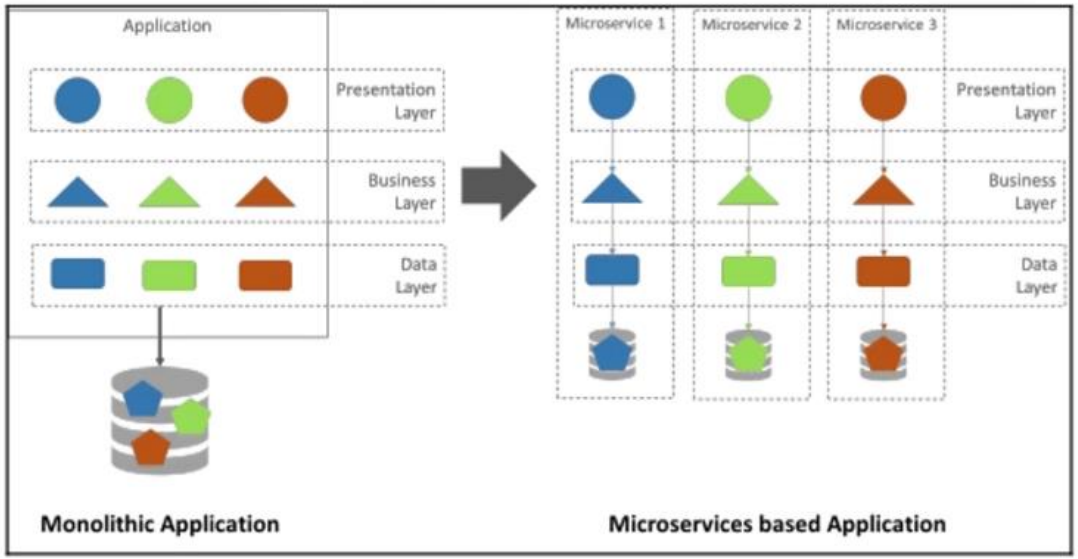
Micro Services

Microservices

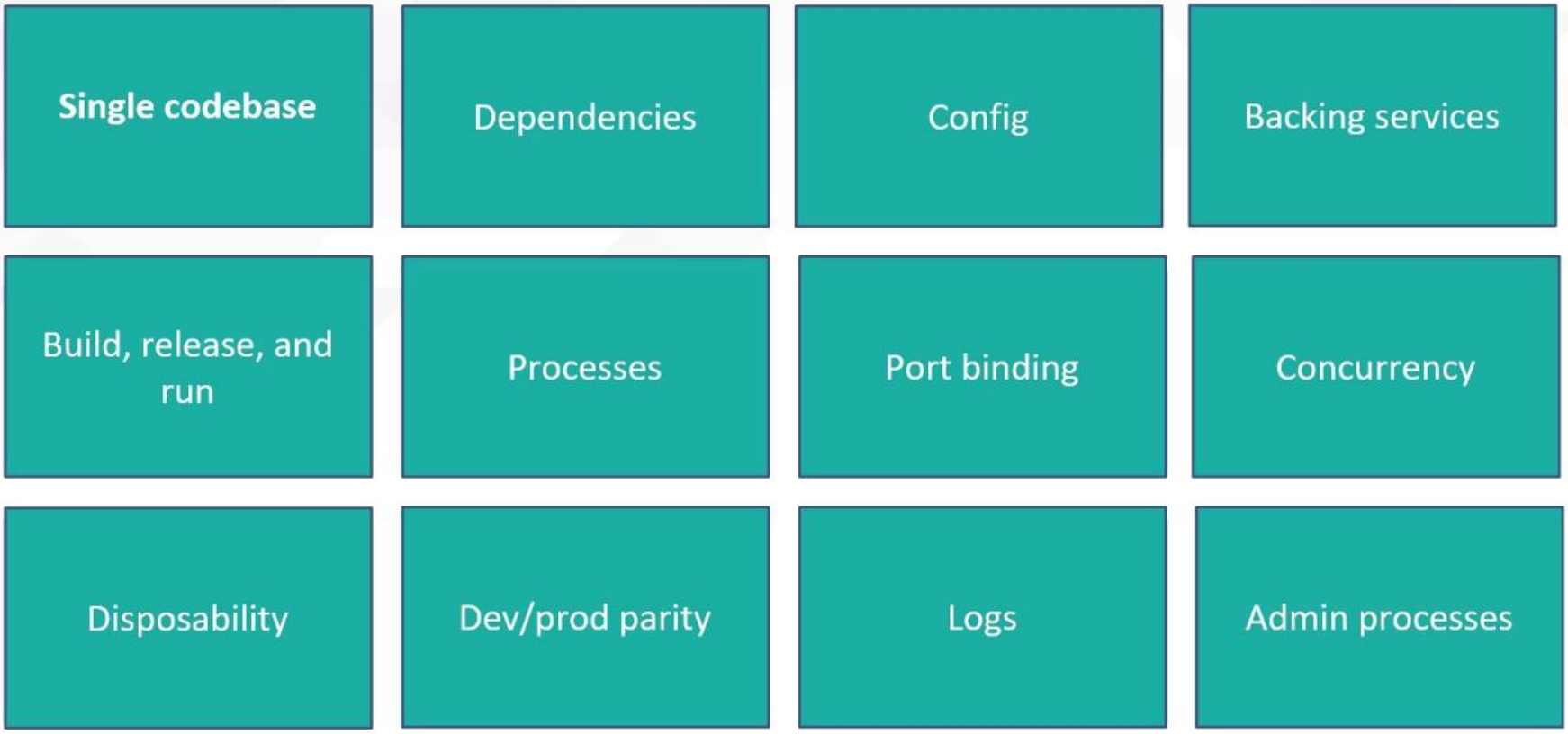
The microservices are designed around the following principles:

- Single-responsibility principle
- Share nothing
- Reactive
- Externalized configuration
- Consistent
- Resilient
- Good citizens
- Versioned

