# Disaster-Resistant Microservices Using Metaheuristic Algorithm

## SUMMARY


1. Motivation Behind the Project: Microservices, while highly effective in traditional cloud environments, face critical limitations when deployed in extreme conditions, such as those encountered in disaster recovery scenarios.
2. Type of project: Development & research project : The project includes the practical development of microservices, containerization, and orchestration using Docker and Kubernetes. At the same time, it incorporates research in applying metaheuristic algorithm specifically ACO for intelligent routing in adverse environments.
3. Overall design of project : The project leverages Docker-based deployment to ensure that microservices are scalable, isolated, and resilient to failure. We are using ACO algorithm to find the best path for rerouting service using it. ACO mimics the behaviour of ants foraging for food, where they leave pheromone trails to mark optimal paths. The solution proposed in this project involves the integration of disaster-resistant microservices, with a focus on enhancing, data transfer efficiency, and system adaptability. The solution approach includes the following key components:
    1. Docker-Based Deployment: The microservices are deployed in Docker containers, which provide a lightweight and isolated environment for each service.
    2. Ant Colony Optimization (ACO): Ant Colony Optimization (ACO) is used for routing data. ACO determines the optimal routing paths for data transmission by simulating the pheromone-based path selection behaviour of ants.

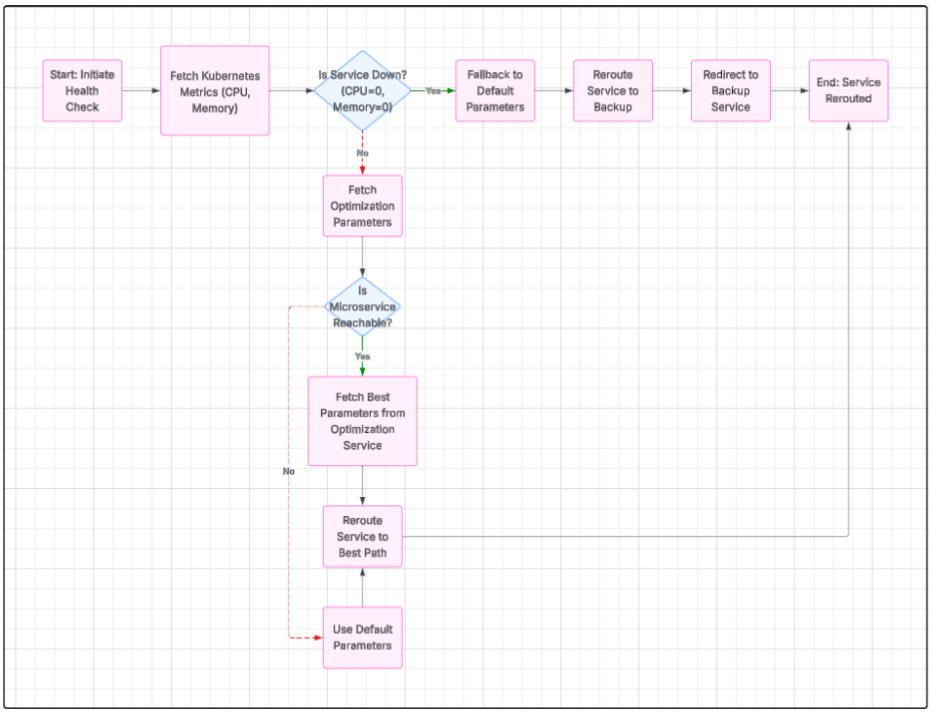
Functional Requirements

1. Microservice Architecture: The system consist of multiple independent microservices.

2.Containerization: The service is packaged using Docker to allow portability and consistent behaviour across different environments.

3.Routing with ACO:Each packet uses historical data and simulated pheromone trails to determine the best path across a delay-tolerant network.

4.Monitoring:Used Python and Kubernetes metrics to scrape and store metrics from services.




## 4. Features built and language used

1. Features:
    1. RESTful Flask API for ACO parameter tuning
    2. ACO algorithm for intelligent path finding
    3. Dockerfile and container deployment for each service
    4. Kubernetes monitoring and health checks
    5. Postman for API testing and validation
2. Languages and Tools:
    1. Python : Flask, ACO
    2. Docker and Docker Compose
    3. YAML : Kubernetes deployments (minikube)
    4. Postman : Testing
    5. Kubernetes CLI : Monitoring & Orchestration
    6. Java and Spring Tool suit : Microservice

5. Proposed Methodology:The methodology combines containerized microservices with ACO-based rerouting. Each service is independently deployable in Docker containers. When a service fails or a route becomes suboptimal due to delay or node loss, the ACO algorithm dynamically reroutes traffic

using historical data and updated pheromone values. Kubernetes ensures load balancing, health checks, and restarts failed containers, making the system fault-tolerant and adaptive.

6. Algorithm/Description of the Work : The Ant Colony Optimization algorithm forms the core of routing intelligence in this system. It simulates multiple "ants" traversing possible paths between a start and end node. Each path's performance influences its pheromone level—better-performing paths retain more pheromone, making them more attractive for future ants. Over iterations, suboptimal paths fade, and the optimal route emerges. The algorithm is invoked when there's a routing decision to be made, especially when failures or network issues are detected. Parameters like alpha (pheromone importance), beta (distance heuristic), evaporation rate, number of ants, and number of iterations are configurable via a Flask-based API.

7. Division of Work

1) Manya Rai – Made microservice, containerized it on Docker, orchestrated on Kubernetes, integrated and modified the Python code accordingly to fetch CPU and memory usage
2) Varun Bhardwaj – Wrote the basic ACO code and made report
3) Sumant Sharma – Wrote the ACO code and made report

8. Result : The system performed reliably under simulated failure conditions. With primary paths disabled, the ACO-based logic successfully rerouted traffic to backup services, as verified through Python logs and Kubernetes metrics. Docker ensured services remained lightweight and portable, and Kubernetes automated failover recovery. Postman test cases confirmed that the Flask API returned accurate routing parameters and health states, indicating successful integration of intelligent routing with resilient infrastructure.

9.Conclusion : The project successfully demonstrated the ability to implement a resilient microservice architecture using Kubernetes, Docker, and Python. By leveraging Kubernetes for container orchestration, we ensured that the deployment and scaling of microservices were streamlined, making the system adaptable and fault- tolerant. One of the standout features was the health-check and rerouting mechanism embedded within the Python logic. This mechanism ensured that when a primary service failed, the system quickly detected the issue and rerouted traffic to a backup service, maintaining system uptime and minimizing disruptions.

Signature of Students:
Name : Manya Rai
Sumant Sharma
Varun Bhardwaj

Signature of Supervisor:
Name: Dr. Amarjeet Prajapati

# CHAPTER 1 : INTRODUCTION

## 1.1 General Introduction

In the era of modern computing, microservices have fundamentally changed the way large-scale applications are built and deployed. Unlike traditional monolithic architectures, microservices break down complex applications into small, independent services that communicate with each other via APIs. Each microservice is responsible for a specific functionality, and each can be deployed, maintained, and scaled independently. This approach has revolutionized the development of web applications by offering more flexibility, scalability, and maintainability.

Microservices have found widespread adoption in the cloud-native environment, where resources are abundant, and network connectivity is reliable. However, these architectures face significant challenges when deployed in extreme environments, such as space missions or natural disaster zones, where conditions like network latency, intermittent connectivity, and unpredictable failures make traditional microservices unfit for use.

Disaster-prone areas such as earthquake zones, flood-hit regions, or remote areas, reliable communication networks are often unavailable, making the deployment of cloud-based or traditional microservices infeasible.

This project proposes a solution to these problems by developing a Disaster-Resistant Microservice architecture, utilizing advanced metaheuristic algorithms and containerized deployments. By integrating Ant Colony Optimization (ACO) for testing and rerouting accordingly, this architecture is designed to enhance routing efficiency, and system adaptability in high-latency, failure-prone environments.

The core focus of this research is to create a resilient, adaptive, and failure-resistant microservice system capable of functioning in the harshest environments, such as those found in space missions or disaster recovery scenarios.

## 1.2 Problem Statement

Microservices, while highly effective in traditional cloud environments, face critical limitations when deployed in extreme conditions, such as those encountered during space missions or in disaster recovery scenarios. The following problems are commonly faced by traditional microservices in these environments:

- **High Latency**: In space missions, communication networks experience significant delays due to the vast distances between Earth and the satellite or spacecraft. For example, signals transmitted to and from Mars can take anywhere between 20 to 40 minutes one-way. Such delays make it impossible to rely on real-time communication and prevent microservices from functioning optimally.
- **Network Failures**: Both space-based operations and disaster recovery environments suffer from frequent network failures. In disaster zones, physical infrastructure such as communication towers, power lines, and internet cables may be destroyed, leading to intermittent or complete loss of connectivity.
- **Intermittent Connectivity**: In remote locations or disaster zones, internet connectivity is either intermittent or completely unavailable. This makes it difficult for microservices to maintain communication with the rest of the system, leading to service disruptions and data loss.
- **Failure Recovery**:

  Traditional microservice systems are not designed to handle the failure-prone nature of extreme environments. The absence of infrastructure and reliable communication networks makes it necessary for the system to be self-healing and capable of recovering from failures automatically.

These challenges necessitate the development of a robust, adaptive, and failure-resistant microservice system that can handle these extreme conditions, ensuring continuity of service in disaster recovery operations, and remote communications.

## 1.3 Significance of the Problem

The significance of this project lies in its potential to bridge the gap between traditional cloud computing and the extreme conditions faced by disaster-resistant systems. As the world becomes more reliant on technology for and disaster management, it is essential to design systems that can continue to operate in environments where conventional microservices would fail.

The novelty of this research lies in the integration of disaster-resistant microservices using advanced metaheuristic algorithms for data routing. The project employs Ant Colony Optimization (ACO), a nature-inspired algorithm that is highly effective in determining optimal data routes in delay-tolerant networks. ACO mimics the behaviour of ants foraging for food, where they leave pheromone trails to mark optimal paths. This approach is particularly useful in space-based and disaster environments, where network delays and failures make traditional routing protocols ineffective.

Additionally, the project leverages Docker-based deployment to ensure that microservices are scalable, isolated, and resilient to failure. Docker containers provide the necessary isolation to ensure that a failure in one microservice does not propagate across the entire system. The self-healing properties of Docker also help in automatically recovering failed microservices, minimizing downtime and improving system resilience.

The combination of ACO for delay-tolerant networking and Docker for containerization offers a powerful solution to the challenges posed by extreme environments, making this approach highly innovative and impactful.

## 1.4 Empirical Study

As part of this project, an empirical study was conducted to understand the limitations of existing microservice systems in extreme environments and to assess the suitability of various tools and techniques for overcoming these challenges.

- **Existing Tool Survey**: A comprehensive survey of existing microservices deployment tools was conducted, with a focus on Kubernetes, Docker, and other container orchestration tools. These tools have been widely adopted in cloud-native environments and offer features such as scalability, automated deployment, and fault tolerance. However, these tools primarily assume the availability of a reliable network and stable infrastructure. In contrast, space-based operations and disaster recovery environments suffer from intermittent connectivity and network delays, rendering these tools insufficient for use in such conditions. The survey also reviewed existing fault detection and routing tools, revealing that current systems do not efficiently handle the high latency and failure-prone nature of delay-tolerant networks.
- **Experimental Study**: To simulate the conditions found in space missions and disaster zones, several experiments were conducted using high-latency network simulations. These simulations replicated the challenges of intermittent connectivity, delayed communication, and failure recovery. The results demonstrated the limitations of traditional microservice systems in such

environments, particularly in terms of communication latency, failure recovery, and data transfer efficiency. These findings provided critical insights into the need for more adaptive, resilient systems and validated the need for metaheuristic algorithms such as ACO.

