DISTRIBUTED ARCHITECTURE

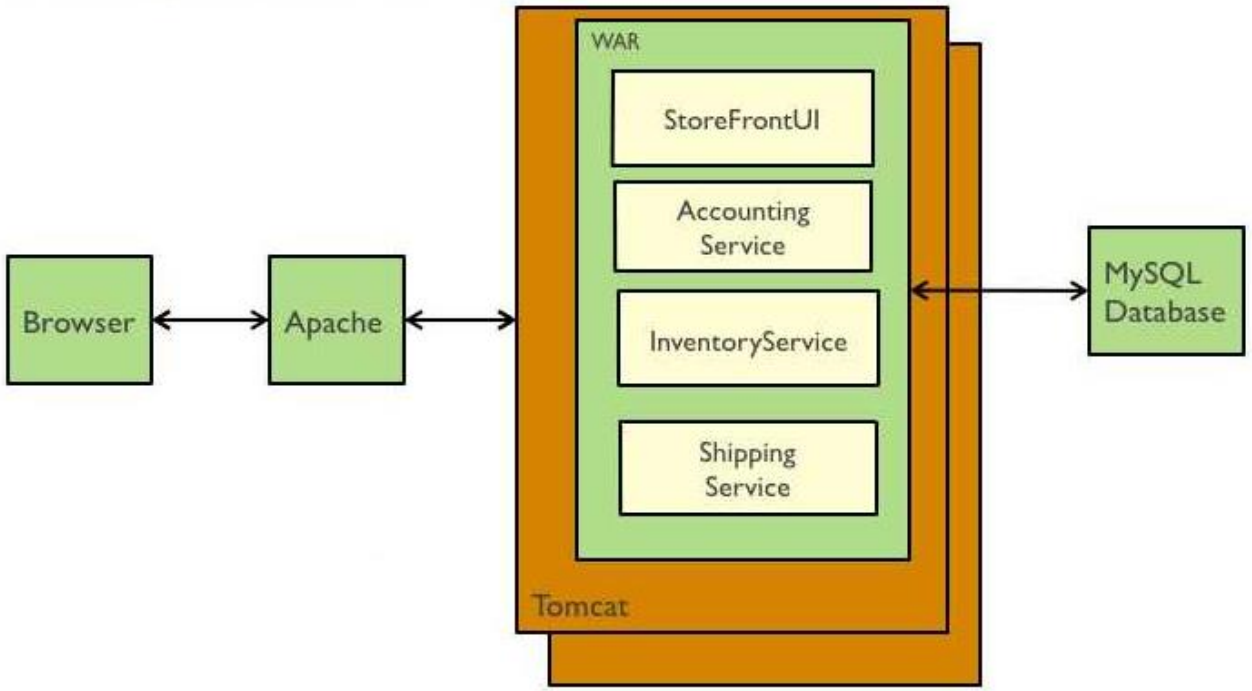
Table Of Content

INTRODUCTION

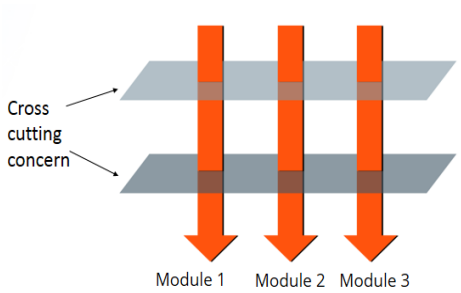
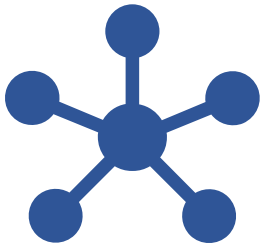
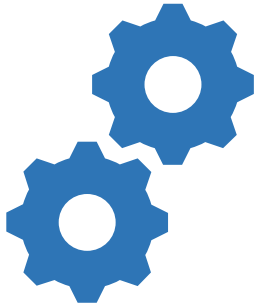
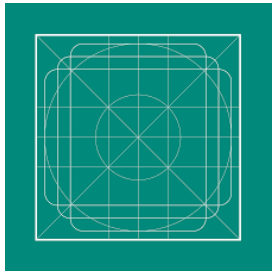
Monolithic Application

Monolithic application has single code base with multiple modules. Modules are divided as either for user interface or business features or technical features. It has single build system which build entire application and/or dependency. It also has single executable or deployable binary.

- Application can be divided into multiple layers like
- **User interface**: This handles all of the user interaction while responding with HTML or JSON or any other preferred data interchange format (in the case of web services).
- **Business logic**: All the business rules applied to the input being received in the form of user input, events, and database exist here.
- **Database access**: This houses the complete functionality for accessing the database for the purpose of querying and persisting objects. A widely accepted rule is that it is utilized through business modules and never directly through user-facing components.



Monolithic Application Pros and Cons

Pros	Cons
 Fewer Cross-cutting Concerns	 Tight Coupling
 Less Operational Overhead	Hard To Understand
 Performance	 Complexity to Add New Feature

Business Disruption

Only 12% of the 1955 Fortune 500 firms were operating in 2015.

The average lifespan of a company on the S&P 500 has decreased from 90 years in 1935 to 18 years today.

86% of ITDMs say they are under "moderate" to "extreme" pressure to deliver IT services faster than they did last year.

82% of organizations are making changes to policy and IT infrastructure to support the proliferation of personal devices.

Global reach through exponential growth in **mobile and social**

Consumers want innovation **and are willing to take more risks**

Barriers to entry have evaporated with **cloud, open APIs, open source**

Change in Business Requirements

Access from Every where

Everything will be digital

Everything needs to connect

Mobile First

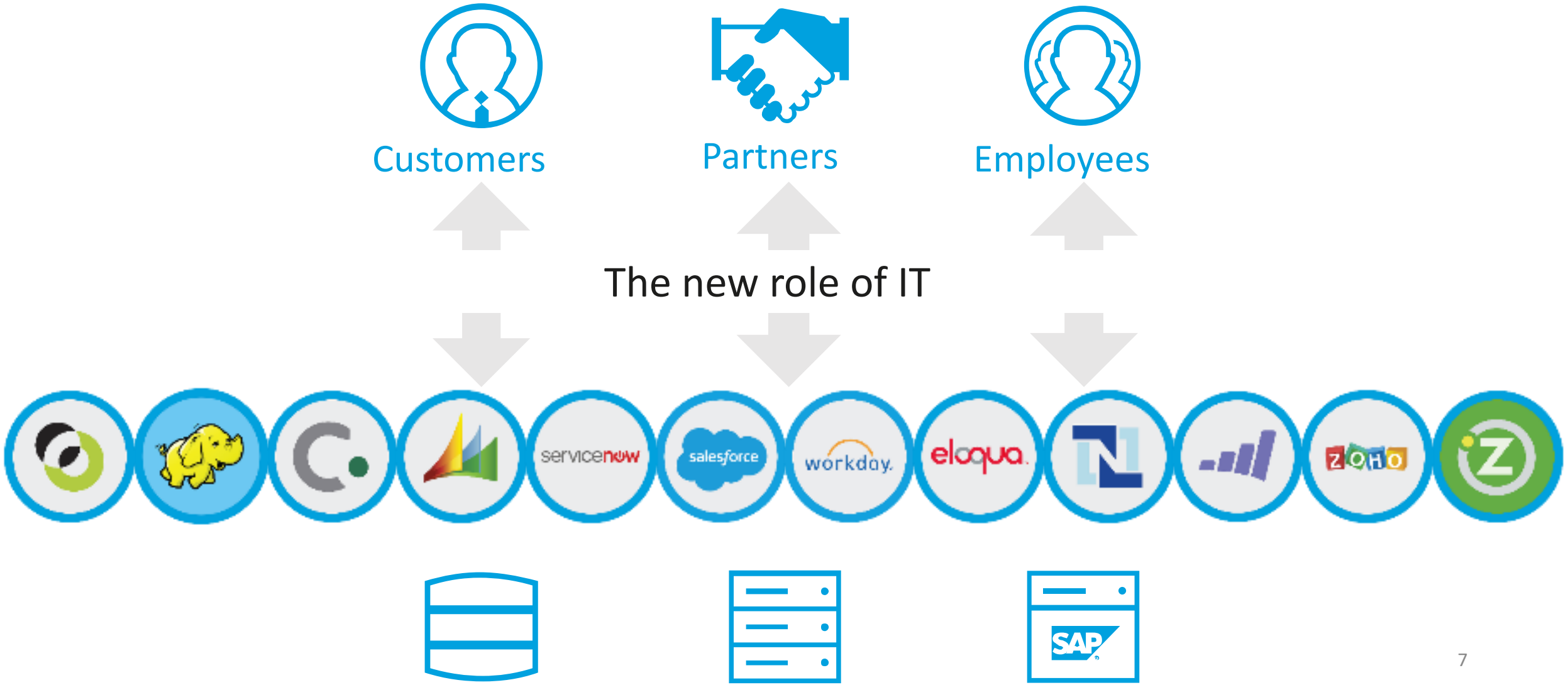
Channel Agility

State of the art Digital experiences

- Accelerate software development cycle
- Minimize hardware and license costs by using leading open source products
- Improve coverage of automation (testing, DevOps) for new development
- Improve developers productivity
- Harmonize technology choices
- Designed with resilience in mind, not avoiding disruptions
- Release at any time, without service disruptions

Role Of Technology

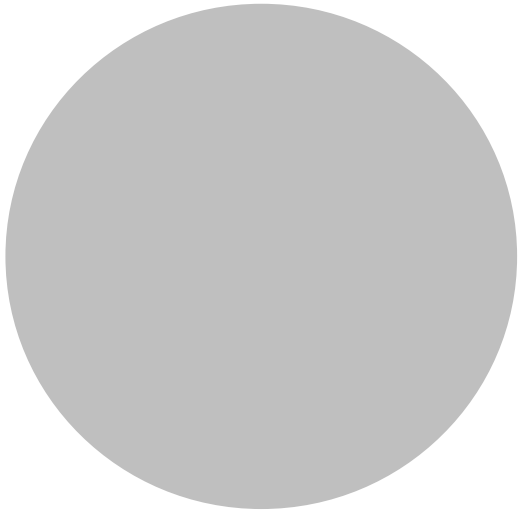
The New Role of IT – Connecting Assets to Audiences



COMPARATIVE ARCHITECTURE PATTERNS

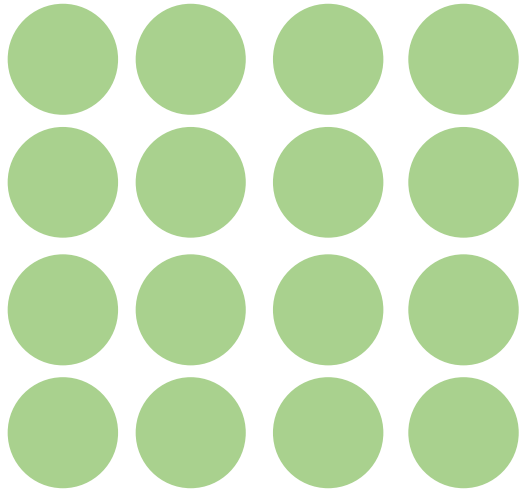
Monolithic VS API Centric VS SOA VS Microservices

Monolithic
Single Unit



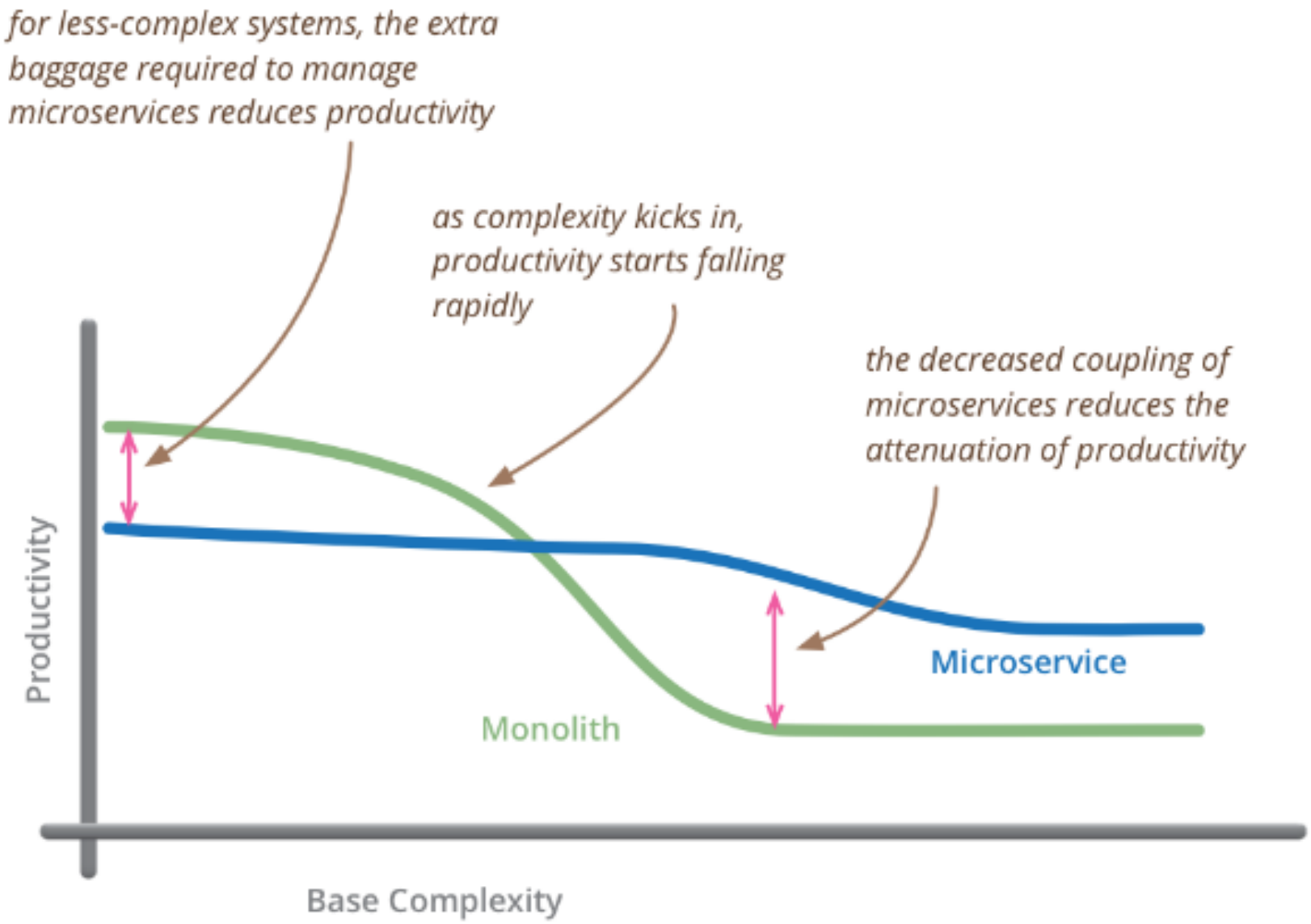
API Centric
Single Unit API

SOA
Coarse Grained

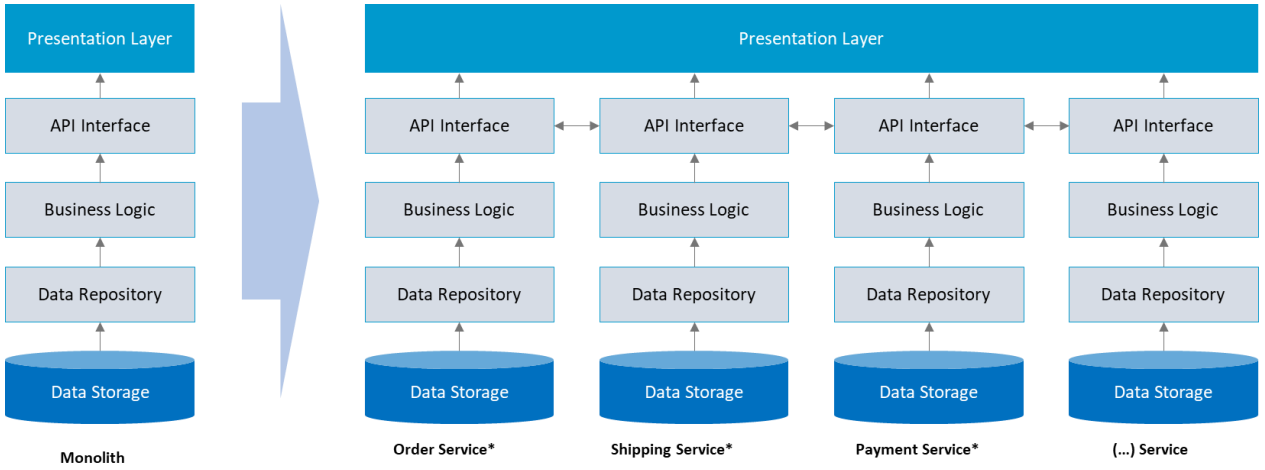


Micro Services
Fine Grained

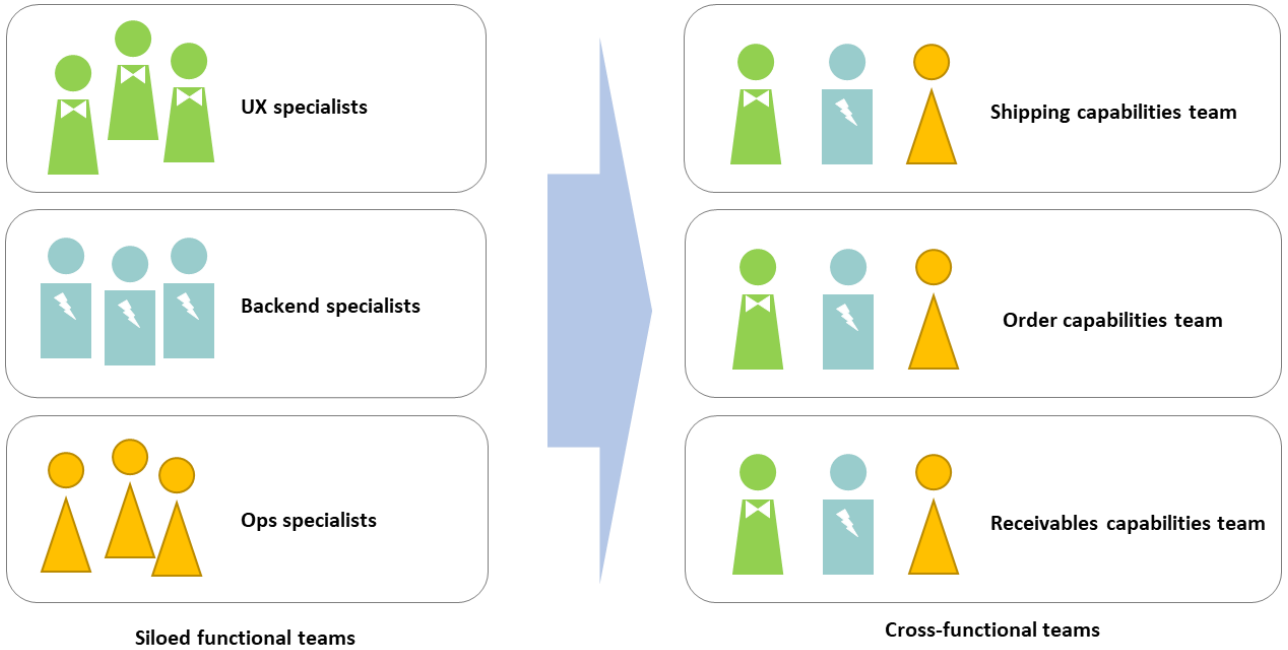
Microservices Vs Monolithic Application



Microservices Vs Monolithic Application

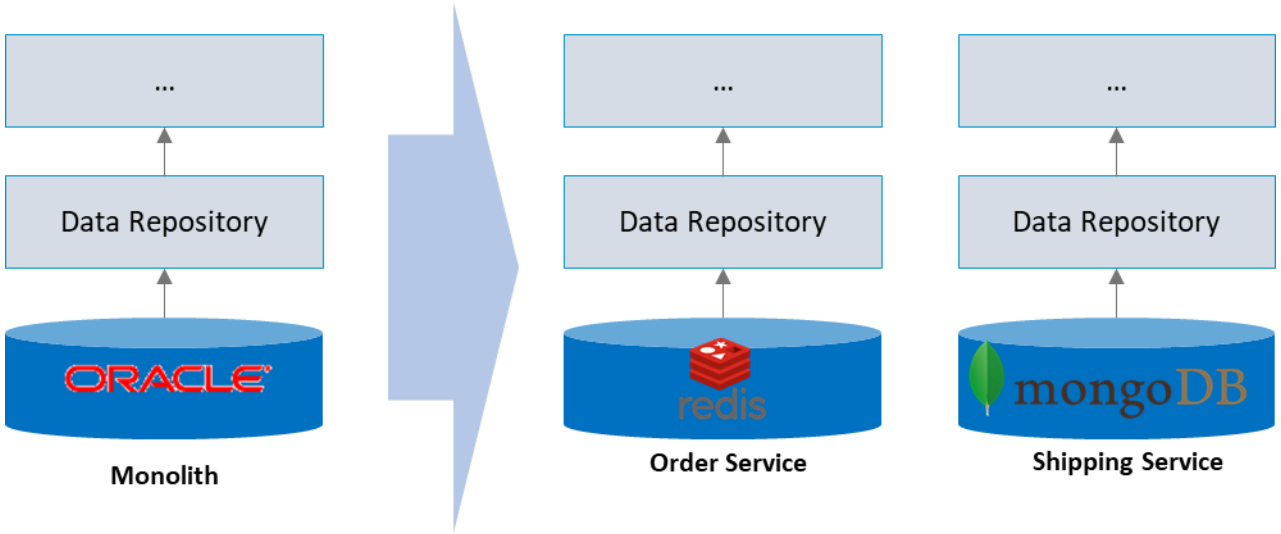


- Smaller Services
- Loosely coupled Services
- Multiple projects

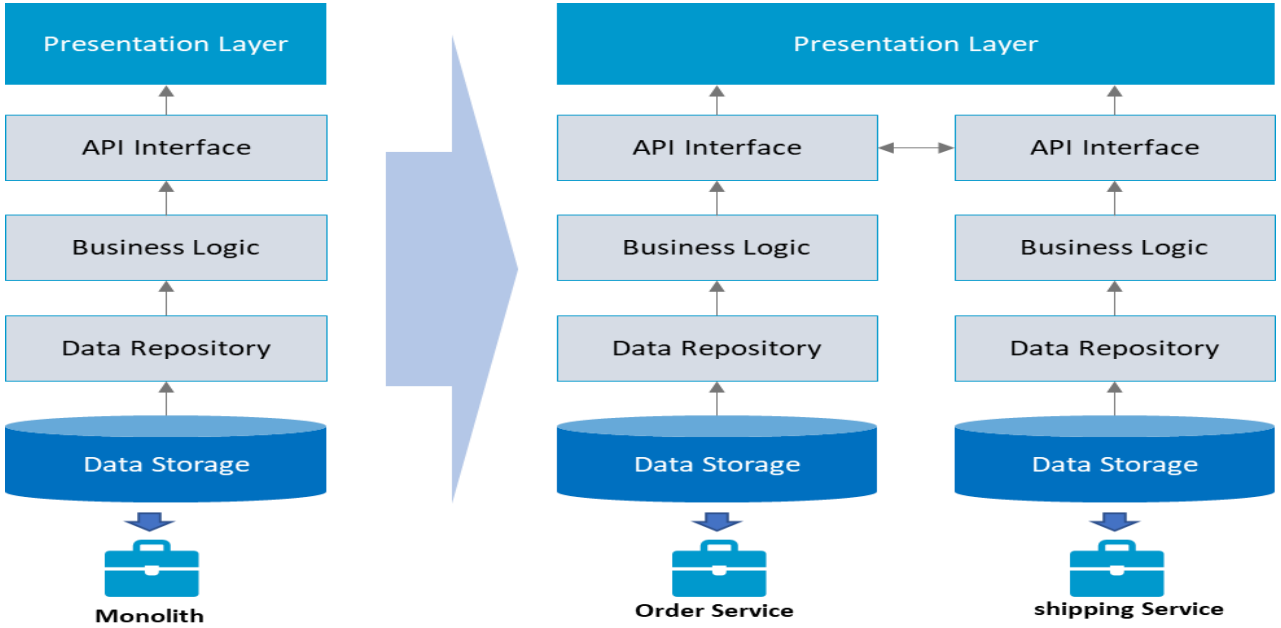


- Services Organized around business instead of technical capabilities.
- Business & It collaboration is increased.