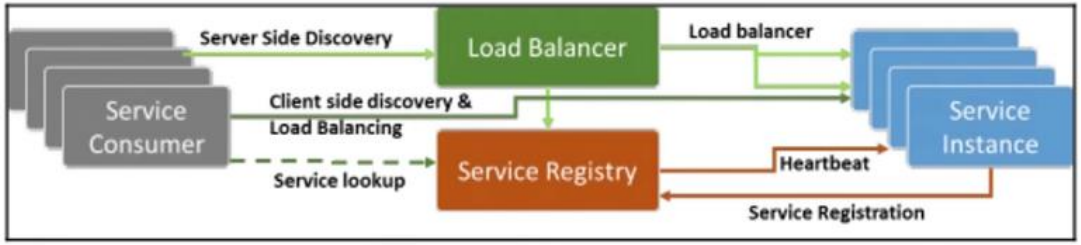


Moving from a monolithic to a microservices-based application

The 12-factor app



Service discovery



The service registry

Client-side load balancing

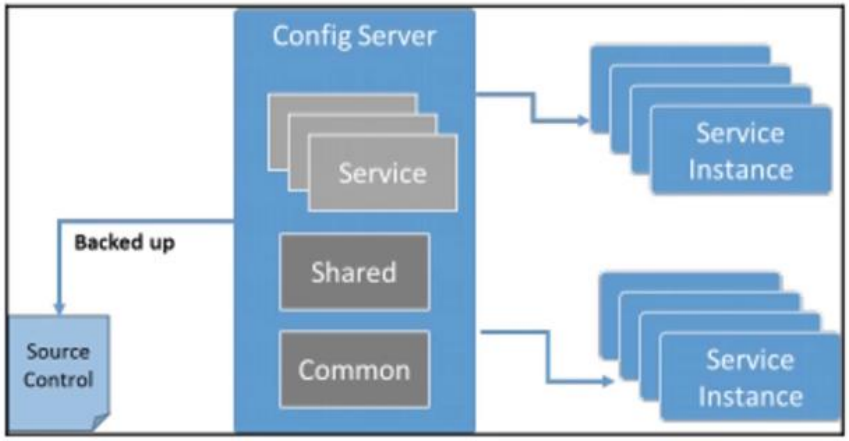
Server-side load balancing

Config server

Service-specific properties

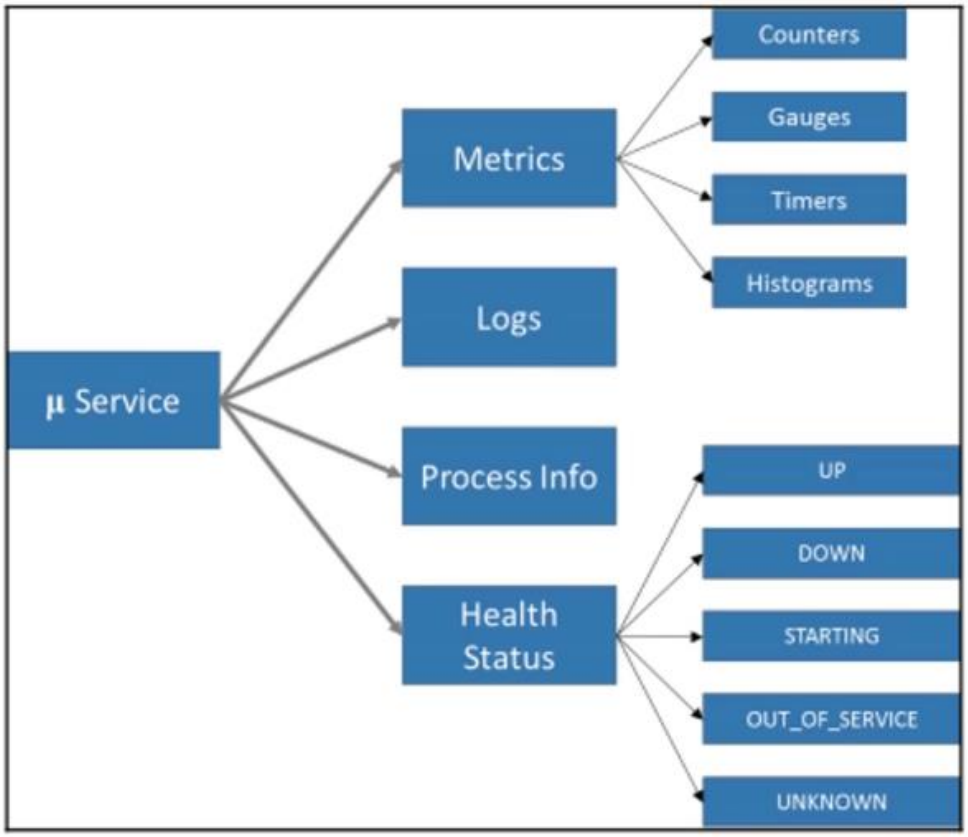
Shared properties

Common properties



The config server

Service management/monitoring



Service metrics

Service health

Service metrics

Process info

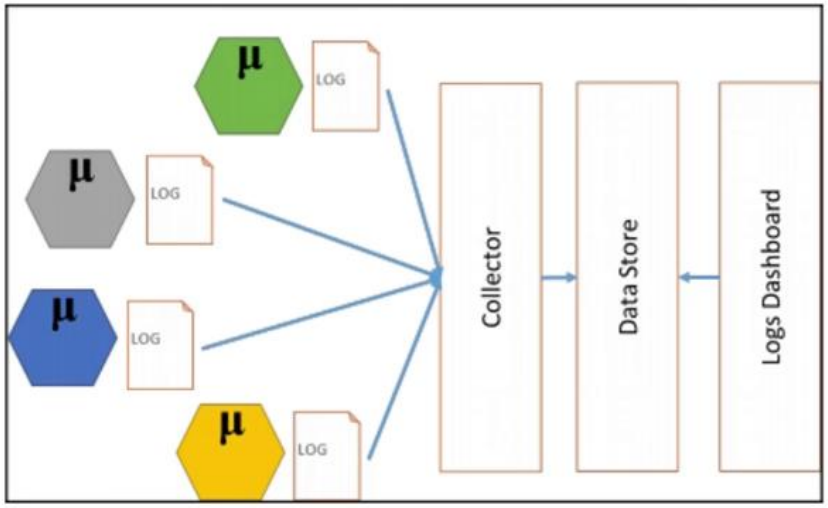
Log events as stream

Container management/orchestration

- Another key infrastructure piece of the microservice environment is container management and orchestration
- The services are typically bundled in a container and deployed in a PaaS environment
- To deploy and manage the dependencies between the containers, there is a need for container management and orchestration software
- It should be able to understand the interdependencies between the containers and deploy the containers as an application

Log aggregation

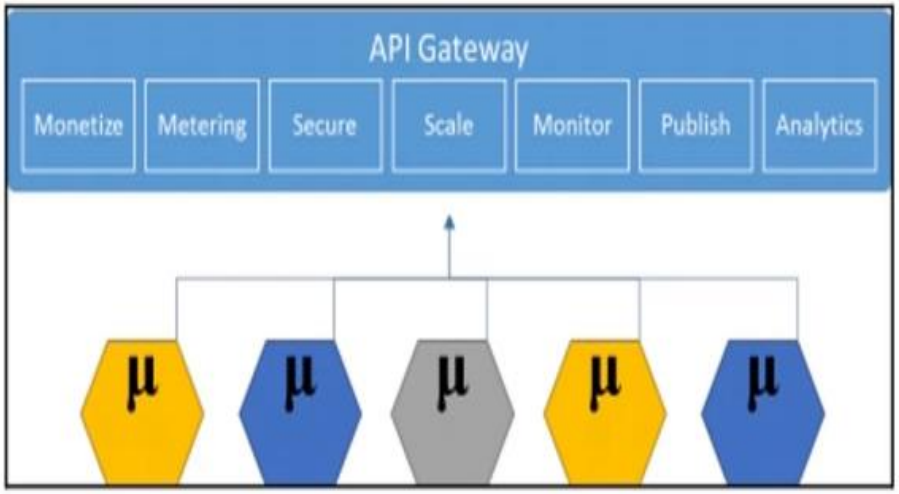
- The containers are meant to be stateless
- The log statements are typically stateful events that need to be persisted beyond the life of the containers
- All logs from the containers are treated as event streams that can be pushed/pulled onto a centralized log repository



Log aggregation

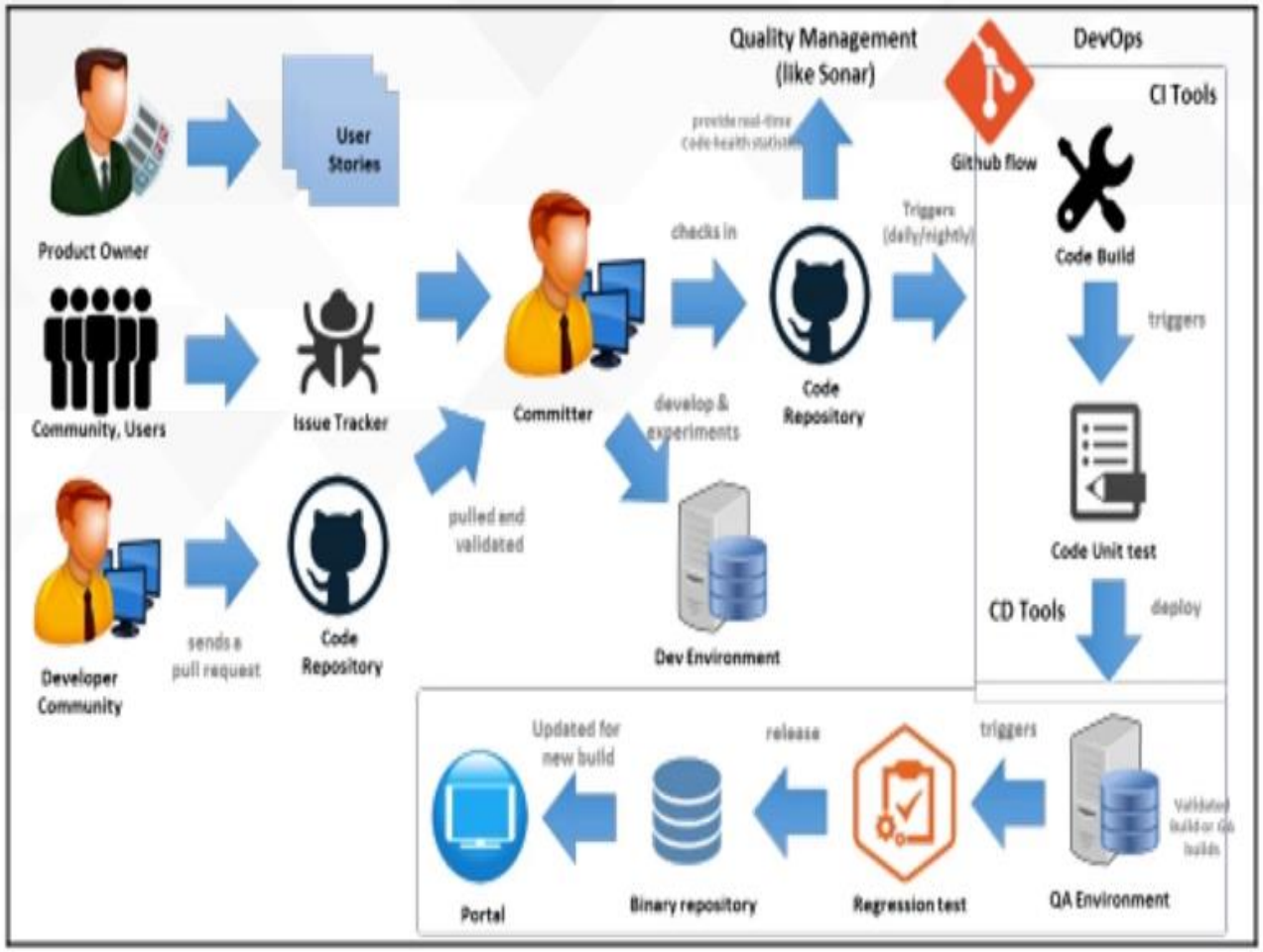
API Gateway/management

- API Gateway/management handles all concerns on behalf of the microservice
- It provides multiple options for managing the service endpoints and can also provide transformation, routing, and mediation capabilities
- The API Gateway is more lightweight, compared to the typical enterprise service bus



API Management Gateway

DevOps



Development Life cycle

The 12-factor app

Single codebase	Dependencies	Config	Backing services
Build, release, and run	Processes	Port binding	Concurrency
Disposability	Dev/prod parity	Logs	Admin processes

Monolithic transformation

Enterprises are attacking the problem of competition and innovation in a two-prong manner:

1. Setting the base platform that provides the core ecosystem as a set of services to deploy and run the microservices

2. The second approach is to chip at the monolithic application, one functional piece at a time, and migrate the core business logic to the microservice model

Popularity of REST, HTTP, and JSON

REST is a state representational style that provides a way to deal with interchange over HTTP.

REST has a lot factors in its favor:

Utilizes the HTTP protocol standard, giving it an immense leg up for anything and everything on WWW

Mechanism to isolate the access to entities while still utilizing the same HTTP request model

Supports JSON as the data format

Popularity of REST, HTTP, and JSON

- REST with JSON has become the dominant model over the SOAP/XML model.
- According to one statistic from ProgrammableWeb:

73% of the APIs on Programmable Web use REST. SOAP is far behind but is still represented in 17% of the APIs.

