Booking Show

1. Requirement

User wants to build an online movie ticket booking platform that caters to both B2B (theatre partners) and B2C (end customers) clients.

1.1. Key goals

- Enable theatre partners to onboard their theatres over this platform and get access to a bigger customer base while going digital.
- Enable end customers to browse the platform to get access to movie across different cities, languages, and genres, as well as book tickets in advance with a seamless experience.

2. Technologies

2.1. Language and Framework

2.1.1. Java Based

- **Spring Boot (Recommended)** – This framework works on top of various languages for Aspect-Oriented programming, Inversion of Control and others. It provides a platform for developers to develop a stand-alone and production-grade spring application that they can "**just run**".
- Monitoring (Actuator)
- Open API Swagger
- Spring Test
- Spring JPA

2.1.2. UI Based

1. Angular JS Vs React JS Vs Type JS (Reading)

2.1.3. Container Orchestration Tools and Services

- **Kubernetes (k8s)**
- **Docker/ Docker compose**
- **Rancher (k3s) Single Node k8s**

2.2. Database

2.2.1. SQL

2.2.1.1. Patterns

- **Private-tables-per-service** – each service owns a set of tables that must only be accessed by that service
- **Schema-per-service** – each service has a database schema that's private to that service
- **Database-server-per-service** – each service has it's own database server.

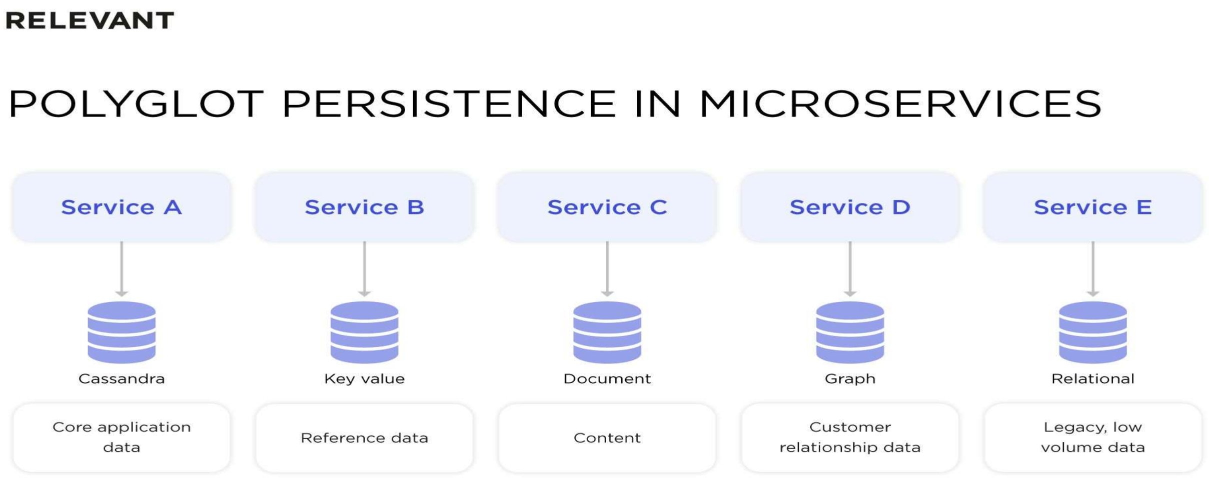
2.2.1.2. DB

1. **PostgreSQL**: PostgreSQL is an open-source object RDBMS with special emphasis on extensibility and standards compliance.
2. **Redis**: Redis is an in-memory data structure store, used as a distributed, in-memory key–value database, cache, and message broker, with optional durability.

2.2.2. No SQL (MongoDB Vs Cassandra)

1. **MongoDB:** MongoDB is the most widely used document-based database. It stores the documents in JSON objects.
2. **Cassandra:** Cassandra is an open-source, distributed database system that was initially built by Facebook (and motivated by Google's Big Table). It is widely available and quite scalable. It can handle petabytes of information and thousands of concurrent requests per second.

Reference: https://relevant.software/blog/microservices-database-management/



2.3. Integration Technologies

2.3.1. Domain Driven Architecture

The bounded context concept originated in Domain Driven Design (DDD) circle. It promotes the object model first approach to service, defining a data model that service is responsible for and is bound to. Achieving this using **REST API. (Synchronous)**

2.3.2. Event Driven Architecture

It builds with communicating over events. An event is defined as a significant change in a state. Achieving this using **Messaging Service like RabbitMQ. (Asynchronous) - Booking, Payment, SeatLock etc.**

2.4. Cloud Technologies:

Cloud computing is the on-demand availability of computer system resources, especially data storage and computing power, without direct active management by the user.

