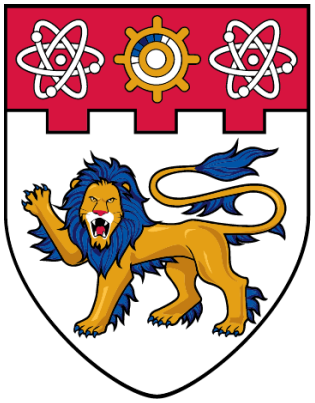

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

NANYANG TECHNOLOGICAL UNIVERSITY

ENHANCING SPEECH RECOGNITION SCALABILITY AND RESILIENCE THROUGH DECOUPLED ARCHITECTURE

Tey Li Zhang Edmund

College of Computing and Data Science

2025

NANYANG TECHNOLOGICAL UNIVERSITY

CCDS24-0015

ENHANCING SPEECH RECOGNITION SCALABILITY AND RESILIENCE THROUGH DECOUPLED ARCHITECTURE

Submitted in Partial Fulfilment of the Requirements
for the Degree of Bachelor of Computing in Computer Science
of the Nanyang Technological University

by

Tey Li Zhang Edmund

College of Computing and Data Science
2025

Abstract

To be done.

Acknowledgments

To be done.

Contents

List of Figures

List of Code Snippets

Chapter 1

Introduction

1.1 Background

Automatic Speech Recognition (ASR) systems, which convert human speech into text, have become integral to modern voice-driven technologies. While current commercial ASR solutions excel in single-language environments, they often struggle with multilingual scenarios, due to inter- and intra-sentence language variety [1]. The research team at Nanyang Technological University (NTU) Speech Lab addresses this challenge through their innovative multilingual ASR model, capable of transcribing speech in English, Malay, Mandarin, and Singlish [2, 3]. This development is particularly significant in Singapore's context, where code-switching between languages and dialects is commonplace in daily communication.

One prominent user of this ASR system is the Singapore Civil Defence Force (SCDF), which leverages the live transcription service for their emergency call centers [4]. The transcription system enables officers to record key information efficiently, saving critical time during emergencies. Given the life-saving nature of these operations, the availability and reliability of the ASR system are paramount. Any system failure or disruption could severely hinder communication and delay emergency response efforts.

Currently, the ASR system is deployed on Kubernetes across Microsoft Azure and Amazon Web Services (AWS) cloud platforms. Figure 1.1 illustrates the previous

architecture of the ASR system. In this setup, users establish a WebSocket connection to the server, referred to as the Master Pod, via the NGINX Ingress Controller. The Master Pod then forwards audio data to a Worker Pod, which processes the transcription and returns the results to the user.

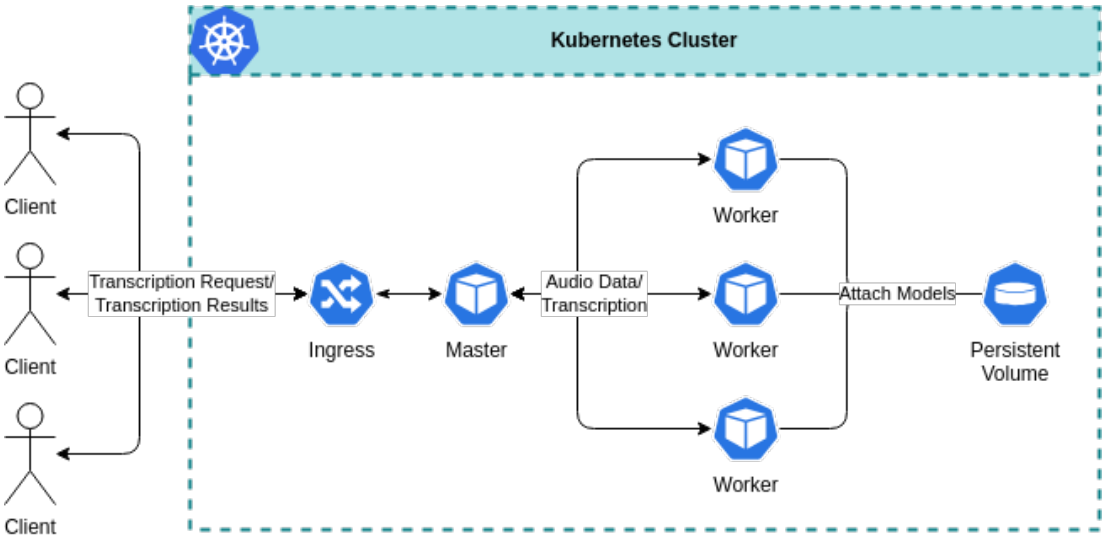

Figure 1.1: Previous Architecture of the ASR System

1.2 Challenges and Limitations

Previous Final Year Project (FYP) students have contributed to various aspects of this ASR system, such as deploying it on AWS with Terraform [5, 6] and enhancing its security [7]. However, several significant limitations persists:

1. **Single Point of Failure:** The system relies on a single Master Pod, creating a single point of failure. If this pod crashes, there is no backup instance to handle requests, leading to potential service outages.

2. **Tight Coupling:** The server and worker pods are tightly integrated, communicating synchronously via WebSocket connections. This restricts scalability, as components are highly dependent on each other, making it difficult to scale individual services independently [8].

3. **Stateful Components:** Both the server and worker pods maintain state informa-

tion, such as audio data and worker statuses. This reliance on stateful components complicates fault handling and exacerbates the system's lack of fault tolerance.

4. **Worker Failures:** Worker pods can fail due to various reasons, including resource exhaustion, node crashes, or network disruptions. These failures leave audio processing tasks incomplete, disrupting the service. More importantly, worker failures directly impact the system's ability to meet its Service Level Objectives (SLOs), particularly in terms of latency and availability.

1.3 Project Scope

The objective of this FYP is to enhance the scalability and availability of the ASR system by transitioning to a decoupled architecture. The previous tightly coupled system struggles to handle fluctuating workloads and maintain service continuity during failures. To address these challenges, this project focuses on redesigning system components, improving scalability mechanisms, and enhancing failure recovery strategies.

1.3.1 In Scope

The project will address the following key areas:

1. **Decoupling System Components with a Message Queue:** Introduce RabbitMQ as a message queue to facilitate asynchronous communication between the server and workers, enabling independent scaling and fault tolerance.

2. **Developing a Dynamic and Predictive Scaling Policy for Workers:** Implement a dynamic scaling policy for worker pods that adjusts the number of instances based on real-time system load, ensuring optimal resource utilization and responsiveness.

3. **Designing Mechanisms to Minimize Latency During Worker Failures:** Develop fault detection and recovery mechanisms to quickly identify and recover from worker failures, minimizing service disruptions and maintaining low latency.

1.3.2 Out-of-Scope

While this project focuses on improving the system's architecture and scalability, the following areas are beyond its scope:

- **Modifications to the ASR Model:** The underlying speech recognition model used for transcription will remain unchanged. Enhancements to its accuracy, multilingual capabilities, or computational efficiency are beyond the project's focus.
- **Security Enhancements:** Although security is a critical aspect of any system, improvements such as encryption mechanisms, authentication, and access controls will not be covered in this research.
- **Monitoring and Observability:** While system performance will be evaluated, the project does not aim to build comprehensive logging, monitoring, or alerting frameworks beyond what is required for testing scalability and fault tolerance.

1.4 Significance and Contributions

This project significantly enhances the scalability, reliability, and fault tolerance of the ASR system, ensuring its availability under high-load conditions and unexpected failures. The improvements directly benefit NTU Speech Lab's research initiatives and support critical applications such as SCDF's emergency call centers, where real-time transcription is essential.

By addressing the limitations of the existing architecture, this project makes the following key contributions:

- **Decoupled System Architecture:** Introduced RabbitMQ as a message queue to enable asynchronous communication between system components, allowing independent scaling of workers and improving overall system robustness.
- **Dynamic and Predictive Worker Scaling:** Implemented an adaptive scaling policy for worker pods, dynamically adjusting the number of instances based on

real-time system load to optimize resource utilization and responsiveness.

- **Automatic Scaling of Master Pods:** Configured autoscaling for Master Pods and other components based on CPU and memory consumption, ensuring efficient resource allocation.
- **Enhanced Fault Recovery Mechanisms:** Developed mechanisms for rapid detection and recovery of failed worker pods, reducing transcription disruptions and maintaining low-latency processing.
- **State Management with Redis:** Externalized application state management to Redis, reducing reliance on stateful components and improving resilience against crashes.
- **Deployment on Kubernetes with AWS:** Deployed the ASR system on Kubernetes clusters hosted on AWS, leveraging infrastructure-as-code tools like Terraform to streamline setup.
- **Refactored Codebase for Maintainability:** Restructured and modularized the existing codebase to align with the new architecture, enhancing maintainability, extensibility, and developer onboarding.
- **Comprehensive System Documentation:** Provided detailed documentation on system architecture, functionality, and key components to facilitate future development, troubleshooting, and maintenance.