Popularity of REST, HTTP, and JSON

Let's cover some high-level reasons why the REST/JSON model is favored over the SOAP/XML model of service development:

SOAP model of contract first approach makes crafting web services difficult.

SOAP is complex compared to REST.

REST is lightweight compared to SOAP and does not tax the bandwidth as much as SOAP.

Support for SOAP outside of the Java world is limited.

XML parsing on the client side is memory and compute intensive.

XML Schema/markup provides structure definitions and validation models but at the expense of additional parsing.

Role of API Gateways

- An API gateway is a singular interface that handles all the incoming requests before redirecting to the internal servers.
- An API gateway typically provides the following functions:

Routes the incoming traffic to the appropriate service hosted with the provider's data center/cloud

Filters all the incoming traffic from all kind of channels—web, mobile, and so on

Implements security mechanisms (such as OAuth) to authenticate and log the service usage

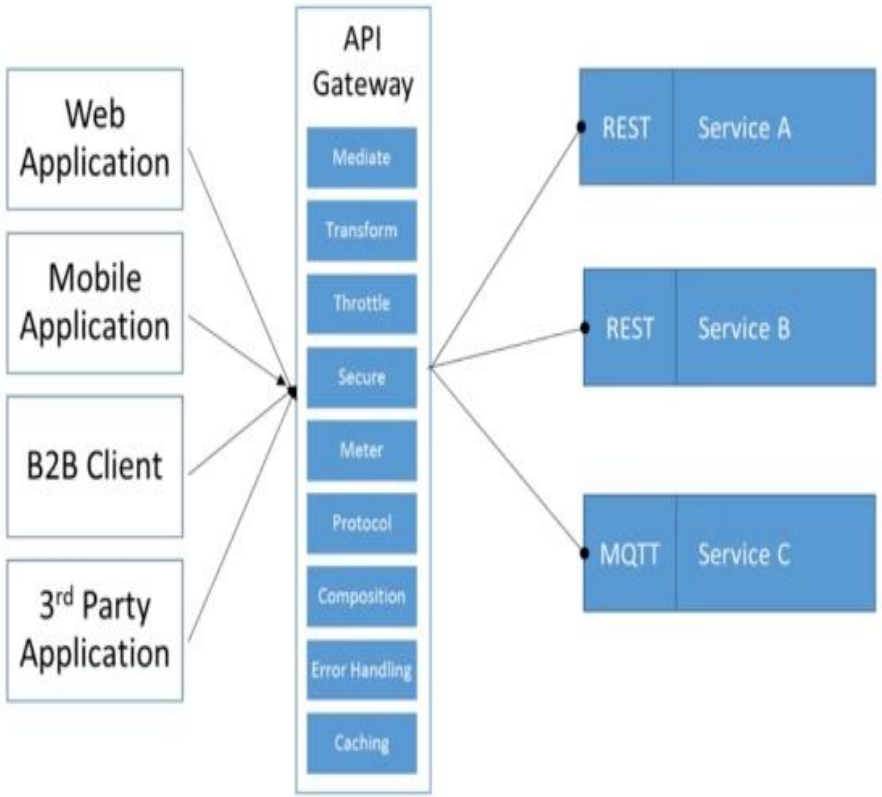
Provides ability to throttle and limit traffic to certain services

Transforms data between the service consumer and provider

Provides one or more APIs that map to an underlying service provider

Role of API Gateways

For different kind of consumers the same underlying service can be split into multiple custom APIs that are exposed to a different set of consumers, so that the consumer sees only the features it needs:



Benefits of an API Gateway

Use of API gateways provides the following benefits:

- Separation of Concerns
- Consumer Oriented
- API Oriented
- Orchestration
- Monitor

Benefits of an API Gateway

Use of API gateways provides the following benefits:

- **Separation of Concerns**
- **Consumer Oriented**
- **API Oriented**
- **Orchestration**
- **Monitor**

- Insulates the microservice providers from the service consumers on the application side.
- This allows the separation of the application tier from the service requesting clients.

Benefits of an API Gateway

Use of API gateways provides the following benefits:

- **Separation of Concerns**
- **Consumer Oriented**
- **API Oriented**
- **Orchestration**
- **Monitor**

- API gateways provide a unified hub for a large number of APIs and microservices.
- This allows the consumer to focus on API utility instead of locating where a service is hosted, managing service request limits, security, and so on.

Benefits of an API Gateway

Use of API gateways provides the following benefits:

- Separation of Concerns
- Consumer Oriented
- API Oriented
- Orchestration
- Monitor

Provides an optimum API based on the type of the client and required protocols.

Benefits of an API Gateway

Use of API gateways provides the following benefits:

- **Separation of Concerns**
- **Consumer Oriented**
- **API Oriented**
- **Orchestration**
- **Monitor**

- Provides the ability to orchestrate multiple services calls into one API call.
- Instead of calling multiple services, it can invoke one API.
- Fewer requests means less invocation overhead and improve the consumer experience overall.
- An API gateway is essential for mobile applications.

Benefits of an API Gateway

Use of API gateways provides the following benefits:

- Separation of Concerns
- Consumer Oriented
- API Oriented
- Orchestration
- Monitor

An API gateway also provides the ability to monitor API invocations, which in turn allows enterprises to evaluate the success of APIs and their usage.

Topic C
Application Decoupling

Bounded Context/Domain-driven Design

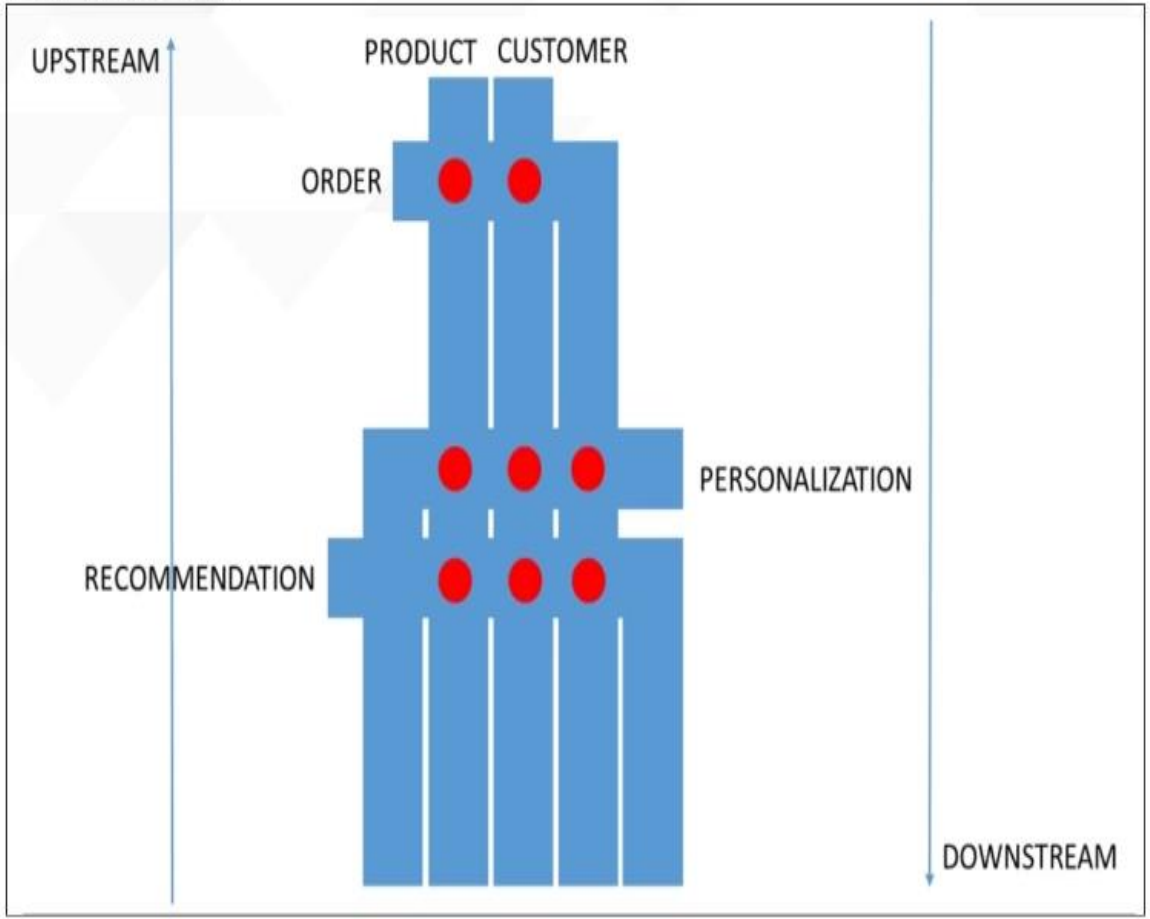
- The bounded context is a domain-driven design paradigm that helps to add a seam and create service groups.
- Bounded contexts work in solution-space to indicate that the services are related and belong to a common functional domain.
- It is built by one team that works with one business unit as per Inverse Conway's law.
- A bounded context may communicate with the other services/business capabilities through:

