Microservices Architecture "The new normal"

Mohamed Sweelam

Senior Software Engineer

mohamedsweelam@fci.helwan.edu.eg

October 14, 2020

Objectives

1. Provide good Arabic content for the topic

Objectives

1. Provide good Arabic content for the topic
2. Overview of Microservices

Objectives

1. Provide good Arabic content for the topic
2. Overview of Microservices
3. Move step forwards towards recent cloud tools

Objectives

1. Provide good Arabic content for the topic
2. Overview of Microservices
3. Move step forwards towards recent cloud tools
4. Leave your fear, and let's do it

Contents

Definition

Monolithic

A monolithic application is self-contained, and independent from other computing applications. The design philosophy is that the application is responsible not just for a particular task, but can perform every step needed to complete a particular function.

Microservices

Microservices is a software development technique that arranges an application as a collection of loosely coupled services.

https://en.wikipedia.org/wiki/Monolithic_application
https://en.wikipedia.org/wiki/Microservices

Why Microservices?

1. Unit design
   - The application consists of loosely coupled services.
   - Each service supports a single business task.

Why Microservices?

1. Unit design
   - The application consists of loosely coupled services.
   - Each service supports a single business task.
2. Flexibility
   - Each microservice can be developed using a programming language and framework that best suits.

Why Microservices?

1. Unit design
   - The application consists of loosely coupled services.
   - Each service supports a single business task.
2. Flexibility
   - Each microservice can be developed using a programming language and framework that best suits.
3. Maintainability
   - Simple, focused, and independent. So the application is easier to maintain.

Why Microservices?

1. Unit design
   - The application consists of loosely coupled services.
   - Each service supports a single business task.
2. Flexibility
   - Each microservice can be developed using a programming language and framework that best suits.
3. Maintainability
   - Simple, focused, and independent. So the application is easier to maintain.
4. Resiliency
   - The application functionality is distributed across multiple services.
   - If a microservice fails, the functionality offered by the others continues to be available.

Why Microservices?

1. Unit design
   - The application consists of loosely coupled services.
   - Each service supports a single business task.
2. Flexibility
   - Each microservice can be developed using a programming language and framework that best suits.
3. Maintainability
   - Simple, focused, and independent. So the application is easier to maintain.
4. Resiliency
   - The application functionality is distributed across multiple services.
   - If a microservice fails, the functionality offered by the others continues to be available.
5. Scalability
   - Each microservice can be scaled independently of the other services.

Microservices vs Monolithic

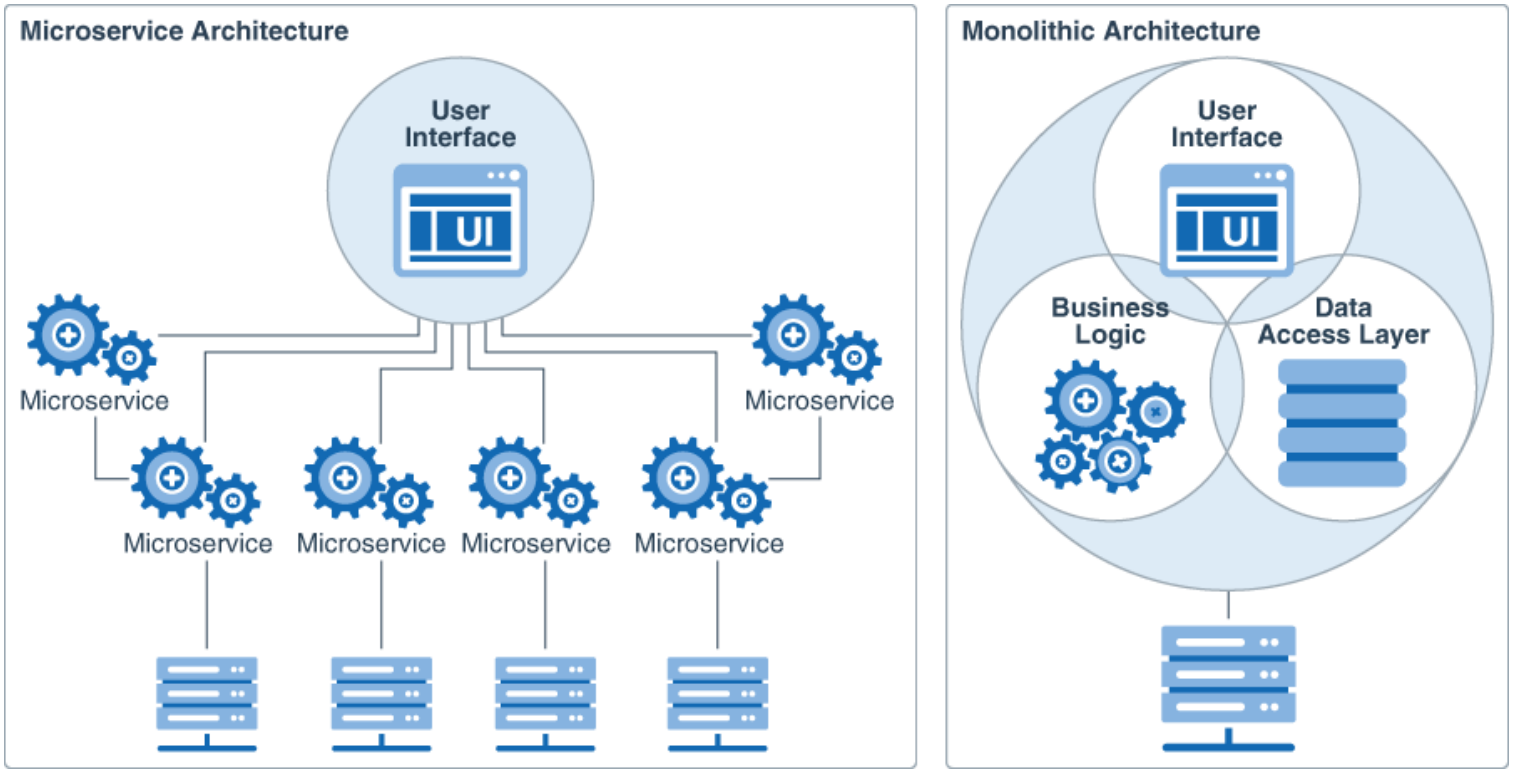


Figure: mivroservices vs monolithic

https://docs.oracle.com/en/solutions/learn-architect-microservice/index.html

Closer Look

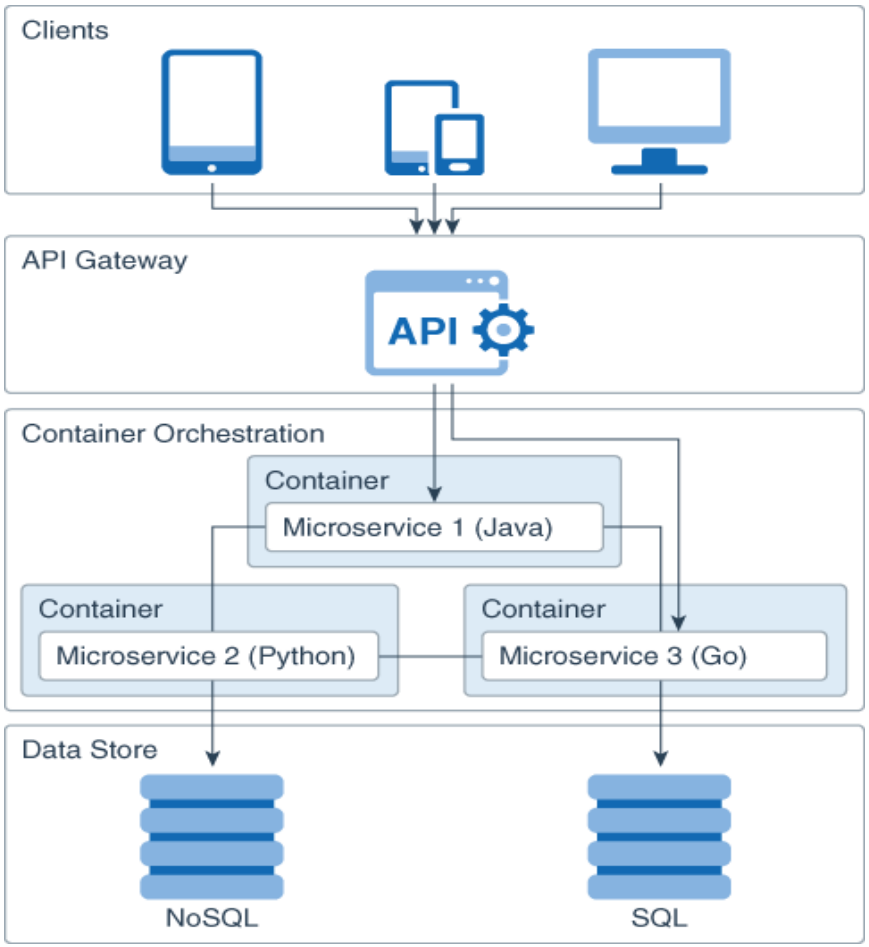


Figure: Microservices In Depth

https://docs.oracle.com/en/solutions/learn-architect-microservice/index.html

Microservices Architecture

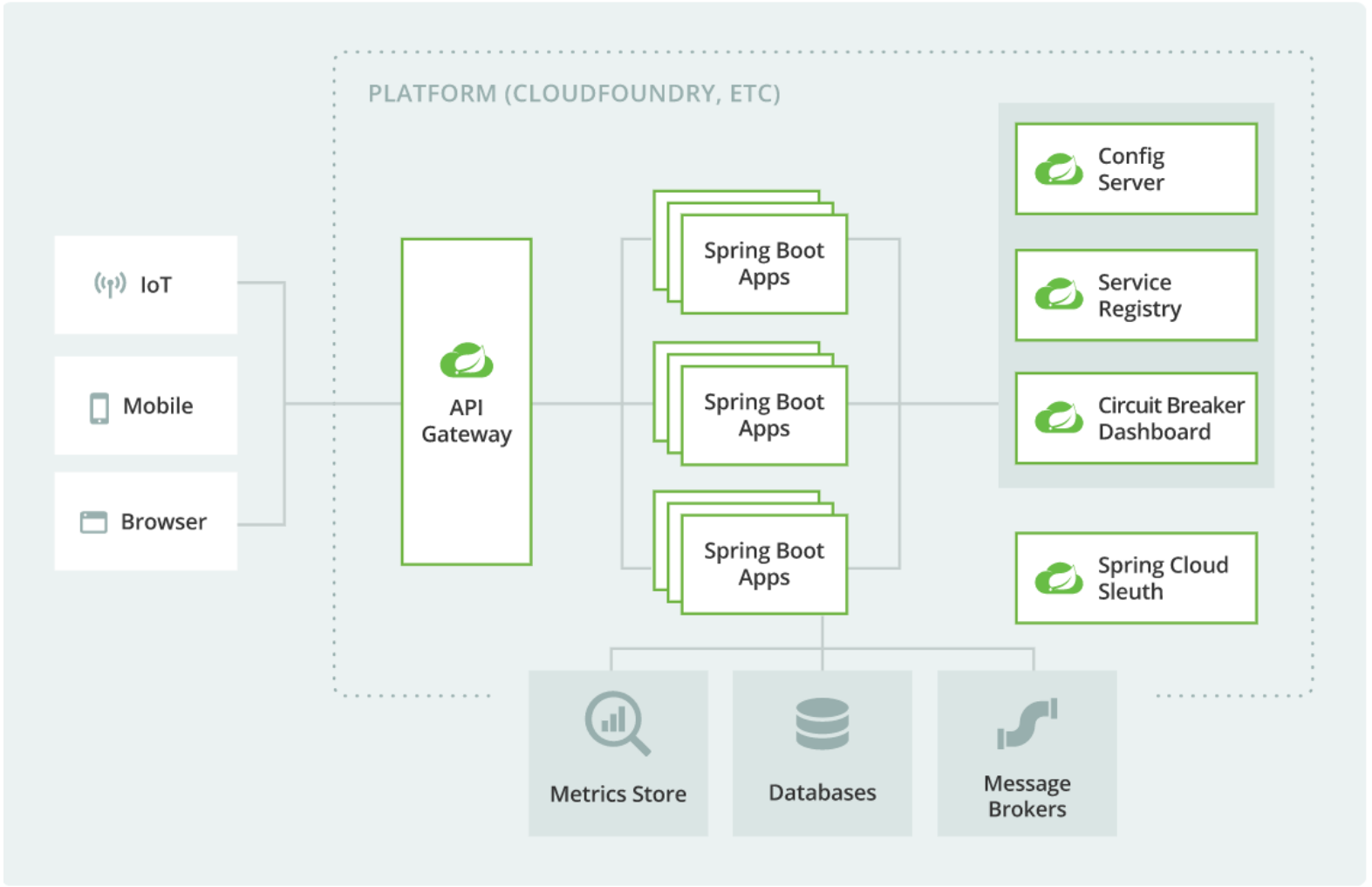
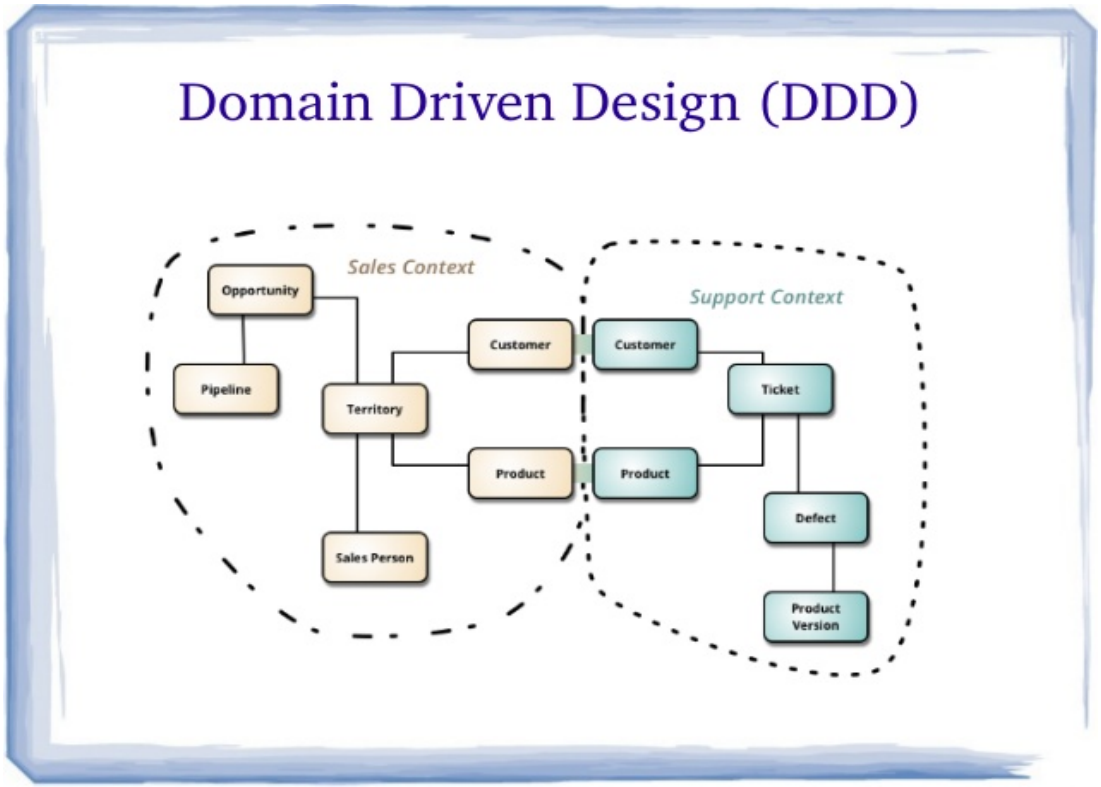


Figure: Microservices with Spring Cloud

Microservice characteristics

Single Responsibility

- Business Boundary
- Function Boundary



https://martinfowler.com/bliki/BoundedContext.html

Communication Design

HTTP communication

Also known as **Synchronous communication**, the calls between services is a viable option for **service-to-service** via REST API.

Message communication

Also known as **Asynchronous communication**, the services push messages to a message broker that other services subscribe to.

Event-driven communication

Another type of **Asynchronous communication**, the services does not need to know the common message structure. Communication between services takes place via events that individual services produce.

https://blog.logrocket.com/methods-for-microservice-communication

HTTP communication

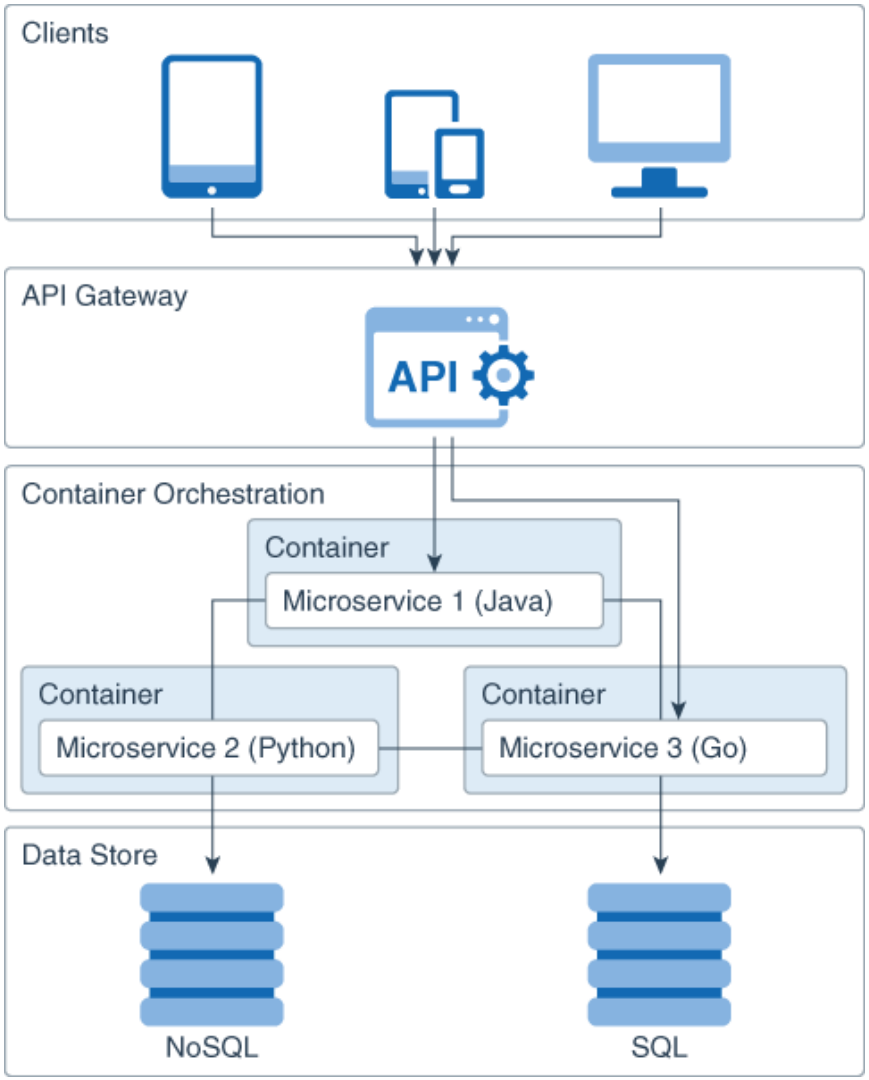


Figure: Synchronous calls

Event-driven communication

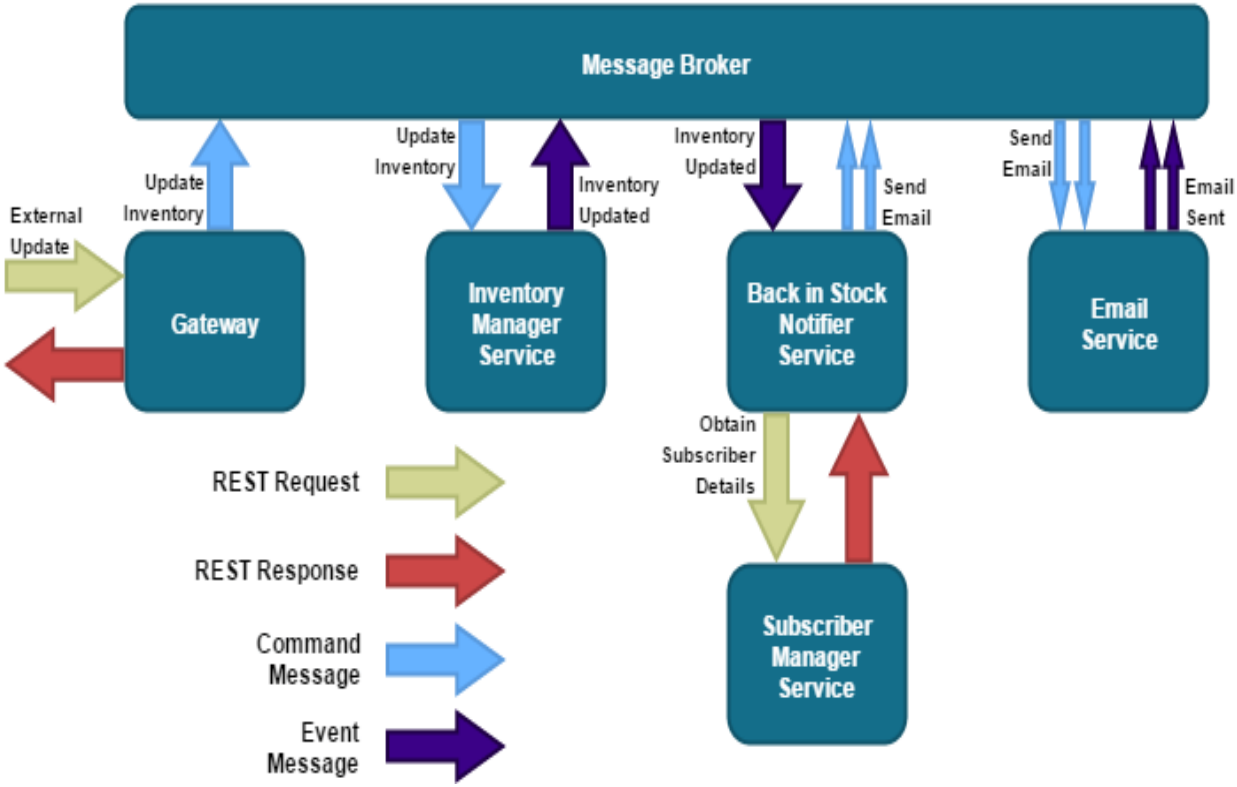


Figure: Asynchronous calls

https://capgemini.github.io/architecture/is-rest-best-microservices

Why not SOAP?

It is possible to build a microservices-based architecture using SOAP which uses HTTP. But:

- it only uses POST messages to transfer data to a server.
- SOAP lacks concepts such as HATEOAS that enable relationships between microservices to be handled flexibly.
- The interfaces have to be completely defined by the server and known on the client.

Microservices; Flexible Software Architecture. "Eberhard Wolff"

API Gateway

API Gateway is a tool that makes it easy for developers to create(1), publish(2), maintain(3), monitor(4), and secure(5) APIs at any scale. APIs act as the "front door" for applications to access data, business logic, or functionality from your backend services.

Figure: Amazon Gateway

https://aws.amazon.com/api-gateway/

Orchestration and API Gateway cont...

