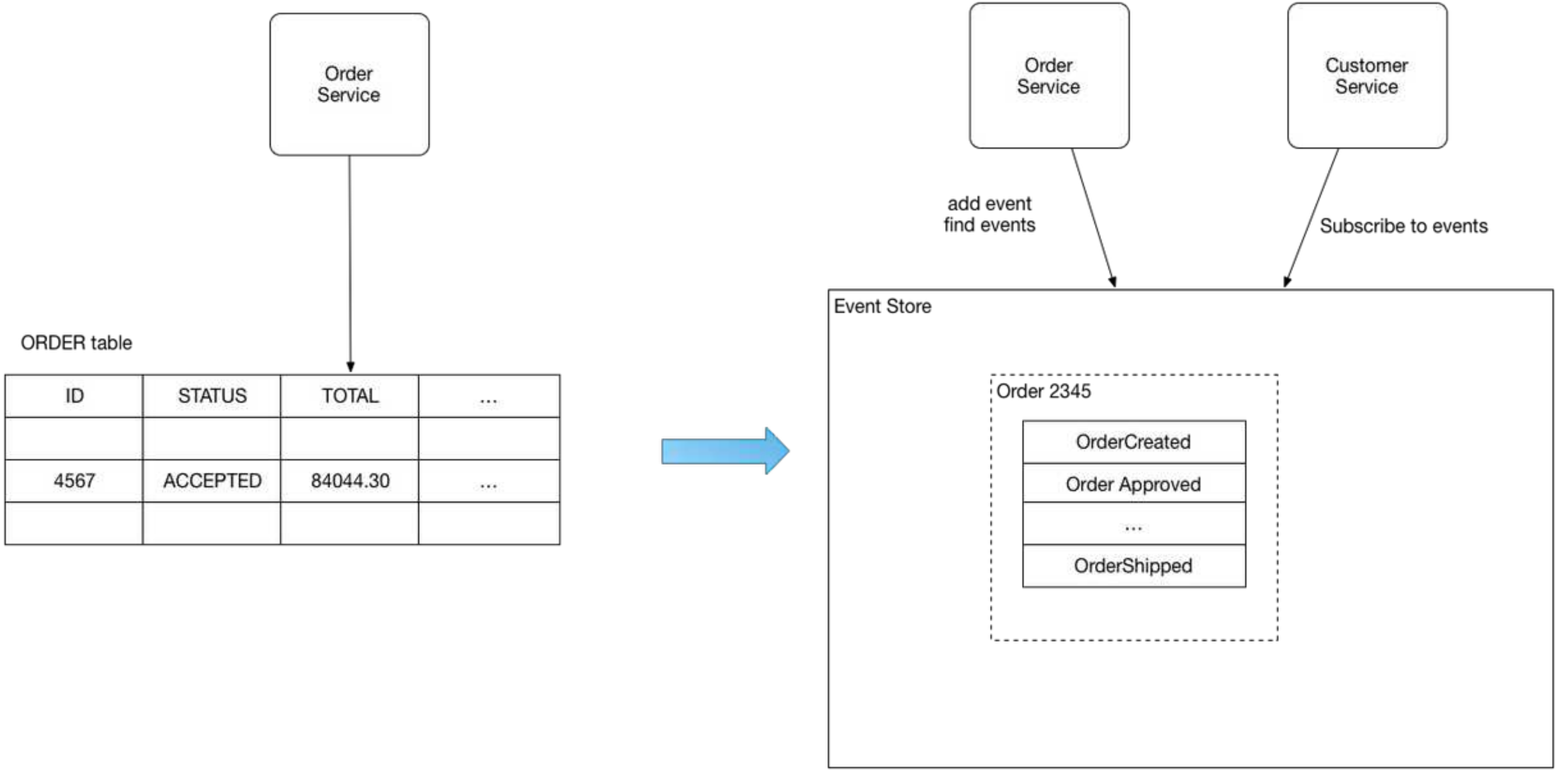
Event sourcing persists the state of a business entity as a sequence of state-changing events

Whenever the state of a business entity changes, a new event is appended to the list of events

Applications persist events in an event store, which is a database of events

The event store behaves like a message broker

Event sourcing - Example



ID	STATUS	TOTAL	...
4567	ACCEPTED	84044.30	...

Event sourcing

Benefits

- Solves one of the key problems in implementing an event-driven architecture and makes it possible to reliably publish events whenever state changes.
- Provides a reliable audit log of the changes made to a business entity
- Makes it possible to implement temporal queries that determine the state of an entity at any point in time.

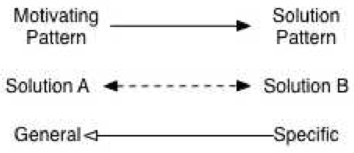
Drawbacks

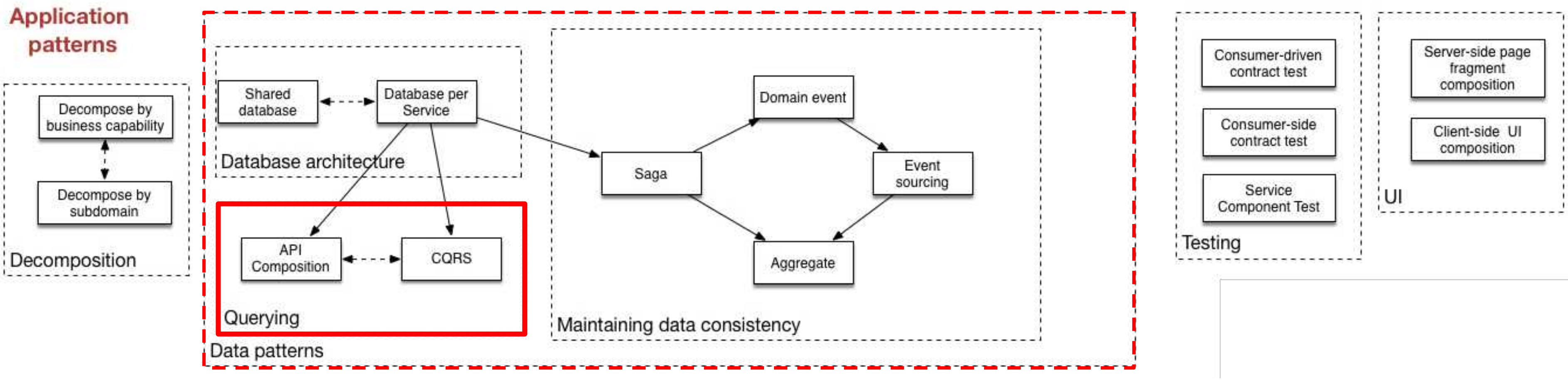
- Different and unfamiliar style of programming.
- The event store is difficult to query since it requires typical queries to reconstruct the state of the business entities.

Related

- Event sourcing implements the Audit logging pattern.

Microservices Patterns Querying



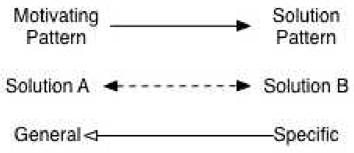


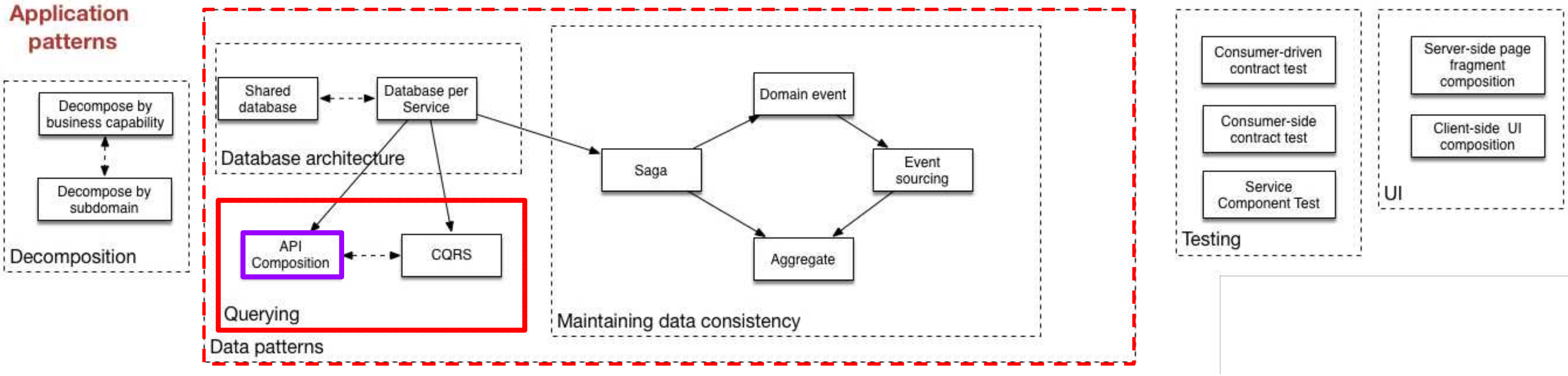
Querying patterns

How to implement a query that retrieves data from multiple services in a microservice architecture?

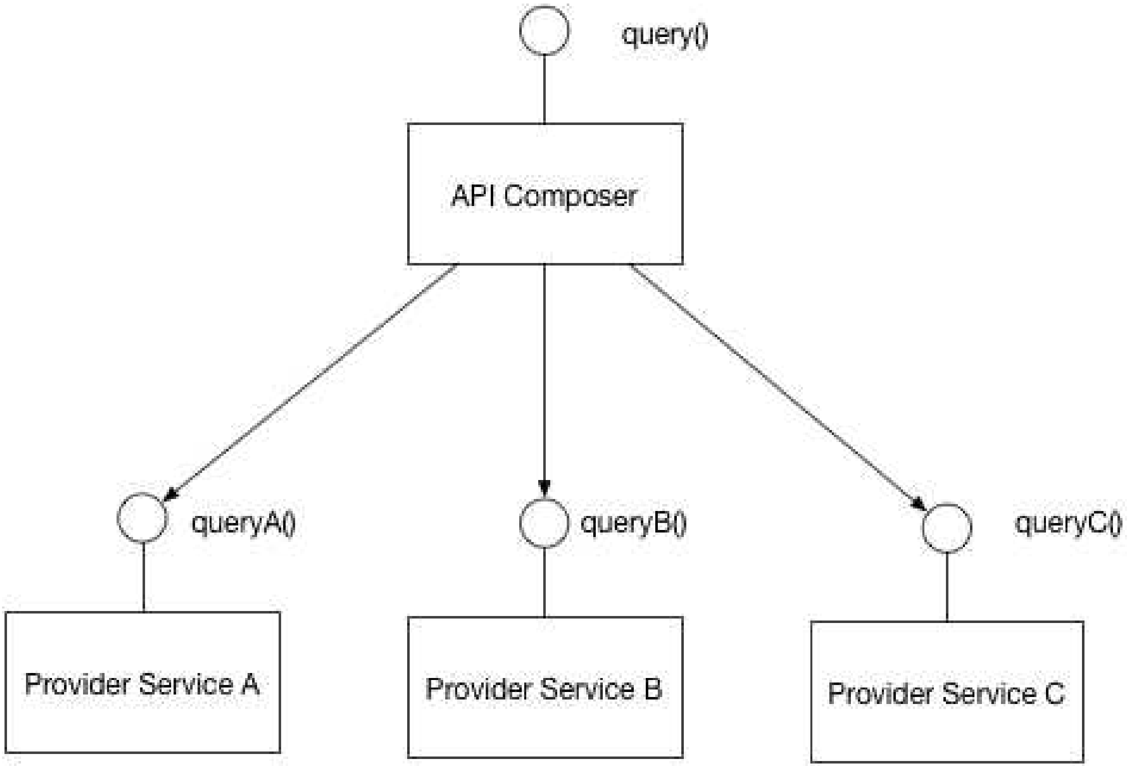
Remark: In shared database, JOIN requests between several tables

Microservices Patterns Querying





API Composition



API Composition

Advantages

- A simple way to query data in a microservice architecture

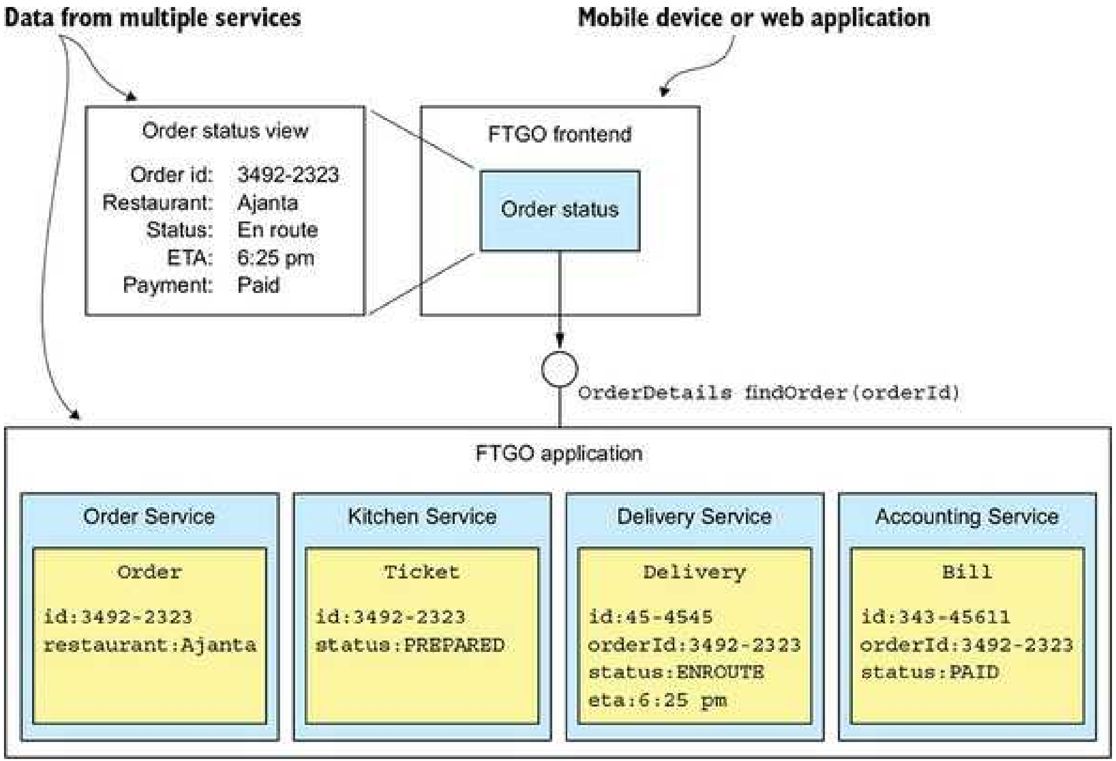
Drawbacks

- Some queries would result in inefficient, in-memory joins of large datasets.

Remark: Research works on Distributed Database Systems : Semi-Joins …

@see ACM SIGMOD, VLDB conf proceeding

API Composition : Example



Data from multiple services

Mobile device or web application

Order status view

Order id:	3492-2323
Restaurant:	Ajanta
Status:	En route
ETA:	6:25 pm
Payment:	Paid

```
OrderDetails findOrder(orderId)
```

```
Order

id:3492-2323
restaurant:Ajanta
```

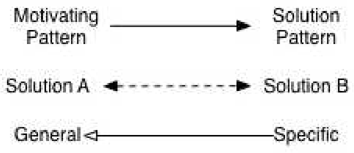
```
Ticket

id:3492-2323
status:PREPARED
```

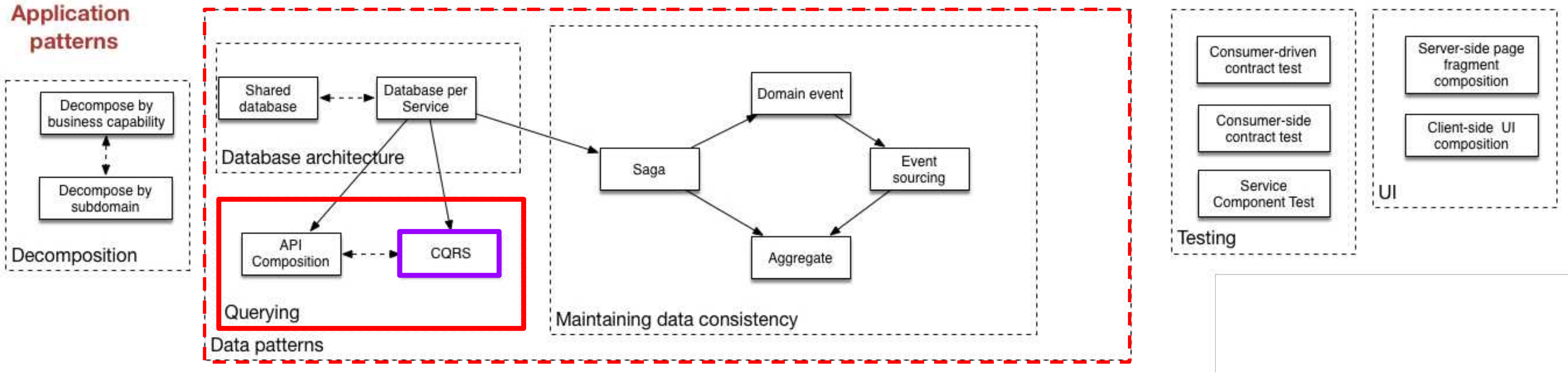
```
Delivery

id:45-4545
orderId:3492-2323
status:ENROUTE
eta:6:25 pm
```

```
Bill

id:343-45611
orderId:3492-2323
status:PAID
```

From https://ajay-yadav109458.medium.com/queries-in-microservice-79a657a928af

Microservices Patterns
Querying





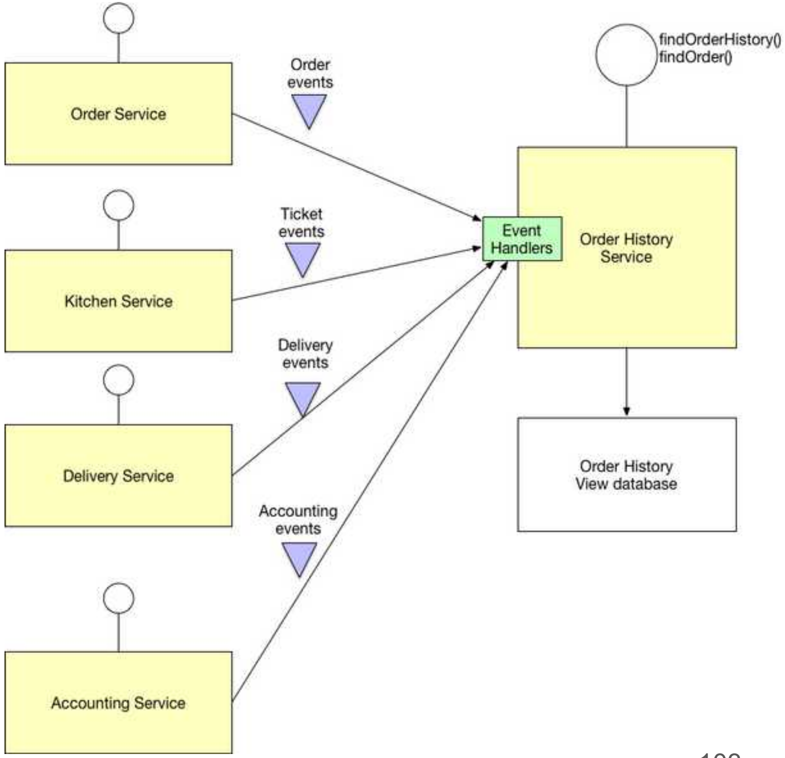
Command Query Responsibility Segregation (CQRS)

Isolate read (query) and write (command) into services

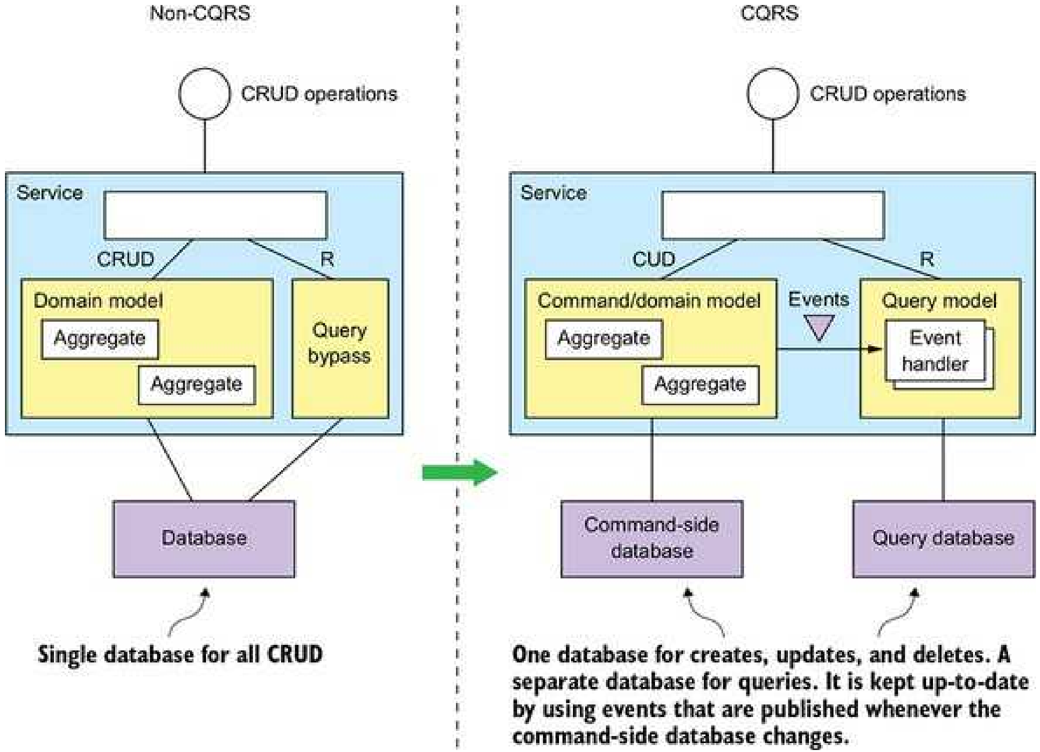
Create a (read-only) view database that replicates the data
The database is populated by subscribing to Domain events published by services
This patterns allows separating command and query components

Remark: Functional and non-functional requirements are different for read and write

- Write : transactional (consistency, isolation, ...), Schema Normalization
- Read : Schema Denormalization for perf, scalability



Non-CQRS versus CQRS



Single database for all CRUD

One database for creates, updates, and deletes. A separate database for queries. It is kept up-to-date by using events that are published whenever the command-side database changes.

https://ajay-yadav109458.medium.com/queries-in-microservice-79a657a928af

Command Query Responsibility Segregation (CQRS)

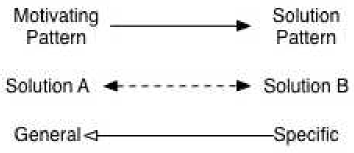
Advantages:

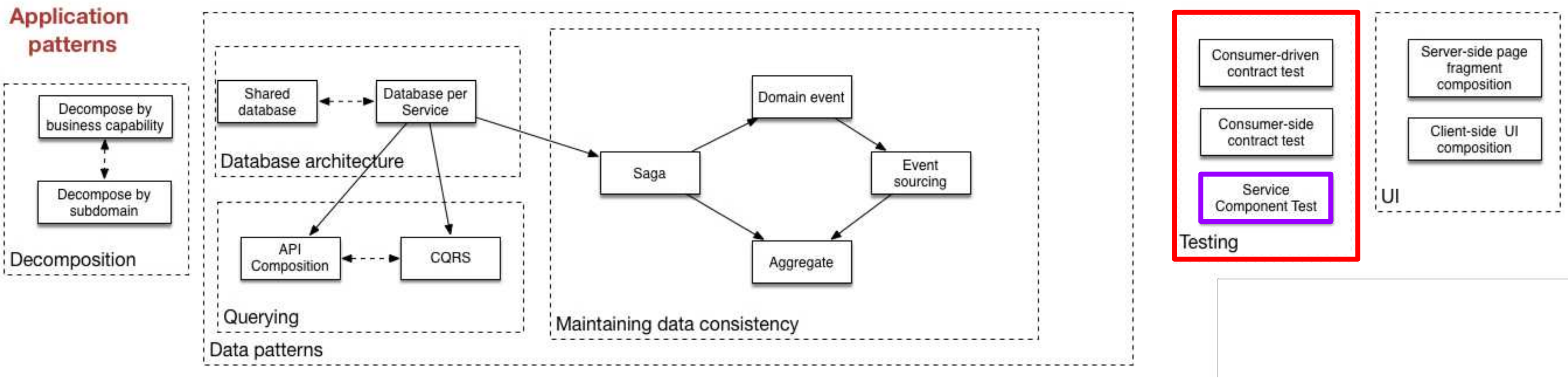
- Supports multiple denormalized views that are scalable and performant
- Improved separation of concerns = simpler command and query models
- Necessary in an event sourced architecture

Drawbacks:

- Increased complexity
- Potential code duplication
- Replication lag/eventually consistent views

Microservices Patterns Testing





Service Component Test

- How to easily test a service?
  - End to end testing (i.e. tests that launch multiple services) is difficult, slow, and expensive.
- Need to design a test suite that tests a service in isolation using test doubles for any services that it invokes.
- Example: Spring Cloud Contract

Service Component Test

Advantages:

- Testing a service in isolation is easier, faster, more reliable and cheap

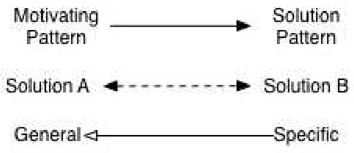
Drawbacks:

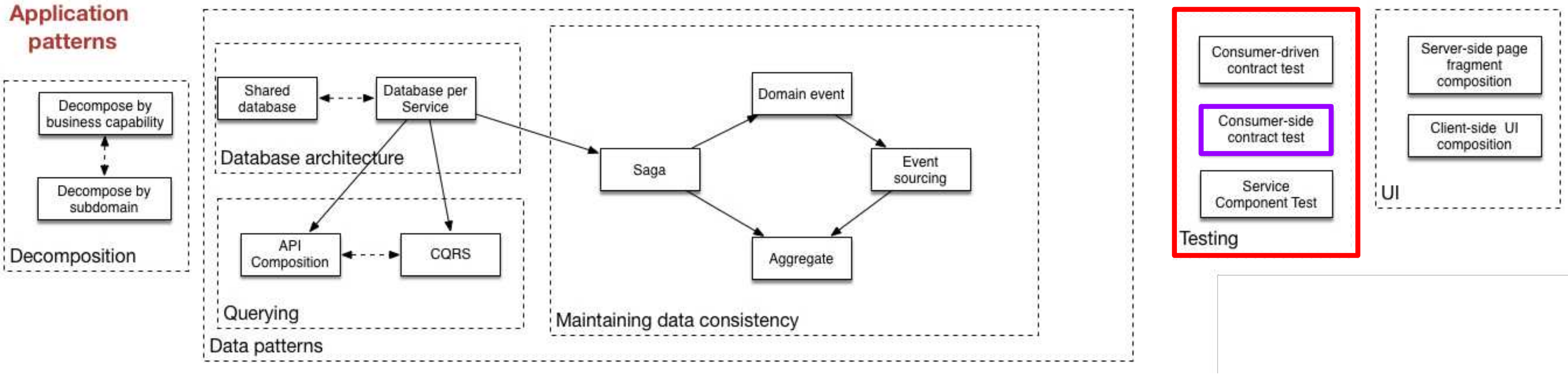
- Tests might pass but the application will fail in production

Issues:

- How to ensure that the test doubles always correctly emulate the behavior of the invoked services?

Microservices Patterns Testing

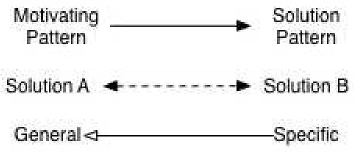


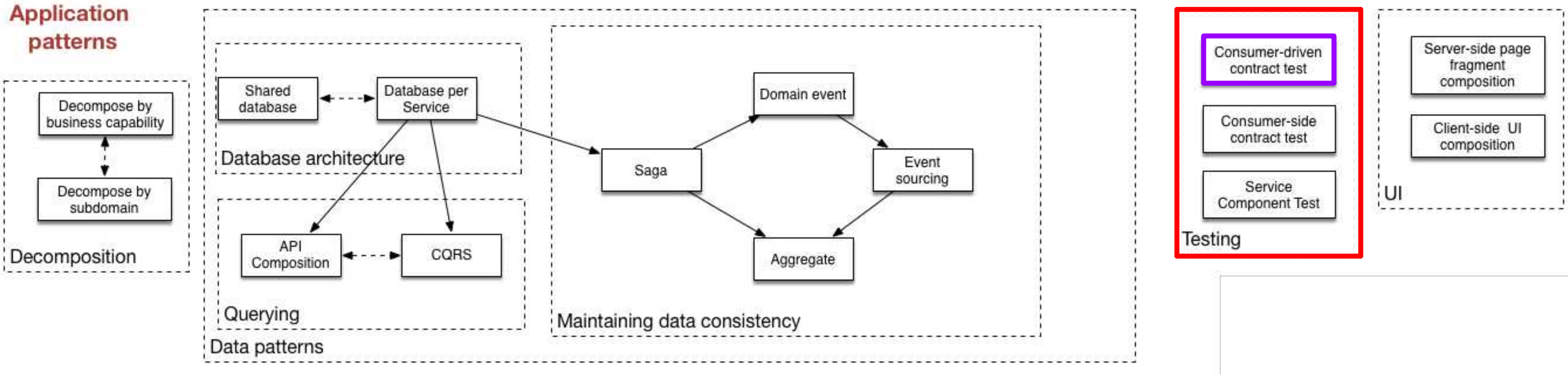


Consumer-side contract test

Test for verifying that the client of a service can communicate with the service

Microservices Patterns Testing





Consumer-driven contract test

How to easily test that a service provides an API that its clients expect?

Need for a test suite for a service that is written by the developers of another service that consumes it.

The test suite verifies that the service meets the consuming service’s expectations.

Example: Spring Cloud Contract.

Consumer-driven contract test

Advantages

- Testing a service in isolation is easier, faster, more reliable and cheap

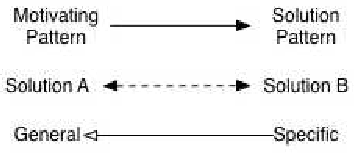
Drawbacks

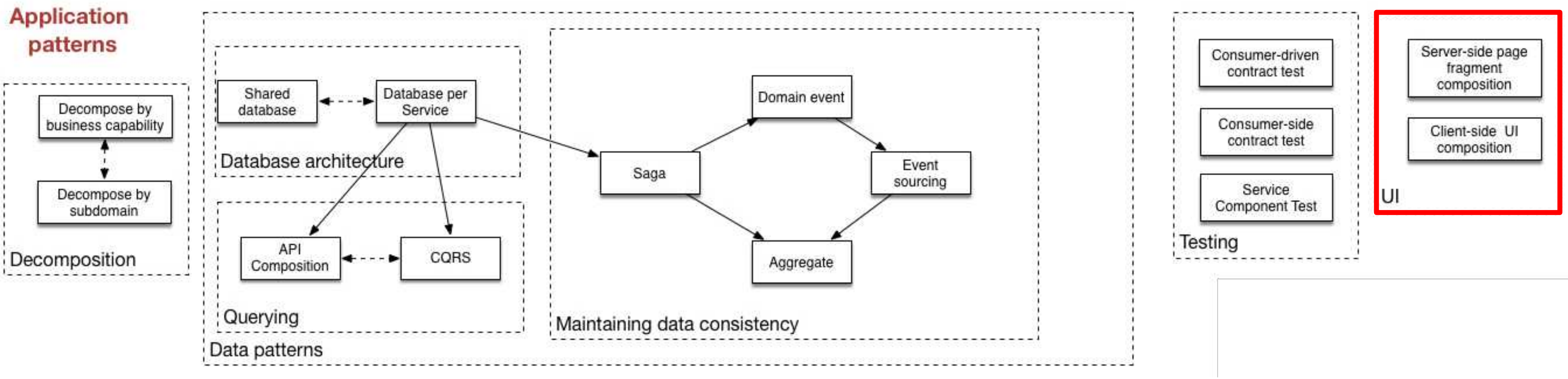
- Tests might pass but the application will fail in production

Issues

- How to ensure that the consumer provided tests match what the consumer actually requires?

Microservices Patterns
UI (User Interface)





UI patterns

Services are developed by business capability/subdomain-oriented teams that are **also responsible** for the **user experience**

Some UI screens/pages display data from multiple service

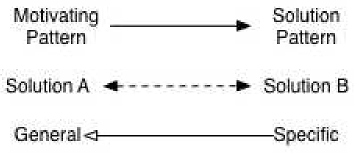
For instance, an e-commerce product detail page can display:

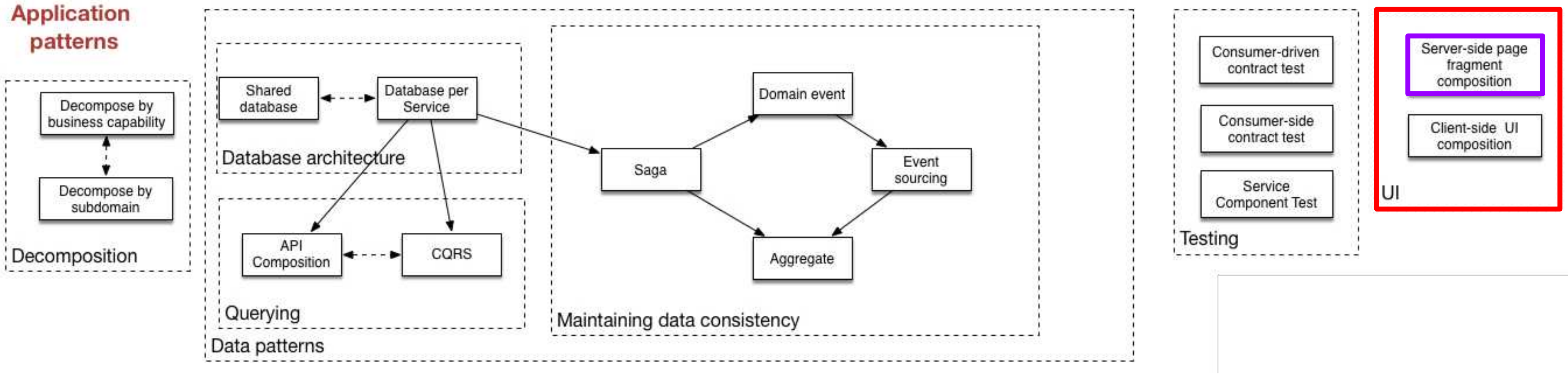
- Basic information about the book such as title, author, price, etc.
- Your purchase history for the book
- Availability
- Buying options
- Other items that are frequently bought with this book
- Other items bought by customers who bought this book
- Customer reviews
- Sellers ranking
- …

Each data item corresponds to a separate service → how it is displayed is the responsibility of a **different team**

How to implement a UI screen or page that displays data from multiple services?