Microservices Vs Monolithic Application

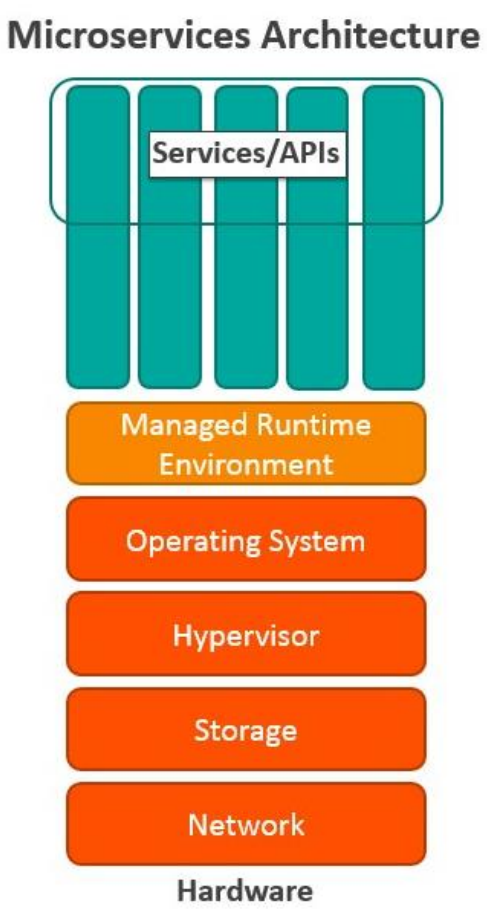


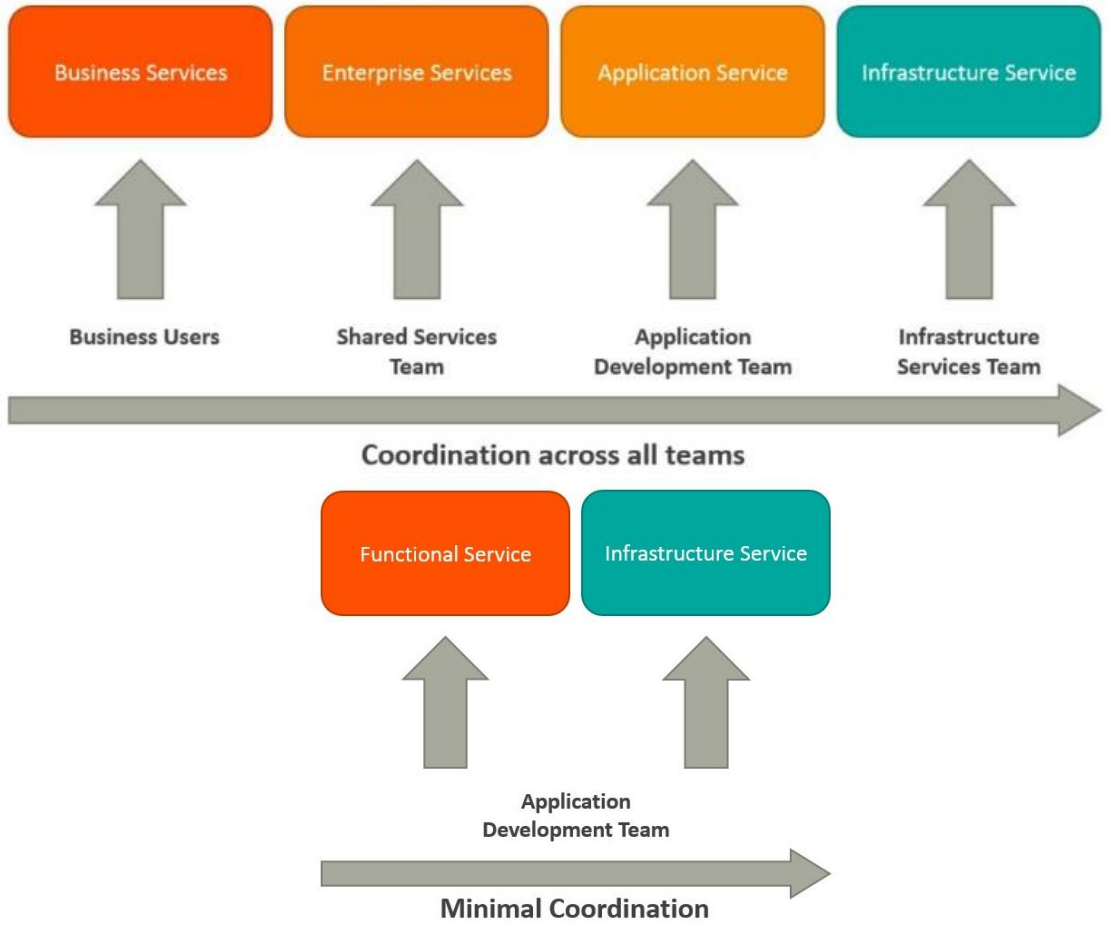
- Each service can choose for the best data store for the business area they are responsible for.
- Each data store can scale independently, depending on the service's performance.



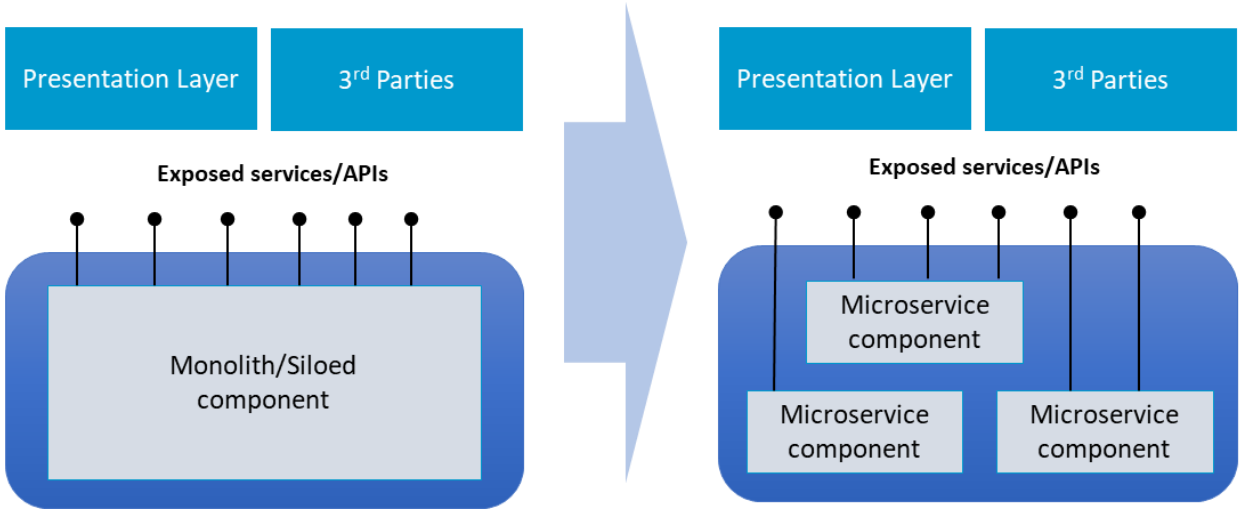
- Change in a functionality require only project's service deployment and the service's own regression test.
- Deployment is fast, since only the service needs to be updated

Microservices Vs SOA Application

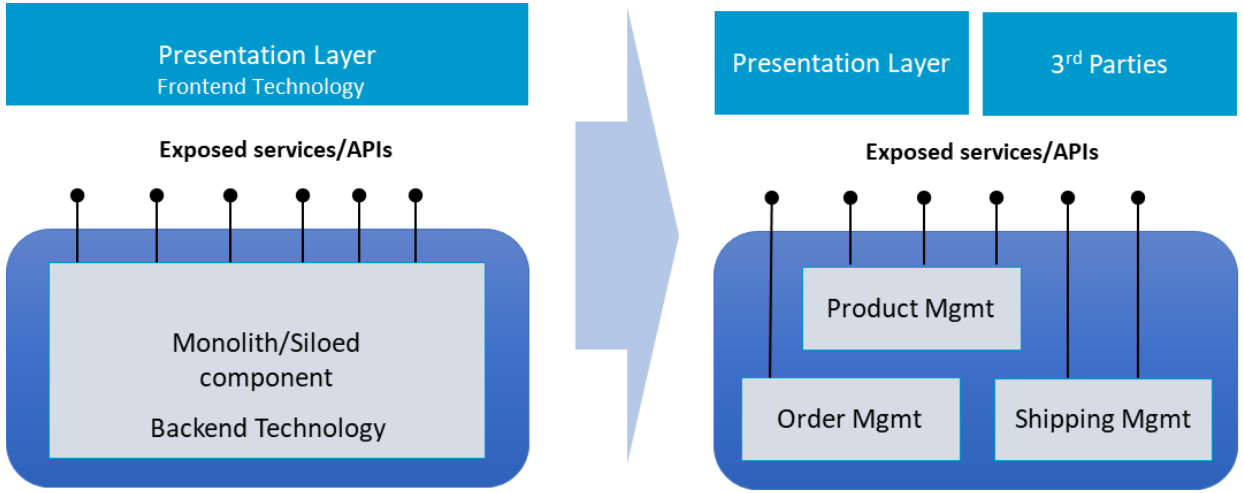




Microservices Vs API Centric Application



- Although business logic is exposed via an API, what's behind these APIs is typically still a monolith. This is opposed to Microservices, where each services that exposes an API has a bounded context

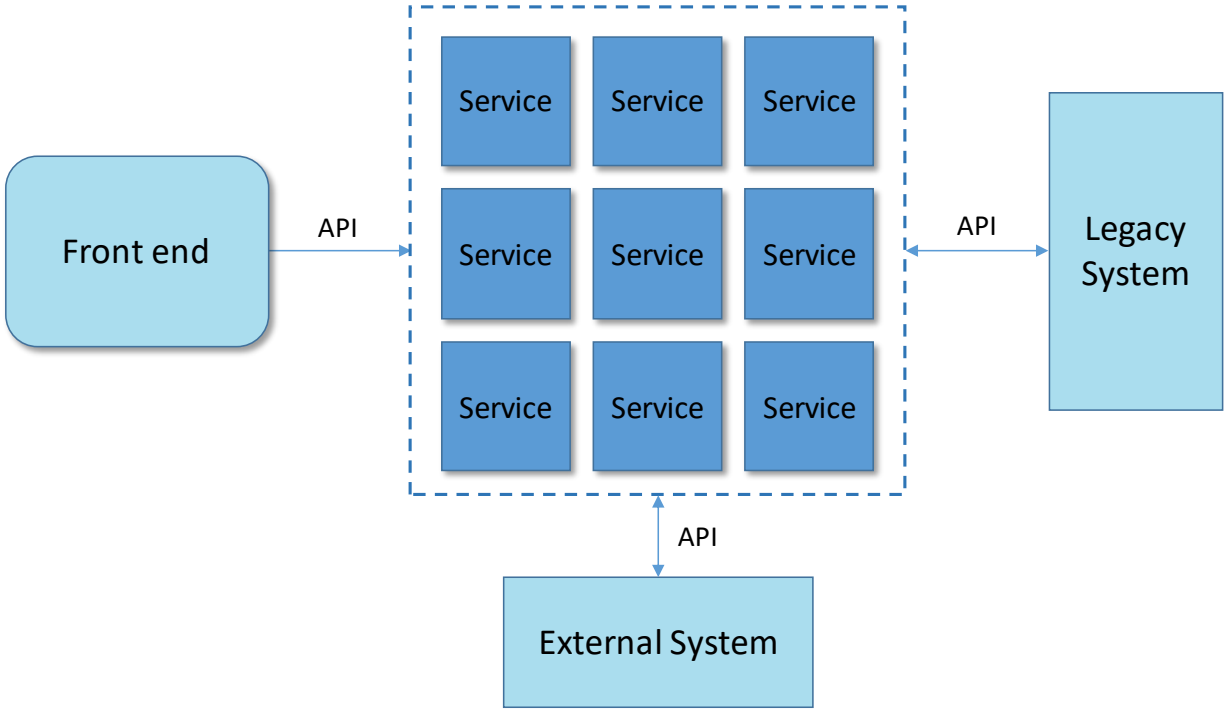


- An API-Centric Architecture effectively decouples front-end and back-end, but this is a decoupling around technology layers.

MICROSERVICES INTRODUCTION

Overview

Microservices split an application into small deployable, running in their own process spaces and communicating via light-weight protocols



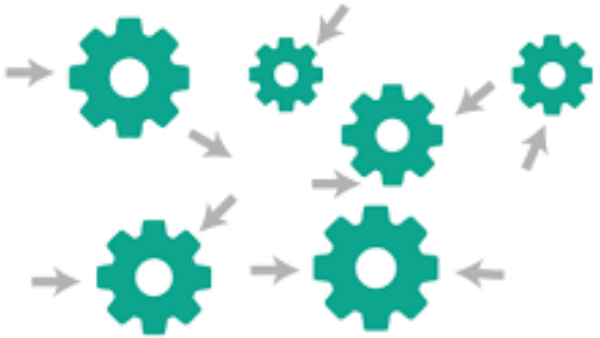
- Have to be loosely coupled
- Include logic and data
- decentralized governance & data management
- Are owned by one team
- Allow autonomous teams with isolated implementation
- May include GUI
- Treated as cattle not pets.
- Smart endpoints, dumb pipes
- Standardize on integration, not platform
- Better Security due to privilege separation

Drivers & Principle

Drivers

- Accelerate software development cycle
- Minimize hardware and license costs by using leading open source products
- Improve coverage of automation (testing, DevOps) for new development
- Improve developers productivity
- Harmonize technology choices
- Designed with resilience in mind
- Release at any time

Solution



Microservice

Principles

- Service-oriented
- Smaller, loosely coupled components
- Designed for scalability
- Distributed
- Resilient
- Lightweight infrastructure
- Embrace the Ecosystem/OSS
- DevOps-enabled
- Platform thinking
- Fail fast

Microservices Complexity

Development Complexity

Distributed

Migration

Versions

Organization

Operational Complexity

Operational Complexity

- How to provision resources in a scalable and cost-efficient way?
- How to operate dozens or hundreds of microservice components effectively without multiplying efforts?
- How to avoid reinventing the wheel across different teams and duplicating tools and processes?
- How to keep track of hundreds of pipelines of code deployments and their interdependencies?
- How to monitor overall system health and identify potential hotspots early on?
- How to track and debug interactions across the whole system?
- How to analyse high amounts of log data in a distributed application that quickly grows and scales beyond anticipated demand?
- How to deal with a lack of standards and heterogeneous environments that include different technologies and people with differing skill sets?
- How to value diversity without locking into a multiplicity of different technologies that need to be maintained and upgraded over time?
- How to deal with versioning?
- How to ensure that services are still in use especially if the usage pattern isn’t consistent?
- How to ensure the proper level of decoupling and communication between services?

Microservices Benefits

Greatly decreased delivery time with streamlined application lifecycle as everything is a container, and can be built and deployed in a totally consistent manner across the landscape

Deploy any technology stack, any framework and any new application component in a matter of minutes as a container

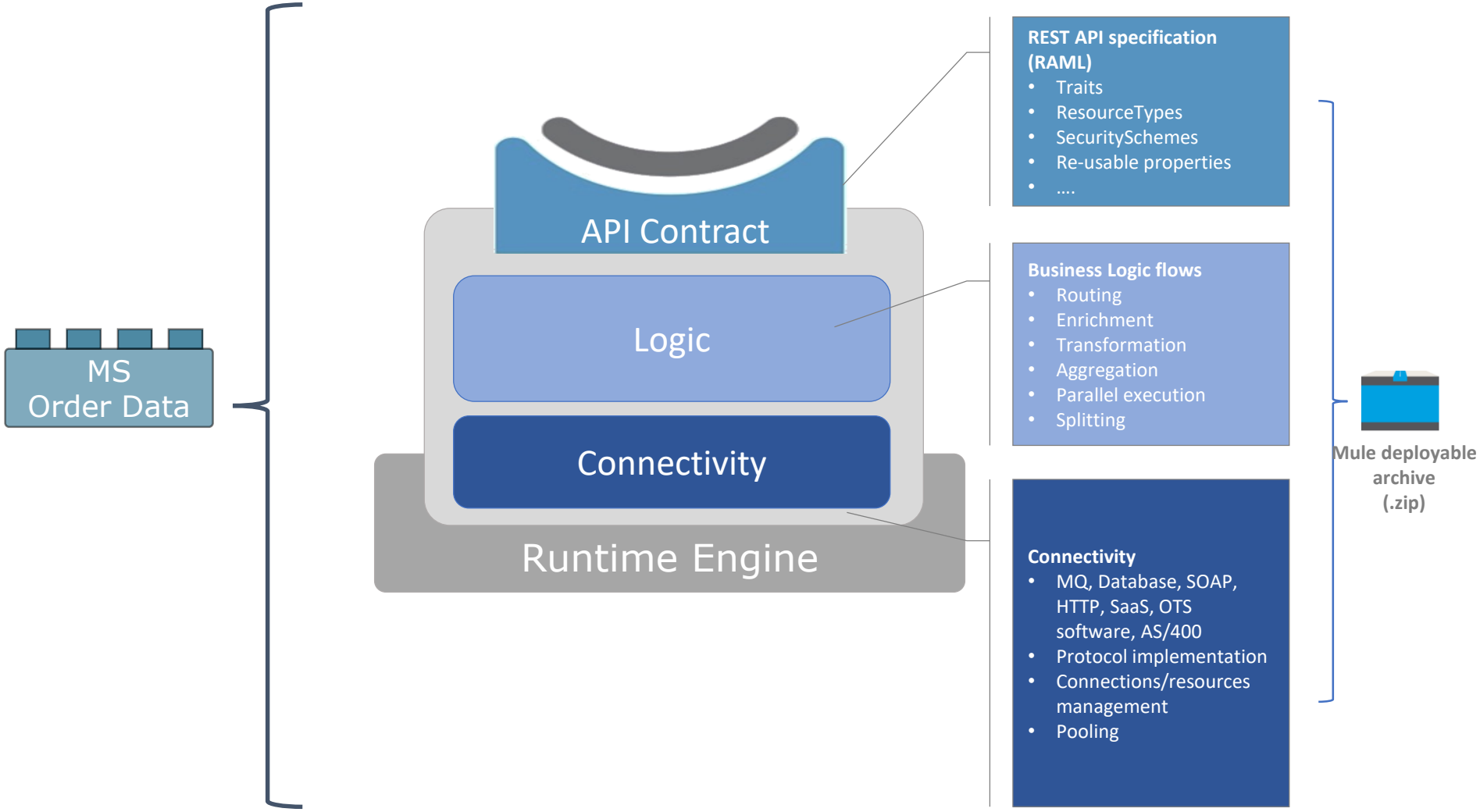
High productivity– never before had developers been able to run the entire application stack locally; now not only they're able to, but can easily recreate any application version in an instant, or simply refresh their environment

Simplified applications – Microservices applications deployed as individual containers have simplified the deployment process to a great extent, compared to the existing architecture

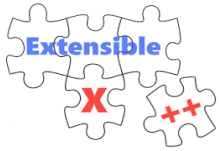
Dramatically lower infrastructure costs as computing resources are pooled as one large multi-tenant cluster that satisfies all project needs, as opposed to limited capacity in proprietary hardware

MICROSERVICES ARCHITECTURE

Microservice Component



Microservices Design Principles



Extensibility

New capabilities or functional services are added with minimal effort and with no impact on the existing functionality. Backward compatibility needs to be considered when changing existing functionality/ capabilities.

Highly Observable

It is crucial to be able to determine the complete state of a system via a joined-up view of what is happening. Log and statistics aggregation and semantic monitoring are relevant for being able to drill down to the source of an issue. Use correlation IDs to allow for tracing of calls through the system

Fail fast/ Isolate Failure

A fail-fast system is designed to immediately report at its interface any failure or condition that is likely to lead to failure. Fail-fast systems are usually designed to stop normal operation rather than attempt to continue a possibly flawed process. Such designs often check a system's state at several points in an operation, so any failures can be detected early. Especially when invoking functionality of other modules or when executing remote/ web service calls. A fail-fast module passes the responsibility for handling errors (but not detecting them) to the calling module.

Replace-ability

Promote ease of innovation through disposable code. It is easy to fail and move on.

Modeled around business domains

Services should be modeled along clearly separable business functionalities which allows to better reflect changes in business processes. Usually this business domain should also be assigned to only one responsible business owner.

Microservices Design Principles

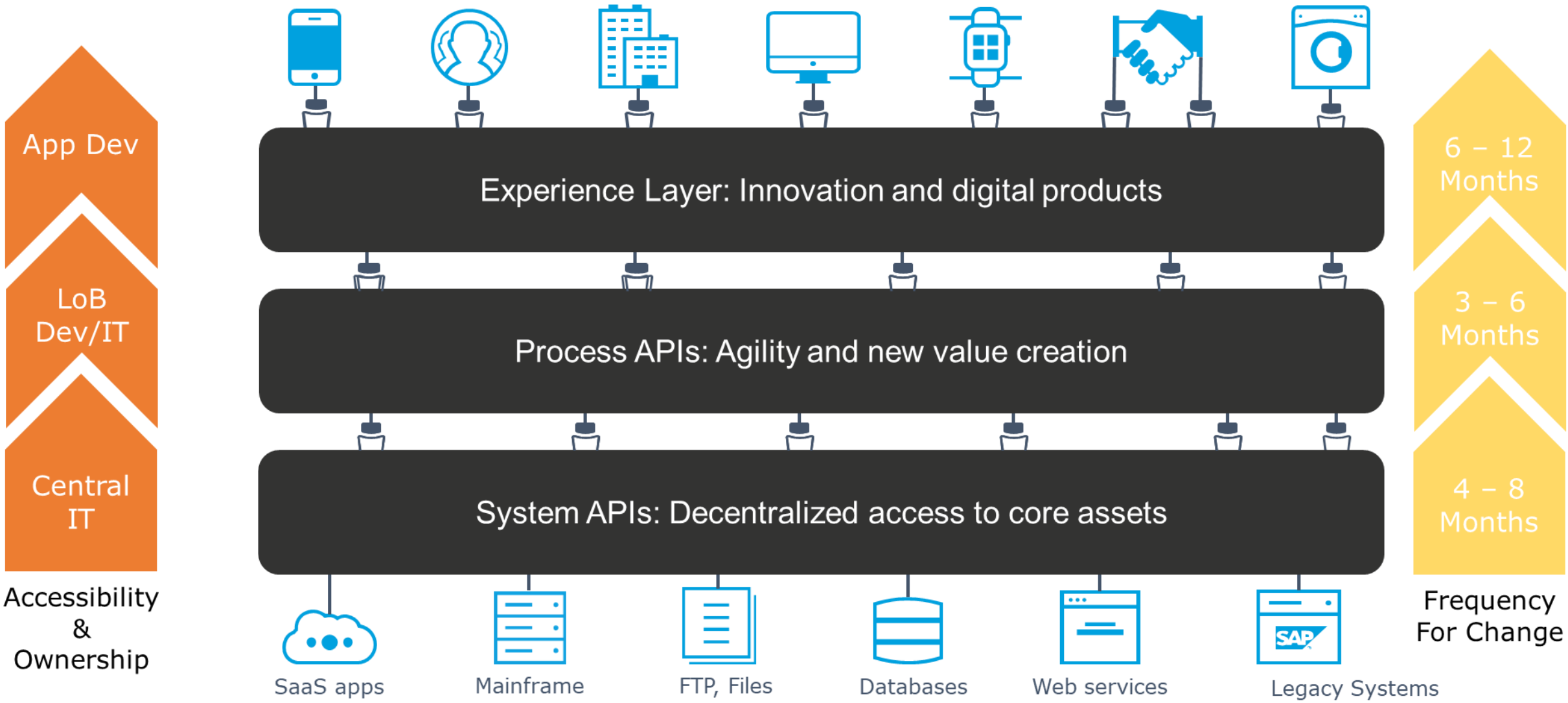
Single Responsibility
MSAs should have only one specific area of responsibility. In case that data from other areas is required, the service needs to request this data from the respective data owning service.