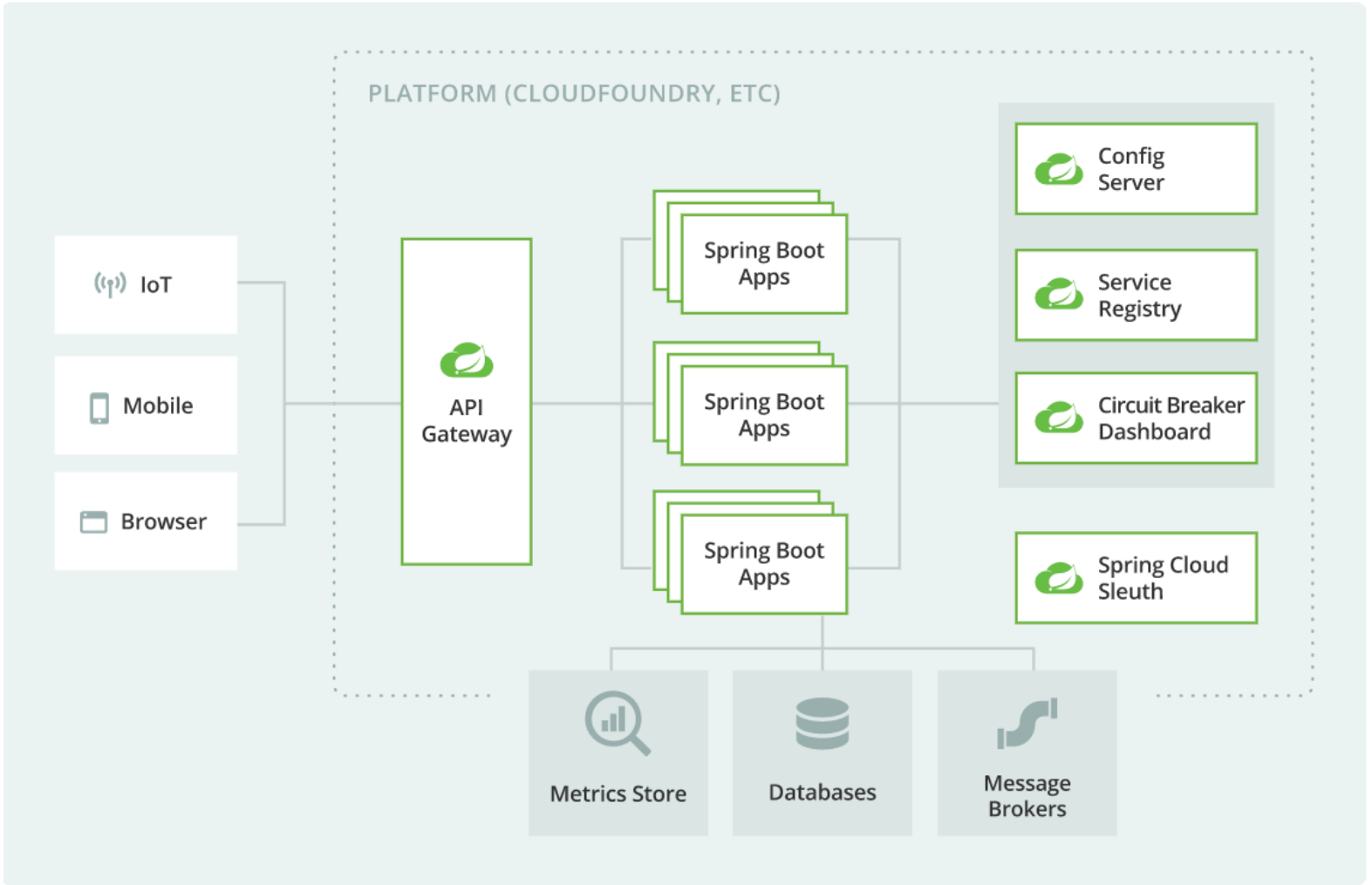


Figure: Microservices with Spring Cloud

Available Market Options

Figure: API Gateway Products

Service Discovery

Problem

In any distributed architecture, we need to find the physical address of where a machine is located.

Solution

Using service discovery, a service can register itself when it is up and healthy. By using such technology you can achieve:

1. Load balanced
   - dynamically load balance requests across all service instances to ensure that the service invocations are spread across all the service instances managed by it.

Service Discovery

Problem

In any distributed architecture, we need to find the physical address of where a machine is located.

Solution

Using service discovery, a service can register itself when it is up and healthy. By using such technology you can achieve:

1. Load balanced
   - dynamically load balance requests across all service instances to ensure that the service invocations are spread across all the service instances managed by it.
2. Resilient
   - client should "cache" service information locally. Local caching allows for gradual degradation of the service discovery feature so that if service discovery service does become unavailable, applications can still function and locate the services based on the information maintained in its local cache.

Service Discovery

Problem

In any distributed architecture, we need to find the physical address of where a machine is located.

Solution

Using service discovery, a service can register itself when it is up and healthy. By using such technology you can achieve:

1. Load balanced
   - dynamically load balance requests across all service instances to ensure that the service invocations are spread across all the service instances managed by it.
2. Resilient
   - client should "cache" service information locally. Local caching allows for gradual degradation of the service discovery feature so that if service discovery service does become unavailable, applications can still function and locate the services based on the information maintained in its local cache.
3. Fault-tolerant
   - detect when a service instance isn't healthy and remove the instance from the list of available services.

Service Discovery with Gateway

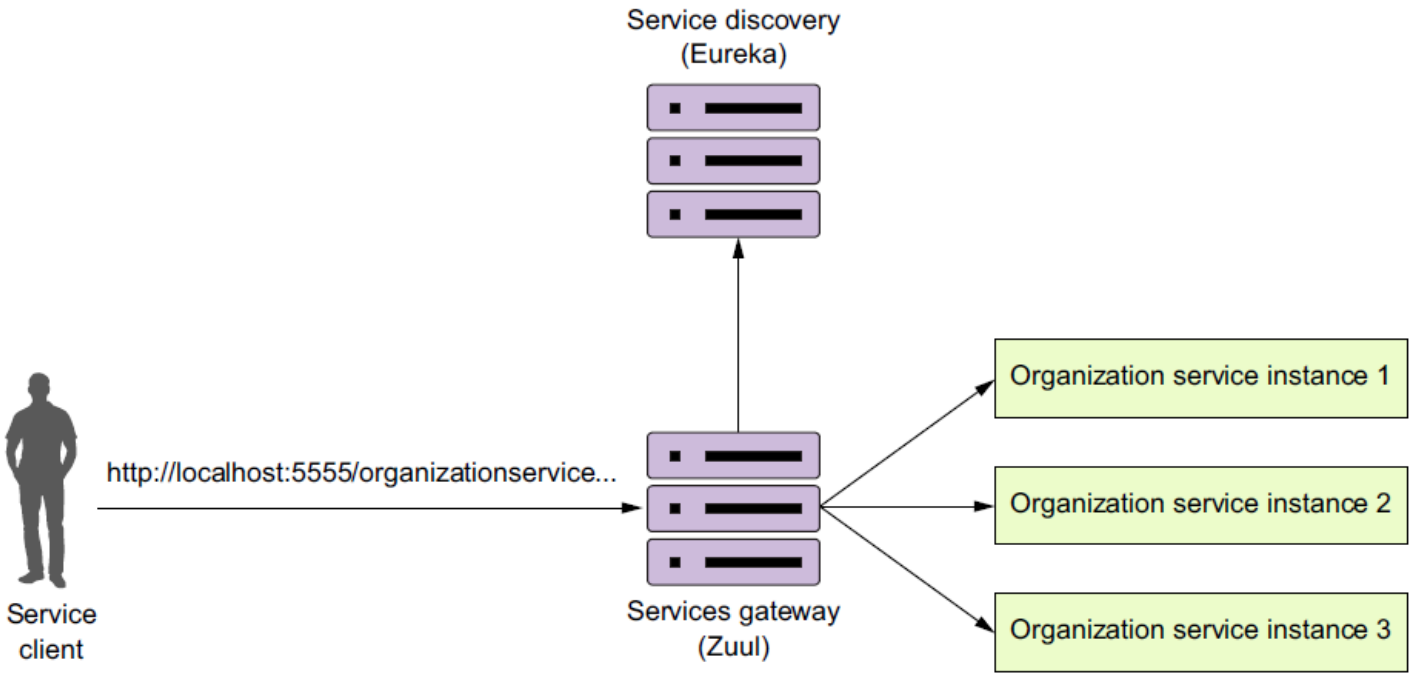


Figure: Service Registry and Gateway

Spring Microservices in Action, "JOHN CARNELL"

Available Market Options

Figure: Service Registry Products

The Twelve-Factor App

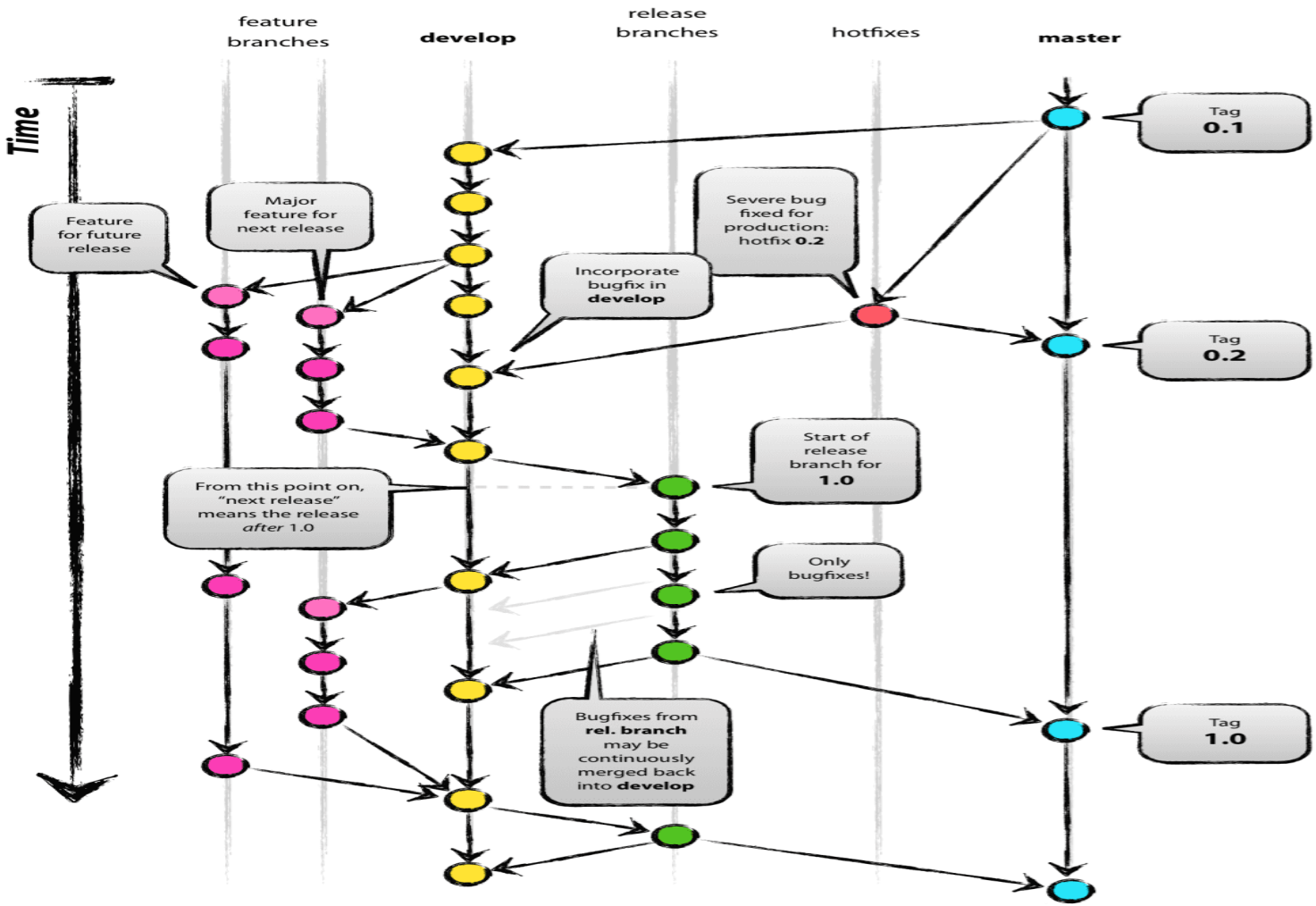
The Twelve-Factor App methodology is a methodology for building **software-as-a-service** applications

1. Codebase
2. Dependencies
3. Config
4. Backing services
5. Build, release, run
6. Processes
7. Port binding
8. Concurrency
9. Disposability
10. Dev/prod parity
11. Logs
12. Admin processes

https://12factor.net

The Twelve-Factor App

1. Codebase
   - One codebase tracked in revision control, many deploys



The Twelve-Factor App

2. Dependencies
   - Explicitly declare and isolate dependencies
   - Consider the magic key **Portability**

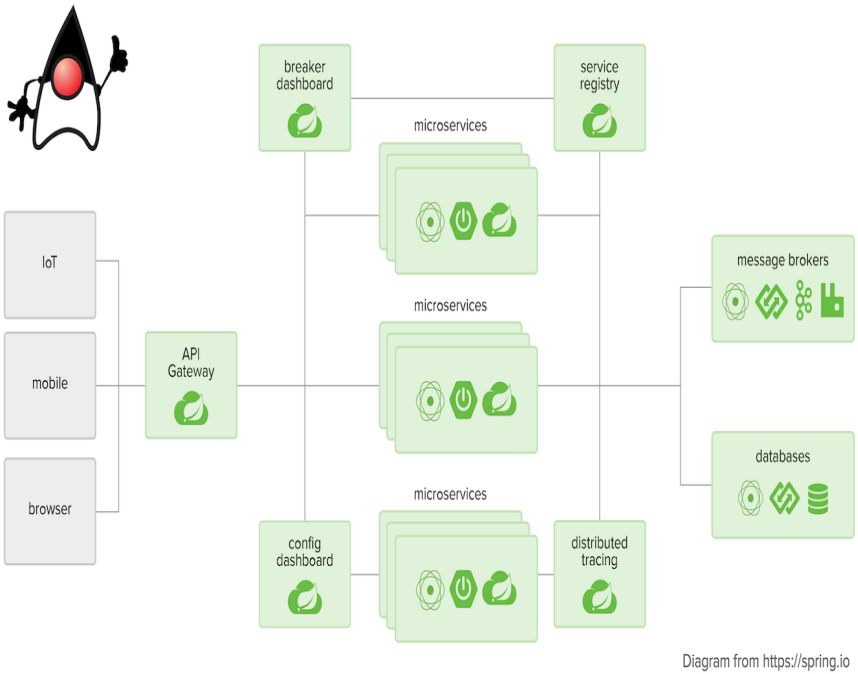


Diagram from https://spring.io

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
        <version>2.3.0.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zipkin</artifactId>
    </dependency>
    <dependency...>
    <dependency...>
    <dependency...>
    <dependency...>
</dependencies>
```

Never ever depend on operating system

The Twelve-Factor App

3. Configuration
   - Config is what is changed from environment to another

The Twelve-Factor App

3. Configuration
   - Config is what is changed from environment to another
   - Config should be provided by the environment, not the code

The Twelve-Factor App

3. Configuration
   - Config is what is changed from environment to another
   - Config should be provided by the environment, not the code

   - **Credentials are not configuration, but secrets**
     - Never ever store credentials in code
     - Don't save credentials in PLAINTEXT with config, but hashed

The Twelve-Factor App

3. Configuration
   - Config is what is changed from environment to another
   - Config should be provided by the environment, not the code
   - **Credentials are not configuration, but secrets**
     - Never ever store credentials in code
     - Don't save credentials in PLAINTEXT with config, but hashed
   - **Configuration in Legacy system is a challenge**
     - Unlike missing dependencies, System will not immediately crashed if configuration is missed
     - Give attention to URLs in legacy code

Externalized and Dynamic Configurations

Problem

Configurations will vary from environment to another, How to manage them?

Solution

Centralize your configuration

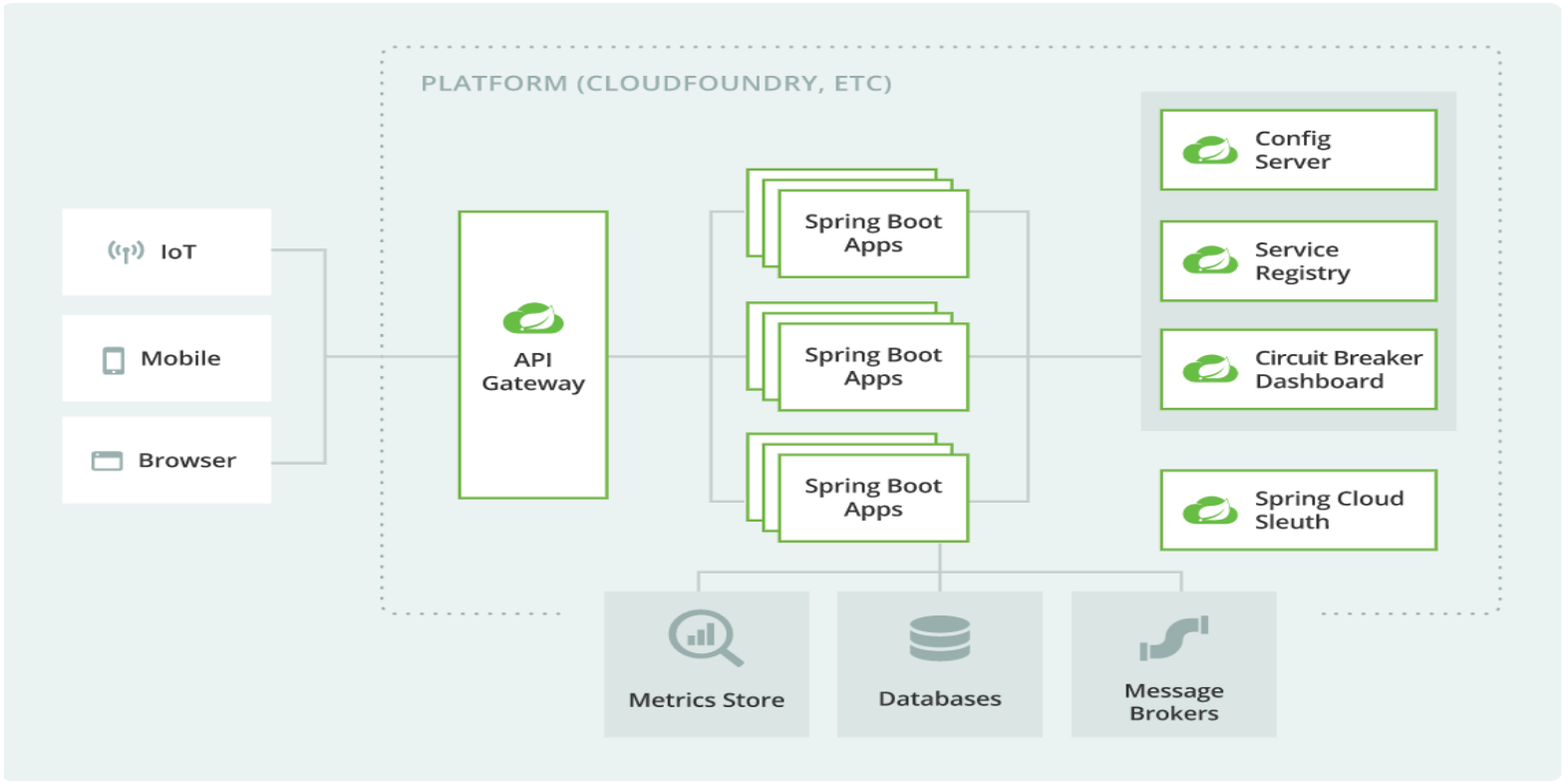


Figure: Microservices with Spring Cloud

Available Market Options

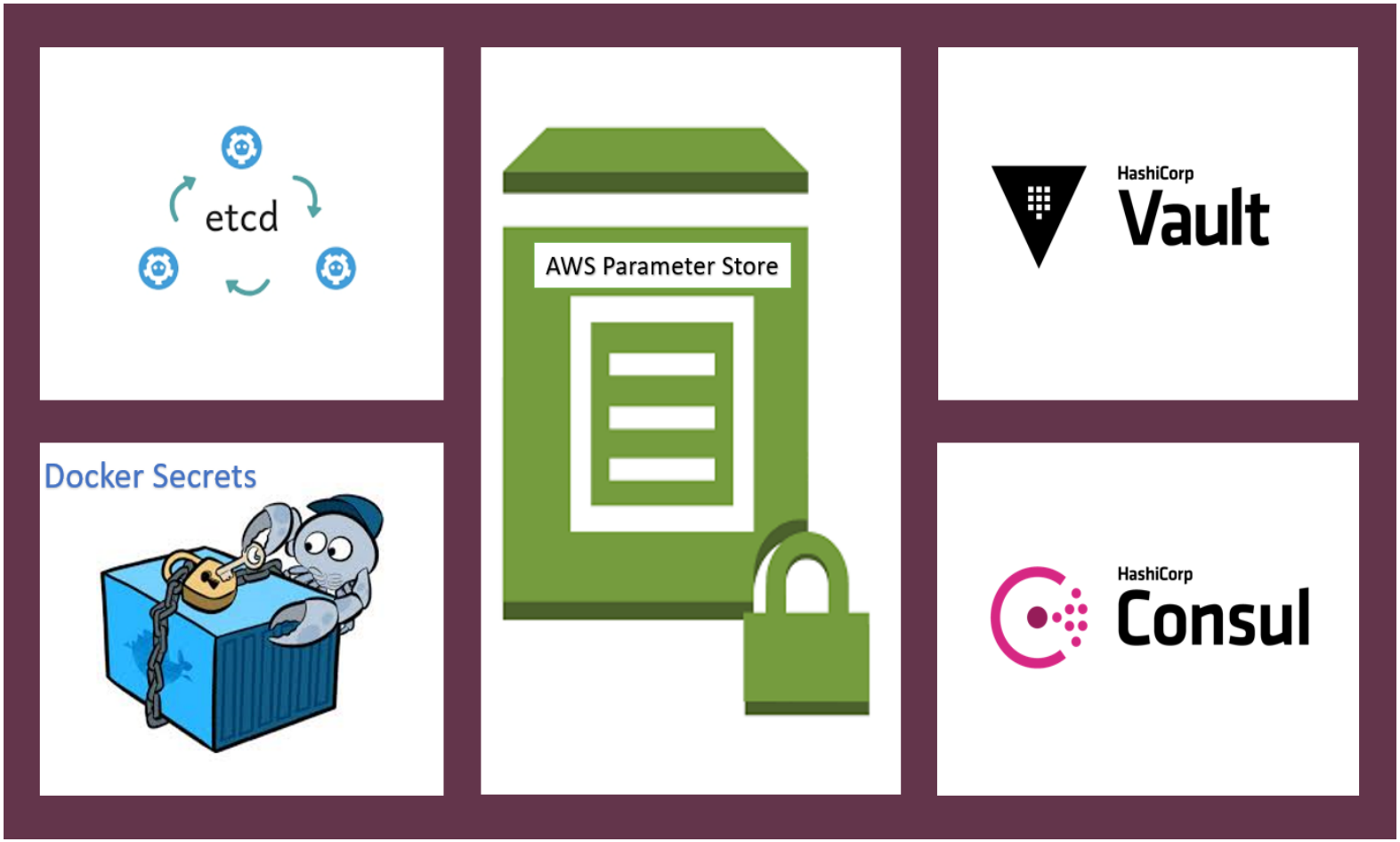


Figure: Popular Config Stores

Circuit Breaker Pattern

Problem

- One of the big differences between in-memory calls and remote calls is that remote calls can fail, or hang without a response until some timeout limit is reached.

Solution

Fault Tolerance

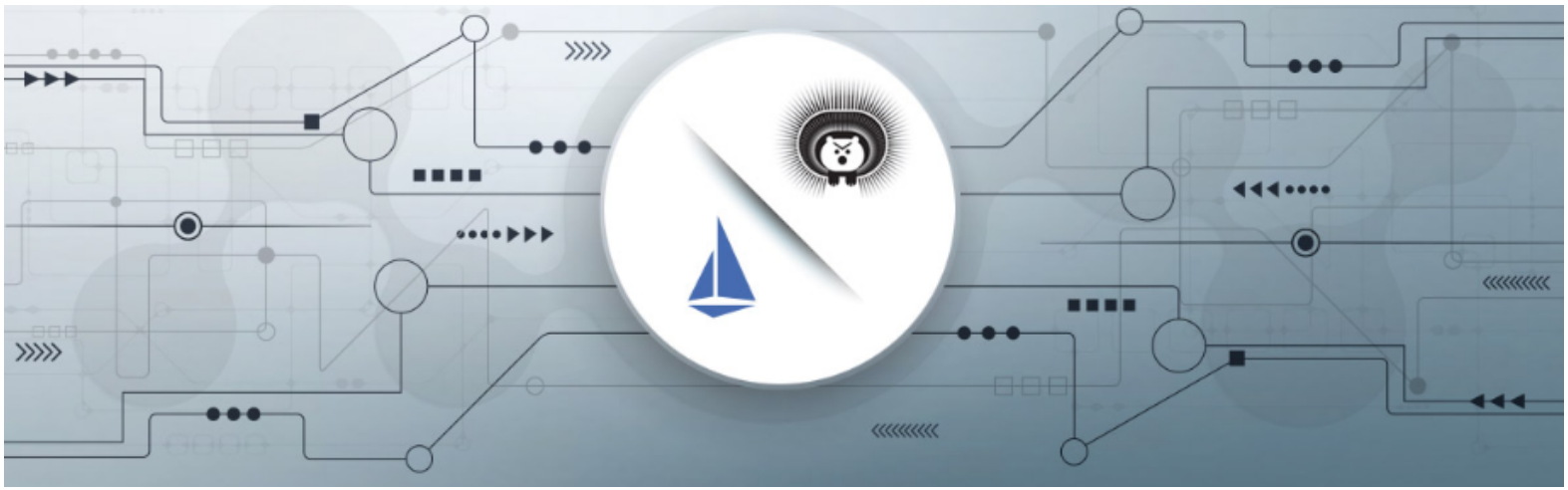


Figure: Circuit Breaker

Circuit Breaker Pattern

Problem

- One of the big differences between in-memory calls and remote calls is that remote calls can fail, or hang without a response until some timeout limit is reached.
- What's worse if you have many callers on a unresponsive supplier, then you can run out of critical resources leading to cascading failures across multiple systems.

Solution

Fault Tolerance

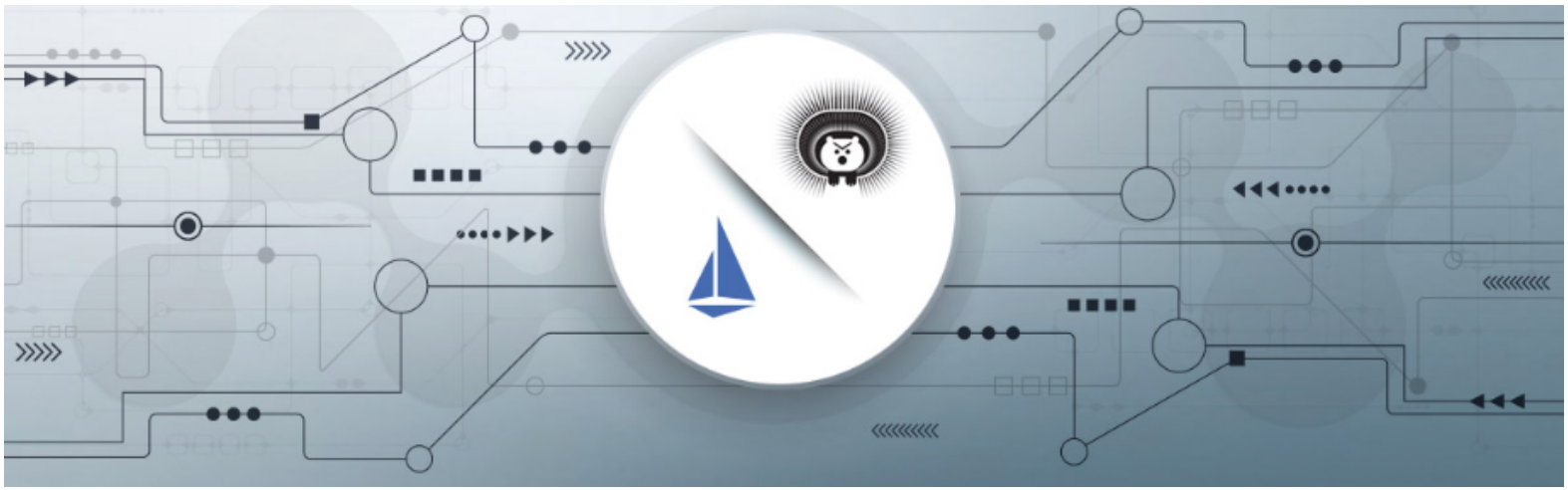


Figure: Circuit Breaker

The Twelve-Factor App

4. Backing Service
   - A backing service is any service the app consumes over the network as part of its normal operation.
   - Examples include data stores, queueing, and caching systems

The Twelve-Factor App

4. Backing Service
   - A backing service is any service the app consumes over the network as part of its normal operation.
   - Examples include data stores, queueing, and caching systems

   How to achieve that?
   - To the App, this is just normal service.

The Twelve-Factor App

4. Backing Service
   - A backing service is any service the app consumes over the network as part of its normal operation.
   - Examples include data stores, queueing, and caching systems

   How to achieve that?
   - To the App, this is just normal service.
   - Should be treating as **attached resource**.

The Twelve-Factor App

4. Backing Service
   - A backing service is any service the app consumes over the network as part of its normal operation.
   - Examples include data stores, queueing, and caching systems

   How to achieve that?
   - To the App, this is just normal service.
   - Should be treating as **attached resource**.
   - Even with third party services like SMTP providers, it is just a resource.

The Twelve-Factor App

4. Backing Service
   - A backing service is any service the app consumes over the network as part of its normal operation.
   - Examples include data stores, queueing, and caching systems

   How to achieve that?
   - To the App, this is just normal service.
   - Should be treating as **attached resource**.
   - Even with third party services like SMTP providers, it is just a resource.
   - A good way for this is as mentioned in config talk.