Microservices Patterns
UI (User Interface)



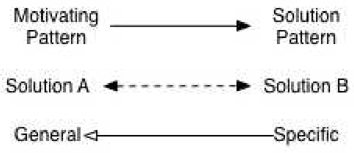


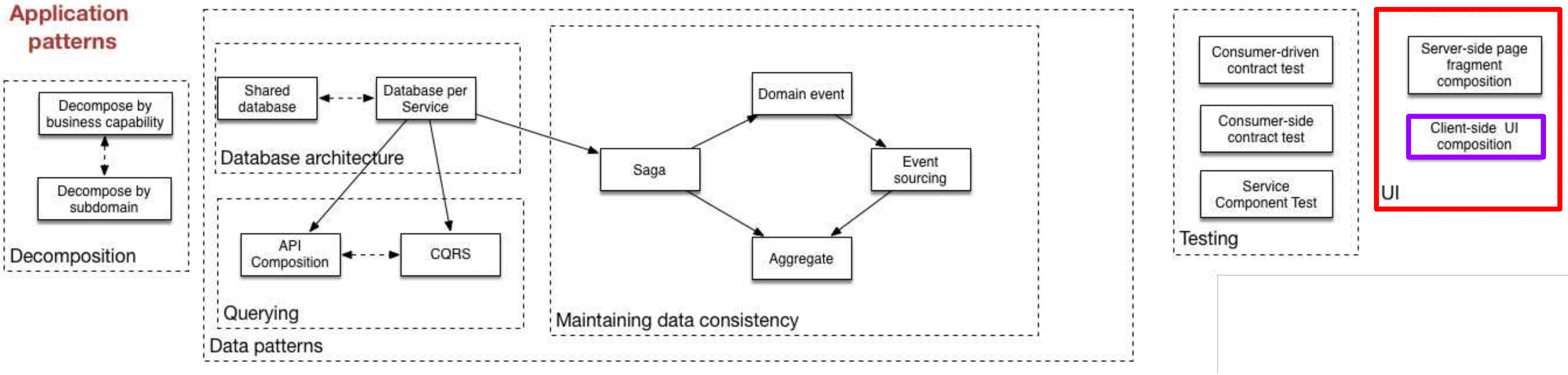
Server-side page fragment composition

Each team develops a web application that generates an **HTML fragment**

The **UI team** develops the page templates that build pages by performing server-side aggregation of the service-specific HTML fragments.

Microservices Patterns
UI (User Interface)





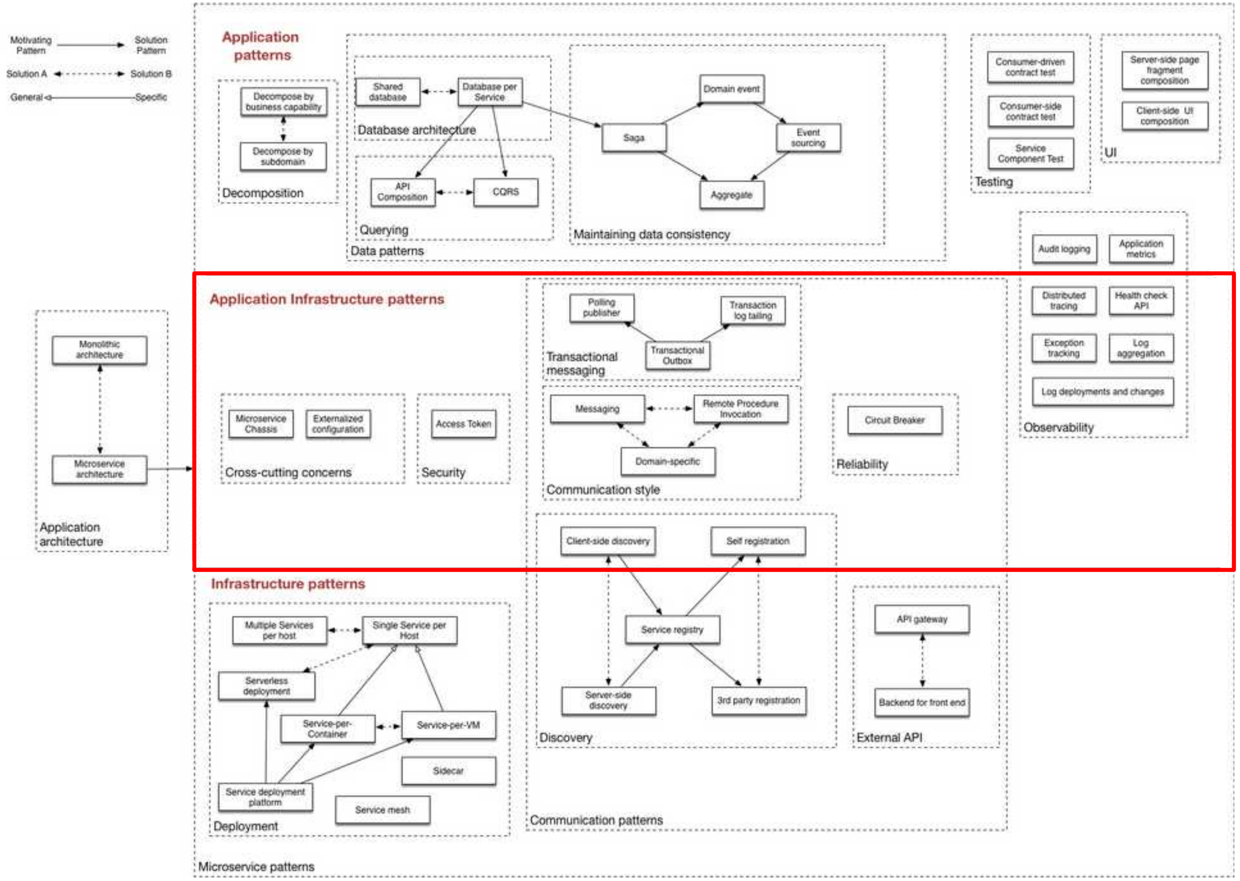
Client-side UI composition

Each team develops a client-side UI component that implements the region of the page/screen for their service.

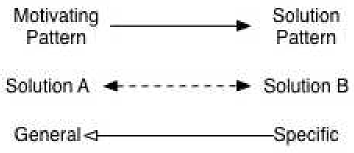
The **UI team** implements the page skeletons that build pages/screens by composing multiple, service-specific UI components.

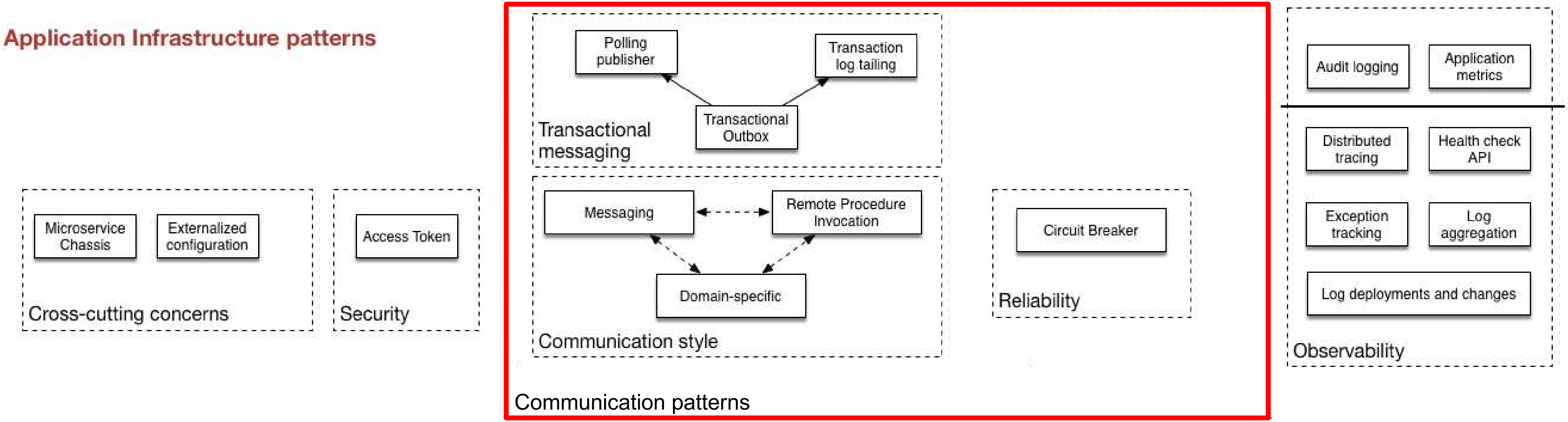
Remark: SPA frameworks are component-based and can load dynamically modules (ie NGx). Each service team provide a set of UI components.

Microservices Patterns

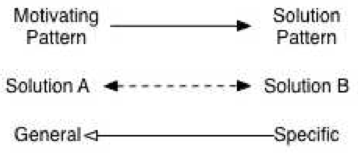


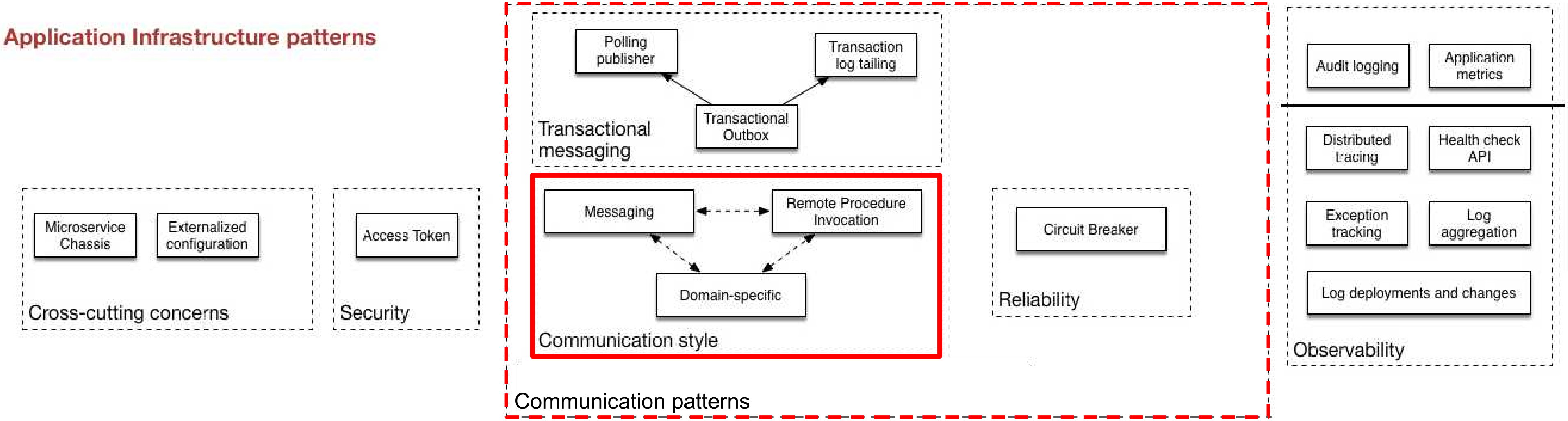
Microservices Patterns
Communication patterns





Microservices Patterns
Communication style



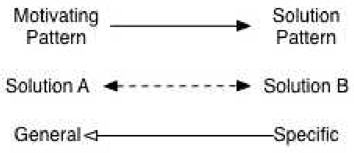


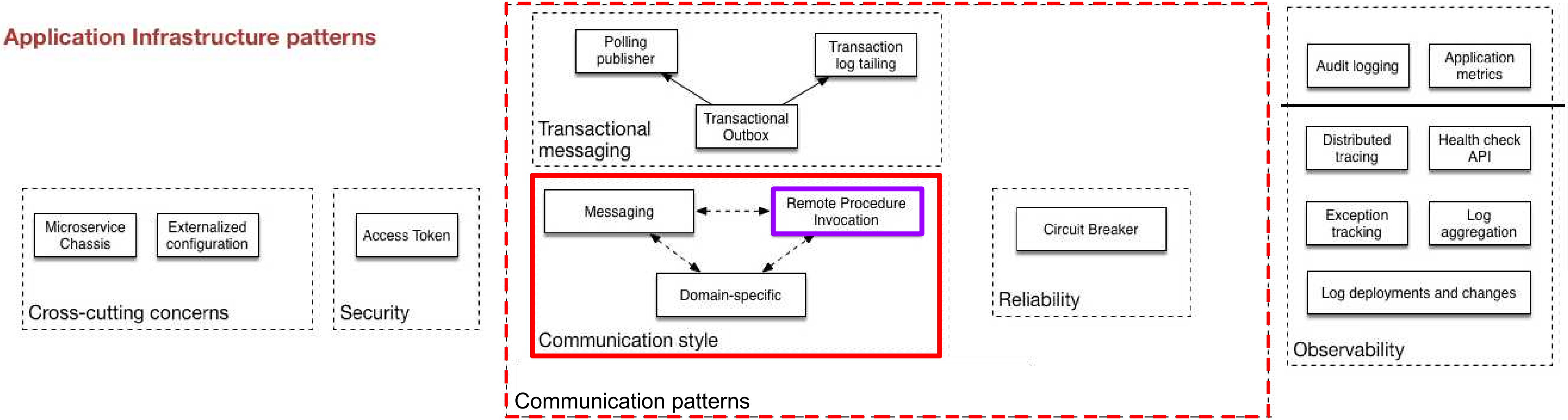
Inter-service communications patterns

There exist various inter-service communication protocols

- Remote Procedure Invocation
- Messaging
- Domain-specific protocol(s)

Microservices Patterns
Communication style





Remote Procedure Invocation (RPI)

A client uses a request/reply-based protocol to make requests to a service

There are numerous examples of RPI technologies

- REST
- gRPC (Protobuf), Thrift, Avro
- OMG CORBA

Remote Procedure Invocation (RPI)

Advantages

- Simple and familiar
- Simpler system since there is no intermediate broker

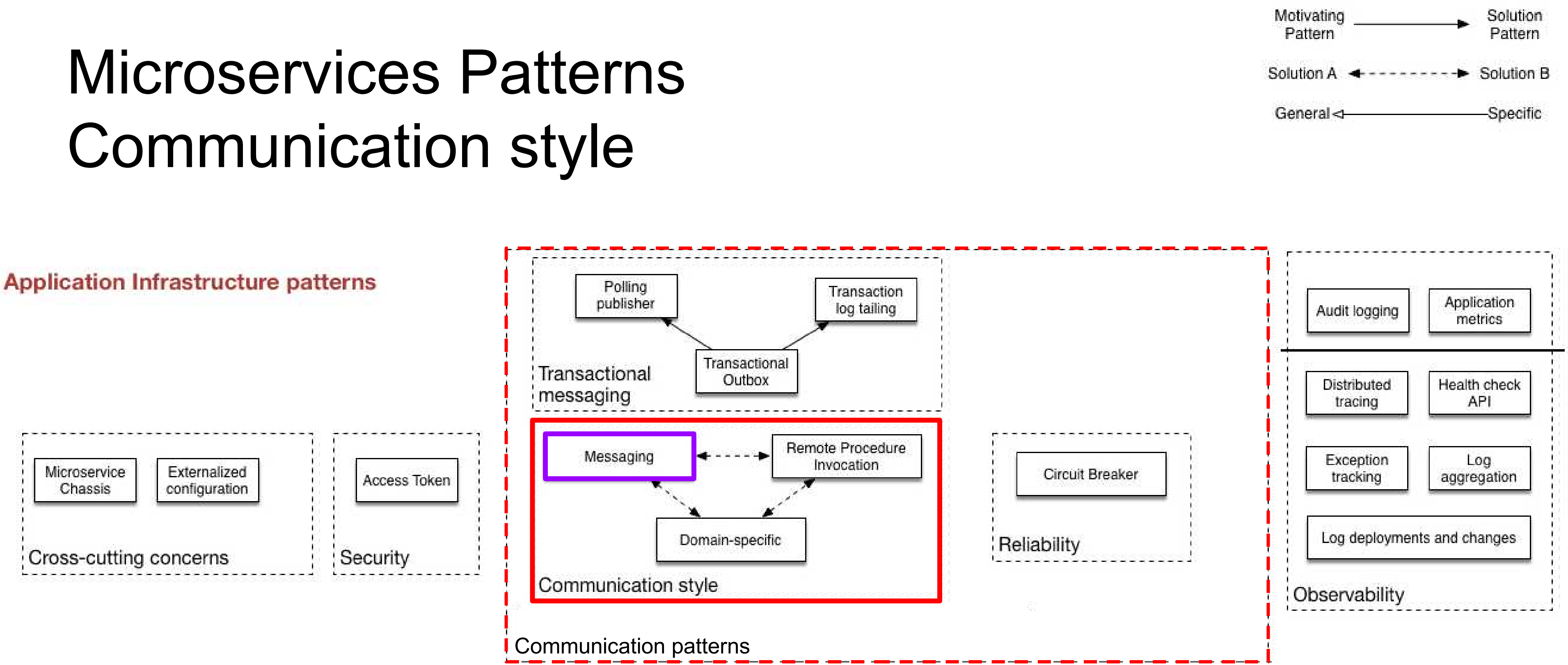
Drawbacks

- not other interaction patterns such as notifications, request/async response, publish/subscribe, publish/async response
- the service must be available for the duration of the interaction

Issues

- Client needs to discover locations of service instances
- API/schema versioning, untagged data and dynamic typing (Avro)

Microservices Patterns
Communication style



Messaging

Perform inter-service communication
by exchanging messages over messaging channels

Examples of messaging technologies

- AMQP (XA ressource)
- MQTT (Unreliable backhauls in IoT networks)

Examples of messaging technologies

- Apache Kafka (intra-datacenter)
- Apache Pulsar
- RabbitMQ

Messaging

Advantages:

- Loose coupling between clients and services
- Improved availability
- Supports a variety of communication patterns (request/reply, notifications, request/async response, publish/subscribe, publish/async response))

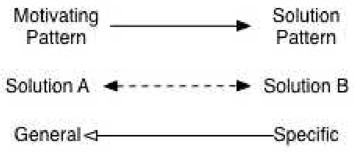
Drawbacks:

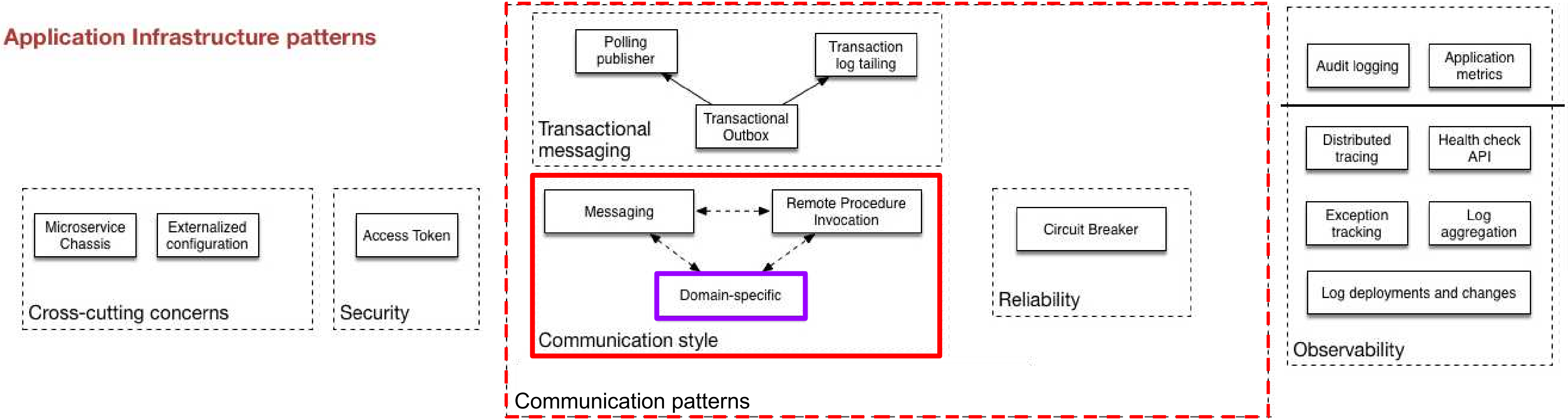
- Additional complexity of message broker
- Implementing request/reply-style communication is more complex

Issues:

- Client needs to discover location of message broker
- Message serialization : Protobuf, Thrift, Avro ...

Microservices Patterns Communication style





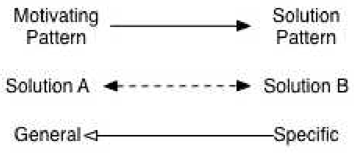
Domain-specific protocols

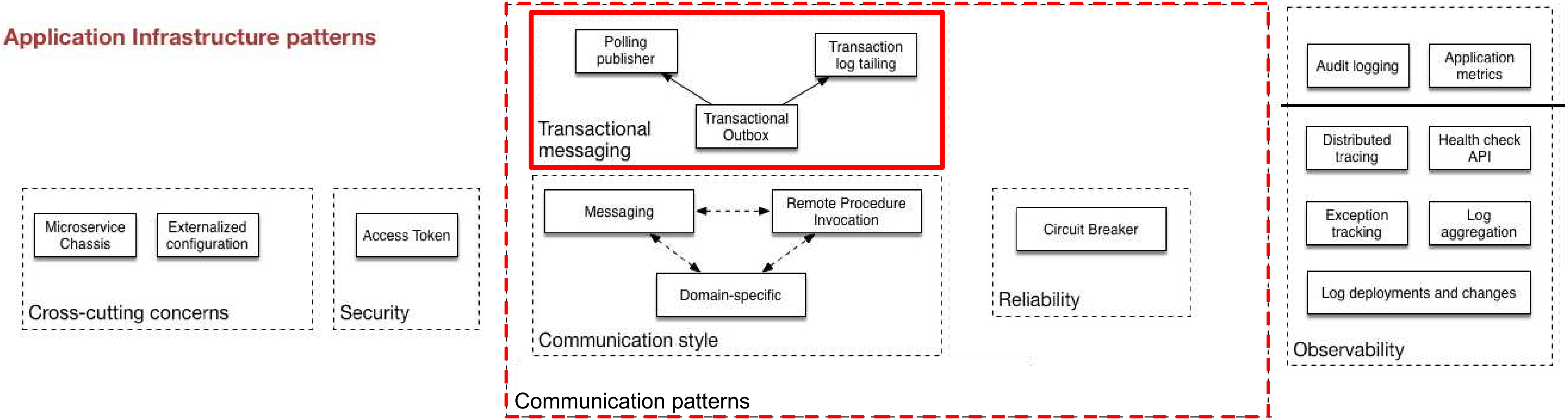
Perform inter-service communication using domain-specific protocols
or with 3rd party legacy systems

Examples of domain-specific protocols:

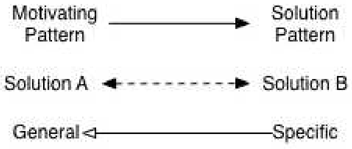
- File transfer protocols: FTP, SFTP, SCP, Sharepoint ...
- Email protocols: SMTP, IMAP
- Media streaming protocols: RTMP, HLS, HDS
- Conferencing : SIP, WebRTC
- Realtime : OMG DDS & RTPS, DDS-XRCE
- Cluster (Sci) : MPI (Broadcast, Scatter)
- ...

Microservices Patterns





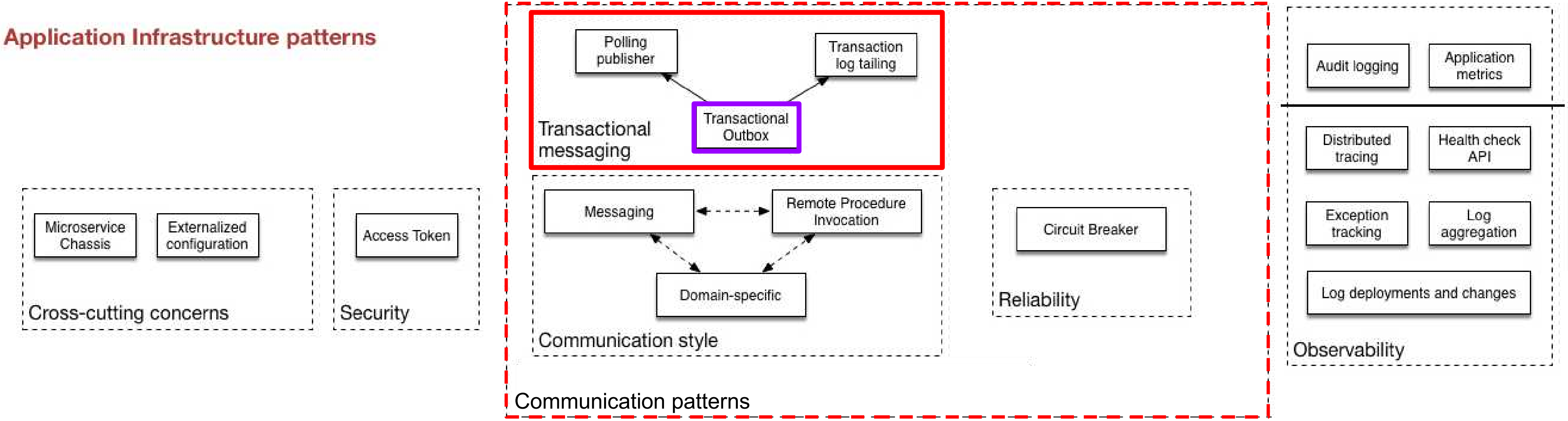
Microservices Patterns
Transactional Messaging



Motivating Pattern → Solution Pattern

Solution A ← - - - → Solution B

General ◁—— Specific



Application Infrastructure patterns

Microservice Chassis

Externalized configuration

Cross-cutting concerns

Access Token

Security

Polling publisher

Transaction log tailing

Transactional Outbox

Transactional messaging

Messaging

Remote Procedure Invocation

Domain-specific

Communication style

Circuit Breaker

Reliability

Communication patterns

Audit logging

Application metrics

Distributed tracing

Health check API

Exception tracking

Log aggregation

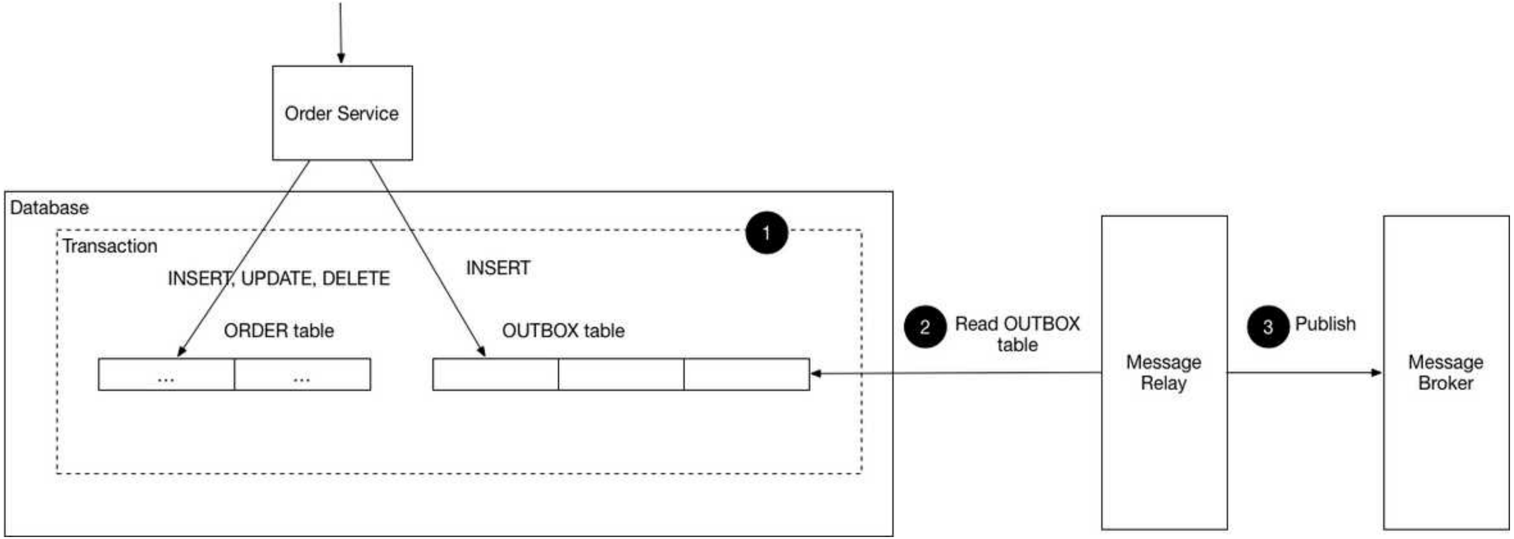
Log deployments and changes

Observability

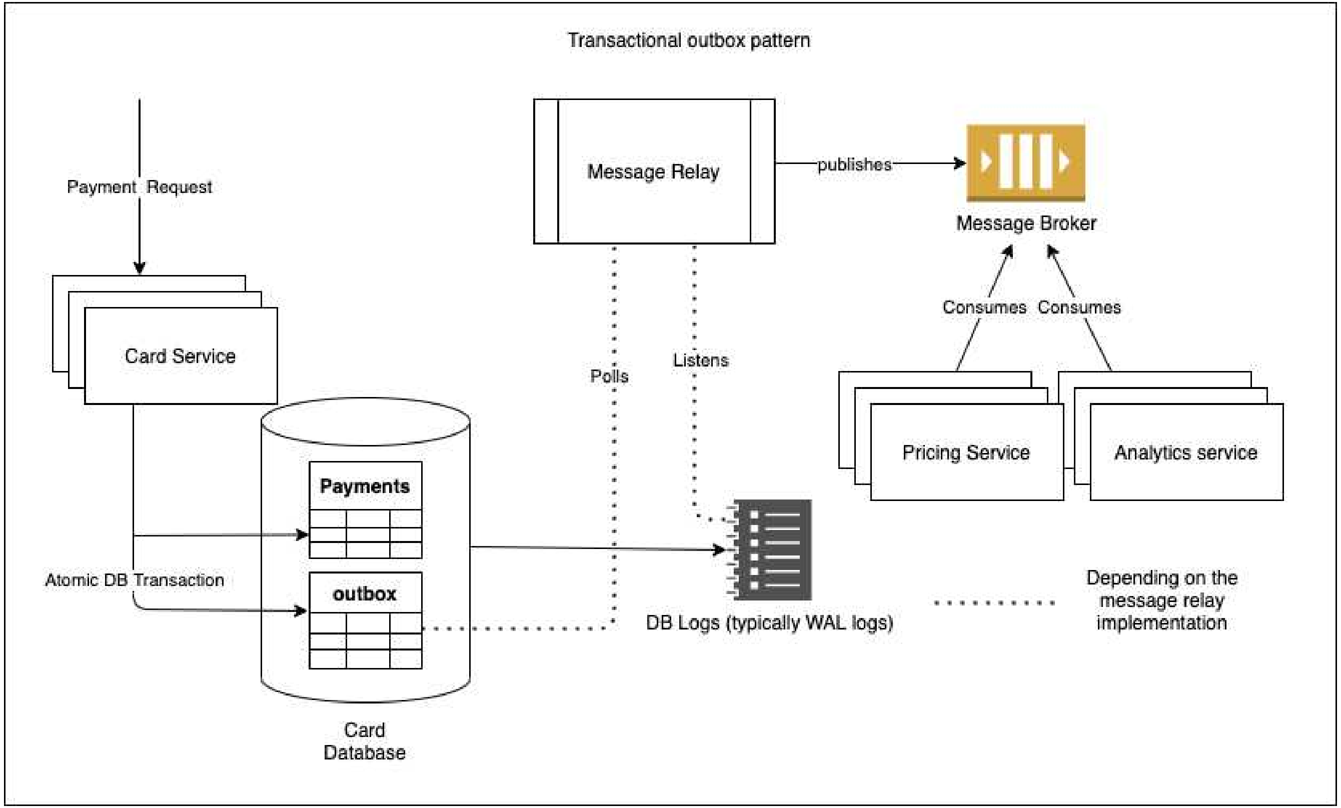
Transaction outbox pattern

A service typically need to atomically update the database and publish messages/events.

2PC is not an option!



Transactional Outbox Pattern



Transaction outbox pattern

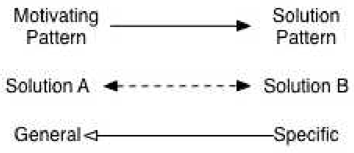
Advantages

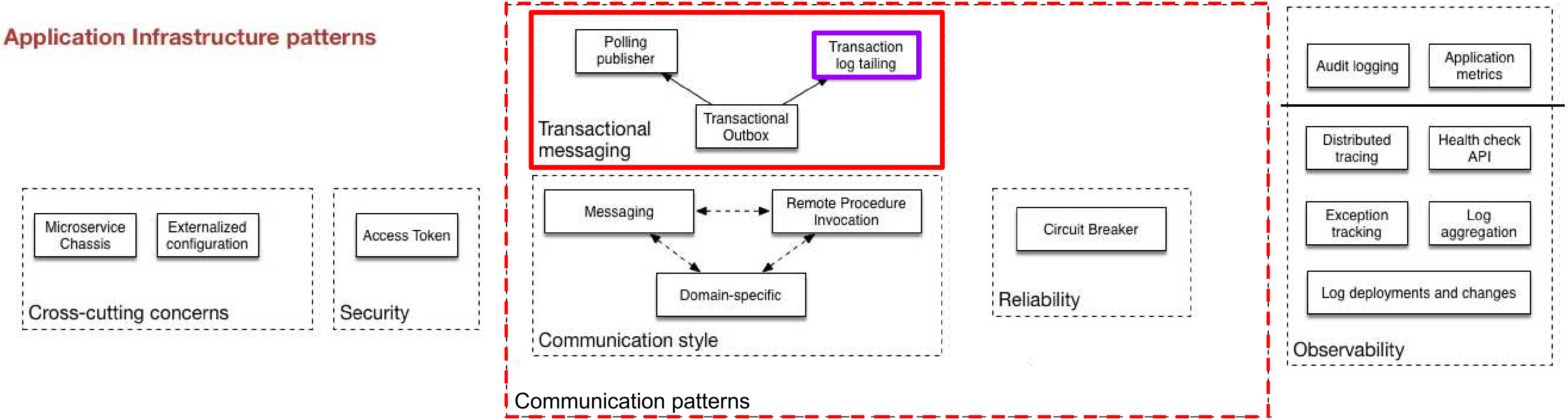
- No 2PC

Drawbacks

- Potentially error prone since the developer might forget to publish the message/event after updating the database.