1.5 Report Organisation

This report is structured into five chapters, each focusing on a specific aspect of the project:

- **Chapter 1: Introduction** - This chapter provides an overview of the project, including its background, importance, objectives, scope, and significance.
- **Chapter 2: Literature Review** - This chapter reviews previous FYP works on the ASR system and introduces the relevant technologies used in the project.

- **Chapter 3: Analysis and Design** - This chapter presents an analysis of the current ASR system architecture and identifies its limitations. It then describes the proposed decoupled architecture, including the use of message queues and dynamic scaling policies.
- **Chapter 4: Detailed Implementation** - This chapter provides a detailed implementation of the new architecture, the development of the dynamic scaling policy, and the mechanisms implemented to handle worker failures.
- **Chapter 5: Conclusion and Future Work** - This chapter summarizes the contributions of the project. It also outlines potential areas for future research and development to further improve the ASR system.

Chapter 2

Literature Review

The development of reliable and scalable Automatic Speech Recognition (ASR) systems requires an understanding of modern distributed systems technologies and architectural design. This literature review examines the technologies and approaches relevant to enhancing ASR system scalability and resilience. The review begins with previous work on the system, followed by an analysis of key technologies in distributed systems, containerization, message queues, cloud infrastructure, and chaos engineering.

2.1 Previous Work

Putra [7] had worked on the same ASR system and his project effectively highlights the advantages of transitioning from a tightly coupled ASR system to a decoupled microservices architecture using Apache Kafka. The use of Kubernetes and other modern cloud-native tools such as Kyverno, Falco, and Knative demonstrates a robust effort to address scalability, reliability, and security challenges in ASR systems. The project discusses the integration of Kafka as the message broker to decouple the master and worker components, ensuring fault tolerance and enabling the system to recover from worker crashes without losing data.

2.1.1 Research Gap

A significant research gap exists in the implementation's choice of message broker technology. The selection of Kafka for the ASR system raises concerns regarding its fit for scenarios requiring high data accuracy and context preservation. While Kafka's high throughput and durability are valuable for many systems, ASR workflows are fundamentally bottlenecked by the processing speed of workers, not the message broker. This mismatch between technology choice and system requirements suggests that RabbitMQ would be a more suitable alternative due to its task-oriented features and built-in support for state-dependent processing, which better aligns with the sequential nature of speech processing tasks.

Furthermore, while Putra's work [7] demonstrated promising results, a practical limitation emerged as the research team no longer has access to his project's codebase. This circumstance has created an opportunity to revisit the system's architecture with fresh perspective, particularly in the areas of component decoupling and scaling mechanisms. The current project therefore aims to not only address the technological fit of the message broker but also to establish a well-documented implementation that can be maintained and evolved by the research team.

2.2 Distributed Systems Architecture

2.2.1 Evolution of Microservices

The transition from monolithic to microservices architecture represents a fundamental shift in distributed systems design. Newman [9] defines microservices as small, autonomous services that work together, focusing on modularity and independent deployability. This architectural style has gained prominence due to its ability to support scalability, maintainability, and team autonomy [10]. In the context of ASR systems, microservices architecture enables independent scaling of components and improved fault isolation.

2.2.2 gRPC in Modern Applications

gRPC is a high-performance Remote Procedure Call (RPC) [11], which led to a significant advancement in service-to-service communication. Niswar et al. [12] conducted performance analyses showing that gRPC outperforms REST APIs and GraphQL in terms of response time for fetching both flat and nested data, as well as CPU utilisation. It utilises Protocol Buffers which provide a language-agnostic interface definition [13], allowing services written in different programming languages to communicate efficiently. This is crucial in microservices architectures where different teams might develop services using different technologies. These features make gRPC particularly suitable for ASR systems where reliable, high-performance communication between components is essential.

2.3 Containerization

2.3.1 Docker

Docker is a service that leverages on operating system level virtualisation to package software into containers [14]. Bernstein [15] explains how Docker containers package applications with their dependencies. This consistency is crucial for ASR systems, where complex model and service dependencies must be managed effectively.

2.3.2 Kubernetes

Kubernetes is a container orchestration tool used to automate the deployment and management of containers [16]. Burns et al. [17] detail its architecture and ability to manage containerized applications at scale. For ASR systems, Kubernetes provides essential features such as automatic scaling, self-healing, and rolling updates [18], which are crucial for maintaining service reliability.

2.3.3 Helm

Helm is a package manager for Kubernetes applications [19]. Helm utilises Charts, as reusable packages that contains pre-configured Kubernetes resources [20], making complex application deployments more manageable. These Charts function as templates that can be customized through value files [19], enabling environment-specific configurations while maintaining consistency in the underlying architecture.

2.4 Amazon Web Services (AWS)

AWS is a cloud service provider that offers a wide range of products and services. These services span compute, storage, networking, database, and container management, among others [21].

2.4.1 Virtual Private Cloud (VPC)

Amazon VPC forms the networking foundation for AWS resources, providing an isolated virtual network environment in the cloud [22]. It enables users to define network architecture with custom IP address ranges, subnets, and routing tables [22]. A key feature is its ability to span multiple Availability Zones (AZs) within a region, enhancing system resilience through geographical distribution [23].

2.4.2 Elastic Compute Cloud (EC2)

Amazon EC2 is a computing service that provides scalable virtual machines (instances) in the cloud [24]. It offers a wide range of instance types optimized for different use cases, from compute-intensive applications to memory-intensive workloads [25]. EC2 instances can be launched across multiple AZs for high availability. EC2 pricing models including on-demand, reserved, and spot instances to optimize costs based on workload patterns [26].

2.4.3 Identity and Access Management (IAM)

IAM provides fine-grained access control to AWS resources [27]. It implements the principle of least privilege through a comprehensive policy framework that defines who (principal) can do what (actions) on which resources under specific conditions [28]. IAM enables organizations to manage user identities, roles, and permissions centrally, ensuring secure access to cloud resources while maintaining compliance requirements.

2.4.4 Elastic Kubernetes Service (EKS)

Amazon EKS is a managed Kubernetes service that simplifies the deployment, management, and scaling of containerized applications [29]. It automatically manages the availability and scalability of the Kubernetes control plane across multiple AZs [29]. EKS integrates seamlessly with other AWS services and supports various deployment models, including hybrid architectures that span cloud and on-premises environments [30].

2.4.5 Elastic Container Registry (ECR)

Amazon ECR is a managed container registry service that simplifies the storage, management, and deployment of container images [31]. It provides encrypted image storage and integrates with AWS IAM for access control [32]. ECR features automatic image scanning for vulnerabilities [33] and lifecycle policies for image management [34], making it an essential component in container-based architectures.

2.4.6 Elastic File System (EFS)

Amazon EFS provides scalable, fully managed network file storage for use with AWS cloud services and on-premises resources [35]. Supporting the Network File System version 4 (NFSv4) protocol [36], EFS can be accessed concurrently by thousands of compute instances [37]. It automatically scales throughput when files are added or removed [37], making it ideal for applications requiring shared file access across multiple instances or containers.

2.5 Infrastructure-as-Code (IaC)

IaC is an approach to managing and provisioning computing infrastructure through configuration files, rather than through physical hardware configuration or interactive configuration tools. This method allows for the automation of infrastructure setup, ensuring consistency and reducing the risk of human error [38].

2.5.1 Terraform

Terraform is an IaC tool that allows for the declarative management of cloud resources [39]. This means that we define the desired final state of our architecture, and Terraform will apply changes only when necessary to achieve that state [40]. Unlike traditional manual deployment through cloud provider consoles (often referred to as ”ClickOps” [41]), Terraform enables organizations to define their infrastructure using declarative configuration files. This code-driven approach transforms infrastructure deployment from a manual, error-prone process into an automated, version-controlled workflow.

Terraform enables teams to consistently replicate environments across development, testing, and production stages [38], ensuring that infrastructure configurations remain identical at each phase. By maintaining infrastructure as code, teams can version control their changes, enabling peer reviews and the ability to roll back modifications if issues arise. Terraform also manages dependencies between different cloud resources automatically [42], reducing the complexity of infrastructure deployment.