```
1  public void useHardCodedResources(long userId) {
2    final String URL = "https://10.20.30.90/api/user/";
3    User user = restTemplate.getObject(URL + userId, User.class);
4    ...
5  }
6
7  public void useAttachedResources(String resourceConfig, long userId) {
8    User user = restTemplate.getObject(resourceConfig + userId, User.class);
9    ...
10 }
```

The Twelve-Factor App

4. Backing Service
   - A backing service is any service the app consumes over the network as part of its normal operation.
   - Examples include data stores, queueing, and caching systems

   How to achieve that?
   - To the App, this is just normal service.
   - Should be treating as **attached resource**.
   - Even with third party services like SMTP providers, it is just a resource.
   - A good way for this is as mentioned in config talk.

```
1  public void useHardCodedResources(long userId) {
2    final String URL = "https://10.20.30.90/api/user/";
3    User user = restTemplate.getObject(URL + userId, User.class);
4    ...
5  }
6
7  public void useAttachedResources(String resourceConfig, long userId) {
8    User user = restTemplate.getObject(resourceConfig + userId, User.class);
9    ...
10 }
```

   - This way you can switch between services smoothly on different environments via configurations provided by the environment.

Development Culture

Usually building a microservices software requires good development strategy to follow, and the most valuable one is

Development Culture

Usually building a microservices software requires good development strategy to follow, and the most valuable one is

Agile

1. Working in small planned sprints period

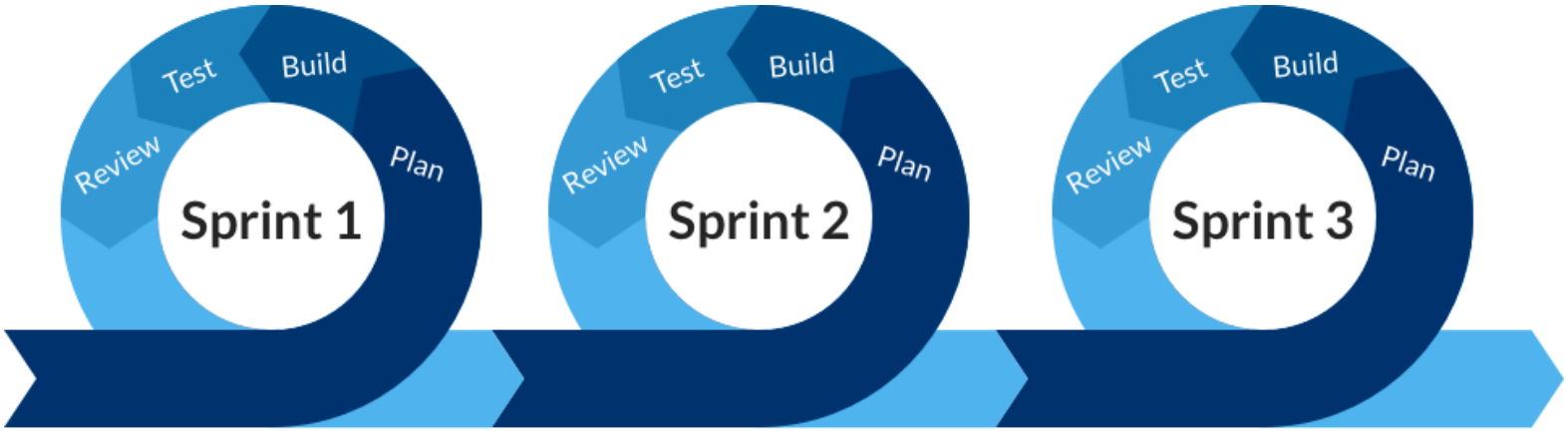


Figure: Agile Methodology

Development Culture

Usually building a microservices software requires good development strategy to follow, and the most valuable one is

Agile

1. Working in small planned sprints period
2. Each sprint is 2 to 4 weeks

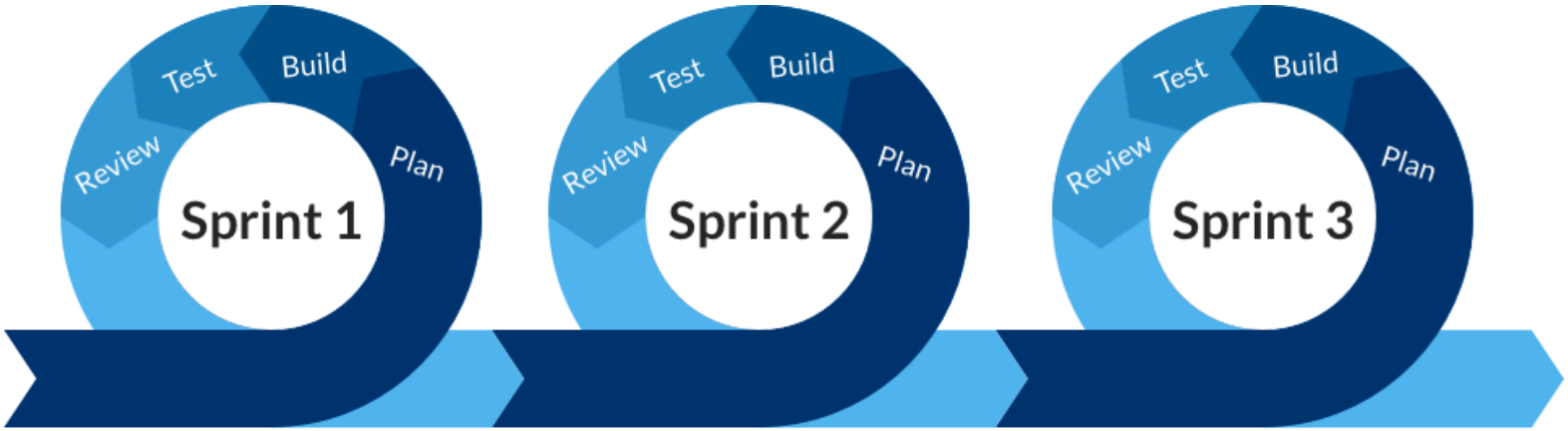


Figure: Agile Methodology

Development Culture

Usually building a microservices software requires good development strategy to follow, and the most valuable one is

Agile

1. Working in small planned sprints period
2. Each sprint is 2 to 4 weeks
3. Stand-up meeting for 15 mins to discuss challenges and day to day work

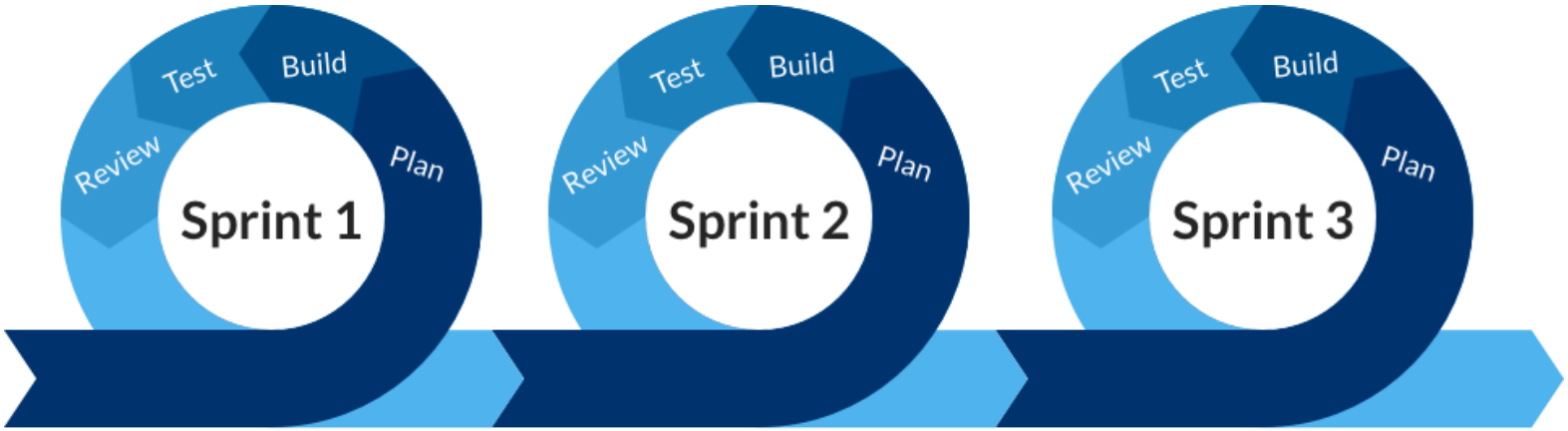


Figure: Agile Methodology

Development Culture

Usually building a microservices software requires good development strategy to follow, and the most valuable one is

Agile

1. Working in small planned sprints period
2. Each sprint is 2 to 4 weeks
3. Stand-up meeting for 15 mins to discuss challenges and day to day work
4. Feedback and Retrospective meetings after each sprint is very important

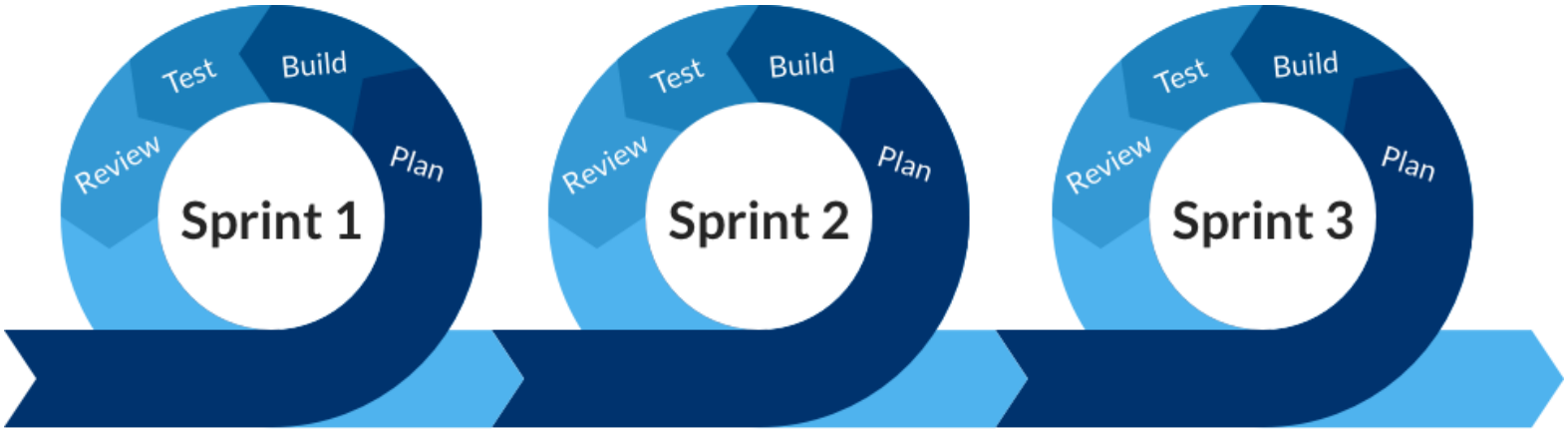
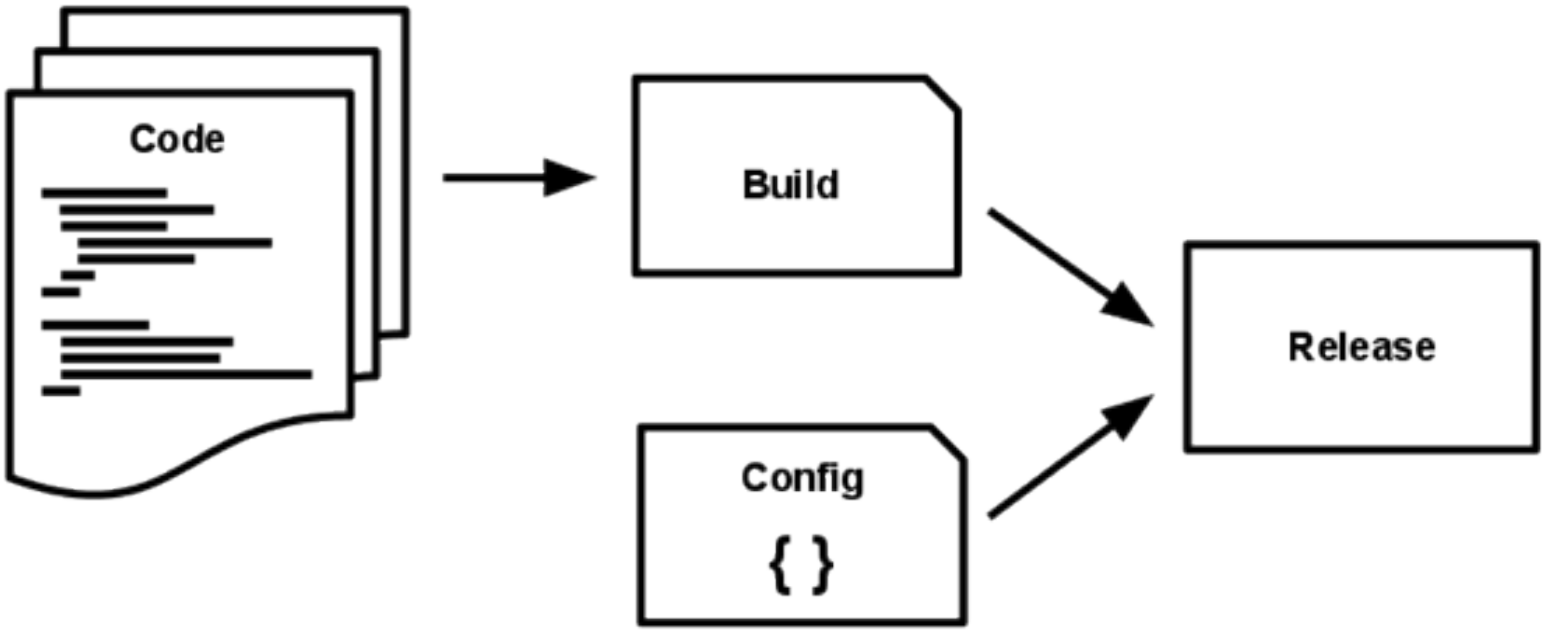


Figure: Agile Methodology

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
   Strictly separate build and run stages. In Microservices, this process is always related directly to development culture; How to be sure each time work will be delivered correctly?



The Twelve-Factor App

5. Build, Release and Run (CI & CD)
    - Build
        - The process where you convert the code into executable bundles.

Continuous Delivery, “JEZ HUMBLE“ & “DEVID FARLEY“

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
   - Build
     - The process where you convert the code into executable bundles.
     - Each build should be correct, and to achieve that, you have to use Continues Integration flow.

Continuous Delivery, “JEZ HUMBLE“ & “DEVID FARLEY“

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
   - Build
     - The process where you convert the code into executable bundles.
     - Each build should be correct, and to achieve that, you have to use Continues Integration flow.
     - Each team member must write the unit and integration tests which will be executed before and after the build.

Continuous Delivery, “JEZ HUMBLE“ & “DEVID FARLEY“

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
   - Build
     - The process where you convert the code into executable bundles.
     - Each build should be correct, and to achieve that, you have to use Continues Integration flow.
     - Each team member must write the unit and integration tests which will be executed before and after the build.
     - Tests which are written, must be strictly reviewed.

Continuous Delivery, “JEZ HUMBLE“ & “DEVID FARLEY“

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
   - Build
     - The process where you convert the code into executable bundles.
     - Each build should be correct, and to achieve that, you have to use Continues Integration flow.
     - Each team member must write the unit and integration tests which will be executed before and after the build.
     - Tests which are written, must be strictly reviewed.
     - It is nice today to use TDD while developing to gain more experience and understanding.

Continuous Delivery, “JEZ HUMBLE“ & “DEVID FARLEY“

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
   - Build
     - The process where you convert the code into executable bundles.
     - Each build should be correct, and to achieve that, you have to use Continues Integration flow.
     - Each team member must write the unit and integration tests which will be executed before and after the build.
     - Tests which are written, must be strictly reviewed.
     - It is nice today to use TDD while developing to gain more experience and understanding.
     - Don't Comment Out Failing Tests

Continuous Delivery, "JEZ HUMBLE" & "DEVID FARLEY"

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
   - Build
     - The process where you convert the code into executable bundles.
     - Each build should be correct, and to achieve that, you have to use Continues Integration flow.
     - Each team member must write the unit and integration tests which will be executed before and after the build.
     - Tests which are written, must be strictly reviewed.
     - It is nice today to use TDD while developing to gain more experience and understanding.
     - Don't Comment Out Failing Tests
     - Don't push broken code.