Exposing internal APIs or services

Emitting events on the event bus

Classification into Up/Downstream Services

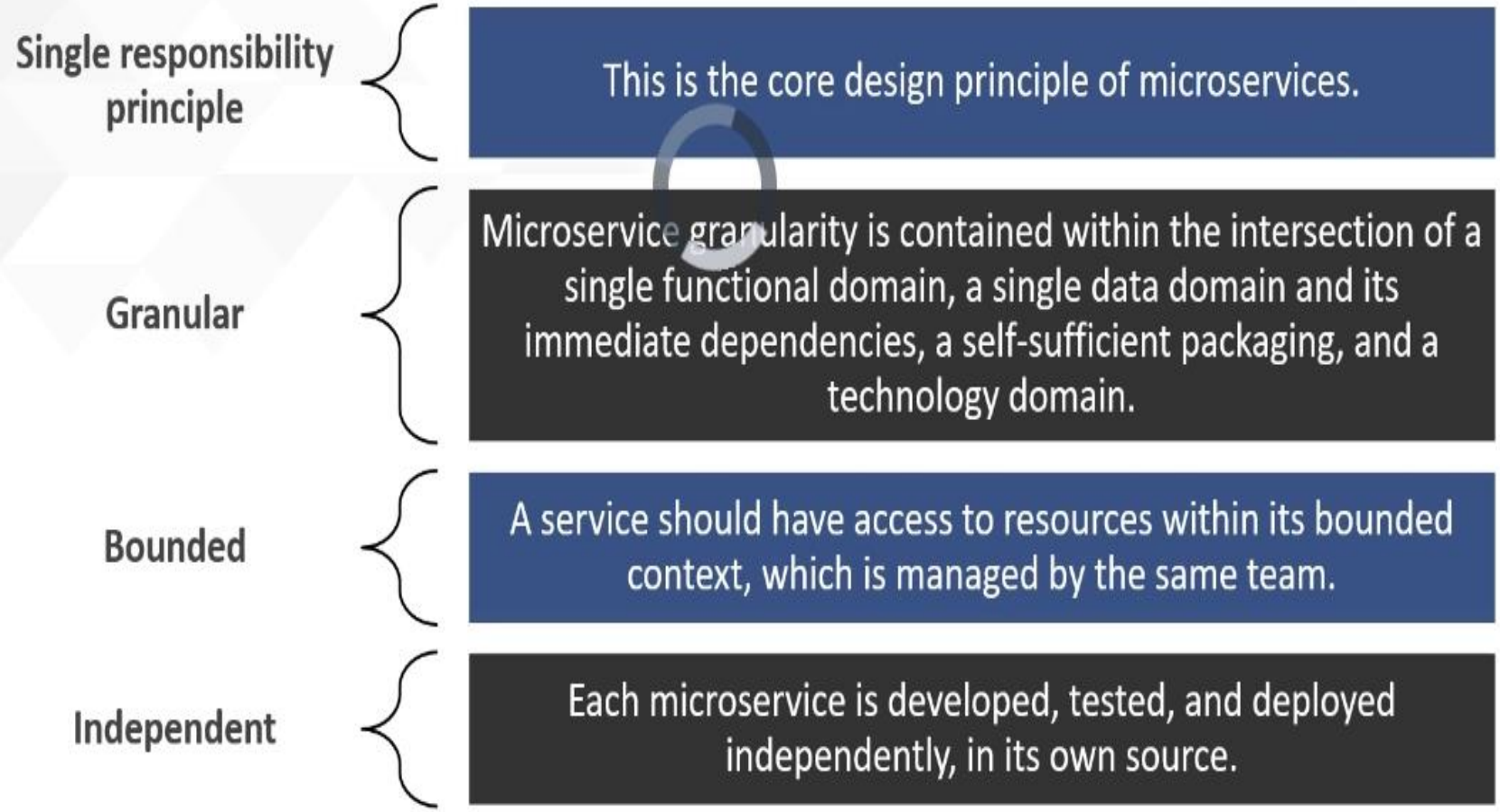
Let's take a simplified view of an e-commerce application, where the core entities are CUSTOMER and PRODUCT.



Topic D
Microservice Identification

Microservice Identification

- The name microservice does not necessarily mean that the service has to be small in size.
- But it has the following characteristics:



Differences Between Microservices and Service-Oriented Architecture (SOA)

Here are the differences between microservices and service-oriented architecture (SOA):

A service executes the entire business unit of work.

A service has its own private database or a database that is shared only in its bounded context and can store the information required to service the business unit of work.

A service is a smart endpoint and typically exposes a REST interface with a contract definition in Swagger or similar repository.

Service Granularity

Here are the types of services:

Atomic or system services

These are the services that do a unit level of work and are enough to service the request by either referring to a database or a downstream source.

Composite or process services

- These services depend on the coordination between two or more atomic services.
- Composite microservices are discouraged unless the business case already involves using existing atomic services.

Experience services

- These services are tied to the customer journey and are deployed at the edge of the infrastructure.
- These services handle requests from the mobile and web applications.
- These services are exposed through a reverse proxy using tools such as API gateways.

Topic E
Microservice Design Guidelines

Microservice Design Guidelines

These guidelines are in line with the 12-factor applications guidelines given by Heroku engineers:

- Lightweight
- Reactive
- Stateless
- Atomic
- Externalized configuration
- Consistent
- Resilient
- Good citizens
- Versioned

Microservice Design Guidelines

These guidelines are in line with the 12-factor applications guidelines given by Heroku engineers:

- **Lightweight**
- Reactive
- Stateless
- Atomic
- Externalized configuration
- Consistent
- Resilient
- Good citizens
- Versioned

- Microservices have to be lightweight in order to facilitate smaller memory footprints and faster startup times.
- This facilitates faster MTTR, and allows for services to be deployed on smaller runtime instances.
- Compared to heavy runtime times, such as application servers, smaller runtimes such as Tomcat, Netty, Node.js, and Undertow are more suited.
- The services should exchange data in lightweight text formats, such as JSON, or binary formats, such as Avro, Thrift, or Protocol Buffers.

Microservice Design Guidelines

These guidelines are in line with the 12-factor applications guidelines given by Heroku engineers:

- Lightweight
- **Reactive**
- Stateless
- Atomic
- Externalized configuration
- Consistent
- Resilient
- Good citizens
- Versioned

- This is applicable to services with highly concurrent loads or slightly longer response times.
- Typical server implementations block threads to execute imperative programming styles.
- Blocking threads could increase operating system overheads.
- The Reactive style operates on nonblocking I/O, uses call back handlers, and reacts to events.
- This does not block threads and as a result, increases the scalability and load handling characteristics of the microservices much better.
- Database drivers have started supporting reactive paradigms, for example, MongoDB Reactive Streams Java Driver.

Microservice Design Guidelines

These guidelines are in line with the 12-factor applications guidelines given by Heroku engineers:

- Lightweight
- Reactive
- Stateless
- **Atomic**
- Externalized configuration
- Consistent
- Resilient
- Good citizens
- Versioned

- This is the core design principle of microservices.
- They should be easy to change, test, and deploy.
- All these can be achieved if the services are reasonably small and do the smallest business unit of work that can be done independently.
- The services will be easier to modify and independently deploy.
- Composite microservices may be required on a need basis but should be limited in design.

Microservice Design Guidelines

These guidelines are in line with the 12-factor applications guidelines given by Heroku engineers:

- Lightweight
- Reactive
- Stateless
- **Atomic**
- Externalized configuration
- Consistent
- Resilient
- Good citizens
- Versioned

- This is the core design principle of microservices.
- They should be easy to change, test, and deploy.
- All these can be achieved if the services are reasonably small and do the smallest business unit of work that can be done independently.
- The services will be easier to modify and independently deploy.
- Composite microservices may be required on a need basis but should be limited in design.

Microservice Design Guidelines

These guidelines are in line with the 12-factor applications guidelines given by Heroku engineers:

- Lightweight
- Reactive
- Stateless
- Atomic
- Externalized configuration
- **Consistent**
- Resilient
- Good citizens
- Versioned

- Services should be written in a consistent style as per the coding standards and naming convention guidelines.
- Common concerns such as serialization, REST, exception handling, logging, configuration, property access, metering, monitoring, provisioning, validations, and data access should be consistently done through reusable assets, annotations, and so on.
- It should be easier for another developer from the same team to understand the intent and operation of the service.

Microservice Design Guidelines

These guidelines are in line with the 12-factor applications guidelines given by Heroku engineers:

- Lightweight
- Reactive
- Stateless
- Atomic
- Externalized configuration
- **Consistent**
- Resilient
- Good citizens
- Versioned

- Services should be written in a consistent style as per the coding standards and naming convention guidelines.
- Common concerns such as serialization, REST, exception handlin logging, configuration, property access, metering, monitoring, provisioning, validations, and data access should be consistently done through reusable assets, annotations, and so on.
- It should be easier for another developer from the same team to understand the intent and operation of the service.

Microservice Design Guidelines

These guidelines are in line with the 12-factor applications guidelines given by Heroku engineers:

- Lightweight
- Reactive
- Stateless
- Atomic
- Externalized configuration
- Consistent
- **Resilient**
- Good citizens
- Versioned

- Services should handle exceptions arising from technical reasons and business reasons and not crash.
- They should use patterns such as timeouts and circuit breakers to ensure that the failures are handled carefully.

Microservice Design Guidelines

These guidelines are in line with the 12-factor applications guidelines given by Heroku engineers:

- Lightweight
- Reactive
- Stateless
- Atomic
- Externalized configuration
- Consistent
- Resilient
- **Good citizens**
- Versioned

- Report their usage statistics, number of times accessed, average response times, and so on through JMX API, and/or publish it through libraries to central monitoring infrastructures, log audit, error, and business events in the standards prescribed.
- Expose their condition through health check interfaces.

Microservice Design Guidelines

These guidelines are in line with the 12-factor applications guidelines given by Heroku engineers

- Lightweight
- Reactive
- Stateless
- Atomic
- Externalized configuration
- Consistent
- Resilient
- Good citizens
- Versioned

- Microservices may need to support multiple versions for diffe clients.
- The deployments and URL should support semantic versionin that is, X.X.X.