2.6 In-memory Data Storage

2.6.1 Redis

Redis is a high-performance, in-memory data store commonly used as a cache and a key-value database [43]. Redis is fast, and thus well-suited for use in ASR systems, where it can store session information and temporary transcription data. This enables rapid access to frequently used data and facilitates reliable state management, ensuring smooth and responsive system performance.

2.7 Message Queues

Message queues enable asynchronous communication between services in distributed systems, providing temporary message storage and reliable delivery mechanisms [44]. This asynchronous pattern facilitates decoupling of producers and consumers [45], allowing components to scale independently and operate without direct dependencies on each other.

2.7.1 Apache Kafka

Apache Kafka is designed as a distributed log-based messaging system, and its main use case is for ingesting and streaming real-time data [46]. Kafka's architecture centers around append-only logs (topics) divided into partitions, where messages are immutably stored and accessed via offset-based positioning [47].

2.7.2 RabbitMQ

RabbitMQ implements the Advanced Message Queuing Protocol (AMQP) [48] and operates as a broker-based message queue [49]. It provides sophisticated message routing capabilities through exchanges and queues, supporting various patterns including publish-subscribe, and request-reply communications [50]. RabbitMQ offers features such as message acknowledgments, dead letter queues, and priority queuing [50], making it particularly suitable for complex message routing requirements.

2.7.3 Comparison of Kafka and RabbitMQ

While both systems are robust message brokers, their architectural differences significantly impact their suitability for ASR applications. Below, we compare Kafka and RabbitMQ across key dimensions relevant to ASR systems.

Message Ordering

Kafka's partitioned architecture, while enabling high throughput, cannot guarantee message ordering across partitions. Dobbelaere and Esmaili [51] note that Kafka's

ordering guarantees are limited to individual partitions.

RabbitMQ, through its queue-based architecture, maintains strict FIFO (First-In-First-Out) ordering within queues in the same channel [51], making it more suitable for ASR systems where speech context and sequence are crucial.

Message Delivery Guarantees

Kafka provides at-least-once delivery semantics through offset management [51], but consumers must handle offset commits carefully to avoid message reprocessing.

RabbitMQ's acknowledgment mechanism offers more flexible delivery guarantees, with built-in support for message acknowledgment [51] and automatic requeuing of unprocessed messages [52].

Processing Requirements

For ASR systems, where maintaining speech context is paramount [53], RabbitMQ's single-queue consumer model aligns better with the need for sequential processing.

The research shows that while Kafka's throughput advantage is significant for high-volume streaming [51], this benefit is less relevant for ASR workloads where processing speed is typically bounded by the speech recognition models rather than message throughput.

Message Queue Selection

Based on these considerations and supported by Dobbelaere and Esmaili's [51] findings, RabbitMQ emerges as the more appropriate choice for ASR systems due to its strong message ordering guarantees, flexible acknowledgement mechanisms, and support for sequential processing requirements. By leveraging RabbitMQ's features, ASR systems can ensure accurate transcription results and maintain the context of speech data throughout the processing pipeline.

2.8 Chaos Engineering

Chaos engineering is an approach to deliberately introducing failures to identify weaknesses and improve resilience [54]. By simulating conditions like Kubernetes pod failure, network delay, and node stress [55], chaos engineering helps us observe system behaviour under stress and improve system robustness [54]. It also enhances incident response time by enabling a better understanding of failure scenarios [54]. Some chaos engineering tools include Chaos Mesh [56] and AWS Fault Injection Simulator (FIS) [57].

Chapter 3

Analysis and Design Approach

3.1 Previous Architecture

The previous architecture of the ASR system, as illustrated in Figure 1.1, was deployed on a Kubernetes cluster hosted on AWS.

Transcription requests were routed through an NGINX Ingress Controller, which forwarded them to a Master Pod responsible for managing transcription tasks. Upon receiving a request, the Master Pod authenticated it and, if a worker Pod was available, initiated a WebSocket connection. The audio data was then transmitted to the worker Pod for transcription.

Each Worker Pod was associated with a model attached via a Persistent Volume Claim (PVC). After processing, the Worker Pod sent the transcription results back to the Master Pod, which then forwarded them to the client.

3.2 Challenges and Design Evolution

Based on the evaluation of the previous architecture, several challenges were identified. The new design addresses these by adopting a more decoupled, scalable, and fault-tolerant approach.

3.2.1 Decoupling Master and Worker Pods

The reliance on synchronous WebSocket communication between the Master and Worker Pods introduced significant challenges in fault tolerance and scaling. Any failure in either component would disrupt ongoing transcription tasks. Additionally, scaling Worker Pods dynamically was constrained direct communication.

Proposed Solution: Asynchronous Messaging

To enhance scalability and resilience, the new architecture adopts an message queue approach using RabbitMQ (Figure 3.1).

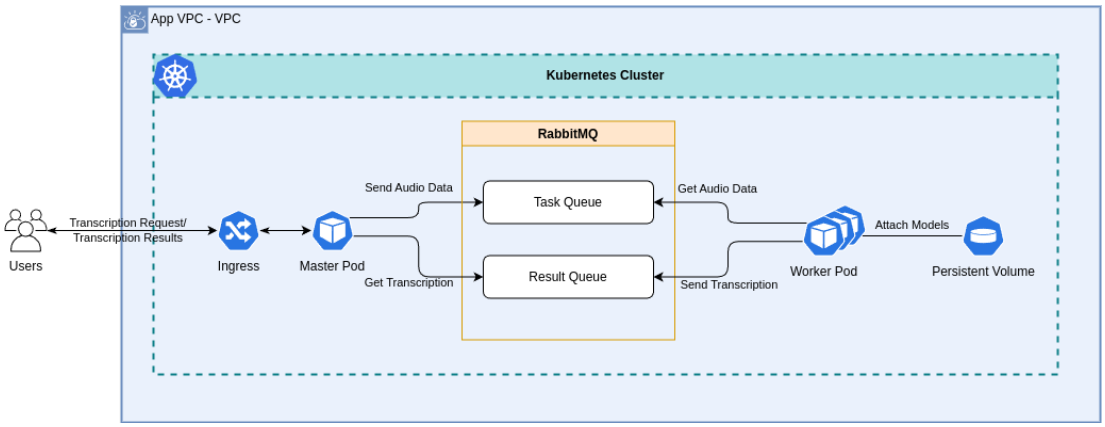

Figure 3.1: Decoupling the Master and Worker Pods

In the new design, the Master Pod publishes transcription tasks to a RabbitMQ task queue. Worker Pods asynchronously consume tasks, process the audio, and publish results to a results queue. The Master Pod retrieves transcription results and sends them back to the client.

This design eliminates direct dependencies between components, allowing independent scaling and improved fault tolerance. RabbitMQ was chosen over alternatives like Kafka due to its robust support for message acknowledgments, ensuring no transcription task is lost in case of failures.

3.2.2 Enhancing State Management and Fault Tolerance

Previously, the Worker Pods stored audio data in an in-memory queue. If a worker pod crashed, any buffered data was lost, requiring clients to retransmit. This dependency on volatile storage complicated recovery and degraded system reliability.

Proposed Solution: Redis for State Persistence

To mitigate data loss and improve fault tolerance, the application state is now stored in Redis, ensuring that if a service fails, it can recover essential state information. Additionally, RabbitMQ's message durability ensures that queued transcription tasks are not lost if a worker crashes. If a worker pod fails mid-transcription, a new worker can pick up the task from the queue without requiring client intervention. This redesign significantly enhances fault tolerance and ensures a seamless recovery mechanism.

3.2.3 Scaling and Load Management

The previous deployment relied on a single master pod, creating a single point of failure. Additionally, worker scaling required manual intervention, making it inefficient and slow in responding to traffic fluctuations.

Proposed Solution: Kubernetes Autoscaling and Worker Manager

The new architecture incorporates:

- **Kubernetes Horizontal Pod Autoscaler (HPA):** Automatically scales master pods based on request load, preventing service disruptions.
- **Dynamic Worker Scaling via Worker Manager:** A new Worker Manager service dynamically adjusts worker pod replicas based on configurable `SCALING_TARGET` (See Code 4.4).

The dynamic scaling policy is managed by an additional service called the Worker Manager, which is described in detail in *Chapter 4: Detailed Implementation*. The Worker Manager monitors the current state of worker pods for each model and adjusts the number of pods accordingly to maintain the scaling target.

Key features of this solution include:

- **Load-based scaling:** The Worker Manager scales worker pods up or down based on current traffic and processing load, ensuring optimal resource utilization.
- **Minimized scaling disruptions:** To prevent excessive scaling activity, the scaling policy incorporates a configurable `CHECK_INTERVAL` that limits the frequency of scaling operations.