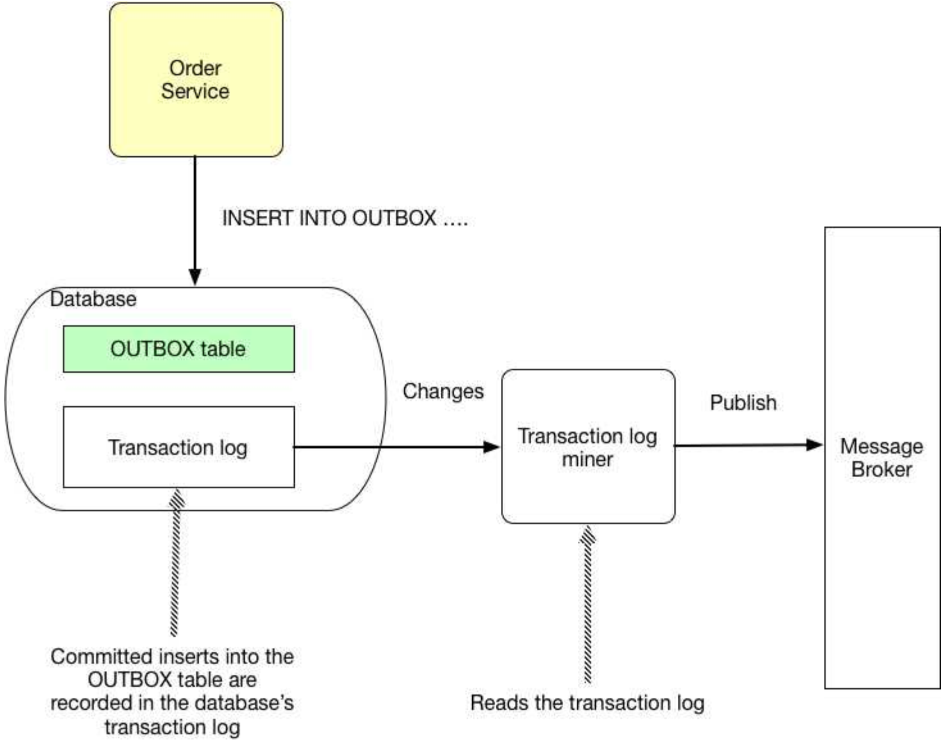
Microservices Patterns
Transactional Messaging





Transaction log tailing

Problem: How to publish messages/events into the outbox in the database to the message broker?



Transaction log tailing

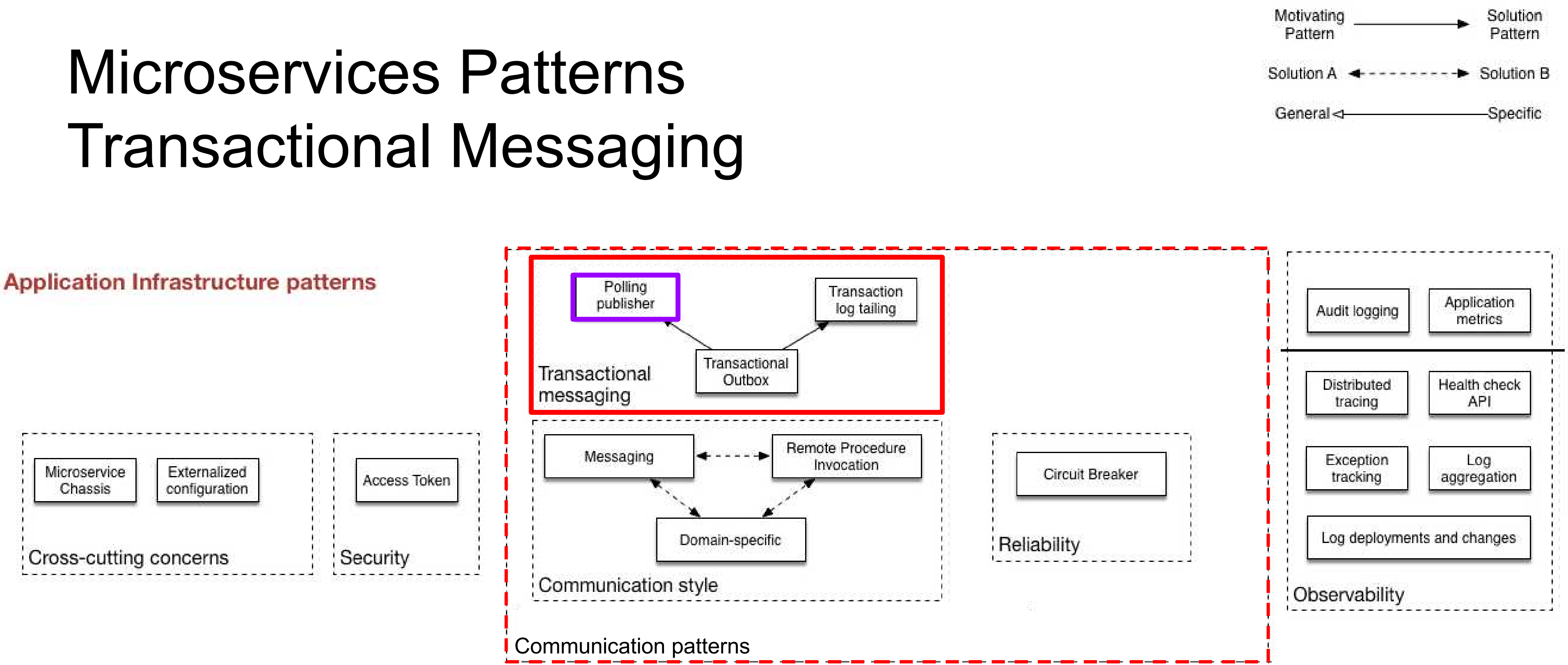
Advantages

- No 2PC
- Guaranteed to be accurate

Drawbacks

- Relatively obscure (but becoming increasingly common)
- Requires database specific solutions
- Tricky to avoid duplicate publishing

Microservices Patterns
Transactional Messaging



Polling publisher

Solution : Publish messages by polling the outbox in the database.

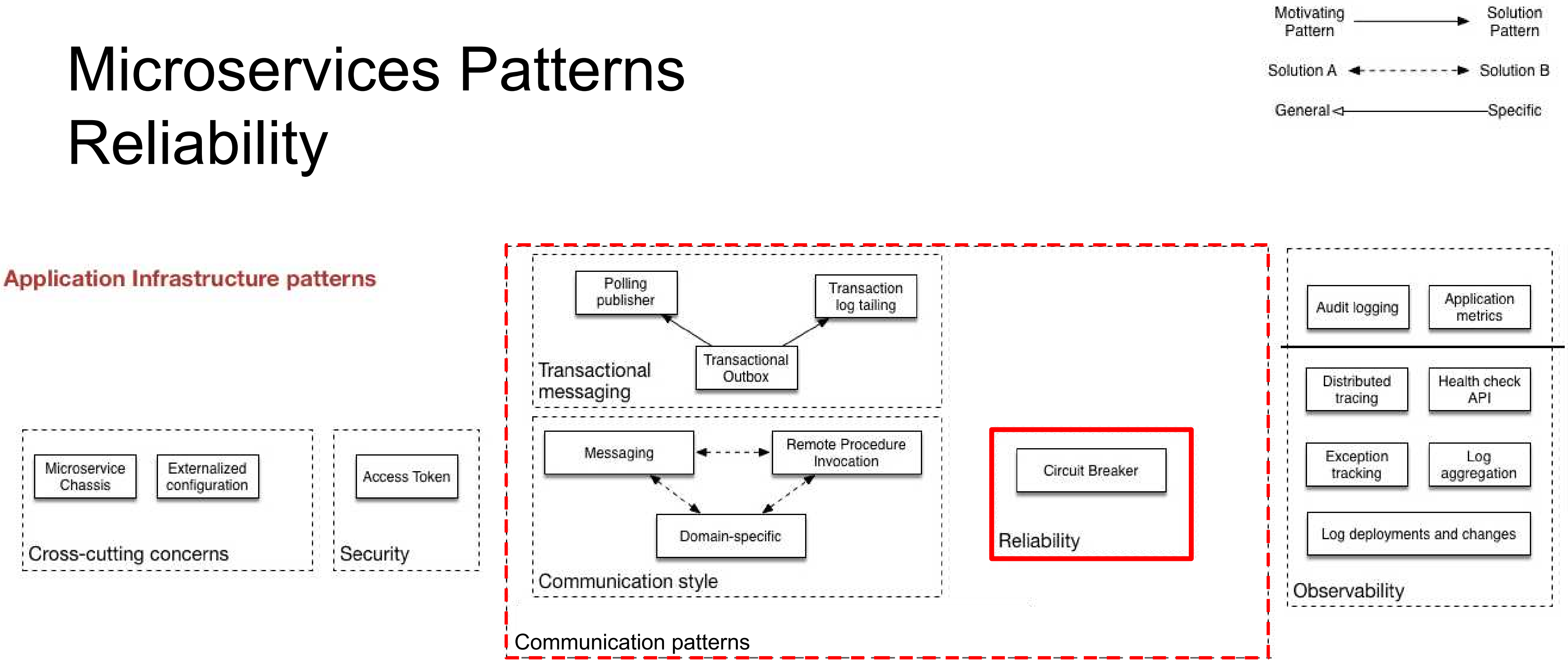
Advantages

- Works with any SQL database

Drawbacks

- Tricky to publish events in order
- Not all NoSQL databases support this pattern

Microservices Patterns
Reliability



Circuit Breaker

Context

service is unavailable

service is exhibiting high latency

lead to resource exhaustion in the caller and **failure cascades**

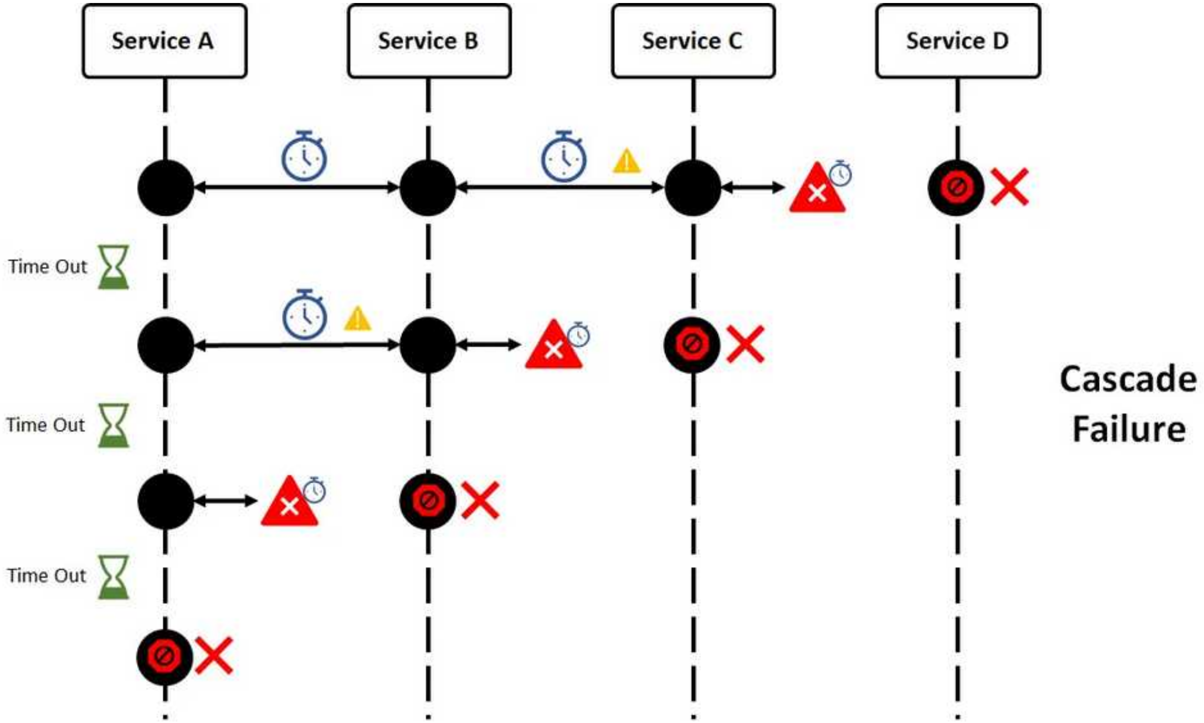
Problem: How to prevent a network or service failure from cascading to other services?

Circuit breaker

Client-side Proxy (RPI pattern) that functions in a similar fashion to an electrical circuit breaker.

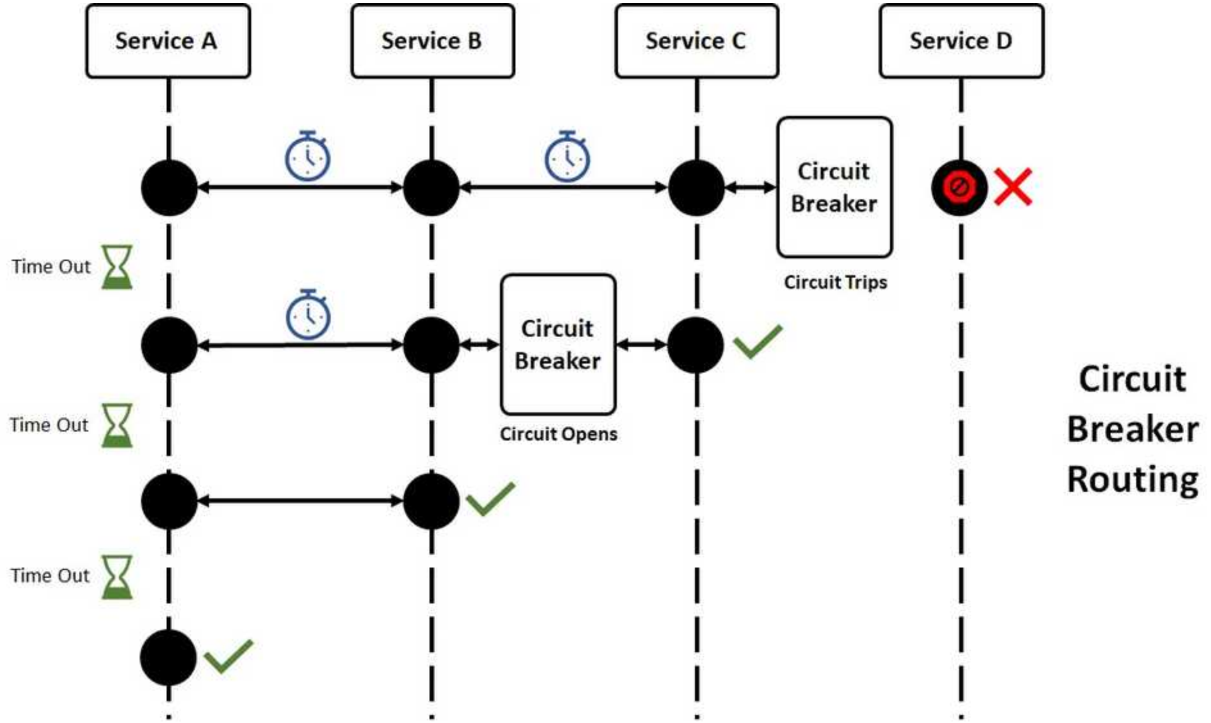
- When the number of consecutive failures crosses a threshold, the circuit breaker trips
- After the timeout expires the circuit breaker allows a limited number of test requests to pass through
- If those requests succeed the circuit breaker resumes normal operation
- Otherwise, if there is a failure the timeout period begins again

Circuit Breaker



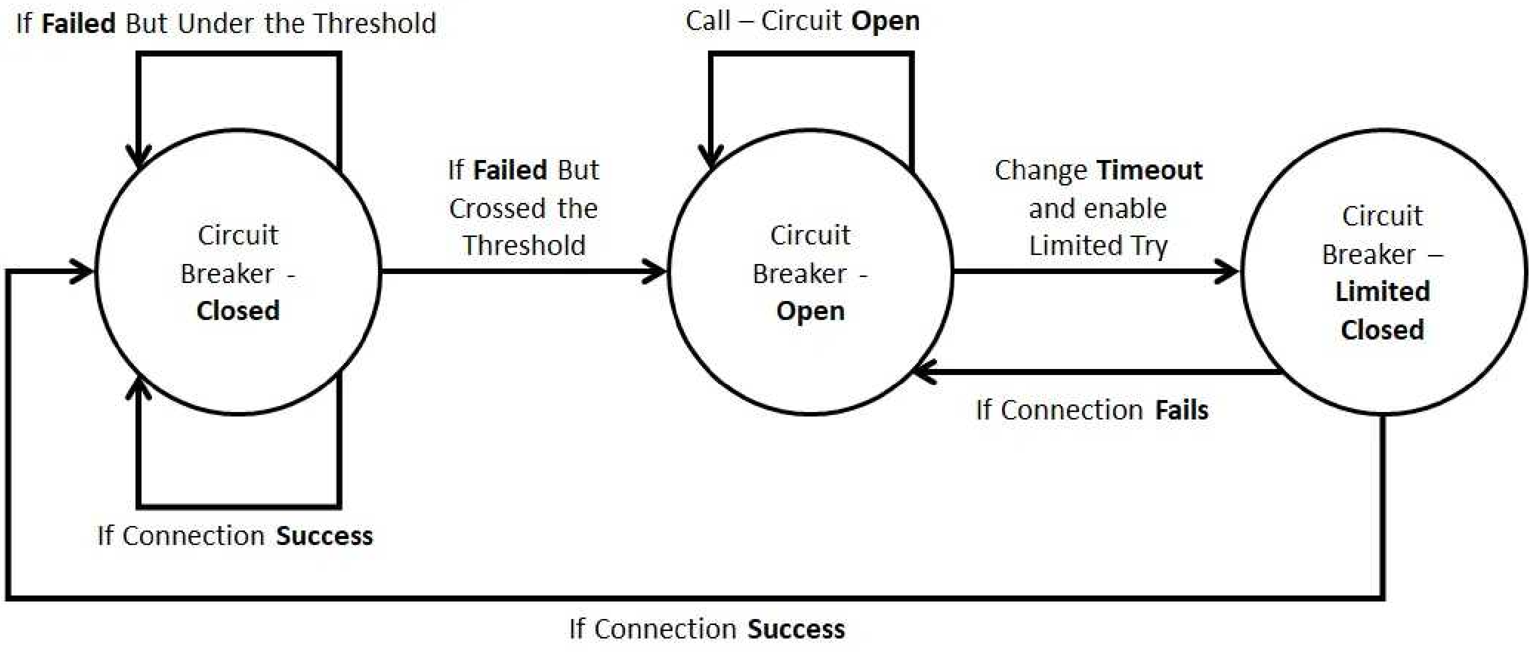
Infography from https://digitalvarys.com/what-is-circuit-breaker-design-pattern/

Circuit Breaker



Infography from https://digitalvarys.com/what-is-circuit-breaker-design-pattern/

Circuit Breaker


If **Failed** But Under the Threshold
Call – Circuit **Open**
Circuit Breaker - **Closed**
If **Failed** But Crossed the Threshold
Circuit Breaker - **Open**
Change **Timeout** and enable Limited Try
Circuit Breaker – **Limited Closed**
If Connection **Fails**
If Connection **Success**
If Connection **Success**

Infography from https://digitalvarys.com/what-is-circuit-breaker-design-pattern/

Circuit Breaker

Advantages:

- Services handle the failure of the services that they invoke

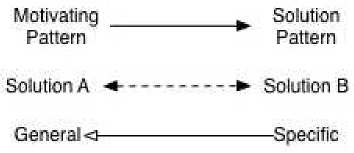
Drawbacks:

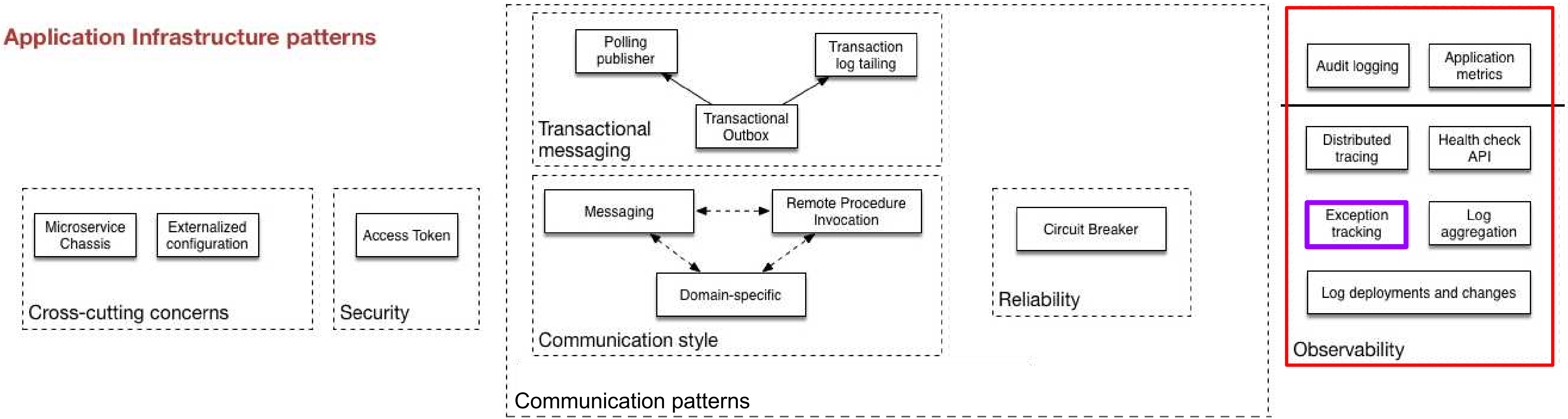
- choose timeout values without creating false positives or introducing excessive latency.

Exemple : Netflix Hystrix

https://dzone.com/articles/circuit-breaker-design-pattern-using-netflix-hystr

Microservices Patterns
Observability





Exception tracking

Errors sometimes occur when handling requests

- When an error occurs, a service instance throws an exception

Problem: How to understand the behavior of an application and troubleshoot problems?

- Exceptions must be de-duplicated, recorded, investigated by developers and the underlying issue resolved
- Any solution should have minimal runtime overhead

Exception tracking

Solution: Report all exceptions to a centralized exception tracking service that aggregates and tracks exceptions and notifies developers

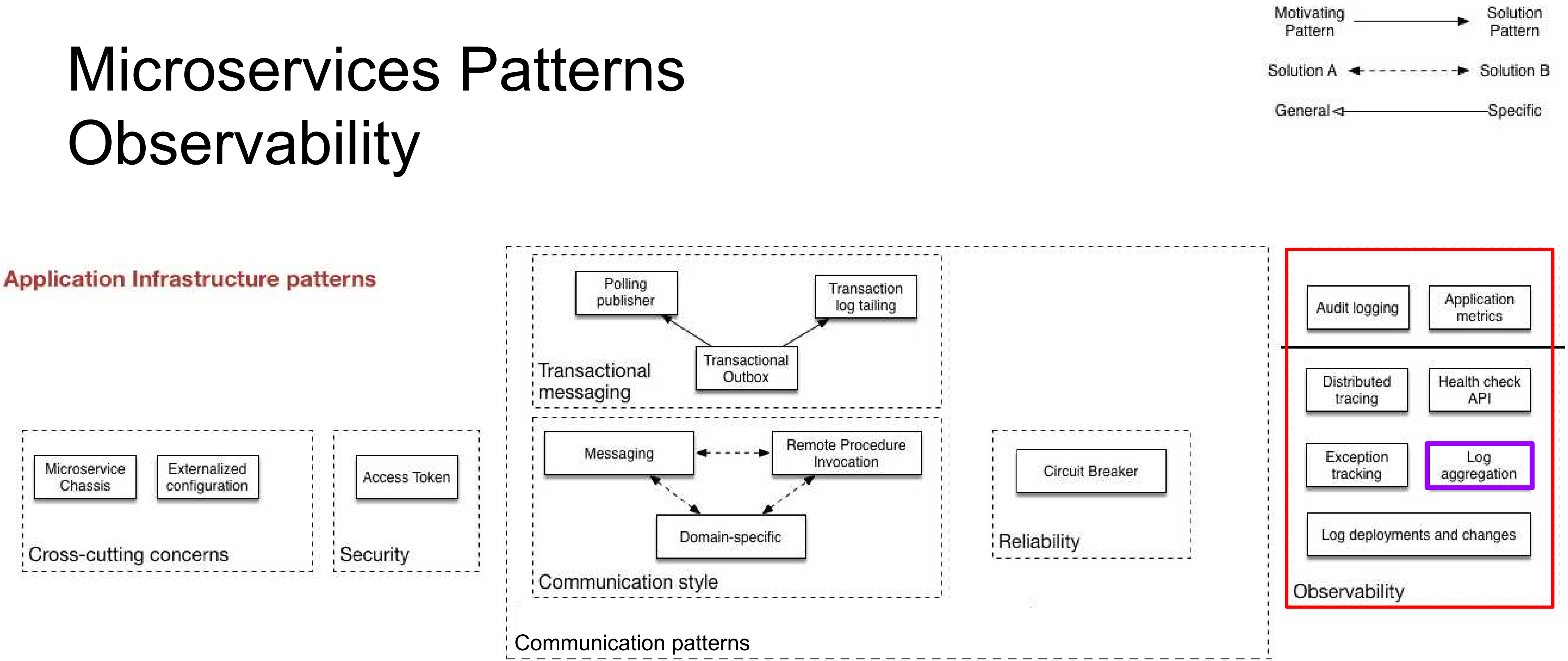
Advantages

- Make it easy to view exceptions and track their resolution

Drawbacks

- The exception tracking service is additional infrastructure

Microservices Patterns Observability



Log aggregation

Service instances write information to a log files in a standardized format

- The log file contains errors, warnings, information and debug messages

Problem: How to understand the behavior of an application and troubleshoot problems?

- Any solution should have minimal runtime overhead

Log aggregation

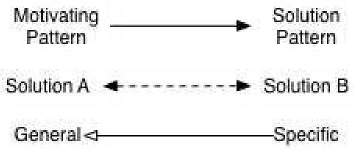
Solution:

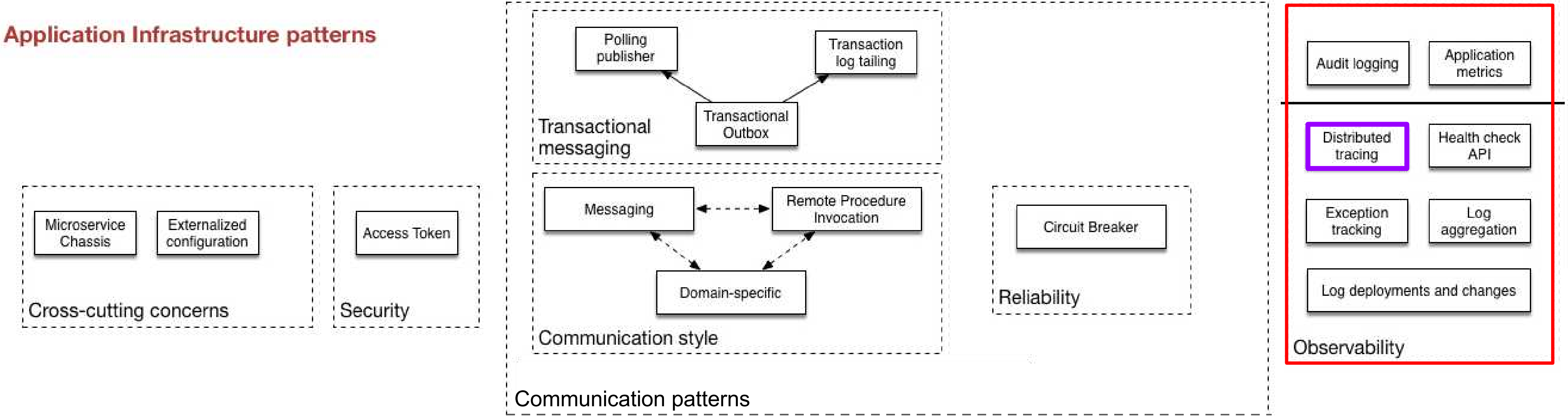
- Use a centralized logging service that aggregates logs from each service instance
- Users can:
  - search and analyze the logs
  - configure alerts that are triggered when certain messages appear in the logs

Examples: AWS Cloud Watch

Issue: handling a large volume of logs requires substantial infrastructure

Microservices Patterns
Observability





Distributed tracing

Requests often span multiple services

To handle a request, a service often perform several operations: database queries, message publications, etc.

Problem: How to understand the behavior of an application and troubleshoot problems?

- External monitoring only tells you the overall response time and number of invocations - no insight into the individual operations
- Any solution should have minimal runtime overhead
- Log entries for a request are scattered across numerous logs

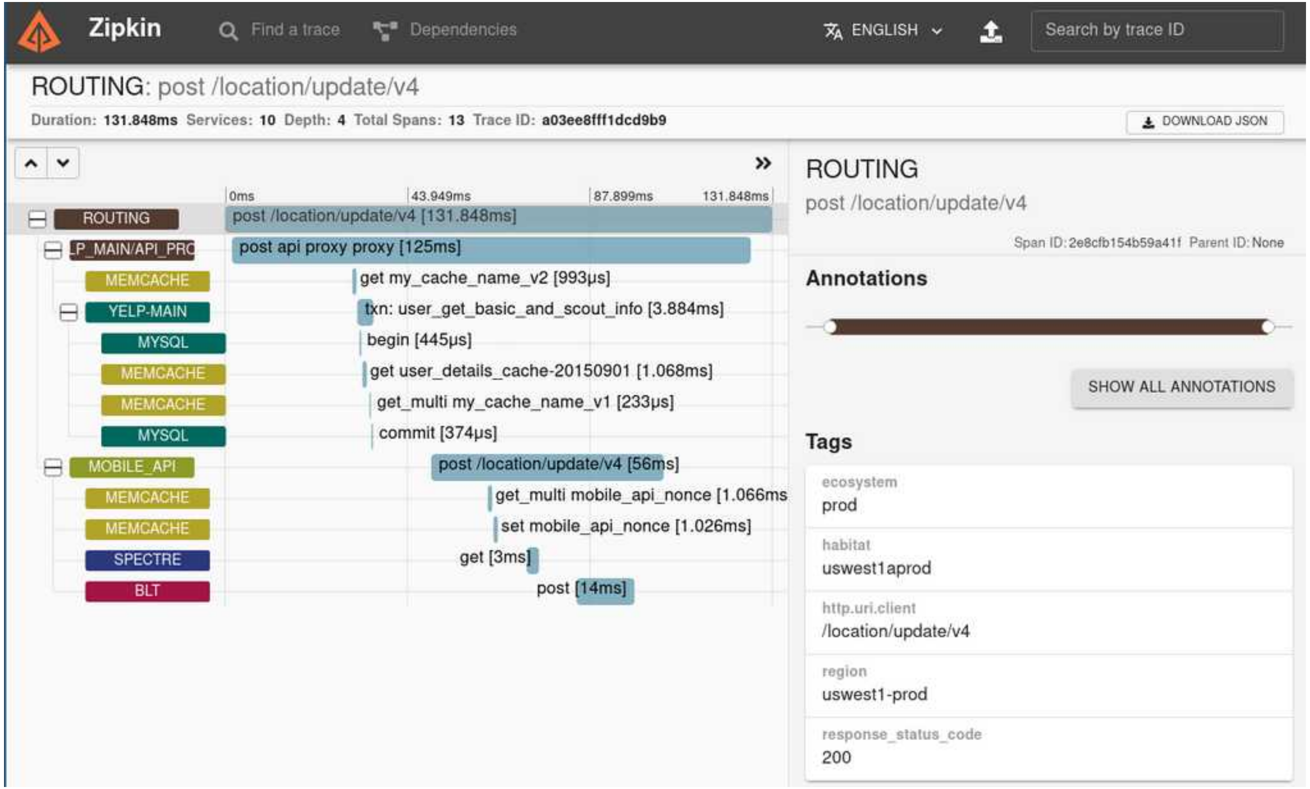
Distributed tracing

Solution: Instrument services to:

- Assign each external request a unique external request id
- Pass the external request id to all services that are involved in handling the request
- Include the external request id in all log messages
- Record information (e.g. start time, end time) about the requests and operations performed when handling an external request in a centralized service

Note: this instrumentation might done by a Microservice Chassis framework.