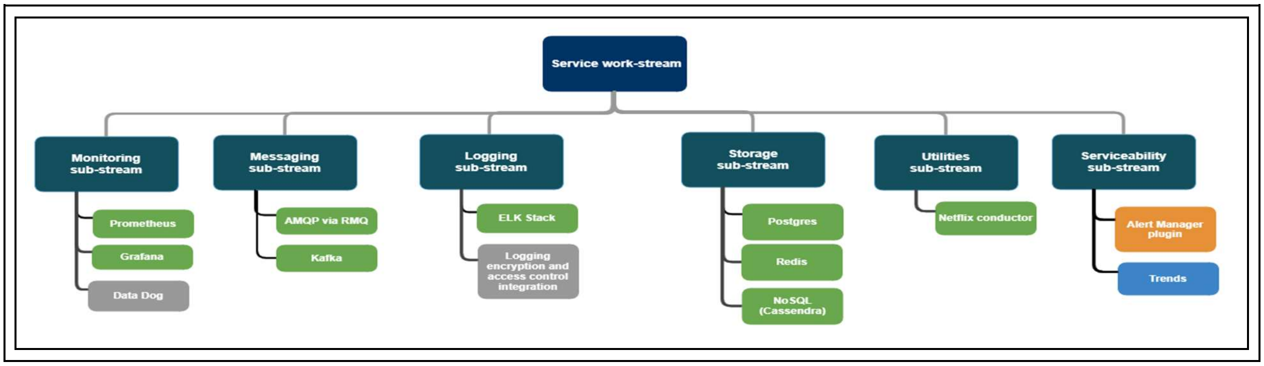
2.4.1. List some top cloud technologies

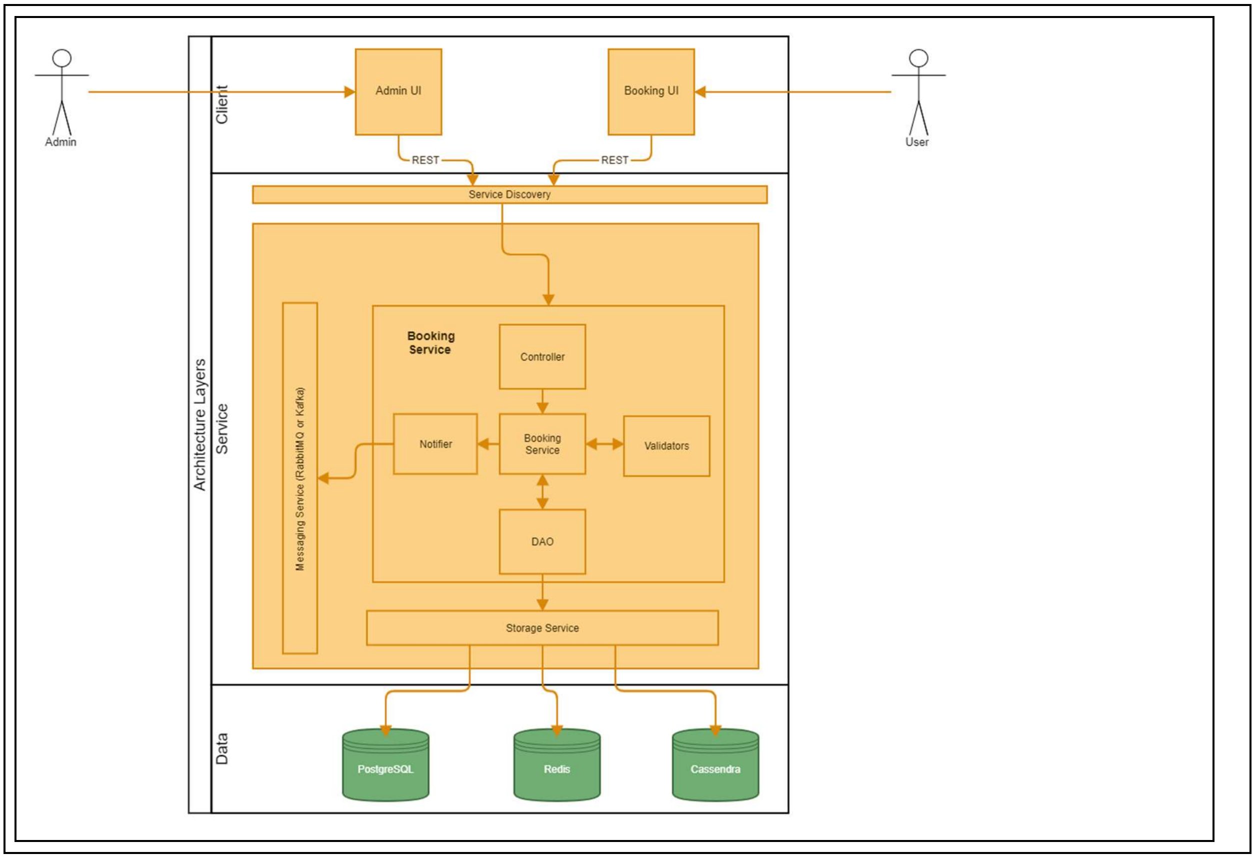
- Amazon Web Services (AWS)
- Microsoft Azure.
- Google Cloud.