Microservice Design Guidelines

Microservices will need to leverage additional capabilities that are typically built at an enterprise level such as:

Dynamic service registry | **Log aggregation** | **External configuration** | **Provisioning and auto-scaling** | **API gateway**

Microservice registers itself with a service registry when up.

Microservice Design Guidelines

Microservices will need to leverage additional capabilities that are typically built at an enterprise level such as:

Dynamic service registry | **Log aggregation** | **External configuration** | **Provisioning and auto-scaling** | **API gateway**

- The logs generated by a microservice can be aggregated for central analysis and troubleshooting.
- The log aggregation is a separate infrastructure and typically built as an async model.
- Products such as Splunk and ELK Stack in conjunction with event streams such as Kafka are used to build/deploy the log aggregation systems.

Microservice Design Guidelines

Microservices will need to leverage additional capabilities that are typically built at an enterprise level such as:

- Dynamic service registry
- Log aggregation
- External configuration
- Provisioning and auto-scaling
- API gateway

The microservice can get the parameters and properties from an external configuration such as Consul and Zookeeper to initialize and run.

Microservice Design Guidelines

Microservices will need to leverage additional capabilities that are typically built at an enterprise level such as:

- Dynamic service registry
- Log aggregation
- External configuration
- Provisioning and auto-scaling
- API gateway

The service is automatically started by a PaaS environment if it detects a need to start an additional instance based on incoming load, some services failing, or not responding in time.

Microservice Design Guidelines

Microservices will need to leverage additional capabilities that are typically built at an enterprise level such as:

- Dynamic service registry
- Log aggregation
- External configuration
- Provisioning and auto-scaling
- API gateway

A microservice interface can be exposed to the clients or other divisions through an API gateway that provides abstraction, security, throttling, and service aggregation.

Topic F
Microservice Patterns

Content Aggregation Patterns

- With microservices and bounded context, there is an additional responsibility of content aggregation.
- A client may need information that spans multiple domains or business areas.
- The content required may not be available with one service.
- These patterns help identify and model the experience services category mostly.
- Hence there are various patterns for aggregation that can be applied.
- They are:

Aggregation by client

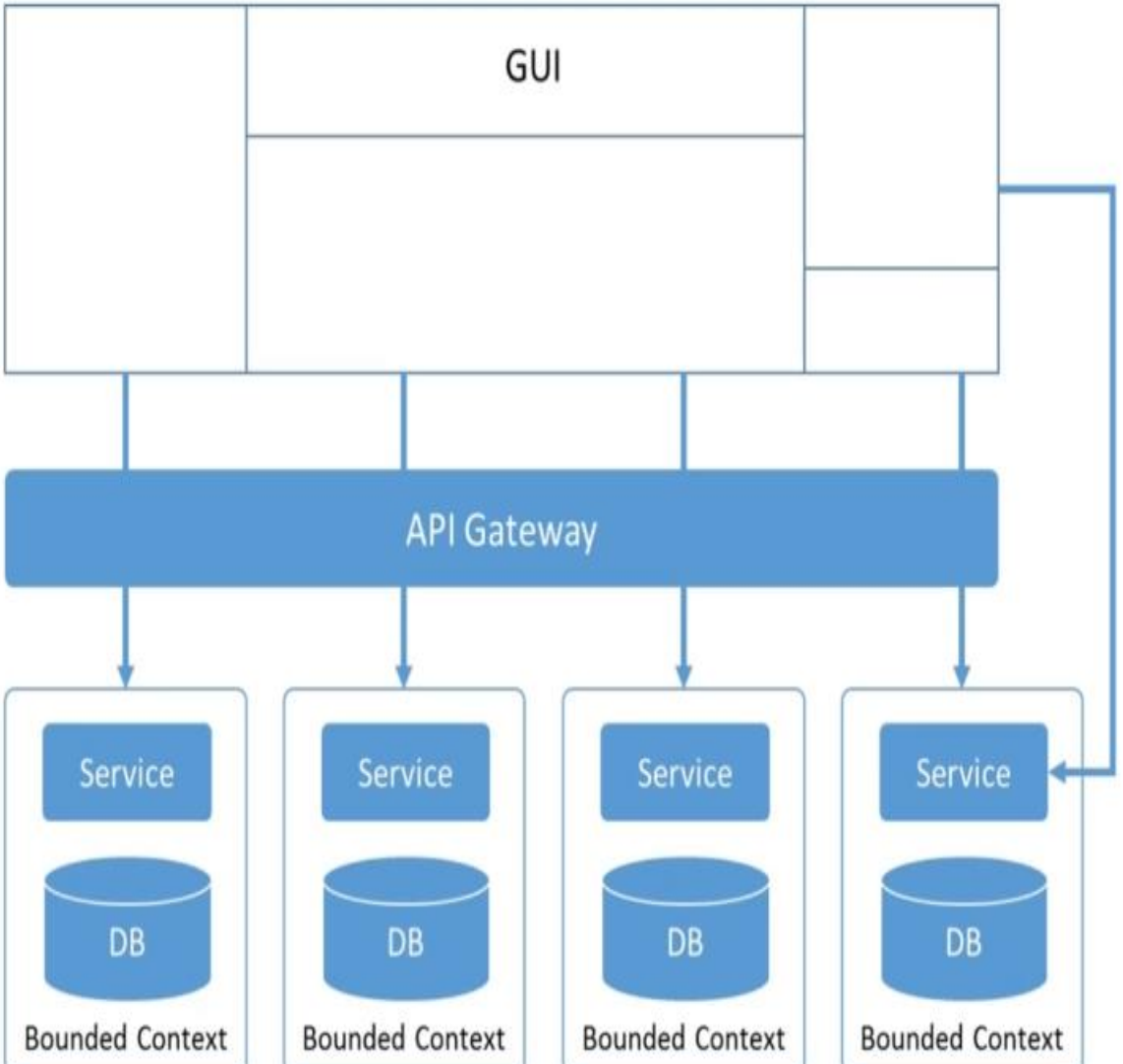
API aggregation

Microservice aggregation

Database aggregation

Aggregation by Client

- Aggregation at the last mile.
- This applies to web browsers or a reasonable processing capable user interface.
- This pattern is typically used in the home page that aggregates various subject areas.
- It's the pattern popularly used by Amazon.

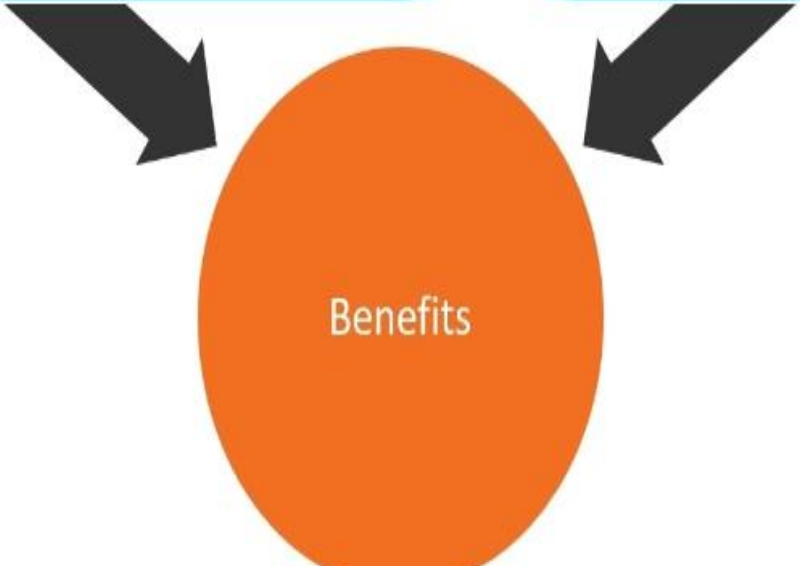


Aggregation by Client – Benefits

The benefits of using the aggregation by the client pattern are as follows:

- Decoupled approach at the services layer.
- Easier for agility and maintainability at each individual service.

- Faster perceived performance at the UI layer, since the requests, can run in parallel to populate the various areas on the screen.
- More enhanced when there is a higher bandwidth available to fetch data in parallel.



Aggregation by Client – Benefits

The benefits of using the aggregation by the client pattern are as follows:

- Decoupled approach at the services layer.
- Easier for agility and maintainability at each individual service.

- Faster perceived performance at the UI layer, since the requests, can run in parallel to populate the various areas on the screen.
- More enhanced when there is a higher bandwidth available to fetch data in parallel.