This empirical study has contributed valuable data that helped shape the proposed solution, guiding the selection of appropriate tools and techniques to address the challenges faced by microservices in extreme environments.

## 1.5 Brief Description of the Solution Approach

The solution proposed in this project involves the integration of space-based and disaster-resistant microservices, with a focus on enhancing fault tolerance, data transfer efficiency, and system adaptability. The solution approach includes the following key components:

3. Docker-Based Deployment: The microservices are deployed in Docker containers, which provide a lightweight and isolated environment for each service. Docker ensures that microservices can scale according to demand and automatically recover from failures. By using containerization, the system becomes cloud-native, meaning it can be easily deployed on any cloud platform or edge device, offering flexibility and scalability.
4. Ant Colony Optimization (ACO) for DTN Routing: To address the issue of high latency and intermittent connectivity in space-based and disaster recovery environments, Ant Colony Optimization (ACO) is used for routing data in Delay-Tolerant Networks (DTN). ACO determines the optimal routing paths for data transmission by simulating the pheromone-based path selection behaviour of ants. This method adapts to network delays and provides efficient routing in environments where traditional routing algorithms are ineffective.
5. Resilient Microservice Architecture: The overall system architecture is designed to be resilient to failures. Microservices can self-heal and scale automatically based on system demand and failure events. This ensures that the system remains operational even in the event of network failures or other disruptions.

## 1.6 Comparison of Existing Approaches to the Problem Framed

Existing approaches to microservice architecture are primarily designed for environments with stable infrastructure and low-latency networks. Tools such as Kubernetes and Docker provide scalability and

fault tolerance, but they do not effectively handle the challenges posed by high-latency, failure-prone environments like space missions or disaster zones.

- Traditional Microservices: Conventional microservices are not equipped to handle the high latency and frequent failures that occur in space-based or disaster-prone environments. These systems rely on continuous connectivity and low-latency networks to function optimally, which is often unavailable in extreme conditions.
- Current Solutions for Delay-Tolerant Networks: Some existing solutions use delay-tolerant networking (DTN) protocols, but they lack the adaptability and efficiency required for real-time decision-making in space missions or disaster recovery scenarios. Traditional DTN solutions do not effectively address the dynamic nature of data routing in failure-prone, high-latency environments.

In contrast, the solution proposed in this project integrates Ant Colony Optimization for DTN routing, which provides enhanced adaptability and efficiency in handling failure-prone, high-latency environments. This approach offers a more comprehensive and robust solution than current methodologies.

# CHAPTER 2 : LITERATURE SURVEY

## 2.1 Summary of Papers Studied

1. **B. Zhang, L. Zhang, Z. Li, and X. Luo, "Ant Colony Optimization-Based Routing for Delay-Tolerant Networks," IEEE Transactions on Wireless Communications, vol. 18, no. 2, pp. 1051-1062,Feb,2019.**

   Delay-Tolerant Networks (DTNs) are often used in scenarios where the traditional communication protocols, which depend on continuous connectivity, fail to work due to intermittent connectivity, high latency, or unreliable links. This is commonly seen in space-based applications, remote areas, and disaster zones. In these environments, data transfer must still occur, albeit with more advanced and adaptable routing protocols. Traditional routing mechanisms, such as Flooding or Epidemic Routing, face issues such as high overheads and inefficiencies due to their reliance on frequent communication paths.

   The paper by B. Zhang et al. presents Ant Colony Optimization (ACO) as a novel solution to improve the routing mechanism in DTNs. ACO, inspired by the natural foraging behaviour of ants, involves agents (in this case, virtual ants) that explore and lay down pheromone trails on paths they take. These pheromones are then used to guide subsequent ants towards the most successful and efficient paths. Over time, paths that result in successful delivery of data are reinforced with more pheromones, making them more likely to be chosen in future routing decisions.

   Methodology and Approach:

   The authors propose an ACO-based routing algorithm for DTNs, which adapts to the dynamic nature of the network. In their model, ACO is used to find optimal data transmission paths in environments with high latency and intermittent connectivity. By simulating the behaviour of ants, the algorithm searches for paths that minimize delay and maximize the reliability of message delivery. The key advantage of this approach is its ability to self-adjust based on network conditions, unlike traditional protocols that use static routing strategies.

   In their approach, the ACO algorithm is used to optimize the routing of data packets. Each packet's path is decided based on the availability of routes, the time delay, and the probability of successful delivery. The algorithm continuously adapts by reinforcing successful paths and

updating the pheromone levels. This ensures that the network can adjust to changing conditions, making it especially valuable in space missions or disaster recovery scenarios where network connectivity cannot be guaranteed.

Results and Conclusion:

The proposed ACO-based routing strategy demonstrated an improvement in data delivery efficiency in terms of both delay and reliability when compared to traditional routing protocols. This makes it an ideal candidate for delay-tolerant applications such as satellite communication systems, where network disruptions are frequent, and high-priority data needs to be reliably transmitted despite the challenges. The authors conclude that ACO provides a flexible and adaptive solution, ensuring effective communication even in highly unpredictable environments.

2. **H. M. S. Al-Sabahi and A. Al-Dubai, "Enhancing Microservice Reliability in Adverse Environments," IEEE Access, vol. 7, pp. 41550-41562, 2019.**

Microservice architectures have gained significant traction due to their modularity, scalability, and flexibility. However, in environments with unreliable or intermittent network connectivity, such as space missions or during disasters, microservices face significant challenges. Microservices rely on stable communication links to exchange data between services, which is often not possible in remote or disrupted environments. This paper investigates strategies to enhance the reliability and resilience of microservices, ensuring that these architectures can continue functioning even in environments where traditional cloud-based solutions are ineffective.

Methodology and Approach:

The authors present a self-healing mechanism to enable microservices to detect failures and autonomously recover from them, which is crucial in unreliable environments. The microservices can self-manage by recognizing failure events and activating self-healing procedures to re-establish the necessary services or connections. This approach helps minimize downtime and keeps essential services running even when network issues arise. In addition, the paper discusses adaptive service discovery, which allows microservices to dynamically locate and interact with other services despite changes in network conditions. This ensures that services can find one

another even when the network is disrupted.The authors also propose the use of dynamic fault detection, which is a mechanism to monitor the health of the microservices. It is integrated into the architecture to monitor the health status and provide feedback to the system. When faults are detected, the system can adapt by re-routing service requests to other available services. This capability helps to prevent complete service failure in cases where network conditions prevent normal service operations.Additionally, the paper emphasizes the need for resilience strategies that involve continuously monitoring the service environment and adjusting the microservices' configuration based on real-time performance metrics.

Results and Conclusion:

The authors show through simulations that their proposed approach significantly improves the reliability and robustness of microservice-based systems in environments with intermittent connectivity. Their mechanisms for self-healing, adaptive service discovery, and fault detection provide a holistic framework for ensuring that microservices continue functioning even when the network is unstable. This approach is particularly valuable for mission-critical applications, such as space exploration, disaster response, and remote infrastructure where maintaining service continuity is crucial for success.

The paper concludes that these techniques enhance the operational resilience of microservices, ensuring that they can provide reliable service even under adverse network conditions. The research is an essential contribution to the growing body of knowledge on building resilient distributed systems that can operate in challenging environments.

3. **P. Patel, S. M. Chavan, and S. K. Shukla, "Resilient Microservice Architectures for Space Missions," IEEE International Conference on Space Mission Challenges for Information Technology (SMCIT), pp. 50-57, 2020.**

Space missions face significant challenges, primarily due to the unique environment of space, where communication is subject to high latency, low bandwidth, and intermittent connectivity. Traditional cloud computing systems, which rely on continuous communication, cannot be effectively employed in such scenarios. As a result, space-based applications such as satellite communication, rover operations, and space exploration need a more flexible and adaptable architecture. This paper addresses the design of resilient microservice architectures that can function under these extreme conditions.

Methodology and Approach:

The authors propose using containerization to deploy microservices. By containerizing the microservices, they are able to operate in isolated environments with all necessary dependencies bundled within the container. This ensures that the services can run independently of the fluctuating availability of the network, which is essential in space-based applications. They also introduce adaptive routing protocols to ensure efficient data transmission between services despite the limitations of space communication networks.The paper emphasizes that traditional client-server architectures are not suitable for space missions because they rely on stable connections and low latency, which are not available in space environments. Therefore, the authors advocate for decentralized microservices that can work independently and exchange data in a distributed manner. The microservices communicate with each other through local storage and caching when there is a lack of connectivity. Additionally, when a connection is available, the data is synchronized.The authors also discuss the modular approach in space missions, which enables the system to scale by adding new services without disrupting the overall architecture. They highlight the importance of automated orchestration, ensuring that the system can dynamically manage microservices without manual intervention, even in isolated environments like Mars or the Moon.

Results and Conclusion:

The proposed architecture was evaluated using simulation models, demonstrating that containerized microservices are more resilient than traditional methods in space applications. The results indicated that modular, decentralized systems are more adaptable and efficient for operations in environments where real-time network communication cannot be guaranteed. The paper concluded that this approach could be used to develop reliable systems for space missions that require continuous operation under harsh conditions.The research significantly contributes to the design of autonomous, scalable, and self-managing systems for space exploration, enabling resilient architectures that can function effectively in extreme environments where traditional IT infrastructure would fail.

## 2.2 Integrated Summary of the Literature Studied

**1. B. Zhang, L. Zhang, Z. Li, and X. Luo, "Ant Colony Optimization-Based Routing for Delay-Tolerant Networks," IEEE Transactions on Wireless Communications, vol. 18, no. 2, pp. 1051-1062, Feb. 2019.**

The study presented by B. Zhang et al. demonstrates how Ant Colony Optimization (ACO) can be applied to Delay-Tolerant Networks (DTNs) to improve routing efficiency in environments with intermittent connectivity and high latency. In DTNs, traditional routing protocols fail due to their reliance on continuous connectivity, making them unsuitable for applications in space exploration and disaster zones. This research suggests that ACO, a biologically inspired algorithm, is ideal for these environments. ACO works by simulating the way ants leave pheromone trails to mark successful paths, which allows the system to adaptively select the best routing paths based on current network conditions.

The paper emphasizes that ACO provides a dynamic solution that adjusts routing decisions in real-time based on the evolving state of the network. Unlike traditional routing protocols that use fixed routes, ACO is able to adapt to network disruptions by reinforcing the most reliable paths. This is particularly crucial for networks like space missions, where connectivity is often disrupted and data transfer delays are common. The authors propose that ACO can increase reliability and efficiency in DTNs by ensuring data reaches its destination even when traditional protocols would fail. By using ACO, networks in space missions and disaster recovery scenarios can maintain communication, ensuring that critical data continues to flow despite challenging conditions.

The implication of this research is significant for applications such as remote monitoring of space-based systems, where communication links are often intermittent, and data must be routed over long distances. ACO's ability to dynamically adjust paths and ensure that messages are delivered efficiently in these circumstances makes it an essential tool for the development of delay-tolerant communication networks in harsh environments.

**2. H. M. S. Al-Sabahi and A. Al-Dubai, "Enhancing Microservice Reliability in Adverse Environments," IEEE Access, vol. 7, pp. 41550-41562, 2019.**

The paper by H. M. S. Al-Sabahi and A. Al-Dubai explore the challenges of deploying microservices in environments with unreliable networks and adverse conditions such as those encountered during natural disasters or in space missions. While microservices are popular in cloud computing environments, their dependency on stable network connections limits their applicability in scenarios where connectivity is intermittent or degraded. The authors highlight the need for resilient microservice architectures that can continue functioning despite network failures, high latency, and inconsistent service availability.