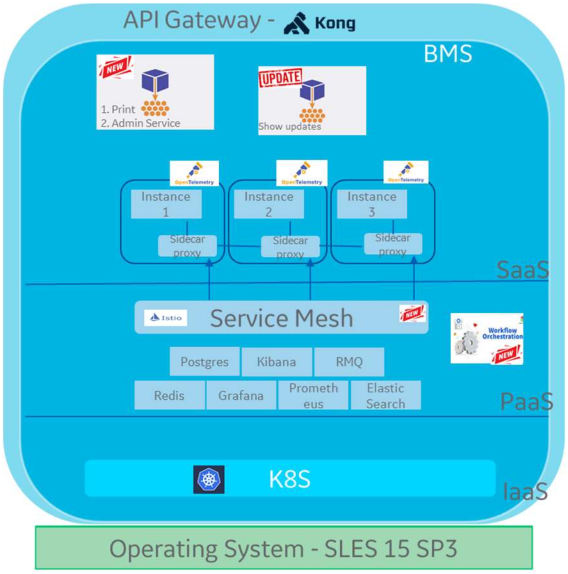
2.5. Preferred editor to build and present solutions

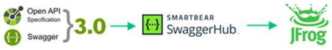
- IntelliJ IDEA

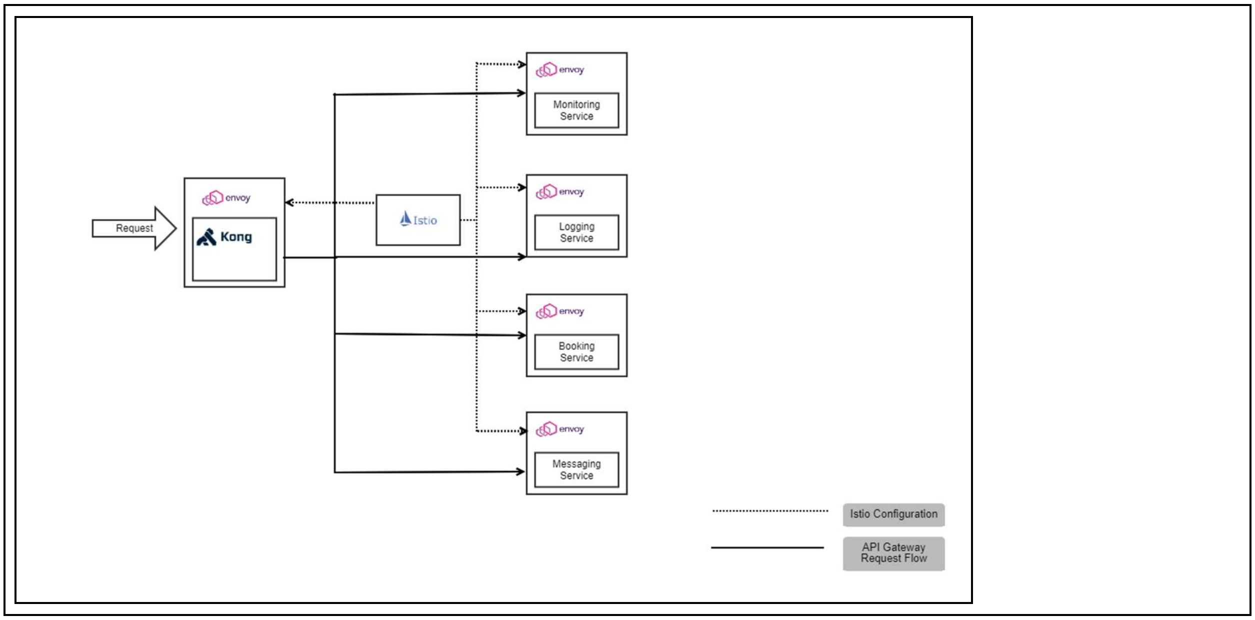
3. Architecture and Design

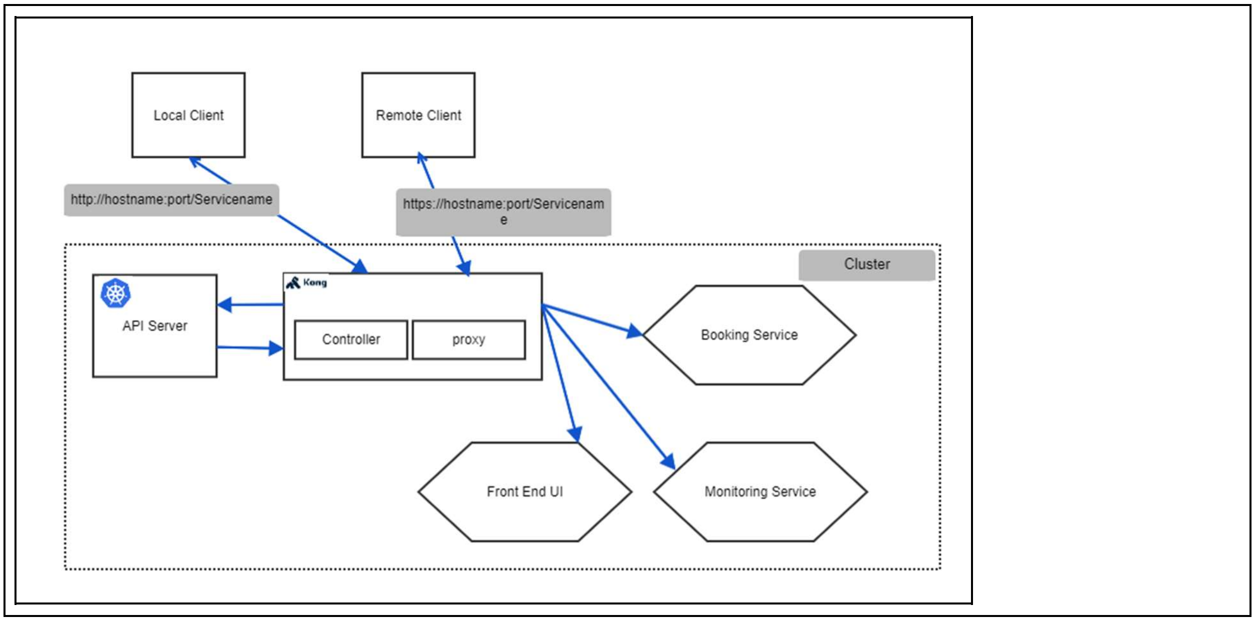