Continuous Delivery, "JEZ HUMBLE" & "DEVID FARLEY"

5. Build, Release and Run (CI & CD)
   - Build

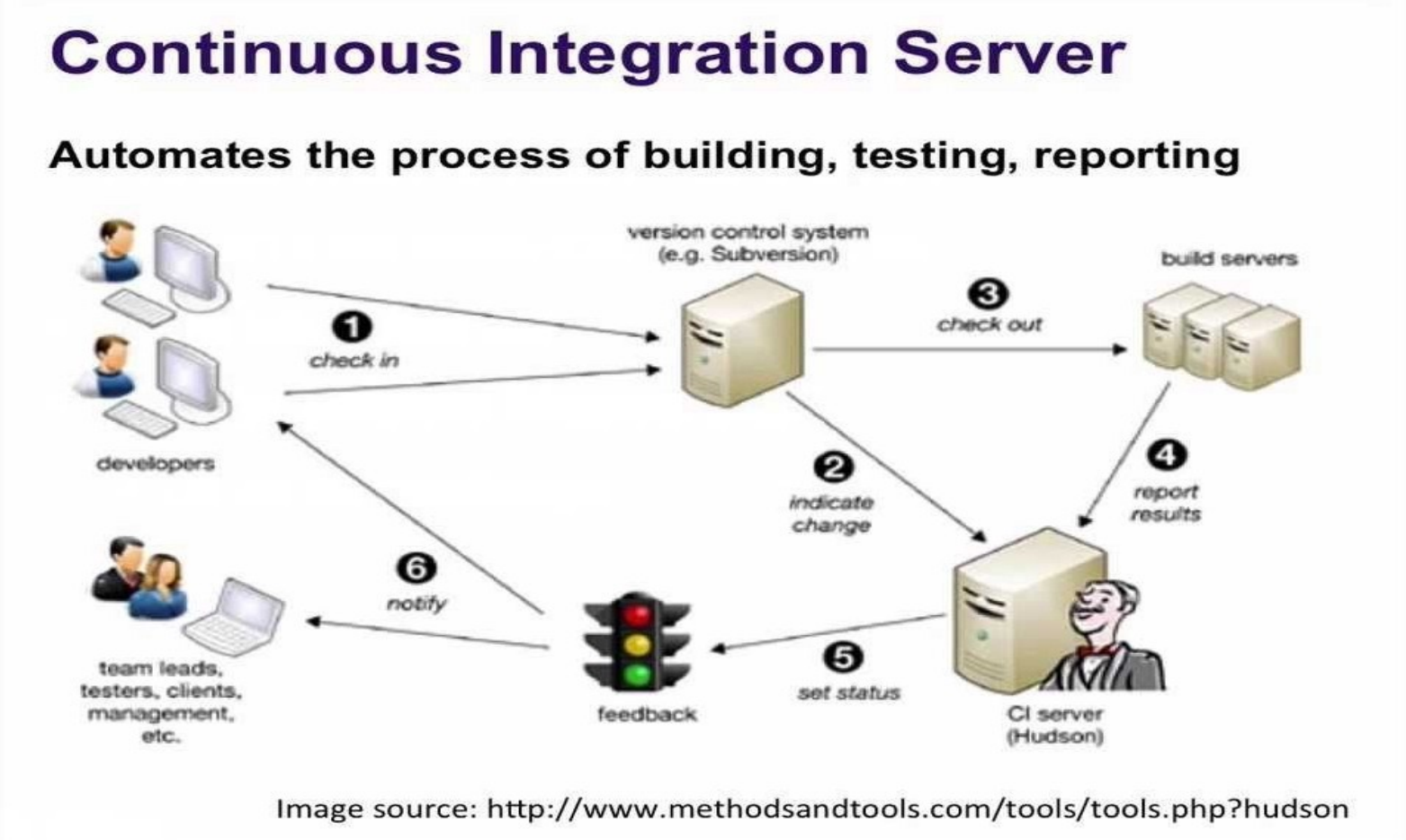


Figure: CI Workflow

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
    - Release
        - The process where you combine the build with deploy's config.

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
    - Release
        - The process where you combine the build with deploy's config.
        - If the tests passed , build stage is done; and move to release stage.

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
   - Release
     - The process where you combine the build with deploy's config.
     - If the tests passed , build stage is done; and move to release stage.
     - The resulting release contains both the build and the config and is ready for immediate execution in the execution environment.

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
   - Release
     - The process where you combine the build with deploy's config.
     - If the tests passed , build stage is done; and move to release stage.
     - The resulting release contains both the build and the config and is ready for immediate execution in the execution environment.
     - Every release should always have a unique release ID, such as a timestamp of the release (such as 2011-04-06-20:32:17) or an incrementing number (such as v100).

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
   - Release
     - The process where you combine the build with deploy's config.
     - If the tests passed , build stage is done; and move to release stage.
     - The resulting release contains both the build and the config and is ready for immediate execution in the execution environment.
     - Every release should always have a unique release ID, such as a timestamp of the release (such as 2011-04-06-20:32:17) or an incrementing number (such as v100).
     - Release are immutable, any change should be mapped with a new release.

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
   - Release
     - The process where you combine the build with deploy's config.
     - If the tests passed , build stage is done; and move to release stage.
     - The resulting release contains both the build and the config and is ready for immediate execution in the execution environment.
     - Every release should always have a unique release ID, such as a timestamp of the release (such as 2011-04-06-20:32:17) or an incrementing number (such as v100).
     - Release are immutable, any change should be mapped with a new release.
     - Using tagging, unique IDs and timestamp will be the only way to apply rollback if you want.

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
    - Run
        - also known as “runtime”, runs the app in the execution environment, by launching some set of the app’s processes against a selected release.

The Twelve-Factor App

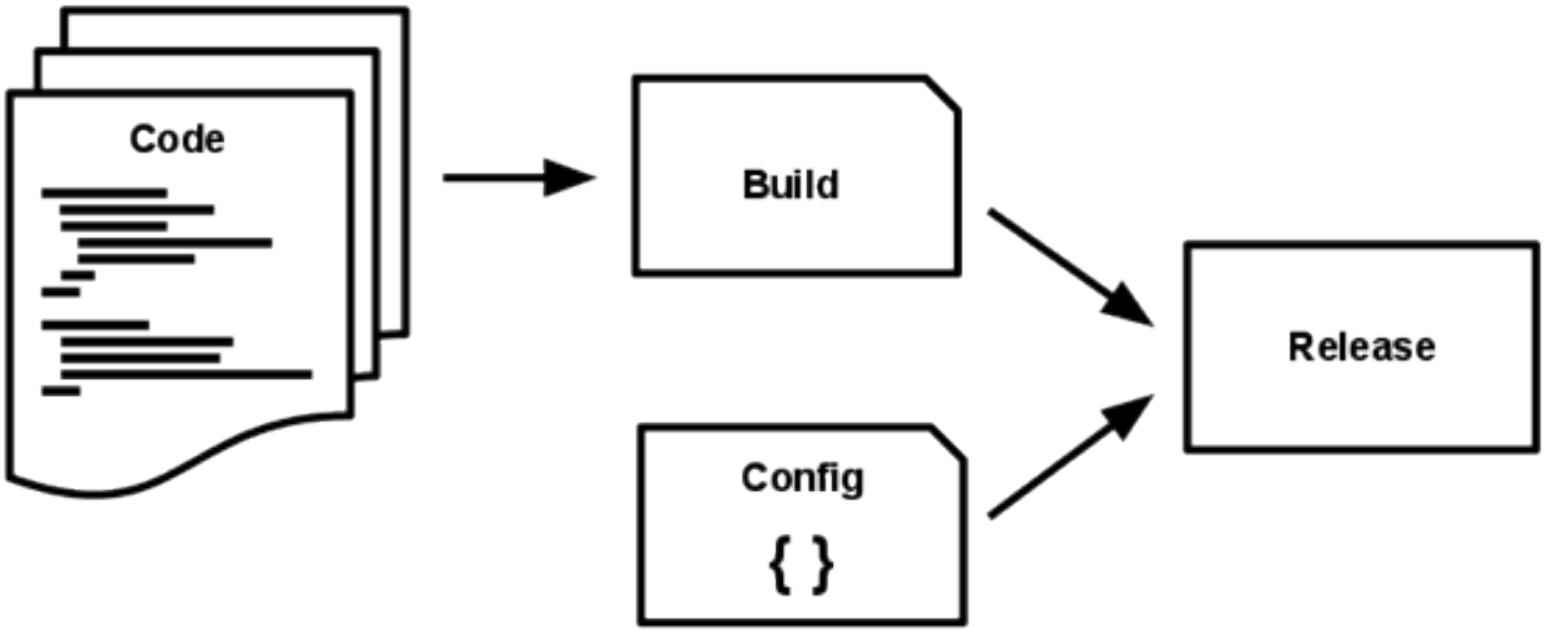
5. Build, Release and Run (CI & CD)
    - Run
        - also known as “runtime”, runs the app in the execution environment, by launching some set of the app’s processes against a selected release.
        - Builds are initiated by the app’s developers whenever new code is deployed. Runtime execution, by contrast, can happen automatically in cases such as a server reboot, or a crashed process being restarted by the process manager.

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
   - Run
     - also known as “runtime”, runs the app in the execution environment, by launching some set of the app’s processes against a selected release.
     - Builds are initiated by the app’s developers whenever new code is deployed. Runtime execution, by contrast, can happen automatically in cases such as a server reboot, or a crashed process being restarted by the process manager.
     - the run stage should be kept to as few moving parts as possible, since problems that prevent an app from running can cause it to break in the middle of the night when no developers are on hand.

The Twelve-Factor App

5. Build, Release and Run (CI & CD)
   Strictly separate build and run stages. In Microservices, this process is always related directly to development culture; How to be sure each time work will be delivered correctly?



The Twelve-Factor App

6. Processes

   Execute the app as one or more **stateless** processes

   - Twelve-factor processes are stateless and share-nothing, and indeed Microservices are the same.

The Twelve-Factor App

6. Processes

   Execute the app as one or more **stateless** processes

   - Twelve-factor processes are stateless and share-nothing, and indeed Microservices are the same.
   - Any data that needs to persist must be stored in a stateful backing service, typically a database.

The Twelve-Factor App

6. Processes

   Execute the app as one or more **stateless** processes

   - Twelve-factor processes are stateless and share-nothing, and indeed Microservices are the same.
   - Any data that needs to persist must be stored in a stateful backing service, typically a database.
   - Asset packagers like *django-assetpackager* use the filesystem as a cache for compiled assets. This process of compiling should be done during the **build** stage.

The Twelve-Factor App

6. Processes

   Execute the app as one or more **stateless** processes

   - Twelve-factor processes are stateless and share-nothing, and indeed Microservices are the same.
   - Any data that needs to persist must be stored in a stateful backing service, typically a database.
   - Asset packagers like *django-assetpackager* use the filesystem as a cache for compiled assets. This process of compiling should be done during the **build** stage.
   - For example, using filesystems and caching memory is a violation of twelve factor.

The Twelve-Factor App

6. Processes

   Execute the app as one or more **stateless** processes

   - Twelve-factor processes are stateless and share-nothing, and indeed Microservices are the same.
   - Any data that needs to persist must be stored in a stateful backing service, typically a database.
   - Asset packagers like *django-assetpackager* use the filesystem as a cache for compiled assets. This process of compiling should be done during the **build** stage.