Distributed tracing



ROUTING: post /location/update/v4

Duration: 131.848ms Services: 10 Depth: 4 Total Spans: 13 Trace ID: a03ee8fff1dcd9b9

0ms | 43.949ms | 87.899ms | 131.848ms

ROUTING post /location/update/v4 [131.848ms]
LP_MAIN/API_PRC post api proxy proxy [125ms]
MEMCACHE get my_cache_name_v2 [993μs]
YELP-MAIN txn: user_get_basic_and_scout_info [3.884ms]
MYSQL begin [445μs]
MEMCACHE get user_details_cache-20150901 [1.068ms]
MEMCACHE get_multi my_cache_name_v1 [233μs]
MYSQL commit [374μs]
MOBILE_API post /location/update/v4 [56ms]
MEMCACHE get_multi mobile_api_nonce [1.066ms
MEMCACHE set mobile_api_nonce [1.026ms]
SPECTRE get [3ms]
BLT post [14ms]

ROUTING

post /location/update/v4

Span ID: 2e8cfb154b59a41f Parent ID: None

Annotations

Tags

ecosystem
prod

habitat
uswest1aprod

http.uri.client
/location/update/v4

region
uswest1-prod

response_status_code
200

Distributed tracing

Advantages:

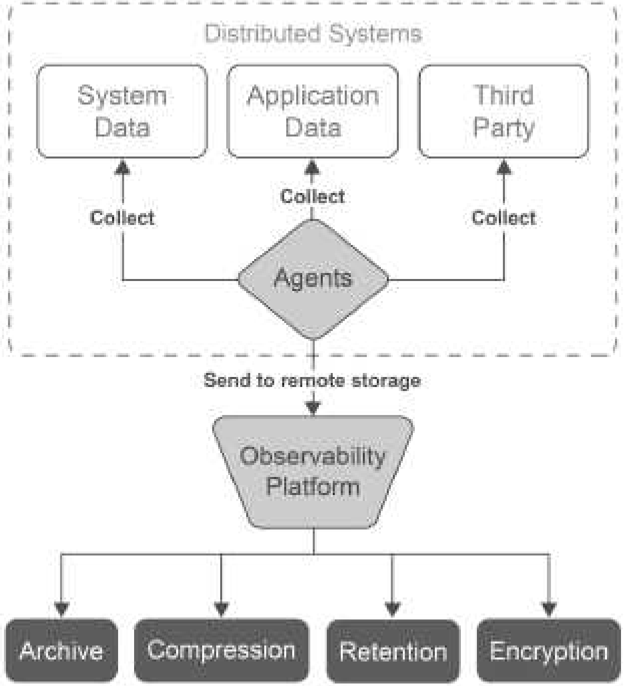
- Useful insight into the behavior of the system including the sources of latency
- Enables developers to see how an individual request is handled by searching across aggregated logs for its external request id

Drawbacks:

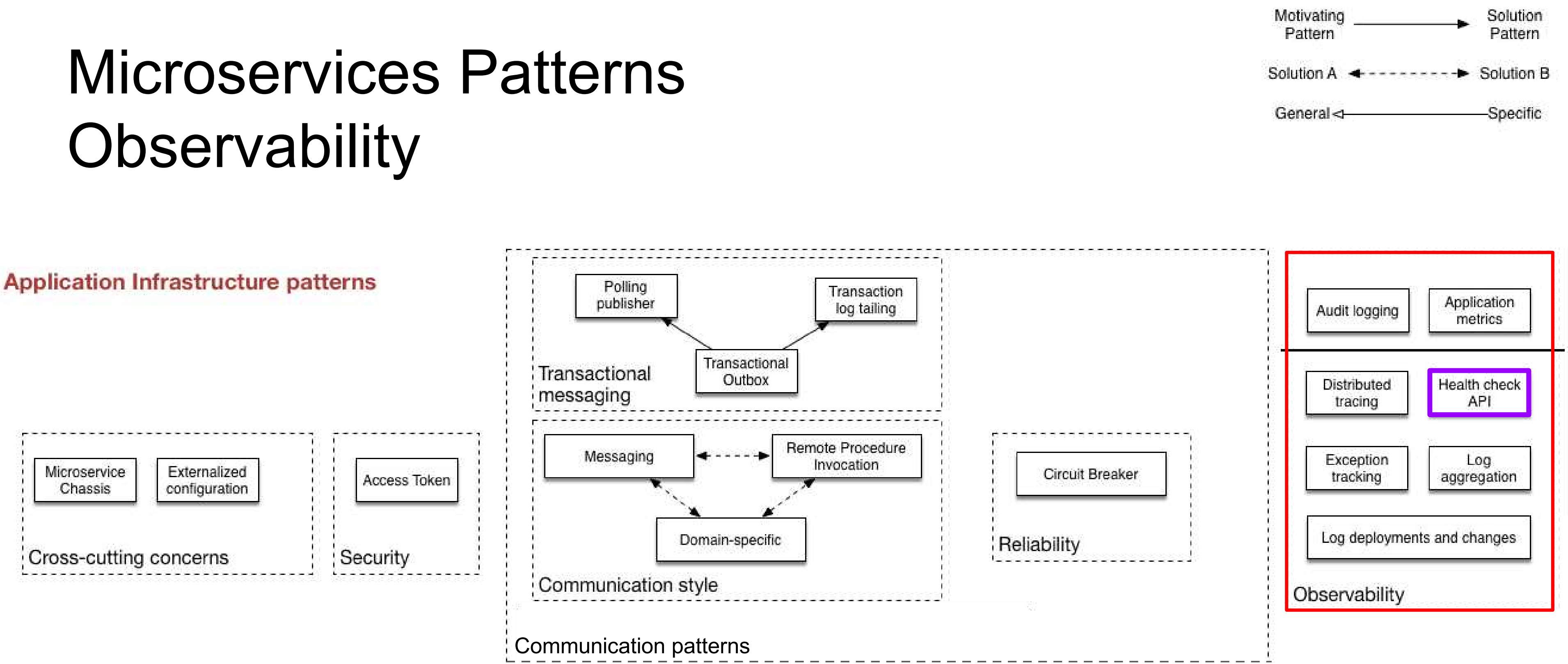
- Aggregating and storing traces can require significant infrastructure

Examples

- Zipkin, Jaeger, Opentelemetry, Opentracing, Datadog ...


Distributed Systems
System Data
Application Data
Third Party
Collect
Collect
Collect
Agents
Send to remote storage
Observability Platform
Archive
Compression
Retention
Encryption

Microservices Patterns Observability



Health Check API

A service instance can be incapable of handling requests yet still be running

- For example, it might have ran out of database connections

When this occurs:

- The monitoring system should generate a alert
- The load balancer or service registry should not route requests to the failed service instance

Problem: How to detect that a running service instance is unable to handle requests?

Health Check API

Solution:

- Implement, in each service, an health check API endpoint (e.g. HTTP /health) that returns the health of the service
- The health monitoring service (service registry or load balancer) periodically invokes the endpoint to check the health of the service instance

Health Check API

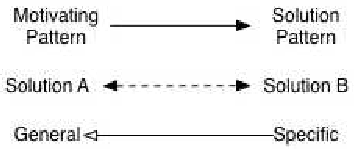
Advantages

- Enables the health of a service instance to be periodically tested

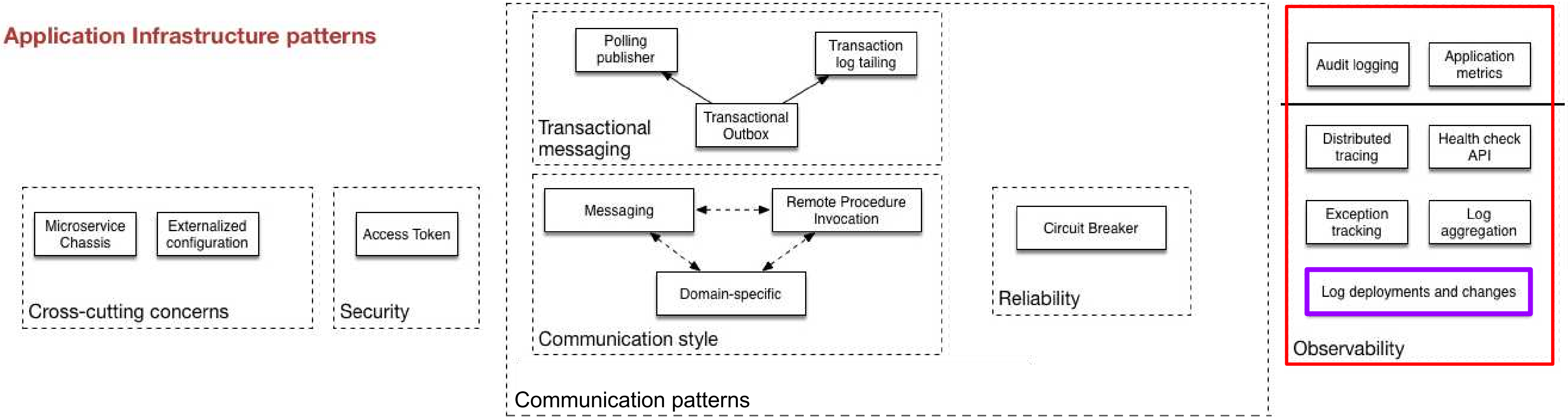
Drawbacks

- The health check might not be sufficiently comprehensive and so requests might still be routed to a failed service instance

Microservices Patterns



Application Infrastructure patterns



Log deployments and changes

Problem: How to understand the behavior of an application and troubleshoot problems?

- Note that it useful to track when deployments and other changes occur since issues usually occur immediately after a change

Solution: Log every deployment and every change to the (production) environment

Log deployments and changes

Examples:

- A deployment tool can publish a pseudo-metric whenever it deploys a new version of a service
- This metric can then be displayed alongside other metrics enabling changes in application behavior to be correlated with deployments

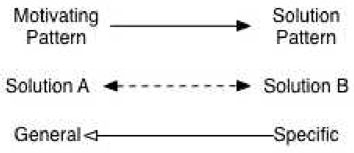
AWS Cloud Trail provides logs of AWS API calls

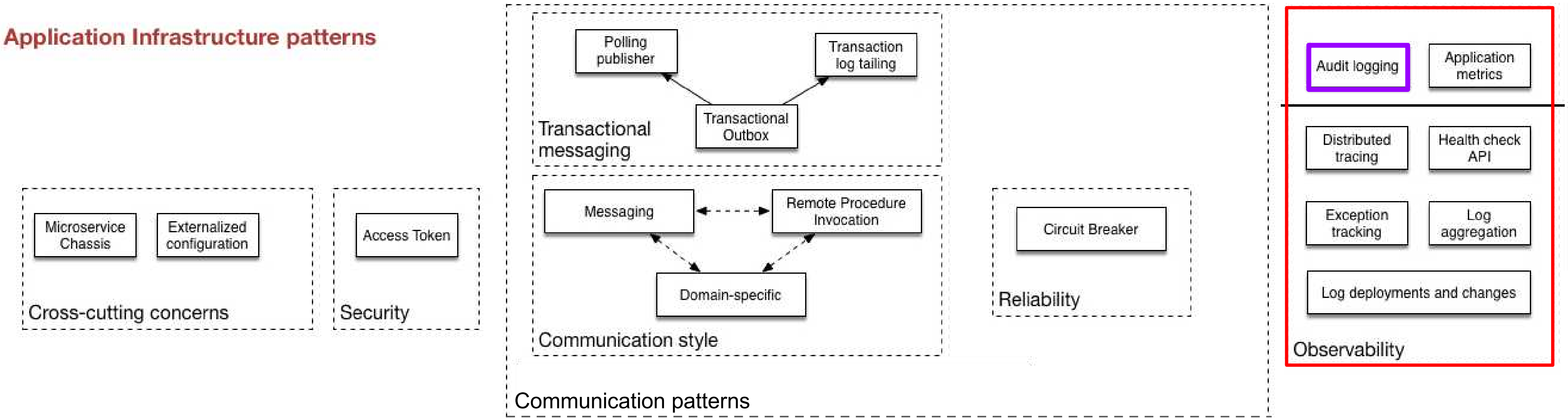
Log deployments and changes

Advantages

- Faster resolution of problems: deployments and changes can easily be correlated with observed issues

Microservices Patterns





Audit Logging

Problem: How to understand the behavior of users and the application and troubleshoot problems?

- It is useful to know what actions a user has recently performed: customer support, compliance, security, etc.

Solution: Record user activity in a database.

Audit Logging

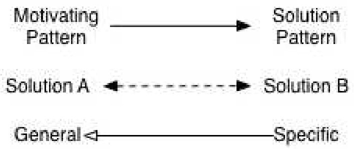
Advantages

- Provides a record of user actions

Drawbacks

- The auditing code is intertwined with the business logic → complexifies the business logic

Microservices Patterns

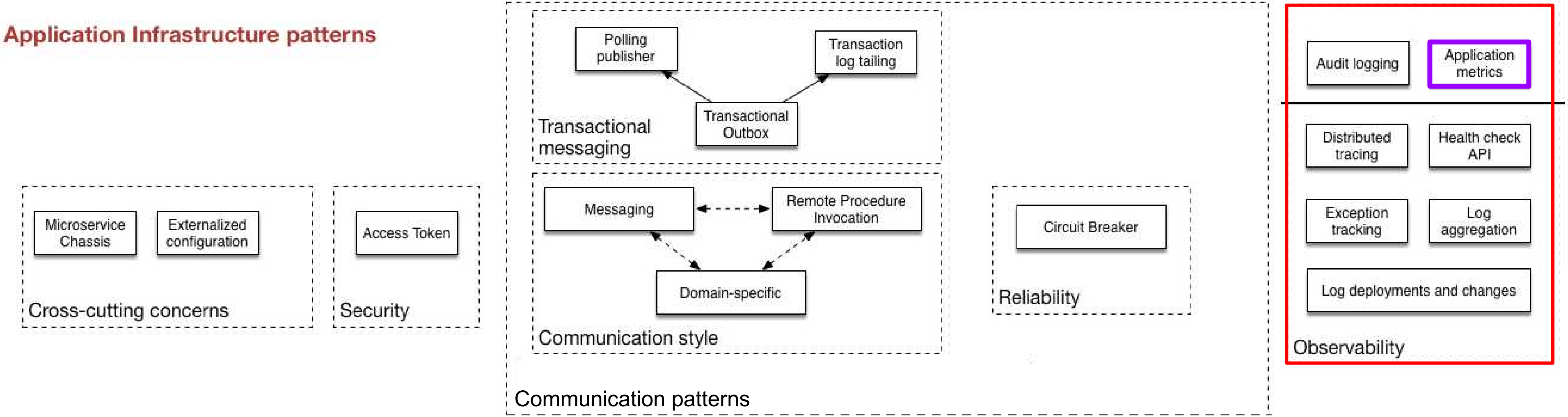


Motivating Pattern → Solution Pattern

Solution A ◂- - - - - - - -▸ Solution B

General ◁———— Specific

Application Infrastructure patterns



Cross-cutting concerns

Microservice Chassis

Externalized configuration

Security

Access Token

Communication patterns

Transactional messaging

Polling publisher

Transaction log tailing

Transactional Outbox

Communication style

Messaging

Remote Procedure Invocation

Domain-specific

Reliability

Circuit Breaker

Observability

Audit logging

Application metrics

Distributed tracing

Health check API

Exception tracking

Log aggregation

Log deployments and changes

Application Metrics

Problem: How to understand the behavior of an application and troubleshoot problems?

- The solution should have minimal runtime overhead

Solution:

- Instrument a service to gather statistics about individual operations
- Aggregate metrics in centralized metrics service
- Provides reporting and alerting

Two models for gathering metrics: push, pull

Application Metrics

Examples:

- Instrumentation libraries
  - Coda Hale/Yammer Java Metrics Library
  - Prometheus
  - Telegraf
- Metrics aggregation services
  - Prometheus
  - Kapacitor
  - AWS Cloud Watch

Application metrics

Advantages:

- Provide deep insight into application behavior

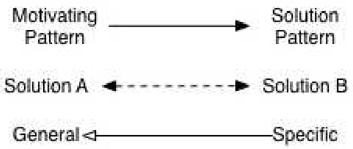
Drawbacks:

- Metrics code is intertwined with business logic

Issues:

- Aggregating metrics can require significant infrastructure

Microservices Patterns

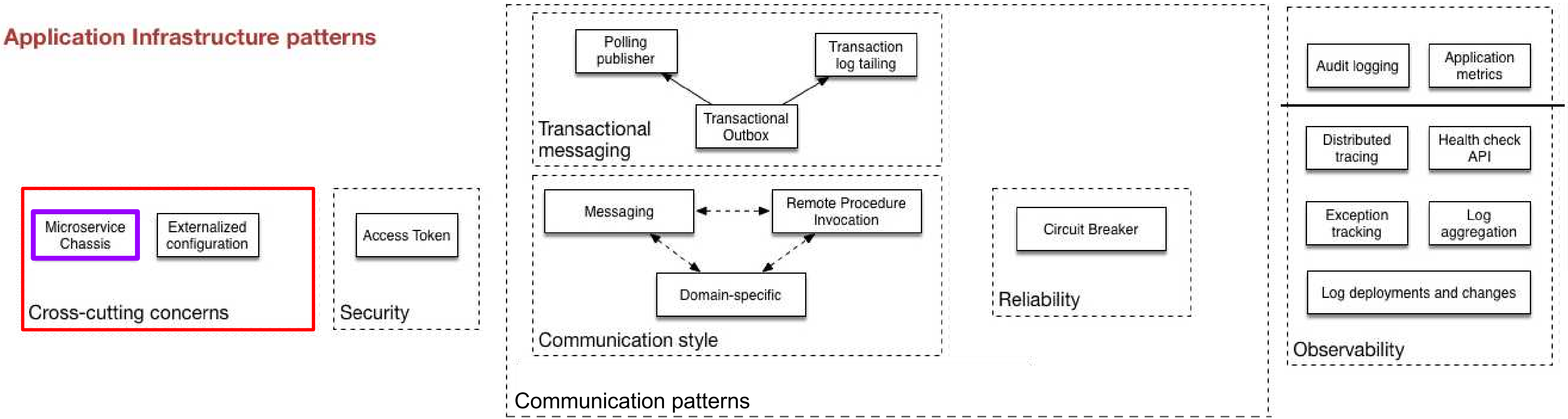


Motivating Pattern → Solution Pattern

Solution A ◂- - - - ▸ Solution B

General ◁—— Specific

Application Infrastructure patterns



Microservice Chassis

Externalized configuration

Cross-cutting concerns

Access Token

Security

Polling publisher

Transaction log tailing

Transactional Outbox

Transactional messaging

Messaging

Remote Procedure Invocation

Domain-specific

Communication style

Circuit Breaker

Reliability

Communication patterns

Audit logging

Application metrics

Distributed tracing

Health check API

Exception tracking

Log aggregation

Log deployments and changes

Observability

Microservice Chassis

- Many cross-cutting concerns:
  - Externalized configuration
  - Logging
  - Health checks
  - Metrics
  - Distributed tracing

- Tens or hundreds of services
  - Developers cannot afford to spend, for each service, a few days configuring the mechanisms to handle cross-cutting concerns

Microservice Chassis

Requirements:

- Creating a new microservice should be fast and easy

Solution: Build your microservices using a microservice chassis framework, which handles cross-cutting concerns

Examples of microservice chassis frameworks

- Java
  - Spring Boot and Spring Cloud, Dropwizard
- Go
  - Gizmo, Micro, Go kit

Microservice Chassis

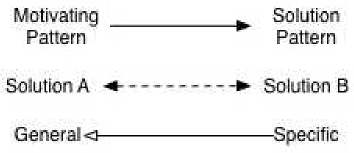
Advantages

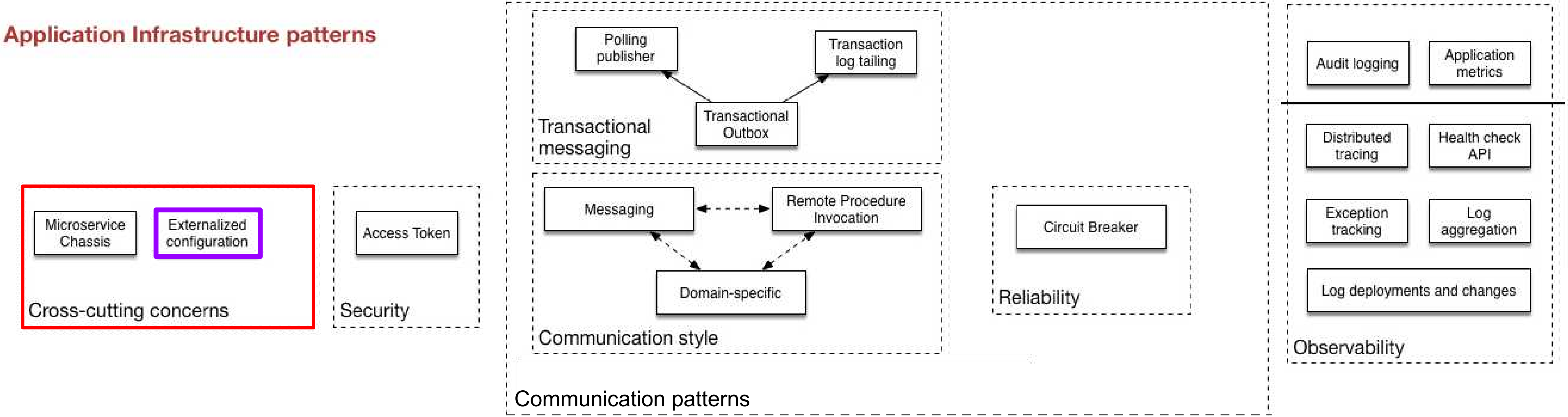
- Developers can quickly get started with developing a microservice

Drawbacks

- Obstacle to adopting a new programming language or framework
  - Requires a microservice chassis for each programming language/framework

Microservices Patterns





Externalized Configuration

An application typically uses one or more infrastructure and 3rd party services:

- Infrastructure services: Service registry, Message broker, Database server
- 3rd party services: payment processing, bulk email and messaging, etc.

Problem: How to enable a service to run in multiple environments without modification?

- A service must be provided with configuration explaining how it connects to the external/3rd party services
- A service must run in multiple environments (dev, test, qa, staging, production) without modification and/or recompilation
- Different environments have different instances of the external/3rd party services:
    - QA database vs. production database
    - Test credit card processing account vs. production credit card processing account

Externalized Configuration

Solution:

- Externalize all application configuration including the database credentials and network location
- On startup, a service reads the configuration from an external source, e.g. OS environment variables, etc.

Externalized Configuration

Example:

```
@Component
class RegistrationServiceProxy @Autowired()(restTemplate: RestTemplate) extends RegistrationService {

  @Value("${USER_REGISTRATION_URL}")
  var userRegistrationUrl: String = _
```

conf. injected at startup

dev Profile

application-**dev**.yml

```
web:
  image: sb_web
  ports:
    - "8080:8080"
  links:
    - eureka
  environment:
    USER_REGISTRATION_URL: http://REGISTRATION-SERVICE/user
```

Note:

`REGISTRATION-SERVICE` is the logical name of the service. It is resolved using Client-side discovery.

Externalized Configuration

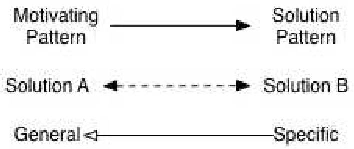
Advantages:

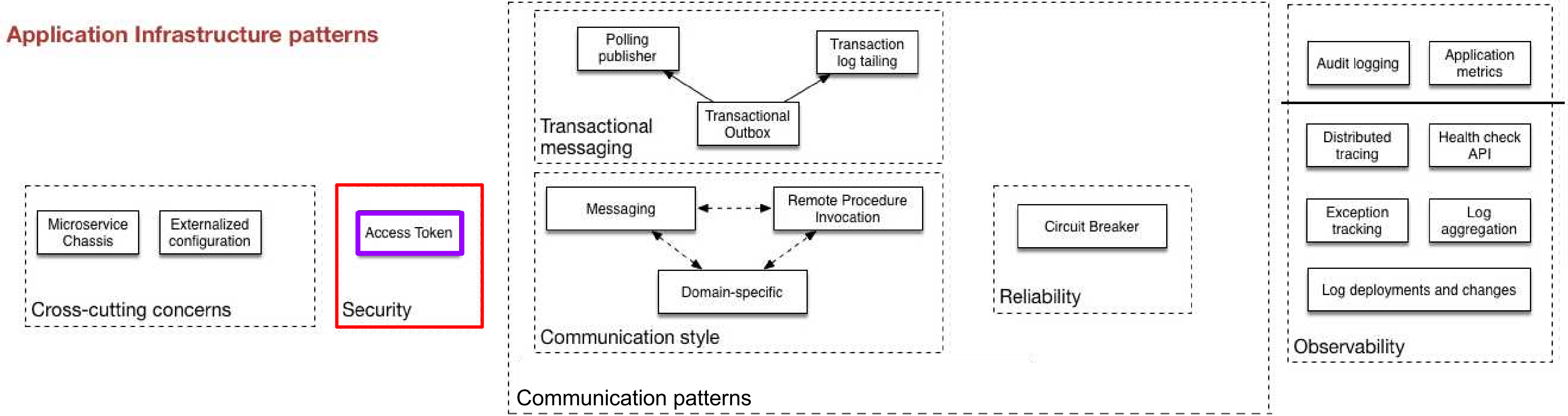
- The application runs in multiple environments without modification and/or recompilation

Issues:

- How to ensure that when an application is deployed the supplied configuration matches what is expected?

Microservices Patterns





Access Token

The API gateway authenticates requests, and forwards them to the services, which might in turn invoke other services.

Problem: How to communicate the identity of the requestor to the services that handle the request?

Solution:

- The API Gateway authenticates the request and passes an access token that securely identifies the requestor in each request to the services
- A service can include the access token in requests it makes to other services

Access Token

Advantages

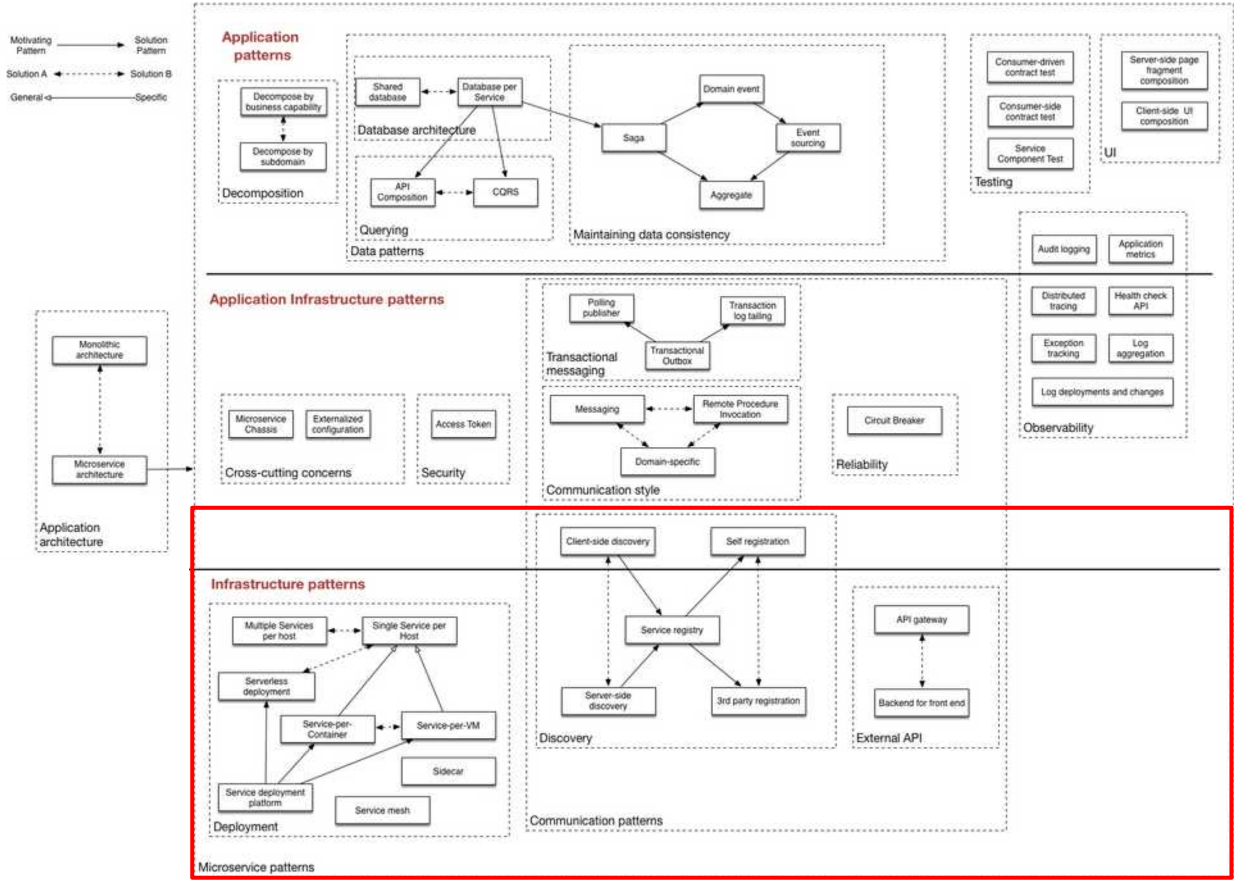
- The identity of the requestor is securely passed around the system
- Services can verify that the requestor is authorized (RBAC) to perform an operation

Examples

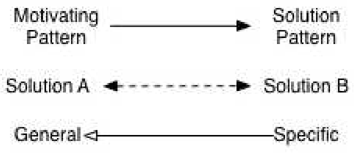
- JWT, OAuth2
- Identity managers: Keycloak, OpenAM, Okta (IMaaS) …

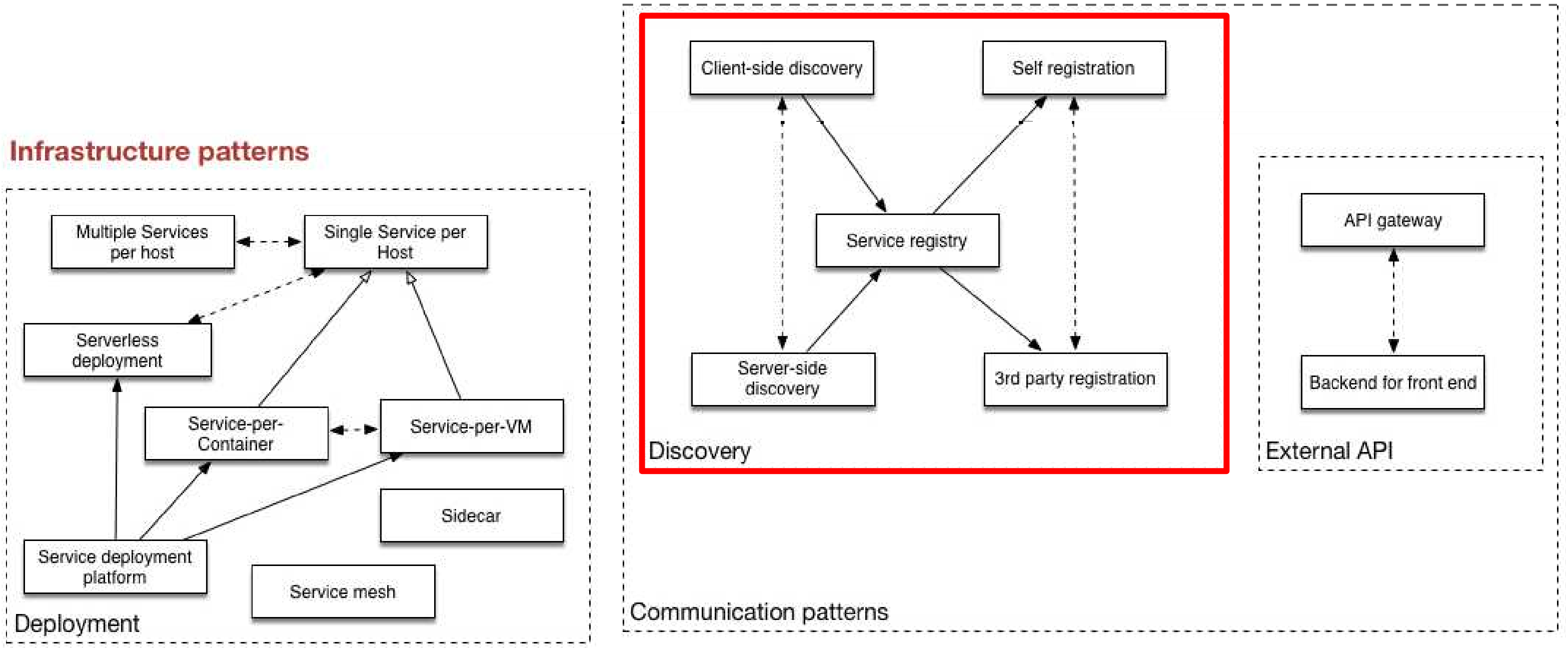
Exercice: have glance on JWT exchanged between JHipster generated frontend and backend and decode then with https://jwt.io/

Microservices Patterns



Microservices Patterns

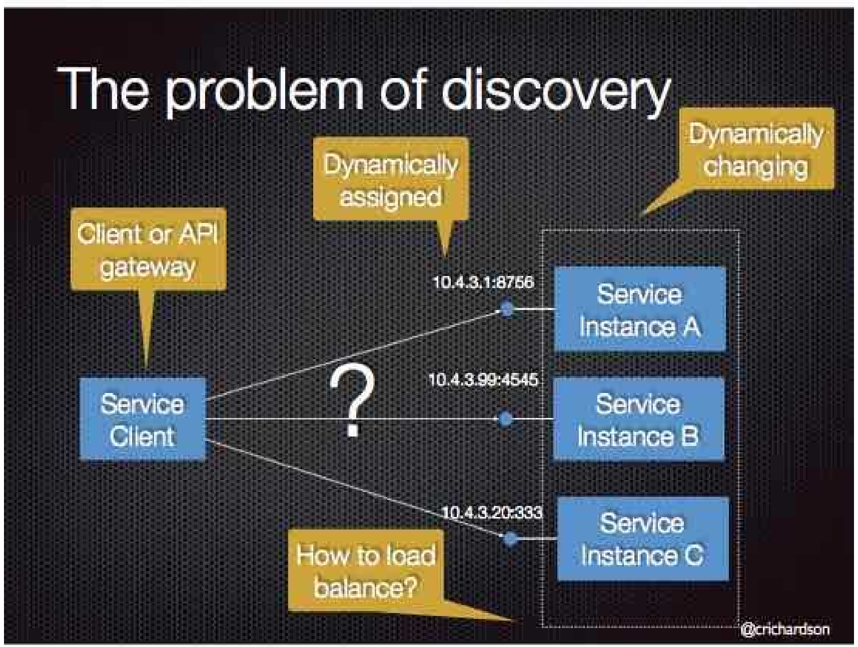




Service Discovery

Services need to call one another

- *Monolithic application*: services invoke one another through language-level method or procedure calls
- *Traditional distributed system*: services run at fixed, well known locations (hosts and ports)
- *Microservice-based application:* virtualized or containerized environments where **the number of instances of a service and their locations changes dynamically**

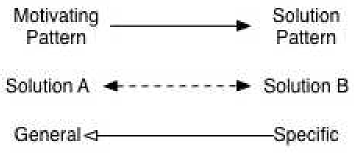


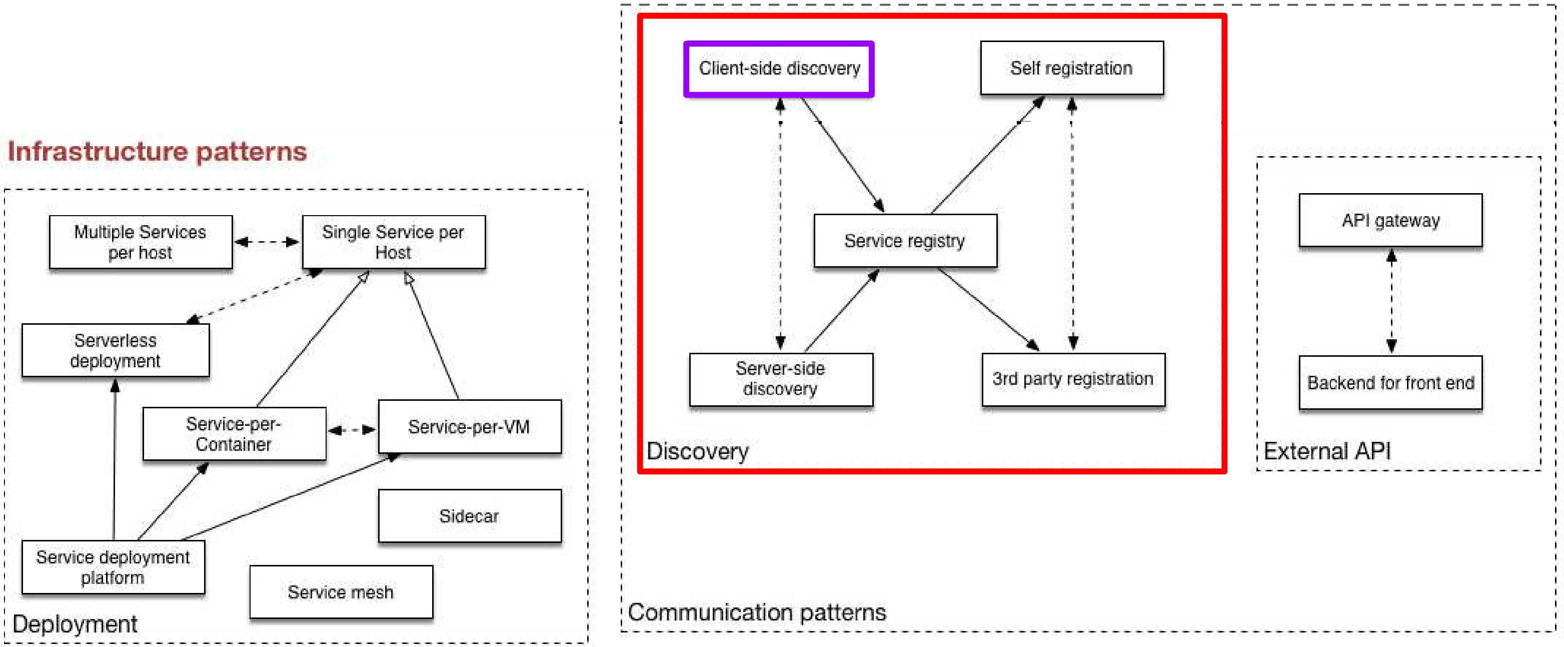
Service Discovery

How does the client of a service (the API gateway or another service) discover the location of a service instance?