3.2.4 Improving Documentation and Code Readability

The previous ASR system codebase lacked sufficient documentation, making it difficult to onboard new developers and troubleshoot issues. Without clear documentation, understanding system behavior and implementing new features was time-consuming.

Proposed Solution: Structured Documentation Strategy

To address these challenges, the new codebase will prioritize comprehensive and clear documentation. A detailed `README` file will provide an overview of the project, including its architecture, purpose, and key components. Additionally, inline comments will be incorporated throughout the codebase to explain the functionality and intent of each component.

This approach helps to onboard new developers faster, through providing them with more context to understand the system better. Additionally, it ensures maintainability by ensuring the developers can easily understand and modify the codebase in the future.

An inline comment explaining the purpose of a function or section of code can significantly improve readability. For instance, Code A.1 shows an example of an inline comment that describes the purpose of a function and its parameters.

By integrating documentation and inline comments into the development process, the new codebase will improve the maintainability and quick onboarding of new developers.

Chapter 4

Detailed Implementation

This chapter provides an in-depth explanation of the design, architecture, and deployment of the newly implemented ASR system. It elaborates on the functionality of the system components and how they interact to achieve scalability, fault tolerance, and efficient processing.

4.1 Architecture

The new architecture of the ASR system is illustrated in Figure 4.1. The system is designed with a microservices architecture, where each component is responsible for a specific task.

The system consists of the following components:

- **Client:** The frontend user interface that initiates transcription requests and receives the results.
- **NGINX Ingress Controller:** Routes incoming requests to the Live Processing Service.
- **Live Processing Service:** Manages the transcription tasks, interacts with the Worker Manager to allocate workers, and maintains a WebSocket connection with the client.

- **Worker Manager:** Allocates workers to process audio data based on the requested model and monitors the worker pods.
- **Worker:** Process the audio data using the specified ASR model and return the transcription results.
- **Message Queues:** RabbitMQ is used to decouple the communication between the Live Processing Service and the Worker.
- **Redis:** Stores the state information of the system, such as worker statuses and task details.

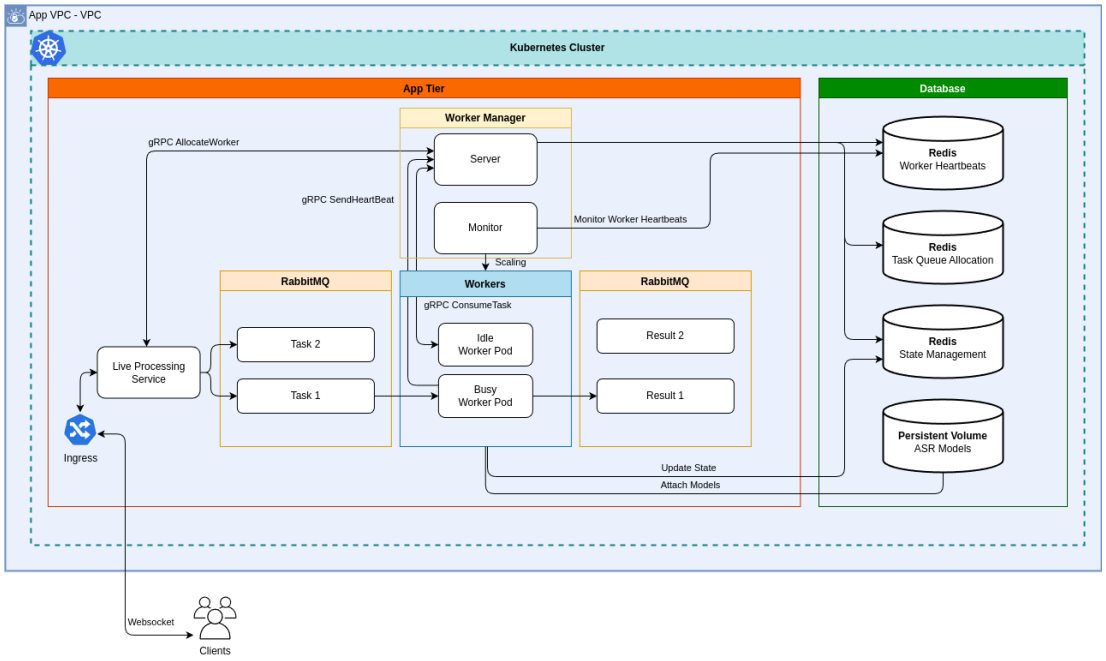

Figure 4.1: New Architecture of the ASR System

4.1.1 System Flow

The sequence diagram in Figure 4.2 provides a detailed visualization of the ASR system's flow, showing the interactions between components during the transcription process.

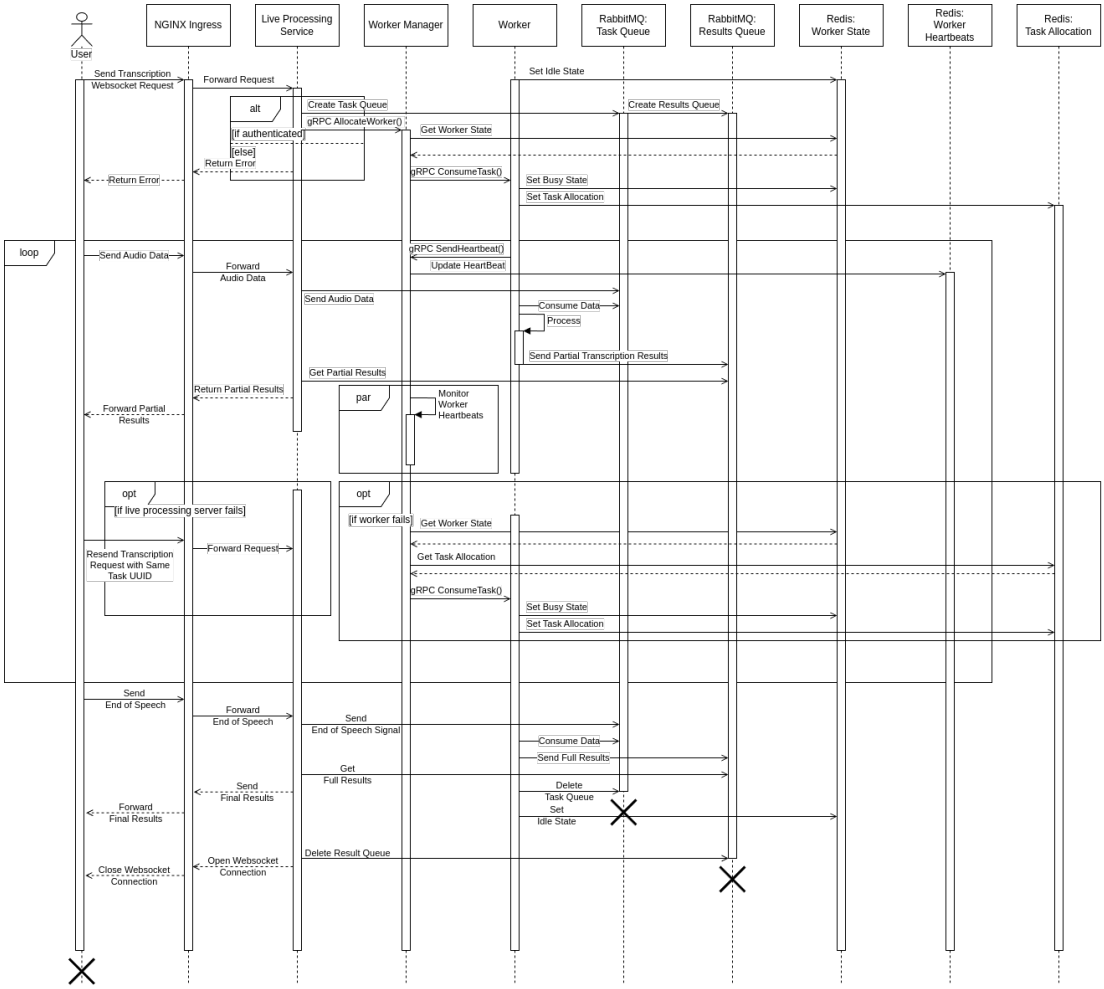

Figure 4.2: Sequence Diagram of the ASR System

Main Flow

1. The client establishes a WebSocket connection through the NGINX Ingress Controller to the Live Processing Service.
2. Upon receiving the connection, the Live Processing Service creates two queues in RabbitMQ for each task:
    - `task.<UUID of the task>`: Queue for the audio data to be processed.
    - `result.<UUID of the task>`: Queue for the transcription results of the task.
3. The Live Processing Service invokes the Worker Manager's gRPC API to allocate

a worker for the task.