Same Technology Base
For standardization reasons a reference technology stack (e.g. Java, .NET, JavaScript, etc.) should be used for implementation of MSAs.

One Operator
Operations of all MSAs should be unified: Monitoring: provide a standard interface for health status and basic monitoring metrics (e.g. runtime parameters, utilization, etc.). Logging: overall log aggregator. Deployment: automatic and easy (re)deployment. Shutdown/ (re)start: automated restarts of failed nodes, automated starting/ stopping of nodes.

Statelessness
Services should not hold any state data whenever possible in order to remove resource overhead for state management. State data is handled by the service consumer or by an external component and needs to be supplied to the service when needed. By reducing resource consumption, the service can handle more requests in a reliable manner.

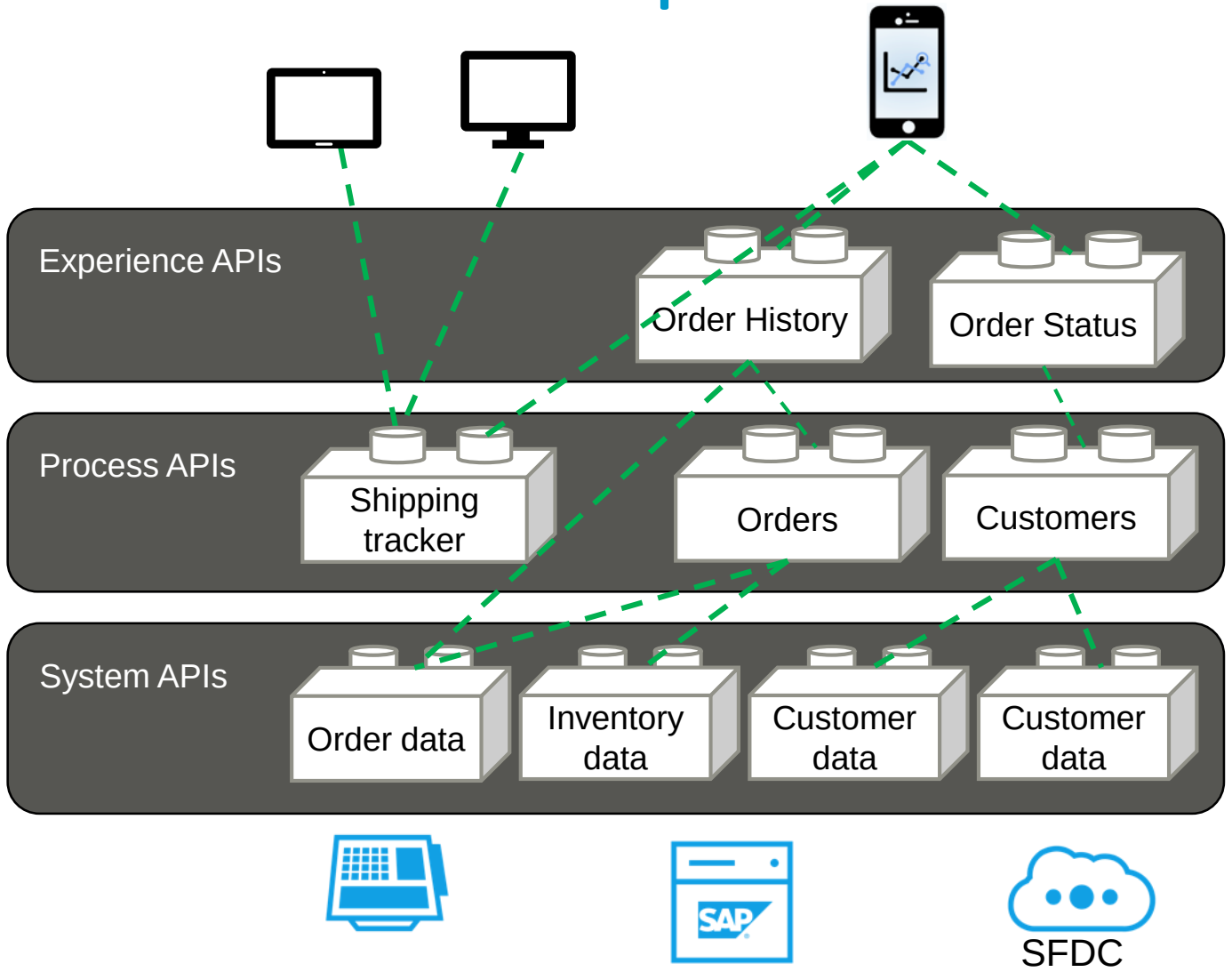
Microservices Classification



Microservices Classification Example

Benefits

1. Architecture
2. Discoverability
3. Engagement
4. Speed
5. Innovation



MICROSERVICES DESIGN CONSIDERATIONS

Design Microservices Architecture

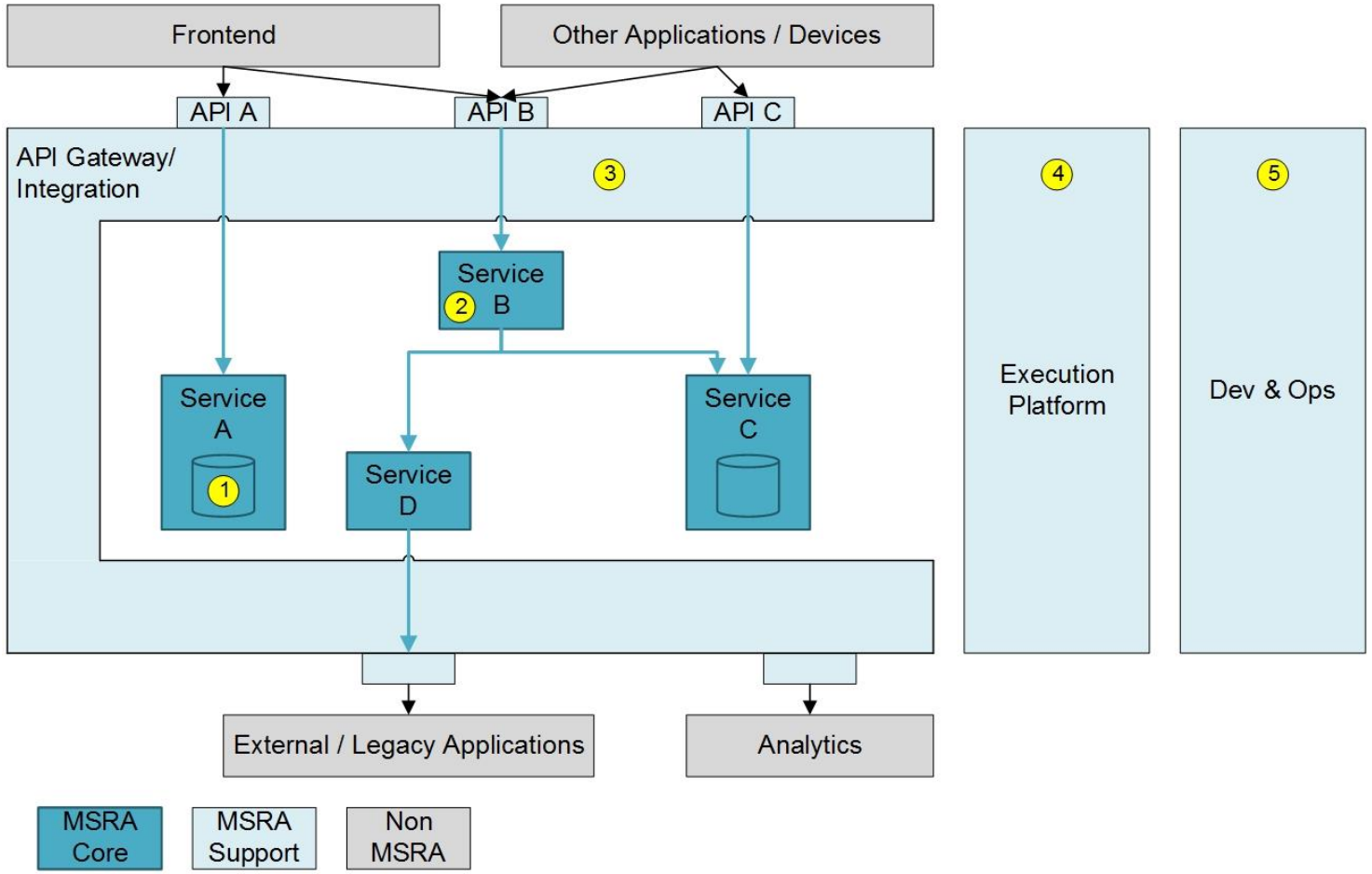
Design challenges –

- High-number of moving parts
- Distributed nature of the Architecture
- Change in terms of organization and teams' dynamics

Design considerations -

- Systems Design Considerations for Microservices Architectures
- Strategies for designing Microservices Architecture
- Microservices Project Strategies

Microservices Architecture Decision



1
2

Microservice

- Runtime (app server vs. no server)?
- Synch. vs. asynch. processing?
- Packaging (container, images, …)?
- Inter-service dependencies (data owner & replication vs. direct calls)?
- Required service-endpoints (/health, /info)?
- Format (json, protobuf, thrift …)?

3

Integration

- Direct calls vs. service registry vs. messaging?
- Use API Manager to control access to services?
- How to handle services lookups (discovery, routing, etc.)?
- IAM?

4
5

Exec Platform & Dev & Ops

- Deployment model?
- Load balancing (per MS vs. central vs. client side)?
- How to handle distributed logging?
- How to handle distributed metrics gathering?
- How to handle distributed monitoring?
- How to handle configurations?
- How to handle secrets management?

Microservices Architecture Designing Strategies

- Design around “Business Capabilities”
- Microservices Decomposition
  - DDD and BDD
  - Event Storming
- Size and Scope of Microservices

Design Services Around Business Capabilities

When designing Microservices, the most common pattern for identification of services is the use of “Business Capabilities” of the application domain. Each Microservices implements a part of the application domain.

This enables the possibility to:

- Separate the domain on multiple independent Microservices.
- Have each Microservices doing one thing well (*single-responsibility*).
- Have different teams working independently on different components at the same time.

Microservices Decomposition

There are several techniques for identification and design of Microservices. The most popular are:

Domain Driven Design (DDD)

- Methodology used to model application domains.
- Can be used to identify Microservices “boundaries” and how they inter-relate with each other.
- DDD offers several other principles and guidelines to govern the modelling of inter-microservice relationships and internal microservice implementation.
- DDD defines the “ubiquitous language” that governs the discussions and concrete implementations of Microservices. The actual implementations may follow other approaches, such as Behavioral-Driven Design (BDD), but they are done using the common understanding provided by DDD
- Developed by Eric Evans, reference: *Domain-Driven Design: Tackling Complexity in the Heart of Software*.

Event Storming

- Workshop technique that enables identifying application domains involving different stakeholders (developers, architects, business owners, etc.).
- Especial focus on identification of “domain events” (part of DDD). However, from that process it also enables discovering other elements of the domain. model
- Developed by Alberto Brandolini, reference: *Event Storming*

Domain Driven Design (DDD)

DDD is an approach to identify models and implementation strategies of software systems. It provides (among other things) guidelines for:

Strategic design

- Think about the different aspects and parts of the system being modelled

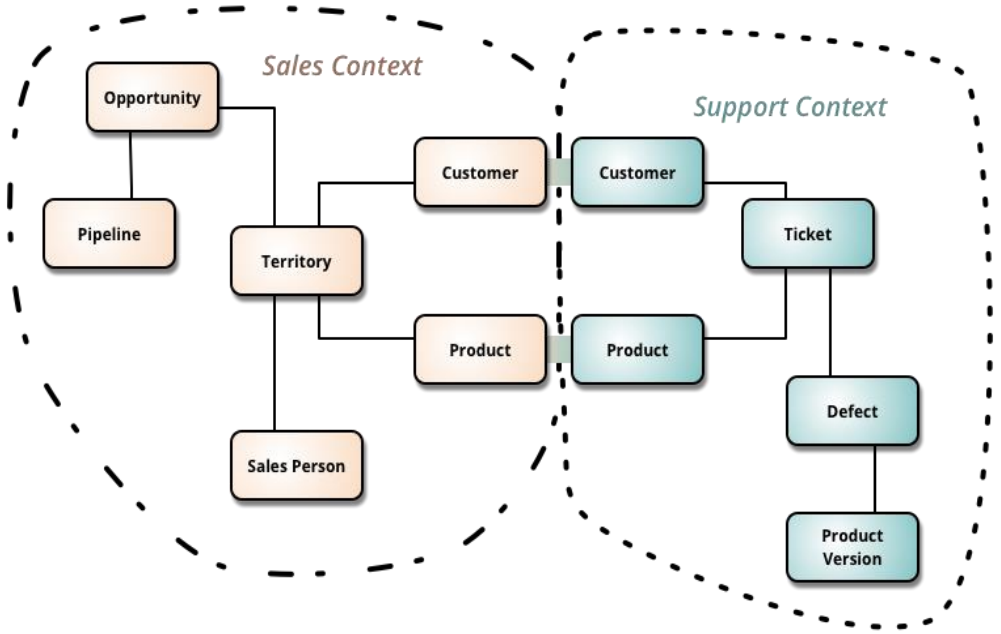
Domain modelling:

- Define a common language (*ubiquitous language*) for the domain, which is composed of several independent parts (*bounded contexts*) and their relationships (*context map)*
- Defines elements ("*building blocks*") to model the internals of each *bounded context*

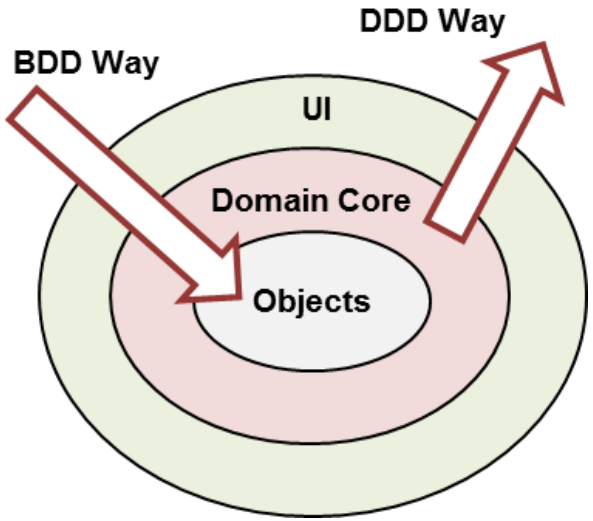
Model integrity:

- Defines strategies and patterns for defining and maintaining "inter-bounded context" integrations

DDD fosters the identification of independent contexts within an application domain, and how to preserve their integrity. Example: there can be a Sales and Support contexts. Each of these contexts can be implemented independently of each other (with proper inter-service communication and integration, as different contexts have specific representations of particular objects, e.g.: customer and product). This is why DDD is popularly used to implement microservices architectures.



DDD and BDD usage



Domain-Driven Design focus:

- Inside/Middle-out Approach (understand domain)
- Provide common vocabulary and structure

Behavioral-Driven Design focus:

- Outside-in Approach (understand user requirements to build)
- Provide structure to collect detailed requirements to build software

Guidelines:

- First define common domain understanding, using DDD and Event Storming (*problem space definition*)
- Use BDD (and similar approaches) to driven concrete implementations (using common understanding, elements and language defined from domain design)
- Developers/Business must be involved on DDD and BDD efforts, so they have a common language and understanding of domain and developments

Size and Scope of Microservices

One of the most common discussions around Microservices is: “how big should a Microservice be?”. The short answer is: they should be designed and built based on “scope”, not “size”.

Size Considerations:

- Though “scope” plays a more important weight, early adopters of Microservices propose having Microservices build and maintained by small teams (a “two pizza box” should be enough to feed the team). Given that, considerations on the size of a microservice should be taken into account, so that:
  - Small team is able to build and maintain the microservice (and most of the times a series of Microservices)
  - New team members can quickly catch-up with the codebase and start building new things

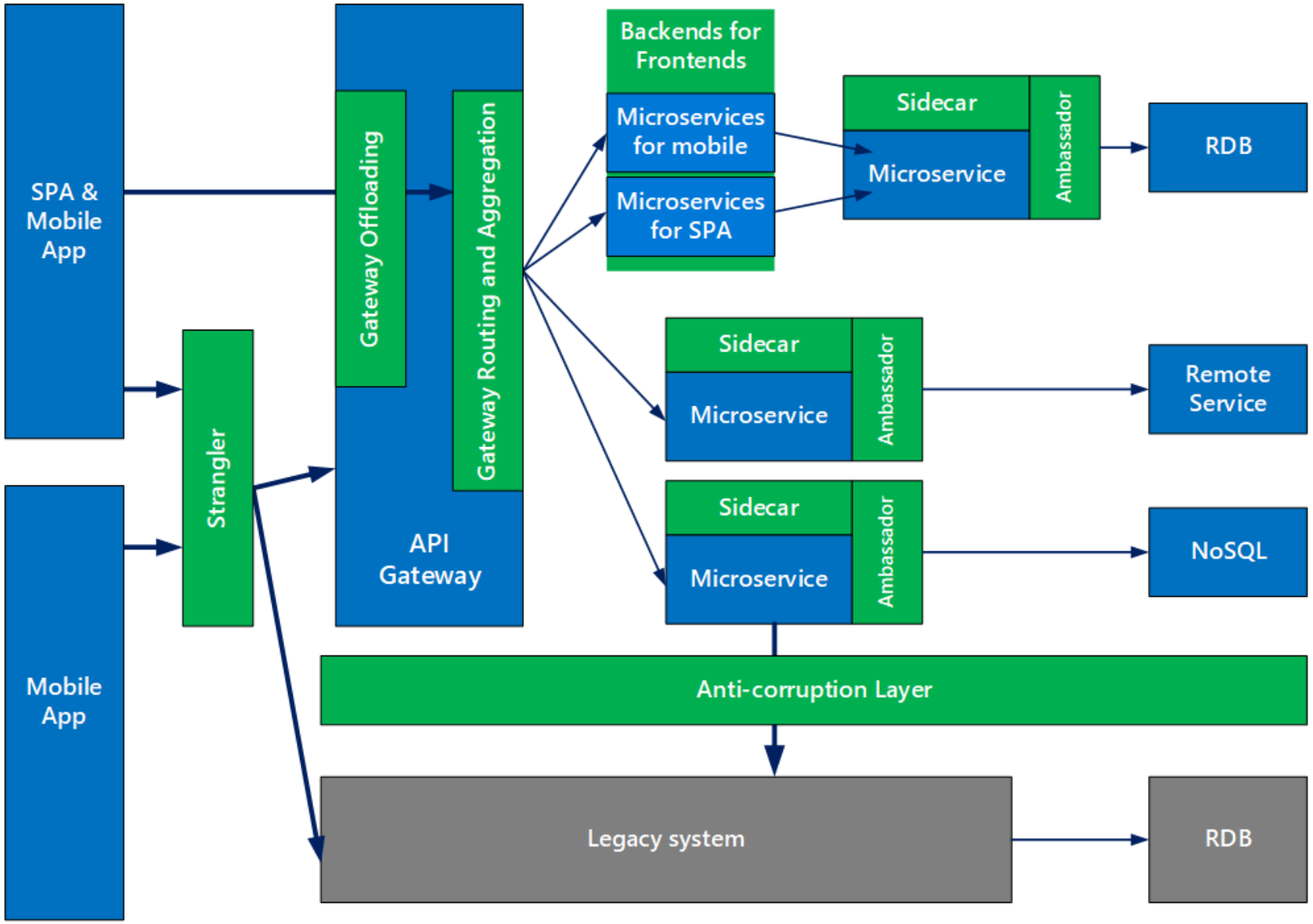
Scope Considerations:

- Build Microservices around Bounded Contexts (from DDD)
  - Clear scope on the application domain, with clear view on the relations with other Bounded Contexts
- Single-Responsibility Principle (SRP)
  - Every microservice should be designed around one responsibility over a single part of the functionality provided by the application. The responsibility should be entirely encapsulated by the Microservices, i.e.: all its functionality should be narrowly aligned with that responsibility.
  - Rule of thumb: a microservice should only have one scope of changes.

MICROSERVICES PATTERNS

DESIGN PATTERNS

Microservices Design patterns



Ambassador pattern

Context and problem:

- Resilient cloud-based applications require features such as circuit breaking, routing, metering and monitoring, and the ability to make network-related configuration updates.
- It may be difficult to update legacy applications or existing code libraries to add these features.
- Network calls may also require substantial configuration for connection, authentication, and authorization.

Solution:

- Put client frameworks and libraries into an external process that acts as a proxy between your application and external services.
- Deploy the proxy on the same host environment as your application to allow control over routing, resiliency, security features, and to avoid any host-related access restrictions.

When to use this Pattern:

- Often used with legacy applications, or other applications that are difficult to modify, in order to extend their networking capabilities.
- It is useful when need to build a common set of client connectivity features for multiple languages or frameworks.
- Do not use it when network request latency is critical.
- Do not use it when connectivity features cannot be generalized and require deeper integration with the client application.