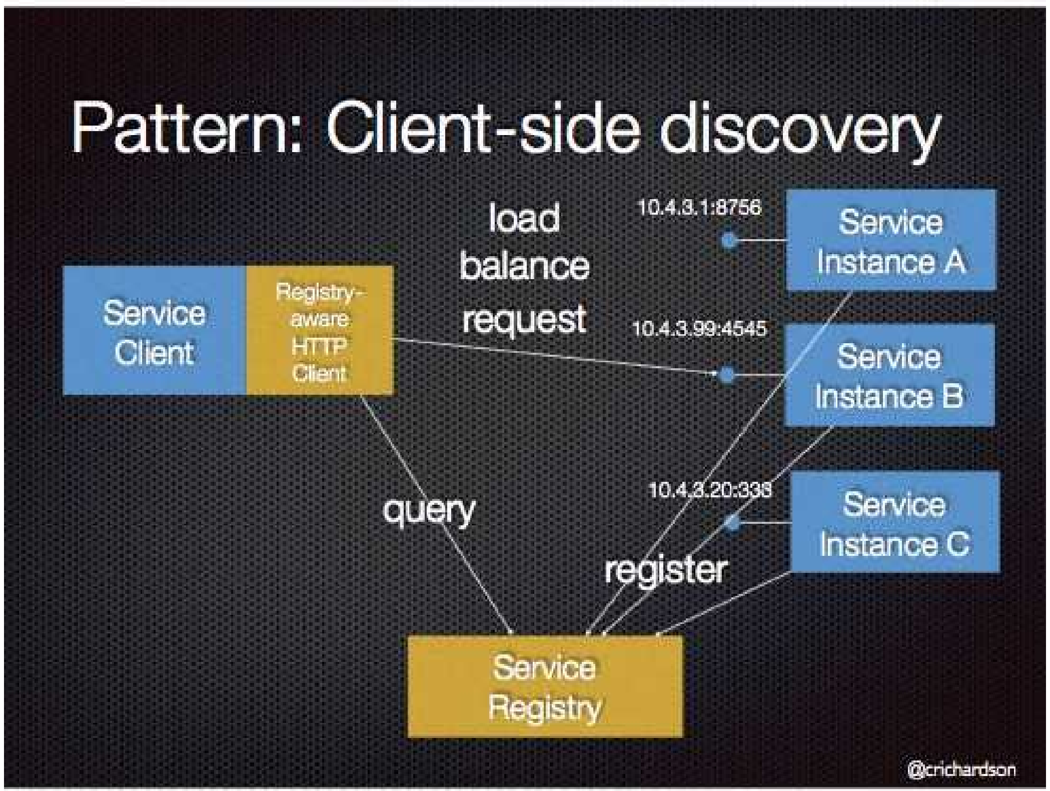
- Each instance of a service exposes a remote API
    - HTTP/REST, or Thrift etc. at a particular location (host and port)
- The number of services instances and their locations changes dynamically
- Virtual machines and containers are usually assigned dynamic IP addresses
- The number of services instances might vary dynamically (EC2 Autoscaling Group …)

Microservices Patterns





Client-side Service Discovery



Pattern: Client-side discovery

Client-side Service Discovery - Example

```
@Component
class RegistrationServiceProxy @Autowired()(restTemplate: RestTemplate) extends RegistrationService {

  @Value("${user_registration_url}")
  var userRegistrationUrl: String = _

  override def registerUser(emailAddress: String, password: String): Either[RegistrationError, String] = {

      val response = restTemplate.postForEntity(userRegistrationUrl,
        RegistrationBackendRequest(emailAddress, password),
        classOf[RegistrationBackendResponse])
       ...
}
```

Client-side Service Discovery - Example

```
@Configuration
@EnableEurekaClient
@Profile(Array("enableEureka"))
class EurekaClientConfiguration {

  @Bean
  @LoadBalanced // Ribbon
  def restTemplate(scalaObjectMapper : ScalaObjectMapper) : RestTemplate = {
    val restTemplate = new RestTemplate()
    restTemplate.getMessageConverters foreach {
      case mc: MappingJackson2HttpMessageConverter =>
        mc.setObjectMapper(scalaObjectMapper)
      case _ =>
    }
    restTemplate
  }
```

Client-side Service Discovery - Example

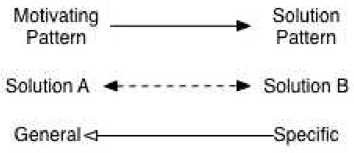
Advantages:

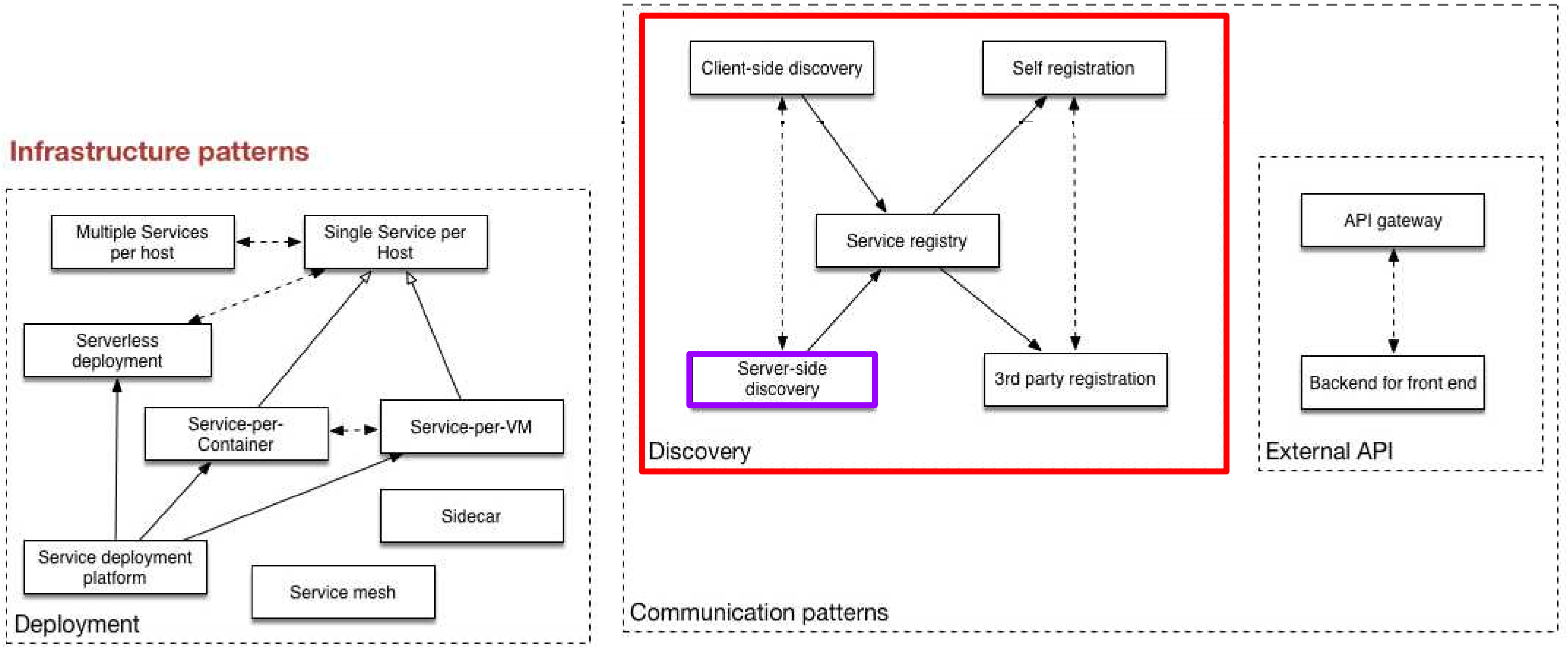
- Fewer moving parts and network hops compared to Server-side Discovery

Drawbacks:

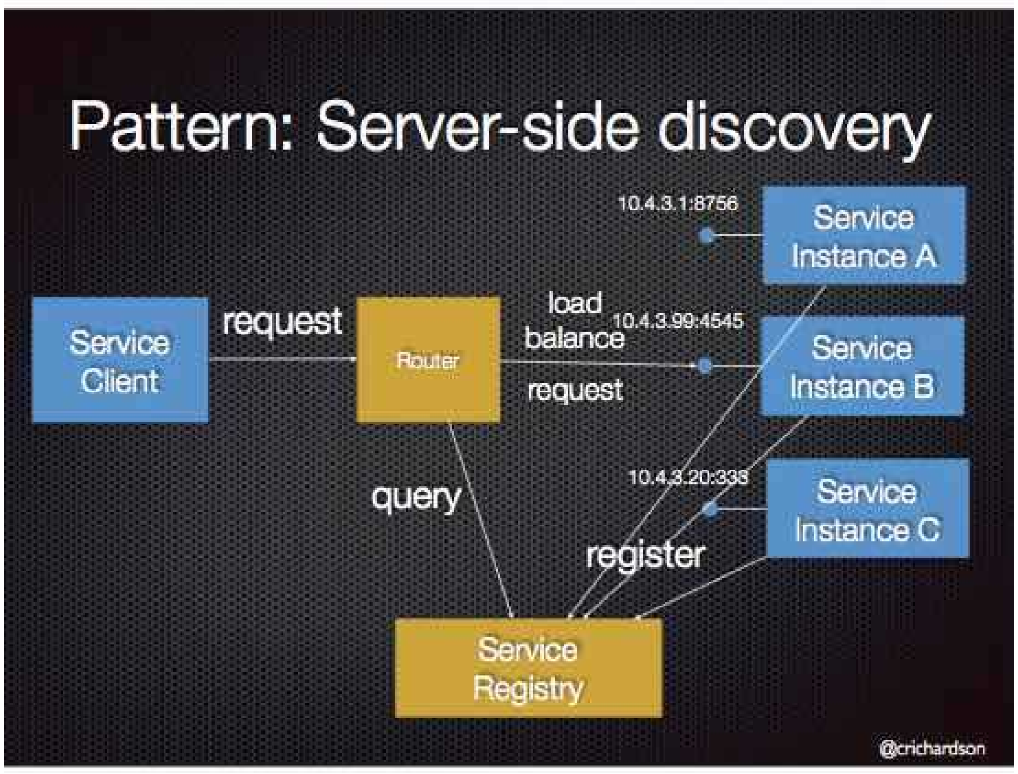
- This pattern couples the client to the Service Registry
- Developers need to implement client-side service discovery logic for each programming language/framework used by the application, e.g Java/Scala, JavaScript/NodeJS.
    - Netflix Prana provides an HTTP proxy-based approach to service discovery for non-JVM clients.

Microservices Patterns





Server-side Service Discovery



Server-side Service Discovery - Examples

AWS Elastic Load Balancer (ELB)

Clustering solutions such as Kubernetes and Marathon

Server-side Service Discovery

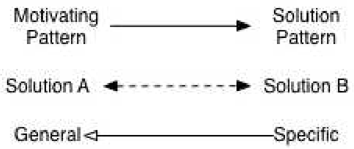
Advantages:

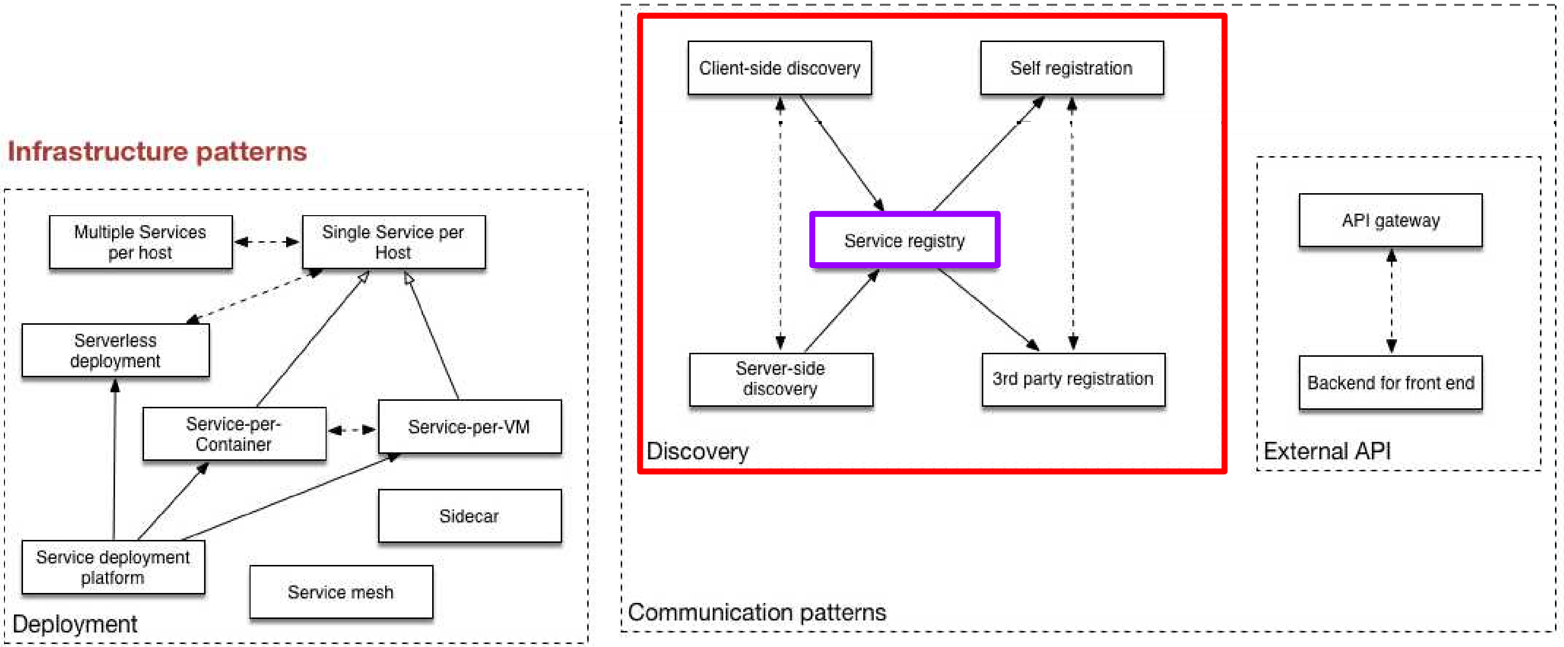
- Client code simpler than with client-side discovery
- Some cloud environments provide this functionality, e.g. AWS Elastic Load Balancer

Drawbacks:

- Unless it’s part of the cloud environment, the router is another system component that must be installed and configured (and replicated for availability and capacity)
- The router must support the necessary communication protocols (e.g HTTP, gRPC, Thrift, etc)
- More network hops are required than when using Client Side Discovery

Microservices Patterns





Service registry

Problem: How do clients of a service (Client-side discovery) and/or routers (Server-side discovery) know about the available instances of a service?

- Exposes a remote API (HTTP/REST, Thrift ...) at a particular location (host and port)
- Dynamic changes of number of services instances and their locations

Service registry

Solution:

A database of services instances, their instances and their locations

- register on startup
- deregistered on shutdown
- invoke a service instance’s health check API

Service registry - Examples

Examples

- Netflix Eureka, JHipster Registry
  - commonly used services: Apache Zookeeper, Consul, Etcd
- Implicit service registry
  - Kubernetes, Marathon, AWS ELB ...

Service registry

Advantages

- Client of the service and/or routers can discover the location of service instances

Drawbacks

- Yet another infrastructure component that must be setup, configured and managed.
  - The service registry is a critical system component!

Service registry

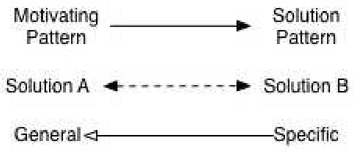
Two options to register service instances:

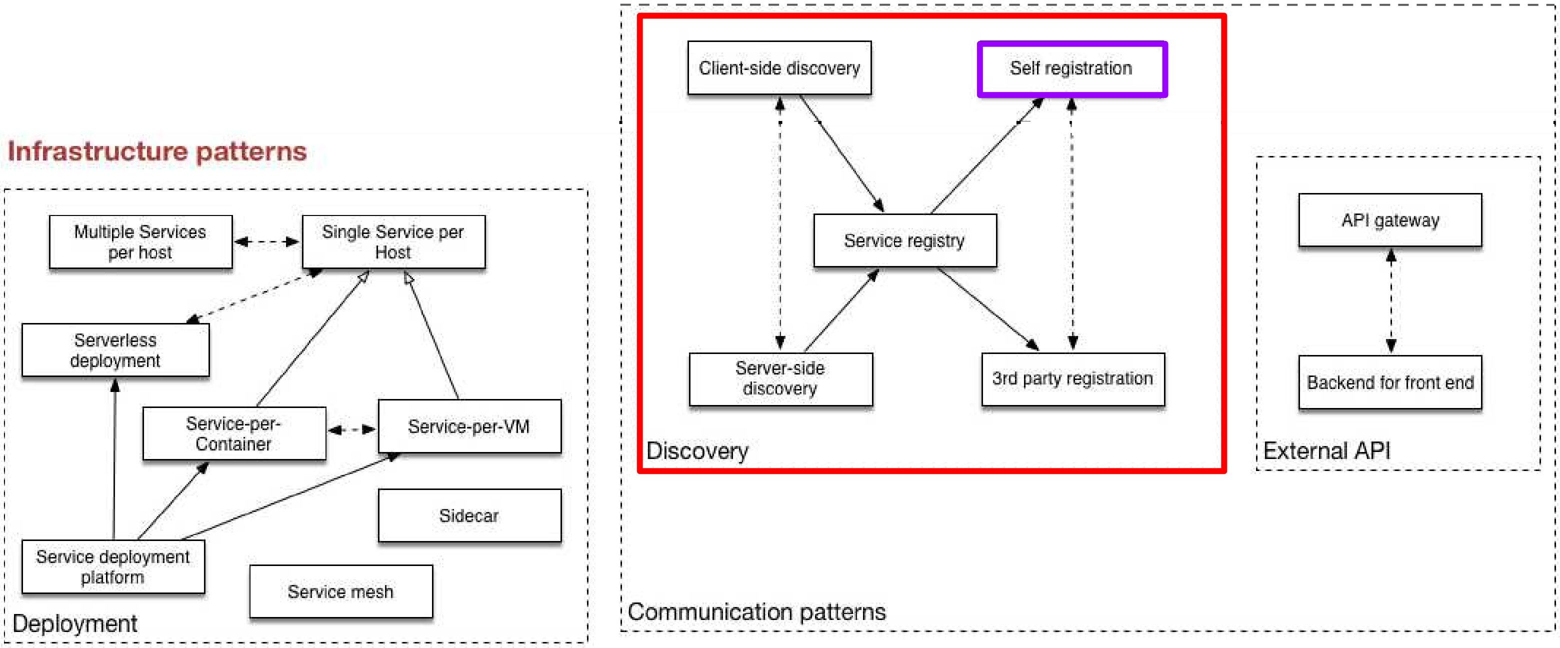
- Self registration pattern
- 3rd party registration pattern

Additional remarks:

- Service registry instances must be deployed on fixed and well known IP addresses.
- Clients are configured with those IP addresses.

Microservices Patterns





Self-Registration pattern

Problem: How are service instances registered with and unregistered from the service registry?

- Service instances must be registered with the service registry on startup and unregistered on shutdown
- Service instances that crash must be unregistered from the service registry
- Service instances that are running but incapable of handling requests must be unregistered from the service registry

Self-Registration pattern

Solution:

- A service instance is responsible for registering itself with the service registry
  - On **startup** the service instance registers itself (host and IP address) with the service registry
  - The client must **periodically** renew its registration so that the registry knows it is still alive
  - On **shutdown**, the service instance unregisters itself from the service registry

Self-Registration pattern

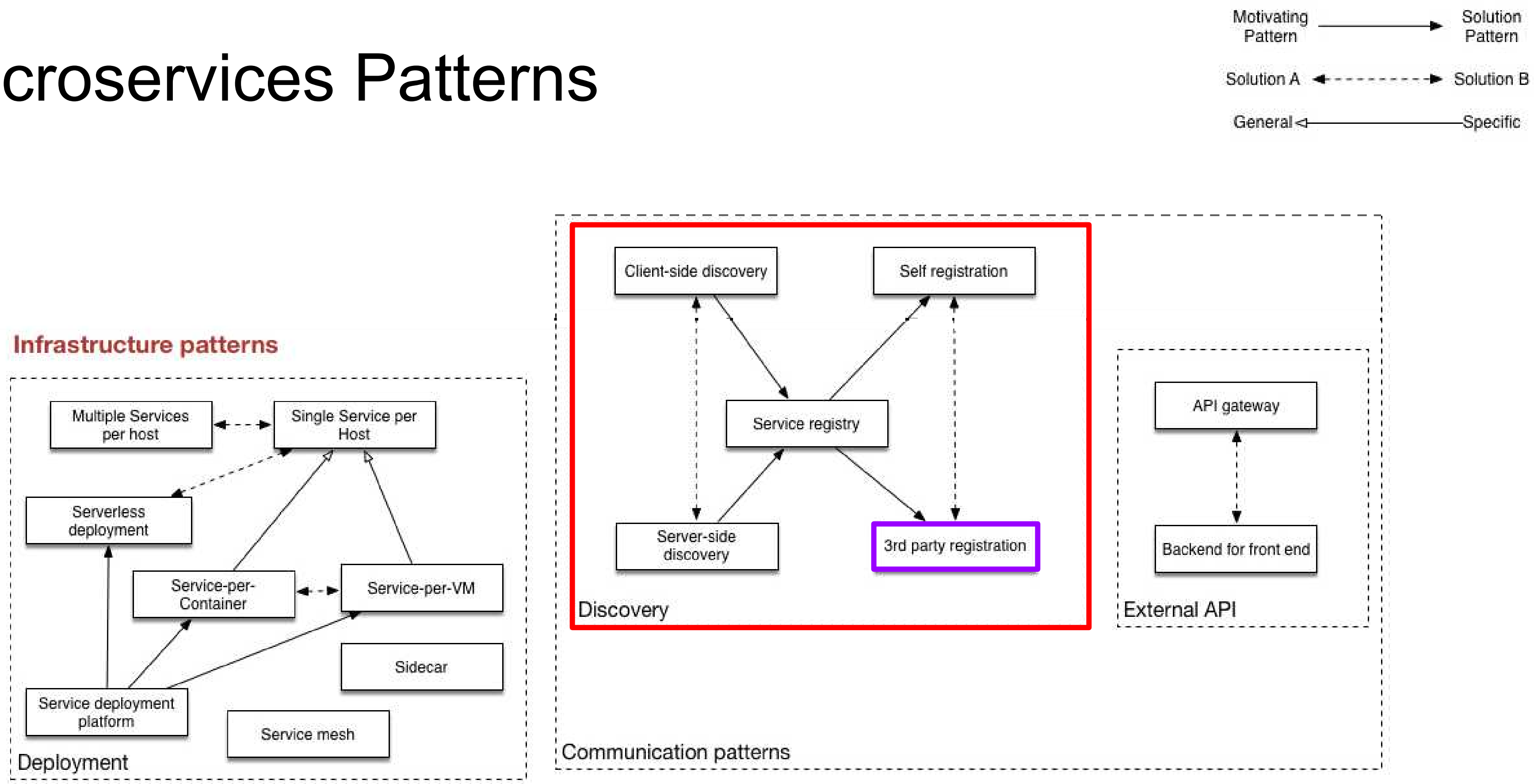
Advantages

- A service instance knows its own state and can refined state model: “STARTING, AVAILABLE, …” rather than “UP/DOWN”

Drawbacks

- Couples the service to the Service Registry
    - Developers must implement service registration logic in each programming language/framework that they use to write your services, e.g. NodeJS/JavaScript, Java/Scala, etc.
    - A service instance that is running yet unable to handle requests will often lack the self-awareness to unregister itself from the service registry

Microservices Patterns



3rd Party Registration pattern

Solution:

- A 3rd party registrar is responsible for registering and unregistering a service instance with the service registry
- When the service instance starts up, the registrar registers the service instance with the service registry
- When the service instance shuts downs, the registrar unregisters the service instance from the service registry

Examples:

- Netflix Prana, AWS Autoscaling Groups, Joyent's Container buddy, Registrator, Clustering frameworks such as Kubernetes and Marathon

3rd Party Registration pattern

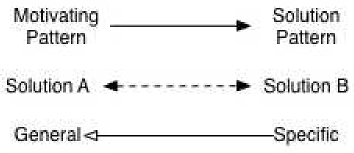
Advantages

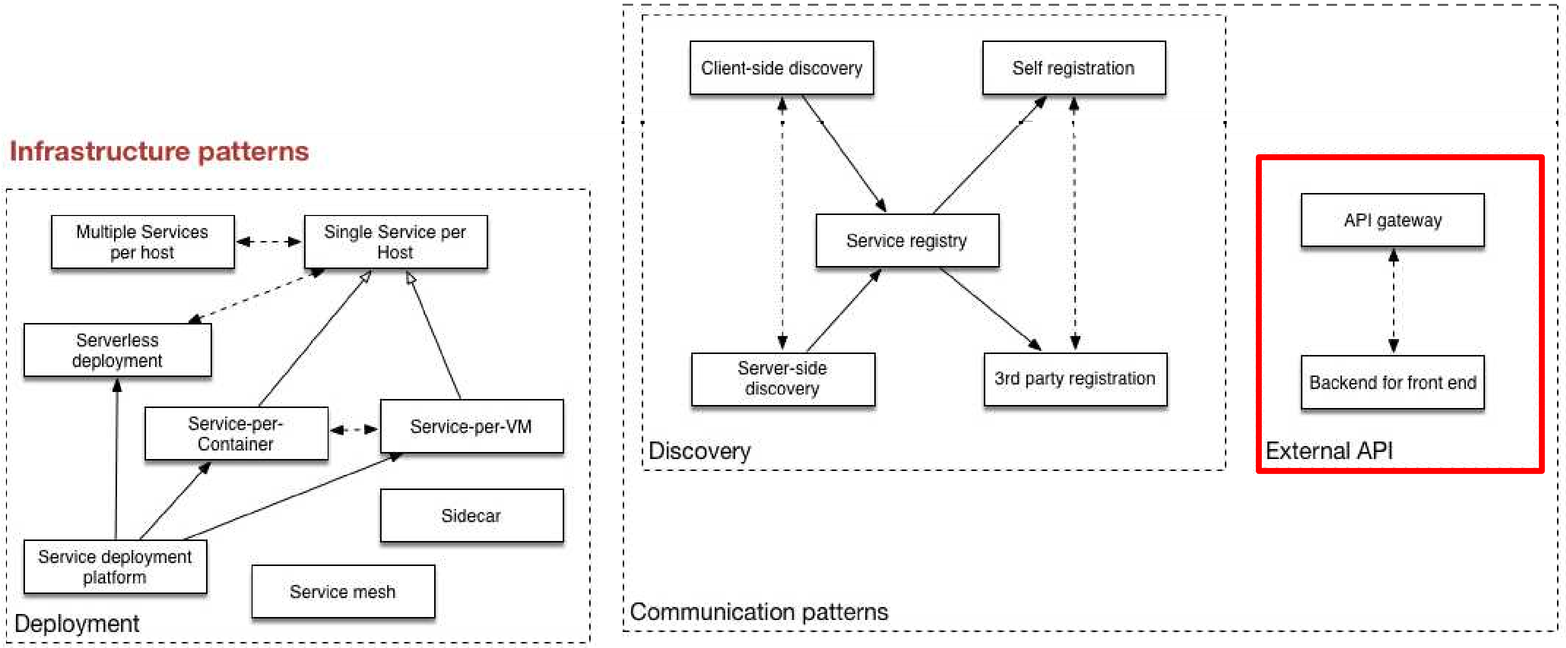
- The service code is less complex than when using the Self Registration pattern since its not responsible for registering itself
- The registrar can perform health checks on a service instance and register/unregister the instance based the health check

Drawbacks

- Superficial knowledge of the state of the service instance e.g. RUNNING or NOT RUNNING
- Another critical component that must be installed, configured and maintained

Microservices Patterns





External API

Example of an online store selling books:

- Need to develop multiple versions of the product details user interface:
    - HTML5/JavaScript-based UI for desktop and mobile browsers
    - Native Android and iPhone clients
    - Expose product details via a REST API for use by 3rd party applications
- A product details UI can display a lot of information about a product.
    - Basic information about the book such as title, author, price, etc.
    - Your purchase history for the book
    - Availability
    - Buying options
    - Customer reviews
    - …

External API

The online store uses the Microservice architecture pattern → the product details data is spread over multiple services:

- Product Info Service
- Pricing Service
- Order service
- Inventory service
- Review service

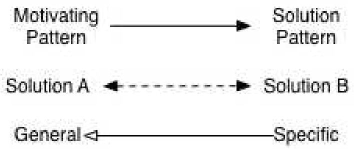
Consequently, the code that displays the product details needs to fetch information from all of these services.

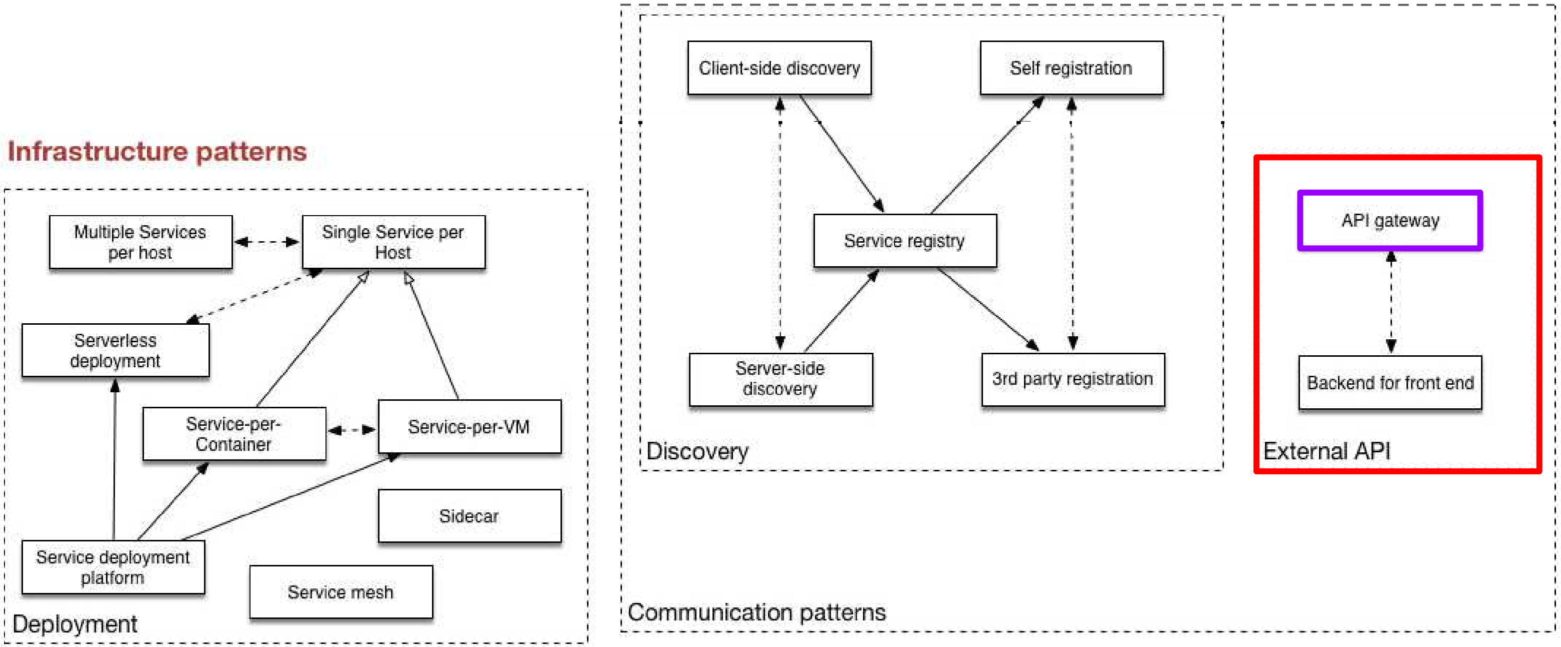
External API

Problem: How do the clients of a Microservices-based application access the individual services?