4. The Live Processing Service places audio data into the task queue.
5. The Worker Manager checks the Redis database for an idle worker with the requested ASR model and assigns it to the task. The worker is notified via a gRPC call.
6. The worker updates its state in Redis to `BUSY` and begins processing the audio data.
7. Partial transcription results are sent to the result queue as the audio is processed.
8. The Live Processing Service retrieves these partial results from the result queue and streams them back to the client via the WebSocket connection.
9. Workers send periodic heartbeats to the Worker Manager to confirm their health status.
10. When the client detects the end of speech, it notifies the Live Processing Service, which marks the task as complete by sending a special End of Speech (`EOS`) message to the task queue.
11. Upon processing the `EOS` message, the worker updates its state in Redis to `IDLE` and sends the final transcription result to the result queue.
12. The Live Processing Service retrieves the final result from the result queue and delivers it to the client via WebSocket.
13. Once all results are delivered, the WebSocket connection is closed.

Worker Failure Flow

To maintain fault tolerance, the system detects and recovers from worker failures as follows:

1. The Worker Manager detects a failure when it stops receiving heartbeats from a worker.

2. It gets a new idle worker with the same ASR model from Redis and reassigns the task to the new worker.

3. The new worker retrieves the pending task from the task queue, updates its state to `BUSY`, and resumes processing the audio data.

Live Processing Service Server Failure Flow

To handle failures of the Live Processing Service:

1. The client reestablishes a WebSocket connection to the Live Processing Service and resends the transcription request using the same task UUID.

2. The Live Processing Service uses the existing task and result queues to continue processing and retrieving results, ensuring no data is lost.

4.2 Live Processing Service

The Live Processing Service manages WebSocket connections with clients to facilitate real-time transcription. Its primary functions include:

1. **WebSocket Communication:** Establishing and managing WebSocket connections to stream transcription results in real time.

2. **Task Management:** Creating and managing task and result queues in RabbitMQ for each transcription request.

3. **Worker Allocation:** Requesting worker allocation from the Worker Manager based on the required ASR model.

The Live Processing Service is built using Tornado [58], an asynchronous web framework optimized for handling long-lived connections and scaling to thousands of concurrent WebSocket sessions. This makes it well-suited for real-time applications such the ASR system.

4.2.1 WebSocket Handler

The WebSocket handler manages WebSocket connections with clients and streams transcription results back to them. It processes the following events:

- **open:** Authenticates incoming WebSocket requests, initializes task and result queues in RabbitMQ, requests worker allocation from the Worker Manager, and starts consuming messages from the result queue.
- **on_message:** Receives audio data from the client and places it into the task queue. If the message contains the final audio chunk, it deletes the associated task and result queues.
- **close:** Closes the WebSocket connection with the client and performs cleanup operations.

4.2.2 Asynchronous RabbitMQ

To maintain non-blocking performance, the Live Processing Service utilizes an asynchronous RabbitMQ client compatible with Tornado. This enables concurrent task management without disrupting the event loop.

A dedicated asynchronous RabbitMQ client with exponential backoff was implemented to handle network disruptions and RabbitMQ failures. Code 4.1 illustrates the implementation.

Code 4.1: Asynchronous RabbitMQ Client

```
async def connect(self):
    """
    Connect to RabbitMQ.
    """
    attempts = 0
    while attempts < RABBITMQ_MAX_RETRIES:
        try:
            self.logger.info(f"trying to connect to
                {self.host}:{self.port}")
```

```
            self.logger.info(f"using {self.username} as username")
            self.connection = await aio_pika.connect_robust(
                host=self.host,
                port=self.port,
                login=self.username,
                password=self.password,
            )
            self.logger.info("Connected to RabbitMQ")
            break
        except Exception as e:
            self.logger.error(f"Failed to connect to RabbitMQ: {e}")
            attempts += 1
            await asyncio.sleep(RABBITMQ_BACKOFF_FACTOR * attempts)
```

4.2.3 Worker Allocation via gRPC

The Live Processing Service requests worker allocation from the Worker Manager using gRPC. This ensures low-latency, efficient communication, which is critical for real-time transcription.

To allocate a worker for a transcription task, the Live Processing Service sends a gRPC request specifying the required ASR model. The Worker Manager responds with an available worker instance. This interaction leverages Protocol Buffers for efficient data serialization and the gRPC stub generated from the service definition.

Code 4.2 illustrates the gRPC request process.

Code 4.2: gRPC Request for Worker Allocation

```
async def allocate_worker_from_worker_manger(self, model_name,
    task_queue):
      """
      Allocate an idle worker from the WorkerManager service.

      Args:
```

```
        model_name (str): Name of the model to allocate the worker for.
        task_queue (str): Name of the task queue to assign to the
            worker.
    """
    async with grpc.aio.insecure_channel(
        f"{WORKER_MANAGER_SERVICE}:{WORKER_MANAGER_PORT}"
    ) as channel:
        stub =
            worker_manager_pb2_grpc.WorkerManagerServiceStub(channel)
        request = worker_manager_pb2.AllocateWorkerRequest(
            model_name=model_name, task_queue=task_queue
        )
        response = await stub.AllocateWorker(request)

        if response.success:
            logger.info(
                f"Worker {response.worker_name} allocated successfully:
                    {response.message}"
            )
            return response.worker_name
        else:
            logger.error(f"Failed to allocate worker:
                {response.message}")
            return None
```

4.3 Worker

The primary function of the Worker is to process audio data using the specified ASR model and return transcription results.

Each Worker loads an ASR model developed by the NTU Speech Lab, which is based on the Kaldi ASR toolkit [59]. The model is loaded into memory at startup, enabling

low-latency transcription of incoming audio data.

4.3.1 Audio Processing

Once the Worker Manager assigns an idle Worker to a task, the Worker retrieves the audio data from the task queue in RabbitMQ. It then processes the audio using the ASR model and publishes the transcription results to the result queue in RabbitMQ.

To facilitate fault tolerance, the Worker updates its state in Redis when processing a task. It marks itself as `busy` and records the assigned task. This allows the Worker Manager to track active tasks and reassign them if a Worker fails.

4.3.2 Heartbeat Monitoring

To enable health monitoring, the Worker sends periodic heartbeat signals to the Worker Manager. These heartbeats allow the Worker Manager to detect failures and reallocate tasks accordingly. Further details on this mechanism are provided in Subsection 4.4.2.

4.4 Worker Manager

The Worker Manager is responsible for managing and coordinating worker pods to ensure efficient task processing and fault tolerance. Its core responsibilities include:

1. **Worker Allocation:** Assigning idle workers to process incoming audio data.
2. **Health Monitoring:** Tracking worker health through periodic heartbeats.
3. **Task Reassignment:** Detecting worker failures and reallocating tasks to maintain continuity.
4. **Scaling Policy:** Dynamically adjusting the number of worker pods based on system load.

The Worker Manager consists of two primary components:

- **Worker Manager Server:** Exposes gRPC APIs for worker allocation and heartbeat monitoring.

- **Worker Manager Monitor:** Detects worker failures, reallocates tasks, and enforces scaling policies.

4.4.1 Worker Manager Server

The Worker Manager Server provides two main gRPC endpoints (Code A.2):

- `AllocateWorker`: Assigns an idle worker to handle a given transcription task.
- `SendHeartbeat`: Receives periodic heartbeats from workers to track their health status.

4.4.2 Worker Manager Monitor

The Worker Manager Monitor oversees worker health and task reallocation. It listens for heartbeats and updates worker status in Redis. If a worker fails to send a heartbeat within a predefined timeout, it is considered unavailable, and its task is reassigned.

Code 4.3 illustrates the fault tolerance mechanism.