   - For example, using filesystems and caching memory is a violation of twelve factor.
   - Session state data is a good candidate for a datastore that offers time-expiration, such as Memcached or Redis.

The Twelve-Factor App

6. Processes

   Execute the app as one or more **stateless** processes

   - Twelve-factor processes are stateless and share-nothing, and indeed Microservices are the same.
   - Any data that needs to persist must be stored in a stateful backing service, typically a database.
   - Asset packagers like *django-assetpackager* use the filesystem as a cache for compiled assets. This process of compiling should be done during the **build** stage.
   - For example, using filesystems and caching memory is a violation of twelve factor.
   - Session state data is a good candidate for a datastore that offers time-expiration, such as Memcached or Redis.
   - It is important to know also that, Microservices must be totally stateless, and there is no way to keep state for backend services http requests.

The Twelve-Factor App

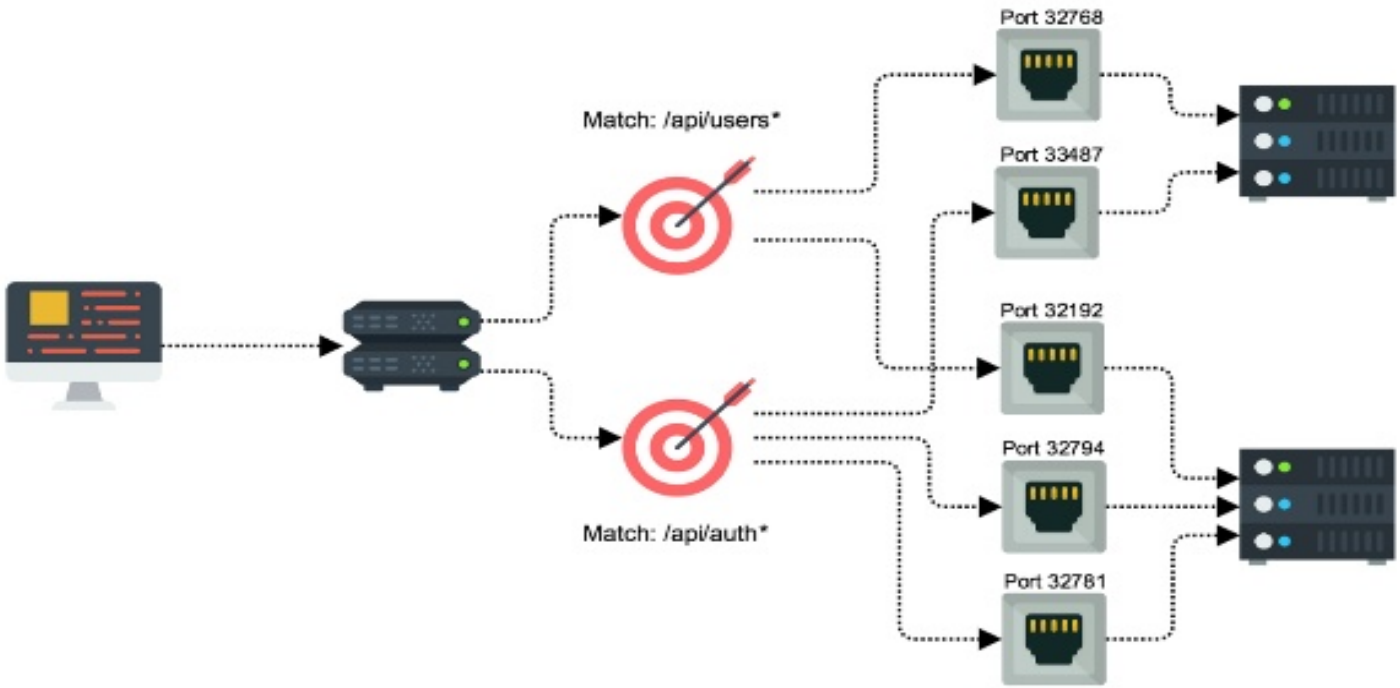
7. Port Binding
   Export services via port binding

The Twelve-Factor App

7 Port Binding

Export services via port binding

- The Microservice is self-contained and does not rely on injection of a webserver to create a web-facing service.

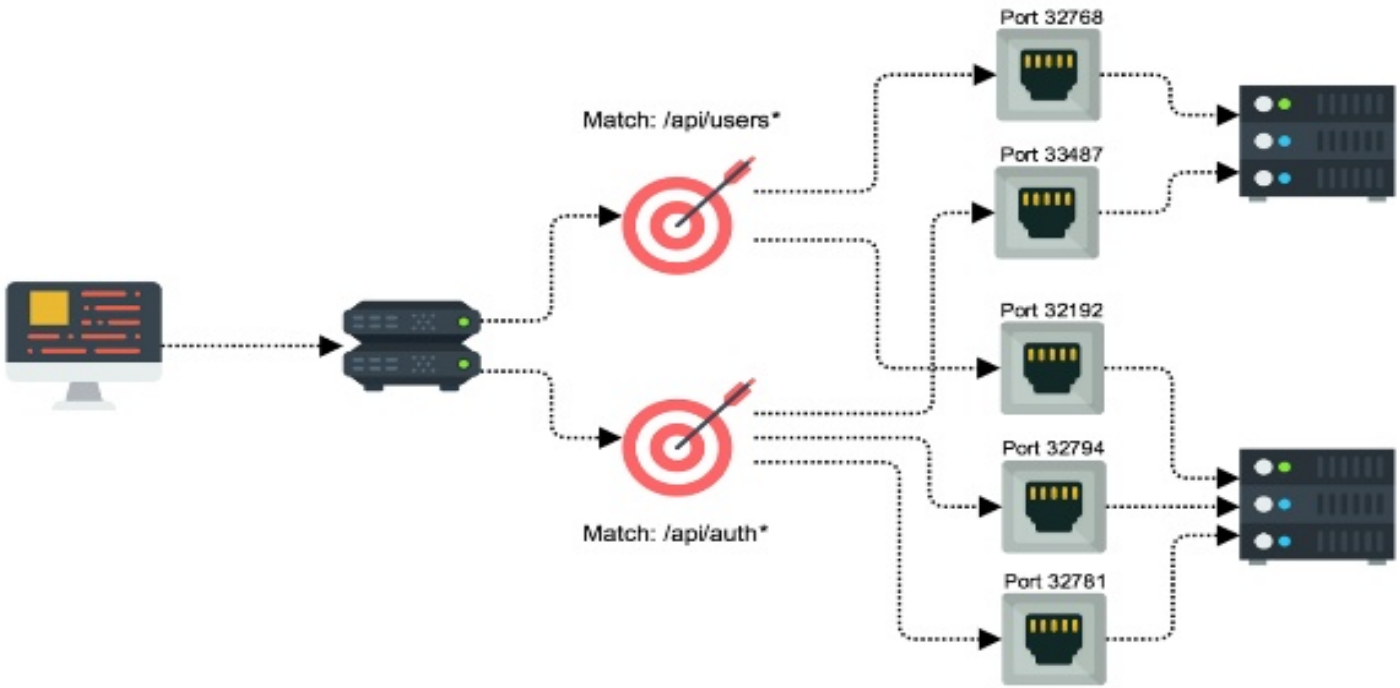


The Twelve-Factor App

7 Port Binding

Export services via port binding

- The Microservice is self-contained and does not rely on injection of a webserver to create a web-facing service.
- In a local development environment, you can visit a service URL like http://localhost:5000/ to access the service exported by their app.

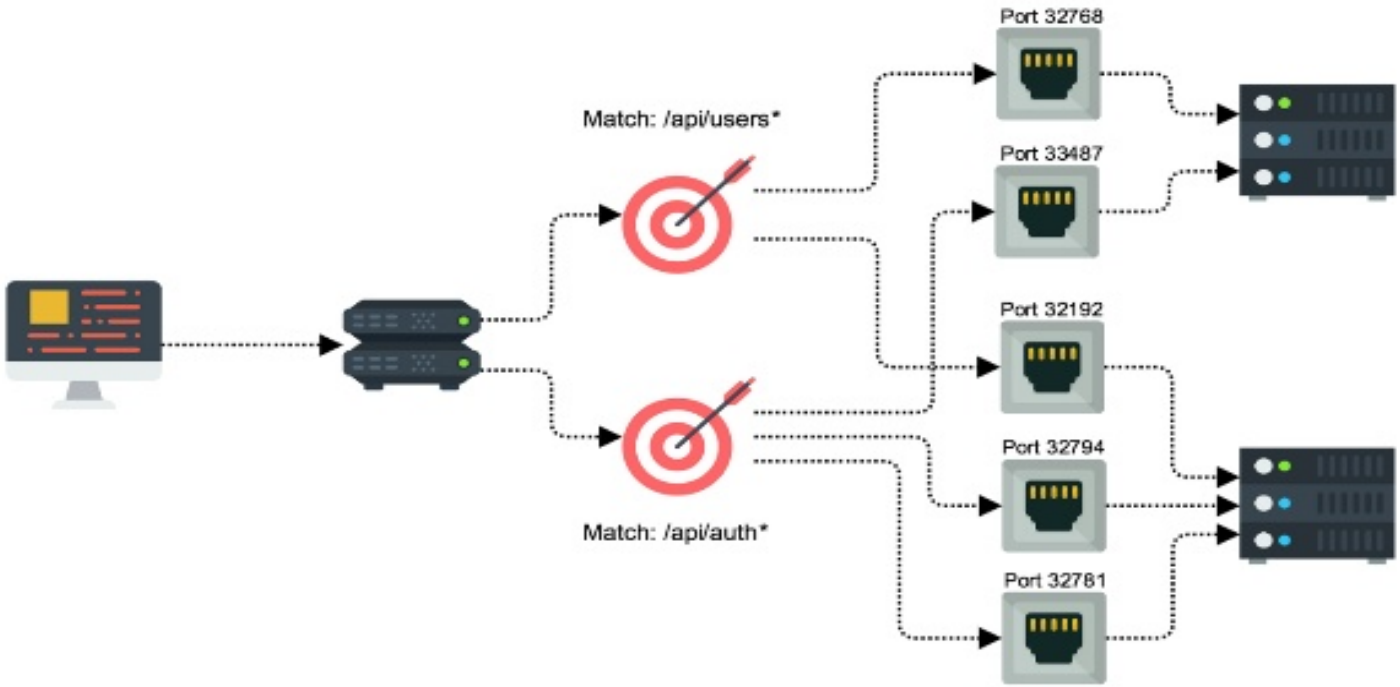


The Twelve-Factor App

7 Port Binding

Export services via port binding

- The Microservice is self-contained and does not rely on injection of a webserver to create a web-facing service.
- In a local development environment, you can visit a service URL like http://localhost:5000/ to access the service exported by their app.
- In deployment, a routing layer handles routing requests from a public-facing hostname to the port-bound web processes.



Data Sharing and Management

In old monilithic style, life was someway easy, but with microservices data management; you need to play harder.

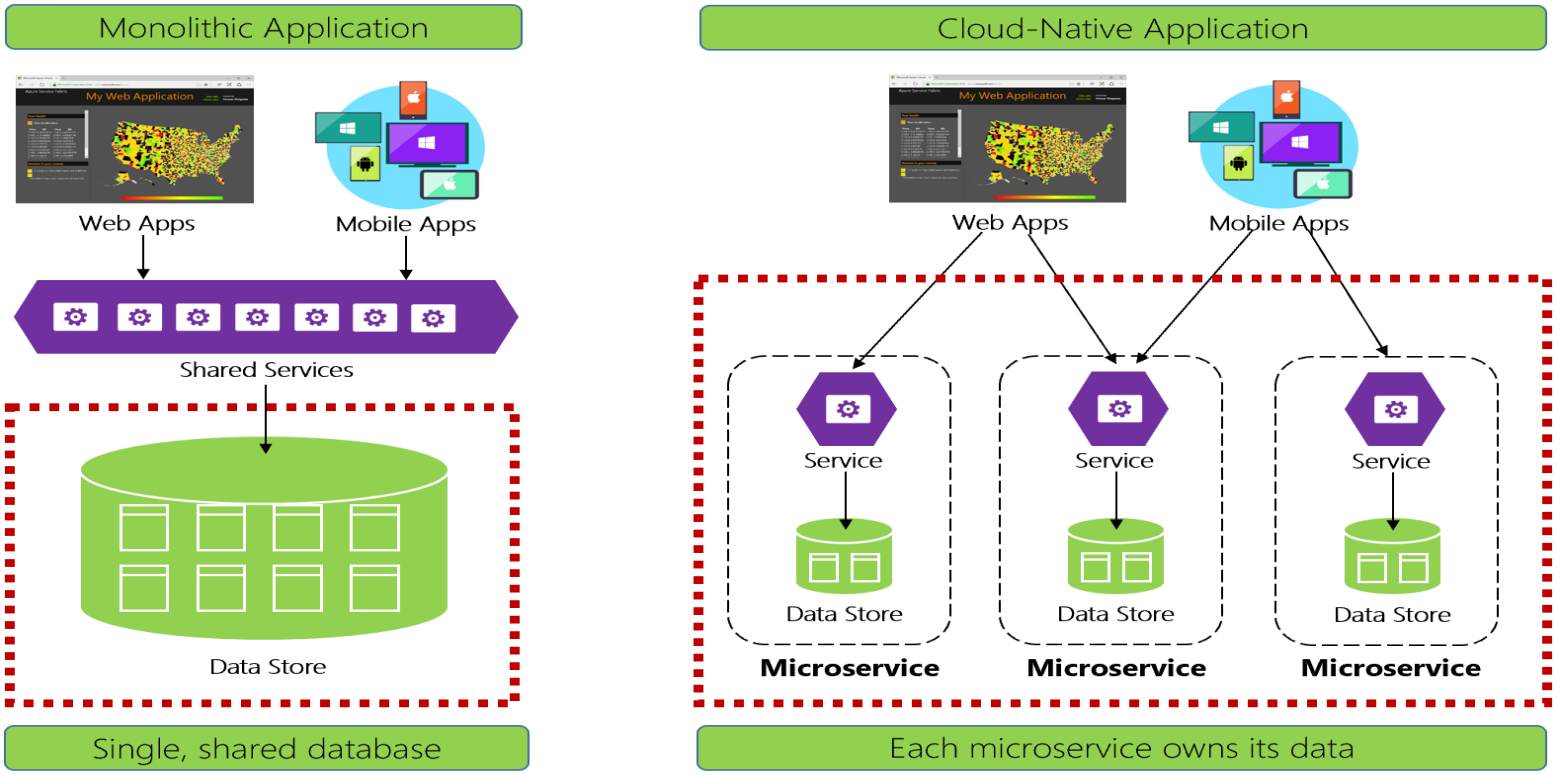


Figure: Distributed Data Architecture

https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/distributed-data

Data Sharing and Management

Why database per service?

- Domain data is encapsulated within the service

Data Sharing and Management

Why database per service?

- Domain data is encapsulated within the service
- Data schema can evolve without directly impacting other services code and functionality

Data Sharing and Management

Why database per service?

- Domain data is encapsulated within the service
- Data schema can evolve without directly impacting other services code and functionality
- Each data store can independently scale

Data Sharing and Management

Why database per service?

- Domain data is encapsulated within the service
- Data schema can evolve without directly impacting other services code and functionality
- Each data store can independently scale
- Data store failure in one service won't directly impact other services

Data Sharing and Management

Why database per service?

- Domain data is encapsulated within the service
- Data schema can evolve without directly impacting other services code and functionality
- Each data store can independently scale
- Data store failure in one service won't directly impact other services

BUT

You can also use single DB server and achieve some sort of that, How?

Data Sharing and Management

Why database per service?

- Domain data is encapsulated within the service
- Data schema can evolve without directly impacting other services code and functionality
- Each data store can independently scale
- Data store failure in one service won't directly impact other services

BUT

You can also use single DB server and achieve some sort of that, How?

- Schema management; that is each service has its own schema

Data Sharing and Management

Why database per service?

- Domain data is encapsulated within the service
- Data schema can evolve without directly impacting other services code and functionality
- Each data store can independently scale
- Data store failure in one service won't directly impact other services

BUT

You can also use single DB server and achieve some sort of that, How?

- Schema management; that is each service has its own schema
- Tables' access privilege

Data Sharing and Management

Why database per service?

- Domain data is encapsulated within the service
- Data schema can evolve without directly impacting other services code and functionality
- Each data store can independently scale
- Data store failure in one service won't directly impact other services

BUT

You can also use single DB server and achieve some sort of that, How?

- Schema management; that is each service has its own schema
- Tables' access privilege
- Database readonly views

Data Sharing and Management

Why database per service?

- Domain data is encapsulated within the service
- Data schema can evolve without directly impacting other services code and functionality
- Each data store can independently scale
- Data store failure in one service won't directly impact other services

BUT

You can also use single DB server and achieve some sort of that, How?

- Schema management; that is each service has its own schema
- Tables' access privilege
- Database readonly views

Which way is better?

A: It depends. However it is recommended to follow best practices.

The more simple and focused service you have, the easier management you get. A perfect way to do that, is applying DDD principles.

Data Sharing and Management

Why database per service?

- Domain data is encapsulated within the service
- Data schema can evolve without directly impacting other services code and functionality
- Each data store can independently scale
- Data store failure in one service won't directly impact other services

BUT