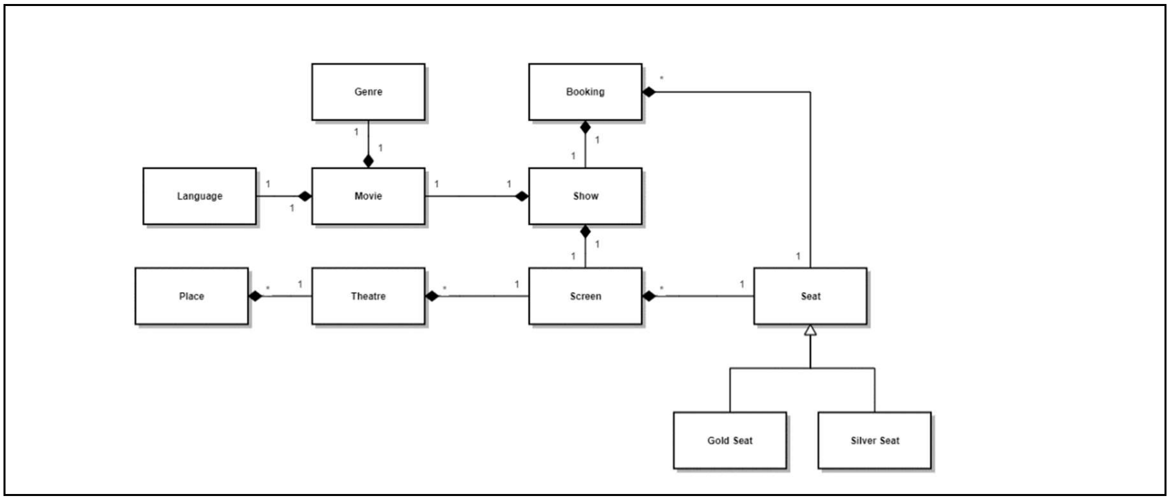

Admin
User
Architecture Layers
Client
Admin UI
Booking UI
REST
REST
Service
Service Discovery
Messaging Service (RabbitMQ or Kafka)
Booking Service
Controller
Notifier
Booking Service
Validators
DAO
Storage Service
Data
PostgreSQL
Redis
Cassendra