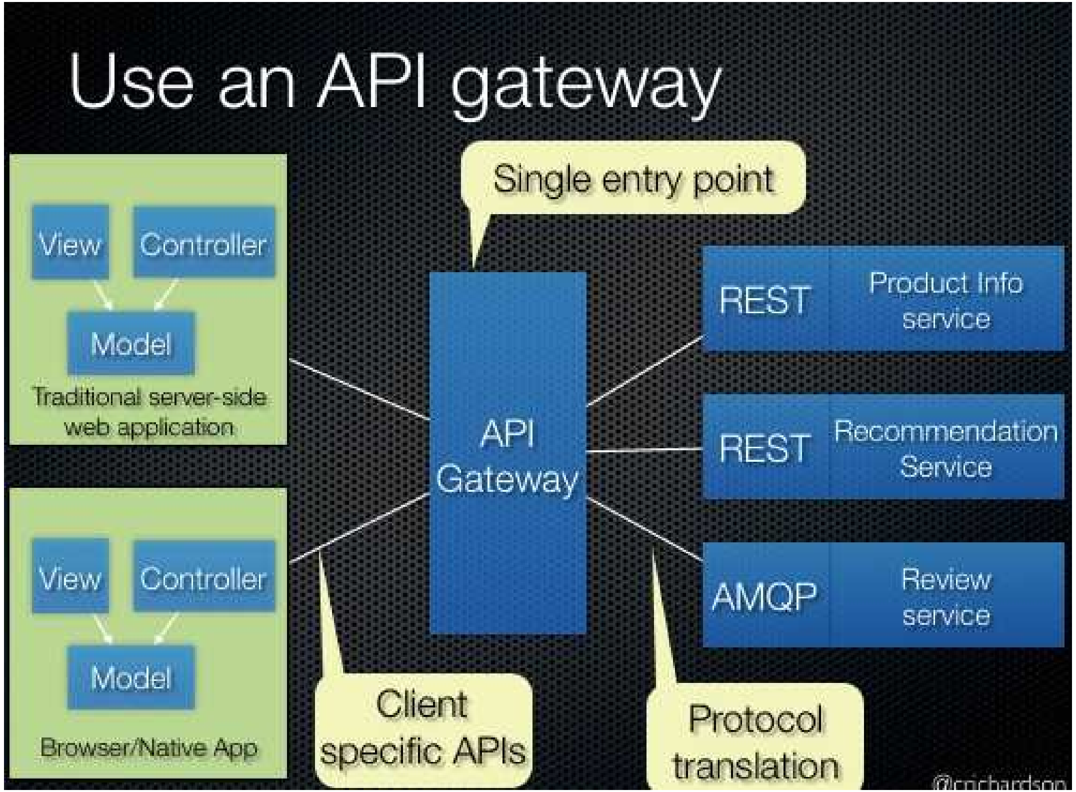
- The granularity of APIs provided by microservices is often different than what a client needs
- Different clients need different data
- Network performance is different for different types of clients
- The number of service instances and their locations (host+port) changes dynamically
- Partitioning into services can change over time and should be hidden from clients
- Services might use a diverse set of protocols, some of which might not be web friendly

Microservices Patterns





API Gateway


Use an API gateway
Single entry point
View
Controller
Model
Traditional server-side web application
View
Controller
Model
Browser/Native App
API Gateway
REST
Product Info service
REST
Recommendation Service
AMQP
Review service
Client specific APIs
Protocol translation
@crichardson

API Gateway

Advantages:

- Insulates the clients:
  - from how the application is partitioned into microservices
  - from the problem of determining the locations of service instances
- Provides the optimal API for each client
- Reduces the number of requests/roundtrips
- Simplifies the client by moving logic for calling multiple services from the client to API gateway
- Translates from a “standard” public web-friendly API protocol to whatever protocols are used internally

API Gateway

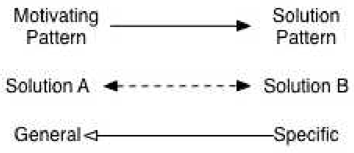
Drawbacks:

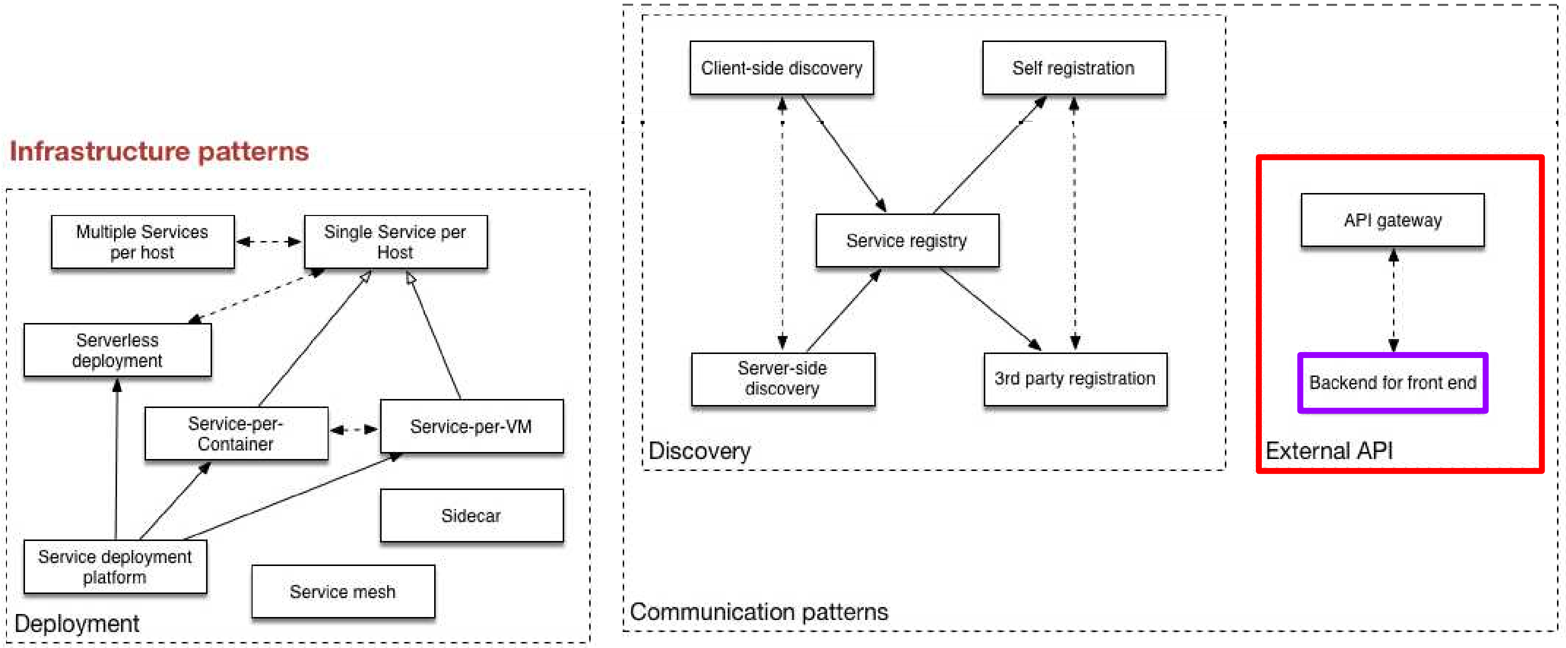
- Increased complexity
- Increased response time

Issues:

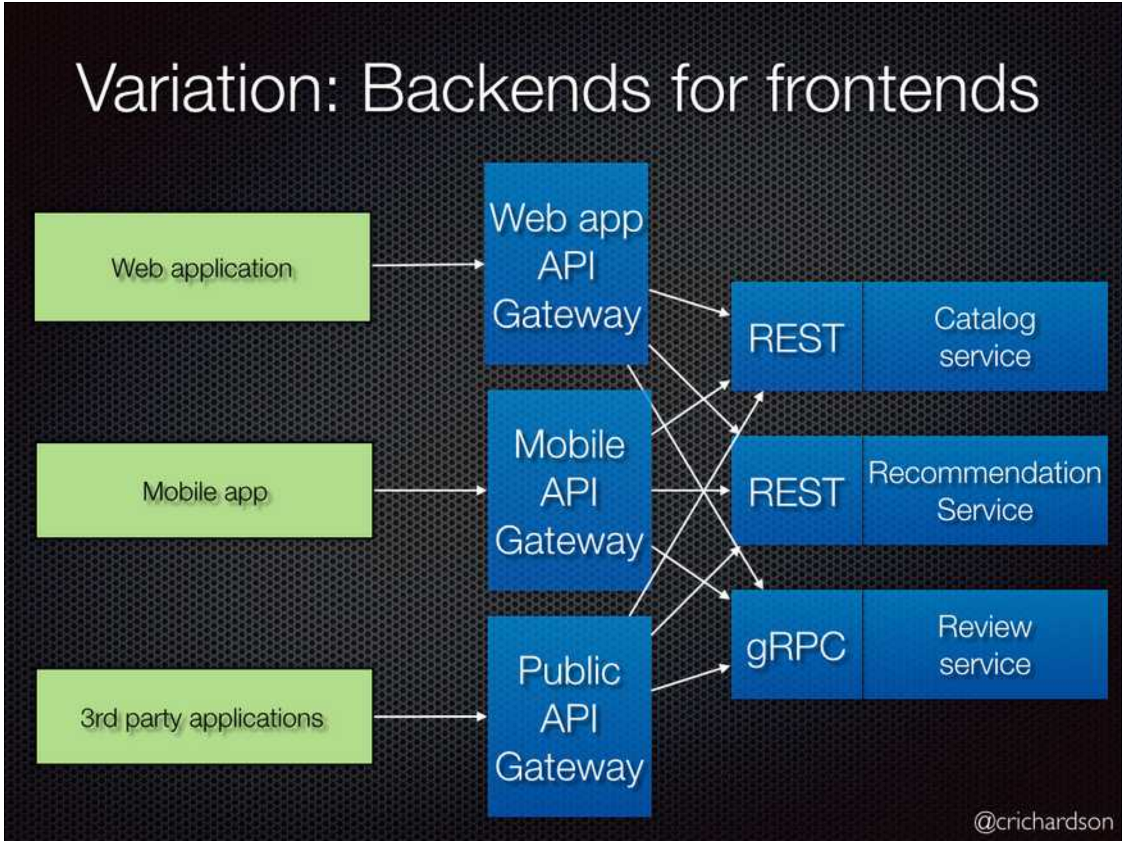
- How implement the API gateway?
  - An event-driven/reactive approach

Microservices Patterns





Backend for front-end



Variation: Backends for frontends

Backend for front-end

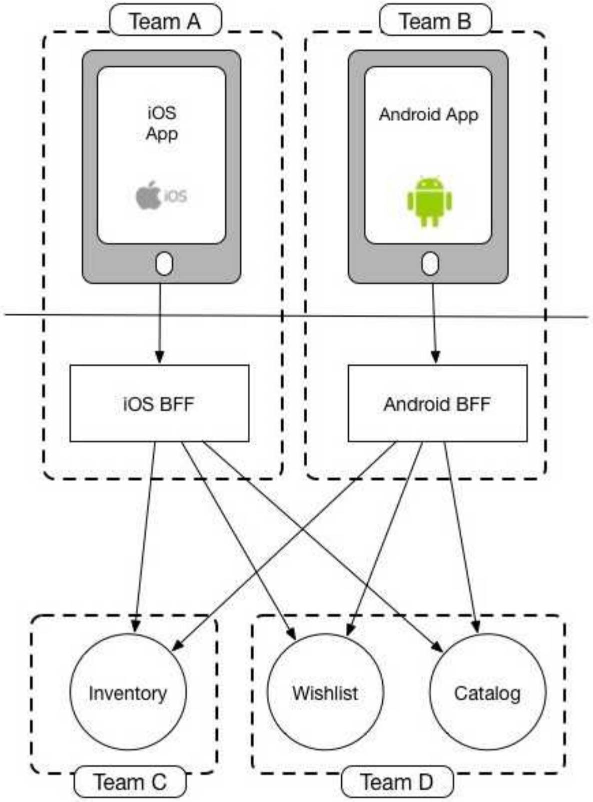


Fig from https://blog.bitsrc.io/bff-pattern-backend-for-frontend-an-introduction-e4fa965128bf

Backend for front-end

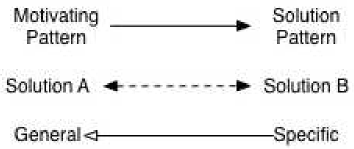
Advantages

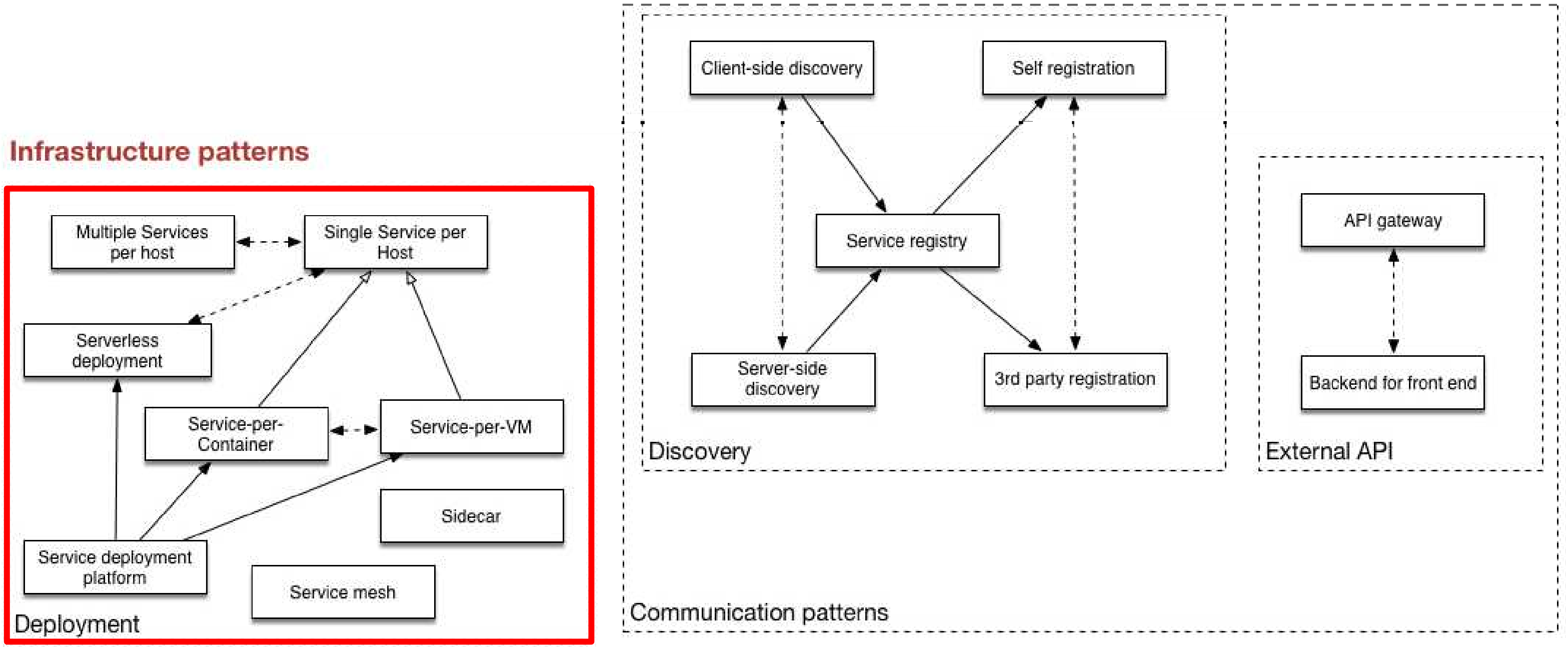
Separation of concerns — Frontend requirements will be separated from the backend concerns. This is easier for maintenance.

- **Easier to maintain and modify APIs** — The client application will know less about your APIs' structure, which will make it more resilient to changes in those APIs.
- **Better error handling in the frontend** — Server errors are meaningless to the frontend user most of the time. Instead of directly returning the error server sends, the BFF can map out errors that need to be shown to the user. This will improve the user experience.
- **Multiple device types can call the backend in parallel** — While the browser is making a request to the browser BFF, the mobile devices can do the same. It will help obtain responses from the services faster.
- **Better security** — Certain sensitive information can be hidden, and unnecessary data to the frontend can be omitted when sending back a response to the frontend. The abstraction will make it harder for attackers to target the application.
- **Shared team ownership of components** — Different parts of the application can be handled by different teams very easily. Frontend teams get to enjoy ownership of both their client application and its underlying resource consumption layer; leading to high development velocities. The below diagram shows an example of such a team separation along with BFFs.

Pitfalls

Microservices Patterns



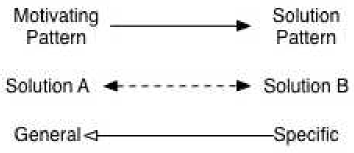


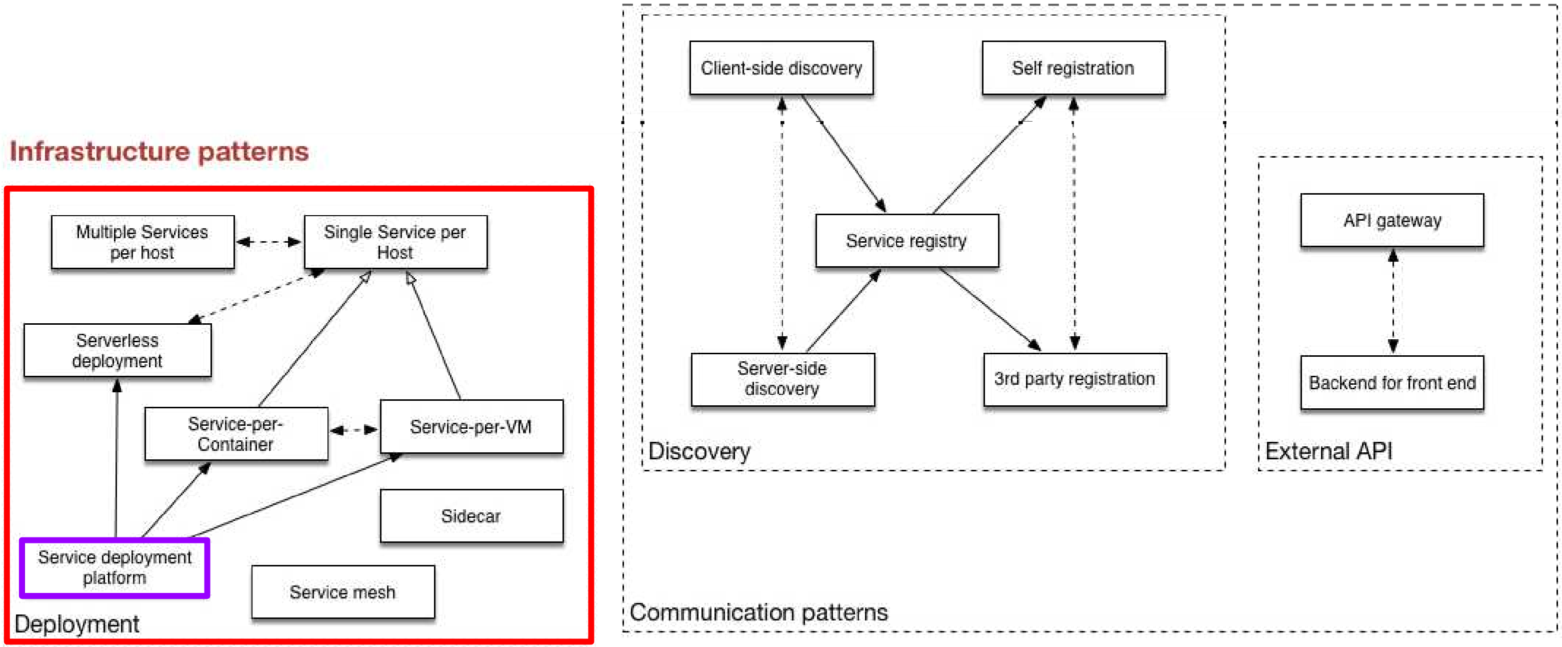
Service deployment

Problem: How are services packaged and deployed?

- Variety of languages, frameworks, and framework versions
- Multiple service instances for throughput and availability
- Services must be independently deployable and scalable
- Service instances need to be isolated from one another
- Building and deploying a service should be fast
- Developers should be able to constrain the resources (CPU and memory) consumed by a service
- Developers need to monitor the behavior of each service instance
- Deployment needs to be reliable and efficient

Microservices Patterns





Service deployment platform

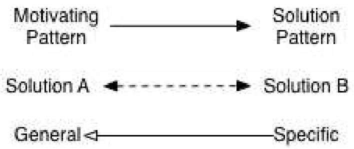
Solution: Use a deployment platform

- Automated infrastructure for application deployment.
- Provides a service abstraction (set of highly available (e.g. load balanced) service instances)

Examples:

- IaaS (Amazon EC2, Google Cloud, Azure, Digital Ocean, private Openstack IaaS …)
- Container orchestrators (Kubernetes, KIND, Docker swarm, Rancher …)
- Serverless platforms (AWS Lambda, Azure Functions, Google Cloud Functions, OpenWhisk …)
- PaaS (Heroku, Cloud Foundry, AWS Elastic Beanstalk, …)

Microservices Patterns



Motivating Pattern

Solution Pattern

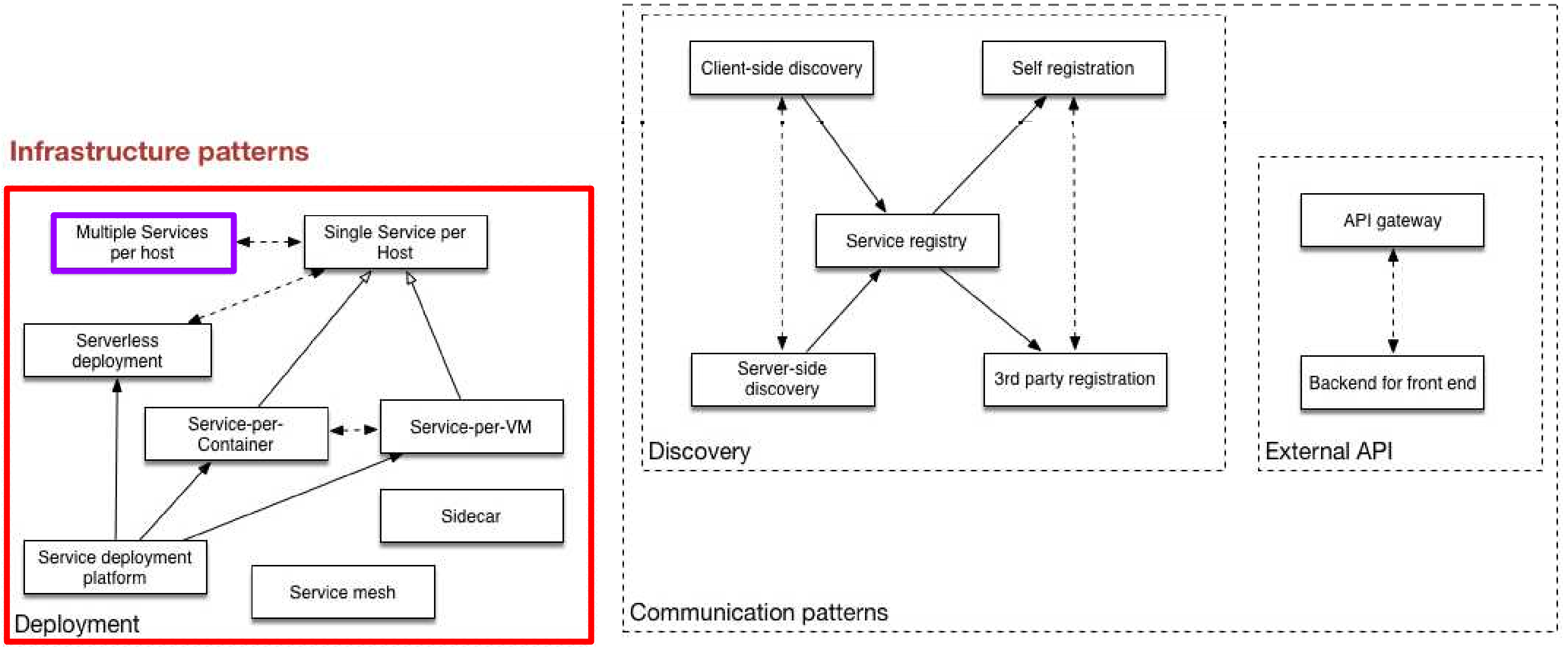
Solution A

Solution B

General

Specific

Infrastructure patterns



Multiple Services per host

Single Service per Host

Serverless deployment

Service-per-Container

Service-per-VM

Sidecar

Service deployment platform

Service mesh

Deployment

Client-side discovery

Self registration

Service registry

Server-side discovery

3rd party registration

Discovery

API gateway

Backend for front end

External API

Communication patterns

Multiple service instances per host

Solution: Run multiple instances of different services on a host (Physical or Virtual machine).

Ways for deploying a service instance on a shared host

- Deploy each service instance as a JVM process
    - Tomcat or Jetty instances per service instance.
- Deploy multiple service instances in the same JVM.
    - Web applications or OSGI bundles.

Multiple service instances per host

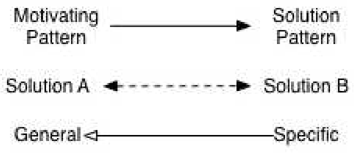
Advantages:

- More efficient resource utilization than the Service Instance per host pattern

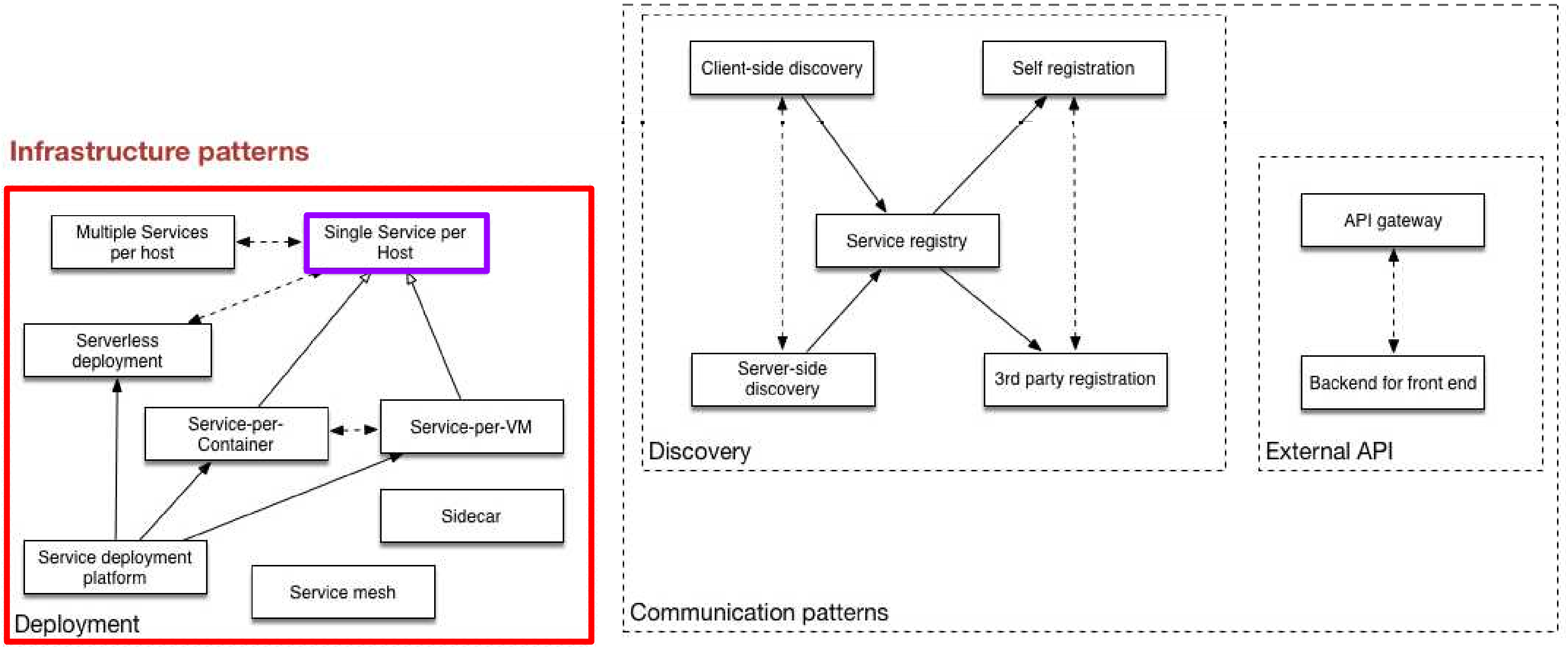
Drawbacks:

- Risk of conflicting resource requirements
- Risk of conflicting dependency versions
- Difficult to limit the resources consumed by a service instance
- When multiple services are deployed in the same process
    - Difficult to monitor resource consumption of individual services
    - Difficult to isolate services

Microservices Patterns



Infrastructure patterns



Single service instance per host

Solution: Deploy each single service instance on its own host

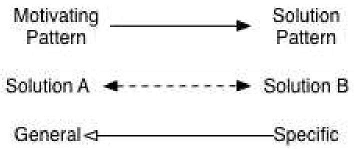
Advantages:

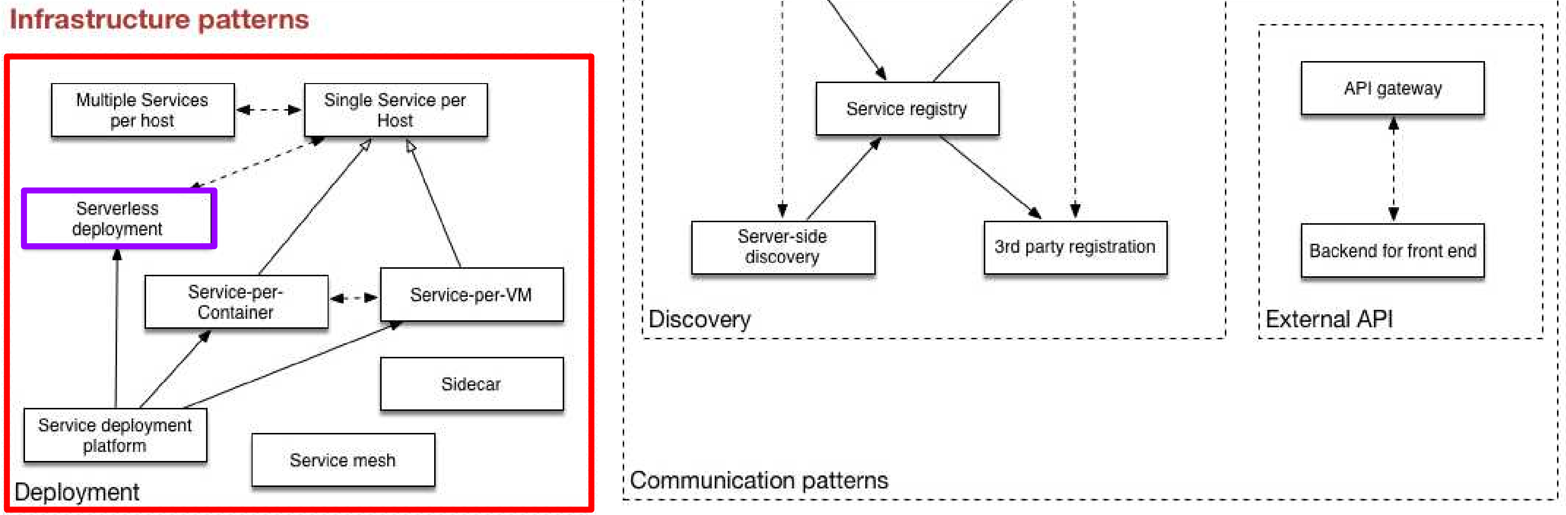
- Services instances are isolated from one another
- No conflicting resource requirements or dependency versions
- A service instance can only consume at most the resources of a single host
- Straightforward to monitor, manage, and redeploy each service instance

Drawbacks:

- Less efficient resource utilization compared to Multiple Services per Host (because there are more hosts)

Microservices Patterns





Serverless deployment

Solution:

- Use a deployment infrastructure that hides any concept of servers
- The infrastructure takes your service’s code and runs it
- You are charged for each request based on the resources consumed

To deploy a service using this approach:

- Package the code (e.g. as a ZIP file)
- Upload it to the deployment infrastructure
- Describe the desired performance characteristics

Serverless deployment - Examples

Examples

- AWS Lambda, Google Cloud Functions, Azure Functions
- OpenWhisk

Serverless deployment

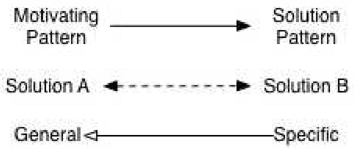
Advantages:

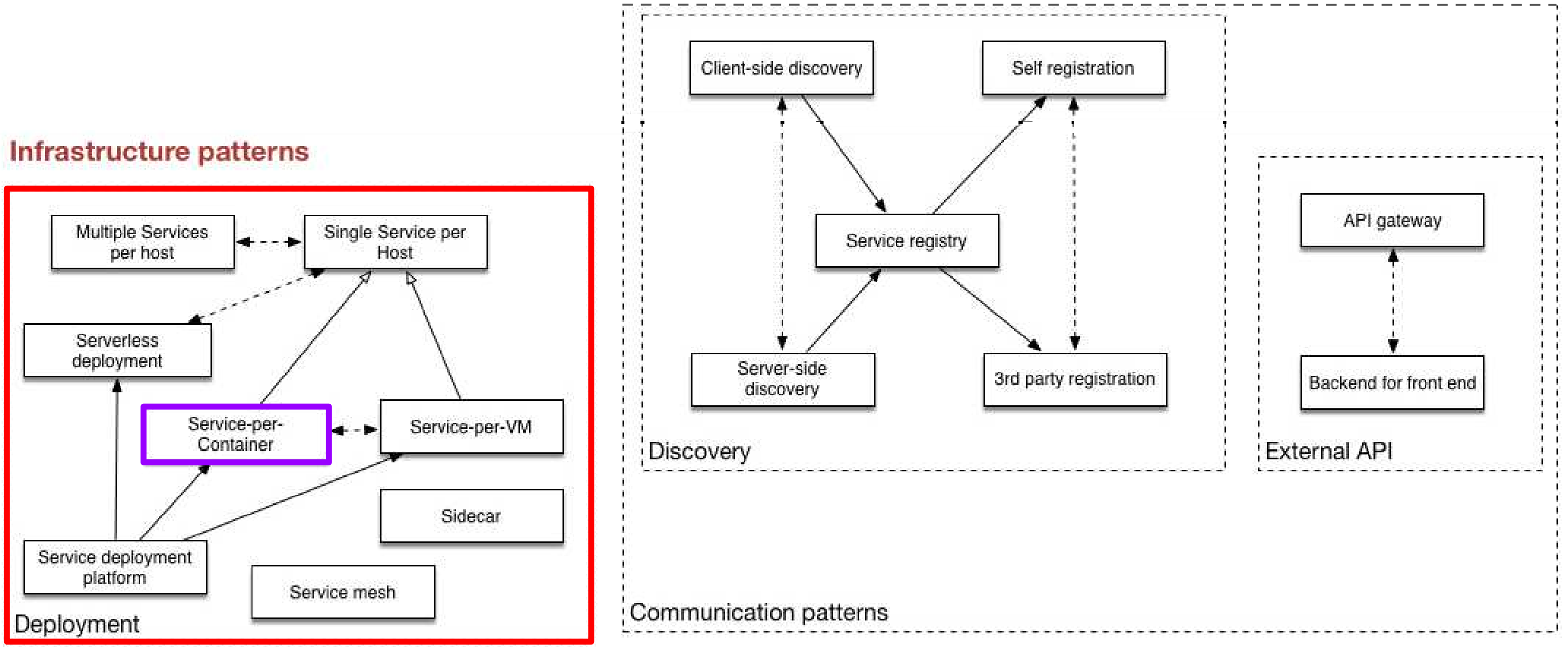
- Eliminates the need to spend time on managing low-level infrastructure.
- Focus on the functional code.
- Extremely elastic
- Pay for request

Drawbacks:

- Supports a few languages.
- Only suitable for stateless applications
- Cannot deploy a long running stateful application (database or broker).
- Limited “input sources”
- Functions must startup quickly

Microservices Patterns





Service instance per container

Solution: Package the service as a container image and deploy each service instance as a container

Examples:

- Kubernetes
- Marathon/Mesos
- Amazon EC2 Container Service

Note: The most popular container technology is Docker

Service instance per container

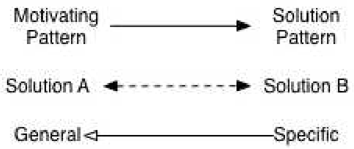
Advantages:

- Scale up and down a service by changing the number of container instances.
- Encapsulates the details of the technology used to build the service.
- Limits on the CPU and memory consumed by a service instance
- Extremely fast to build.
- Extremely fast to start.