Aggregation by Client – Trade-offs

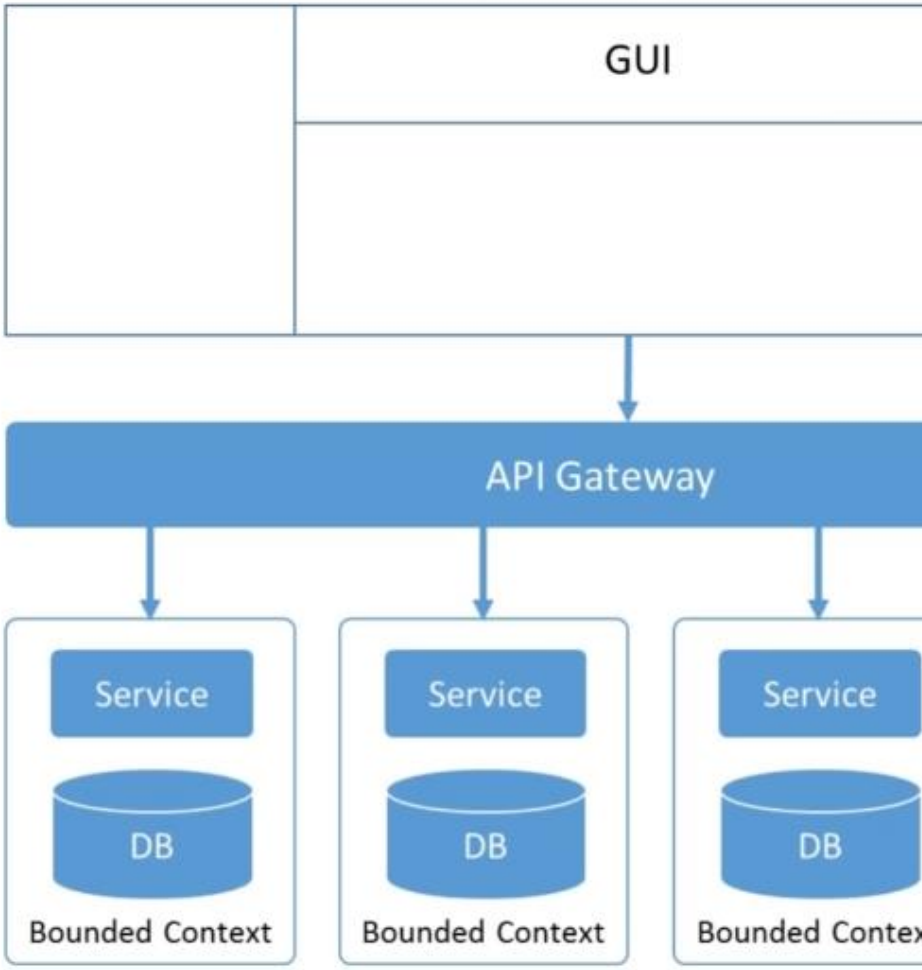
The trade-offs associated with the aggregation by the client pattern are as follows:

Sophisticated user interface processing capabilities, such as Ajax and single-page application required

The knowledge of aggregation is exposed at the UI layer, hence if the similar output was given as a dataset to a third-party, aggregation would be required

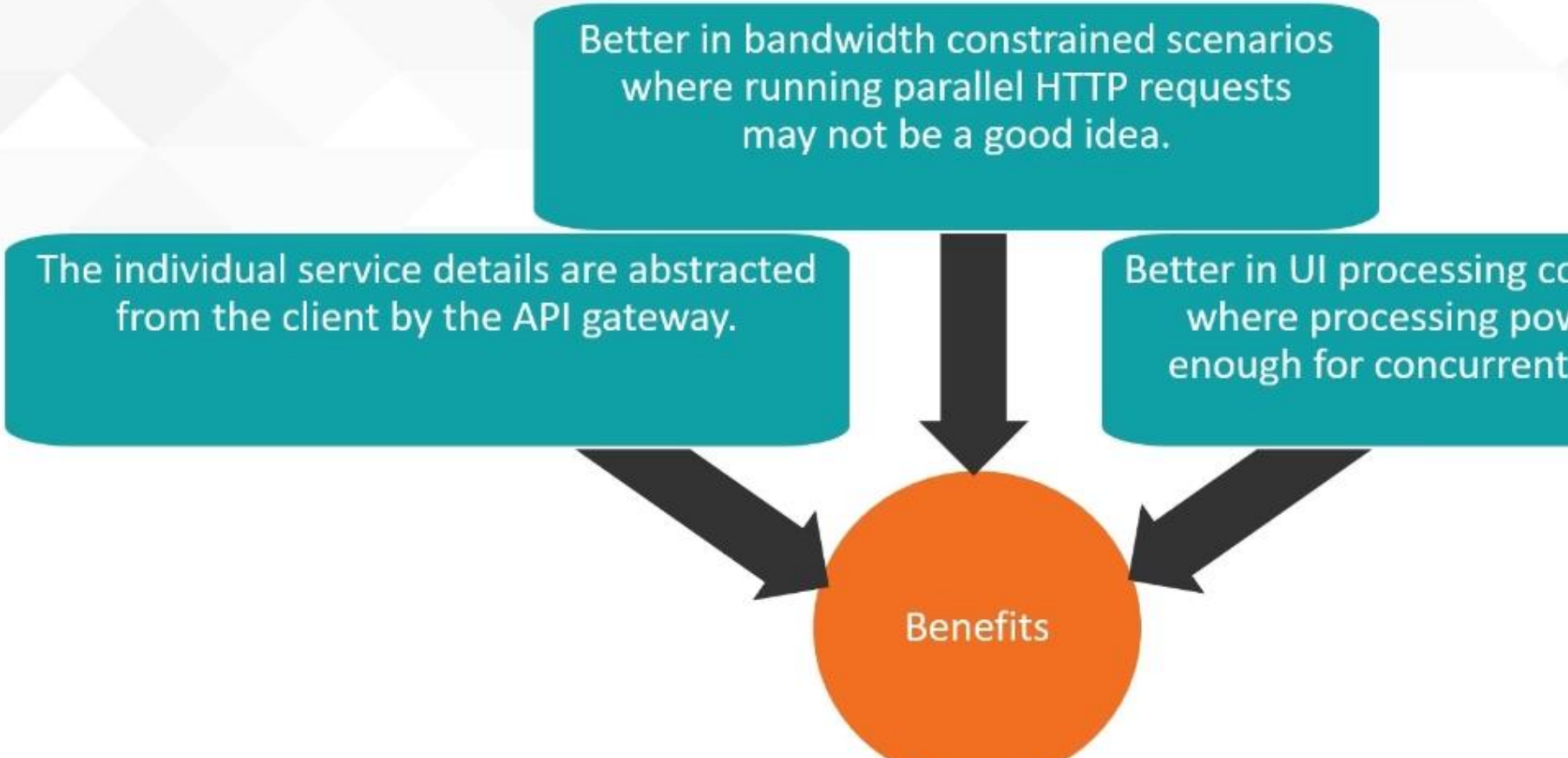
API Aggregation

- Aggregation at the gates applies to mobile or third-party use cases that do not want to know the details of the aggregation and instead would want to expect one data structure over a single request.
- The API gateways are designed to do this aggregation and then expose a unified service to the client.



API Aggregation – Benefits

The benefits of using the API aggregation pattern are as follows:



Microservice Aggregation – Trade-offs

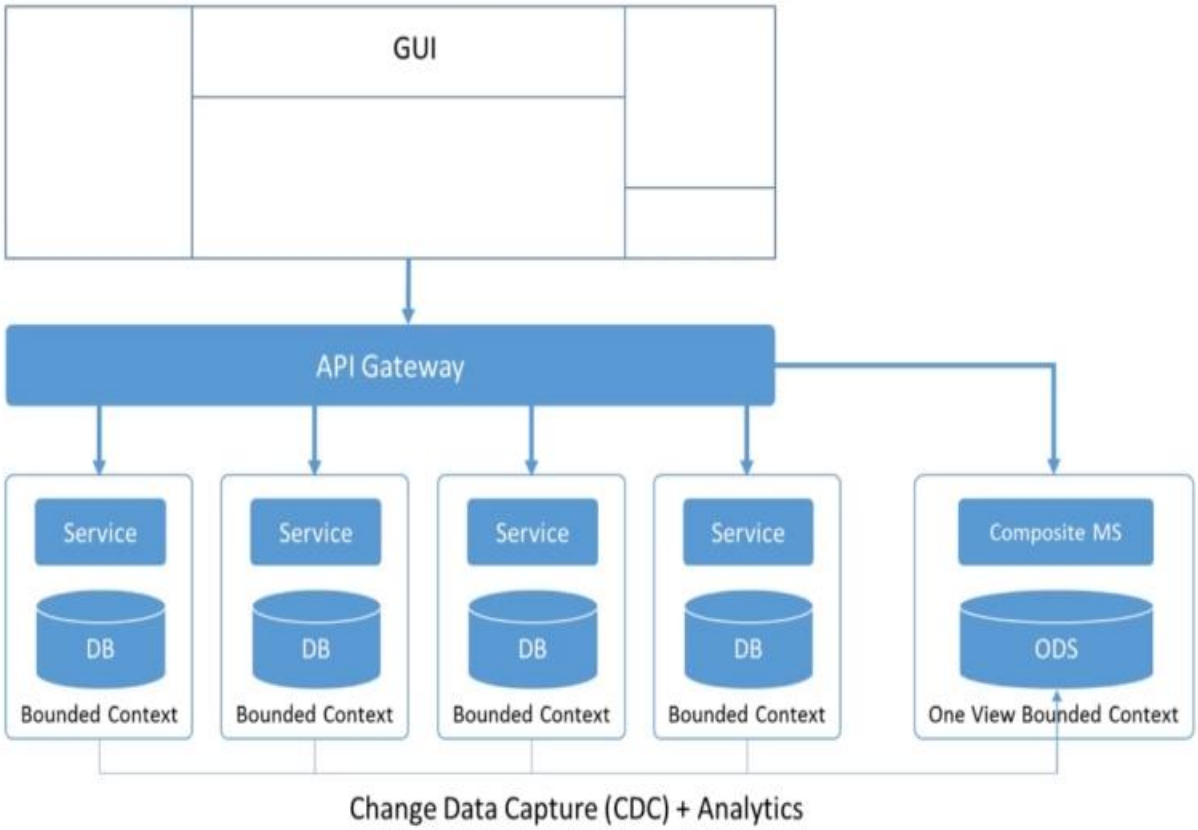
The trade-offs associated with the microservice aggregation pattern are as follows:

Lower latency and more code, as there is an additional hop introduced due to an additional step.

- More chances of failure or making mistakes.
- Parallel aggregation from microservices will need sophisticated code such as reactive or call back mechanisms.

Database Aggregation

- In the aggregation at the data tier approach, data is pre-aggregated into an operational data store typically a document database.
- This approach is useful for scenarios where there is additional business inference on the aggregated data that is difficult to compute in real time through a microservice.