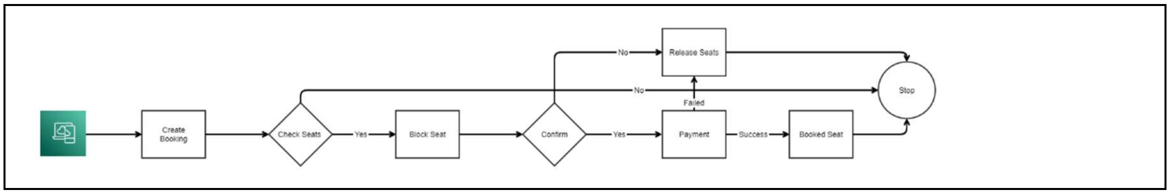

API Gateway - Kong
BMS
NEW
1. Print
2. Admin Service
UPDATE
Show updates
OpenTelemetry
Instance 1
Sidecar proxy
Instance 2
Sidecar proxy
Instance 3
Sidecar proxy
SaaS
Istio
Service Mesh
NEW
Workflow Orchestration
NEW
Postgres
Kibana
RMQ
Redis
Grafana
Prometheus
Elastic Search
PaaS
K8S
IaaS
Operating System - SLES 15 SP3


Open API Specification
Swagger
3.0
SMARTBEAR SwaggerHub
JFrog


Request
envoy
Kong
Istio
envoy
Monitoring Service
envoy
Logging Service
envoy
Booking Service
envoy
Messaging Service
Istio Configuration
API Gateway Request Flow


Local Client
Remote Client
http://hostname:port/Servicename
https://hostname:port/Servicename
Cluster
API Server
Kong
Controller
proxy
Booking Service
Front End UI
Monitoring Service





4. Monitoring

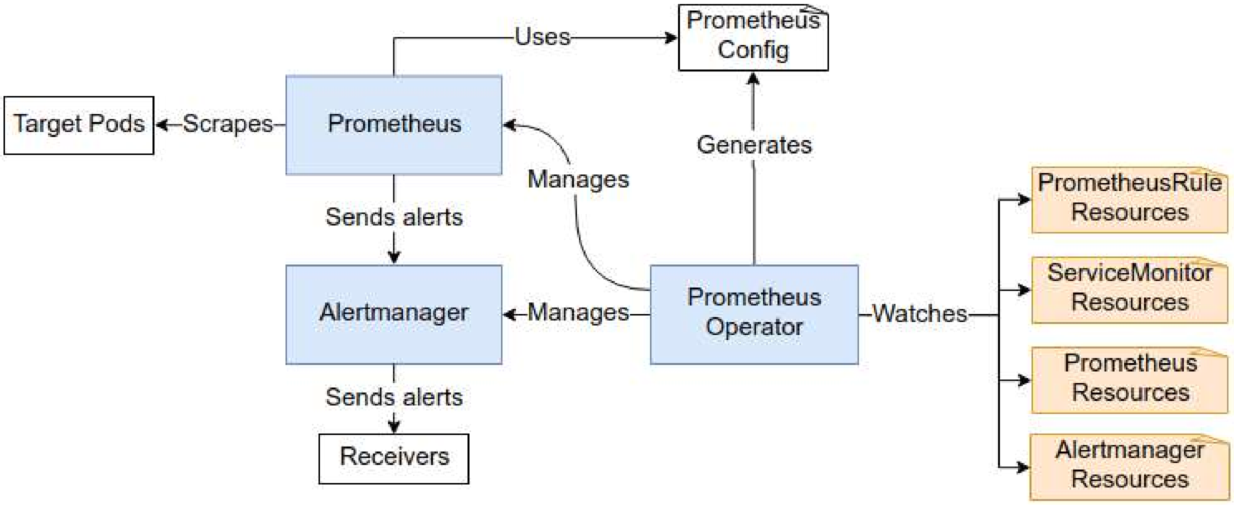
4.1.1. Prometheus operator (Architecture)

The Prometheus Operator provides Kubernetes native deployment and management of Prometheus and related monitoring components. The purpose is to simplify and automate the configuration of a Prometheus based monitoring stack for Kubernetes clusters.