Drawbacks:

- Security issues
- The infrastructure for deploying containers is not as rich as the infrastructure for deploying virtual machines.

Microservices Patterns


Motivating Pattern

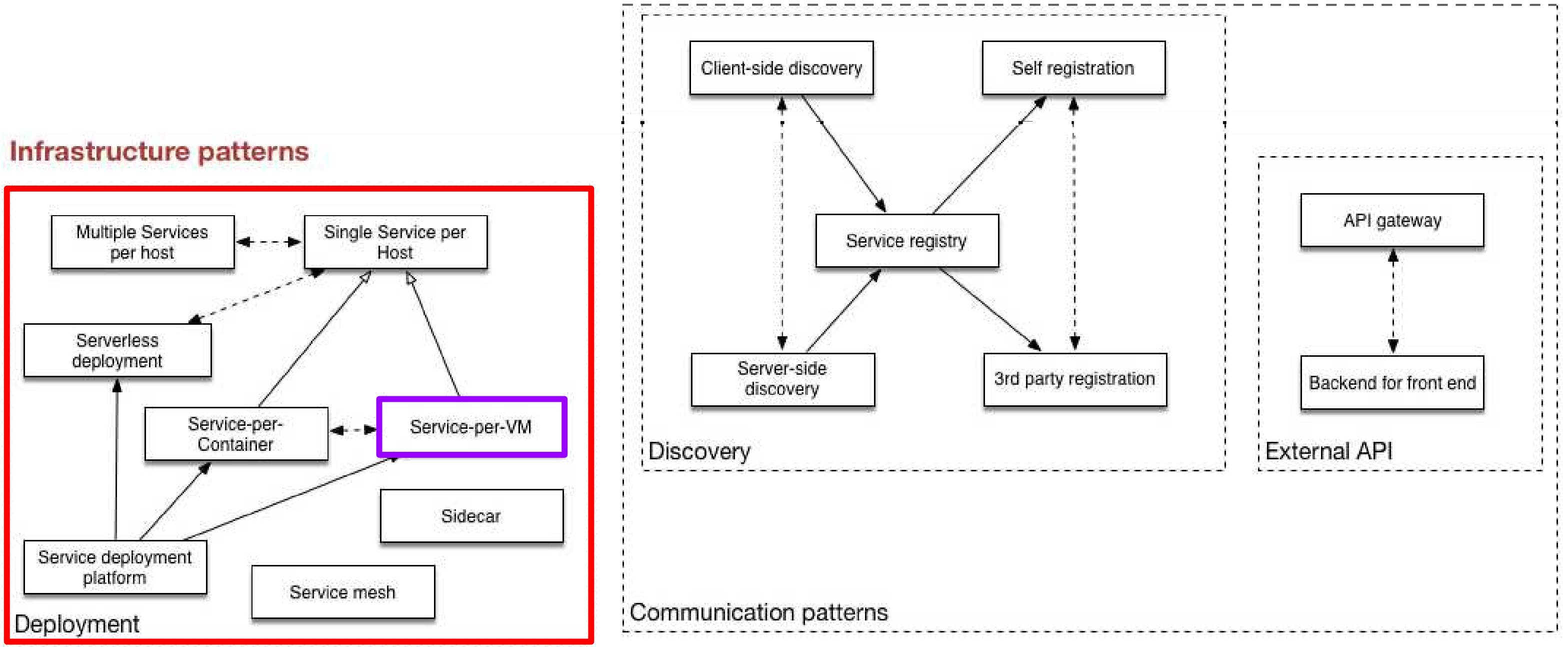
Solution Pattern

Solution A

Solution B

General

Specific


Infrastructure patterns

Multiple Services per host

Single Service per Host

Serverless deployment

Service-per-Container

Service-per-VM

Sidecar

Service deployment platform

Service mesh

Deployment

Client-side discovery

Self registration

Service registry

Server-side discovery

3rd party registration

Discovery

API gateway

Backend for front end

External API

Communication patterns

Service Instance per VM

Solution: Package the service as a virtual machine image and deploy each service instance as a separate VM

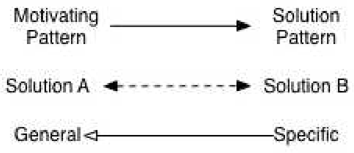
- Example: Netflix packages each service as an EC2 AMI and deploys each service instance as a EC2 instance.

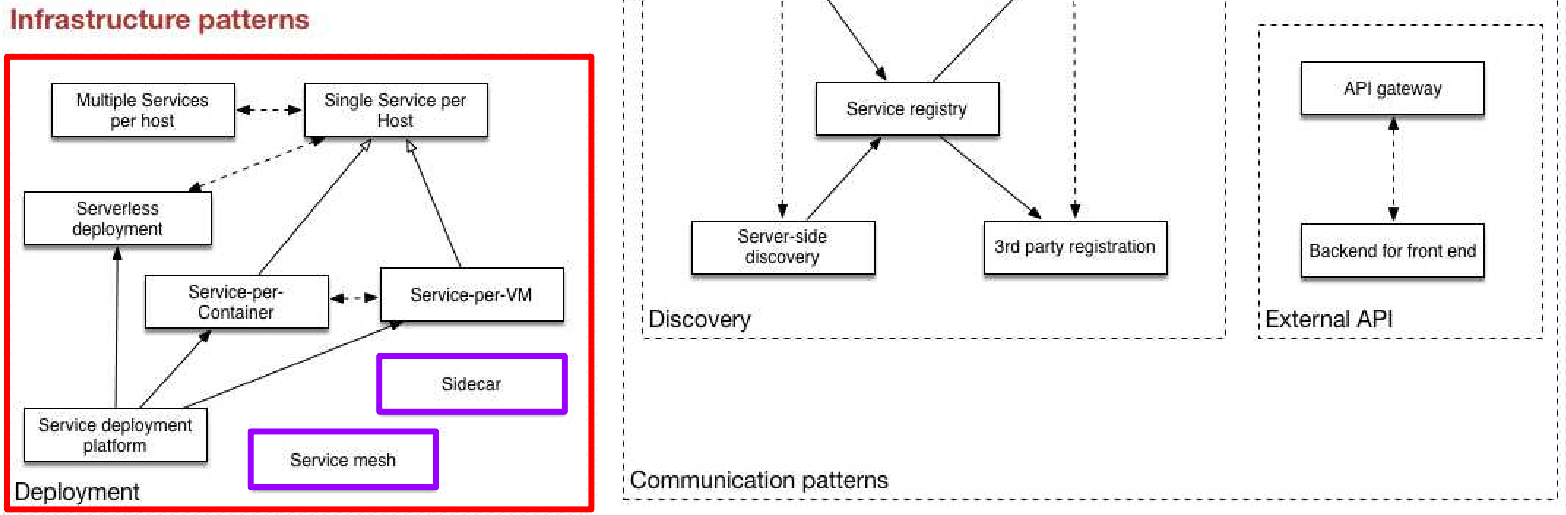
Service Instance per VM

Advantages

- Straightforward to scale the service by increasing the number of instances
- The VM encapsulates the details of the technology used to build the service
- Each service instance is isolated
- A VM imposes limits on the CPU and memory consumed by a service instance
- IaaS solutions such as AWS provide a mature and feature rich infrastructure for deploying and managing virtual machines
    - Elastic Load Balancer

Microservices Patterns





Hydrid deployment

Service-per-container or Service-per-VM

for normal traffic (par per hour)

Serverless microservice

when request peak (fast startup/elasticity and pay per request)

Drawbacks: 2 implementations of the same MS

Deployment

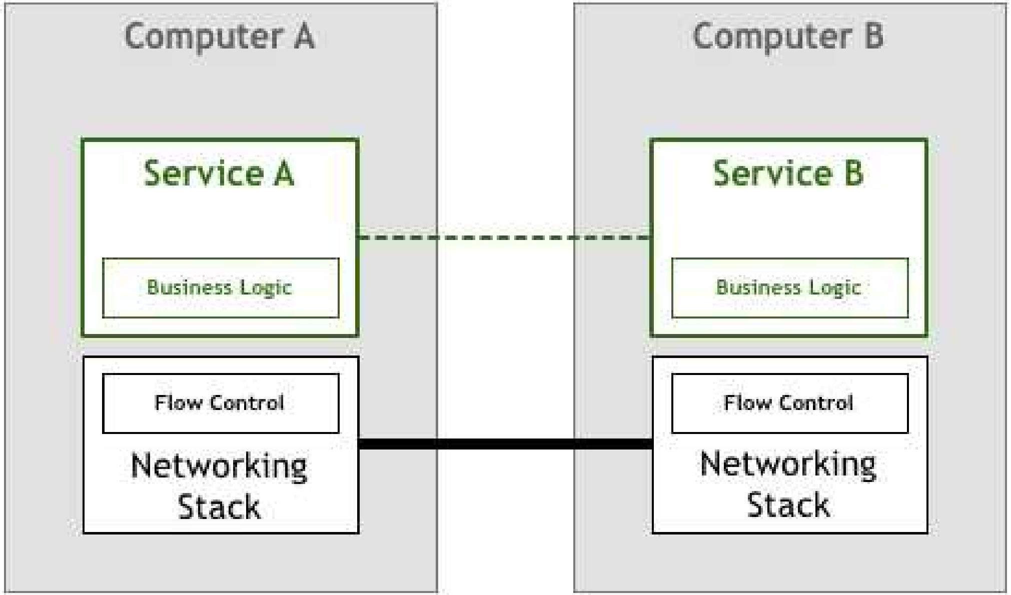
Service mesh

- dedicated infrastructure layer for handling service-to-service communication and global cross-cutting of concerns to make these communications more reliable, secure, observable and manageable.
- Examples: Istio, Linkerd, Maesh ...

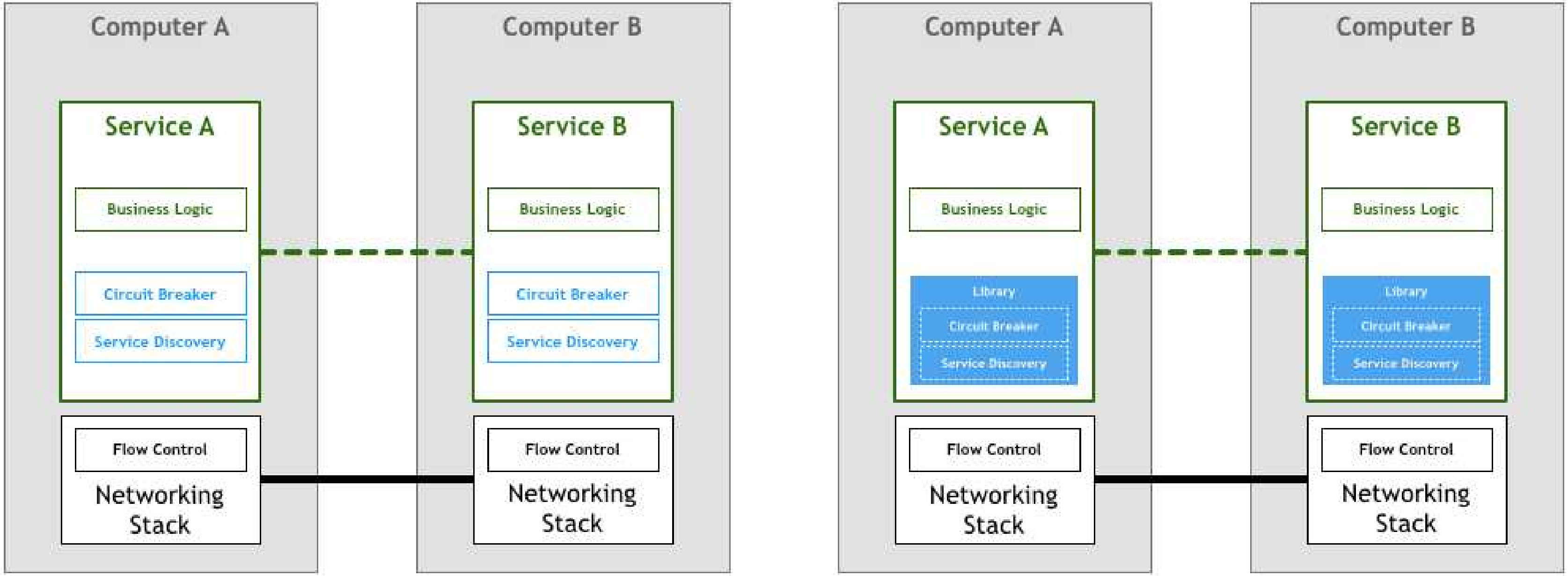
Sidecar

- Communication proxy between microservices in the mesh
  - routing according load, version, mode (prod, dev), A/B testing, …
- Examples: Envoy, Spring Boot Sidecar