Anti-Corruption Layer pattern

Context and problem:

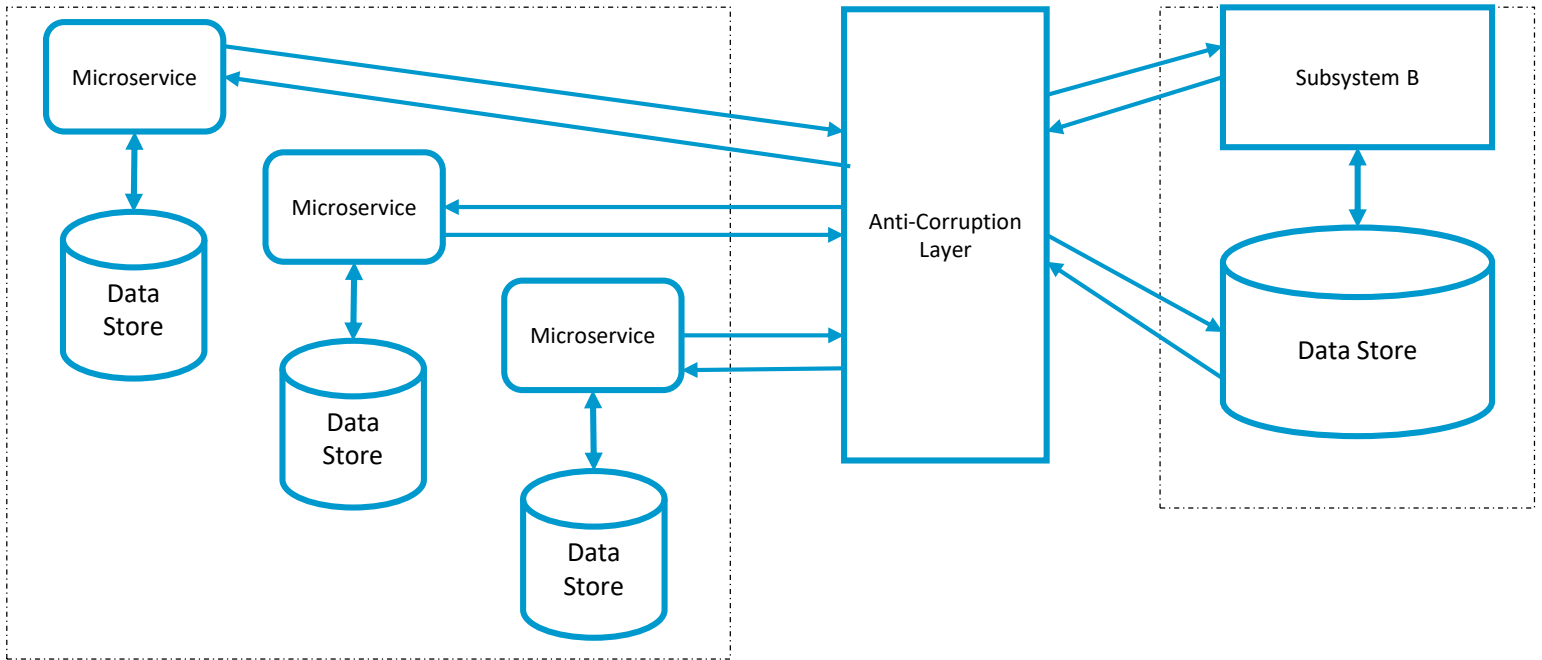
- Most applications rely on other systems for some data or functionality. Often these legacy systems suffer from quality issues such as convoluted data schemas or obsolete APIs.
- Maintaining access between new and legacy systems can force the new system to adhere to at least some of the legacy system's APIs or other semantics.

Solution:

- Implement a façade or adapter layer between different subsystems that don't share the same semantics.

When to use this Pattern:

- When a migration is planned to happen over multiple stages, but integration between new and legacy systems needs to be maintained.
- When two or more subsystems have different semantics, but still need to communicate.
- This pattern may not be suitable if there are no significant semantic differences between new and legacy systems.



Backends for Frontends pattern

Context and problem:

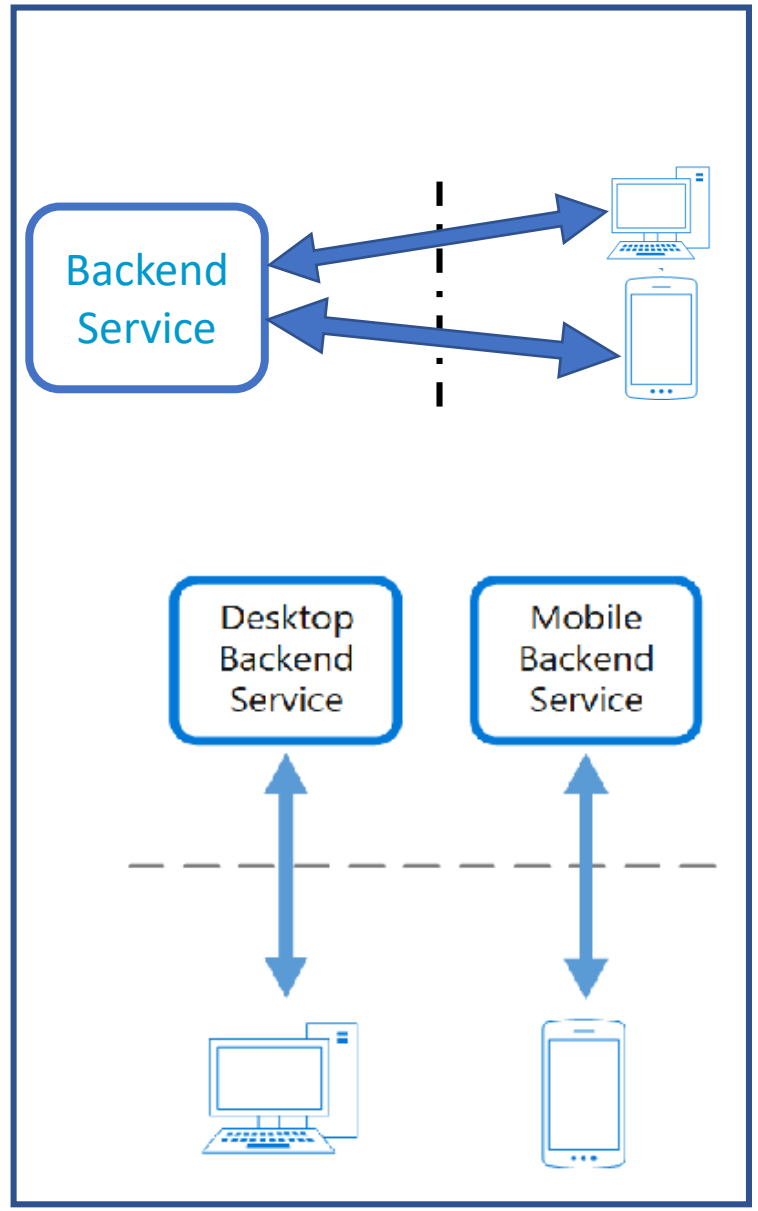
- A cloud-based application may include multiple services, with each service having one or more consumers.
- Excessive load or failure in a service will impact all consumers of the service.
- Moreover, a consumer may send requests to multiple services simultaneously, using resources for each request.

Solution:

- Create separate backend services to be consumed by specific frontend applications or interfaces, it can be optimized for that interface.

When to use this pattern:

- When a shared or general purpose backend service must be maintained with significant development overhead.
- When you want to optimize the backend for the requirements of specific client interfaces.
- An alternative language is better suited for the backend of a different user interface.



Bulkhead pattern

Context and problem:

- Application may include multiple services, with each service having one or more consumers.
- A consumer may send requests to multiple services simultaneously, using resources for each request.
- When the consumer sends a request to a service that is misconfigured or not responding, the resources used by the client's request may not be freed in a timely manner.

Solution:

- ***Bulkhead pattern*** suggest to isolate elements of an application into pools so that if one fails, the others will continue to function.
- This pattern helps to isolate failures, and allows you to sustain service functionality for some consumers, even during a failure.
- A consumer can also partition resources, to ensure that resources used to call one service don't affect the resources used to call another service.

When to use this pattern:

- To isolate resources used to consume a set of backend services.
- To isolate critical consumers from standard consumers.
- To protect the application from cascading failures.
- This pattern may not be suitable when less efficient use of resources may not be acceptable in the project.

Gateway Aggregation pattern

Context and problem:

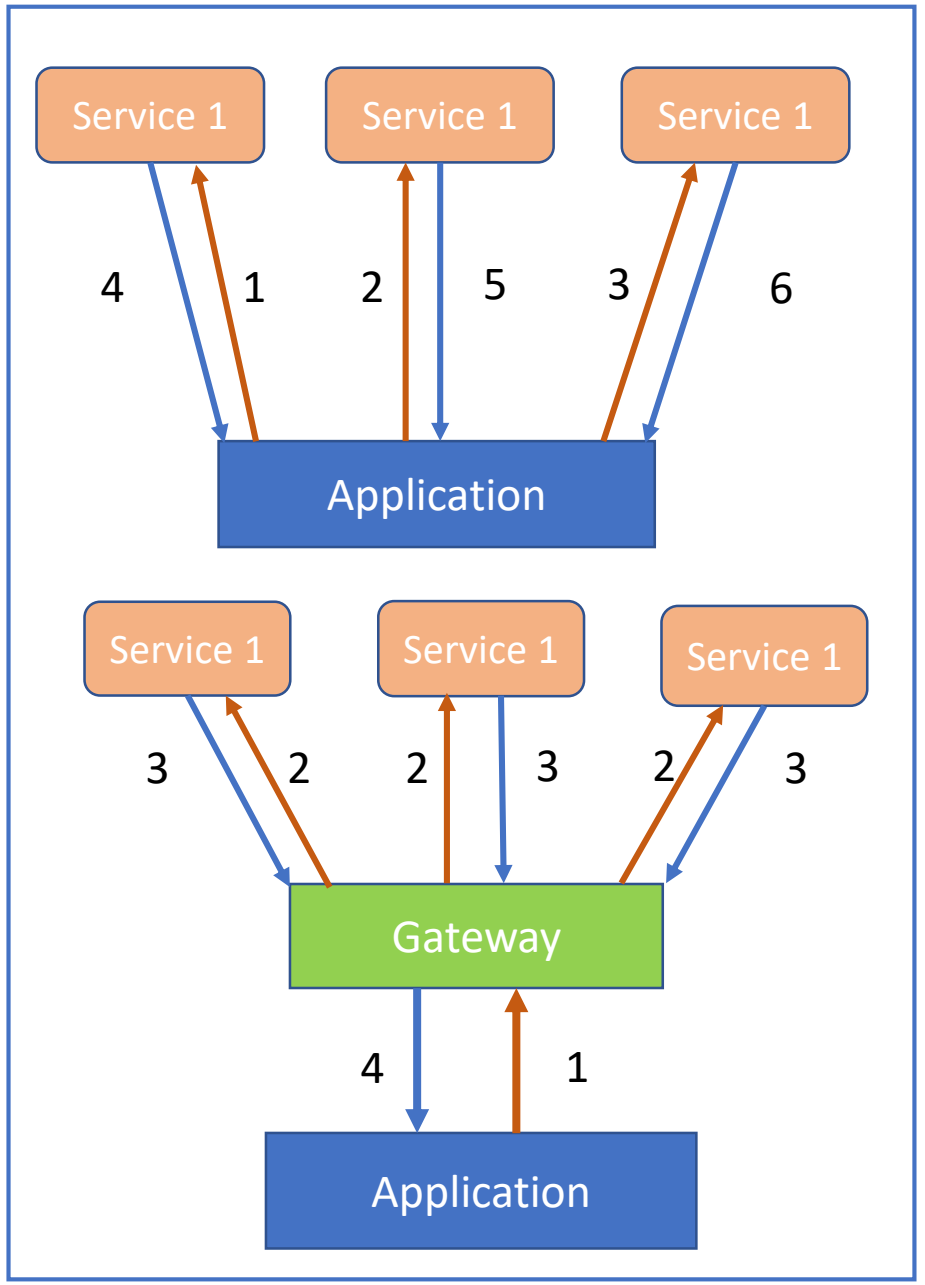
- To perform a single task, a client may have to make multiple calls to various backend services.
- When any new feature or service is added to the application, additional requests are needed, further increasing resource requirements and network calls.

Solution:

- Use a gateway to reduce chattiness between the client and the services.
- This pattern can reduce the number of requests that the application makes to backend services, and improve application performance over high-latency networks.
- To avoid client change if any new service added at backend system.

When to use this pattern:

- It can reduce the number of requests that the application makes to backend services.
- The client may use networks with significant latency, such as cellular networks.



Gateway Offloading pattern

Context and problem:

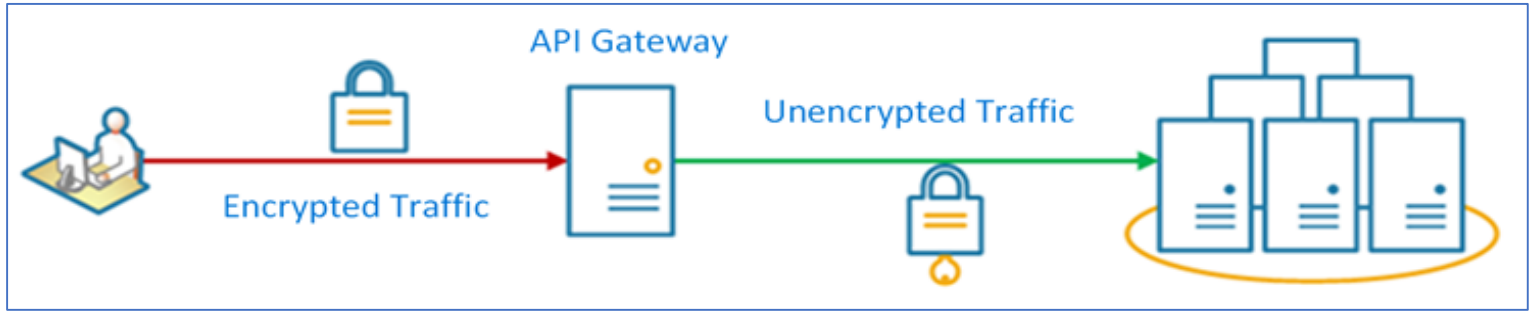
- Some features are commonly used across multiple services, and these features require configuration, management, and maintenance.
- Properly handling security issues and other complex tasks can require team members to have highly specialized skills.

Solution:

- Offload some features into an API gateway, particularly cross-cutting concerns such as certificate management, authentication, SSL termination, monitoring, protocol translation, or throttling.
- Properly handling security issues (token validation, encryption, SSL certificate management) and other complex tasks can require team members to have highly specialized skills.

When to use this pattern:

- It is useful to simplify the development of services by removing the need to distribute and maintain supporting resources, such as web server certificates and configuration for secure websites.
- Allowed dedicated teams to implement features that require specialized expertise, such as security.
- Provide some consistency for request and response logging and monitoring.



Gateway Routing pattern

Context and problem:

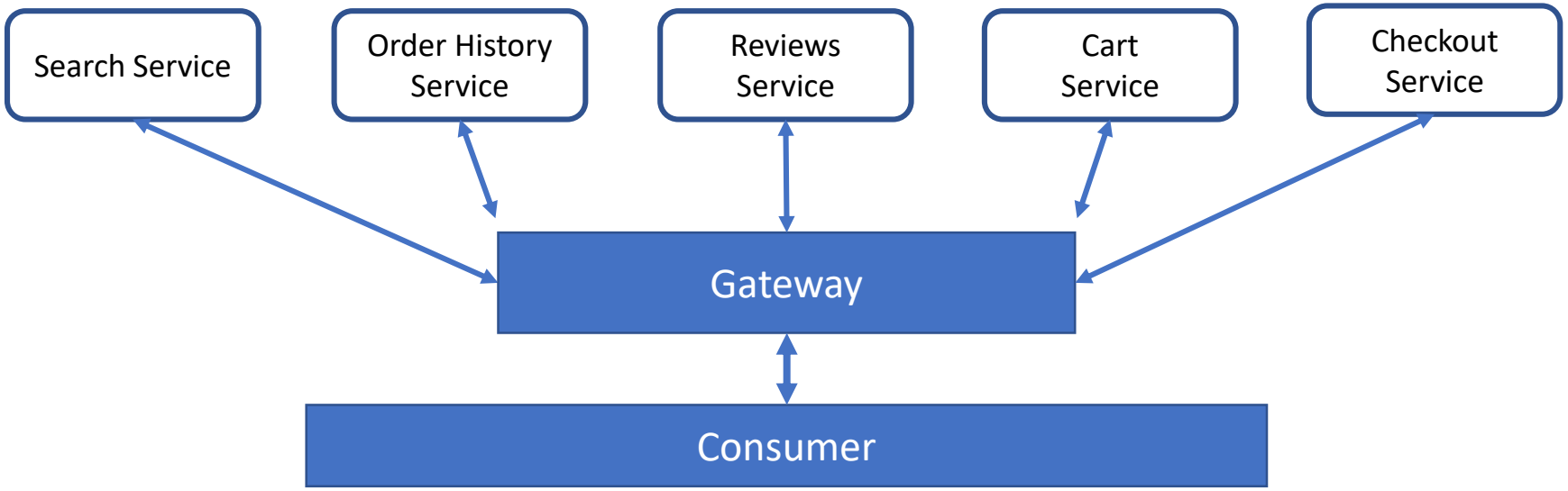
- When a client needs to consume multiple services, setting up a separate endpoint for each service and having the client manage each endpoint can be challenging.

Solution:

- Place a gateway in front of a set of applications, services, or deployments.
- A gateway also lets you abstract backend services from the clients, allowing you to keep client calls simple while enabling changes in the backend services.

When to use this pattern:

- A client needs to consume multiple services that can be accessed behind a gateway.
- You need to route requests from externally addressable endpoints to internal virtual endpoints, such as exposing ports on a VM to cluster virtual IP addresses.



Sidecar pattern

Context and problem:

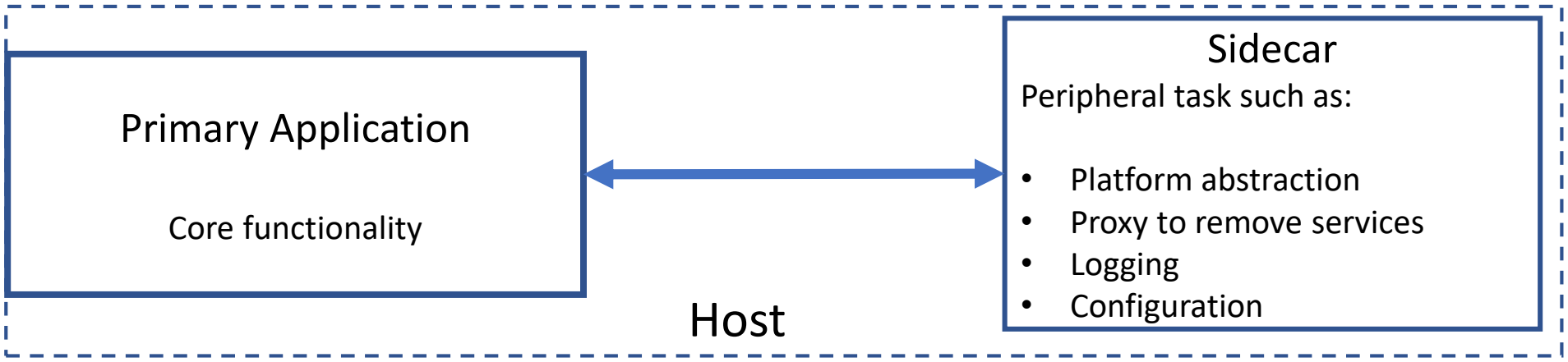
- Applications and services often require related functionality, such as monitoring, logging, configuration, and networking services. These peripheral tasks can be implemented as separate components or services.
- If they are tightly integrated into the application, they can run in the same process as the application, making efficient use of shared resources.
- If the application is decomposed into services, then each service can be built using different languages and technologies.

Solution:

- Co-locate a cohesive set of tasks with the primary application, but place them inside their own process or container, providing a homogeneous interface for platform services across languages.

When to Use this Pattern:

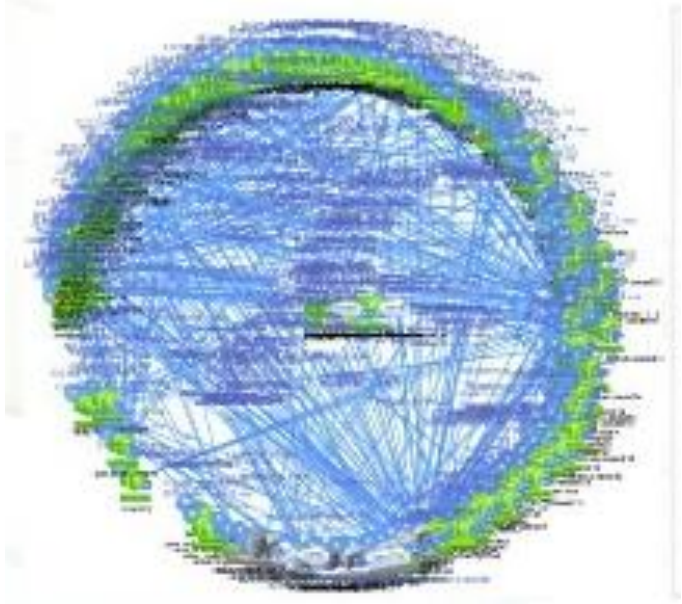
- A component is owned by a remote team or a different organization.
- A component or feature must be co-located on the same host as the application.



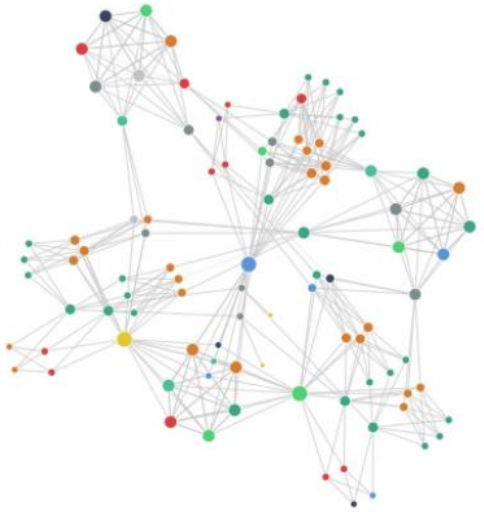
INTEGRATION PATTERNS

SOA Style

- Microservices can be treated as SOA services.
- RESTful interfaces have replaced SOAP as the protocol of choice for service definitions in the wider IT industry
- There are a couple of variants of this SOA style pattern aligning to fine-grained or course-gained services.
- Too fine-grained services leads to what is known as ‘Death Star Architectures’.
- More course-grained microservices leads to a more cohesive architecture as service boundaries are more carefully defined



‘Death Star’
Architecture



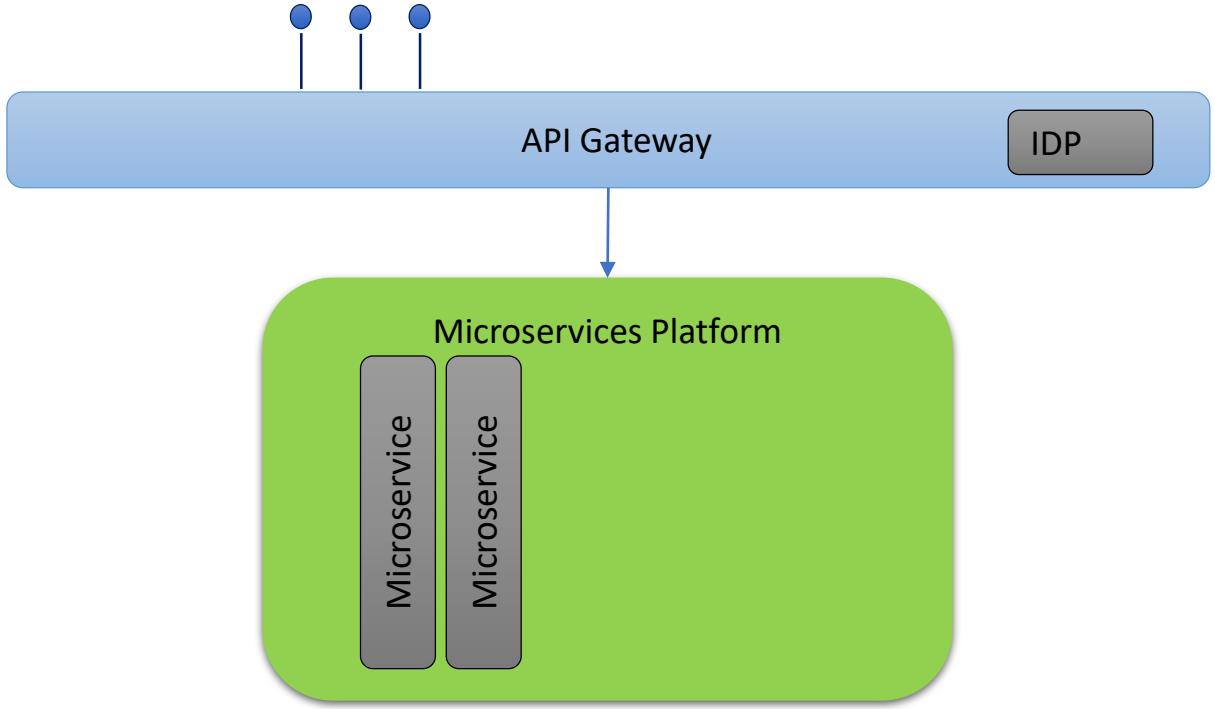
’Cohesive’ Architecture

Characteristics of Pattern

- Point-to- point calls between microservices

API Backing

- It is a common pattern to have APIs running in an API Gateway to be basic service proxies with responsibilities for authorization and security with microservices implementing business logic.