Code 4.3: Worker Fault Tolerance Mechanism

```
async def monitor_heartbeats(self):
    """
    Monitor worker heartbeats and reallocate workers if necessary.
    """
    logger.info("Monitoring worker heartbeats...")
    self.redis_client = self.redis.get_client()
    while True:
        current_time = asyncio.get_event_loop().time()
        # logger.debug(f"Current time: {current_time}")
        for worker_name, value in self.redis_client.hgetall(
            "WorkerHeartbeats"
        ).items():
            lock_key = f"lock:worker_heartbeat:{worker_name}"
            try:
                # Acquire distributed lock
```

```
            if self.acquire_lock(lock_key):
                model_name, last_heartbeat = value.split(",")
                if (
                    current_time - float(last_heartbeat)
                    > WORKER_HEARTBEAT_TIMEOUT
                ):
                    logger.info(
                        f"Worker {worker_name} missed heartbeat.
                            Allocating new worker."
                    )
                    try:
                        task_queue = self.redis_client.hget(
                            "TaskAllocation",
                                f"Worker:{worker_name}"
                        )
                    except Exception as e:
                        logger.error(f"Failed to get task queue:
                            {str(e)}")
                        raise
                    self.allocate_worker(model_name, task_queue)
                    self.redis_client.hdel("WorkerHeartbeats",
                        worker_name)
                    logger.info(f"Worker {worker_name}
                        deallocated.")
            else:
                logger.warning(f"Failed to acquire lock for
                    {worker_name}")
        except Exception as e:
            logger.error(
                f"Failed to monitor heartbeat for {worker_name}:
                    {str(e)}"
            )
        finally:
```

```
                # Release distributed lock
                self.release_lock(lock_key)

            await asyncio.sleep(WORKER_MANAGER_MONITOR_INTERVAL)
```

To prevent race conditions when multiple Worker Manager pods are deployed, the Worker Manager first acquires a distributed lock before checking a worker's last heartbeat. If a worker exceeds the timeout threshold, it is removed, and a new worker is assigned to its task queue.

Testing Worker Fault Tolerance

To validate the fault tolerance mechanism, the Worker Manager was tested under failure scenarios, such as unexpected worker crashes.

To simulate a worker failure, worker pod processing the audio data can be delibrately terminated using the following command:

```
kubectl delete -n asr-sdk --now <worker-pod-name> \
  --cascade=background
```

Figure 4.3 illustrates the Worker Manager detecting that the worker pod `worker-stateful-english-0` has failed to send a heartbeat. As a result, it reallocates the task to an available idle worker, `worker-stateful-english-1`, ensuring uninterrupted processing.

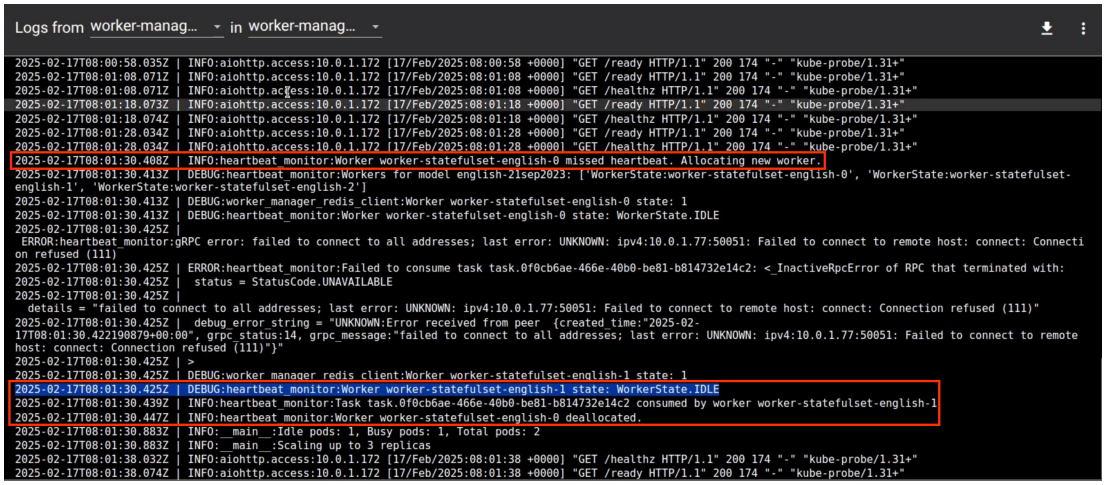

Figure 4.3: Worker Fault Tolerance Testing

Worker Scaling Policy

The Worker Manager dynamically scales worker pods based on system load. A configurable threshold, SCALING_TARGET, defines the minimum number of idle pods required at any time. The Worker Manager periodically evaluates worker utilization and adjusts the number of replicas accordingly.

Code 4.4 shows the worker scaling mechanism.

Code 4.4: Worker Scaling Policy

```
async def monitor_and_scale():
    """
    Monitor the number of busy pods for each model and scale the
        statefulset up or down based on the number of idle pods.
    """

    while True:
        for model in MODELS:
            total_pods, busy_pods = get_pod_states(model)
            idle_pods = total_pods - busy_pods
            logger.info(
```

```
            f"Idle pods: {idle_pods}, Busy pods: {busy_pods}, Total
                pods: {total_pods}"
        )

        if idle_pods < SCALING_TARGET:
            # Scale up
            new_replicas = total_pods + (SCALING_TARGET - idle_pods)
            logger.info(f"Scaling up to {new_replicas} replicas")
            scale_statefulset(new_replicas)
        elif idle_pods > SCALING_TARGET:
            # Scale down
            new_replicas = total_pods - (idle_pods - SCALING_TARGET)
            logger.info(f"Scaling down to {new_replicas} replicas")
            scale_statefulset(new_replicas)

    # Wait for the next check
    await asyncio.sleep(CHECK_INTERVAL)
```

The system checks idle pods every `CHECK_INTERVAL` seconds, ensuring that scaling adjustments do not occur too frequently. This is to allow for the previous scaling operation to taken effect before the next check, preventing excessive scaling activity.

Testing Worker Scaling

To validate the worker scaling policy, the system was tested under varying workloads to observe how the Worker Manager adapts in real time.

Figure 4.4 shows the logs of the Worker Manager, illustrating the dynamic scaling behavior of the Worker Manager. When a transcription task starts and an available worker is assigned to process it, the number of idle pods drops below the `SCALING_TARGET`, triggering an automatic scale-up. Conversely, once the task is completed and the worker becomes idle again, the Worker Manager scales down the replicas to optimize resource usage.

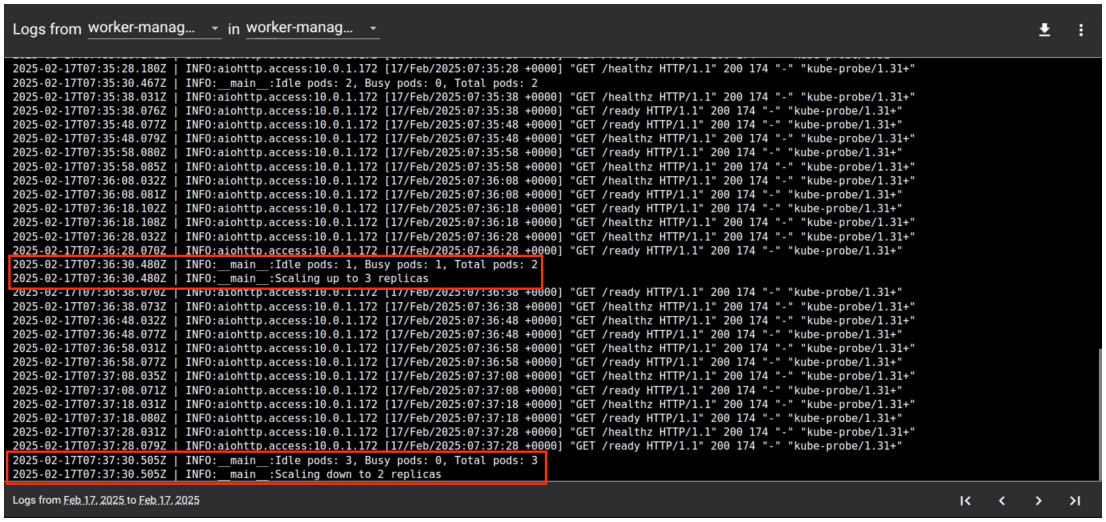

Figure 4.4: Worker Scaling Up and Down

Health and Readiness Checks

The Worker Manager Monitor performs health and readiness checks to ensure it is ready to handle requests. These checks are essential for maintaining the reliability and availability of the system.

Code 4.5 shows the implementation of the health and readiness checks.

Code 4.5: Worker Manager Monitor Health and Readiness Checks

```
async def healthz(request):
    return web.Response(text="OK")


async def ready(request):
    # Check Redis connection
    try:
        redis_client = WorkerManagerRedisClient().get_client()
        redis_client.ping()
    except Exception as e:
        logger.error(f"Readiness check failed: Redis connection error:
            {str(e)}")
        return web.Response(status=500, text="Redis connection error")
```

```
    # Check if the monitor can acquire a lock
    lock_key = "readiness_check_lock"
    if not monitor.acquire_lock(lock_key, timeout=5):
        logger.error("Readiness check failed: Unable to acquire lock")
        return web.Response(status=500, text="Unable to acquire lock")
    monitor.release_lock(lock_key)

    return web.Response(text="OK")
```

The health check (`healthz`) provides a basic indicator of whether the Worker Manager Monitor is running. It responds with an "OK" message to confirm the service is active. The readiness check (`ready`) performs the following validations to ensure the monitor is ready to function:

1. **Redis Connection:** Verifies the monitor can connect to Redis by sending a PING request.
2. **Lock Acquisition:** Ensures the monitor can acquire and release a distributed lock, which is critical for managing shared resources.