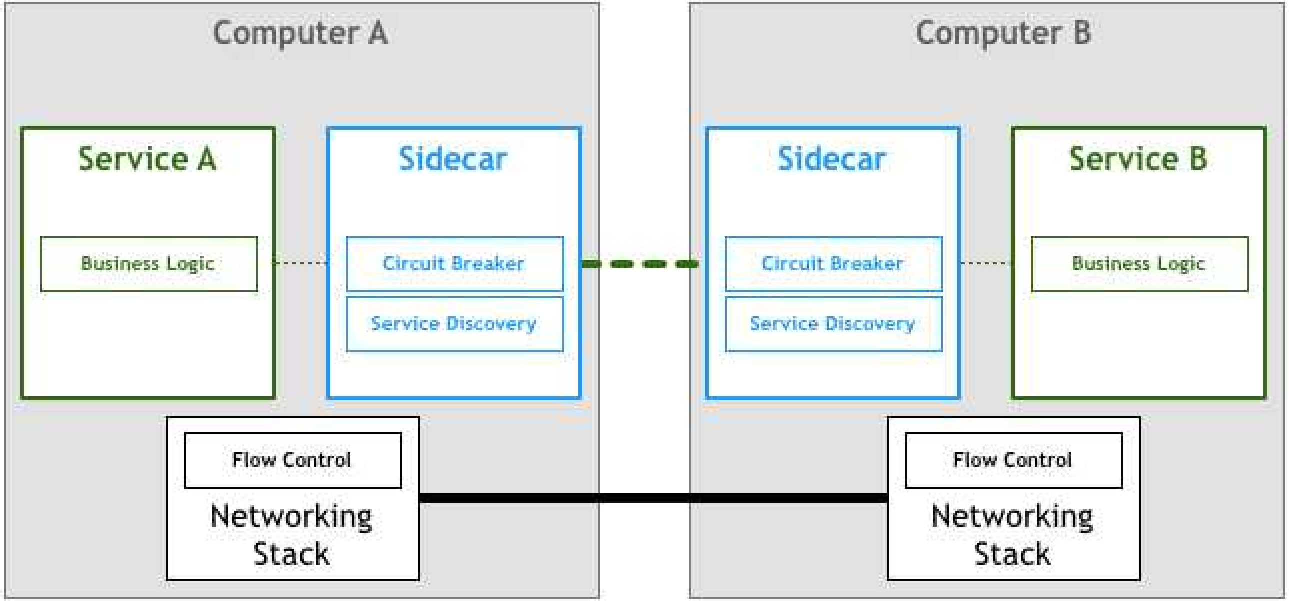
Service Mesh

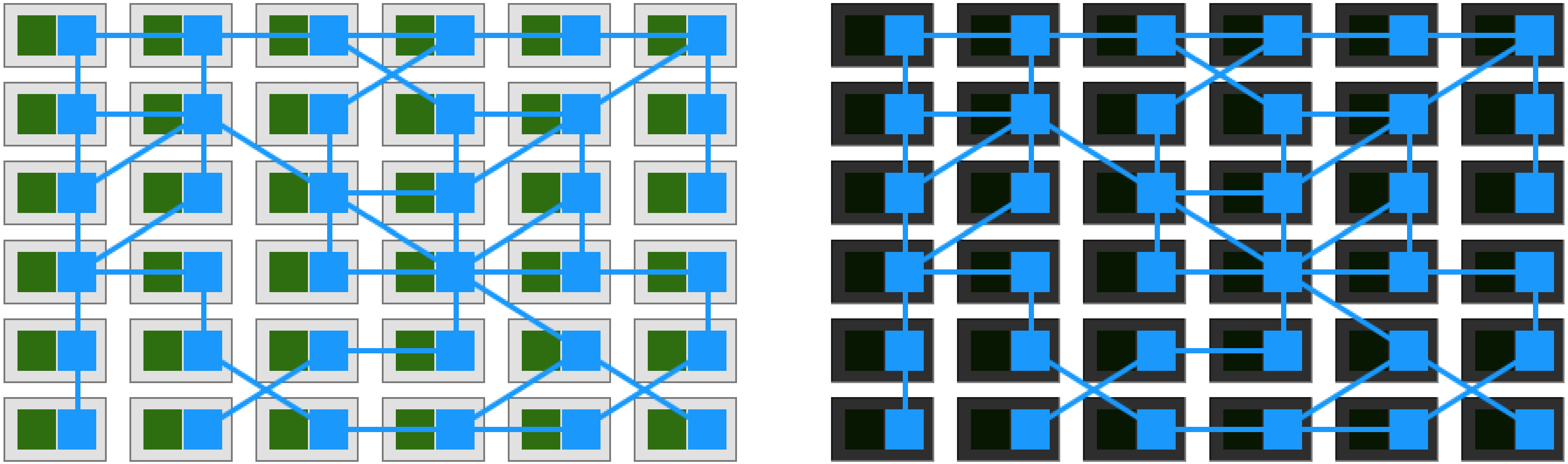


Service Mesh

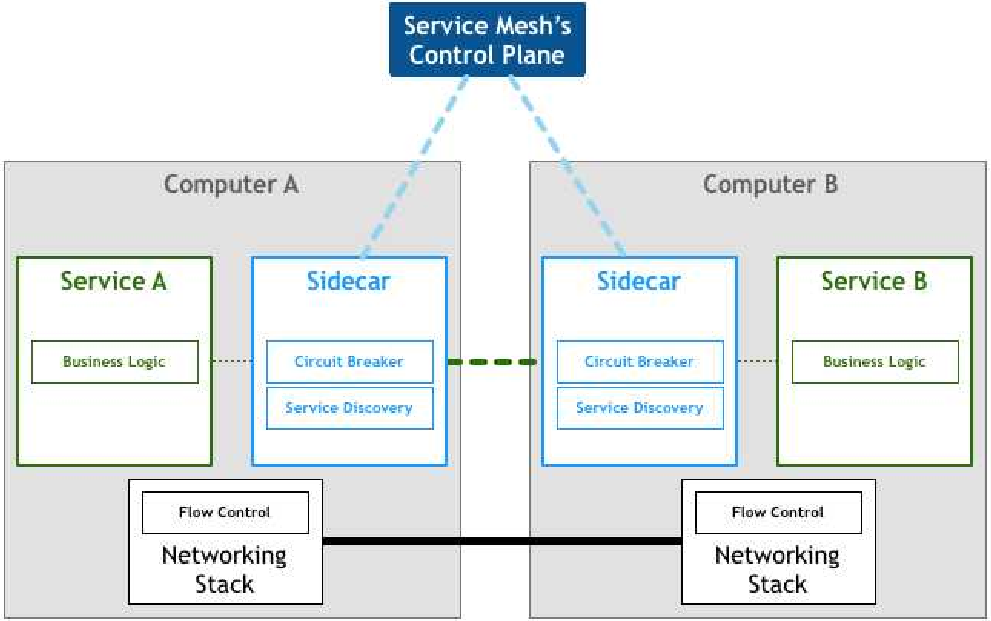


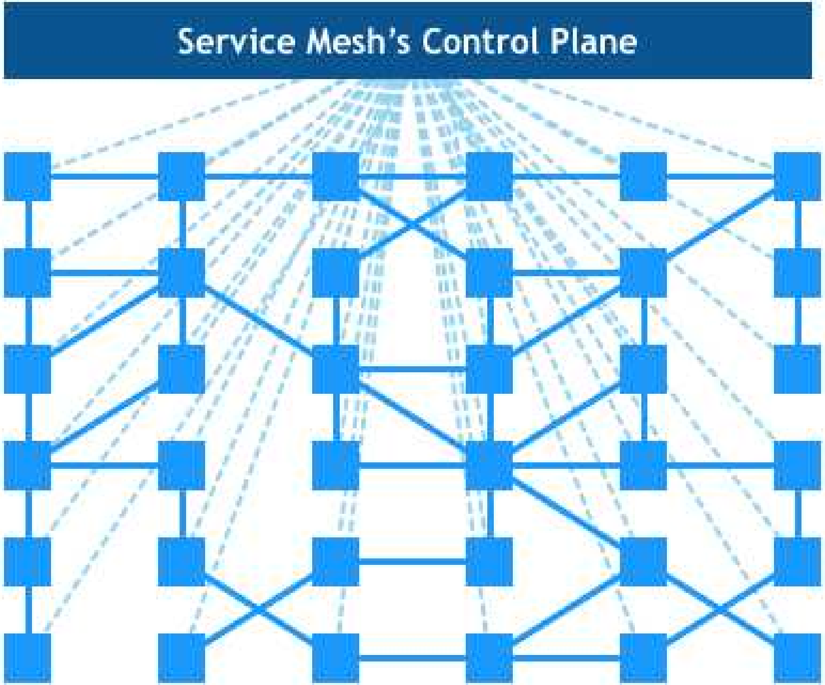
Service Mesh - Sidecar pattern



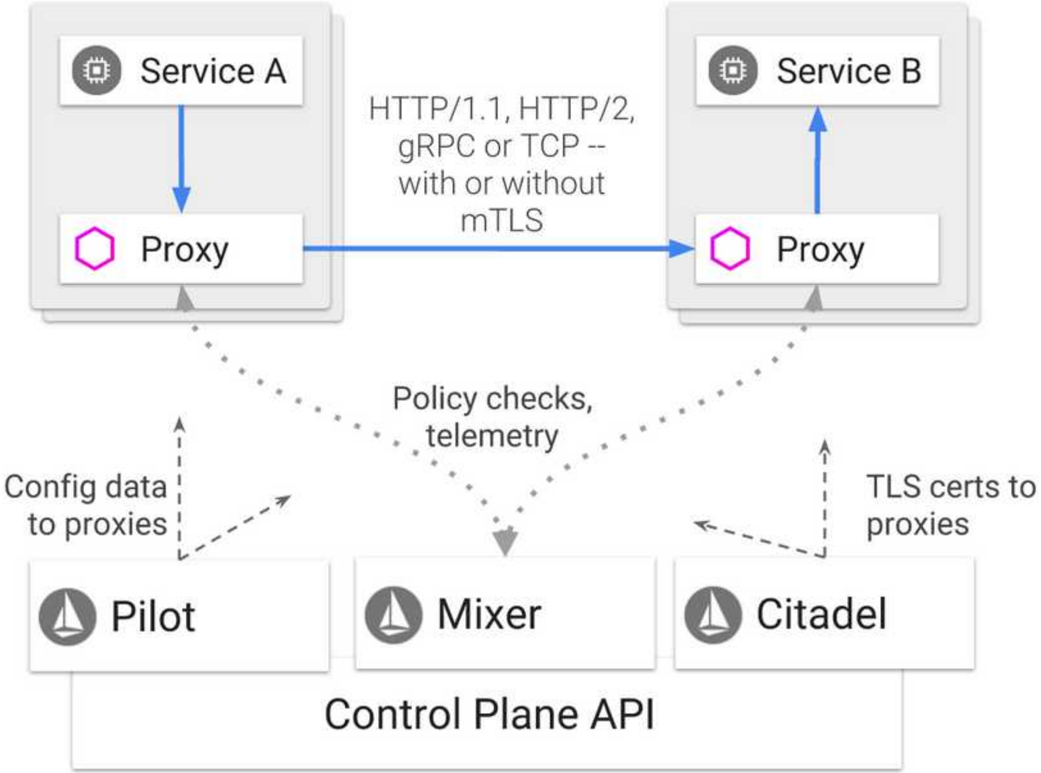


Service Mesh





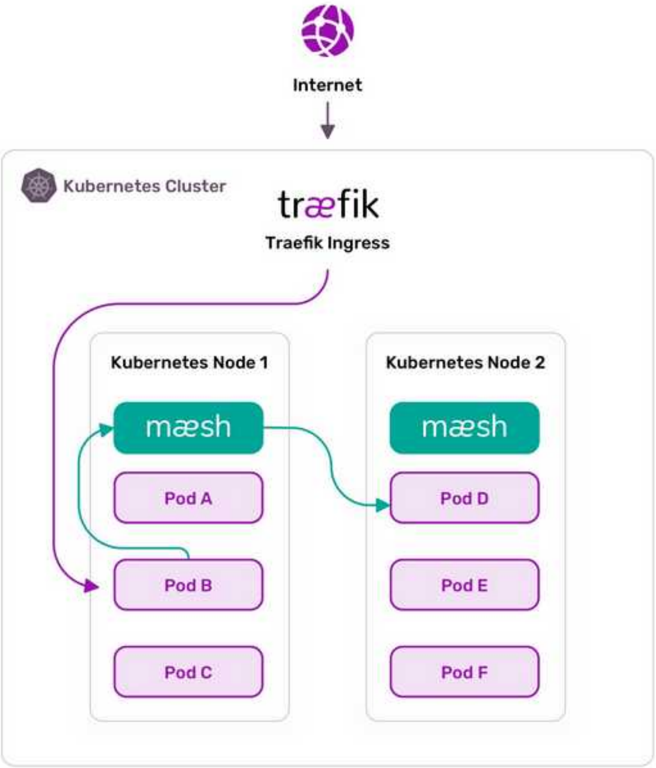
Service Mesh - Example Istio



Service Mesh - Example Maesh

Kubernetes

Traefik



Case studies

Netflix

Devoxx

Case Study : NETFLIX

Leader in subscription internet TV service

created in 1997

158 paid million members

~190 countries, 10s of languages

1000s of device types

Microservices hosted on AWS

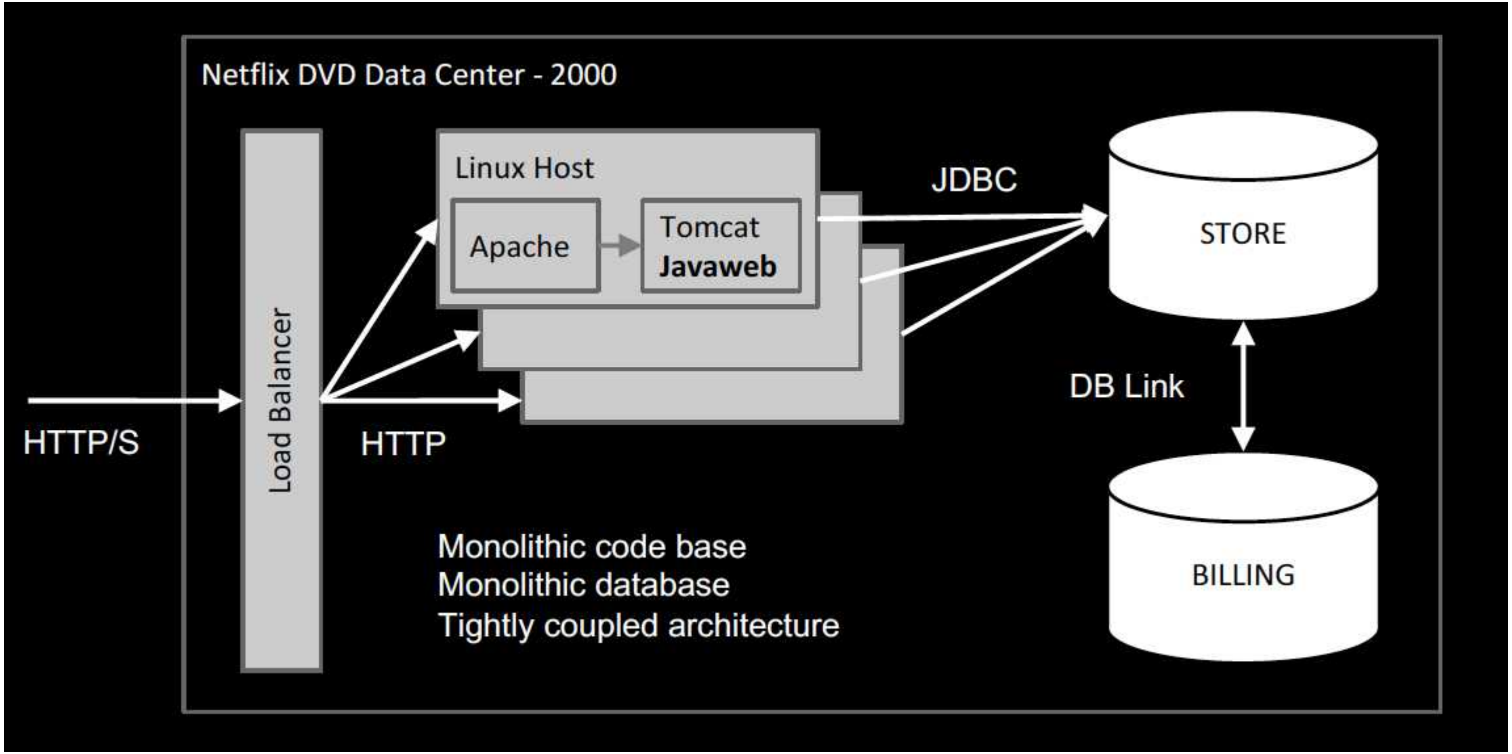
Open-source for Microservices platforms

Josh Evans – Engineering Leader

Mastering Chaos

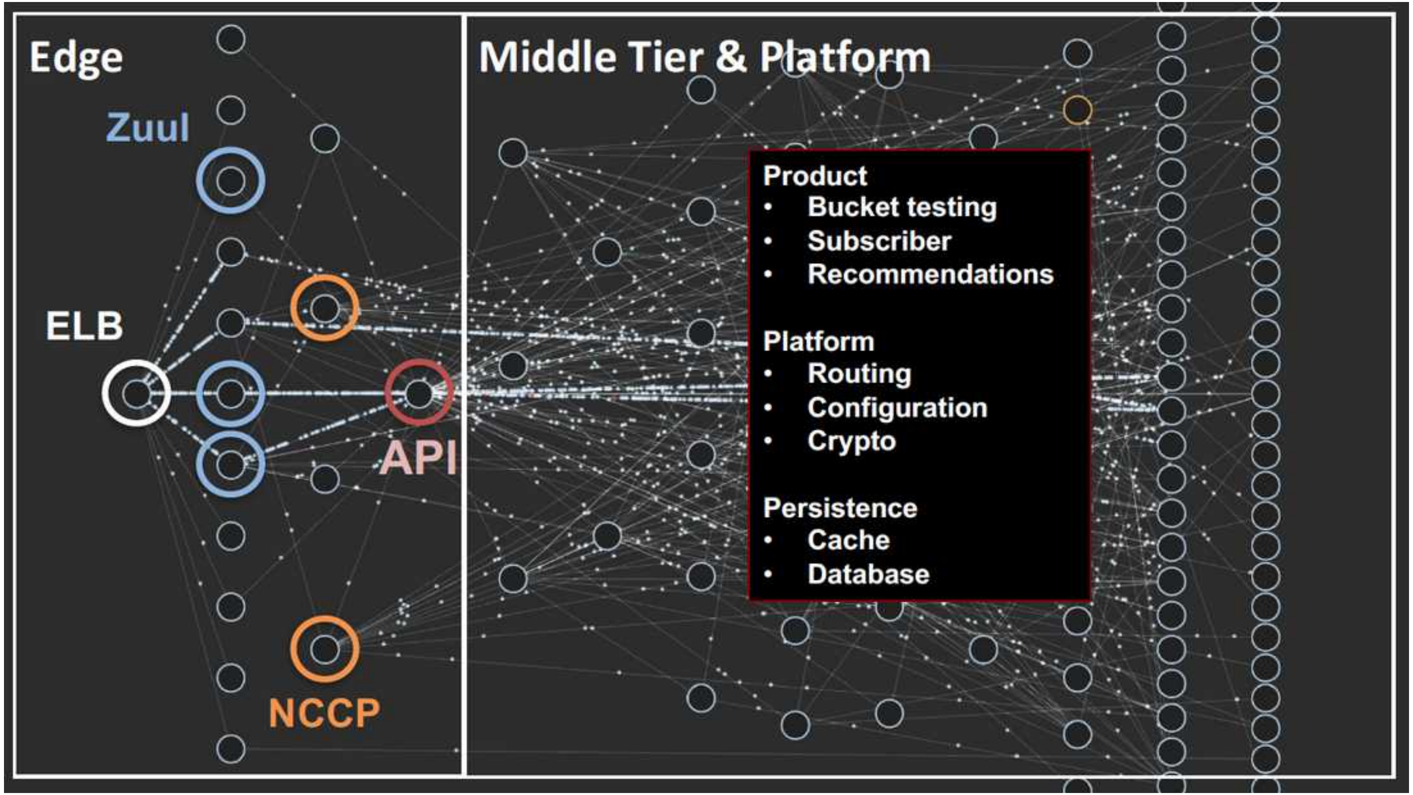
A Netflix Guide to Microservices

Case Study : NETFLIX



Monolithic code base
Monolithic database
Tightly coupled architecture

Case Study : NETFLIX

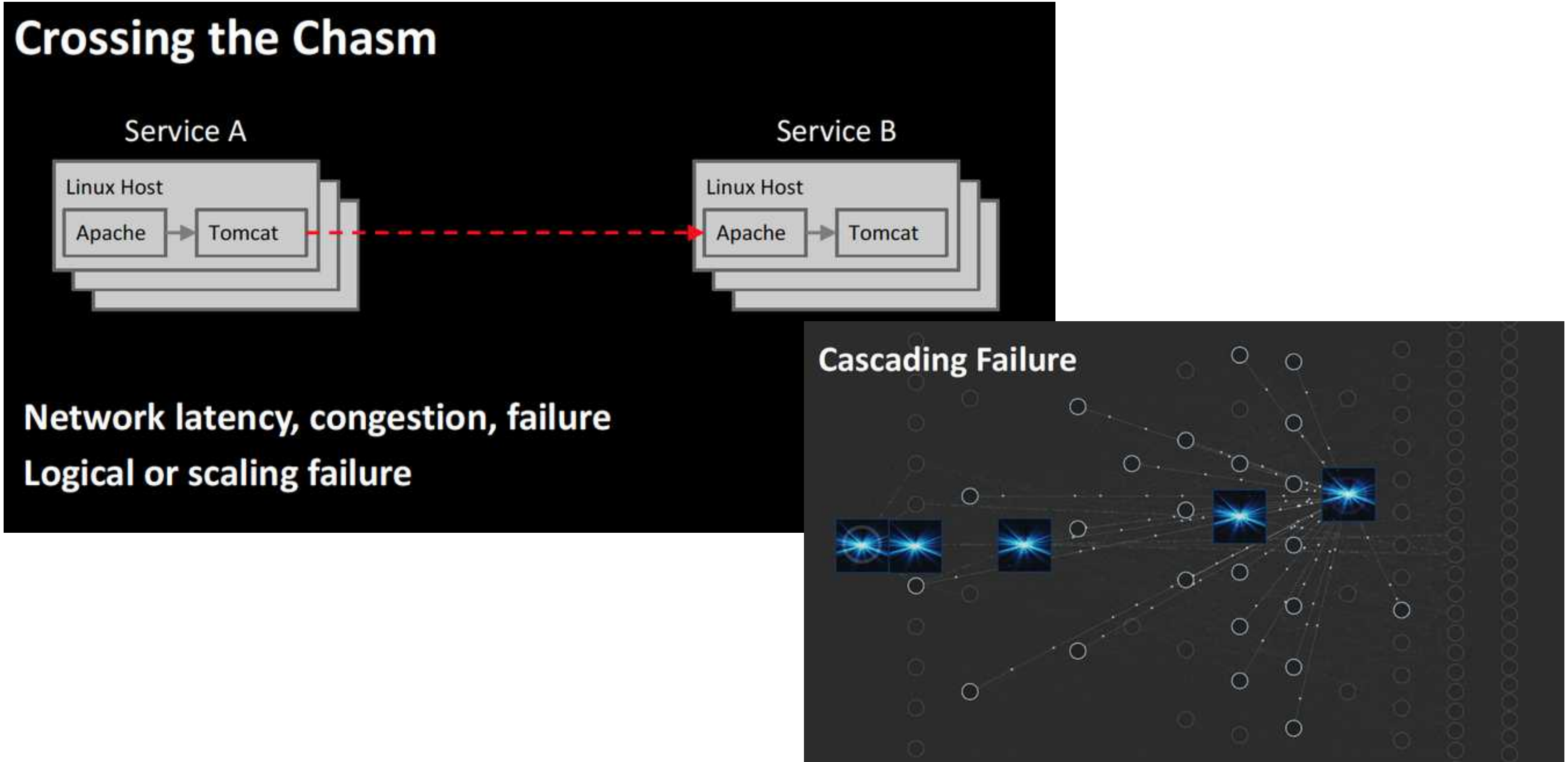


Product
- Bucket testing
- Subscriber
- Recommendations

Platform
- Routing
- Configuration
- Crypto

Persistence
- Cache
- Database

Case Study : NETFLIX



Crossing the Chasm

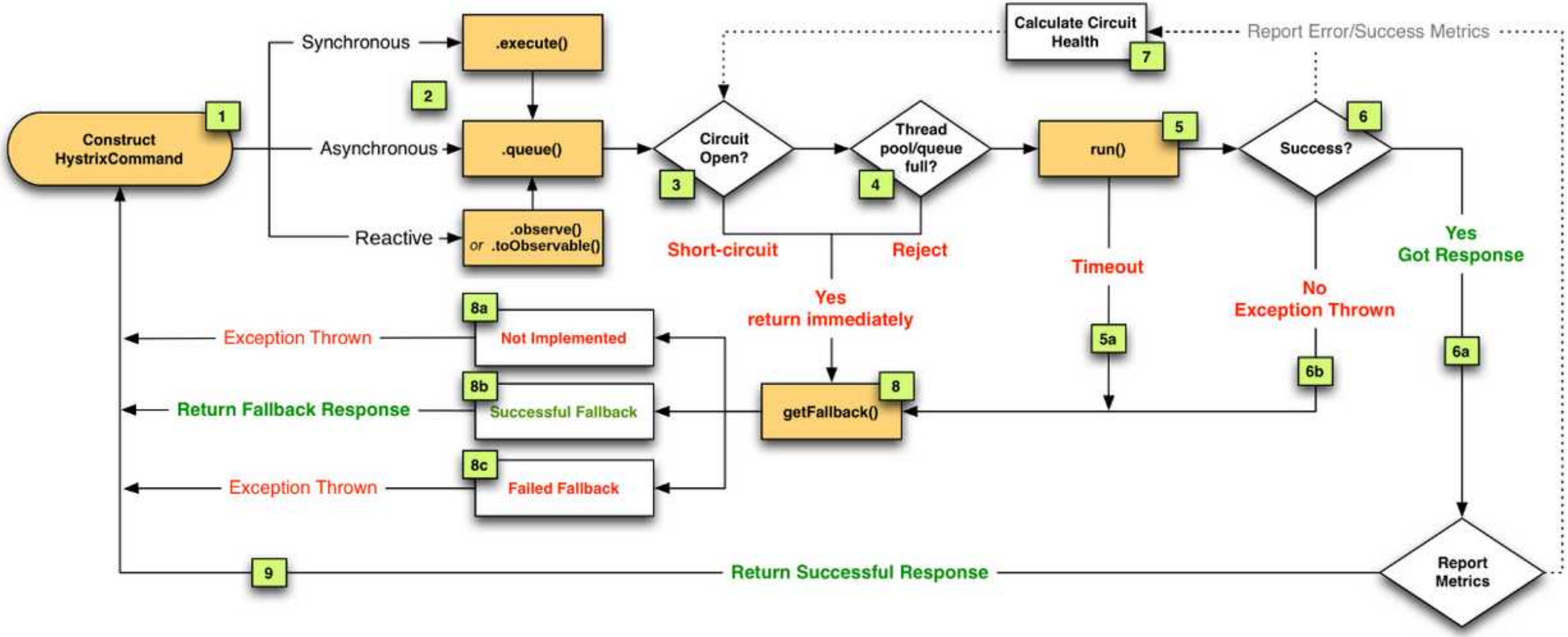
Network latency, congestion, failure

Logical or scaling failure

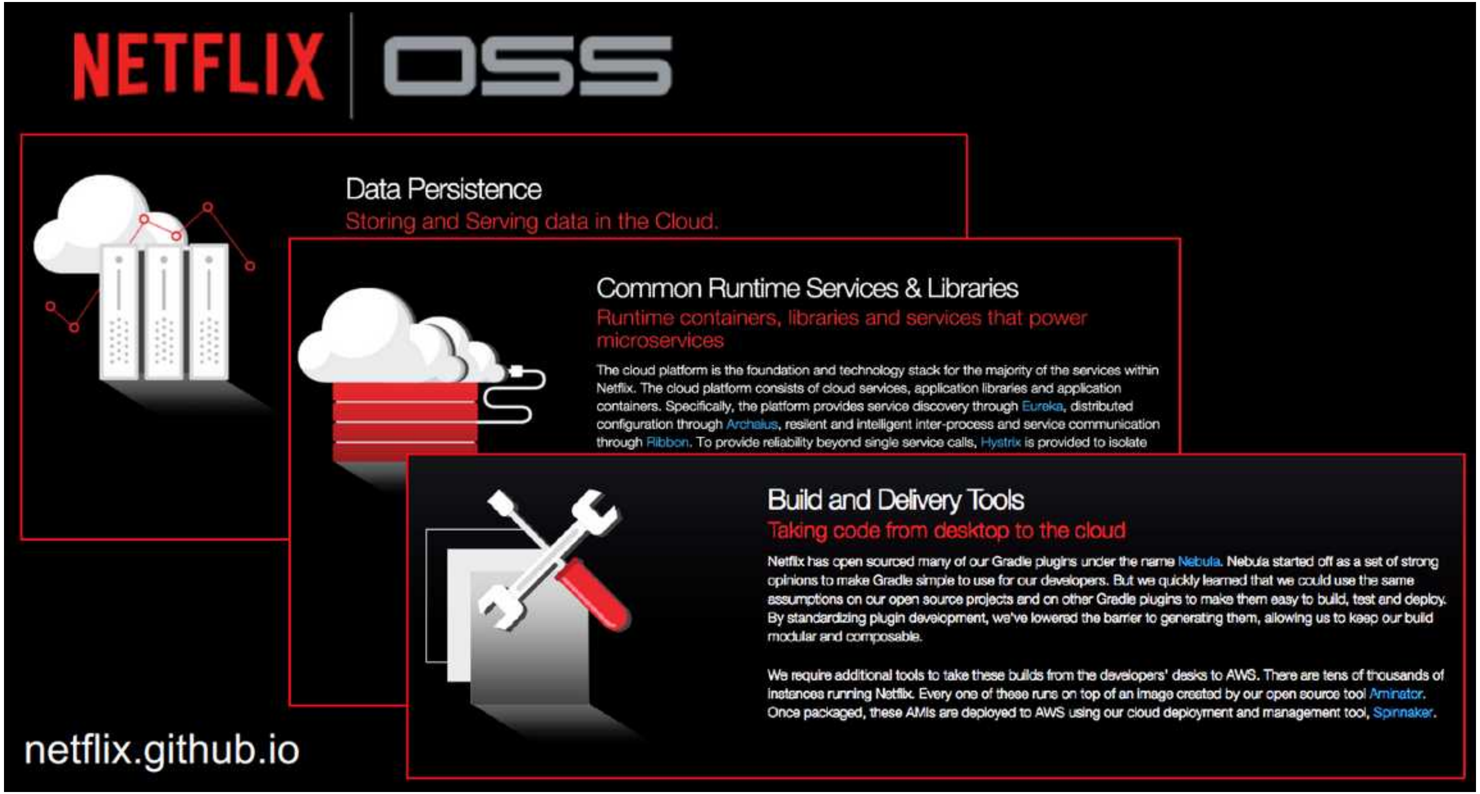
Cascading Failure

Case Study : NETFLIX

HYSTRIX
DEFEND YOUR APP



Case Study : NETFLIX



NETFLIX | OSS

Data Persistence

Storing and Serving data in the Cloud.

Common Runtime Services & Libraries

Runtime containers, libraries and services that power microservices

The cloud platform is the foundation and technology stack for the majority of the services within Netflix. The cloud platform consists of cloud services, application libraries and application containers. Specifically, the platform provides service discovery through Eureka, distributed configuration through Archaius, resilent and intelligent inter-process and service communication through Ribbon. To provide reliability beyond single service calls, Hystrix is provided to isolate

Build and Delivery Tools

Taking code from desktop to the cloud

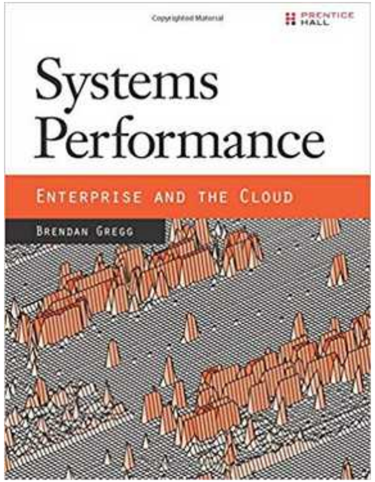
Netflix has open sourced many of our Gradle plugins under the name Nebula. Nebula started off as a set of strong opinions to make Gradle simple to use for our developers. But we quickly learned that we could use the same assumptions on our open source projects and on other Gradle plugins to make them easy to build, test and deploy. By standardizing plugin development, we've lowered the barrier to generating them, allowing us to keep our build modular and composable.

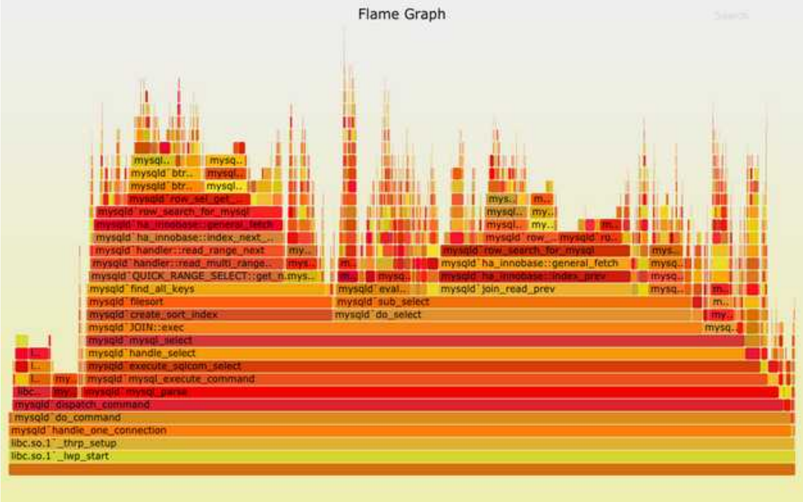
We require additional tools to take these builds from the developers' desks to AWS. There are tens of thousands of instances running Netflix. Every one of these runs on top of an image created by our open source tool Aminator. Once packaged, these AMIs are deployed to AWS using our cloud deployment and management tool, Spinnaker.

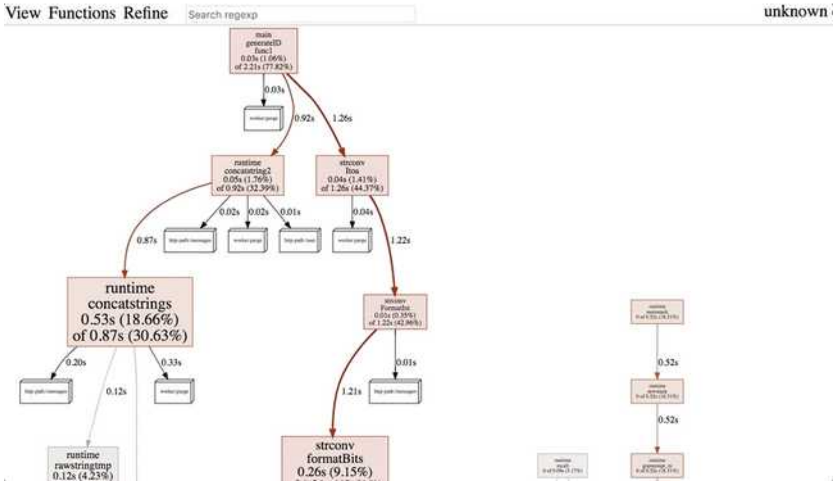
netflix.github.io

Case Study : NETFLIX

Performance Debugging

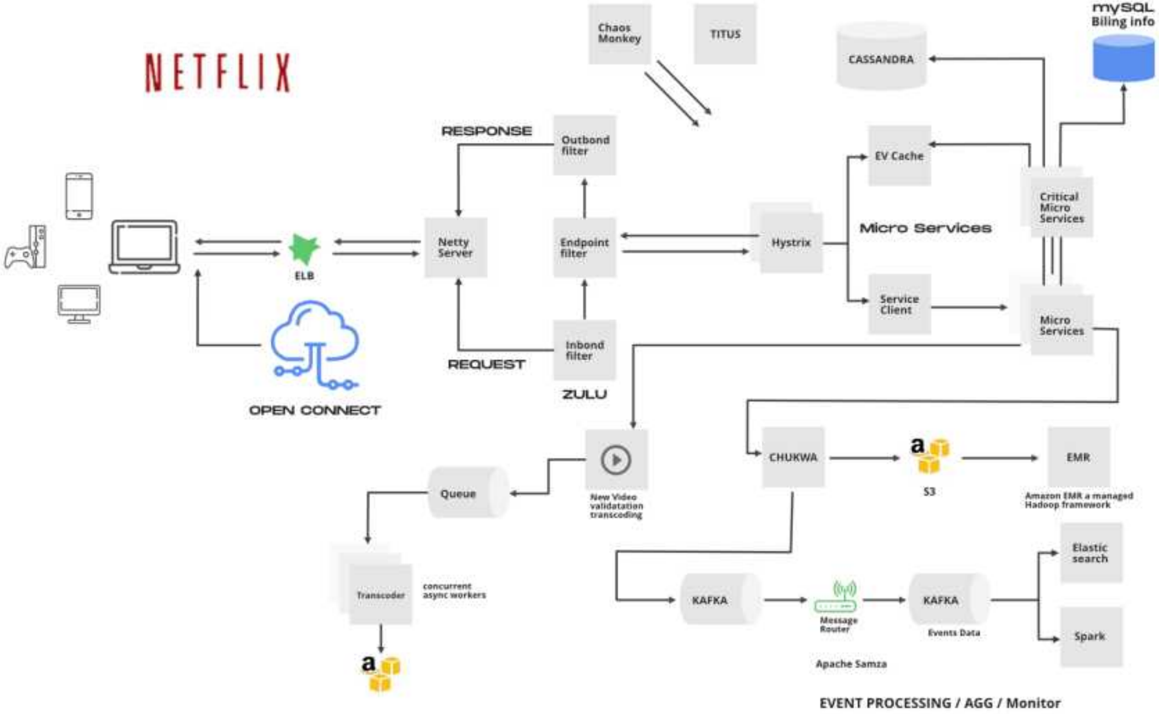






Extra : Netflix Backend

https://dev.to/gbengelebs/netflix-system-design-backend-architecture-10i3



Case Study : DEVOXX™

Annual Java, Android and HTML5 community conference

- started in 2001 by the Belgium JUG

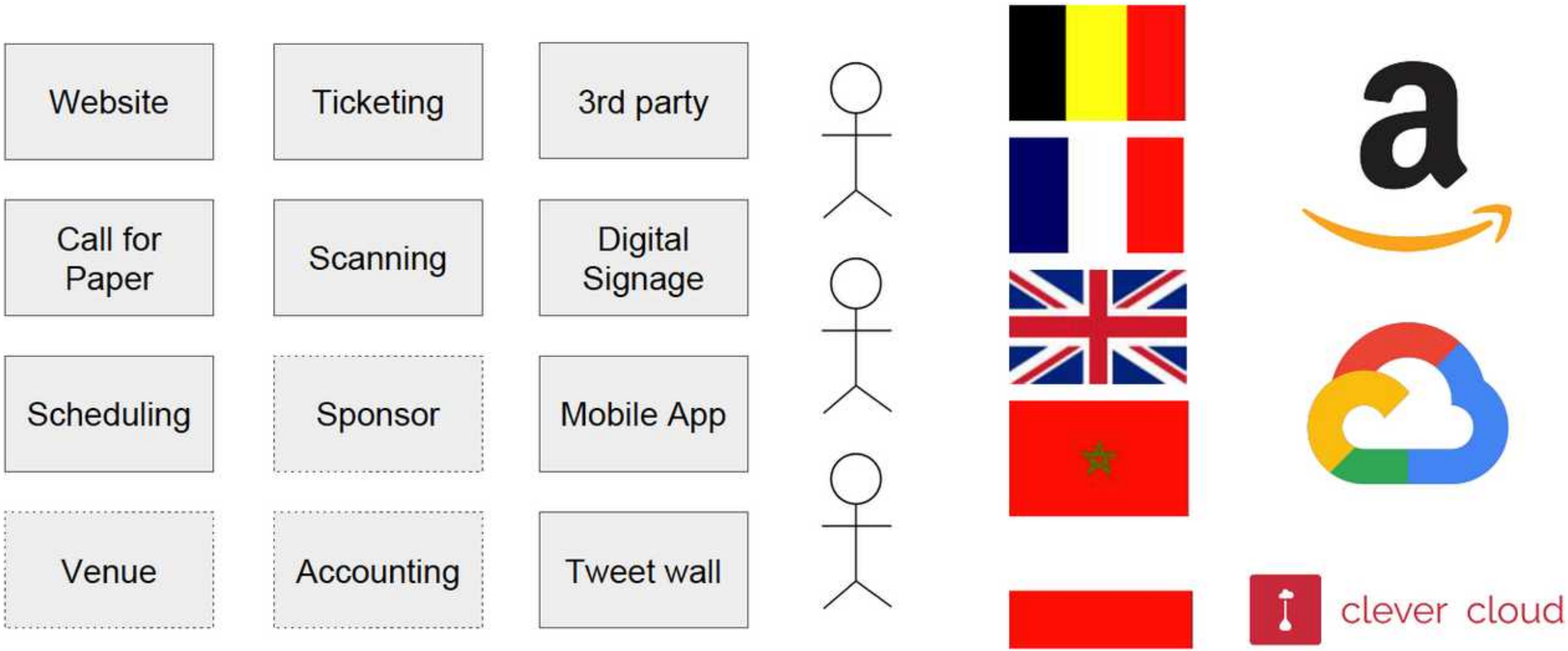
Biggest vendor-independent Java conference in the world

- Devoxx Belgium 2017 : 3400 attendees from 40 different countries
- Several regional and national Devoxx + Devoxx Kids

Need for a conference management system

https://fr.slideshare.net/agoncal/custom-and-generated-code-side-by-side-with-jhipster

Case Study : DEVOXX™



Website

Ticketing

3rd party

Call for Paper

Scanning

Digital Signage

Scheduling

Sponsor

Mobile App

Venue

Accounting

Tweet wall

https://fr.slideshare.net/agoncal/custom-and-generated-code-side-by-side-with-jhipster

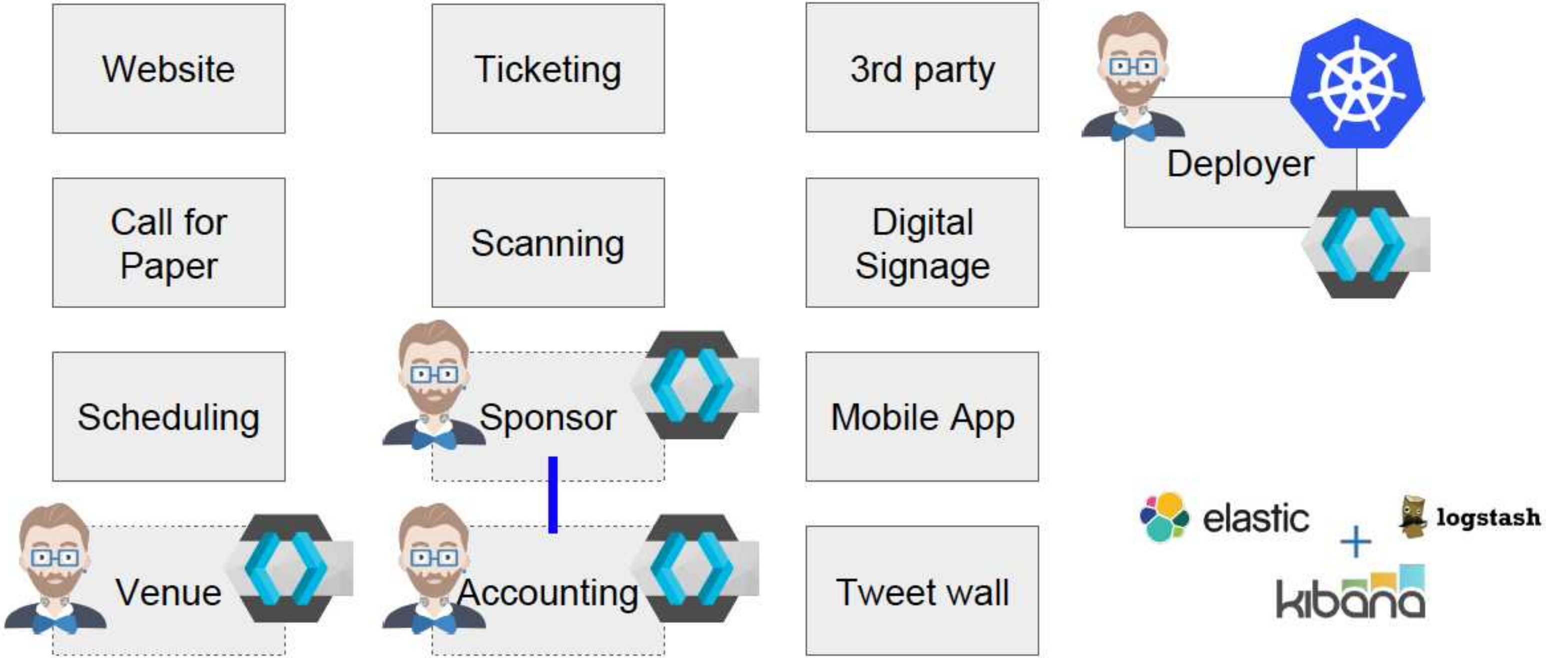
Case Study : DEVOXX™

Website	Ticketing	3rd party
Call for Paper	Scanning	Digital Signage
Scheduling	Sponsor	Mobile App
Venue	Accounting	Tweet wall

Grew up organically
No communication
No SSO
No multitenancy
No container
Fed up of polyglot

https://fr.slideshare.net/agoncal/custom-and-generated-code-side-by-side-with-jhipster

Case Study : DEVOXX™



Website

Ticketing

3rd party

Deployer

Call for Paper

Scanning

Digital Signage

Scheduling

Sponsor

Mobile App

Venue

Accounting

Tweet wall

https://fr.slideshare.net/agoncal/custom-and-generated-code-side-by-side-with-jhipster

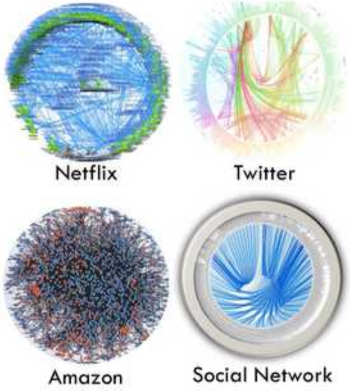
Microservices Benchmarks

TrainTicket

TeaStore

DeathStarBench

- Research paper: http://www.csl.cornell.edu/~delimitrou/papers/2019.asplos.microservices.pdf
- Code available from: https://github.com/delimitrou/DeathStarBench/



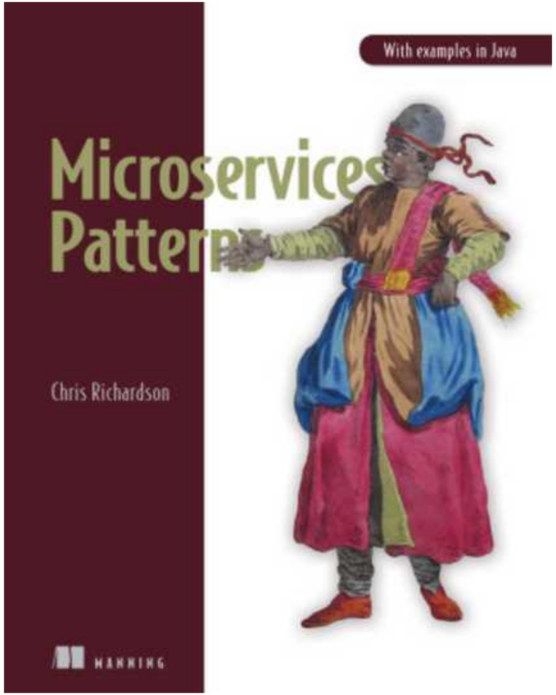
Microservices and FinOps

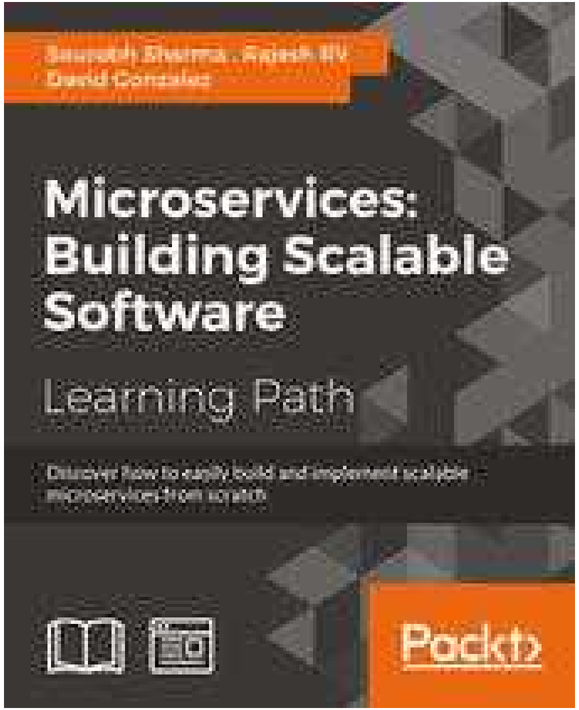
Distributed tracing for FinOps and compliance

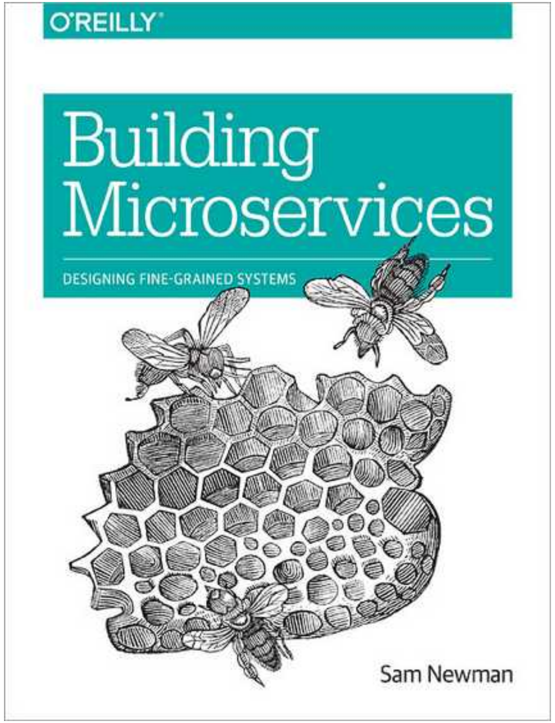
https://horovits.medium.com/observability-into-your-finops-taking-distributed-tracing-beyond-monitoring-48a51e32e78a

Resources

Books about Microservices




Saurabh Sharma , Rajesh RV
David Gonzalez
Microservices: Building Scalable Software
Learning Path
Discover how to easily build and implement scalable microservices from scratch
Packt



More books about Microservices