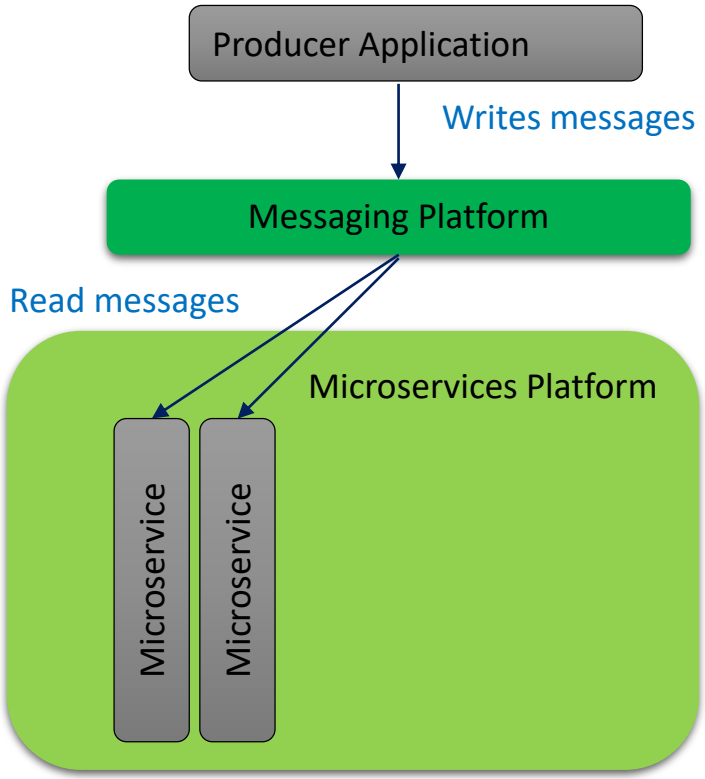


Characteristics of Pattern

- Microservices hidden behind API gateway

Message Orientated Microservices

- This pattern enables microservices to be able to be scaled through using the capabilities of messaging technology that allows multiple consumers to process messages with on a ‘first come, first served’ model.

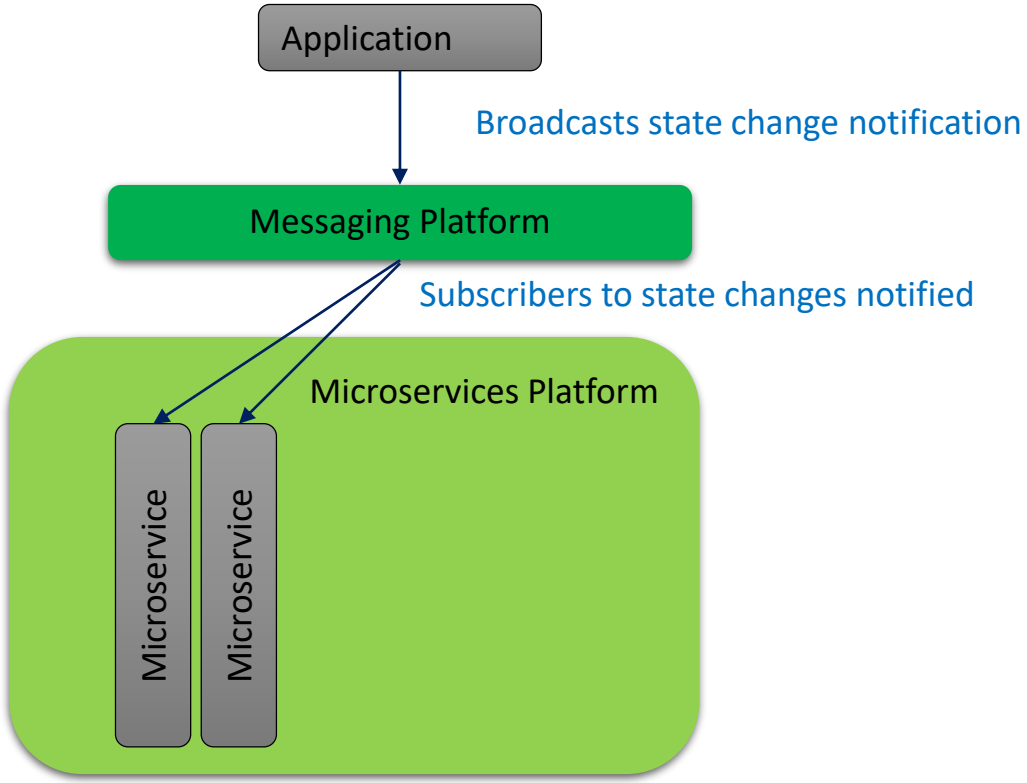


Characteristics of Pattern

- Microservices read messages from a queue

Event Driven

- This pattern enables loose coupling of system by emitting notification of state change.
- A application e.g. a SoR emits a notification of a business change. As an optimization this event may contain the business data (before & after) to enable interested microservices to not need to access the database directly.

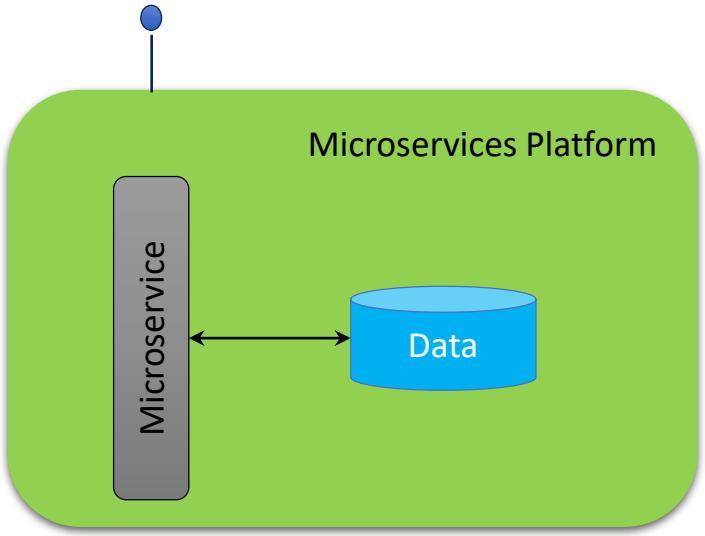


Characteristics of Pattern

- Microservices read messages from a queue containing system state changes

Isolated State

- Microservice can define an API and maintain their own state
- Whilst this is initially an easy pattern to implement as dependence on it and data volumes increase scalability becomes an issue

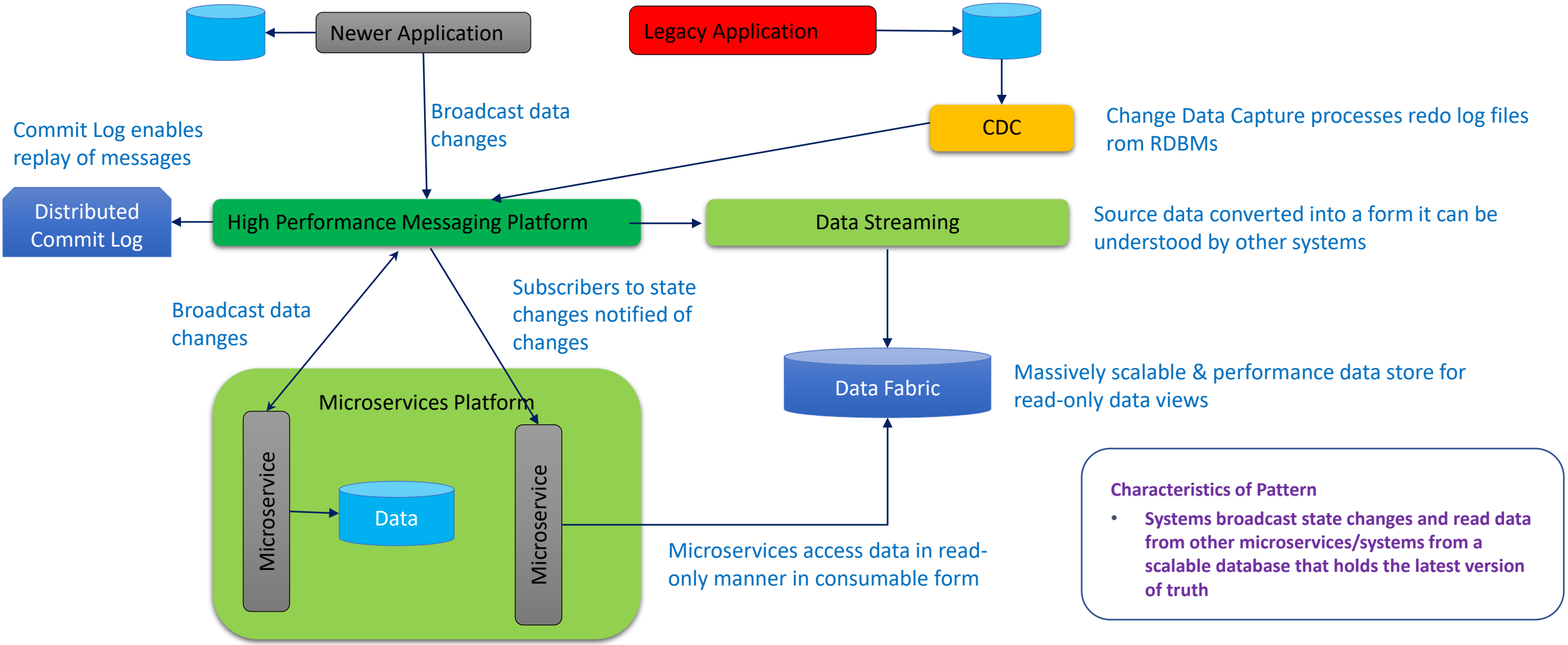


Characteristics of Pattern

- Microservices have a bounded context and hold their own data

Replicated State

- This pattern provides a scalable solution that enables the state of an application to be known at any point in time aka ‘Event Sourcing’



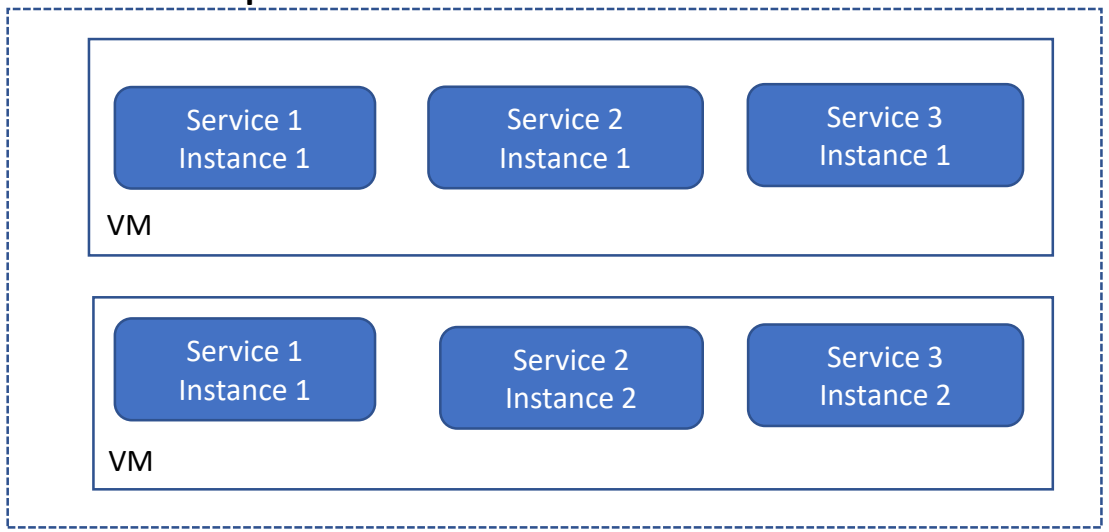
Characteristics of Pattern

- **Systems broadcast state changes and read data from other microservices/systems from a scalable database that holds the latest version of truth**

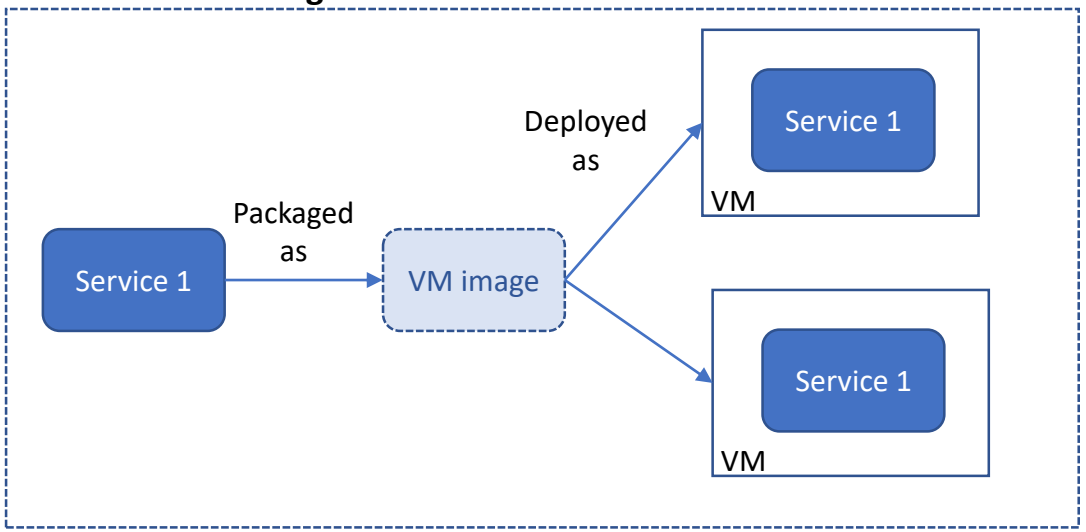
DEPLOYMENTS PATTERNS

Deployment patterns

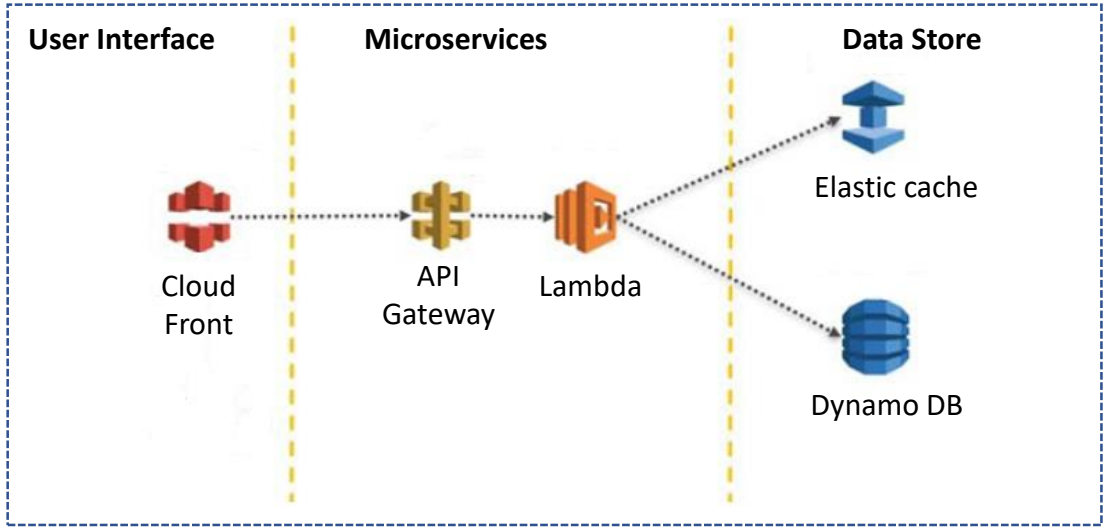
Multiple Service Instance Per VM



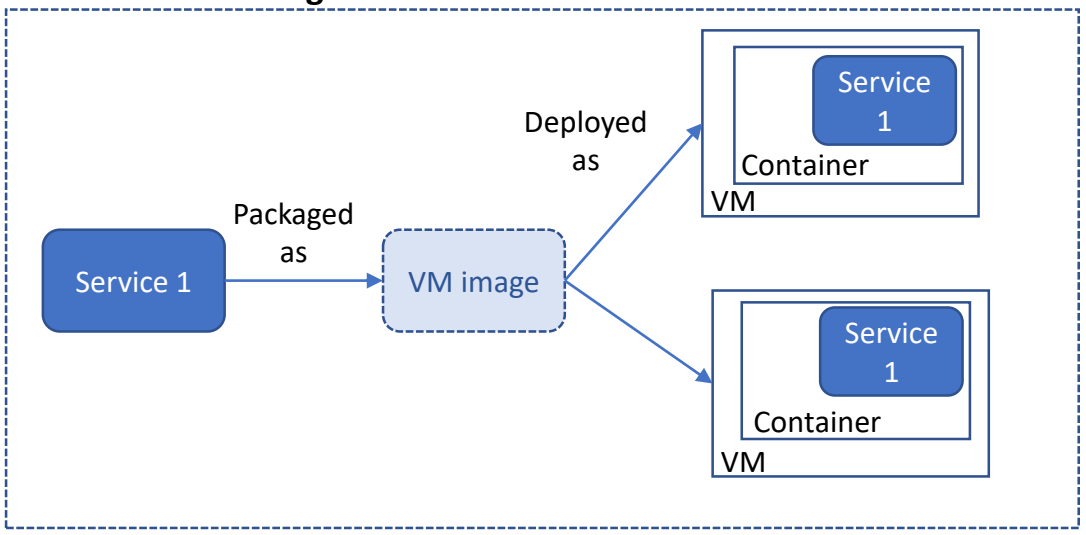
Single Service Instance Per VM



Server less Deployment



Single Service Instance Per Container



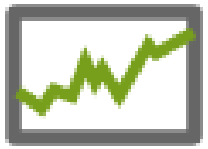
Microservices Deployment challenges

Availability

Management & Monitoring

Data management

Design & implementation



Performance & Scalability

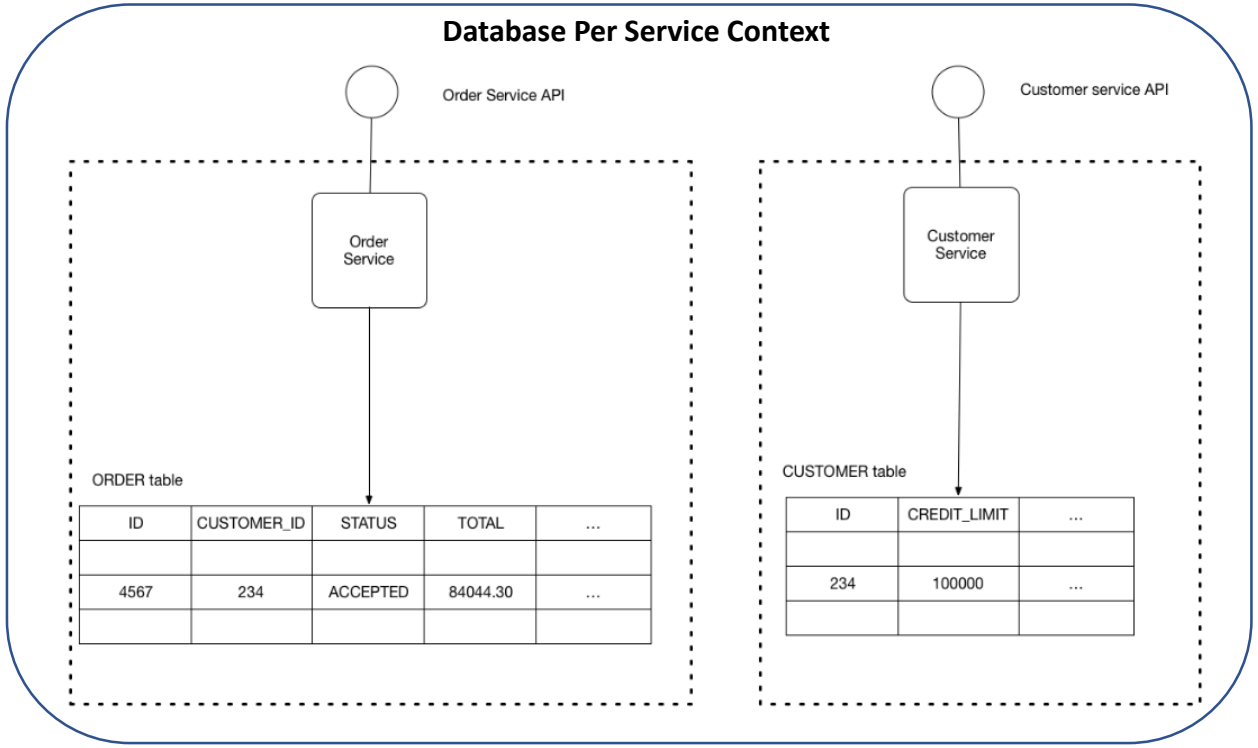
Resiliency

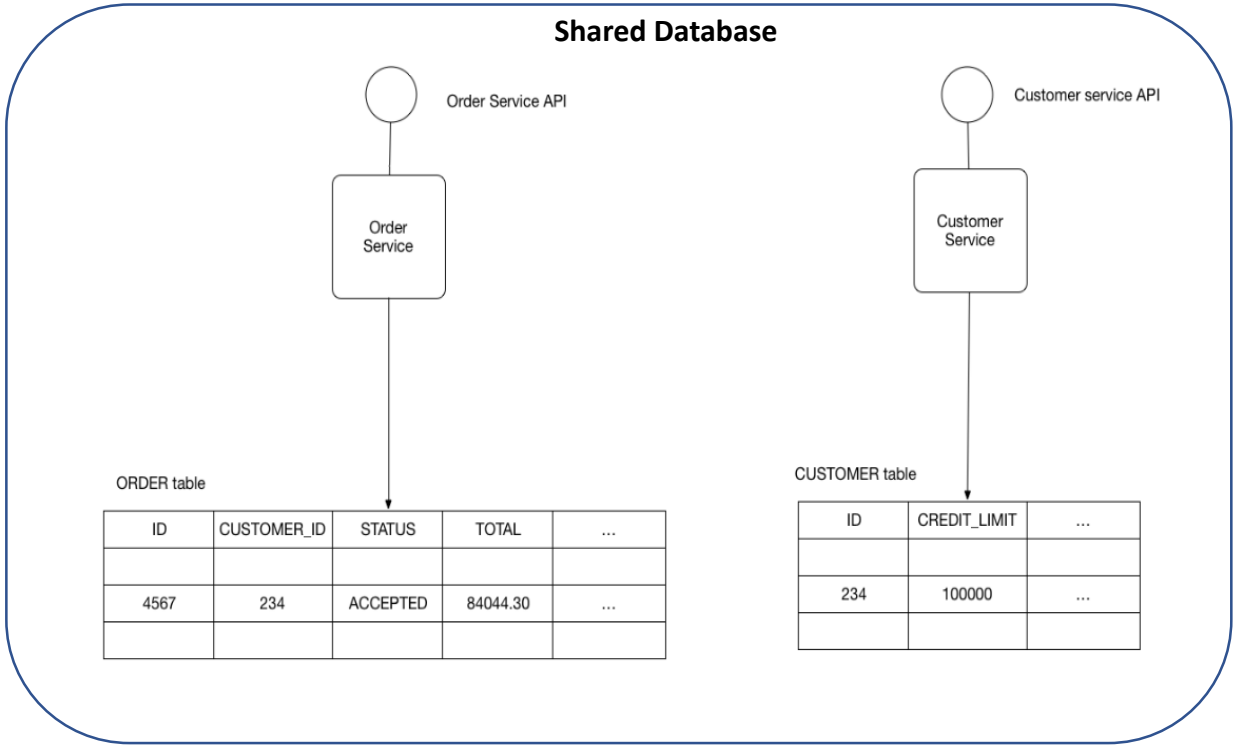
Messaging

Security

MICROSERVICES DATA MANAGEMENT

Data Management





Benefits

•Helps ensure that the services are loosely coupled. Changes to one service's database does not impact any other services.

