Contents

[1- Introduction 2](#_Toc188136302)

[a. Garage Parts Management overview/Problem Definition: 2](#_Toc188136303)

[2- Requirement Overview (Functional and Non-Functional) 2](#_Toc188136304)

[a. Functional Requirements 2](#_Toc188136305)

[b. Non-Functional Requirements 2](#_Toc188136306)

[3- Assumptions 3](#_Toc188136307)

[4- System Architecture 3](#_Toc188136308)

[a. High Level Architecture Diagram 3](#_Toc188136309)

[b. High Level Components with Low Level Design 4](#_Toc188136310)

[5- How Non-Functional Requirements will be handled? 5](#_Toc188136311)

[a. Security 5](#_Toc188136312)

[b. Reliability and Fault Tolerance 6](#_Toc188136313)

[c. Availability 7](#_Toc188136314)

[d. Deployment Strategy and CI/CD 7](#_Toc188136315)

[e. Observability 8](#_Toc188136316)

[f. Extensibility 8](#_Toc188136317)

[g. Cost Optimizations 8](#_Toc188136318)

# Introduction)

## Garage Parts Management overview/Problem Definition:

B-Garage, a popular automobile repair shop in town X, currently services two-wheeler vehicles and plans to expand to four-wheelers. Their manual inventory and order management system, involving monthly checks and separate books, is no longer sufficient. Frequent stockouts within 1-2 weeks are disrupting service. This document outlines the design for an automated inventory and order management system to address this issue.

# Requirement Overview (Functional and Non-Functional)

## Functional Requirements

1. Inventory and Order should be persisted for audit requirement.
2. There are only two suppliers of parts (Local - named as Supplier-A /International - as Supplier-B)
3. Every part must have

* Threshold limit (if stock goes below this threshold, then automated order with minimum Qty should be placed)
* Supplier (Supplier-A or Supplier-B)
* Available Qty
* Minimum Order Qty

4. To get the discount benefit for orders to Supplier-B, order should be placed only between 12:00 AM to 01:00 AM

5. There is no discount benefit for order to Supplier-A, hence order to Supplier-A can be placed at any time.

6. Provide feature to add new parts, modify existing parts, parts supplier, and available qty.

## Non-Functional Requirements

1. Security: The solution should be secure and should provide proper Authentication & Authorization.
2. Availability: The solution should provide high availability with minimal downtime.
3. Extensibility: The solution should be extensible and future changes should be easy to apply.
4. Reliability: The solution should be reliable and meets all customer needs.
5. Observability: We should be able to monitor the status of the requests.
6. Performance: The solution should be highly performant and should provide latency in milli seconds.
7. Modularity: The solution should be developed using proper modularity.
8. Fault Tolerance: The solution should be fault tolerant.