The Prometheus operator includes, but is not limited to, the following features:

- **Kubernetes Custom Resources**: Use Kubernetes custom resources to deploy and manage Prometheus, Alertmanager, and related components.
- **Simplified Deployment Configuration**: Configure the fundamentals of Prometheus like versions, persistence, retention policies, and replicas from a native Kubernetes resource.
- **Prometheus Target Configuration**: Automatically generate monitoring target configurations based on familiar Kubernetes label queries; no need to learn a Prometheus

specific configuration language.



The Operator creates and acts on the following Kubernetes CRDS.

- **Prometheus:** Which defines a desired Prometheus deployment. The Operator ensures at all times that a deployment matching the resource definition is running.
- **ServiceMonitor** : Which declaratively specifies how groups of services should be monitored. The Operator automatically generates Prometheus scrape configuration based on the definition.
- **PrometheusRule** : Which defines a desired Prometheus rule file, which can be loaded by a Prometheus instance containing Prometheus alerting and recording rules.
- **Alertmanager** : Which defines a desired Alertmanager deployment. The Operator ensures at all times that a deployment matching the resource definition is running.
- **Probe:** This custom resource definition (CRD) allows to declarative define how groups of ingresses and static targets should be monitored.
- **PodMonitor**: This custom resource definition (CRD) allows to declaratively define how a dynamic set of pods should be monitored. Which pods are selected to be monitored with the desired configuration is defined using label selections? This allows an organization to introduce conventions around how metrics are exposed, and then following these conventions new pods are automatically discovered, without the need to reconfigure the system.
- **AlertmanagerConfig** This custom resource definition (CRD) declaratively specifies subsections of the Alertmanager configuration, allowing routing of alerts to custom receivers, and setting inhibit rules.

4.1.1.1. Prometheus Operator endpoint to scrape autoconfiguration

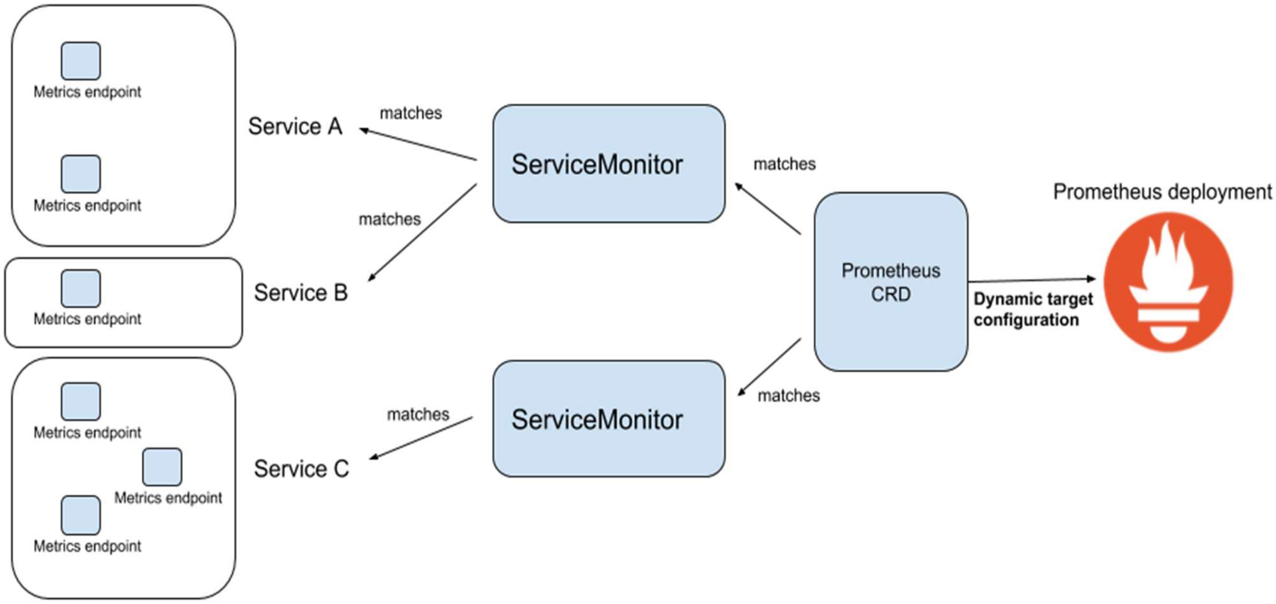
4.1.1.1.1. ServiceMonitor

The ServiceMonitor custom resource definition (CRD) allows to declaratively define how a dynamic set of services should be monitored. Which services are selected to be monitored with the desired configuration is defined using label selections? This allows an organization to introduce conventions around how metrics are exposed, and then following these conventions new services are automatically discovered, without the need to reconfigure the system.