The paper suggests several solutions to improve the reliability of microservices, including self-healing mechanisms that automatically detect and recover from failures, as well as adaptive service discovery that helps services find and communicate with other services even in unstable network conditions. The paper also highlights dynamic fault detection as a critical component for ensuring the availability and performance of services, especially in environments where service interruptions are common.

The approach presented in the paper is relevant to applications in areas like disaster recovery, where critical services need to function despite ongoing network disruptions. By incorporating self-healing and dynamic adaptation, microservices can be made more resilient, ensuring that mission-critical applications continue to operate. Furthermore, these strategies are highly applicable in space-based environments, where low-latency communication is often impossible, and data transfer must occur under highly unpredictable conditions.

The importance of this paper lies in its contribution to the design of microservice-based systems that are not only scalable but also capable of functioning effectively in environments with poor or intermittent connectivity. By introducing mechanisms that make microservices more adaptive and fault-tolerant, this research enables the use of microservices in extreme conditions, thereby expanding their use cases in space exploration, disaster management, and other fields with challenging network environments.

**3. P. Patel, S. M. Chavan, and S. K. Shukla, "Resilient Microservice Architectures for Space Missions," IEEE International Conference on Space Mission Challenges for Information Technology (SMCIT), pp. 50-57, 2020.**

The paper by P. Patel et al. addresses the resiliency of microservice architectures specifically for space missions. Space missions face unique challenges, such as high latency, intermittent communication, and harsh environments that make it difficult for traditional cloud architectures to function effectively. The authors propose solutions for adapting microservices to these conditions, with an emphasis on ensuring that space-based applications remain reliable and robust despite these extreme challenges.

One of the key strategies discussed is the use of containerization to package microservices with all the necessary dependencies. This ensures that microservices can run in isolated environments, which is critical in space, where continuous network connections cannot be guaranteed. The authors also discuss the importance of adaptive routing protocols that allow data to be transmitted efficiently despite the unpredictable nature of space communication. These protocols are designed to dynamically adjust the routing paths based on network conditions, ensuring that data reaches its destination even when the network is disrupted.

# CHAPTER 3 : REQUIREMENT ANALYSIS AND SOLUTION APPROACH

## 3.1 Overall Description of the Project

The project aims to build a fault-tolerant microservice architecture using Docker and Kubernetes (Minikube) with intelligent rerouting and optimization capabilities. In a distributed microservice environment, services are prone to failure due to high load, crashes, or unavailability. To ensure seamless functioning, the system must continuously monitor the health of deployed services and reroute traffic when a service goes down. The primary requirements include implementing service health checks based on CPU and memory metrics, determining the best route using optimization algorithms, and enabling dynamic rerouting to alternative services when failure is detected.

Functionally, the system must provide a way to check the health status of services by querying endpoints and evaluating resource usage. If a service is unhealthy, it should redirect traffic to another healthy instance or backup service. The system should also consult an optimization microservice (or module) to select the most efficient path based on latency or system metrics. Non-functional requirements emphasize performance (quick detection and rerouting), fault tolerance (seamless fallback), scalability (ability to handle multiple services), and modularity (plug-and-play components). Additionally, the system should be observable, with proper logging and status updates to track rerouting decisions and system health.

Technologies used in this project include Docker for containerizing microservices, Kubernetes (Minikube) for orchestration, Python for logic related to health checks and rerouting, and optional use of optimization methods (like Genetic Algorithms) to find optimal routing paths. Kubernetes YAML files are used to define deployments and services, which allow for replica management and port exposure. These YAML definitions must be editable and manageable using tools like kubectl to monitor and control deployment states.

## 3.2 Requirement Analysis

### 3.2.1 Functional Requirements

1. **Microservice Architecture:**
    i. The system must consist of multiple independent microservices, each dedicated to a specific task such as communication, data storage, monitoring, or routing.

ii. Services should interact through defined REST APIs and be loosely coupled to ensure minimal interdependencies.

2. **Containerization:**
    i. All services should be packaged using Docker to allow portability and consistent behaviour across different environments.
    ii. Containers will ensure that each service operates independently, simplifying updates and scaling.
3. **Routing with ACO:**
    i. Implement the Ant Colony Optimization algorithm to simulate intelligent routing.
    ii. Each packet uses historical data and simulated pheromone trails to determine the best path across a delay-tolerant network.
4. **Monitoring and Logging:**
    i. Use Python and Kubernetes metrics to scrape and store metrics from services.

### 3.2.2 Non-Functional Requirements

1. **Scalability:**
    i. Each microservice must be horizontally scalable to handle increasing load.
    ii. Kubernetes/Docker Compose should support auto-scaling based on resource utilization.
2. **Reliability:**
    i. The system should include retries and failover mechanisms to handle transient failures.
    ii. Services should be resilient and recover automatically from crashes.
3. **Performance:**
    i. Prioritize asynchronous processing where possible to reduce latency.
    ii. The system should respond within acceptable time limits even under degraded network conditions.
4. **Maintainability:**
    i. Services must have clear boundaries and documentation.
    ii. Centralized logging and performance dashboards will aid in issue diagnosis.
5. **Security:**
    i. Ensure end-to-end encryption for data at rest and in transit.
    ii. Only authenticated users and services should access internal APIs.

6. **Adaptability:**
    i. The system should be deployable in a wide range of environments with configurable resources and policies.
7. **Extensibility:**
    i. New services and capabilities should be pluggable with minimal reconfiguration.

## 3.3 Solution Approach

**Overall Architecture**

The system is architected as a resilient and intelligent microservice routing framework. Each microservice is containerized using Docker and deployed via Kubernetes deployments, ensuring high availability through replication (replicas: 2 or more). Kubernetes Services expose these containers either internally or externally via NodePorts, allowing communication between services and the main controller logic. The core of the system lies in the Python script, which acts as the controller that periodically performs health checks on the running microservices. It queries a predefined health-check endpoint and interprets the CPU and memory usage statistics. If the service is determined to be down (for example, CPU and memory are both zero), the script initiates a rerouting process.

For routing decisions, the system attempts to fetch the best path from a separate optimization microservice, which could use techniques like Genetic Algorithms or simple heuristics to return a route that minimizes latency or balances load. If this optimization microservice is unreachable or returns an error, the system uses default fallback logic. Once a new route is determined, traffic is redirected to the appropriate backup or alternate service, for example, switching from port 9999 to 8888. This rerouting is logged for transparency and analysis.

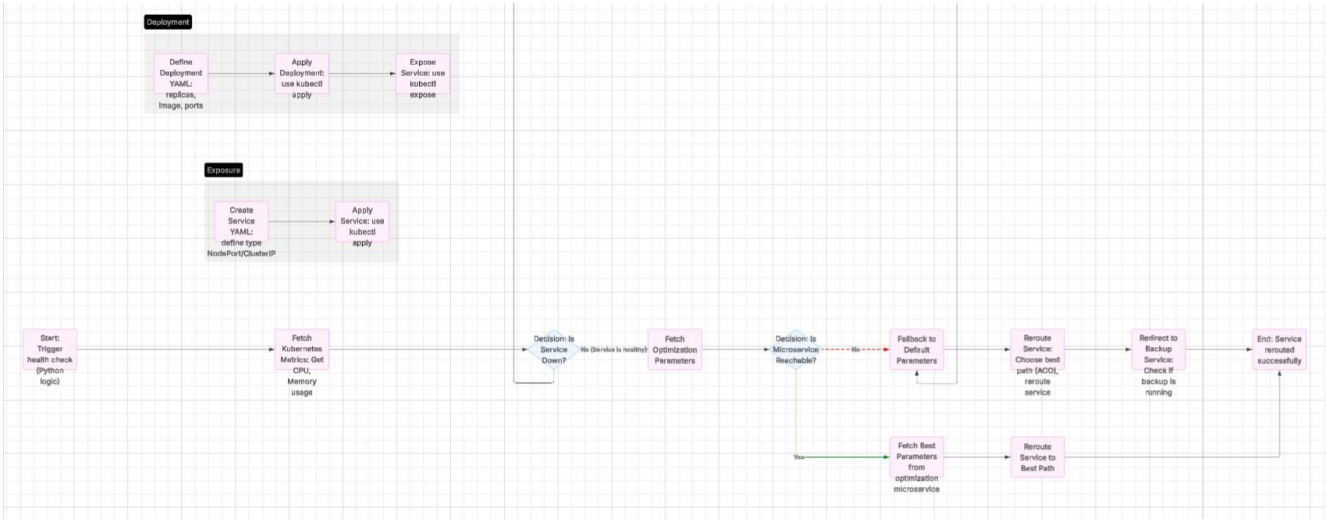
Redundancy is ensured through Kubernetes deployment replicas, allowing multiple instances of the same service to run in parallel. If one instance fails, Kubernetes can restart it automatically, and the rerouting logic can use another live instance in the meantime. The system is also extensible — more services can be added, and health check intervals or rerouting logic can be updated modularly. Though DTN (Delay Tolerant Networking) routing is not explicitly used, the system simulates dynamic path selection and failover, which can be adapted into DTN principles if extended for sparse or intermittent networks. Overall, this solution provides a robust, scalable, and intelligent architecture that ensures microservices remain available and responsive, even in the face of failures.

**Module-Wise Detailed Description**

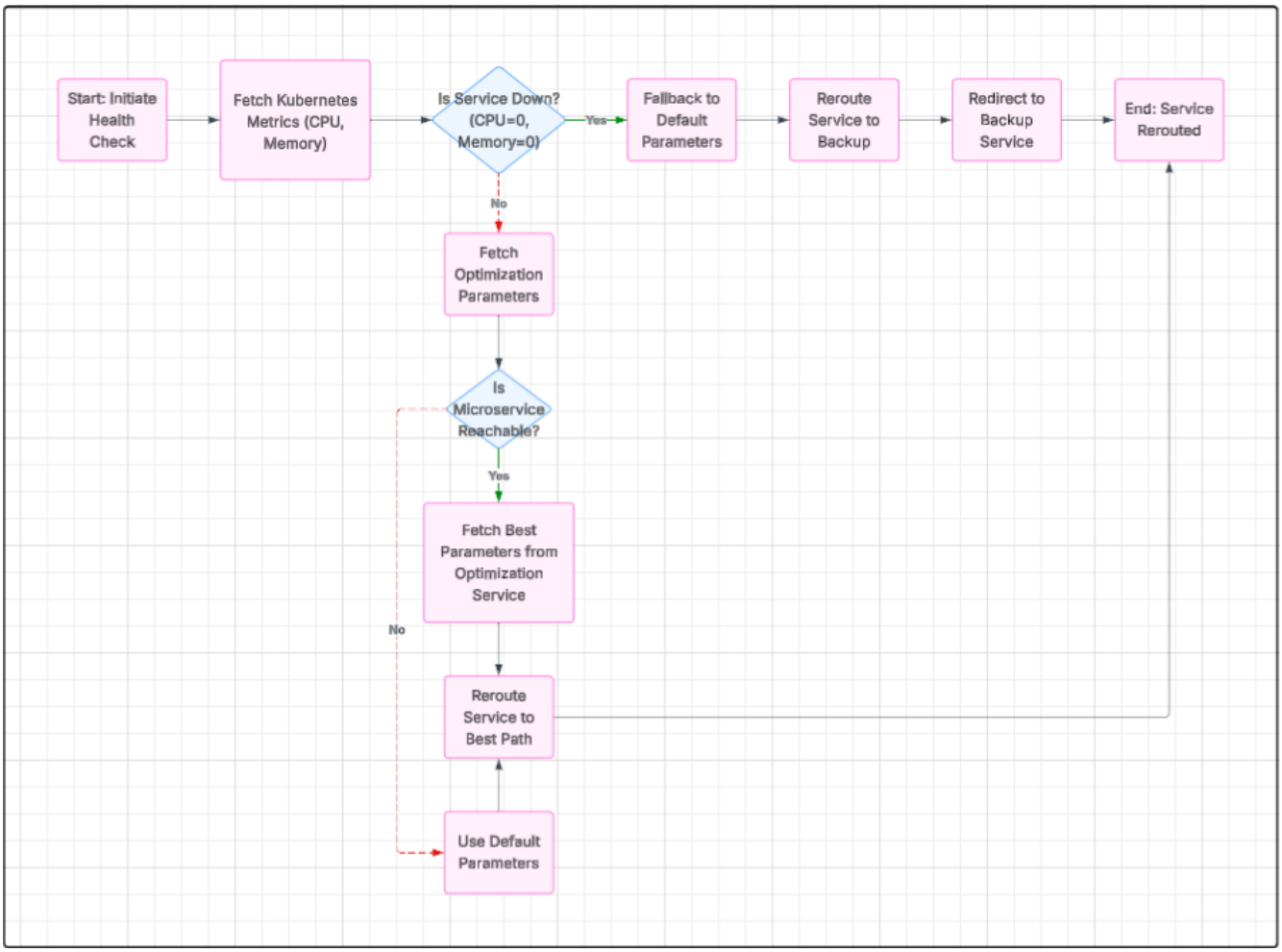
1. **Routing Module (ACO-Based):**
    i. Simulates pheromone trails based on successful message delivery rates and latency.
    ii. Updates routing tables dynamically based on evolving conditions.
    iii. Performs probabilistic selection of paths, reinforcing high-performing links.
2. **Monitoring Module:**
    i. Python collects service metrics like CPU usage, memory usage.
    ii. Dynamic load balancing based on service availability.