When deployed on Kubernetes, it continuously monitors the readiness and liveness endpoints to determine if the service is ready to receive traffic or if it needs to be restarted. Figure 4.5 shows the successful readiness and liveness check at every 10 seconds interval.

4.5 Infrastructure Deployment

The infrastructure is deployed using Terraform, an Infrastructure-as-Code (IaC) tool that allows for the provisioning of cloud resources in a declarative manner. Terraform compares the desired state defined in configuration files with the current state of the cloud environment and makes the necessary changes to achieve the desired state.

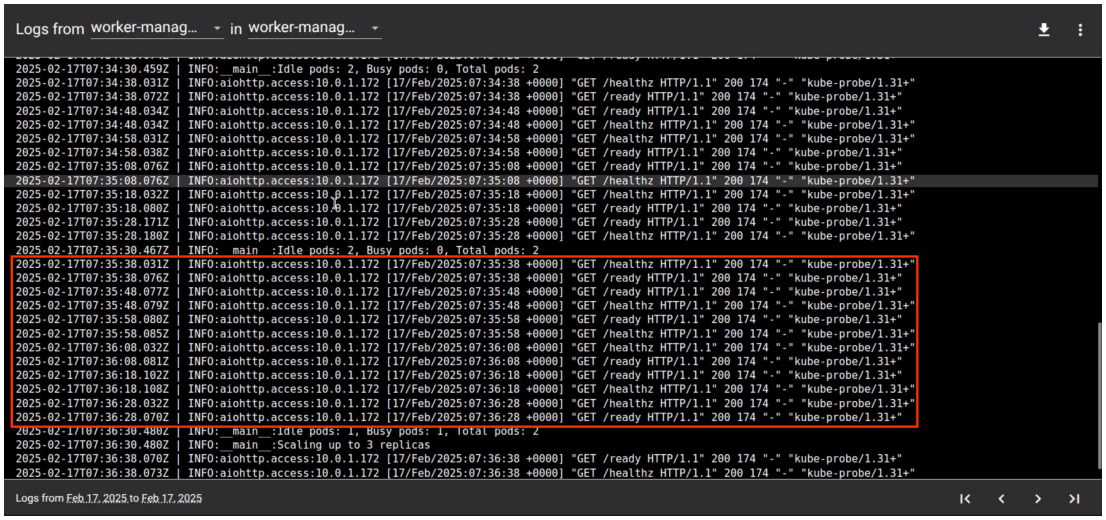

Figure 4.5: Kubernetes Readiness and Liveness Checks

4.5.1 AWS Access Key and CLI Configuration

To interact with AWS services, it is necessary to create an AWS access key. This can be done by navigating to the account credentials section in the AWS Management Console and creating an access key.

Once the access key is generated, the AWS CLI can be configured by running the command:

```
aws configure
```

The CLI will prompt for the access key, secret key, region, and output format. For ease of use, a named profile can also be set up using:

```
aws configure --profile <profile-name>
```

This enables switching between different AWS accounts and regions as needed.

4.5.2 Setting Up Terraform Configuration

Terraform is used to manage infrastructure as code. There are three main Terraform commands:

- `terraform init`: Initializes the configuration, downloads necessary providers,

and sets up modules.

- `terraform plan`: Generates an execution plan to show the changes Terraform will make to the infrastructure. Code A.3 shows an example of the Terraform execution plan output.
- `terraform apply`: Applies the changes to create or update infrastructure resources.

Once we verify the plan, we can apply the changes to the infrastructure using:

```
terraform apply
```

This will update our cloud environment to match the desired state defined in the Terraform configuration files.

4.5.3 Storing State Files in S3

To enable collaboration among multiple developers, the Terraform state file can be stored in an Amazon S3 bucket. The state file contains the current infrastructure state and helps Terraform determine necessary changes for future deployments.

Code 4.6 shows the Terraform configuration for setting up the state bucket. We can set the bucket name and tags as variables to customize the bucket's name and attributes. This can be done by defining the variables in a separate `variables.tf` file.

Code 4.6: Terraform Configuration for Setting Up State Bucket

```
resource "aws_s3_bucket" "terraform_state" {
  bucket = var.bucket_name

  tags = var.tags
}

resource "aws_s3_bucket_versioning" "terraform_state" {
  bucket = aws_s3_bucket.terraform_state.id
  versioning_configuration {
    status = "Enabled"
```

```
  }
}

resource "aws_s3_bucket_public_access_block" "terraform_state" {
  bucket = aws_s3_bucket.terraform_state.id

  block_public_acls    = true
  block_public_policy  = true
  ignore_public_acls   = true
  restrict_public_buckets = true
}
```

Amazon S3 is a highly durable, scalable, and secure object storage service provided by AWS [60]. By leveraging S3 for Terraform state file storage, teams can ensure that the state file is reliably stored and accessible across all environments. This approach also promotes collaboration, consistency, and secure infrastructure management.

To prevent race conditions when multiple developers modify the infrastructure simultaneously, a DynamoDB table [61] is used to store the state lock. Code 4.7 shows the DynamoDB table configuration.

Code 4.7: Terraform Configuration for Setting Up DynamoDB Table

```
resource "aws_dynamodb_table" "terraform-lock" {
    name = "terraform-lock"
    hash_key = "LockID"
    read_capacity = 10
    write_capacity = 10

    attribute {
      name = "LockID"
      type = "S"
    }
```

```
    tags = {
      Name = "Terraform Lock Table"
    }
}
```

The Terraform backend configuration can then be updated to use the S3 bucket and DynamoDB table, as shown in Listing 4.8.

Code 4.8: Terraform Backend Configuration

```
terraform {
    backend "s3" {
        bucket         = "terraform-state-bucket"
        key            = "terraform.tfstate"
        region         = "ap-southeast-1"
        dynamodb_table = "terraform-lock"
    }
}
```

4.5.4 Deploying Terraform Resources

The Terraform configuration files reference Song's Terraform modules [5], with enhancements such as storing the Terraform state in an S3 bucket and simplifying deployment by creating an EFS CSI driver for the EFS volume. This driver allows Kubernetes pods to mount the EFS volume, enabling seamless integration.

The Terraform configuration for creating the EFS CSI driver and attaching it to the EFS volume is shown in Code A.4. This configuration sets up the necessary IAM role, policy attachments, and the EFS CSI driver for an EKS cluster.

To deploy these resources, the following commands are executed:

- `terraform init`: Initializes the Terraform configuration and downloads the necessary providers and modules.
- `terraform plan`: Generates an execution plan that shows the changes Terraform

will make to the infrastructure.

- `terraform apply`: Applies the changes to the infrastructure and deploys the resources.

4.5.5 Deploying EFS and Attaching to Models

To store the ASR models, an Elastic File System (EFS) can be deployed. The following steps outline the deployment process:

1. Create an EC2 instance in a public subnet.
2. Attach the EFS volume to the instance.
3. Copy the ASR model files to the EFS volume.

Code A.5 shows the Terraform configuration for setting up an EC2 instance and an SSH key.

After the EC2 instance is created, follow these steps to transfer the model files:

1. Retrieve the EC2 public IP from the AWS console.
2. SSH into the EC2 instance using `ssh -i <key.pem> ec2-user@<public-ip>`.

On the EC2 instance:

1. `sudo mkdir -p /mnt/efs`: Create a directory to mount the EFS volume.
2. `sudo mount -t nfs4 -o nfsvers=4.1 <mount-target address>:/ /mnt/efs`: Mount the EFS volume to the directory.
3. `sudo chmod go+rw /mnt/efs`: Update the permissions of the directory to allow read and write access.
4. `scp -r -i "<private key name>.pem" <model-directory>@<public-ip>:/mnt/efs`: Copy the model files to the EFS volume.