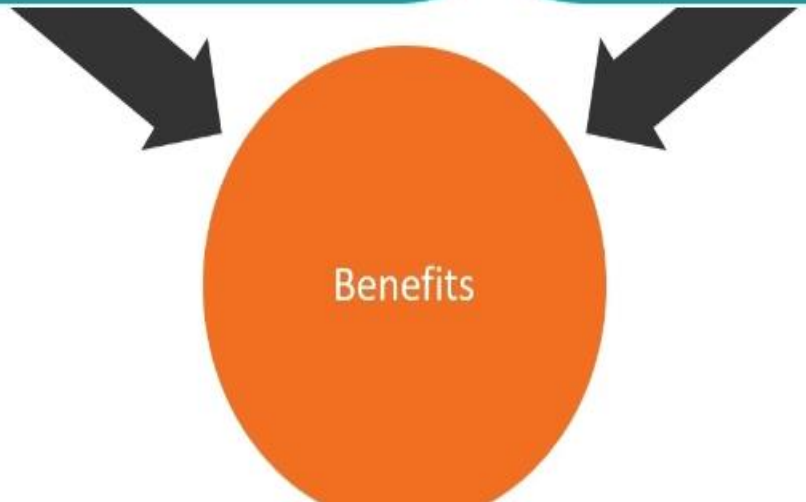


Database Aggregation – Benefits

The benefits of using the database aggregation pattern are as follows:

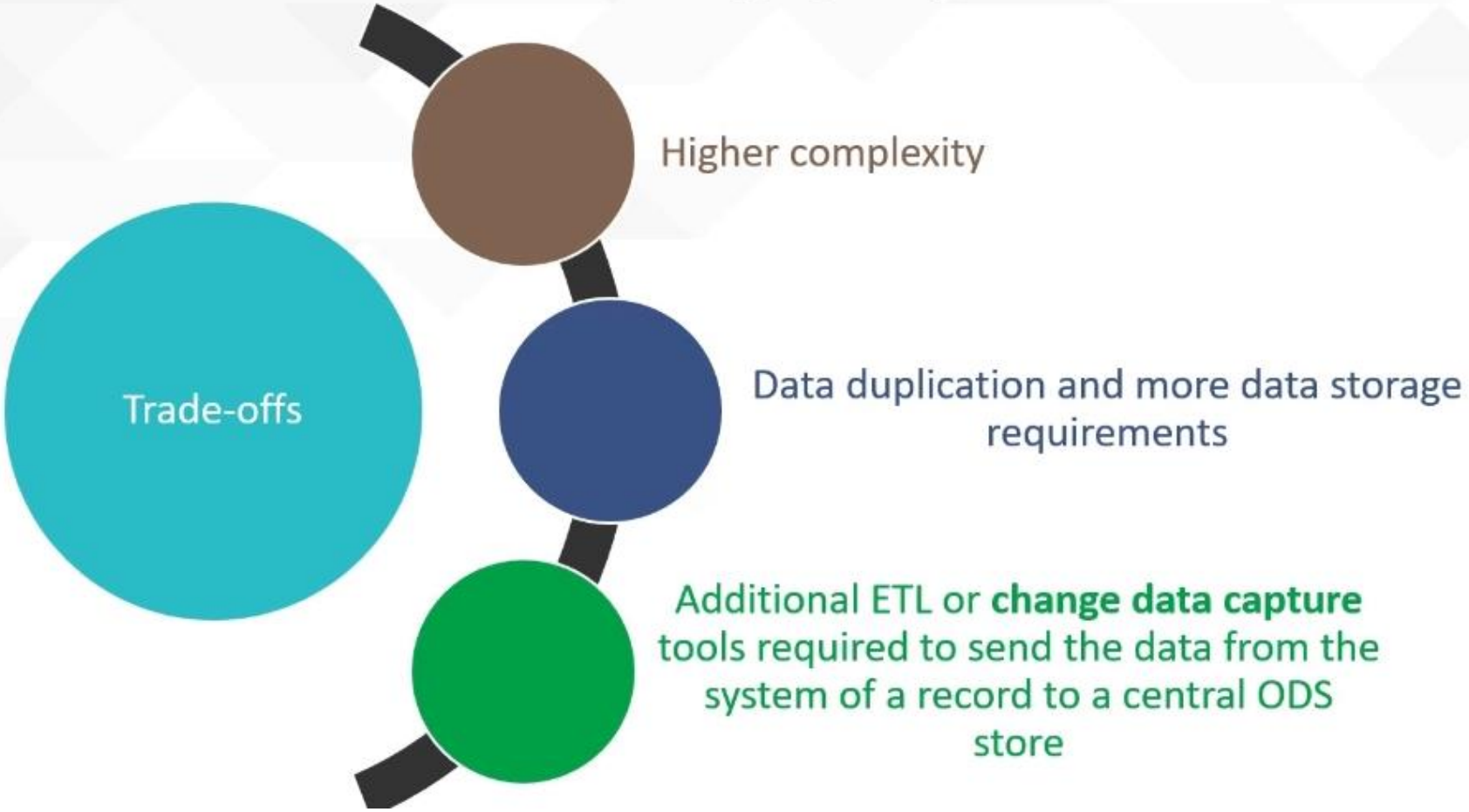
- Additional enrichment of data by analytical jobs is possible.
- In a customer 360° view, based on the customer portfolio aggregated in the ODS, additional analytics can be applied for next-best-action scenarios.

More flexible and capable as compared to the earlier approaches, and finer control on the data model can be exercised.



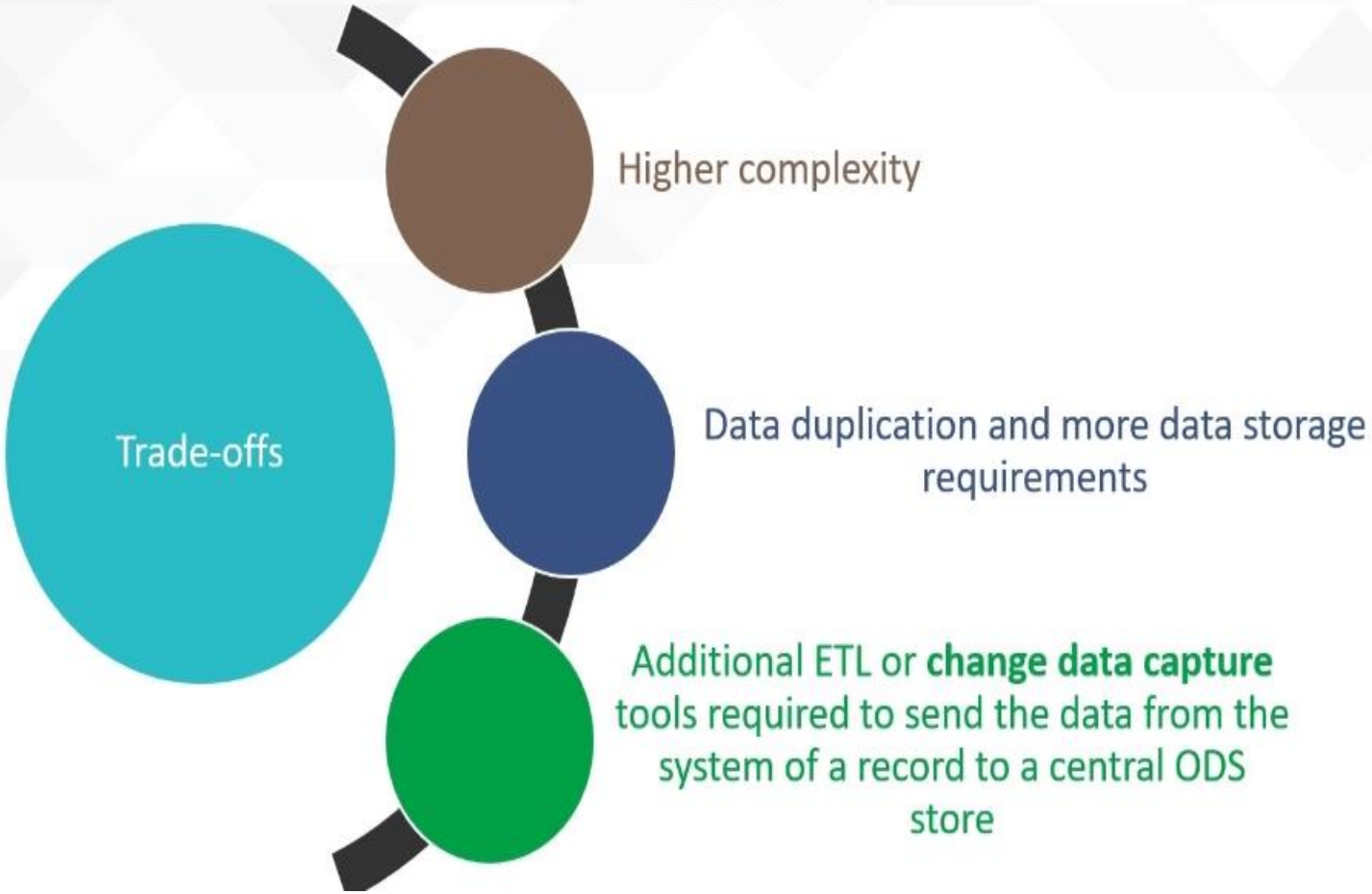
Database Aggregation – Trade-offs

The trade-offs associated with the database aggregation pattern are as follows:



Database Aggregation – Trade-offs

The trade-offs associated with the database aggregation pattern are as follows:



Coordination Models

The coordination styles of composite service micro flows are:

- **Asynchronous parallel**
- **Asynchronous sequential**
- **Orchestration using request/response**
- **Collapsing the microservices**