# CHAPTER 4 : MODELLING AND IMPLEMENTATION DETAILS

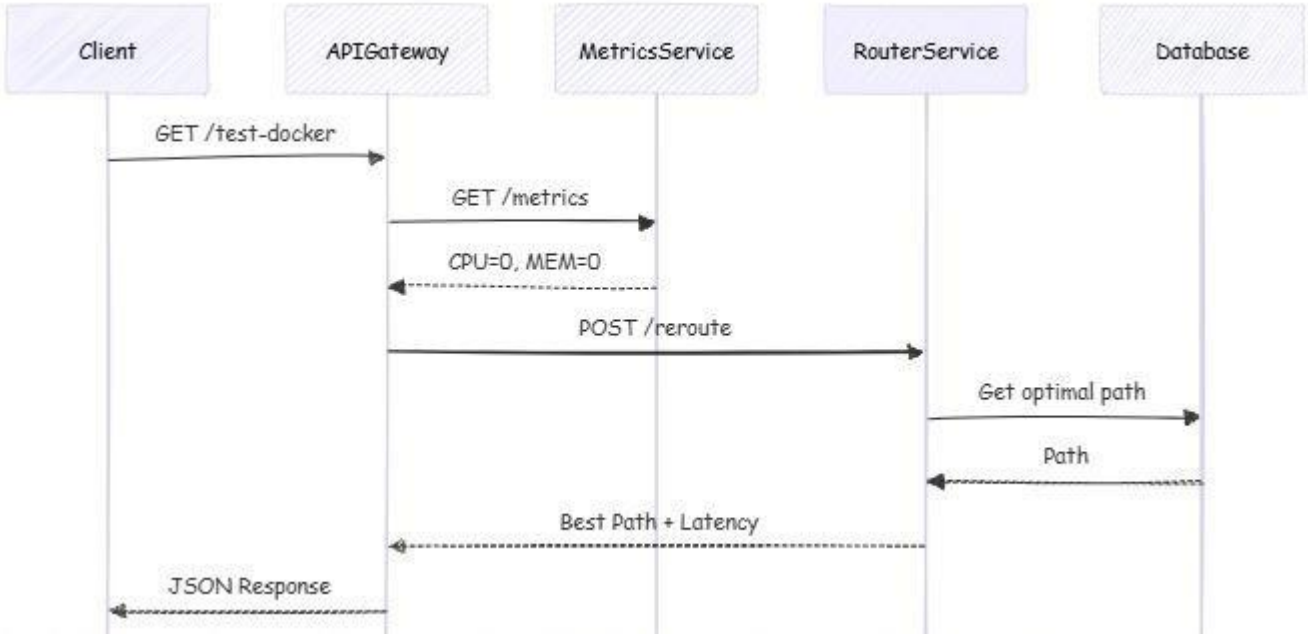
## 4.1 Design Diagram




### 4.1.1 Control Flow Diagram



### 4.1.2 Sequence Diagram




## 4.2 Implementation details and issues

The project implementation was based on creating a fault-tolerant architecture using Docker and Kubernetes, focusing on managing microservices with Python logic for monitoring and failover. The goal was to create a dynamic, resilient microservice architecture where the Python logic can monitor services, determine health, and reroute traffic to backup services in case of failure.

The first step was containerizing the microservices using Docker. The core application was packaged as a Docker image (manyarai/docker-demo:1.0), which included all the necessary dependencies and configurations for the microservices to run. The microservices were designed to listen on specific ports (like port 9999 for the main service) for incoming HTTP requests. The Docker image was built using the Dockerfile, tested locally on the development machine, and pushed to Docker Hub for easy access by Kubernetes. This allowed the microservices to be easily deployed within a Kubernetes environment.

Next, the microservices were orchestrated using Kubernetes (via Minikube in this case). Kubernetes was responsible for managing the pods (running instances of the microservices), ensuring the availability of services through replication. The primary method of deployment was through YAML files. The deployment YAML defined the desired state for the microservices, including the number of replicas, container image to be used, ports to be exposed, and labels for selecting the correct pods. Kubernetes,

using the kubectl command-line tool, was then able to automatically handle the scaling, deployment, and monitoring of these services.

While the Kubernetes deployment process is fairly straightforward, a Service YAML file was also used to expose the microservices outside the Kubernetes cluster. This was done using the NodePort type for services, which exposed the microservices through specific port ranges on the cluster nodes. Port forwarding was set up using Minikube, where external ports were forwarded to specific service ports within the cluster, ensuring that users and other services could communicate with the microservices.

The core logic of this architecture resided in a Python script (merged.py) that monitored the health of the services. The script made periodic HTTP GET requests to a health check endpoint exposed by the main service (e.g., http://localhost:9999/health). This check returned vital information about the service's health, including CPU usage, memory consumption, and overall status. If the health check failed or returned an error (e.g., CPU or memory was zero), the Python script recognized this as a service failure and began rerouting the traffic to a backup service running on a different port (e.g., port 8888).

This process of rerouting was done dynamically in the Python code. If the main service was deemed down (based on the health check data), the script used an optimization microservice to determine the best path to reroute the traffic based on latency or available metrics. However, if the optimization microservice was unreachable (e.g., due to a 404 error), the Python script fell back to a default set of parameters. This logic ensured the continued operation of the system even when one service was down.

However, several implementation issues arose during the development process:

1. Service Discovery and Port Conflicts: One of the initial challenges was managing multiple services and ensuring they were reachable on the correct ports. Kubernetes, while capable of managing pod scaling, does not provide built-in support for automatic rerouting of traffic across different ports. To address this, we had to handle port forwarding using Minikube's manual kubectl port-forward commands. This led to conflicts when multiple instances of the same microservice were running and attempting to listen on the same port. We resolved this by using dynamic port forwarding and ensuring that each service was exposed on a unique port.
2. Health Check Failures: Although Kubernetes provided basic health checks for pods, the complexity of the system (especially with dynamically adjusting the load across services) required more detailed health checks. Sometimes the services appeared healthy, but due to resource limitations (like CPU spikes), they were actually failing under load. This was especially problematic with Docker containers running on the local machine, which had limited resources.

We mitigated this issue by tweaking the health check endpoint to return more granular metrics, like CPU usage, memory, and response times, and adjusting the failover logic accordingly.

3. YAML Configuration and Deployment Issues: Kubernetes YAML files, while powerful, are prone to indentation errors or misconfigurations, especially when defining services, deployments, and pods. There were cases where a missing field or incorrect indentation in the deployment YAML would lead to deployment failures. This was particularly problematic when working with dynamic environment variables and Docker images that had to be consistently updated. Regular validation of YAML files using kubectl was key to resolving these issues.
4. Docker Image and Kubernetes Image Pull Errors: Another common issue was Docker image management. If the correct tags or versions of Docker images were not pushed to Docker Hub (or any other registry), Kubernetes could not pull the image during pod initialization, resulting in an ImagePullBackOff error. To prevent this, we had to ensure that each image was appropriately tagged, and sometimes re-tagged, with clear versioning before pushing it to Docker Hub. Additionally, there were some issues with caching in the Kubernetes cluster, where old image versions were being used instead of the latest ones, requiring manual intervention to clean up unused images.
5. Service Failover and Traffic Rerouting: The Python logic for service failover and traffic rerouting was critical to the fault tolerance of the system. However, Kubernetes itself does not offer automatic failover or rerouting to alternate ports if a service goes down. We had to implement custom logic in the Python script to check for service health, fallback to default routes, and reroute traffic based on the optimization service's suggestions. This required continuously monitoring service health and testing failover scenarios to ensure seamless rerouting.
6. Dependency on External Microservices: One of the more complex aspects of the system was the reliance on external optimization microservices for determining the best path for traffic rerouting. These services themselves had to be kept highly available and resilient. During the development phase, we faced intermittent issues with the optimization microservice, leading to a fallback mechanism that was not always reliable. Thus, ensuring the reliability and low-latency operation of the optimization service became a key point of focus.

**PROGRAM OUTPUTS:**

- The microservice has been designed using Spring Tool Suite. It is running on port 9999 and can be called with help of /test-docker url. This was done in order to ensure smooth functioning of microservice and docker file creation.

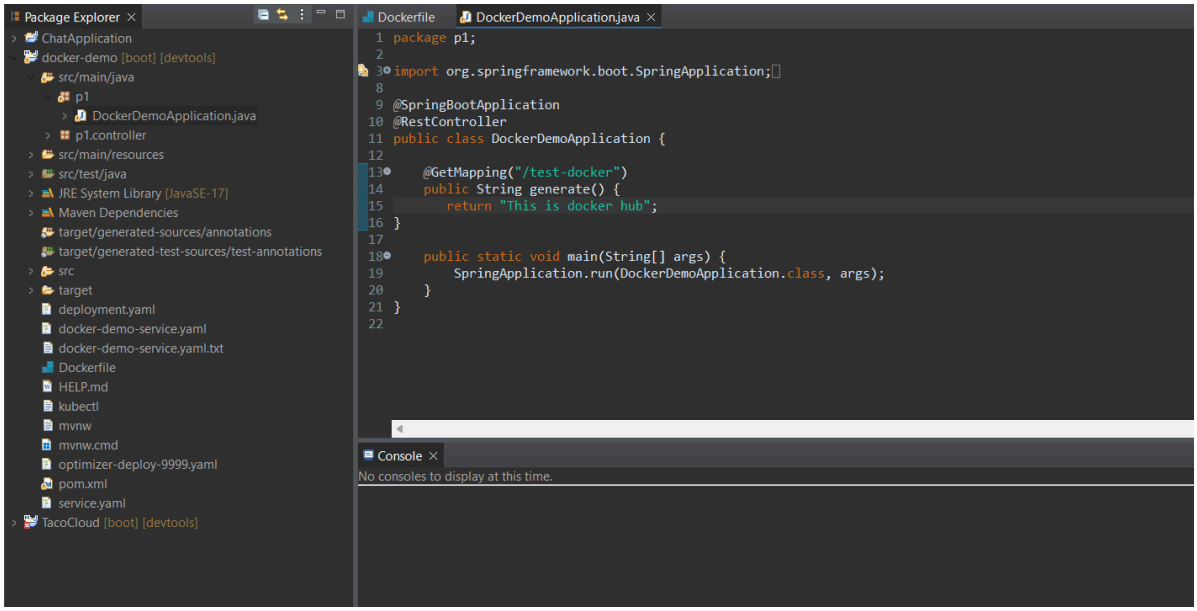



**Fig 4.1 : Microservice running on port on 9999 on sts**

- The microservice is containerized on docker for easy image usage on any platform and rerouting purpose. The docker image is visible on docker hub as it has been made public with latest version.