•Each service can use the type of database that is best suited to its needs.

Drawbacks

•Implementing business transactions that span multiple services is not straightforward. Distributed transactions are best avoided because of the CAP theorem. Moreover, many modern (NoSQL) databases don't support them.

•Implementing queries that join data that is now in multiple databases is challenging.

Benefits

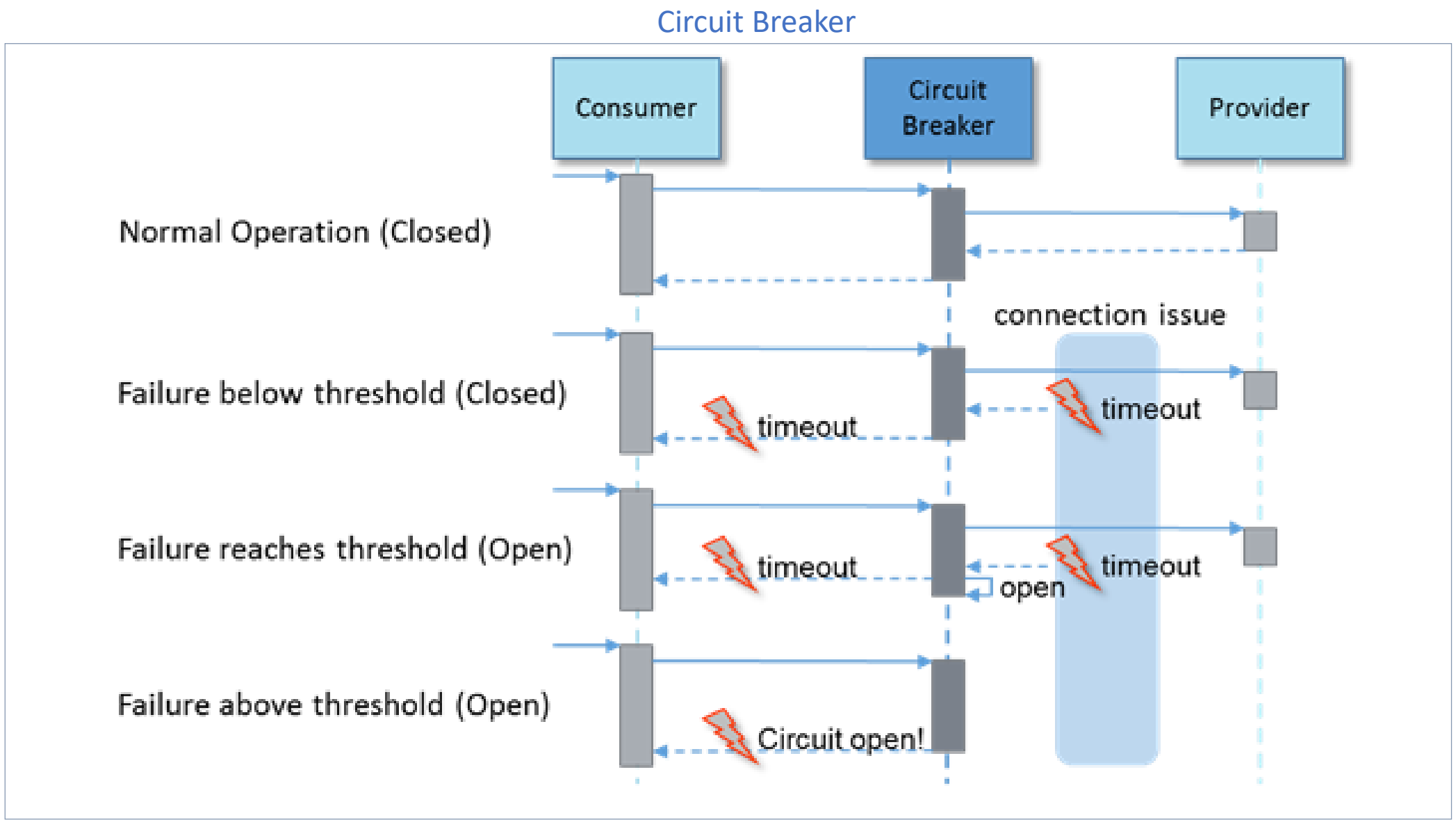
- A developer uses familiar and straightforward ACID transactions to enforce data consistency
- A single database is simpler to operate

Drawbacks

- Development time coupling.
- Runtime coupling .
- Single database might not satisfy the data storage and access requirements of all services

MICROSERVICES RESILIENCE

Reliability : Circuit Breaker



MICROSERVICES PROJECT STRATEGY

Project Strategy : Monolith First

Greenfield/Brownfield project :

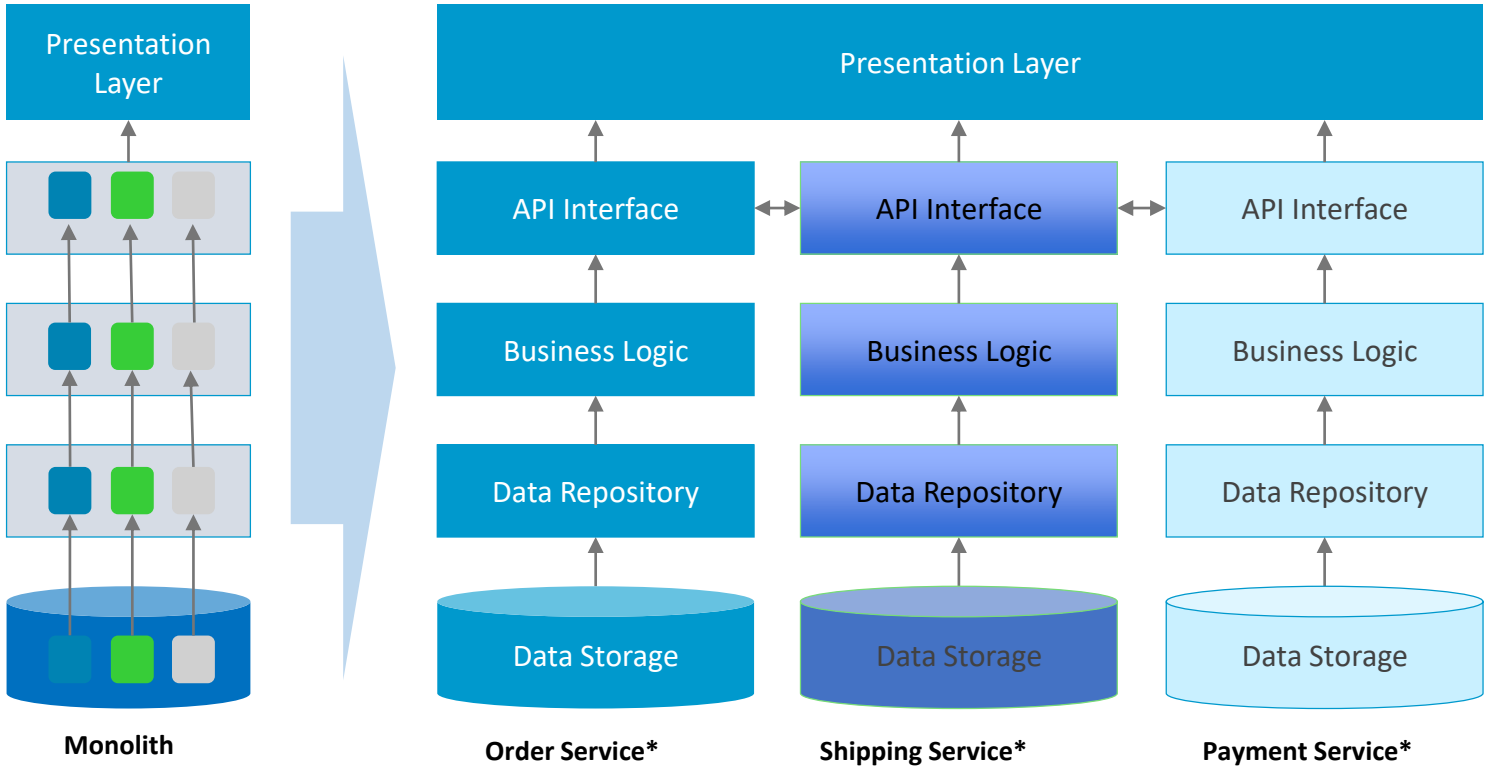
Most projects have properties that fit better with a monolith-first approach. Monolith architectures enable a faster start of developments and also simplify maintainability (which are most of the times the top goals at the beginning of a project).

However, if there is a clear vision that the project will require (eventually) a migration to a Microservices-based architecture, there are a few elements that can be taken into account while implementing the monolith:

- Create modular application around business capabilities
- Create database schemas per application modules

By developing a monolith with these properties, the migration of modules (and their supporting databases) to a Microservices model is simplified.

Another advantage of migrating “Brownfield” monolith to Microservices* is the fact that a lot of domain knowledge has been gathered, which simplifies the further identification and development of Microservices (when compared with a greenfield project, where teams have no experience with the domain).



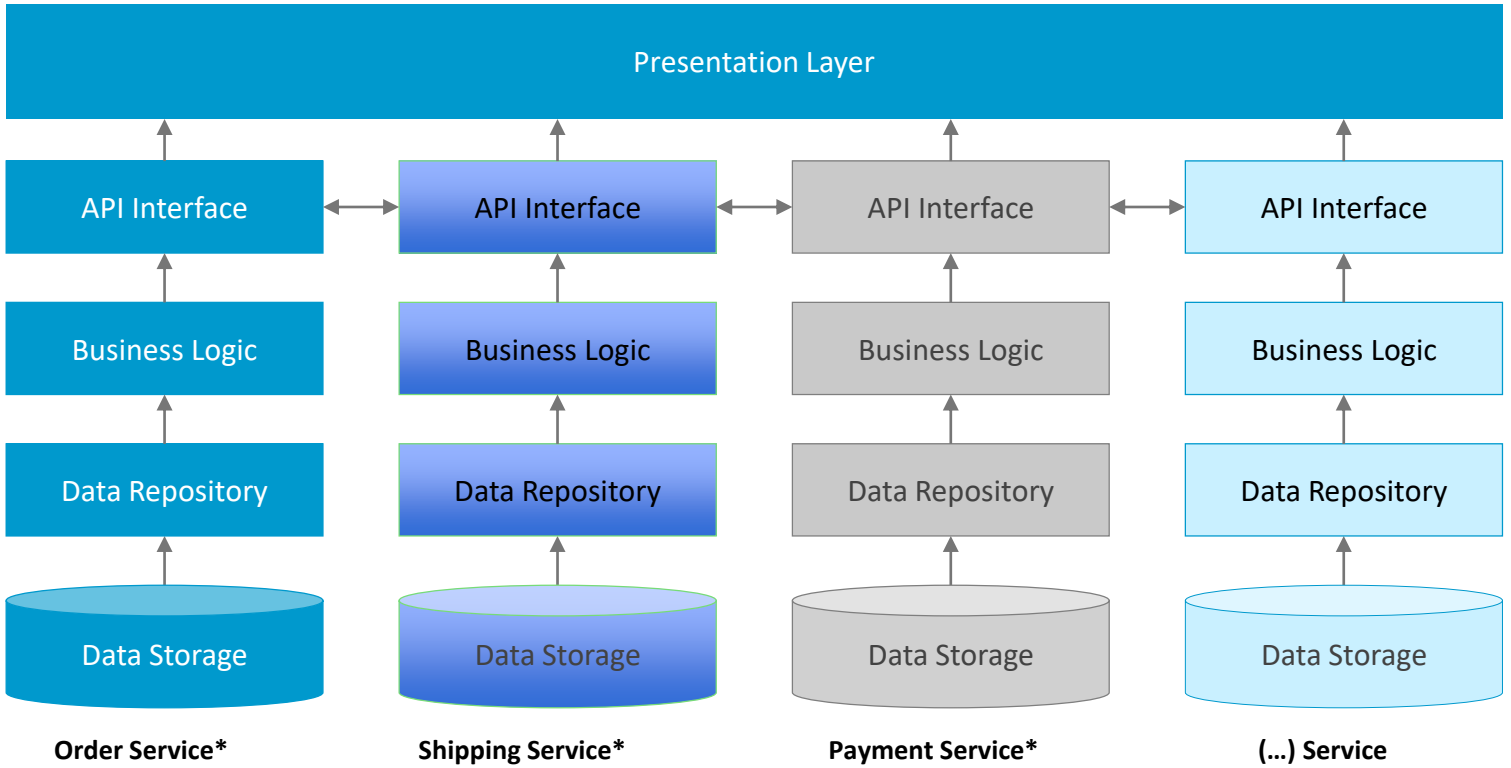
Project Strategy : Microservices First

Green Field Project :

There are situations where implementing a greenfield product as Microservices is the best strategy.

This approach is advised when it is known that the product being developed is solving a “complex” problem and requires different modules to be developed and evolved independently of each other (in an exploratory mode). The decision can be further simplified if it is know that the product will have scalability challenges and the company is willing to address them from the start of the project (and not create technical debt).

In these cases it may be the best strategy to pay the overhead of starting the project with a Microservices Architecture.



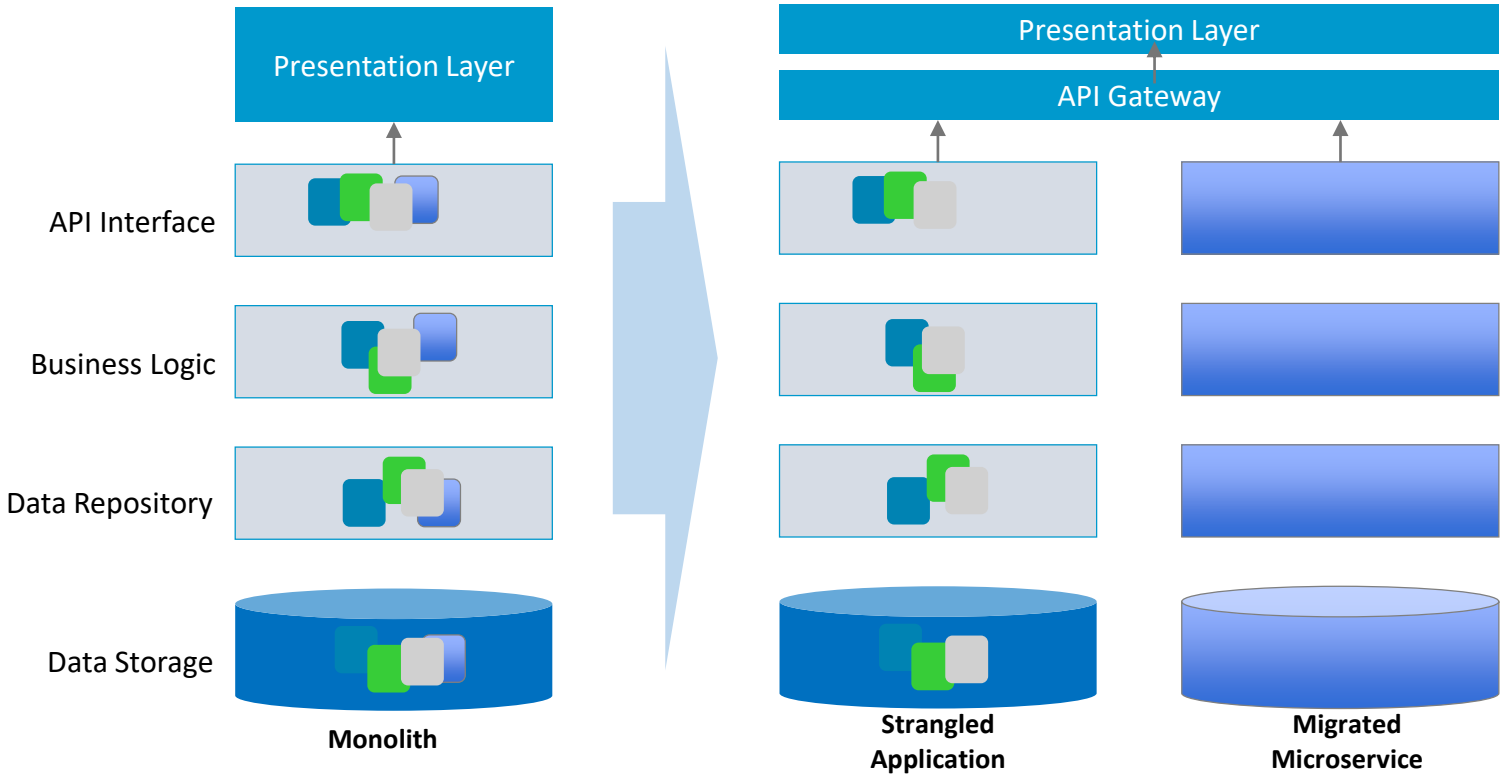
Project Strategy : Strangler Pattern

In this pattern, there is an application in place, which delivers a specific functionality (normally as a Monolith), however this application cannot deliver the desired scalability or speed of development required for the product.

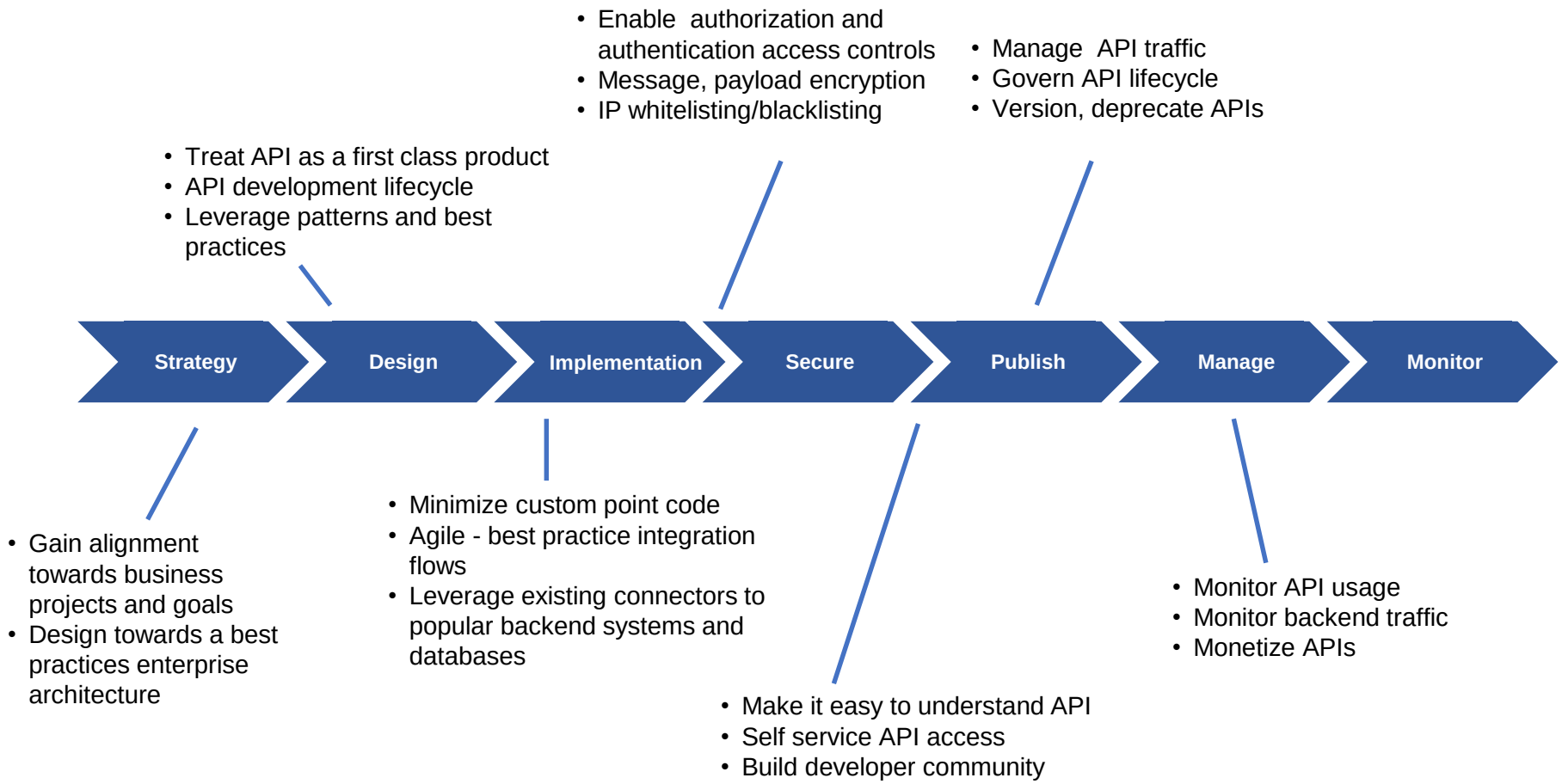
In those cases – and because this application most of the times was not designed from start with the goal of being eventually migrated to a Microservices-based architecture – the Strangler Pattern can be used to gradually migrate the existing monolith into a Microservices Application.