Coordination Models

The coordination styles of composite service micro flows are:

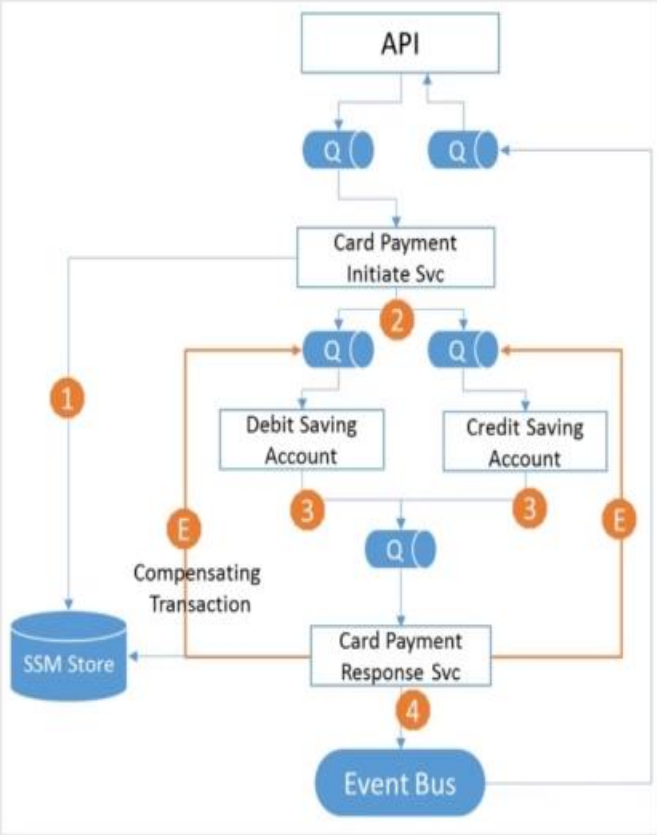
Asynchronous parallel

Asynchronous sequential

Orchestration using request/response

Collapsing the microservices

- A composite service initiates the service calls asynchronously to the constituent atomic services and then listens to the service response.
- If either services fails, it sends a compensating transaction to the other service.
- This is similar to the Scatter-Gather or Composed Message Processor patterns of the EIP.



Coordination Models

The coordination styles of composite service micro flows are:

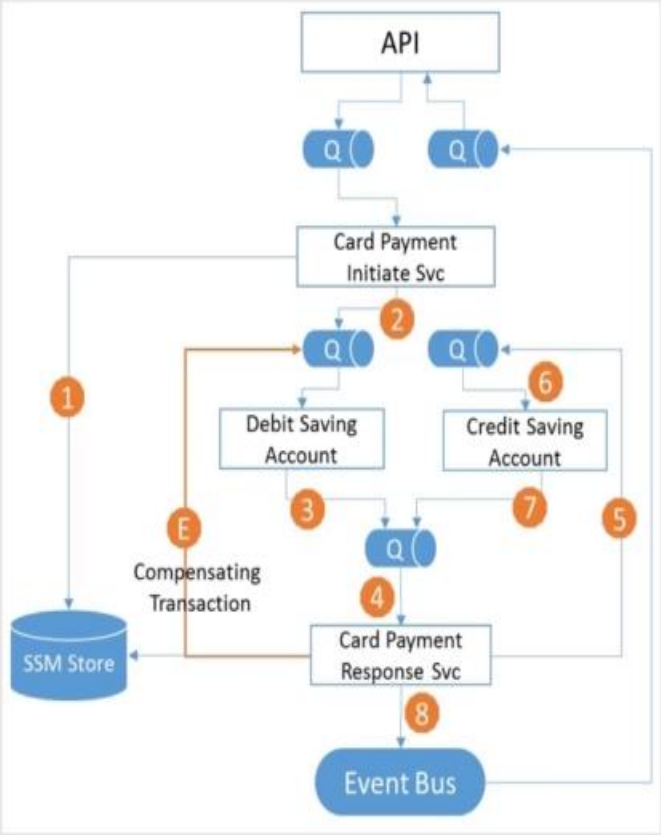
Asynchronous parallel

Asynchronous sequential

Orchestration using request/response

Collapsing the microservices

- Composite services send messages to atomic services sequentially.
- It waits for the previous service to return success before calling the next service.
- If anyone service fails, then the composite service sends the compensating transaction to previously successful services.
- This is similar to the Process Manager pattern in the EIP.

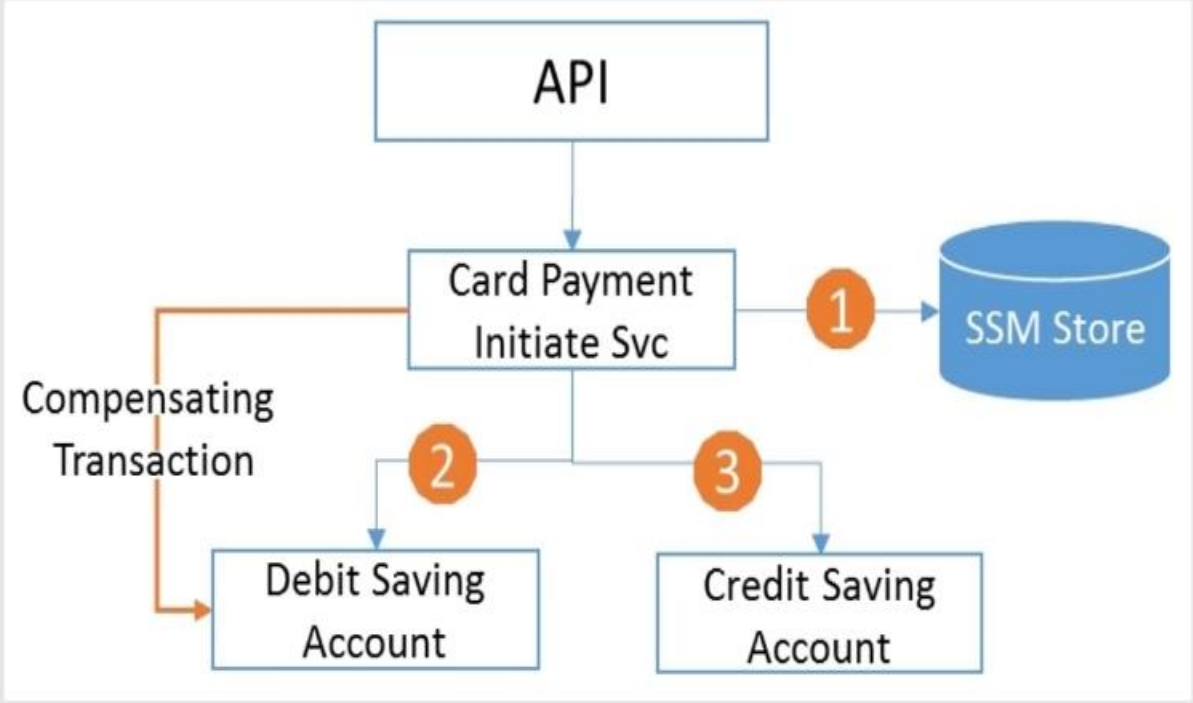


Coordination Models

The coordination styles of composite service micro flows are:

- **Asynchronous parallel**
- **Asynchronous sequential**
- **Orchestration using request/response**
- **Collapsing the microservices**

Similar to the preceding section, but in request/response and sync fashion instead of async messaging.



Coordination Models

The coordination styles of composite service micro flows are:

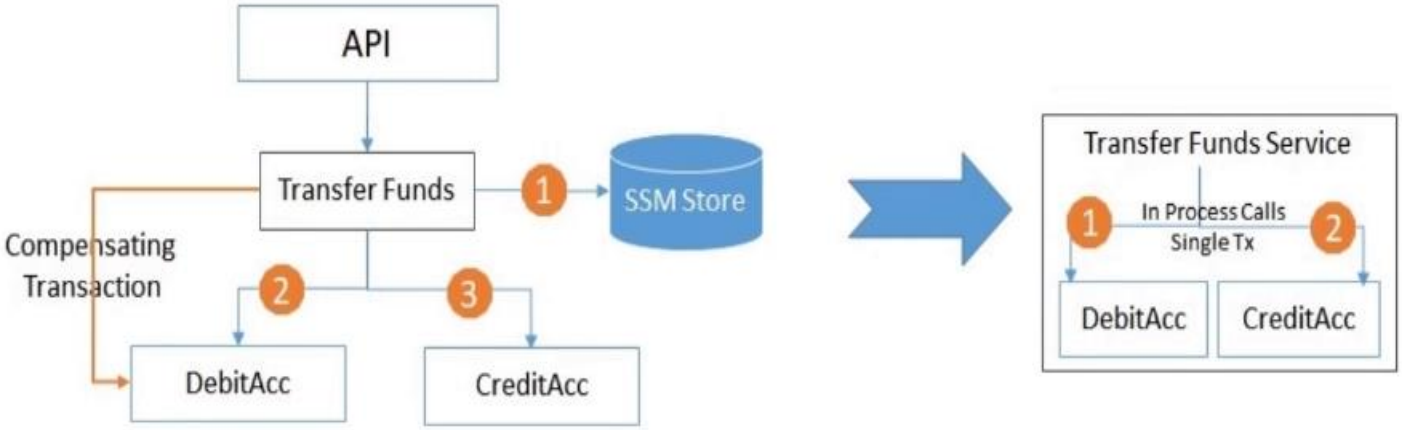
Asynchronous parallel

Asynchronous sequential

Orchestration using request/response

Collapsing the microservices

An option for when there seems to be a coupling between composite and its constituent microservice collapsing the services.



Are 3 services handled by 1 division, one team and hence in one bounded context