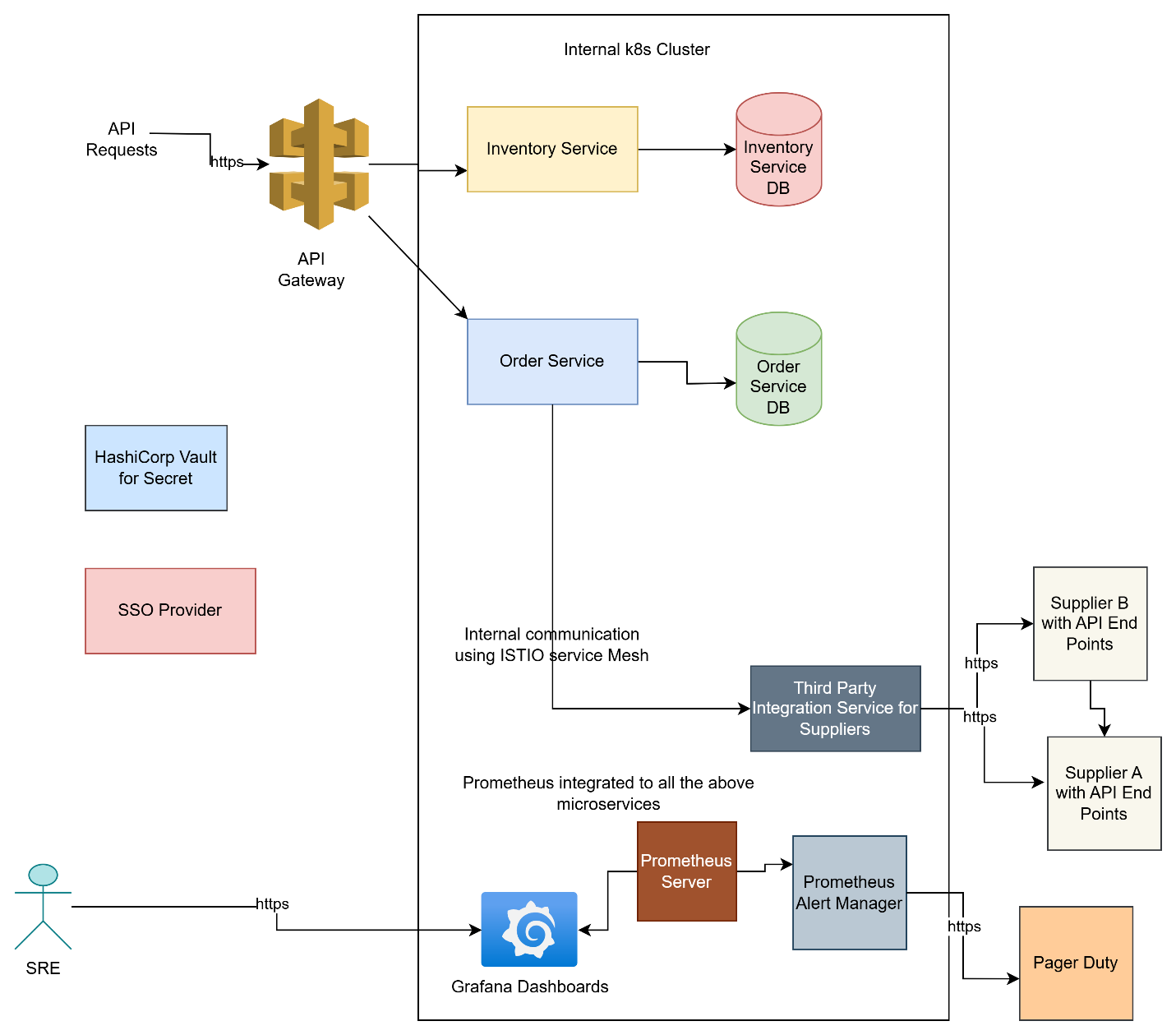
# Assumptions

* Garage owner will manually update available qty of the parts on daily basis.
* No User Interface Components are required as part of this initial design.
* Require to build a cloud native microservices based solution to solve this problem.
* For the initial design and proof of concept any in-memory database like H2 can be used.
* It is not required to fully implement this solution with full low-level code, but it is required to create the data models and major classes like controllers finalized in the design.

AL Madad Peerane Peer Gaus e Azam Dastagir

# System Architecture

## High Level Architecture Diagram



## High Level Components with Low Level Design

1. **Inventory Service**: Manages the Inventory for the Garage

**Inventory Service API**

**Base URL: /api/inventory**

**Get All Parts**

Endpoint: GET /api/inventory

Response:

200 OK with a list of Part objects

**Get Part by ID**

Endpoint: GET /api/inventory/{id}

Path Variable: id (ID of the part)

Response:

200 OK with Part object if found

404 Not Found if the part does not exist

**Create Part**

Endpoint: POST /api/inventory

Request Body: Part (details of the part to be created)

Response:

201 Created with Part object

**Update Part**

Endpoint: PUT /api/inventory/{id}

Path Variable: id (ID of the part)

Request Body: Part (updated details of the part)

Response:

200 OK with updated Part object if successful

404 Not Found if the part does not exist

**Update Part Quantity**

Endpoint: PATCH /api/inventory/{id}/quantity

Path Variable: id (ID of the part)

Request Body: int (new quantity)

Response:

200 OK with updated Part object if successful

404 Not Found if the part does not exist

1. **Automatic Inventory Check Scheduler:** Background job that reads the inventory and perform automatic orders based on the available quantity
2. **Order Service:** This service manages the order

**Order Service API**

Base URL: /api/orders

**Create Order**

Endpoint: POST /api/orders

Request Body: OrderRequest (details of the order to be created)

Response:

201 Created with Order object if successful

404 Not Found if the order creation fails

**Get Order by ID**

Endpoint: GET /api/orders/{id}

Path Variable: id (ID of the order)

Response:

200 OK with Order object if found

404 Not Found if the order does not exist

1. **Third Party Integration Service:** Used to send the actual orders to the correct supplier by calling the REST end points provided by the suppliers
2. **API Gateway:** Acts as a reverse proxy for the backend microservices (APIGEE/Kong) Gateways could be used here.
3. **Postgre SQL:** Database to store order and inventory data, for local development H2 will be used.
4. **Istio Service Mesh:** for microservices communication and security
5. **HashiCorp Vault:** For storing application level secrets
6. **Auth0/OKTA:** For supporting OAUTH2 and Open ID Connect related workflows.
7. **Observability Tools:** For monitoring and alerts on the microservices the tools like Prometheus, Grafana, PagerDuty/ZenDuty will be used.

# How Non-Functional Requirements will be handled?

## Security

We can incorporate following features to enhance security of the data collection platform

* **Security in Design:** Security should be incorporated when you design the system and identify vulnerabilities in the design by creating security threat models. Security threat models are usually created along with the architecture diagram and threat modelling tools like Microsoft Threat Modelling tools can be used to create the diagram and how the components interact with each other and it will automatically list the potential vulnerabilities in the defined architecture.
* **Automatic checks for Security Vulnerabilities:** SCA (Software Composition Analysis) tools like white-source/black-duck should be integrated with the CI/CD pipelines for checking the vulnerabilities in the Open Source Components. SAST (Security Application Testing) tools and DAST (Dynamic Application Security Testing) tools for testing the Application code for security vulnerabilities should also be integrated in the CI/CD pipelines, if any security vulnerability found then build should be failed. These tools provide static code analysis to identify OWASP security issues like SQL Injections, Broken access control etc. and provide mitigations strategies. Also, container security scans like Trivy Scans could be integrated with the CI/CD pipelines to check the container images for the security vulnerabilities.
* **Penetration Testing:** Penetration testing should be performed along with regular QA to identify the security risks and these should be taken in every sprint to provide the resolutions.
* **Regular Security Reviews:** Bi Weekly security reviews should be done for the Securities vulnerabilities reported by the tools like White-source/Veracode and this should be added to the product backlog as technical debt and prioritize in every Sprint to fix these security issues.
* **Data Security:** Security should be applied to the data as well, we need to provide proper data masking for field level security, encryption in transit and rest for the data security when it is stored in database and move between different components for the processing. Data must be deleted once it is no longer required. And we need to adhere to the compliance requirements like GDPR.
* **Role Based Access Controls (RBAC):** Role based access controls must be defined at the user level using the Principle of Least Privilege.
* **Use of TLS/SSL:** All the data transfer should be using the SSL/TLS in the form of HTTPS.
* **Security Trainings:** Regular security trainings must be conducted for the development team to make them aware of security requirements and they are the first line of defense for providing security for the sensitive data.
* **Security during Deployment:** All the internal microservices should be deployed in the private cloud cluster with no access from internet, we can leverage public load balancer or API gateway on top of these microservices with proper ingress/egress rules defined.