Most important properties:

- Identify modules within the monolith application
- Isolate them and place a API Gateway or Load Balancer in front of the application to redirect traffic for that specific functionality to the proper "migrated" Microservice
- This process is repeated until the Monolith Application is completely migrated – or it remains to handle legacy functionality that cannot (or has no advantage) being migrated to a Microservice

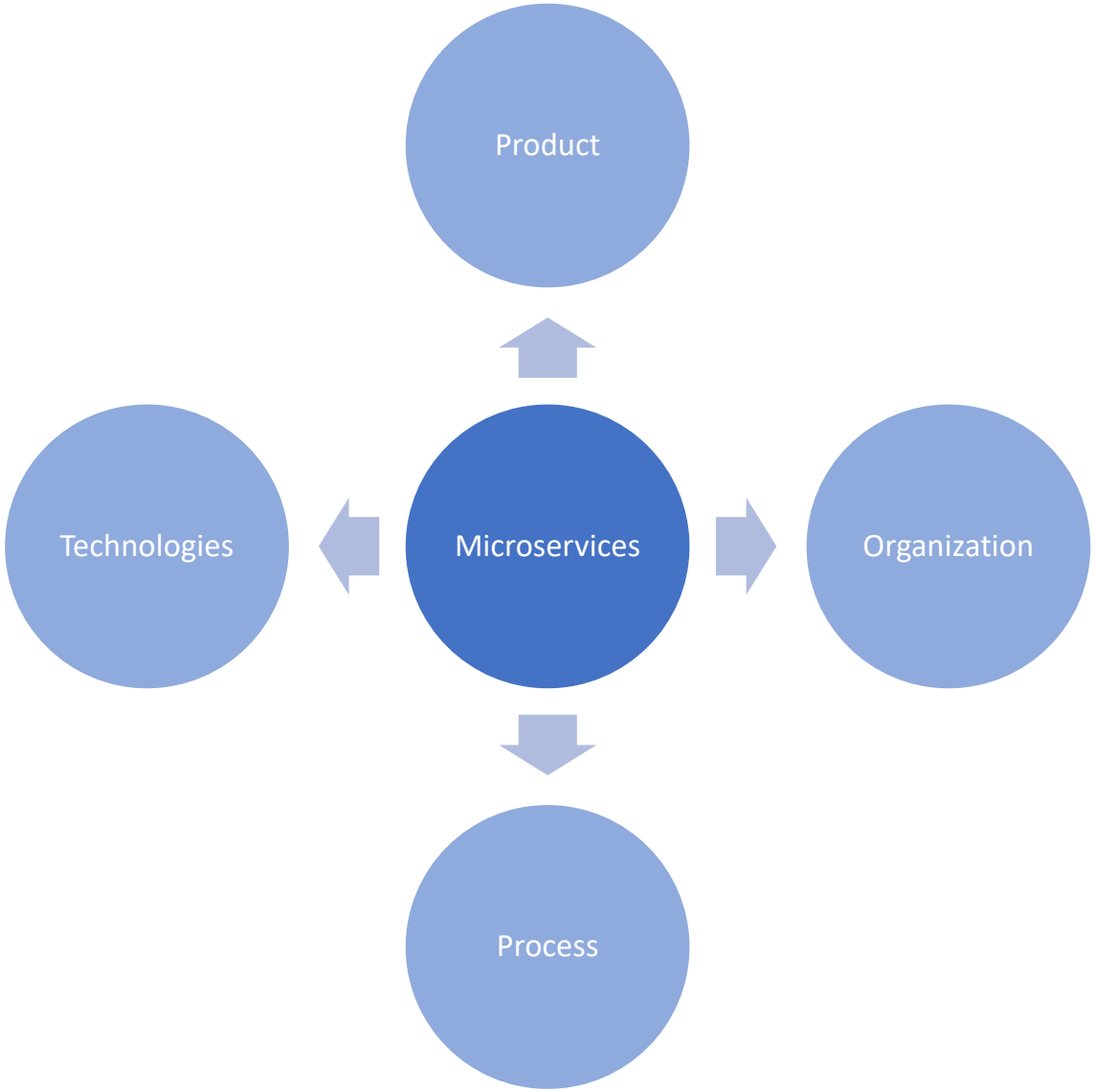


Project Strategy : Comparisons and Evaluations



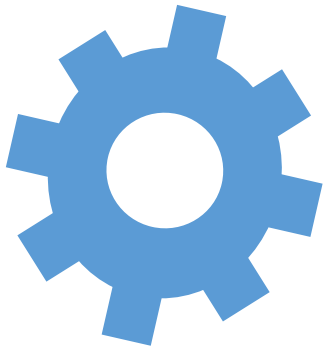
MICROSERVICES ADOPTION CONCERNS

Microservices Implementation Concerns



Product Complexity and Roadmap

Before starting using Microservices, the very first step is to verify whether the product's complexity and roadmap will benefit from Microservices architectural style (and justifies increased overhead of a Microservices architecture).

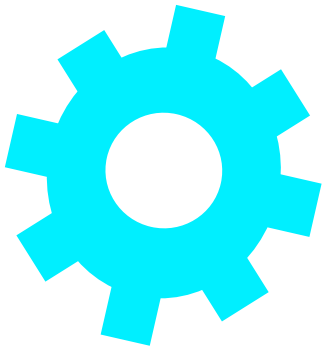


Product

Shall we adopt the Microservices Architecture style for this Product?	
Yes, if:	**No, if:**
• Existing product is an "evolving monolith", with many teams working on it and with an increasingly slower (insufficient) speed to deliver new features. • New product, resolving a "complex"* problem with a large scope (unpredictability and "emergence" on development of each component, requires constant experimentation to define next developments – i.e.: benefits from decoupled development). • Product has high scalability requirements (or issues), which cannot be solved by design/infra updates.	• New product, resolving a "obvious"* or "complicated"* problem with small scope (predictable next steps and actions for product development). • No clear vision of the product roadmap and requiring quick time-to-market (validation). • Product has scalability issues, which can be addressed by design/infra updates.

Organizational structure and maturity

The aim of a Microservices architecture is to enable the decomposition of the application domain into small independently developed and deployable components. In order to fully benefit from that, organizations must adapt their structure and mature on several key areas.



Organizational

Reverse Conway's Law

- Traditionally organizations are build around "functional teams" (e.g.: Development, Test, Operations, etc.), known as Conway's law*. This constrains architecture and processes used when creating a product.
- This is a limiting factor for Microservices architecture as it aims at enabling each team to fully develop and manage their Microservices.
- In order to address that, successful Microservices initiatives are "reversing" Conway's Law, i.e.: architecture and processes drive the organization structure. This is being popularized by the move from "functional teams" to "cross-functional teams.

Trust from Management

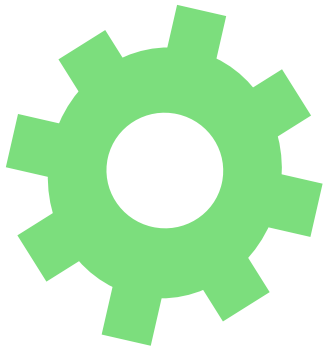
- Changes on the organizational structure can lead to big implications in terms of management, e.g.: higher cost during transition periods.
- However, in order to successfully implement Microservices architectures management must endorse these changes on structure.
- If these changes are not accomplished, the result will be a partial implementation of a Microservices architecture, which will lead to sub-optimal results, if not failure of the initiative.

Team Maturity

- From different studies of early adopters of Microservices, one can observe that successful Microservice initiatives come from "empowering teams" and create conditions for them to continuously improve and grow.
- Teams doing Microservices must have certain levels of maturity on several key areas: DDD for service design; Continuous Integration and Delivery; Automation; Cloud Native Applications patterns - which are fundamentally different; etc..
- It is important for teams to define Maturity Models** to address the essential areas to build Microservices architectures, so that they can consistently build their "evolving architecture", and avoid pitfalls.

Processes for delivery and operations

When developing Microservices-based applications the aspect of development and operations is fundamental. Dev and Ops must be addressed by the team responsible by each Microservice.



Process

Design and Delivery

- Agile methodologies are very popular nowadays to enable faster delivery of software. Microservices provides extra agility for the different teams to plan, develop, test and deploy new Microservices with higher independence.
- It is fundamental having mechanisms to govern design of Microservices (e.g.: identify the Microservices and their inter-dependencies).
- This means that delivery processes must have some cross-team efforts to design and define scope of the different Microservices (e.g.: having DDD and Event Storming workshops; have clear architecture view of the system, i.e.: design before build).

DevOps

- In a Microservices architecture, each team developing a particular microservice must also operate and maintain it.
- This means that teams must be composed of members that will do both development and operations – Dev and Ops.
- The actual composition of the team will also depend on the type of infrastructure used for the Microservices architecture (e.g.: some organizations starting using Serverless technologies, which minimizes the need of classical operations efforts).
- However, even in those organizations, it is fundamental to have designated team members focusing on delivery pipelines, monitoring, etc.

Architecture

- Before starting a microservices-based application, it is essential to have a clear view of the architecture adopted.
- In many “agile organizations” this is minimized or even ignored, but in the case of Microservices it can lead to complications – e.g.: no clear view on basic components of the architecture.
- The actual format of the architecture team/activities may vary, e.g.: some organizations have dedicated cross-team architects, others prefer to define guilds, which involve members of the different teams. Independent of the format, it is essential to invest on clear vision and guiding principles.

Technology Ecosystem

Microservices architecture can bring a lot of advantages. However, these benefits come at the cost of much more complex systems and moving parts. Given this, and before starting a Microservices initiative, it is essential to have several technology elements in place.

Technology

Rapid Application Deployment

- Must have automation of processes in place, so new services can be quickly deployed and tested in the different environments (dev, test, production) - fundamental, as the number of services can quickly grow (easily reach hundreds).
- Provide "deployment pipelines" that support "repeatable" processes for the different Microservices.
- Enable easy/automated management of deployment, rollback etc.

Techniques and Tools

- Continuous Integration (CI)
- Continuous Delivery (CD)

Rapid Provisioning

- Have a flexible infrastructure, where new resources can be provisioned quickly.
- Provisioning of new environments must be performed automatically, i.e.: have repeatable processes to accomplish common activities (e.g.: provision new servers or databases).

Techniques and Tools

- Cloud Computing
- Containers

Basic Monitoring

- Microservices architecture will have many distributed moving parts. To understand and manage it proper monitoring must be in place.
- Monitoring must be aggregated and provide an overview on the health of the different parts of the architecture, so actions to address issues can be taken.
- Elements to monitor: errors, services availability, latency, usage, etc.

Techniques and Tools

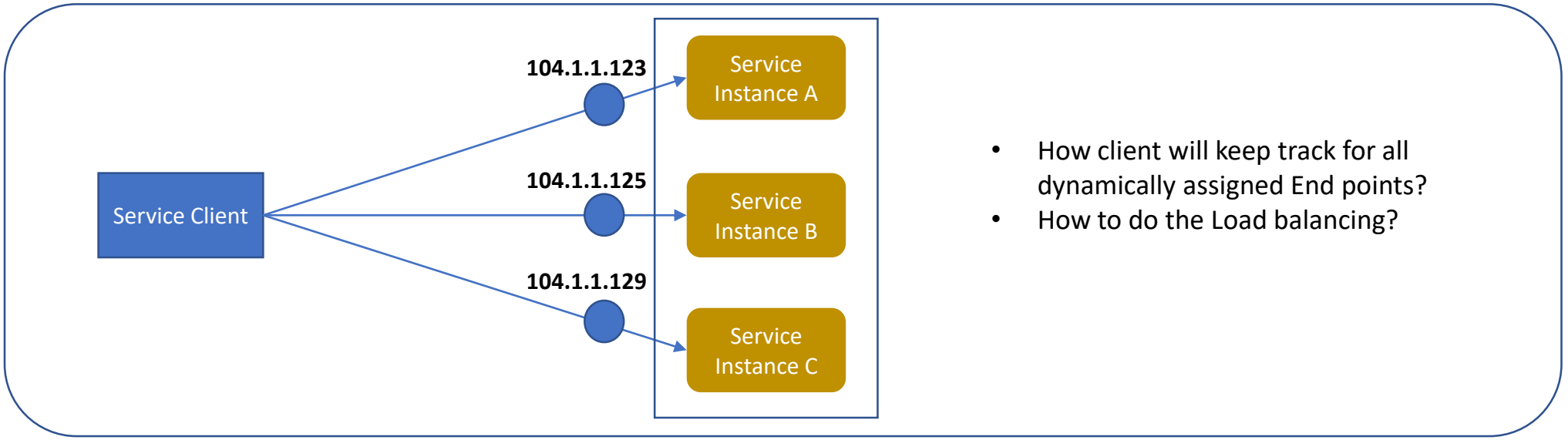
- Infrastructure Monitoring (e.g. ELK)
- Log Management (e.g. Splunk)

MICROSERVICES DISCOVERY

Microservice Discovery

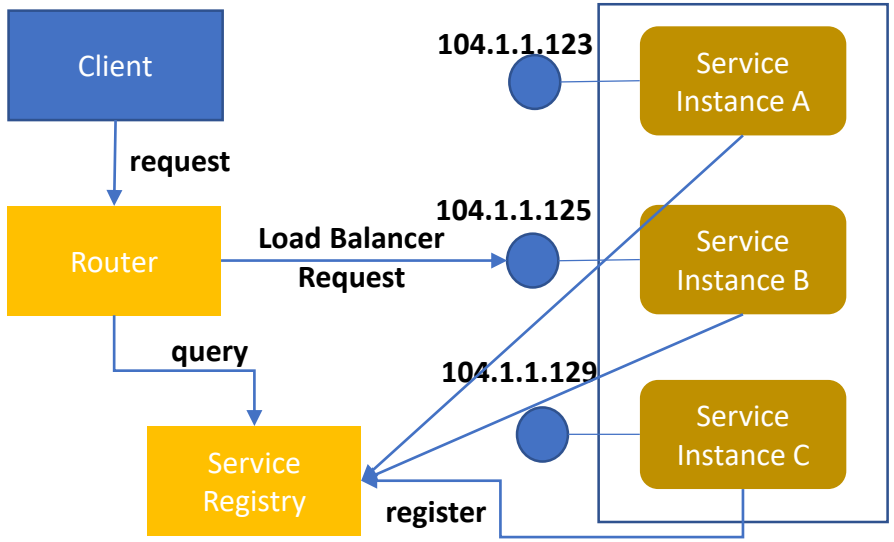
Real Project Scenarios

- Each instance of a service exposes a remote API such as 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. For example, an EC2 Autoscaling Group adjusts the number of instances based on load.



- How client will keep track for all dynamically assigned End points?
- How to do the Load balancing?

Client Side & Server Side Discovery



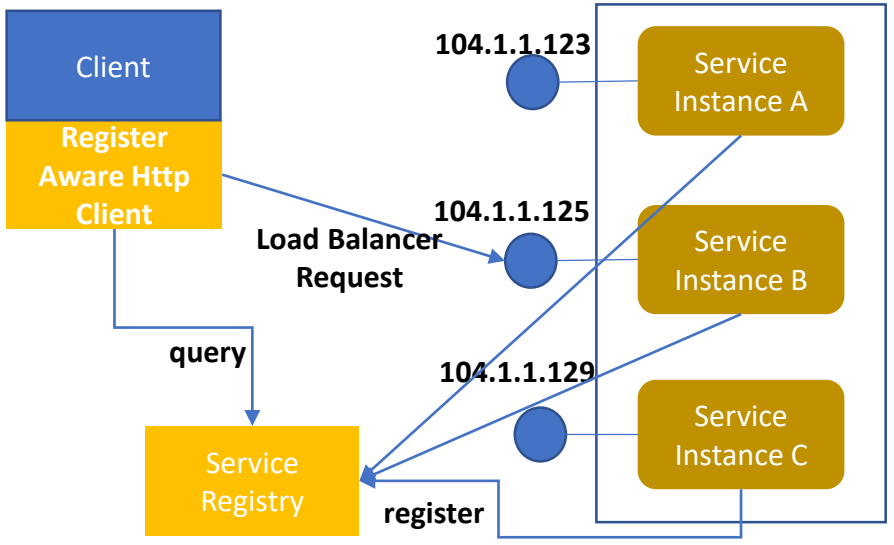
Server Side Discovery

Benefits -

- Compared to client-side discovery, the client code is simpler since it does not have to deal with discovery. Instead, a client simply makes a request to the router
- Some cloud environments provide this functionality, e.g. AWS Elastic Load Balancer

Drawbacks -

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



Client Side Discovery

Benefits -

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

Drawbacks -

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

MICROSERVICES TESTING

Testing a Microservices Architecture

By introducing (sometimes) hundreds of services that can be released multiple times a day, testing complexity increases considerably. Modern testing strategies are leveraged with some considerations due to the nature of Microservices. The primary goal of these new considerations is to enable testing without influencing the ability of releasing the software multiple times a day.

On the next slides, we will give an overview on considerations for:

- Unit Testing
- Integration Testing (Persistence, Gateway)
- Component Testing
- Contract Testing
- End-to-end Testing

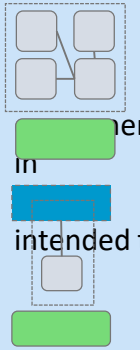
Testing strategies

Traditional testing strategies(Unit Testing, Integration Testing and Component Testing) apply naturally to a Microservices Architecture. In the scope of Microservices, we want to highlight.

Unit Testing

A **unit test** exercises the smallest piece of testable software in the application to determine whether it behaves as expected. It typically written at the class level, or around a related group of classes. The smaller the unit under test, the easier it is to express the behavior using a unit test since the branch complexity of the unit is lower. Often, the difficulty in writing a unit test can highlight when a module should be **broken down** into **independent** more **coherent** pieces and tested individually. Thus, alongside being a useful testing strategy, unit testing is also a **powerful design tool**, especially when combined with test driven development.

Integration Testing



An **integration test** verifies the communication paths and interactions between ents in to detect interface defects and order to verify that they collaborate as intended to achieve some larger piece of behavior. This is in contrast to unit tests where, even if using real collaborators, the goal is to closely test the behavior of the unit under test, not the entire subsystem. Whilst tests that integrate components or modules can be written at any granularity, in microservice architectures they are typically used to verify interactions between layers of integration code and the external components to which they are integrating.

Component Testing

A **component test** limits the scope of the exercised software to a portion of the system under test, manipulating the system through internal code interfaces and using test doubles to isolate the code under test from other components. Testing such components in isolation provides a number of benefits. By limiting the scope to a single component, it is possible to thoroughly acceptance test the behavior encapsulated by that component whilst maintaining tests that execute more quickly than broad stack equivalents.

Isolating the component from its peers using test doubles avoids any complex behavior they may have. It also helps to provide a controlled testing environment for the component, triggering any applicable error cases in a repeatable manner.

In a microservice architecture, the components are the services themselves. By writing tests at this granularity, the contract of the API is driven through tests from the perspective of a consumer.

Testing strategies – Contract Testing and End-to-end Testing

Traditional testing strategies apply naturally to a Microservices Architecture. In the scope of Microservices, we want to highlight:

Contract Testing

An **contract test** is a test at the boundary of an external service verifying that it meets the contract expected by a consuming service. Whenever some consumer couples to the interface of a component to make use of its behavior, a contract is formed between them. This contract consists of expectations of input and output data structures, side effects and performance and concurrency characteristics.

Each consumer of the component forms a different contract based on its requirements. If the component is subject to change over time, it is important that the contracts of each of the consumers continue to be satisfied.

Integration contract tests provide a mechanism to explicitly verify that a component meets a contract.

When the components involved are Microservices, the interface is the public API exposed by each service. The maintainers of each consuming service write an independent test suite that verifies only those aspects of the producing service that are in use. These tests are not component tests. They do not test the behavior of the service deeply but that the inputs and outputs of service calls contain required attributes and that response latency and throughput are within acceptable limits.

End-to-end Testing

An **end-to-end test** verifies that a system meets external requirements and achieves its goals, testing the entire system, from end to end. In contrast to other types of test, the intention with end-to-end tests is to verify that the system as a whole meets business goals irrespective of the component architecture in use. In order to achieve this, the system is treated as a black box and the tests exercise as much of the fully deployed system as possible, manipulating it through public interfaces such as GUIs and service APIs. Since end-to-end tests are more business facing, they often utilize business readable DSLs which express test cases in the language of the domain. As a microservice architecture includes more moving parts for the same behavior, end-to-end tests provide value by adding coverage of the gaps between the services. This gives additional confidence in the correctness of messages passing between the services but also ensures any extra network infrastructure such as firewalls, proxies or load-balancers is correctly configured. End-to-end tests also allow a microservice architecture to evolve over time. As more is learnt about the problem domain, services are likely to split or merge and end-to-end tests give confidence that the business functions provided by the system remain intact during such large scale architectural refactoring.

End to End Testing Considerations

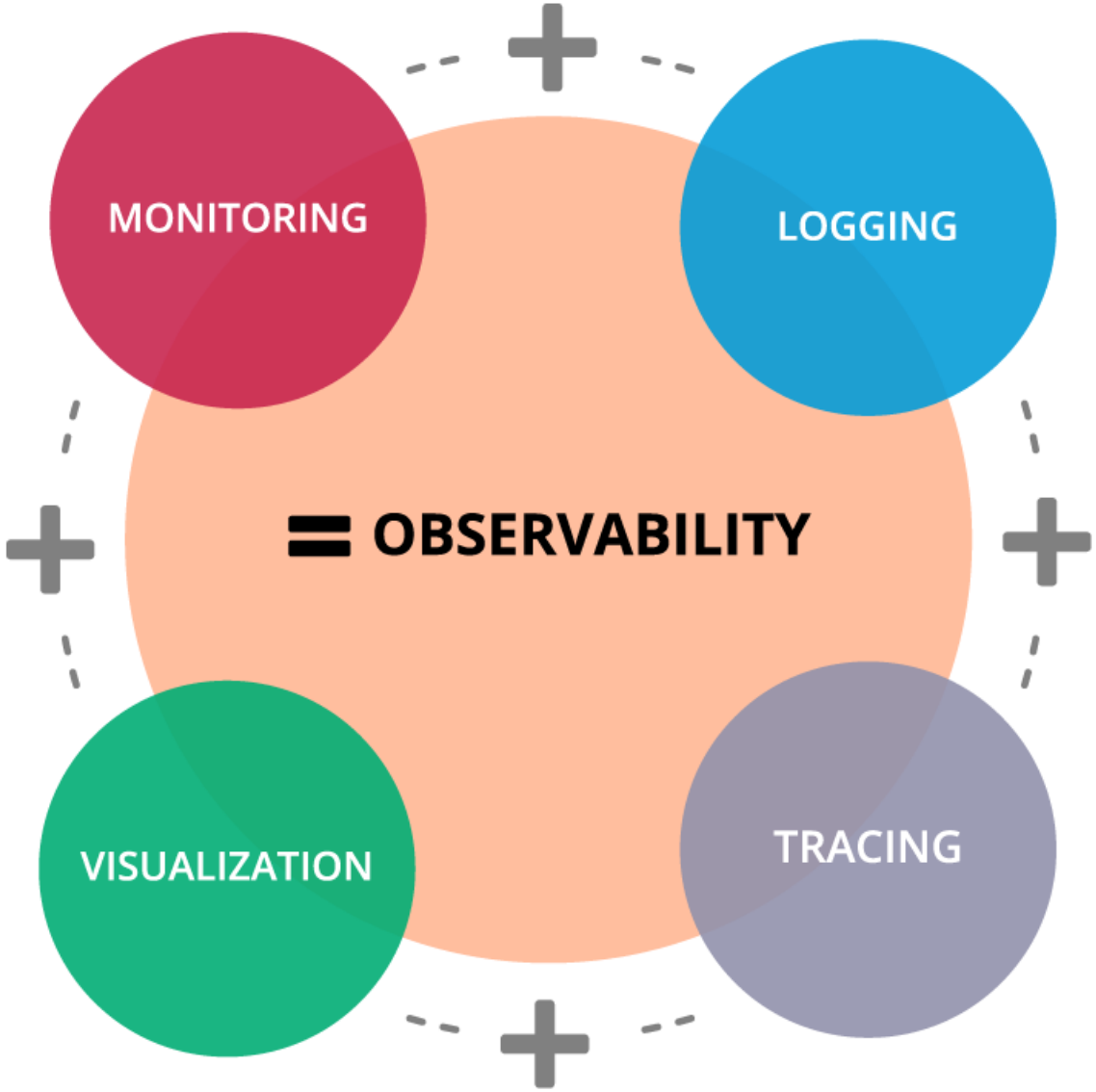
Since end-to-end tests involve more moving parts than the other strategies discussed so far, they in return have more reasons to fail.