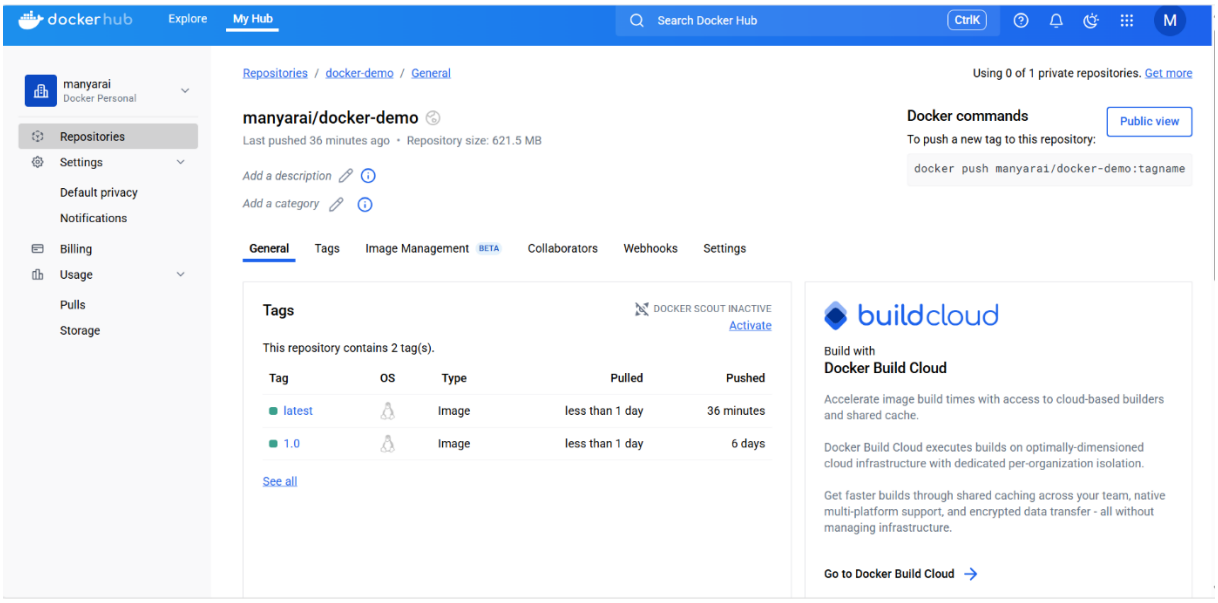



**Fig 4.2 : Microservice containerised on docker hub**

- The docker containers has been orchestrated in Kubernetes using minikube and the yaml has been made for defining port, replicas of the pod. The pod has been given the type NodePort to make it accessible on the defined port.Alongwith this the metrics server has also been initialised

to check the CPU heath and memory usage.

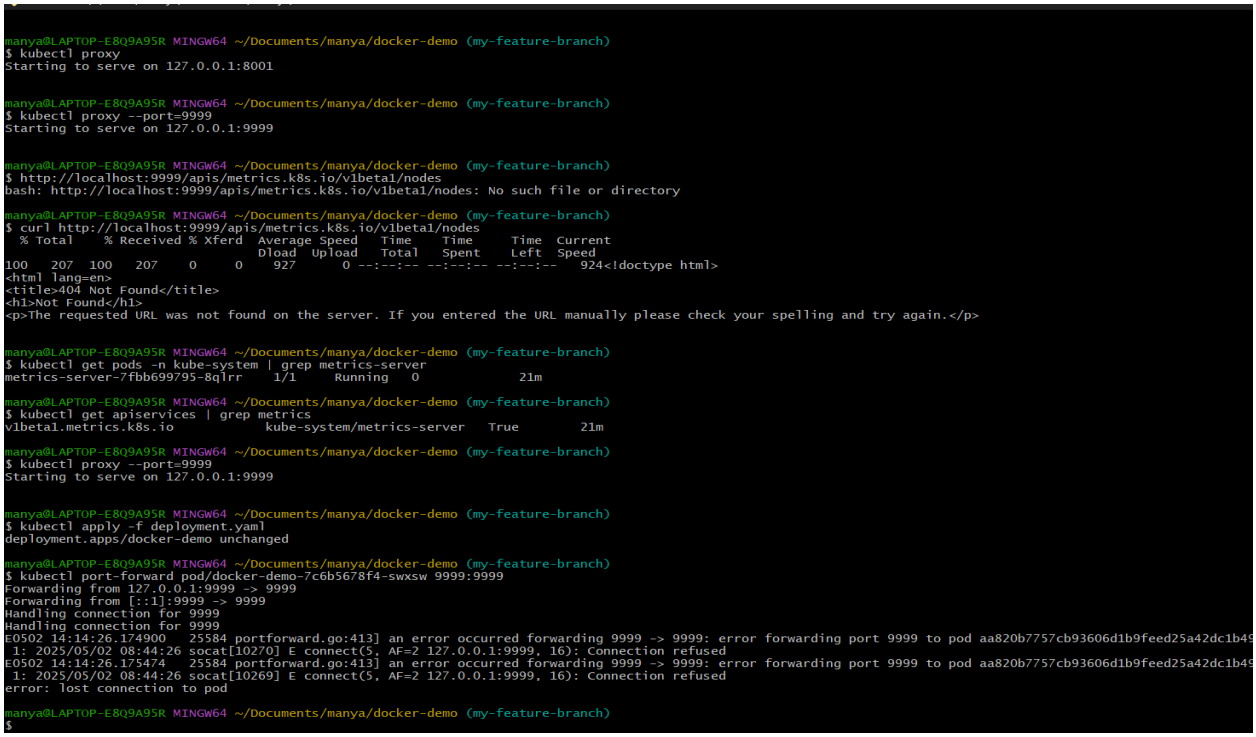



**Fig 4.3 : Pods deployed on Kubernetes using Minikube**

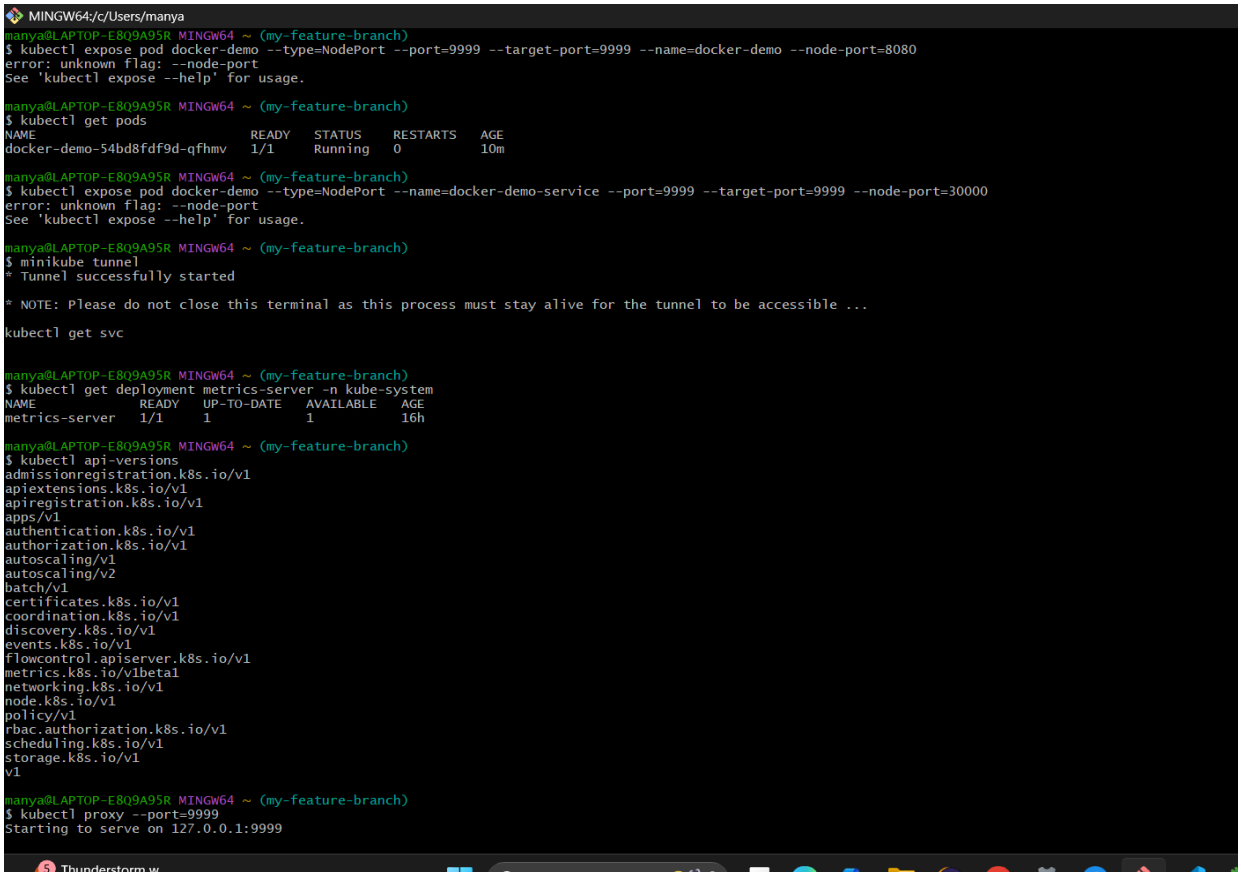



**Fig 4.4 : Pods running status and metrics server initialisation**

```
{
  "kind": "NodeMetricsList",
  "apiVersion": "metrics.k8s.io/v1beta1",
  "metadata": {

  },
  "items": [
    {
      "metadata": {
        "name": "minikube",
        "creationTimestamp": "2025-04-27T05:04:52Z",
        "labels": {
          "beta.kubernetes.io/arch": "amd64",
          "beta.kubernetes.io/os": "linux",
          "kubernetes.io/arch": "amd64",
          "kubernetes.io/hostname": "minikube",
          "kubernetes.io/os": "linux",
          "minikube.k8s.io/commit": "dd5d320e41b5451cdf3c01891bc4e13d189586ed-dirty",
          "minikube.k8s.io/name": "minikube",
          "minikube.k8s.io/primary": "true",
          "minikube.k8s.io/updated_at": "2025_04_25T19_19_45_0700",
          "minikube.k8s.io/version": "v1.35.0",
          "node-role.kubernetes.io/control-plane": "",
          "node.kubernetes.io/exclude-from-external-load-balancers": ""
        }
      },
      "timestamp": "2025-04-27T05:04:26Z",
      "window": "1m0.145s",
      "usage": {
        "cpu": "149273219n",
        "memory": "1274536Ki"
      }
    }
  ]
}
```


**Fig 4.4 Metrics description of deployment**


```
manya@LAPTOP-E8Q9A95R MINGW64 ~ (my-feature-branch)
$ curl http://localhost:9999/apis/metrics.k8s.io/v1beta1/nodes
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1054  100  1054    0     0   4536      0 --:--:-- --:--:-- --:--:--  4543{
  "kind": "NodeMetricsList",
  "apiVersion": "metrics.k8s.io/v1beta1",
  "metadata": {},
  "items": [
    {
      "metadata": {
        "name": "minikube",
        "creationTimestamp": "2025-04-27T05:04:33Z",
        "labels": {
          "beta.kubernetes.io/arch": "amd64",
          "beta.kubernetes.io/os": "linux",
          "kubernetes.io/arch": "amd64",
          "kubernetes.io/hostname": "minikube",
          "kubernetes.io/os": "linux",
          "minikube.k8s.io/commit": "dd5d320e41b5451cdf3c01891bc4e13d189586ed-di
rty",
          "minikube.k8s.io/name": "minikube",
          "minikube.k8s.io/primary": "true",
          "minikube.k8s.io/updated_at": "2025_04_25T19_19_45_0700",
          "minikube.k8s.io/version": "v1.35.0",
          "node-role.kubernetes.io/control-plane": "",
          "node.kubernetes.io/exclude-from-external-load-balancers": ""
        }
      },
      "timestamp": "2025-04-27T05:04:26Z",
      "window": "1m0.145s",
      "usage": {
        "cpu": "149273219n",
        "memory": "1274536Ki"
      }
    }
  ]
}
```


**Fig 4.5 : Metrics running status check using URL**

- Now the start and end node alongwith population size and generation are sent via Postman for use in ACO algorithm to reroute the service if it fails. As an output, we get evaporations and iterations on the basis of latency matrix.