You can also use single DB server and achieve some sort of that, How?

- Schema management; that is each service has its own schema
- Tables' access privilege
- Database readonly views

Which way is better?

A: It depends. However it is recommended to follow best practices.

The more simple and focused service you have, the easier management you get. A perfect way to do that, is applying DDD principles.

While encapsulating data into separate microservices can increase agility, performance, and scalability, it also presents many challenges.

Data Sharing and Management

Cross-Service Queries

Usually you need to integrate to get\query data from other services

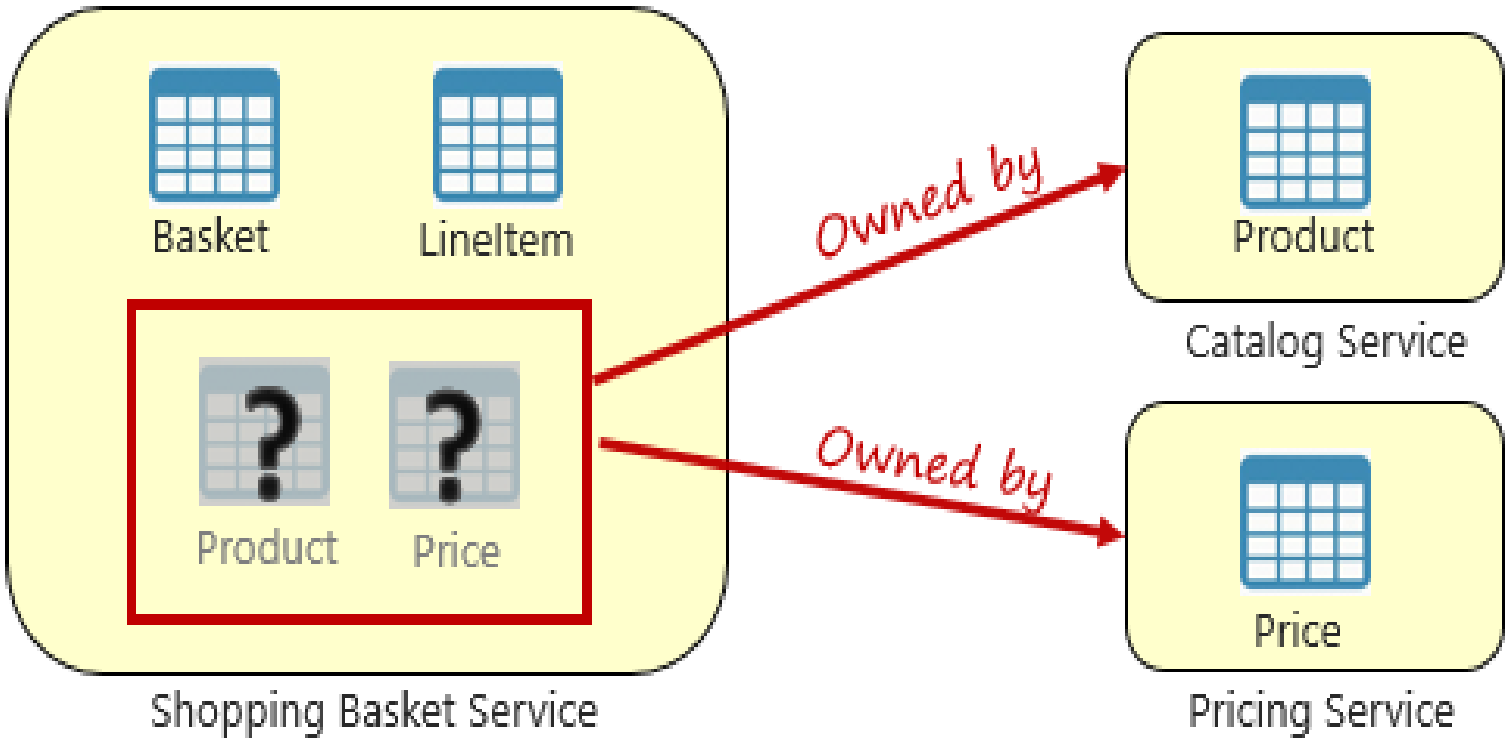


Figure: Querying across microservices

https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/distributed-data

Data Sharing and Management

Cross-Service Queries

Usually you need to integrate to get\query data from other services

- First and easiest way is to query service's database directly, Anti-Pattern

Data Sharing and Management

Cross-Service Queries

Usually you need to integrate to get\query data from other services

- First and easiest way is to query service's database directly, Anti-Pattern
- Also another simple way is through HTTP, but this may leads to coupling issues

Data Sharing and Management

Cross-Service Queries

Usually you need to integrate to get\query data from other services

- First and easiest way is to query service's database directly, Anti-Pattern
- Also another simple way is through HTTP, but this may leads to coupling issues
- Third option is using asynchronous calls via queues, and you should give attention to this

Cross-Service Queries

Usually you need to integrate to get\query data from other services

- First and easiest way is to query service's database directly, Anti-Pattern
- Also another simple way is through HTTP, but this may leads to coupling issues
- Third option is using asynchronous calls via queues, and you should give attention to this
- If the data volume is huge, and is not changed quickly, good way to use is

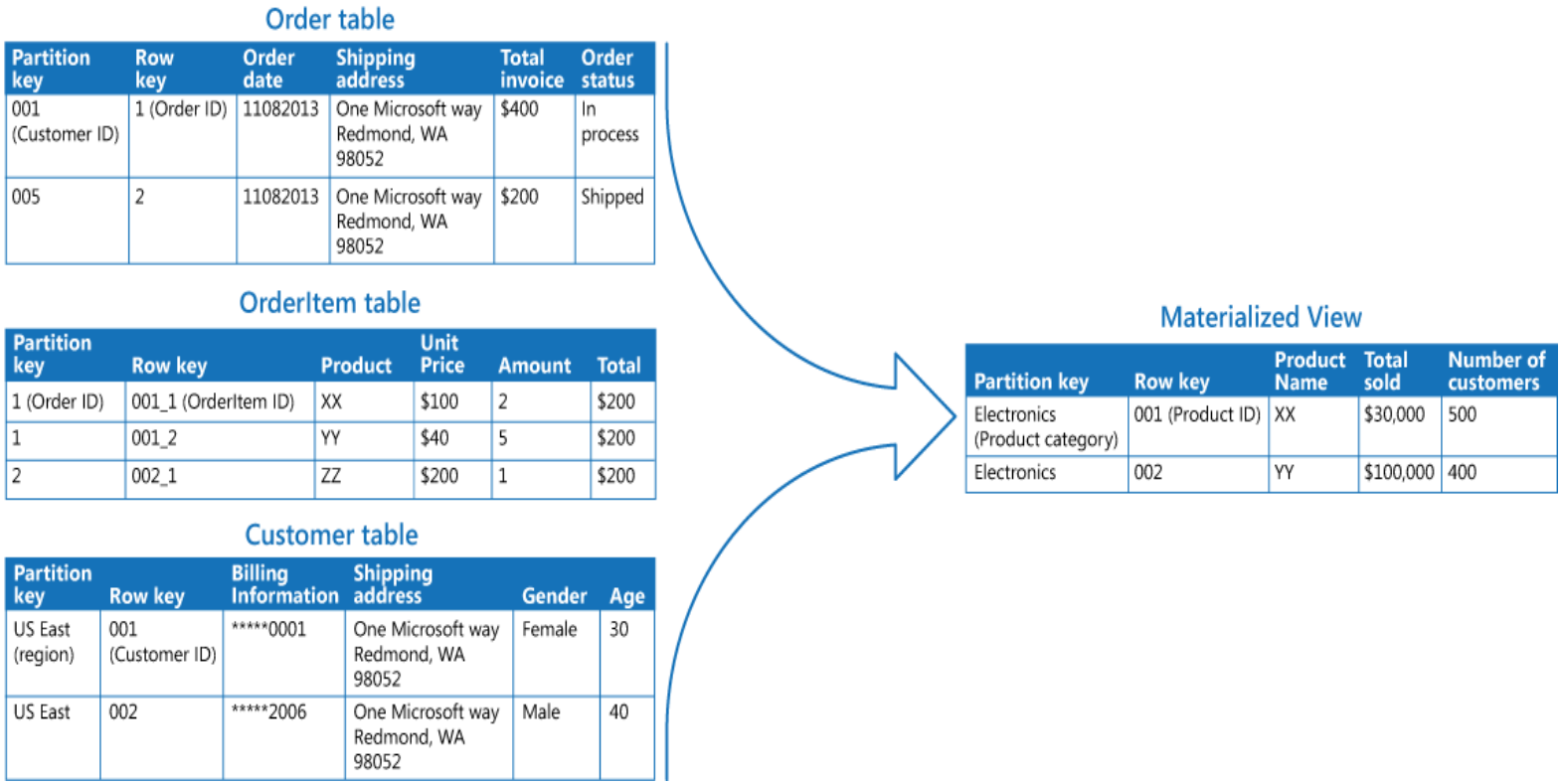


Figure: Materialized Views Pattern

Distributed Transactions

We move from a world of **immediate consistency** to that of **eventual consistency**
That is; in microservices. You can't depend on ACID transaction, but
BASE which is acronym for **B**asic **A**vailability, **S**oft-state, and **E**ventual consistency

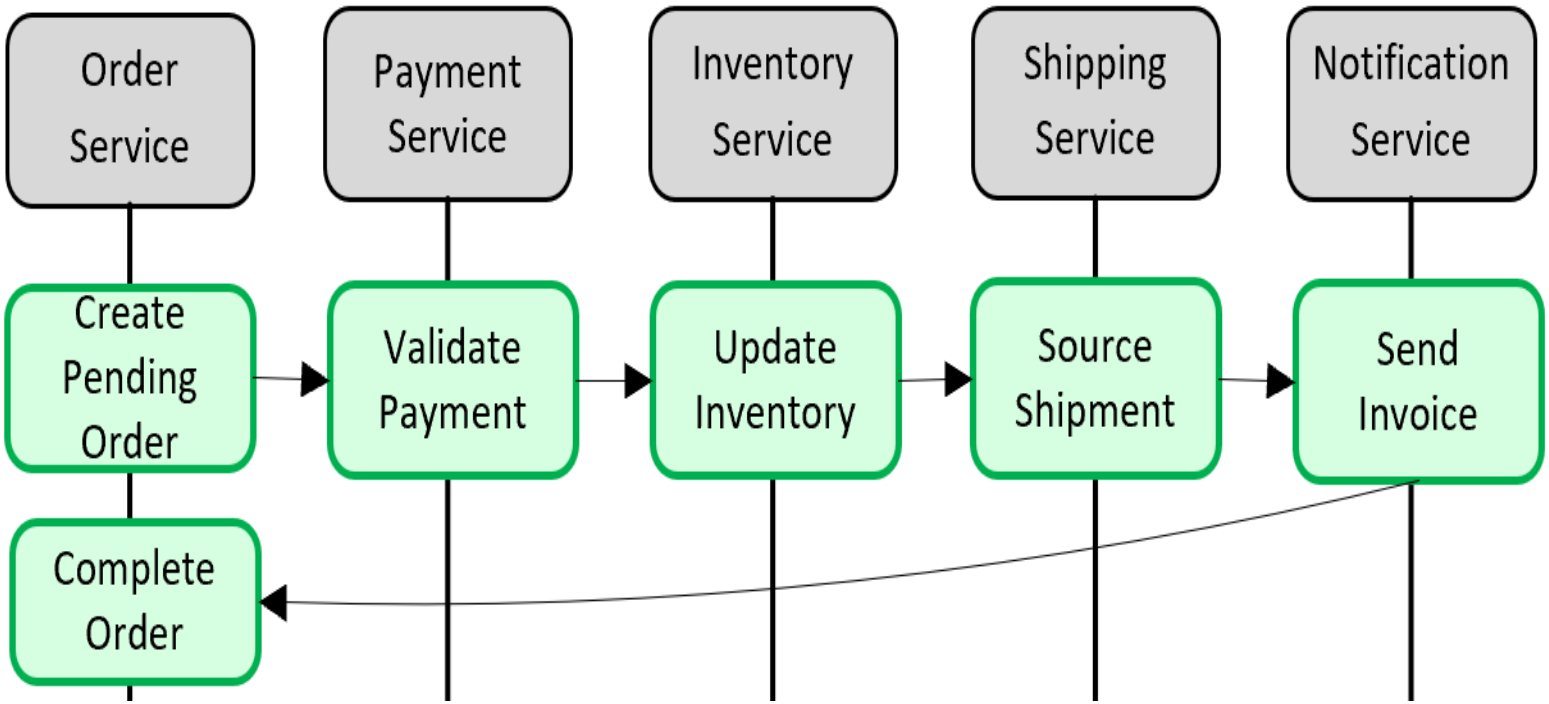


Figure: Transaction across microservices

https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/distributed-data

Data Sharing and Management

Distributed Transactions

- Having multiple data sources on different zones will make consistency issue

https://www.sciencedirect.com/topics/computer-science/two-phase-commit

Data Sharing and Management

Distributed Transactions

- Having multiple data sources on different zones will make consistency issue
- No options to handle that except Programmatic approach

https://www.sciencedirect.com/topics/computer-science/two-phase-commit

Distributed Transactions

- Having multiple data sources on different zones will make consistency issue
- No options to handle that except Programmatic approach
- First available way is 2-Phase-commits **2PC**

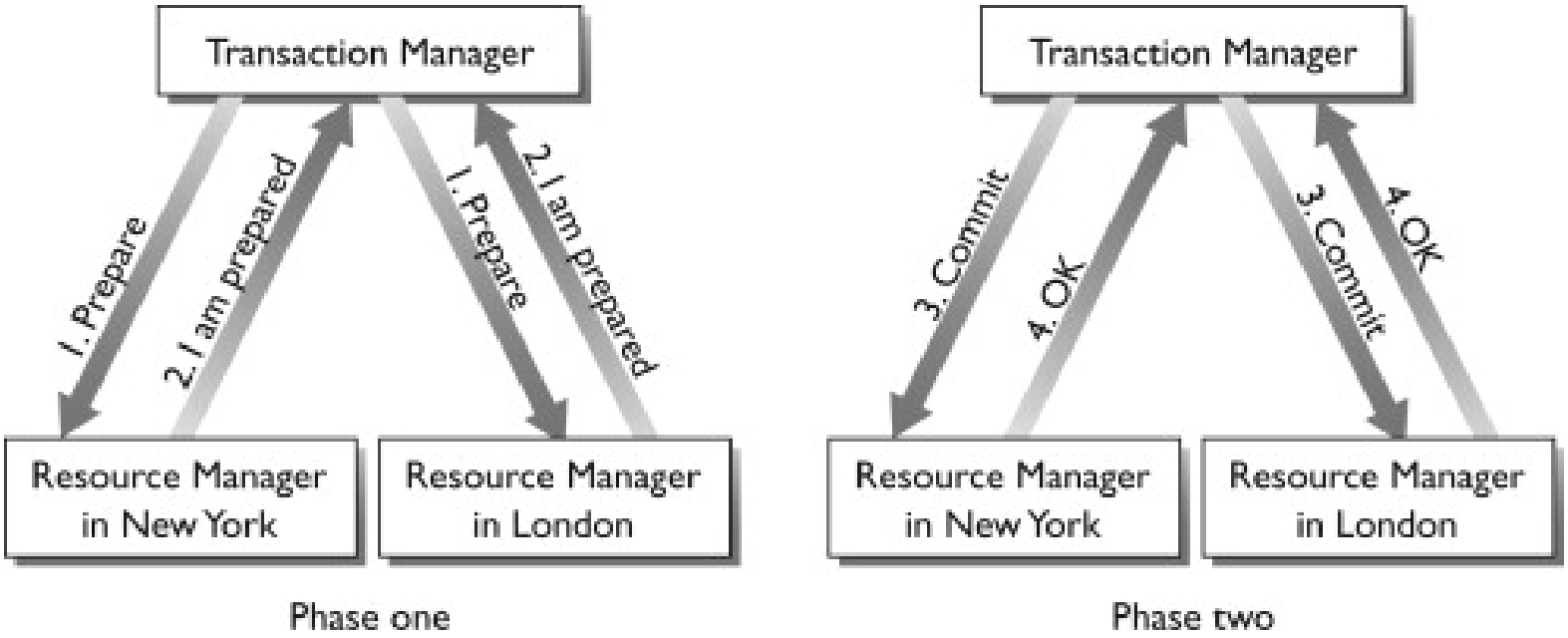


Figure: Two-phase Commit pattern

https://www.sciencedirect.com/topics/computer-science/two-phase-commit

Distributed Transactions

- Having multiple data sources on different zones will make consistency issue
- No options to handle that except Programmatic approach
- First available way is 2-Phase-commits **2PC**

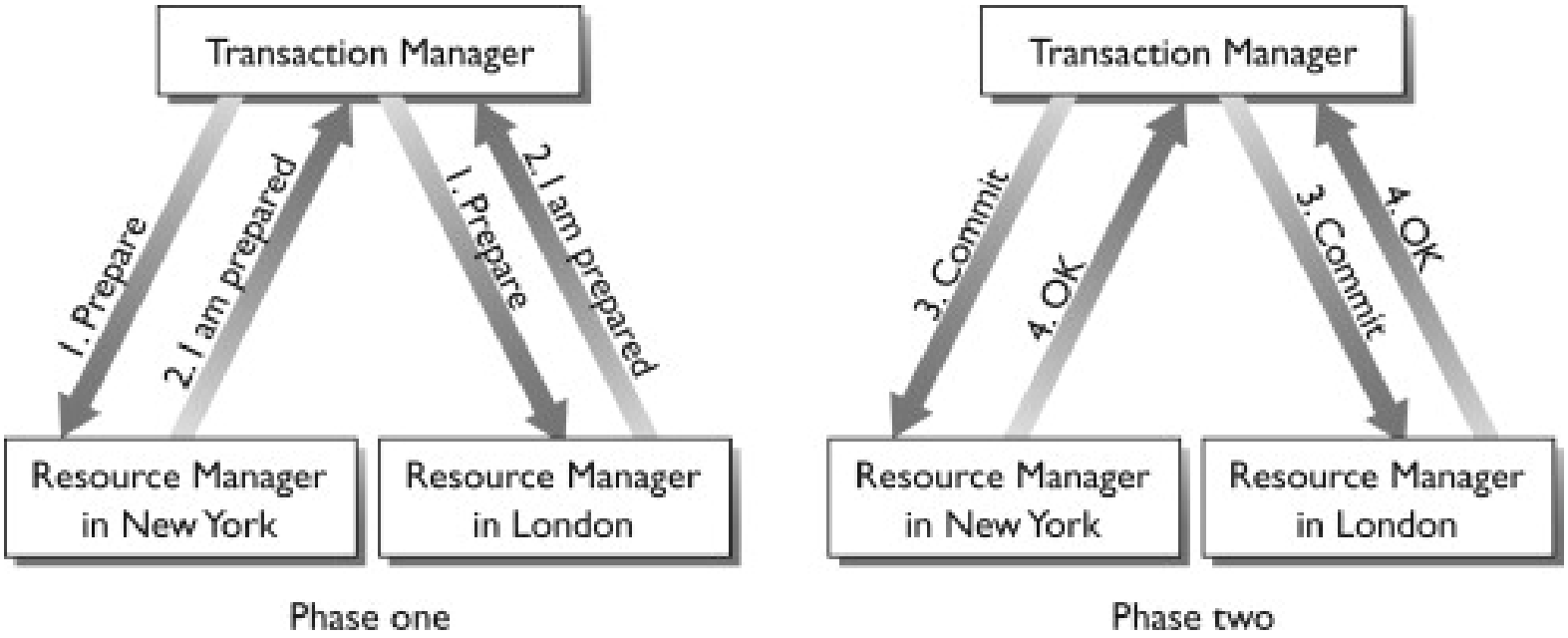


Figure: Two-phase Commit pattern

- In real case, this is impractical, and will lead to locking and time consuming issue

https://www.sciencedirect.com/topics/computer-science/two-phase-commit

Data Sharing and Management

Distributed Transactions

- Having multiple data sources on different zones will make consistency issue
- No options to handle that except Programmatic approach
- First available way is 2-Phase-commits **2PC**

Data Sharing and Management

Distributed Transactions

- Having multiple data sources on different zones will make consistency issue
- No options to handle that except Programmatic approach
- First available way is 2-Phase-commits **2PC**
- Another approach; If moving from point to point is succeeded, continue; else **Compensate**

Distributed Transactions

- Having multiple data sources on different zones will make consistency issue
- No options to handle that except Programmatic approach
- First available way is 2-Phase-commits **2PC**
- Another approach; If moving from point to point is succeeded, continue; else **Compensate**
- The big father of this approach is **Saga pattern**

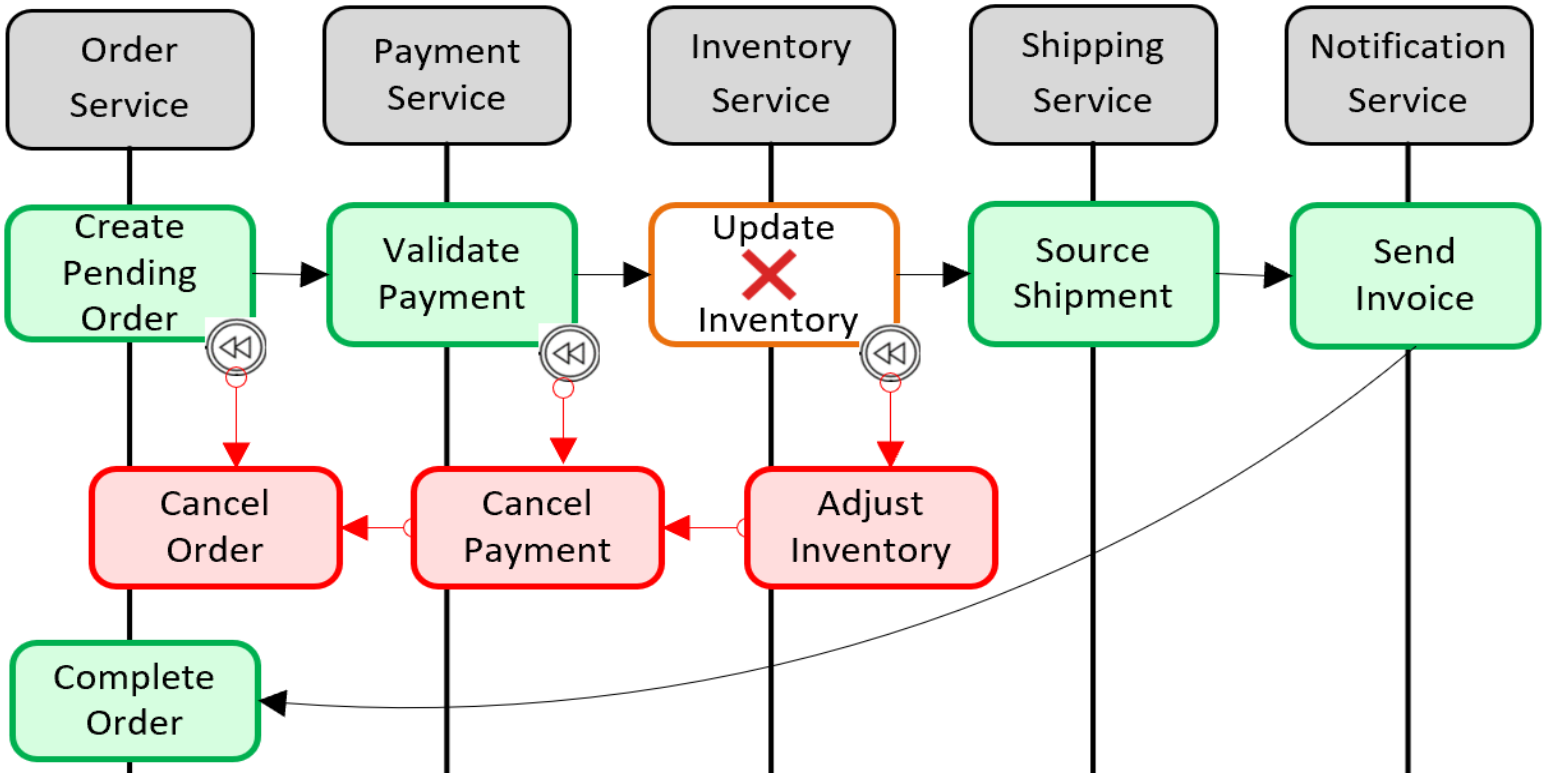


Figure: Rolling back a transaction

Data Sharing and Management

Command and Query Responsibility Segregation (**CQRS**)

Cloud native applications need sometimes to handle huge data.

Data Sharing and Management

Command and Query Responsibility Segregation (**CQRS**)

Cloud native applications need sometimes to handle huge data.

- In monolithic, there is no problem to handle this for simple CRUD operations

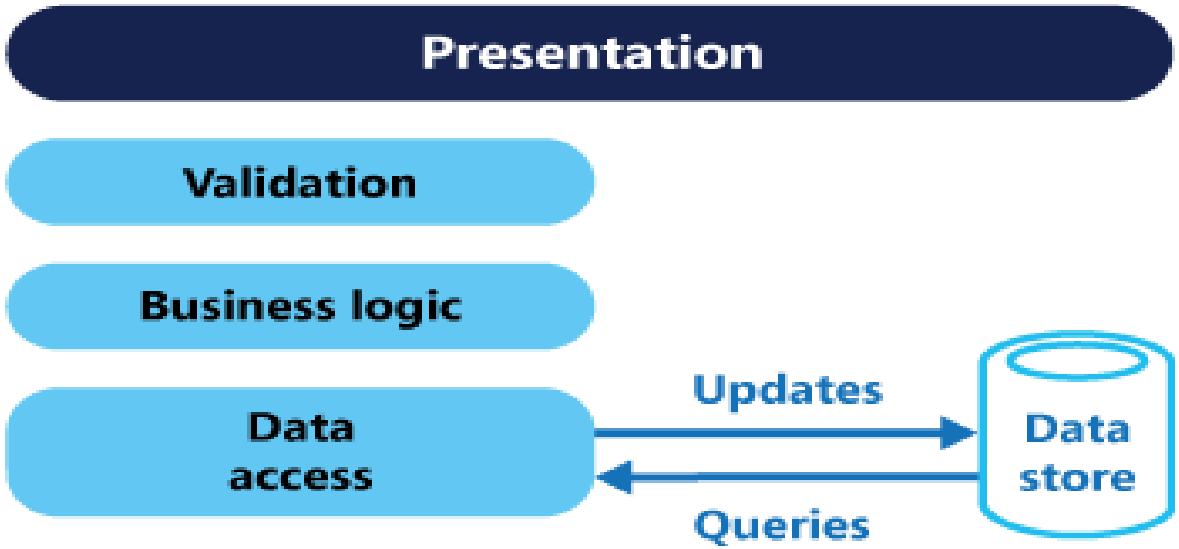


Figure: Traditional Application Architecture

Data Sharing and Management

Command and Query Responsibility Segregation (**CQRS**)

Cloud native applications need sometimes to handle huge data.

- In monolithic, there is no problem to handle this for simple CRUD operations
- Adding good managed indexes for example can solve the issue

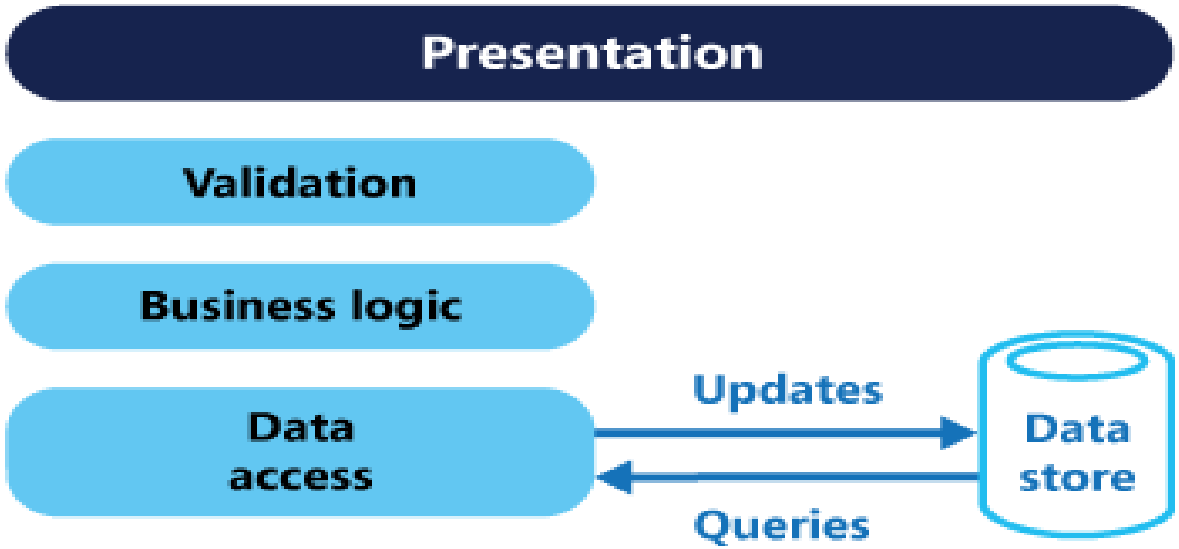


Figure: Traditional Application Architecture

Data Sharing and Management

Command and Query Responsibility Segregation (**CQRS**)

Cloud native applications need sometimes to handle huge data.

- In monolithic, there is no problem to handle this for simple CRUD operations
- Adding good managed indexes for example can solve the issue
- But read is not only the concern you focus on, but also writes

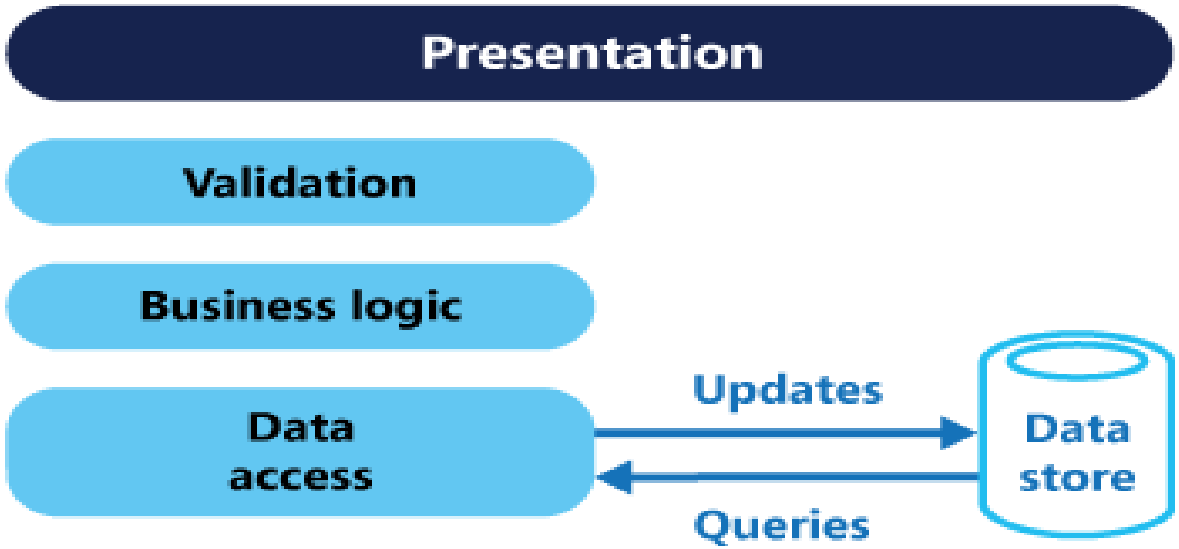


Figure: Traditional Application Architecture

Command and Query Responsibility Segregation (**CQRS**)