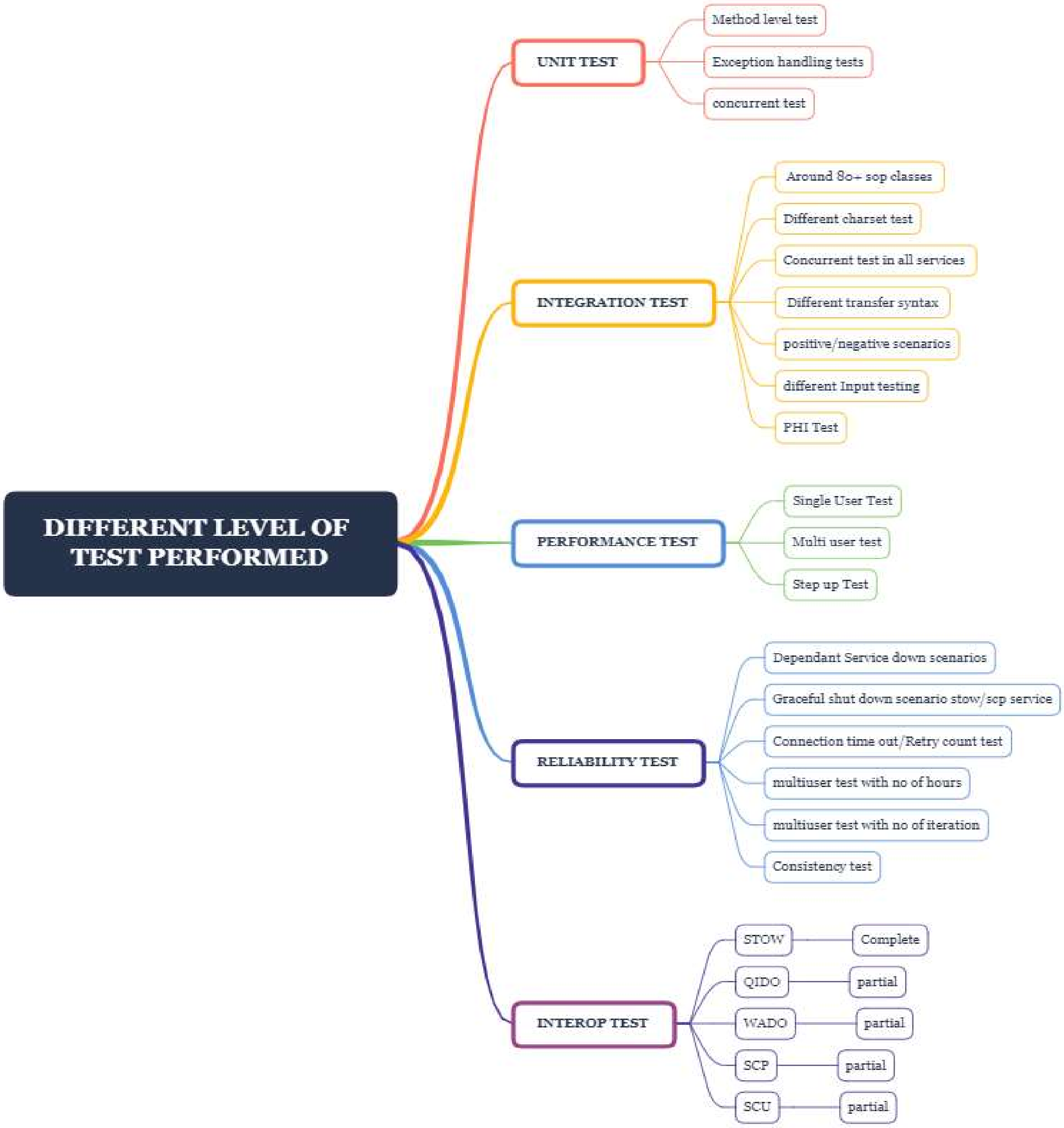
For Prometheus to monitor any application within Kubernetes an Endpoints object needs to exist. Endpoints objects are essentially listing of IP addresses. Typically, an Endpoints object is populated by a Service object. A Service object discovers Pods by a label selector and adds those to the Endpoints object.

The ServiceMonitor object introduced by the Prometheus Operator in turn discovers those Endpoints objects and configures Prometheus to monitor those Pods. The endpoints section of the ServiceMonitorSpec, is used to configure which ports of these Endpoints are going to be scraped for metrics, and with which parameters.