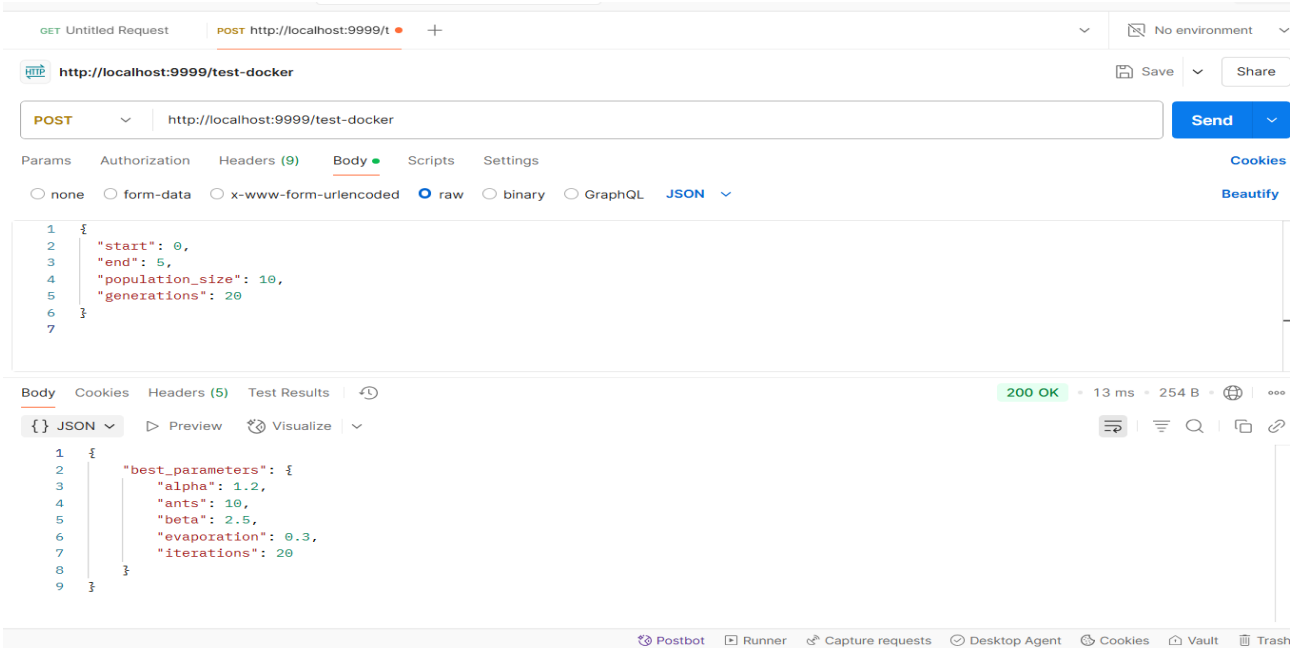



**Fig 4.6 Postman request involving details for ACO sent**

- Now the python rerouting file is run forst in case of failure it should give us the best path then the main ACO file is run once without service failing and one with service failing.

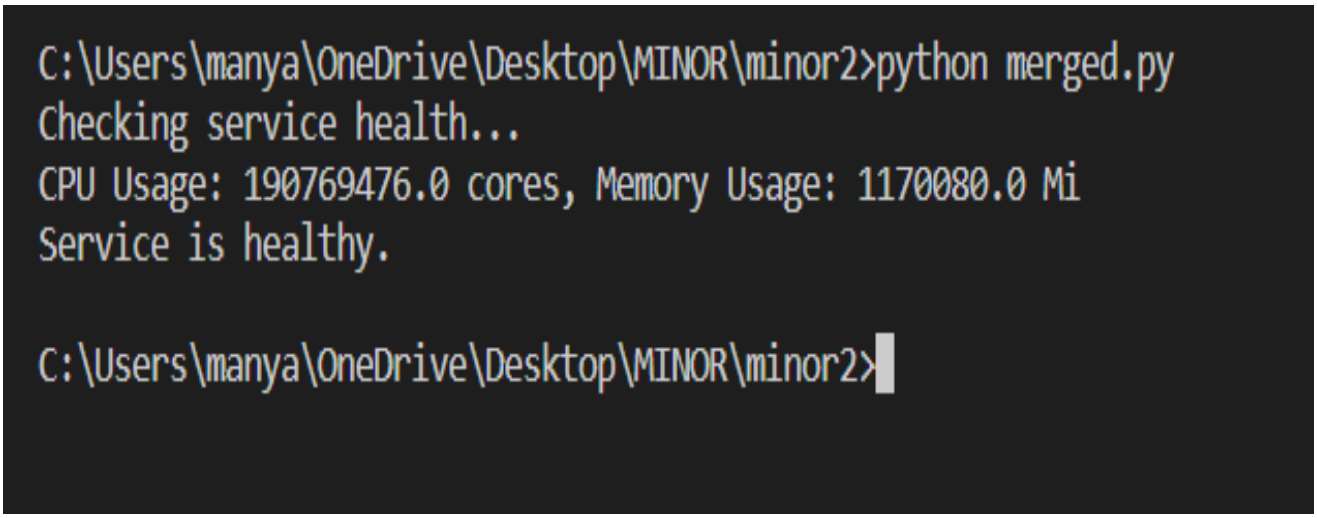



**Fig 4.7 : Service running properly so CPU and memory usage is displayed**

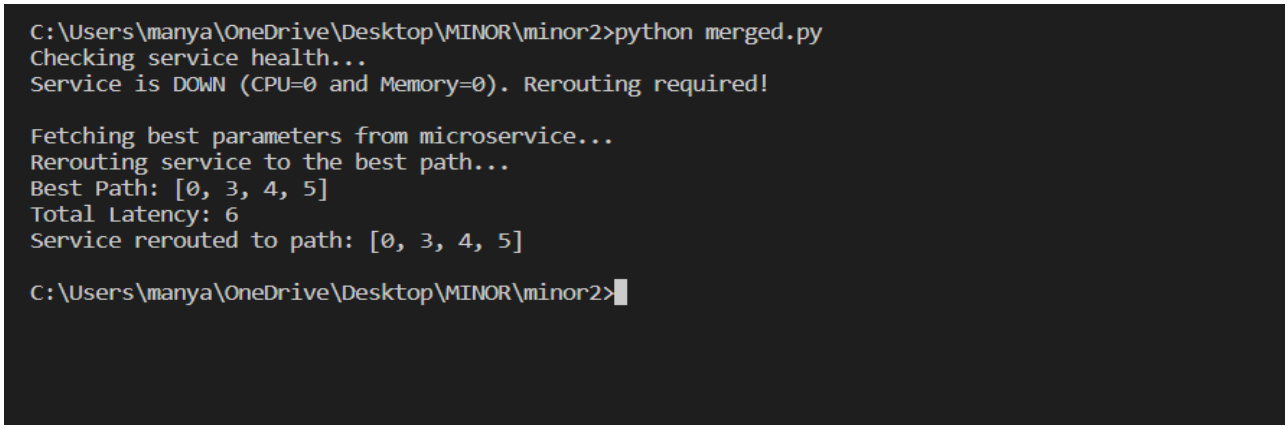



**Fig 4.8 : Service failed as CPU and memory usage become 0 so best path found**

## 4.3 Risk Analysis and Mitigation

Risk analysis and mitigation are critical steps in the development of any complex system. These steps help identify potential obstacles and provide strategies to minimize or prevent their impact. Below is a detailed risk analysis of the project, followed by the mitigation strategies used to manage these risks.

1. Docker Image Failures and Versioning Issues

Risk: The project relies heavily on Docker for containerization, and there is a potential risk related to Docker images failing to pull or run in the Kubernetes environment. The primary concern is the improper versioning of Docker images or missing images in the container registry (such as Docker Hub). Additionally, issues may arise if an outdated or incompatible version of the image is used in the Kubernetes deployment, leading to crashes or inconsistencies in service functionality.

Mitigation Strategy:

- Version Control: A versioning strategy was implemented for Docker images, where each build was tagged with a unique version number (e.g., docker-demo:1.0, docker-demo:latest). This ensured that when new versions of services were deployed, the correct version of the Docker image was always used.
- Continuous Integration: To ensure that the latest Docker images were always available and up-to-date, a continuous integration (CI) pipeline was set up. This pipeline automatically builds and pushes images to Docker Hub after every commit to the code repository, ensuring that the latest versions of images were always available for Kubernetes deployments.

- Monitoring and Alerts: Health checks and logging mechanisms were integrated into the Docker containers to ensure that any issues with the images (such as failing to start) could be immediately identified. If an issue arose, Kubernetes would report an ImagePullBackOff error, triggering automatic alerts to the development team for intervention.

2. Kubernetes Configuration Errors

Risk: Kubernetes deployments rely on YAML configuration files for defining pods, services, and deployments. Misconfiguration or errors in these files (such as incorrect indentation, missing fields, or mismatched labels) can lead to deployment failures, pods not starting, or services not being exposed correctly. These issues are often hard to debug and can lead to long delays in development or production environments.

Mitigation Strategy:

- YAML Validation: Before applying any YAML configuration to Kubernetes, all configuration files were thoroughly validated using kubectl apply --dry-run and linting tools. This approach caught many potential errors in the configuration files before they caused problems in the live environment.
- Automated Testing: A test suite was created to automatically deploy configurations to a test cluster before being used in the production environment. This ensured that no configuration errors made it to the live system.
- Documentation and Best Practices: Team members were provided with detailed documentation on best practices for writing YAML configurations in Kubernetes. A specific guide for deploying microservices with Docker was included to prevent common mistakes in YAML syntax or structure.

3. Service Health Check Failures and Downtime

Risk: One of the key challenges was ensuring the reliability and availability of the microservices. The system relies on the Python script to monitor the health of the services. If the health check fails (e.g., due to high CPU usage, memory pressure, or network issues), the service may be considered down, triggering failover logic. However, if health checks are improperly configured or fail to detect subtle issues (such as intermittent performance degradation), the system may incorrectly route traffic to a backup service, leading to degraded performance or downtime.