Cloud native applications need sometimes to handle huge data.

- In monolithic, there is no problem to handle this for simple CRUD operations
- Adding good managed indexes for example can solve the issue
- But read is not the only matter you focus on, but also writes
- Then, we have two concerns; **read and write**, and this is what CQRS intends to do

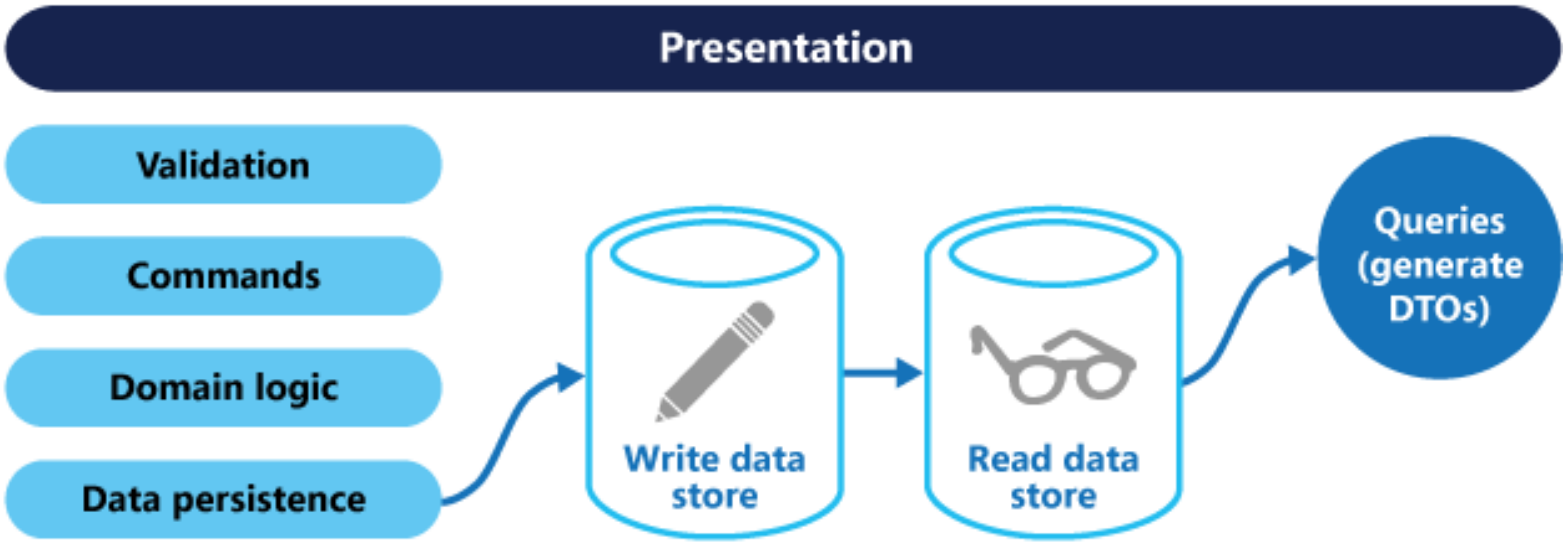


Figure: CQRS Application Architecture

Data Sharing and Management

Command and Query Responsibility Segregation (**CQRS**)

Cloud native applications need sometimes to handle huge data.

- In monolithic, there is no problem to handle this for simple CRUD operations
- Adding good managed indexes for example can solve the issue
- But read is not only the concern you focus on, but also writes
- Then, we have two concerns; **read and write**, and this is what CQRS intends to do

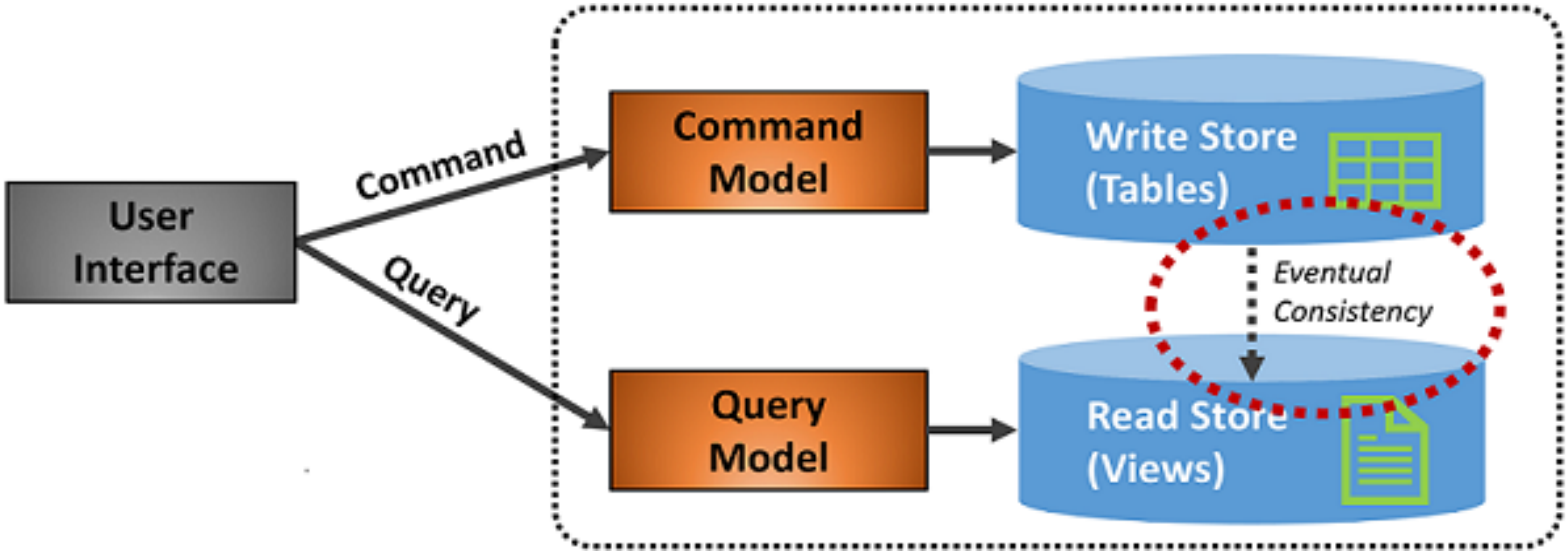


Figure: CQRS Implementation

Data Sharing and Management

Command and Query Responsibility Segregation (**CQRS**)

Cloud native applications need sometimes to handle huge data.

- In monolithic, there is no problem to handle this for simple CRUD operations
- Adding good managed indexes for example can solve the issue
- But read is not only the concern you focus on, but also writes
- Then, we have two concerns; **read and write**, and this is what CQRS intends to do
- You should give attention to **Eventual Consistency**

Data Sharing and Management

Command and Query Responsibility Segregation (**CQRS**)

Cloud native applications need sometimes to handle huge data.

- In monolithic, there is no problem to handle this for simple CRUD operations
- Adding good managed indexes for example can solve the issue
- But read is not only the concern you focus on, but also writes
- Then, we have two concerns; **read and write**, and this is what CQRS intends to do
- You should give attention to **Eventual Consistency**
  1. I watch the weather report and learn that it's going to rain tomorrow.

Data Sharing and Management

Command and Query Responsibility Segregation (**CQRS**)

Cloud native applications need sometimes to handle huge data.

- In monolithic, there is no problem to handle this for simple CRUD operations
- Adding good managed indexes for example can solve the issue
- But read is not only the concern you focus on, but also writes
- Then, we have two concerns; **read and write**, and this is what CQRS intends to do
- You should give attention to **Eventual Consistency**
  1. I watch the weather report and learn that it's going to rain tomorrow.
  2. I tell you that it's going to rain tomorrow.

Data Sharing and Management

Command and Query Responsibility Segregation (**CQRS**)

Cloud native applications need sometimes to handle huge data.

- In monolithic, there is no problem to handle this for simple CRUD operations
- Adding good managed indexes for example can solve the issue
- But read is not only the concern you focus on, but also writes
- Then, we have two concerns; **read and write**, and this is what CQRS intends to do
- You should give attention to **Eventual Consistency**
  1. I watch the weather report and learn that it's going to rain tomorrow.
  2. I tell you that it's going to rain tomorrow.
  3. Your neighbor tells his wife that it's going to be sunny tomorrow.

Data Sharing and Management

Command and Query Responsibility Segregation (**CQRS**)

Cloud native applications need sometimes to handle huge data.

- In monolithic, there is no problem to handle this for simple CRUD operations
- Adding good managed indexes for example can solve the issue
- But read is not only the concern you focus on, but also writes
- Then, we have two concerns; **read and write**, and this is what CQRS intends to do
- You should give attention to **Eventual Consistency**
  1. I watch the weather report and learn that it's going to rain tomorrow.
  2. I tell you that it's going to rain tomorrow.
  3. Your neighbor tells his wife that it's going to be sunny tomorrow.
  4. You tell your neighbor that it is going to rain tomorrow.

Data Sharing and Management

Command and Query Responsibility Segregation (**CQRS**)

Cloud native applications need sometimes to handle huge data.

- In monolithic, there is no problem to handle this for simple CRUD operations
- Adding good managed indexes for example can solve the issue
- But read is not only the concern you focus on, but also writes
- Then, we have two concerns; **read and write**, and this is what CQRS intends to do
- However you need to make your data always in a sync mode
- You should give attention to **Eventual Consistency**
- You need also to make your query model always in a sync state

Data Sharing and Management

Command and Query Responsibility Segregation (**CQRS**)

Cloud native applications need sometimes to handle huge data.

- In monolithic, there is no problem to handle this for simple CRUD operations
- Adding good managed indexes for example can solve the issue
- But read is not only the concern you focus on, but also writes
- Then, we have two concerns; **read and write**, and this is what CQRS intends to do
- However you need to make your data always in a sync mode
- You should give attention to **Eventual Consistency**
- You need also to make your query model always in a sync state
- A good way to achieve this, is to use **Event Sourcing Pattern**

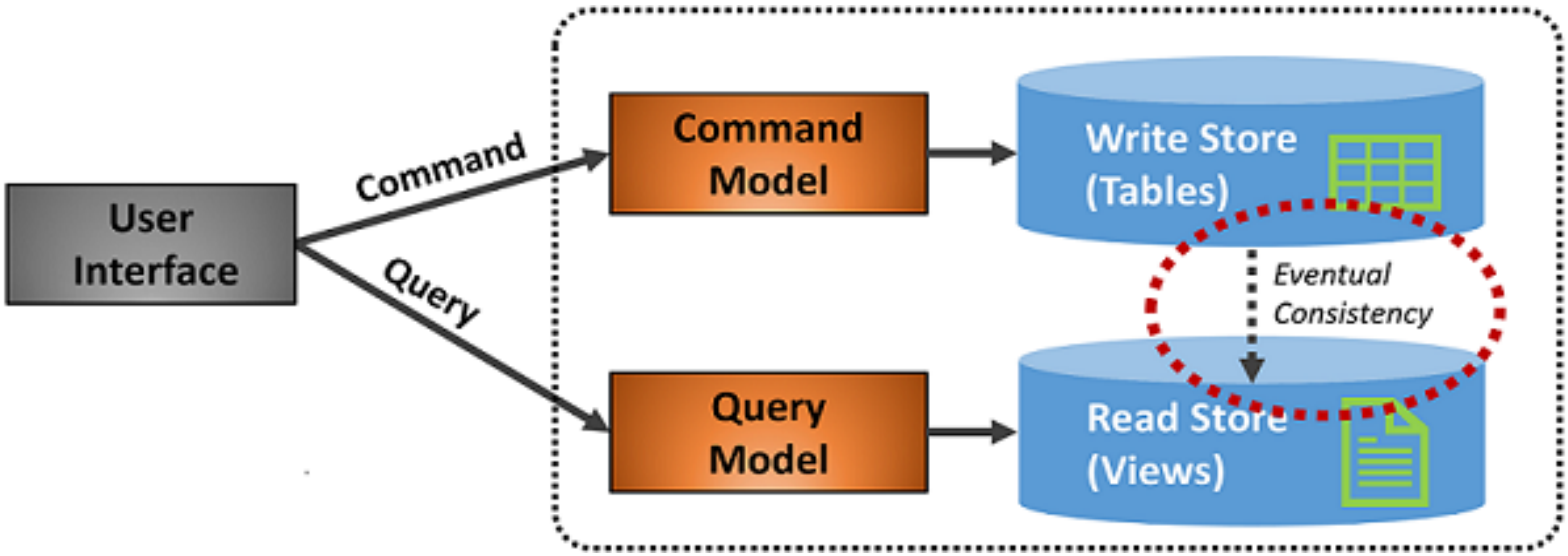


Figure: CQRS Implementation

Data Sharing and Management

Event Sourcing Pattern

Applying a list of events in atomic way is very hard to be applied using distributed transactions!

Data Sharing and Management

Event Sourcing Pattern

Applying a list of events in atomic way is very hard to be applied using distributed transactions!

- How did we get there? (*History is always matter*)

Event Sourcing Pattern

Applying a list of events in atomic way is very hard to be applied using distributed transactions!

- How did we get there? (*History is always matter*)
- An approach to handling operations on data that's driven by a sequence of events, each of which is recorded in an **append-only store**

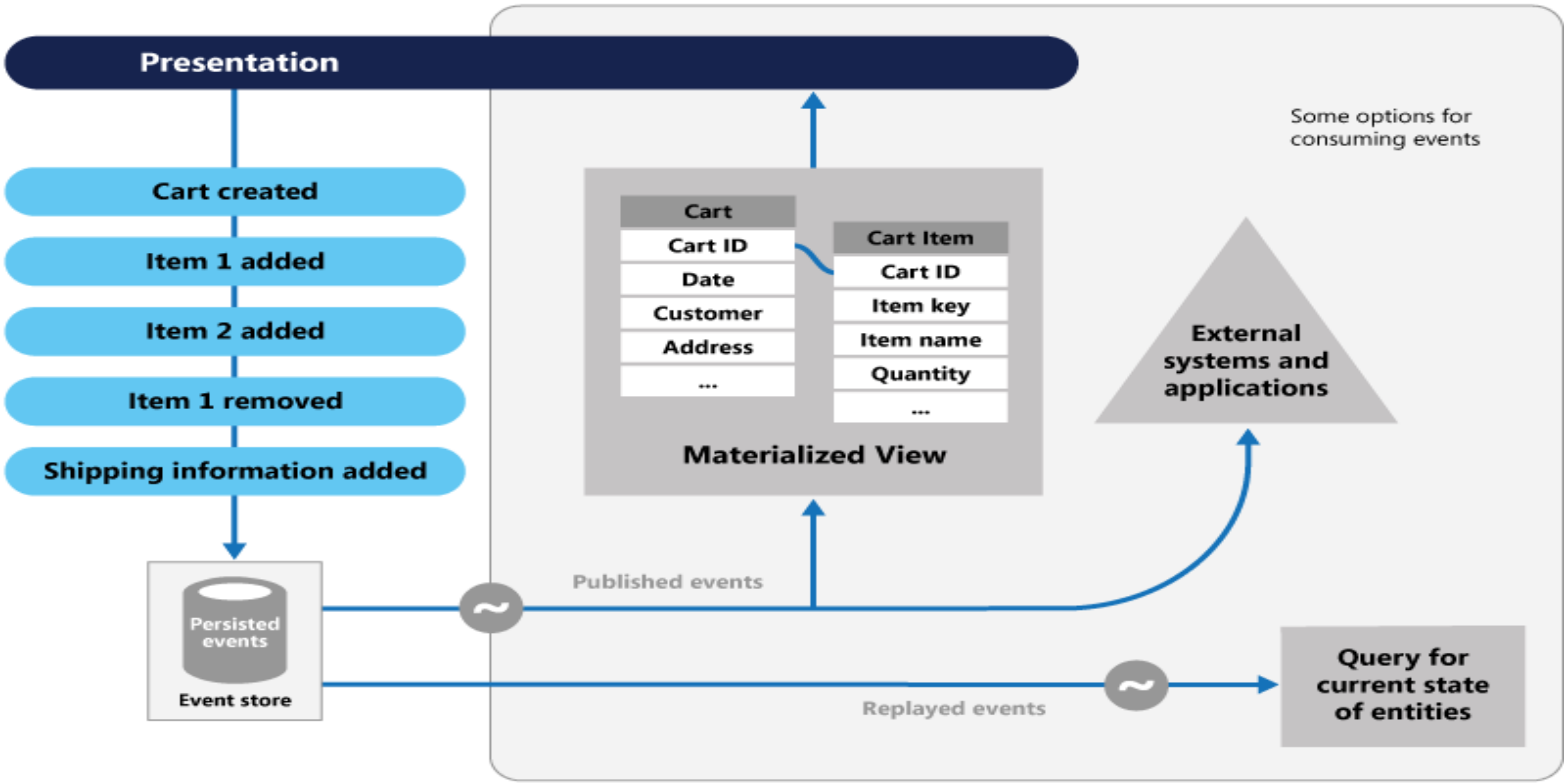


Figure: Event Sourcing