End-to-end tests may also have to account for asynchrony in the system, whether in the GUI or due to asynchronous backend processes between the services. These factors can result in flakiness, excessive test runtime and additional cost of maintenance of the test suite. The following guidelines can help to manage the additional complexity of end-to-end tests:

- **Write as few end-to-end tests as possible**
- **Focus on personas and user journeys**
- **Choose your ends wisely**
- **Rely on infrastructure-as-code for repeatability**
- **Make tests data-independent**

MICROSERVICES OBSERVABILITY

Observability



Observability: Logging and Visualization

Context

Application consists of multiple services and service instances that are running on multiple machines. Requests often span multiple service instances. There will be issue while logs analysis due to –

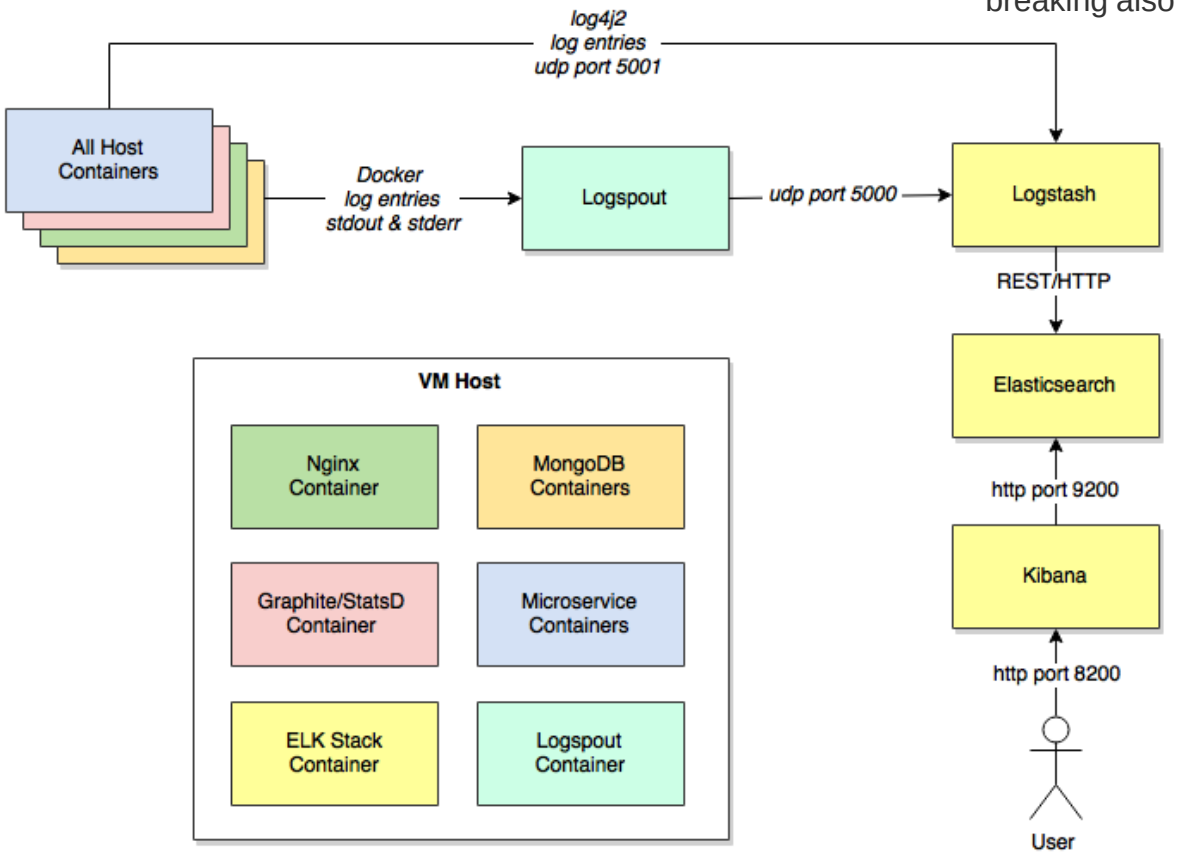
- Multiple Standard way
- Multiple Format Log messages
- Distributed Log Files

Problem

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

Solution

- Use a centralized logging service that aggregates logs from each service instance.
- Best practice for writing logs
  - Each log you are writing must have some information about the request.
  - If a request from a user/application further going to multiple systems than every system should use a common keyword or correlation-Id to trace it in case of any failures.
  - Sending an email through logger if connectivity is breaking also a good option.



Observability: Monitoring

Key aspect of Monitoring Application

Connection Setup Time: It is the amount of time it takes for the TCP three-way handshake to complete. When this metric spikes, it can be an indication of network slowness or an increase in the processing time within the TCP stack of a server. This metric is best monitored with a deviation from normal mindset.

Server Reset Rate: It is a per-second counter that increments as TCP resets are sent by the server. A TCP reset is a tool used by the TCP/IP stack in network devices to immediately close a connection. Technically, a TCP RST should indicate some type of error condition in the TCP stack.

Application Response Time: It is calculated as the time it takes for a server to respond to a data request from a client with a non-zero payload packet. The idea behind this metric is to measure the time it takes for a server to respond to a data request with application data. This metric will give us an idea of how quickly the application is responding to requests. When the response time of this metric increases, the implication is that the application is slowing.

Retransmission Rate: The amount of time taken by a server to respond to a packet sent by the client. Network RTT (Round Trip Time) is a great indicator of the health of the network, as well as the health and response time of the TCP/IP stack of your server. It is calculated as the amount of time, typically in milliseconds, that it takes for a server to respond to a packet sent by a client. This response time can be measured at either the TCP handshake, or continually throughout the capture by measuring the response times of empty ACK responses.

Network traffic monitoring provides great insight into the performance of your applications. While these metrics should give a good starting point, each application should be individually reviewed and a customized monitoring strategy should be developed.

Observability: Distributed tracing

Context

Requests often span multiple services. Each service handles a request by performing one or more operations, e.g. database queries, publishes messages, etc. Distributing tracing is increasingly seen as an essential component for observing distributed systems and microservice applications.

Problem

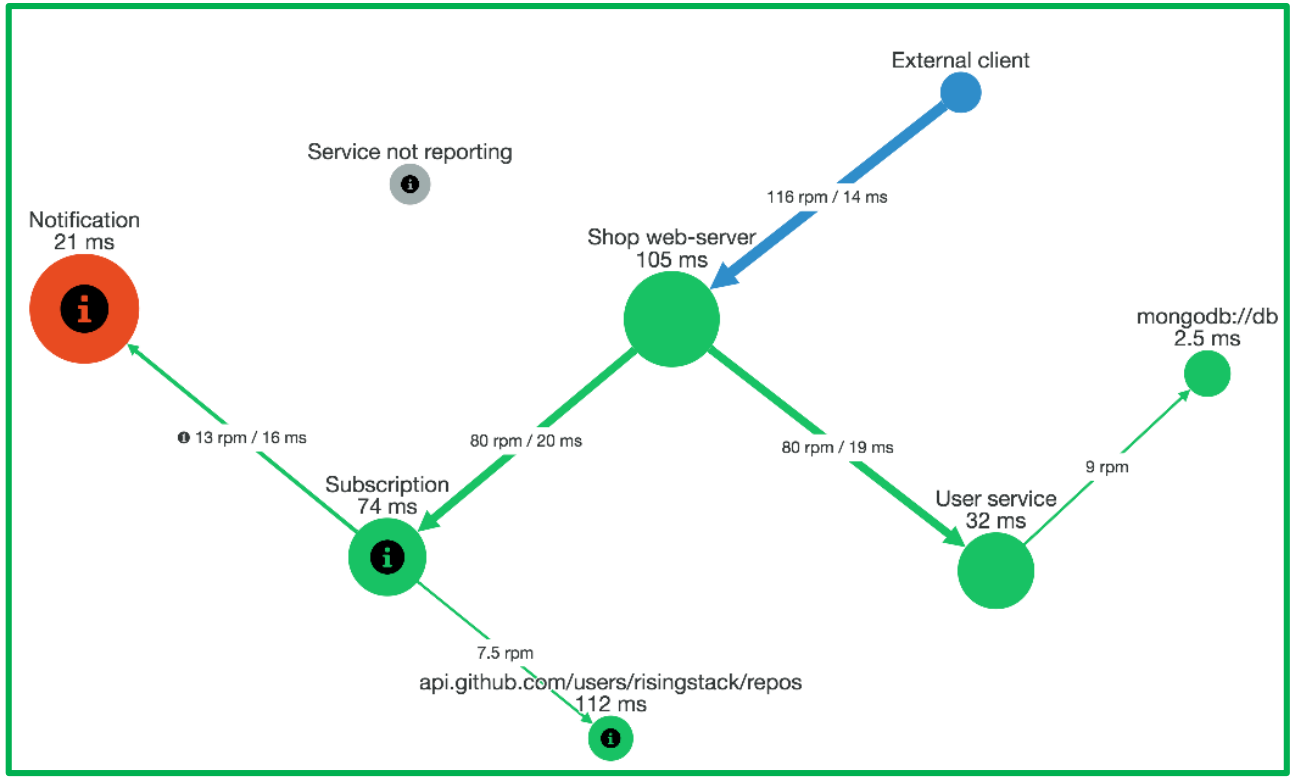
How to understand the behavior of an application and troubleshoot problems?

Forces

- 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

Solution

- Assigns each external request a unique external request id e.g. correlation-Id
- Passes the external request id to all services that are involved in handling the request
- Includes the external request id in all log messages
- Records information (e.g. start time, end time) about the requests and operations



Observability: Health check API

Context

Sometimes a service instance can be incapable of handling requests yet still be running. For example, it might have run out of database connections. When this occurs, the monitoring system should generate an alert. Also, 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?

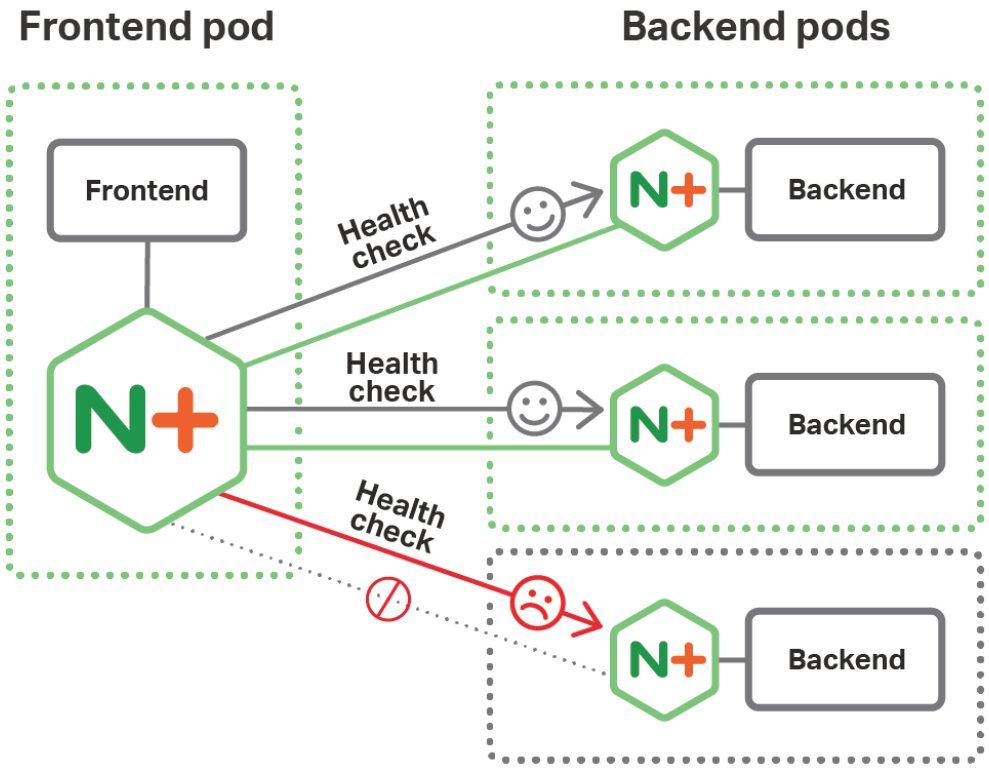
Forces

- An alert should be generated when a service instance fails
- Requests should be routed to working service instances

Solution

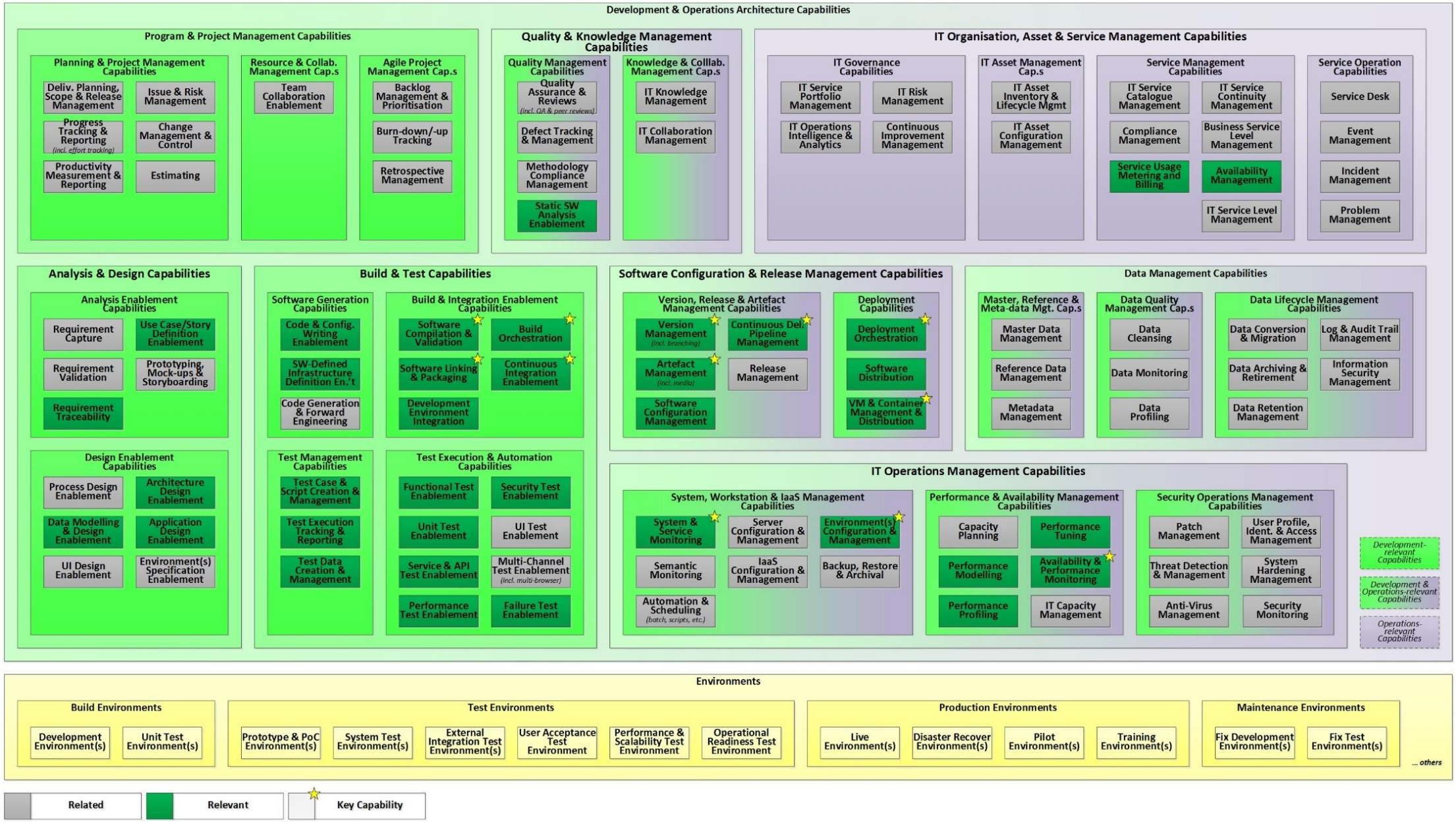
A service has a health check API endpoint (e.g. HTTP /health) that returns the health of the service. The API endpoint handler performs various checks, such as

- the status of the connections to the infrastructure services used by the service instance
- the status of the host, e.g. disk space
- application specific logic

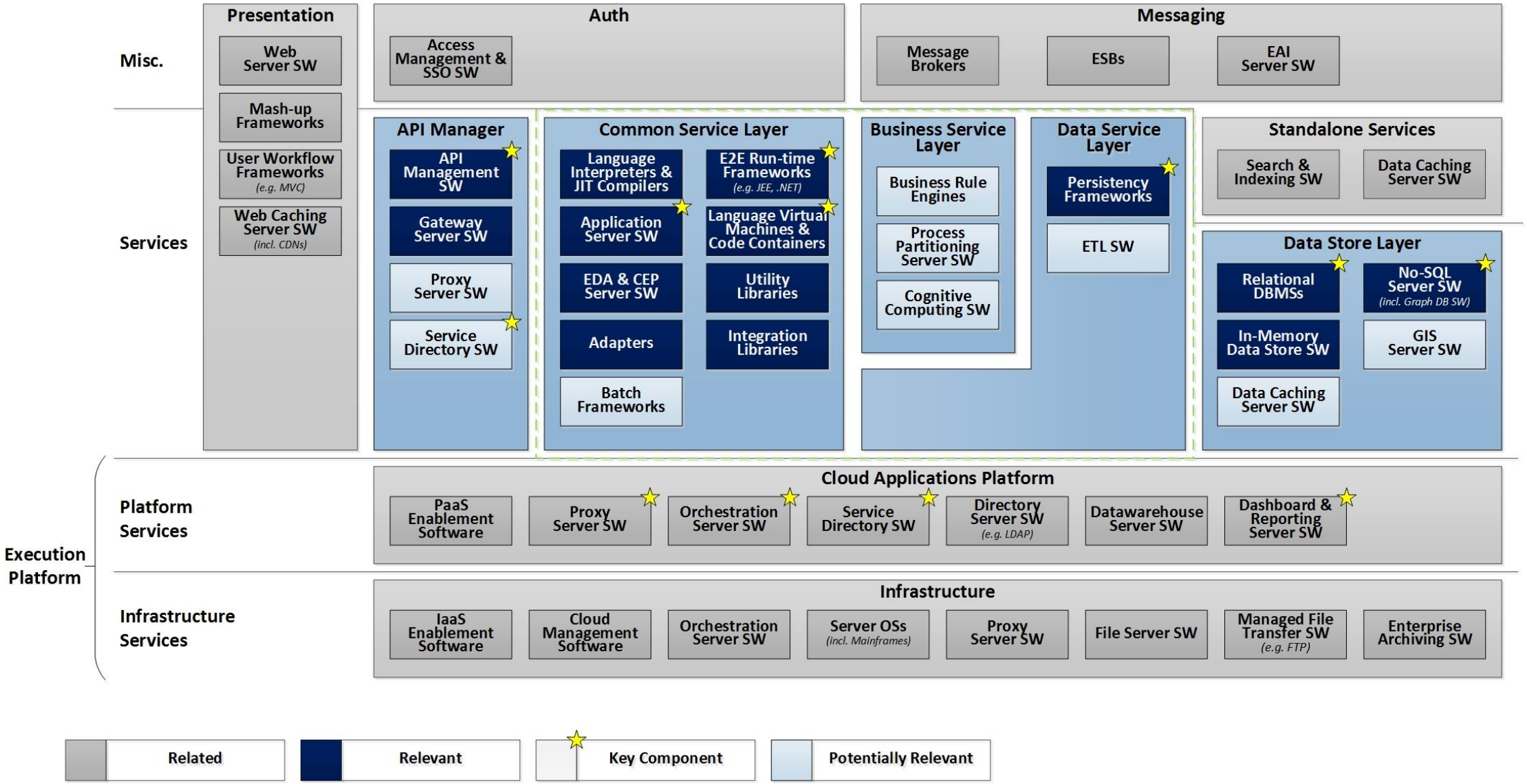


NGNIX Health check

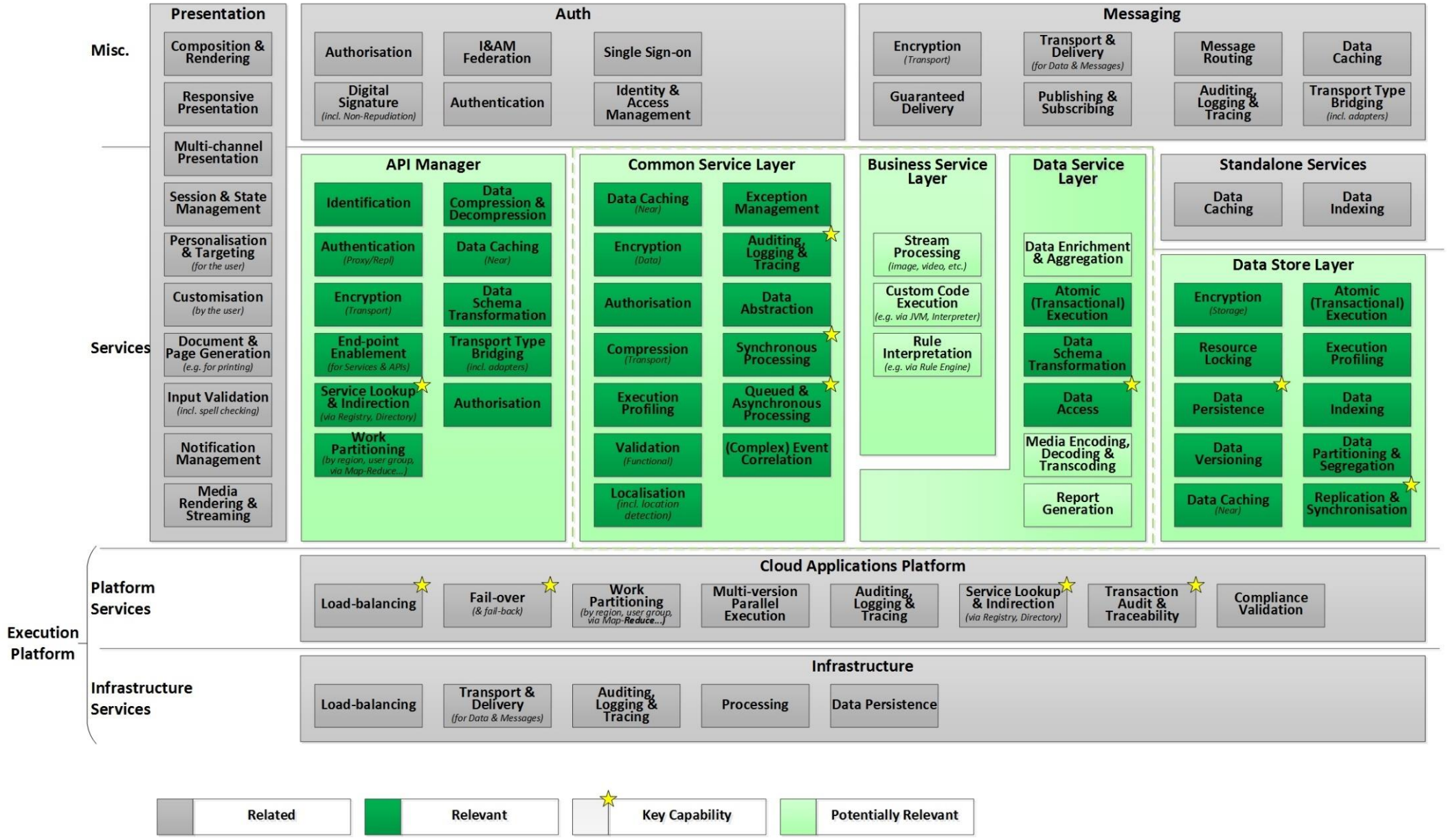
Microservices Reference Dev&Ops Capabilities



Microservices Reference Component



Microservices Reference Runtime Capabilities



THANKS