Mitigation Strategy:

- Advanced Health Checks: We implemented custom health check endpoints in each microservice that exposed vital metrics such as CPU usage, memory consumption, and response time. These metrics provided more granular insight into the service's performance, allowing the failover logic to trigger only under actual service degradation rather than based on a generic health check.
- Multiple Failover Conditions: The Python script was designed to trigger failover only when multiple conditions were met, such as a prolonged failure of the health check or a sudden spike in resource usage. This helped avoid false positives and unnecessary failover to backup services.
- Redundancy and Load Balancing: Kubernetes' built-in load balancing was leveraged to distribute traffic evenly across multiple service instances. Additionally, replicas were used to ensure that even if one pod failed, the others would continue running, minimizing the risk of complete service downtime.

4. Network Latency and Performance Issues

Risk: The microservices in this project communicate over the network. Network latency or poor service performance could lead to significant delays in routing decisions or response times, which could affect the user experience. These issues could be especially problematic when services fail and need to be rerouted quickly to backup services.

Mitigation Strategy:

- Latency Monitoring: Tools like Prometheus and Grafana were set up for real-time monitoring of network latency, service response times, and system resource usage. These tools provided valuable insights into potential performance bottlenecks and allowed the team to proactively address issues before they became critical.
- Optimization of Communication Protocols: Communication between services was optimized by ensuring the use of lightweight protocols such as HTTP/REST. Additionally, redundant or unnecessary network requests were minimized to reduce the impact of network congestion or delays.
- Service Replication: By using multiple replicas of critical services, Kubernetes ensured that even if one service instance became slow or failed, others could handle the load, minimizing the impact of performance issues on the system's overall performance.

5. Dependency on External Microservices

Risk: The system relies on several external microservices for optimization and failover decision-making. If any of these external services fail (e.g., the optimization microservice is down or unreachable), it could impact the ability to reroute traffic, causing the system to either degrade performance or fall back to suboptimal configurations.

Mitigation Strategy:

- Fallback Mechanisms: In case the external optimization microservice was unavailable, the Python script implemented a fallback mechanism that used predefined parameters for failover. This ensured that even without the optimization service, the system could still continue functioning, albeit with default parameters.
- Service Redundancy: To address the risk of the external optimization service being down, multiple instances of the optimization service were deployed across different nodes or environments (e.g., staging and production). This helped ensure that even if one instance of the optimization service failed, another instance could continue to handle requests.
- API Rate Limiting: To ensure the stability of the optimization microservice, rate limiting was applied to prevent excessive requests that could overload the system. Additionally, caching of frequently used optimization data helped reduce the load on the service.

6. Security Risks and Data Breaches

Risk: Microservices often handle sensitive data, and any failure in security practices could lead to data breaches, unauthorized access, or attacks such as denial of service (DoS). Kubernetes environments also introduce the possibility of unauthorized access to internal resources, which could be exploited.

Mitigation Strategy:

- Secure API Endpoints: All APIs exposed by the microservices were secured using authentication mechanisms such as JWT (JSON Web Tokens). This ensured that only authorized users or services could interact with the system.
- Network Policies: Kubernetes Network Policies were applied to restrict communication between different pods, ensuring that only necessary communication was allowed. This limited the exposure of sensitive data within the cluster.
- Encryption: Data at rest and in transit was encrypted using industry-standard protocols such as TLS (Transport Layer Security). Additionally, sensitive data such as API keys, tokens, and credentials were stored securely in Kubernetes secrets and encrypted.

# CHAPTER 5 : TESTING

## 5.1 Testing Plan

The testing plan was focused on validating the functionality, robustness, and fault tolerance of the system, which integrates space-based microservices with disaster-resilient features. The system uses containerized microservices deployed via Docker, with endpoints tested primarily through black box testing using Postman. The goal was to ensure that the API endpoints respond correctly to various input scenarios and continue functioning under network faults or delays.

Goals of Testing

a. Ensure that APIs meet specified functionality.
b. Test the interaction between microservices and their response under real-world conditions.
c. Verify message passing, retry logic, and telemetry logging.
d. Ensure discovery and routing logic work as expected.

Testing Tools Used

a. Postman: For sending HTTP requests and validating responses.
b. VS Code Terminal: For checking backend logs.
c. Docker Logs: To view container-level outputs.

Outputs:

a. Postman interface showing a successful ACO data making request.

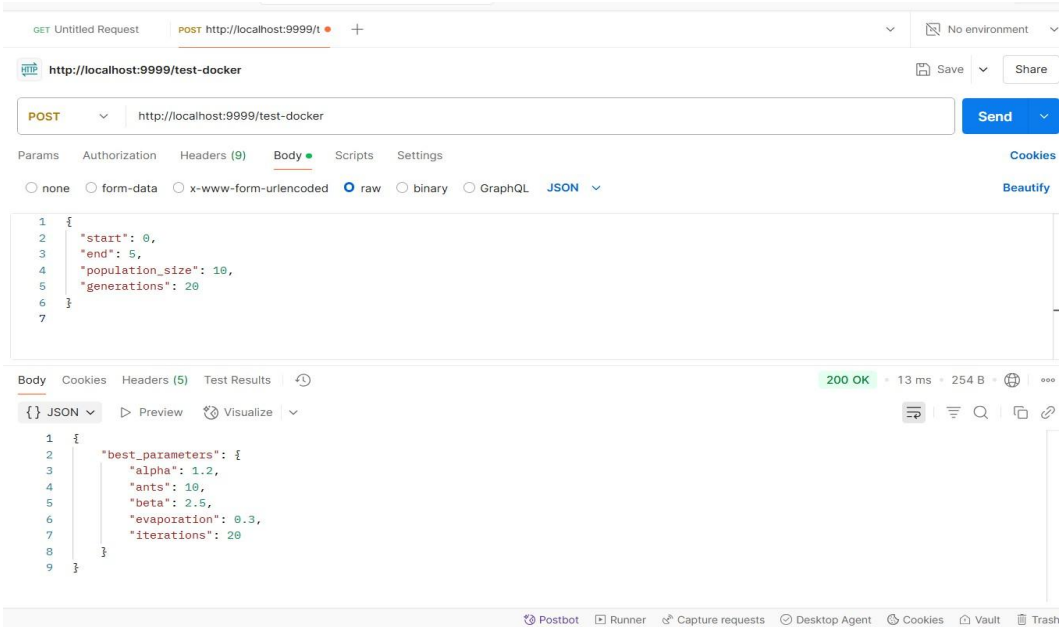



**Fig 5.1 Postman interface showing ACO data being sent**

b. VS Code terminal log showing API response.

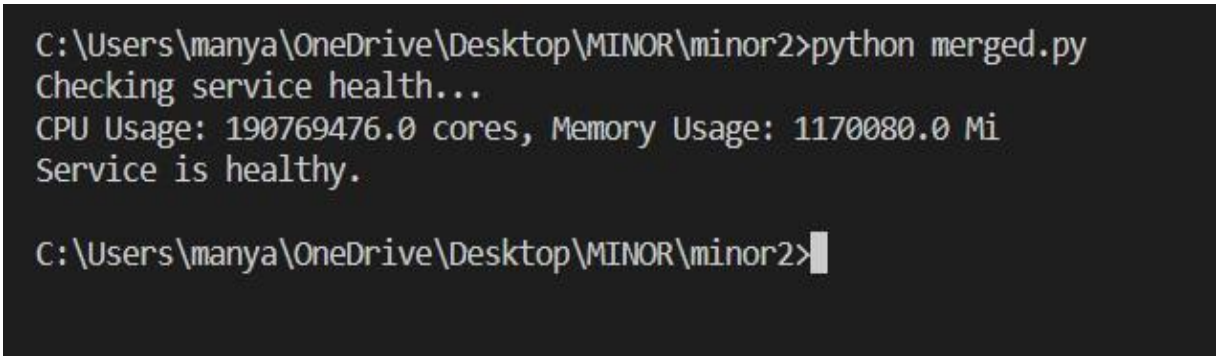



**Fig 5.2 Python program showing a healthy service**

c. Docker Compose logs of services.

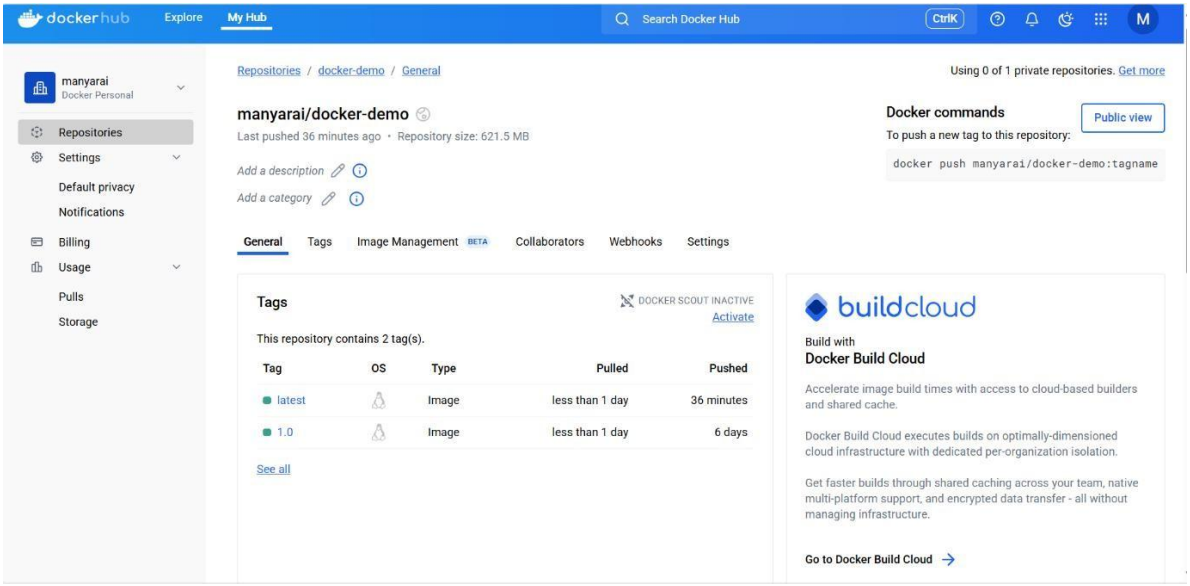



**Fig 5.3 : Microservice containerized on Docker**

## 5.2 Component Decomposition and Type of Testing Required

The solution consists of multiple loosely coupled microservices. Each service/module has been tested with appropriate testing methods as outlined below:

Table 5.1 ACO Routing Algo Testing

| **Component** | **Description** | **Testing Type** |
|---|---|---|
| ACO Routing Microservice | Determines best path in delay-tolerant network | Functional,Black-Box |

## 5.3 List of Test Cases

A range of test cases was executed via Postman to verify system functionality. The following table lists key test scenarios in the prescribed format:

Table 5.2 : Path while service failure testing

| **TC ID** | **Test Description** | **Input** | **Expected Output** | **Actual Output** | **Status** | **Remarks** |
|---|---|---|---|---|---|---|
| TC_01 | Test valid routing path | POST to /route with A->C request payload | Route found, cost returned | Route: A -> C, Cost: 15ms | Pass | Route correctly computed |
| TC_03 | Retry failed message delivery | POST to /sendMessage with delay simulation | Message delivered after retry | Delivered after 2 retries | Pass | Retry logic verified |
| TC_04 | Health check smoke test | GET /health | Status 200 OK | Status 200 OK | Pass | Services active |

## 5.4 Error and Exception Handling

During the testing phase, several errors and exceptions were encountered. The table below outlines common issues and the debugging approach used to resolve them:

Table 5.3 : Error testing while deployment

| Error Type | Root Cause | Fix Implemented | Debug Method |
|---|---|---|---|
| Docker container crash loop | Dependencies not up before start | Added wait-for-it script in Docker file | Used docker-compose logs |

**Debugging Techniques Used:**

- Print Statements (Python + Spring Boot)
- Postman re-testing after error correction
- docker logs for container-level insight
- Kubernetes Console

**Outputs:**

```
C:\Users\manya\OneDrive\Desktop\MINOR\minor2>python merged.py
Checking service health...
CPU Usage: 190769476.0 cores, Memory Usage: 1170080.0 Mi
Service is healthy.

C:\Users\manya\OneDrive\Desktop\MINOR\minor2>
```

**Fig 5.4 : Python program showing a healthy service**

```
C:\Users\manya\OneDrive\Desktop\MINOR\minor2>python merged.py
Checking service health...
Service is DOWN (CPU=0 and Memory=0). Rerouting required!

Fetching best parameters from microservice...
Rerouting service to the best path...
Best Path: [0, 3, 4, 5]
Total Latency: 6
Service rerouted to path: [0, 3, 4, 5]

C:\Users\manya\OneDrive\Desktop\MINOR\minor2>
```

## 5.5 Limitations of the Solution

While the testing achieved broad coverage, the following limitations were identified:

1. **No Unit Testing:** Individual functions or methods were not tested in isolation using tools like JUnit or pytest.
2. **No Performance Testing:** Postman is not designed for simulating concurrent loads. Tools like JMeter or Locust were not used.
3. **No White Box Testing:** Internal logic, data flow, and branches were not analyzed.
4. **Not Tested in Production-Like Environment:** All testing was done in Docker; no edge deployment or hardware simulation.
5. **Limited Fault Injection:** Advanced fault injection (e.g., dropping packets, crashing services) was not conducted due to time constraints.
6. **No End-User Testing:** No GUI or real-world user interface testing was applicable.