Event Sourcing Pattern

By applying this pattern, you will have

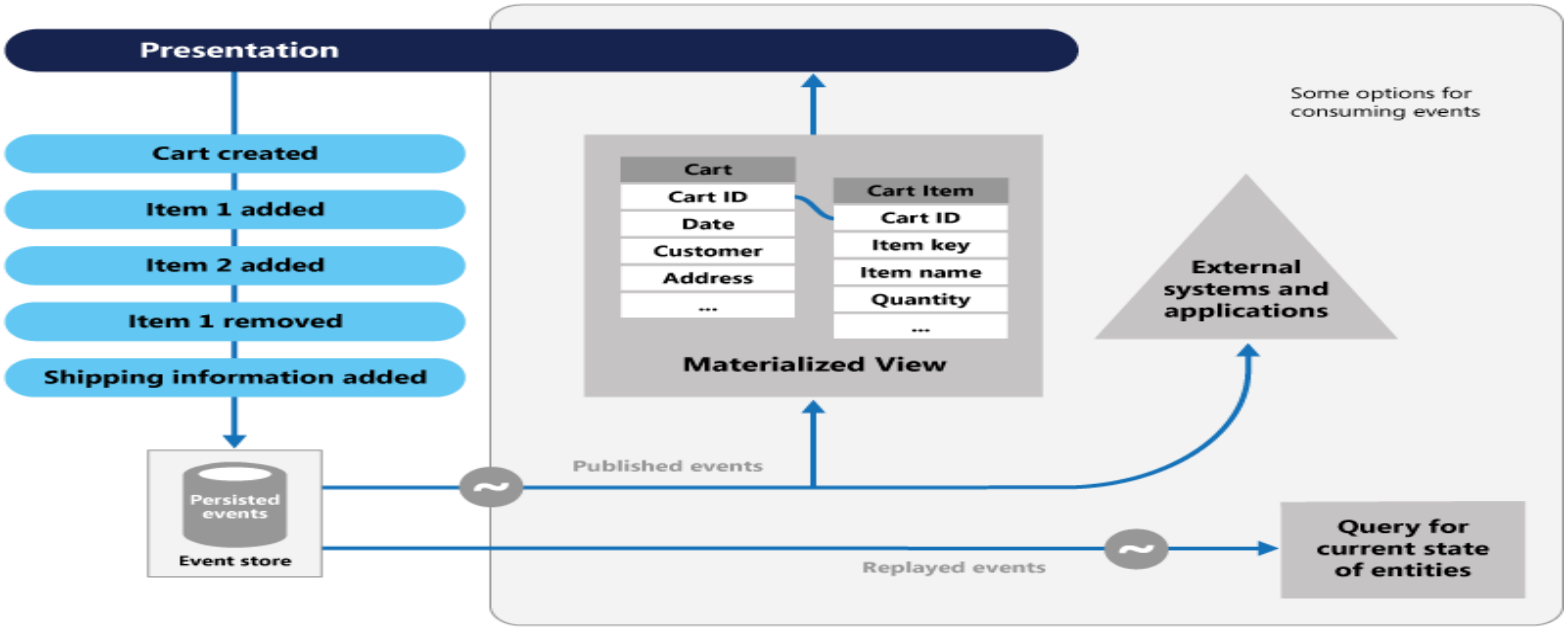


Figure: Event Sourcing

Event Sourcing Pattern

By applying this pattern, you will have

- Better performance, that is your the application code that generates the events will be decoupled from the actual code

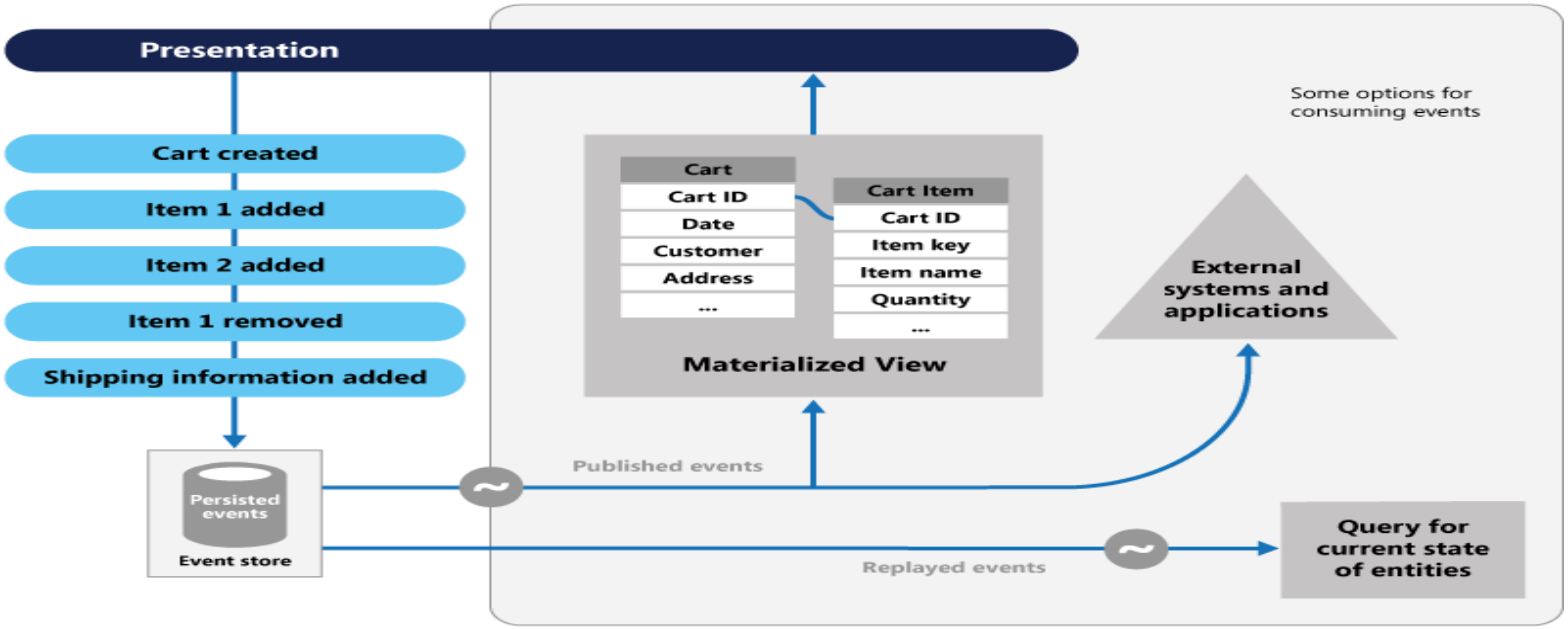


Figure: Event Sourcing

Event Sourcing Pattern

By applying this pattern, you will have

- Better performance, that is your the application code that generates the events will be decoupled from the actual code
- More power to scale up!

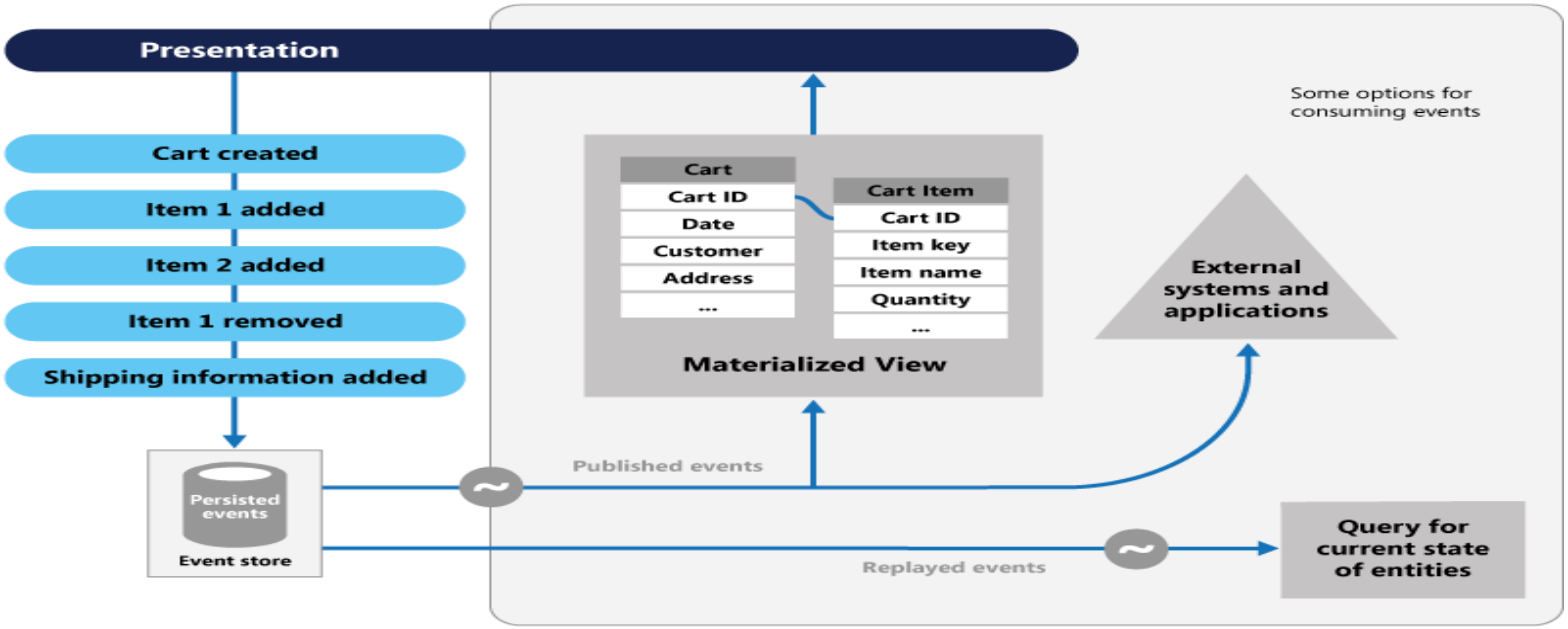


Figure: Event Sourcing

Event Sourcing Pattern

By applying this pattern, you will have

- Better performance, that is your the application code that generates the events will be decoupled from the actual code
- More power to scale up!
- Log history which is fundamental need in any project

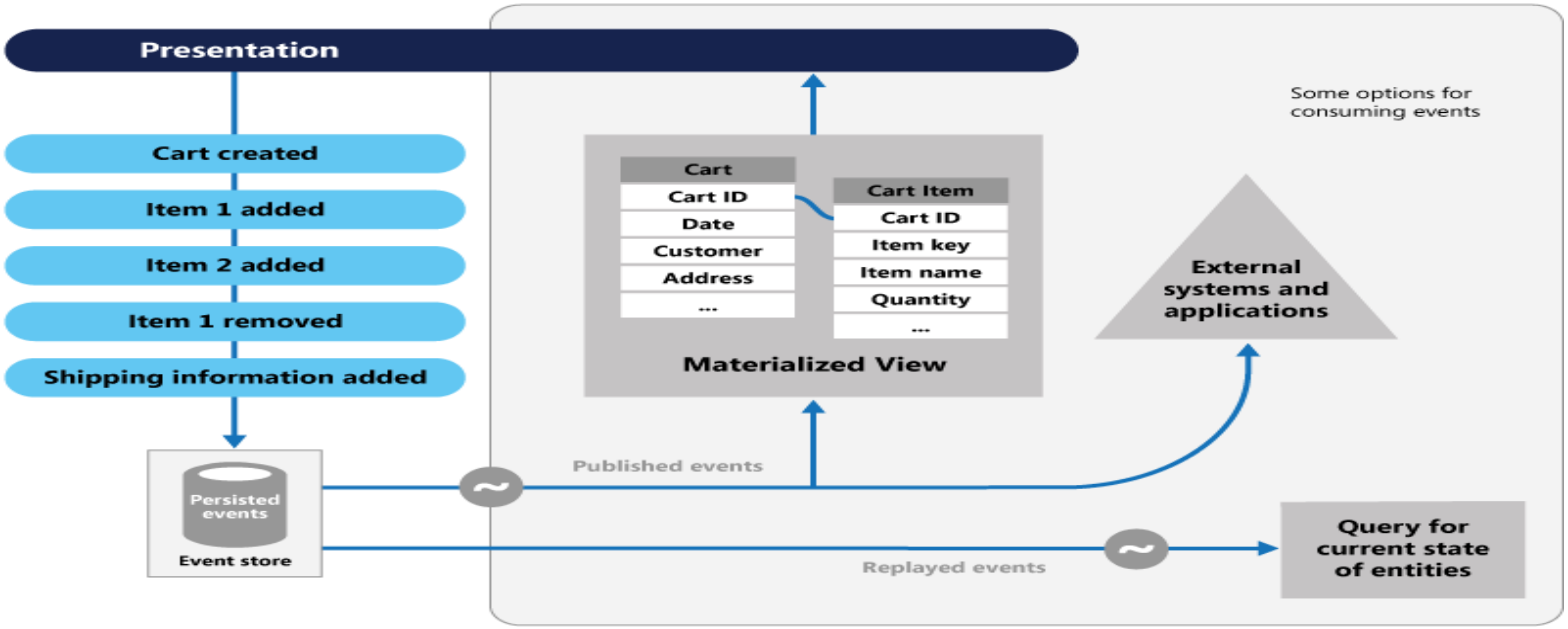


Figure: Event Sourcing

Event Sourcing Pattern

By applying this pattern, you will have

- Better performance, that is your the application code that generates the events will be decoupled from the actual code
- More power to scale up!
- Log history which is fundamental need in any project
- prevent concurrent updates from causing conflicts because it avoids the requirement to directly update objects in the data store

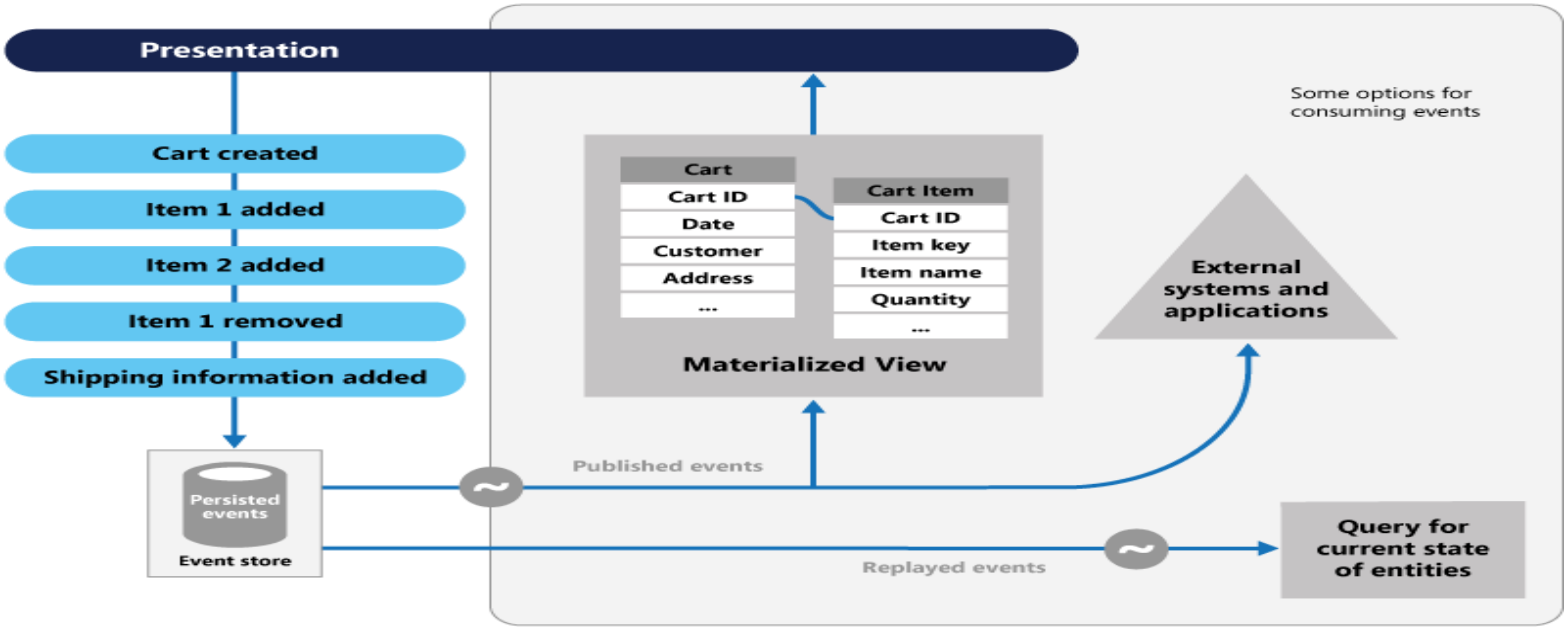


Figure: Event Sourcing

Data Sharing and Management

Caching Pattern

Calling backing services is a heavy process, and to overcome this issue; you need to have some shared data store to be fully managed in run-time across all instances.

Data Sharing and Management

Caching Pattern

Calling backing services is a heavy process, and to overcome this issue; you need to have some shared data store to be fully managed in run-time across all instances.

- How can multiple instances within multiple apps share something?

Data Sharing and Management

Caching Pattern

Calling backing services is a heavy process, and to overcome this issue; you need to have some shared data store to be fully managed in run-time across all instances.

- How can multiple instances within multiple apps share something?
- Why not just loading on startup? *In-memory caching*

Data Sharing and Management

Caching Pattern

Calling backing services is a heavy process, and to overcome this issue; you need to have some shared data store to be fully managed in run-time across all instances.

- How can multiple instances within multiple apps share something?
- Why not just loading on startup? *In-memory caching*
- Caching service is the solution. *Distributed caching*

Caching Pattern

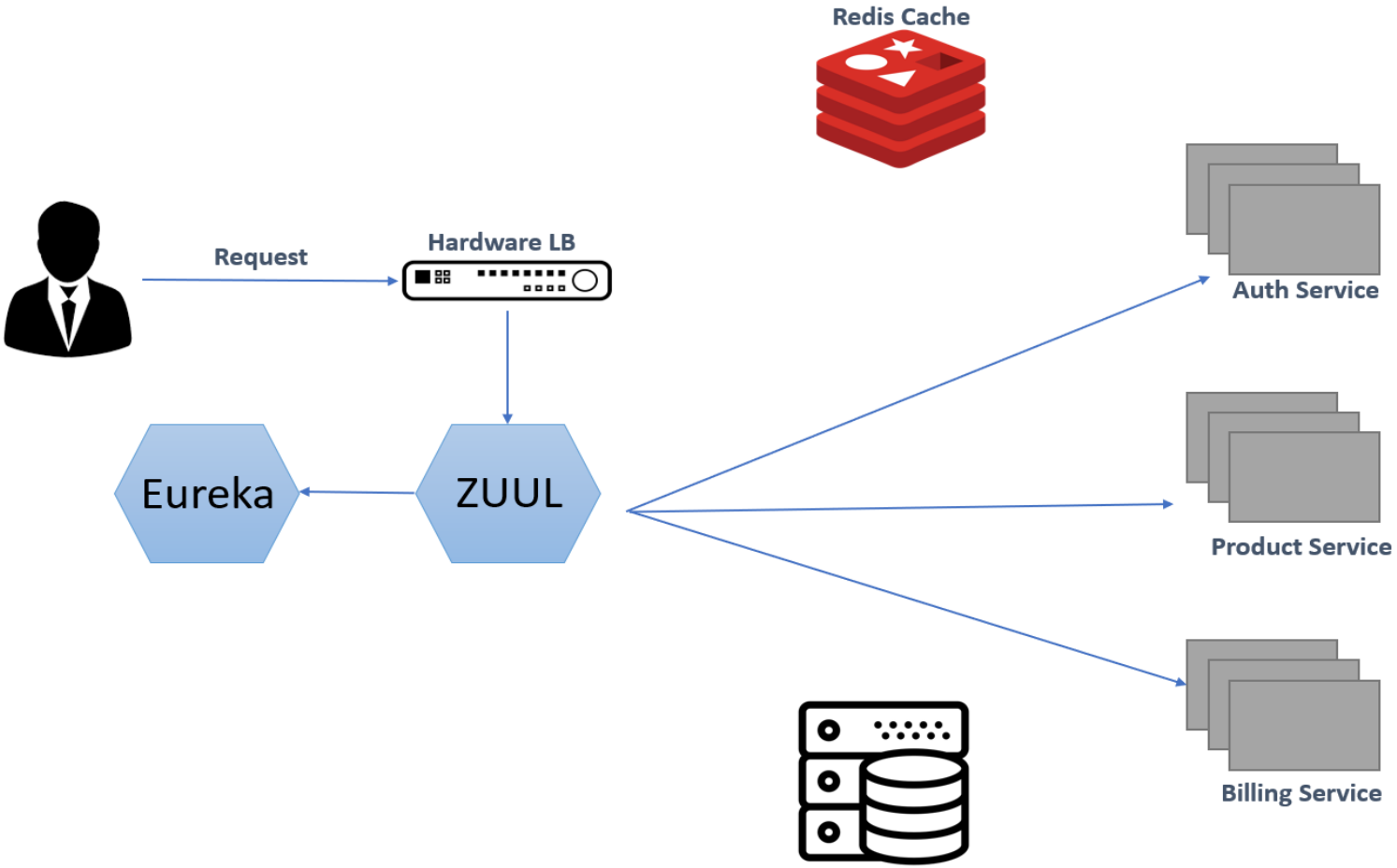


Figure: Typical Scenario

Caching Pattern

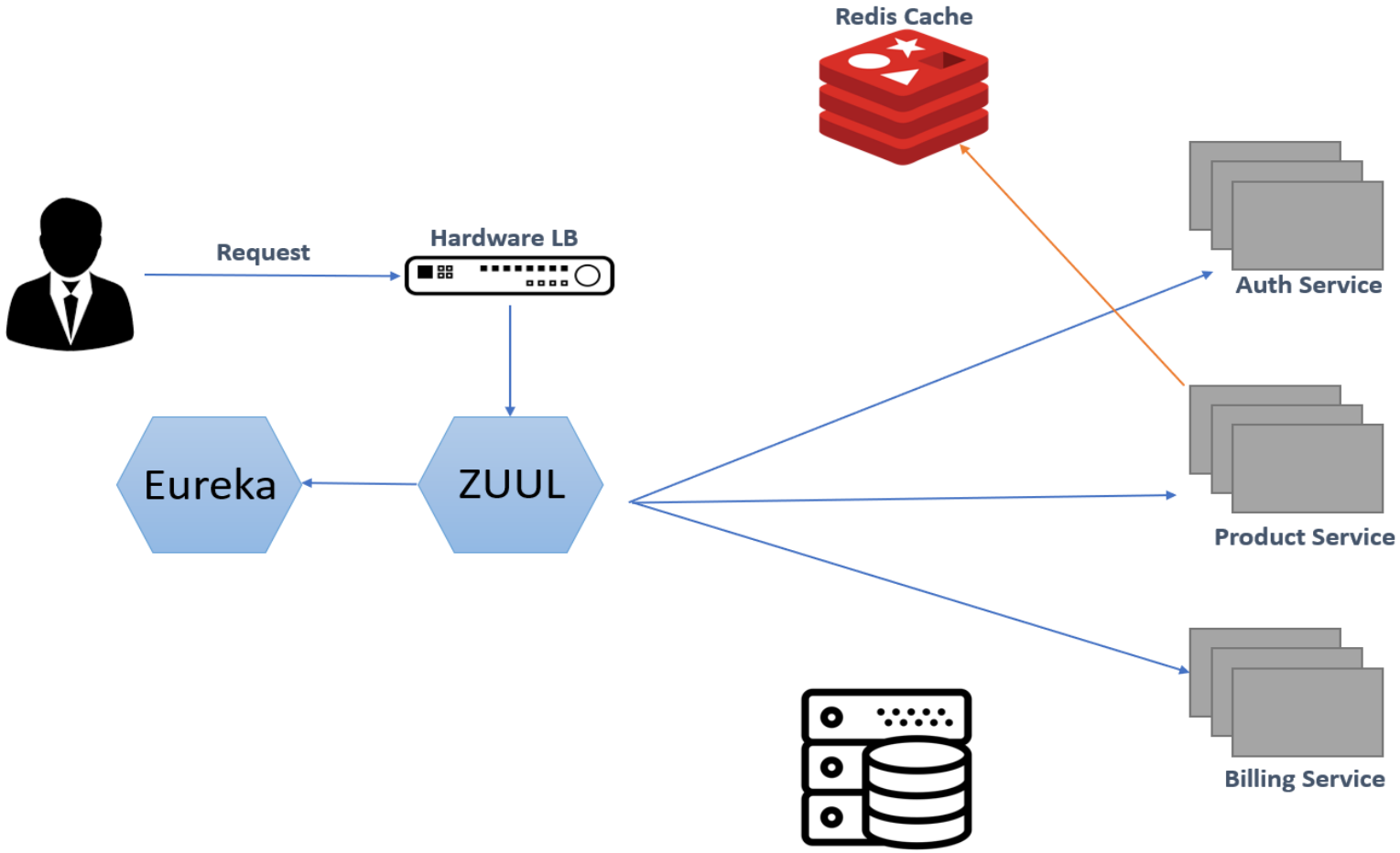


Figure: Typical Scenario

Caching Pattern

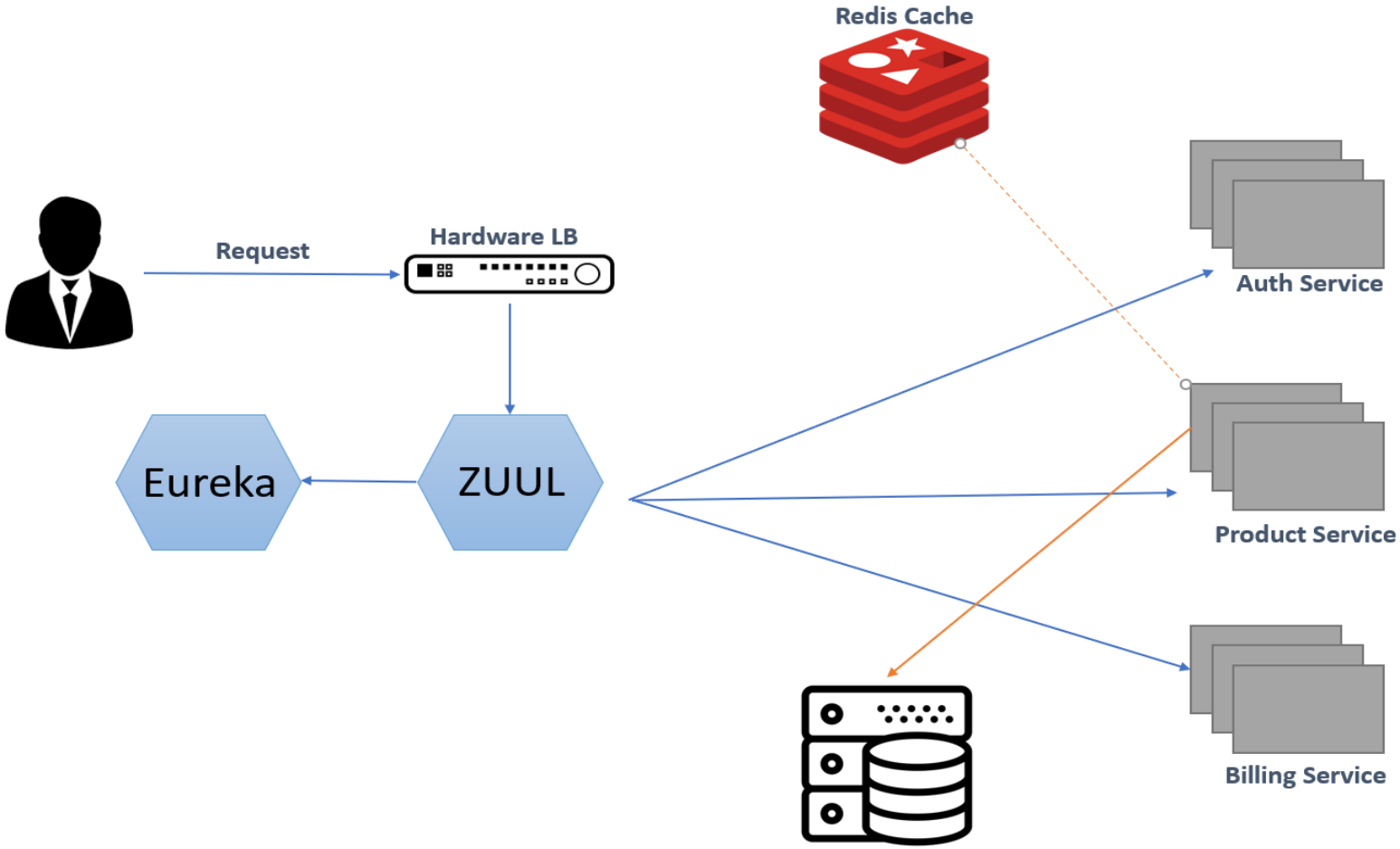


Figure: Typical Scenario

Caching Pattern

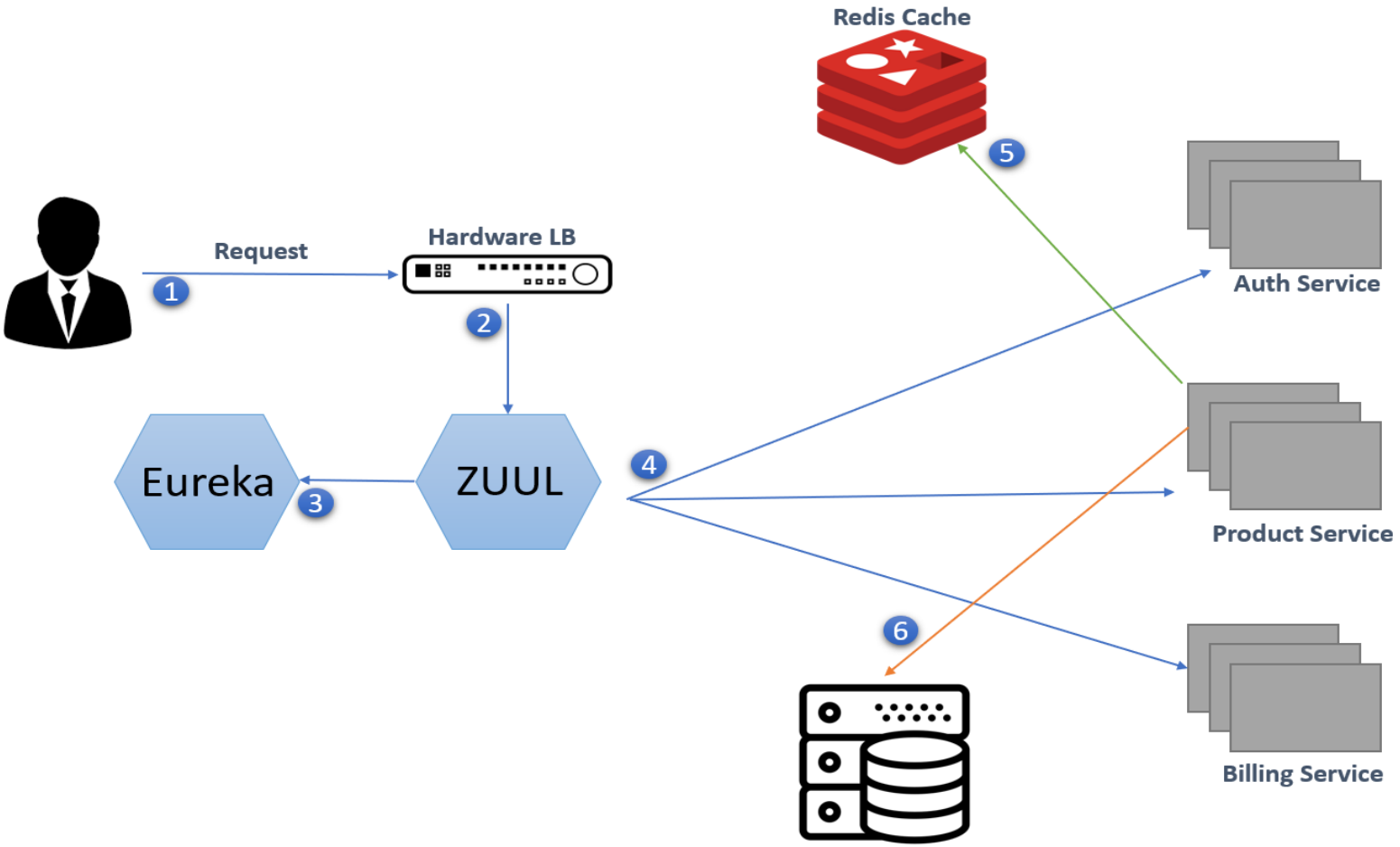


Figure: Typical Scenario

Data Sharing and Management

Caching Pattern

Calling backing services is a heavy process, and to overcome this issue; you need to have some shared data store to be fully managed in run-time across all instances.

- How can multiple instances within multiple apps share something?
- Why not just loading on startup? *In-memory caching*
- Caching service is the solution. *Distributed caching*

Caching Pattern

Calling backing services is a heavy process, and to overcome this issue; you need to have some shared data store to be fully managed in run-time across all instances.

- How can multiple instances within multiple apps share something?
- Why not just loading on startup? *In-memory caching*
- Caching service is the solution. *Distributed caching*

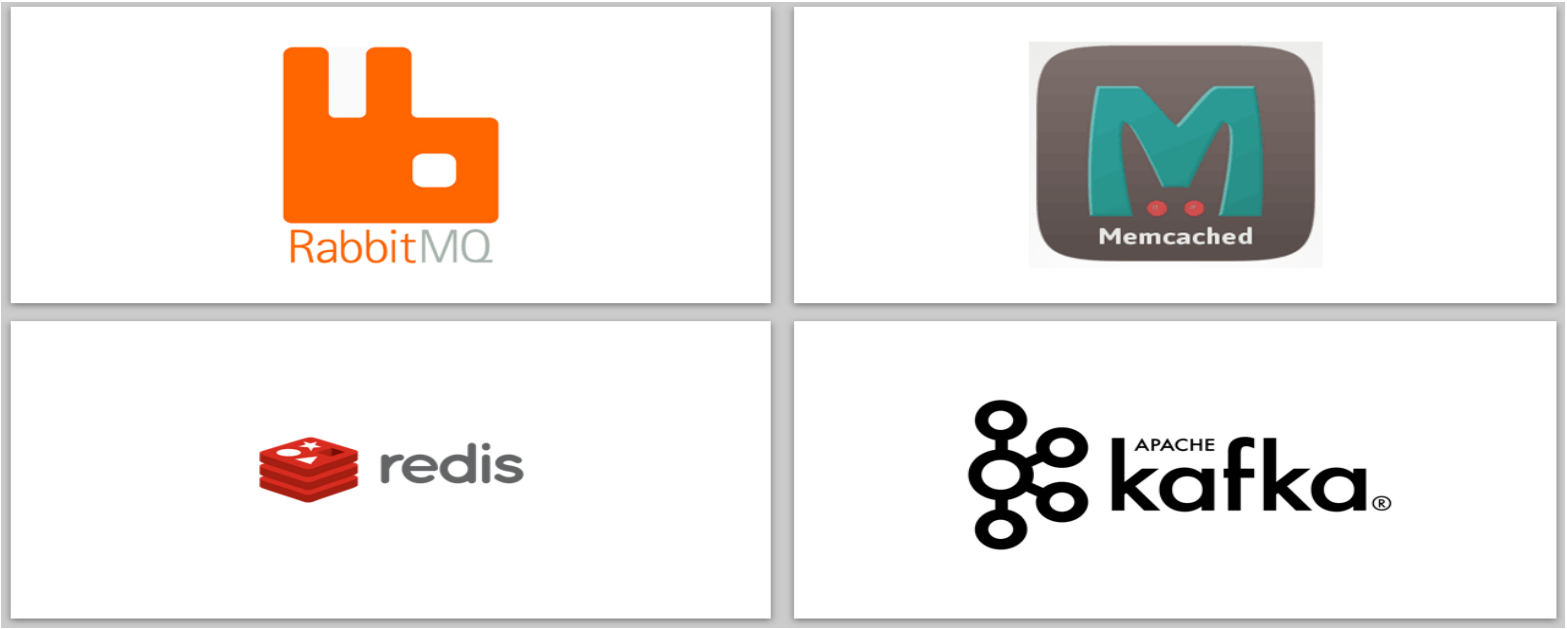


Figure: Shared DS Market Availability

Data Sharing and Management

Caching Pattern

Calling backing services is a heavy process, and to overcome this issue; you need to have some shared data store to be fully managed in run-time across all instances.

- How can multiple instances within multiple apps share something?
- Why not just loading on startup? *In-memory caching*
- Caching service is the solution. *Distributed caching*
- You should consider also availability and data consistency

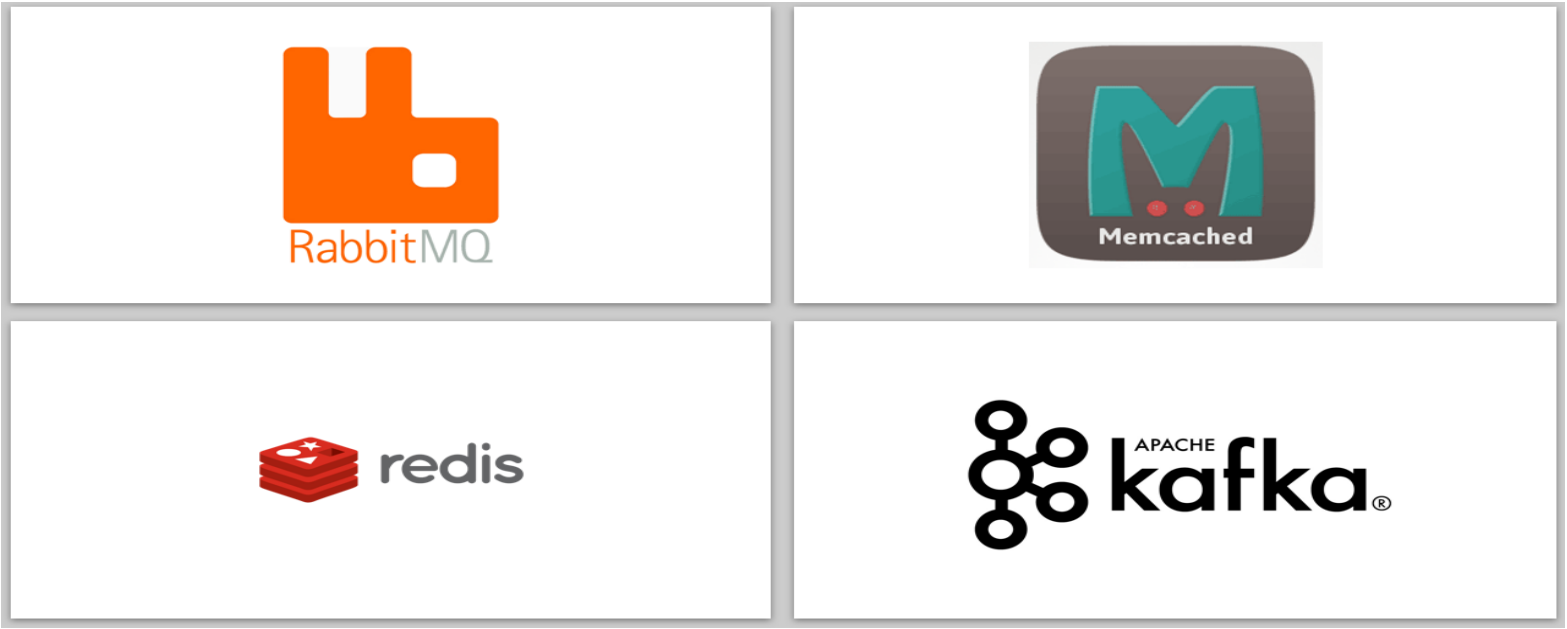


Figure: Shared DS Market Availability

Service Orchestrators

An orchestrator handles tasks related to deploying and managing a set of services, with Orchestrator you can

- Placing services on nodes.

Service Orchestrators

An orchestrator handles tasks related to deploying and managing a set of services, with Orchestrator you can

- Placing services on nodes.
- Monitoring the health of services and restarting unhealthy services.

Service Orchestrators

An orchestrator handles tasks related to deploying and managing a set of services, with Orchestrator you can

- Placing services on nodes.
- Monitoring the health of services and restarting unhealthy services.
- Load balancing network traffic across service instances.

Service Orchestrators

An orchestrator handles tasks related to deploying and managing a set of services, with Orchestrator you can

- Placing services on nodes.
- Monitoring the health of services and restarting unhealthy services.
- Load balancing network traffic across service instances.
- Service discovery

Service Orchestrators

An orchestrator handles tasks related to deploying and managing a set of services, with Orchestrator you can

- Placing services on nodes.
- Monitoring the health of services and restarting unhealthy services.
- Load balancing network traffic across service instances.
- Service discovery
- Scaling the number of instances of a service

Service Orchestration Options

Popular Orchestrators

- Docker Swarm
- Kubernates
- Service Fabric
- Openshift

Cloud Services' Deployment

The old and known style is called On-Premise deployment, and it has its cons.

Today we have awesome cloud tools for deployment.

- AWS
- Microsoft Azure

Monitoring

One of the most hassle part in Microservices is **tracing**! What, How and Why this error occurred?

- Logging
- Tracing
- Monitor matrices

References

The End