## Reliability and Fault Tolerance

Following strategies could be used for making the Data Collection Platform highly reliable and Fault Tolerant.

* **Multi Region Deployments:** Multi Region Cloud Deployment could be used with automatic failover support.
* **Multi Region Replication for the Data:** Data should be replicated across multiple regions.
* **Use Observability Tools to Monitor the State of the services:** Use of observability stack like Prometheus, Grafana along with automatic alerting systems like PagerDuty/ZenDuty could be utilized here to publish the metrics related to throughput for event based services and latency for rest based services, the dashboards should be created to track the uptime of the system, if system goes down then automatic alerts should be generated and notification must be sent using tools like PagerDuty/ZenDuty.
* Design for failover: The user’s traffic should be automatically switch to the redundant resources when failure occurs.
* **Microservices Patterns like Circuit Breaker, Bulkhead, retries must be used:** Using this pattern will increase the reliability of the Data Collection Platform.
* **Regular Performance testing and Fault Tolerance testing:** Testing must be scheduled in every sprint for testing the performance of the system along with fault tolerance where faults will be introduced in the system to identify how it behaves in case of the failures.
* **Use of managed databases where ever possible:** Managed databases like Azure Managed Instance of Apache Cassandra, Azure Managed Redis could be utilized as they will provide High Availability/Reliability/Security by default.
* **Use of Event Driven Architecture:** Use of message queues to store the messages so that they should be processed even if the consuming services are down.

## Availability

For ensuring high availability of the Data Collection Platform following strategies could be used.

* **Service Replication:** Deploying the microservices on multiple Kubernetes pods and exposing them as a service so that proper load balancing should be done.
* **Horizontally Scaling the Microservices:** Defining the proper HPA for the microservices to support autoscaling to reduce the load on individual pods and pods should be auto scaled to handle the increase in the load, followed by down scaling once the load decreases to save the costs.
* **Blue Green Deployment:** This can be used as a deployment strategy with Kubernetes, this could help in avoiding downtime while deploying new features.

## Deployment Strategy and CI/CD

* Deployment using Container Orchestration framework Kubernetes using Helm Charts
* Git Ops tools like Argo CD for managing Kubernetes Clusters
* Use of Rolling upgrades for the deployments to minimize the downtime, blue green deployment strategy could also be used for critical microservices if we need to choose availability vs extra cost in the tradeoffs.
* Rollback strategy in case of failures
* Integration of SONAR, White source, Trivy (For Container security) Scans for identifying and fixing the security vulnerabilities, in case of any critical/blocker vulnerabilities the build should fail.
* Defining KEDA (Kubernetes Event-Driven Autoscaler) to define the HPA (Horizontal POD Scaler) to define the autoscaling based on CPU/Memory Utilizations in order to utilize the K8S POD memory/CPU efficiently.

## Observability

**Use of Observability and Alerting Tools:** Use of observability stack like Prometheus, Grafana along with automatic alerting systems like PagerDuty/ZenDuty could be utilized here to publish the metrics related to throughput for event based services and latency for rest based services, the dashboards should be created to track the uptime of the system, if system goes down then automatic alerts should be generated and notification must be sent using tools like PagerDuty/ZenDuty.

## Extensibility

Principles like Modularity and Cohesion, Loose coupling, Encapsulation, Abstraction, DRY (Don’t repeat yourself) Principle, SOLID principles, Scalability, Observability, Reliability and Fault Tolerance, Testability etc. By using the microservices based architecture the Single Responsibility Principle is followed in the design

## Cost Optimizations

System should support Affordability and support cloud cost optimization, use of proper autoscaling to scale down when the load decreases to save the compute costs. Cloud resource usage should be monitored regularly and automatic shutdown of the resources when not in use (weekends, out of working hours and company holidays) can be utilized for development environments to reduce the engineering costs.