The Prometheus resource includes a field called `serviceMonitorSelector`, which defines a selection of ServiceMonitors to be used. By default, With the Prometheus Operator v0.19.0 and above, ServiceMonitors can be selected outside the Prometheus namespace via the serviceMonitorNamespaceSelector field of the Prometheus resource.



5. Testing Strategy



5.1. Unit Testing & Strategy

Unit tests are written as part of feature/story development to maintain a minimum of 85% condition coverage. All positive, negative and exception paths are covered through unit tests.

Unit tests are written with mocking all dependency objects, so many of the paths which cannot be covered under integration tests are covered part of unit tests. All the tests are executed in CI/CD stages in build pipeline

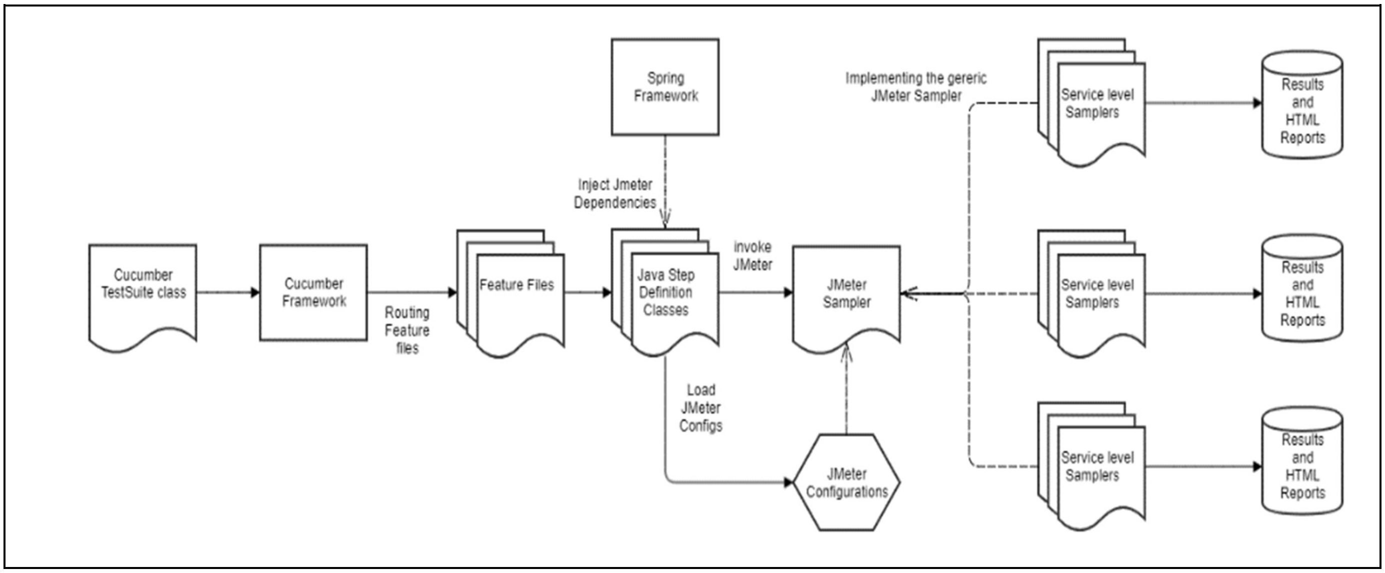
Testing Libraries/Plugins used

1. Junit - Java Unit testing framework
2. Mockito - Mocking framework
3. Power Mock - Extends Mockito with extra capabilities

Below are some of the achieved scenarios through Unit Testing

Scenario	**Description**
Mocking	By mocking the dependant objects the expected behaviour of a method is verified.
DB dependency Mocking	By mocking the db dependencies, the different expected responses are tested.
Publish/Consume message test	RabbitMQ interactions are tested by mocking RabbitMQ client dependencies.
Graceful shutdown tests	By injecting waiting threads into application container executor service, the shutdown scenarios are tested.
Retry count test	Retry scenarios are tested by checking the retry count printed in logs.

Testing Flow


Spring Framework
Implementing the gereric JMeter Sampler
Service level Samplers
Results and HTML Reports
Inject Jmeter Dependencies
Cucumber TestSuite class
Cucumber Framework
Routing Feature files
Feature Files
Java Step Definition Classes
invoke JMeter
JMeter Sampler
Service level Samplers
Results and HTML Reports
Load JMeter Configs
JMeter Configurations
Service level Samplers
Results and HTML Reports