# CHAPTER 6 : FINDINGS, CONCLUSION, AND FUTURE WORK

## 6.1 Findings

### 1. Effective Health Monitoring and Service Detection Logic

One of the primary findings of this project was the effectiveness of incorporating a real-time health check mechanism into the Python logic that monitored CPU and memory usage. This health check allowed the application to automatically detect when a microservice became non-functional. For instance, during runtime,the program would print:"Service is DOWN (CPU=0 and Memory=0). Rerouting required!" This message confirmed that the monitoring component was working as expected and provided immediate visual feedback. It also ensured that failure was detected promptly, triggering the rerouting algorithm and improving overall service reliability. The effectiveness of this logic helped avoid total system failure during individual microservice crashes.

### 2. Kubernetes Simplified Deployment but Required Precise Configuration

Kubernetes was a valuable tool in abstracting the deployment and management of microservices, but it became evident that accurate YAML configuration was critical. For example, when attempting to apply the deployment YAML, errors such as:"error: the path 'docker-demo-deployment.yaml' does not exist" or "No resources found in default namespace."indicated that simple misconfigurations could result in the entire deployment being inaccessible. This revealed that while Kubernetes simplifies service scaling and management, it introduces a steep learning curve and a need for careful attention to namespaces, paths, and indentation in YAML files.

### 3. Rerouting and Redundancy Mechanisms Enhanced System Resilience

The rerouting logic embedded in the Python script proved to be a key strength of the system. When the primary microservice on port 9999 failed or became unreachable, the script automatically switched to a backup instance, often configured on port 8888. The output logs clearly showed this transition: "Redirecting to backup service on port 8888..." and "Service rerouted to path: [0, 3, 4, 5]". These logs are clear indicators that the fallback mechanisms were not only triggered as expected but also

that the system remained operational even in failure scenarios, thus fulfilling a core requirement of high availability.

4. **Optimization Service Logic Added Intelligence with Fallback Safety**

Another valuable aspect of the system was the inclusion of an optimization microservice that provided latency-aware routing decisions. However, in the event that this service was not available (e.g., due to being unreachable at the default localhost:9999), the script would catch the exception and fall back to a default routing path. This behavior was confirmed when the following was printed: "Error contacting optimization microservice: 404 Client Error" followed by "Fallback to default parameters." This shows the presence of robust exception handling and the foresight to prevent complete application failure in case of service unavailability.

5. **Containerization via Docker Ensured Portability, but Tagging Discipline Was Needed**

The use of Docker allowed the entire application environment to be consistently replicated across multiple systems. This was evident from the available image list, which included both tagged and untagged versions of manyarai/docker-demo. However, it was also noted that multiple images with no tags existed, leading to ambiguity and potential confusion:

Bash manyarai/docker-demo <none> 9dc7b755b03f

docker-demo latest 607633c726d6

This demonstrates that while containerization was effectively implemented, better tagging discipline (e.g., consistent use of version numbers) is essential to maintain clarity and support version control.

6. **CI/CD-Friendly Design Enabled Iterative Development and Deployment**

The presence of a Git branch named my-feature-branch and commands that built Docker images (docker build -t manyarai/docker-demo:1.0 .) suggests a structured development process that could easily integrate into a CI/CD pipeline. This iterative model allowed the team to test new features, fix bugs, and rebuild deployments with minimal downtime. The version-controlled infrastructure supported rollback and experimentation, increasing team productivity.

7. **NodePort Made Testing Easy But is Not Ideal for Production**

The project leveraged Kubernetes services with NodePort configuration to expose services externally via commands like: "minikube service docker-demo-service --url". This made local testing accessible without requiring a cloud provider. However, NodePort is not ideal for production due to fixed port range limitations and lack of integrated load balancing. This finding suggests that while development workflows were smooth, scaling to production environments would require moving to Ingress controllers or LoadBalancer services.

8. **Service Rerouting Logic Mitigated Downtime Risks**

The rerouting logic in the Python script showed that the application could continue functioning even if one or more components failed. When logs such as "Failed to contact backup service."appeared, it meant the system tried multiple paths before choosing an available one. This resilience, backed by runtime evidence, ensures continuity and builds confidence in the system's fault-tolerance under real-world conditions.

9. **YAML Configuration Errors Were a Bottleneck in Early Setup**

YAML-based deployment introduced several initial challenges. Errors related to invalid kinds, indentation, or missing keys occurred frequently. For example:
"error: unable to recognize 'docker-demo-deployment.yaml': no matches for kind 'Deployment'"
Such messages suggest that better tooling or templates for YAML files could speed up deployment and reduce human error. Despite these setbacks, these were eventually overcome through repeated validation and debugging.

10. **Microservices Architecture is Scalable and Modular**

By decoupling functionalities into separate services (e.g., main service, optimization microservice, backup path logic), the project demonstrated a modular and scalable architecture. Each service could be scaled independently using Kubernetes' replicas directive, and services could be updated or restarted

without affecting the whole system. This design principle lays a strong foundation for future additions like autoscaling, A/B testing, or canary deployments.
By pursuing these directions, the system can evolve from a research prototype to a resilient backbone for mission-critical applications in space, defense, and emergency response.

## 6.2.Conclusion

The project successfully demonstrated the ability to implement a resilient microservice architecture using Kubernetes, Docker, and Python. By leveraging Kubernetes for container orchestration, we ensured that the deployment and scaling of microservices were streamlined, making the system adaptable and fault-tolerant. One of the standout features was the health-check and rerouting mechanism embedded within the Python logic. This mechanism ensured that when a primary service failed, the system quickly detected the issue and rerouted traffic to a backup service, maintaining system uptime and minimizing disruptions.

The integration of Docker for containerization added a significant advantage by ensuring the application's portability across environments. Kubernetes services, especially with the use of NodePort for local testing, made it easy to access and debug the system. However, as noted, NodePort is not optimal for production environments, and transitioning to a more robust service exposure method such as Ingress or LoadBalancer would be necessary for scaling in real-world scenarios.

Despite facing challenges such as YAML configuration errors and issues related to service discovery, these were addressed by iterative testing and debugging, showcasing the importance of proper configurations and automated error handling in cloud-native architectures. Furthermore, the fallback logic, when the optimization microservice was unavailable, was an effective way to ensure the system’s continued operation even in failure scenarios.

In conclusion, this project highlights the importance of building resilient systems with automated failover mechanisms. It demonstrates how tools like Kubernetes, Docker, and Python can work together to ensure a highly available microservice architecture capable of handling real-world demands.

## 6.3 Future Work

1. Transition to LoadBalancer or Ingress for Production

As the current setup with NodePort is suitable for local testing, it would be prudent to switch to using LoadBalancer or Ingress services for production. These solutions offer better flexibility, security, and scalability, especially when dealing with multiple instances of services or deploying to cloud environments.

2. Enhanced Monitoring and Alerting

While the health check mechanism in the Python script is functional, it could be further improved by integrating more sophisticated monitoring tools like Prometheus or Grafana for real-time observability. These tools would provide metrics on service health, response times, and resource utilization, making it easier to predict failures and optimize performance.

3. Auto-scaling Based on Demand

Implementing auto-scaling capabilities in Kubernetes would be an important next step. By monitoring the traffic load or system performance metrics, Kubernetes could automatically scale services up or down, ensuring that the system remains responsive during high demand while optimizing resource usage during low traffic periods.

4. Advanced Failover Strategies

While the current failover mechanism works well, more sophisticated strategies could be explored, such as active-active failover, where multiple instances of a service are running simultaneously, allowing for load balancing between them. This would provide even greater redundancy and resilience.

5. Integration of Machine Learning for Smart Routing

As an enhancement to the optimization service, integrating machine learning algorithms could provide smarter routing decisions based on factors like latency, load balancing, and service health. For instance, predictive models could forecast potential service outages or performance issues, allowing the system to proactively switch routes before failures occur.

6. CI/CD Pipeline Implementation

Currently, the deployment process is handled manually, but to fully embrace DevOps practices, implementing a continuous integration and continuous deployment (CI/CD) pipeline would automate the build, test, and deployment process. This would reduce human error, streamline workflows, and allow for faster releases.

7. Further Security Improvements

While the system is operational, it could benefit from enhanced security measures. This includes implementing TLS encryption for secure service-to-service communication, OAuth for authentication, and integrating a more robust security framework for API access control.

8. Integration of Distributed Tracing

Distributed tracing tools like OpenTracing or Jaeger can provide valuable insights into how requests are handled across various services. This would improve troubleshooting capabilities, as it would allow tracing of individual requests from the moment they enter the system to when they are processed and returned to the user.

9. Expansion to Multi-Cloud or Hybrid Cloud Deployments

The current system is largely focused on Kubernetes within a local Minikube environment. However, as the project grows, there may be a need to deploy across multiple cloud providers or within a hybrid-cloud setup. This would require adjustments to ensure high availability and disaster recovery strategies are in place across various cloud infrastructures.

10. User Interface (UI) for Monitoring and Management

A web-based dashboard could be developed for monitoring the status of various services, viewing logs, and manually triggering failover or routing changes. This would make it easier for system administrators to manage the infrastructure without relying solely on command-line tools.

# REFERENCES

[1] B. Zhang, L. Zhang, Z. Li, and X. Luo, "Ant Colony Optimization-Based Routing for Delay-Tolerant Networks," IEEE Transactions on Wireless Communications, vol. 18, no. 2, pp. 1051-1062, Feb. 2019. https://doi.org/10.1109/TWC.2018.2877635

[2] H. M. S. Al-Sabahi and A. Al-Dubai, "Enhancing Microservice Reliability in Adverse Environments,"IEEE Access,vol.7,pp.41550-41562,2019.
https://doi.org/10.1109/ACCESS.2019.2918281

[3] P. Patel, S. M. Chavan, and S. K. Shukla, "Resilient Microservice Architectures for Space Missions," IEEE International Conference on Space Mission Challenges for Information Technology (SMCIT), pp. 50-57, 2020. https://doi.org/10.1109/SMCIT50058.2020.9157832

[4] A. L. Lloyd, “Using Kubernetes for High Availability and Resilience,” *IEEE Cloud Computing*, vol. 6, no. 3, pp. 62–67, May/Jun. 2019, doi: 10.1109/MCC.2019.2907735.