4.6 Kubernetes Cluster

`kubectl` is the primary command-line tool for interacting with Kubernetes clusters. It allows users to create, update, and manage resources within the cluster. Below are some commonly used `kubectl` commands:

- `kubectl apply -f <file>`: Applies the configuration defined in the YAML file to the cluster.
- `kubectl get <resource>`: Retrieves information about the specified resource.
- `kubectl describe <resource> <name>`: Provides detailed information about the specified resource.
- `kubectl logs <pod-name>`: Displays the logs of the specified pod.
- `kubectl exec -it <pod-name> -- /bin/bash`: Opens a shell in the specified pod.
- `kubectl delete <resource> <name>`: Deletes the specified resource.

To connect to the EKS cluster, the AWS CLI can be used to update the `kubeconfig` file with the cluster details. The following command retrieves the cluster configuration and updates the kubeconfig file:

```
aws eks --region <region> update-kubeconfig --name <cluster-name>
```

Once the kubeconfig file is updated, kubectl can be used to interact with the EKS cluster.

4.6.1 Deploying ASR System Components

The ASR system can be deployed to the Kubernetes cluster using the Kubernetes manifest files provided in the repository. The deployment process can be initiated with the following command:

```
kubectl apply -f <file>
```

The deployment includes the following components:

- **Live Processing Service Deployment:** Deploys the Live Processing Service to manage transcription tasks.
- **Worker Manager Server and Monitor Deployment:** Deploys the Worker Manager Server to handle gRPC requests and the Worker Manager Monitor to manage worker health.
- **Worker Deployment:** Deploys the worker pods responsible for processing audio data.
- **Persistent Volume and Persistent Volume Claim:** Defines the storage requirements for the EFS volume used by the ASR system.

4.6.2 Horizontal Pod Autoscaler (HPA)

The Kubernetes Horizontal Pod Autoscaler (HPA) automatically adjusts the number of pods in a deployment based on resource utilization, such as CPU or memory, or custom metrics. This ensures efficient resource allocation and improved system responsiveness under varying workloads.

The HPA configuration is defined in a YAML file and applied to the cluster using:

```
kubectl apply -f <file>
```

Code 4.9 provides an example HPA configuration for the Live Processing Server.

Code 4.9: Horizontal Pod Autoscaler for Live Processing Server

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: live-processing-server-hpa
  namespace: asr-sdk
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: live-processing-server-deployment
```

```
minReplicas: 2
maxReplicas: 10
metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 75
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 75
```

This configuration defines the following:

- **Target Deployment:** The HPA scales the `live-processing-server-deployment`.
- **Replica Scaling Range:** The deployment is maintained between a minimum of 2 and a maximum of 10 replicas.
- **Scaling Metrics:** The autoscaler monitors CPU and memory utilization, scaling up or down when usage exceeds or falls below 75%.

Testing the HPA

To verify the HPA's behavior under load, a benchmarking tool such as `wrk` can be used to simulate traffic.

The following command generates a load test by sending 25 concurrent WebSocket requests for 30 seconds using 12 threads:

```
wrk -t12 -c25 -d30s ws://<live-processing-service-url>/client/ws/speech
```

To monitor the HPA scaling behavior in real-time, use:

```
kubectl get hpa -n asr-sdk live-processing-server-hpa --watch
```

As shown in Figure 4.6, after 30 seconds of sustained load, the CPU utilization surpasses the 75% threshold, triggering the HPA to scale the deployment from 2 to 7 replicas.

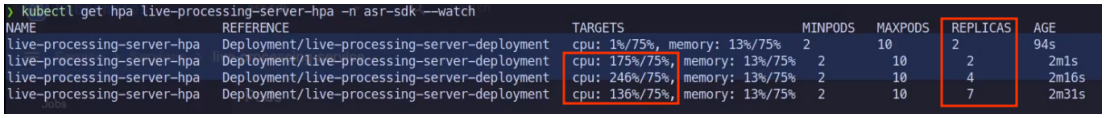

Figure 4.6: Testing of HPA with wrk

4.6.3 Helm Charts

Helm charts were used to deploy the RabbitMQ and Redis clusters. The charts used were from Bitnami, which provides pre-configured Helm charts for popular applications, including RabbitMQ and Redis.

Users can customize their deployments by specifying configuration parameters in a `values.yaml` file. This file allows for modifications such as the number of replicas, resource limits, and persistence settings. The full list of configurable values is available in the official Helm chart repository.

The following commands deploy the Redis and RabbitMQ clusters using Helm:

```
# Deploy Redis cluster
helm install redis oci://registry-1.docker.io/bitnamicharts/redis \
  -f values.yaml --namespace asr-sdk --create-namespace


# Deploy RabbitMQ cluster
helm install rabbitmq oci://registry-1.docker.io/bitnamicharts/rabbitmq \
  -f values.yaml --namespace asr-sdk --create-namespace
```

To ensure data persistence for RabbitMQ and Redis, Amazon Elastic Block Store (EBS) can be configured as the persistent storage backend.

1. Install the EBS CSI driver on the EKS cluster if it is not already installed.

2. Annotate the EBS CSI service account with the IAM role created for EBS via Terraform using the following command:

```
kubectl annotate serviceaccount ebs-csi-controller-sa -n kube-system \
  eks.amazonaws.com/role-arn=<ARN_of_EBS_CSI_Driver_Role>
```

4.6.4 Kubernetes Dashboard

To effectively visualize the Kubernetes cluster and monitor deployed resources, the Kubernetes Dashboard provides a graphical interface for managing the cluster, inspecting workloads, and troubleshooting issues. The dashboard allows users to view real-time metrics, manage deployments, and analyze resource utilization. Figure 4.7 illustrates an example of the Kubernetes Dashboard.

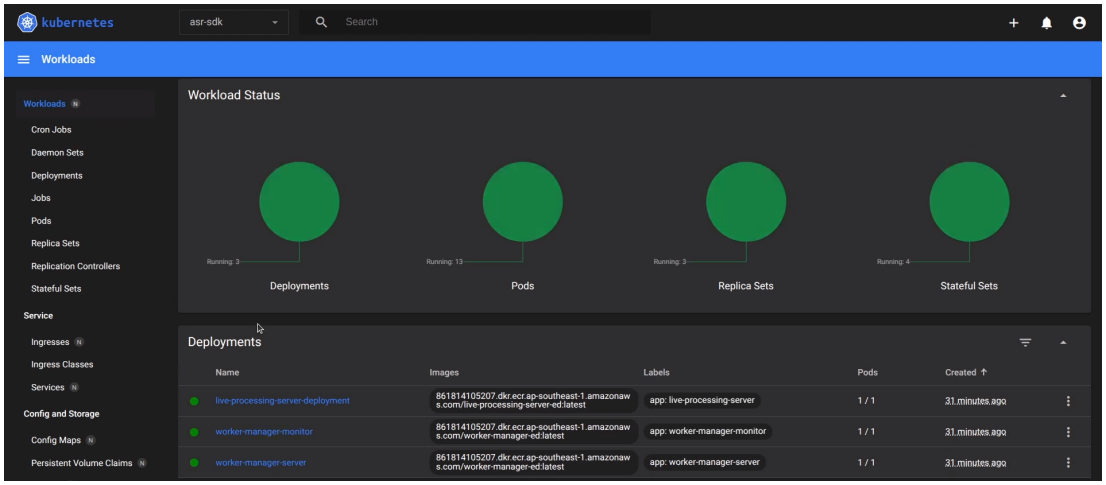

Figure 4.7: Kubernetes Dashboard

Installing the Kubernetes Dashboard

The Kubernetes Dashboard can be installed using Helm with the following commands:

```
helm repo add kubernetes-dashboard https://kubernetes.github.io/dashboard/
helm upgrade --install kubernetes-dashboard \
  kubernetes-dashboard/kubernetes-dashboard \
  --create-namespace --namespace kubernetes-dashboard
```

Configuring Access and Permissions

Before accessing the dashboard, a set of required resources must be configured, including:

- **Service Account:** Grants access to the dashboard.
- **Cluster Role Binding:** Assigns permissions to the service account.
- **Secret Token:** Authenticates the dashboard.

The Kubernetes configuration file in Code A.6 sets up these dependencies.

Accessing the Dashboard

Once the dashboard and necessary permissions are set up, port forwarding can be used to expose the dashboard locally:

```
kubectl -n kubernetes-dashboard port-forward \
  svc/kubernetes-dashboard-kong-proxy 8443:443
```

After running this command, open a web browser and navigate to: `https://localhost:8443`.

Generating Token

To log in to the dashboard, generate an authentication token using:

```
kubectl -n kubernetes-dashboard create token admin-user
```

Chapter 5

Conclusion and Future Work

To be done.

Bibliography

[1] H. Liu, L. P. Garcia, X. Zhang, A. W. H. Khong and S. Khudanpur, *Enhancing code-switching speech recognition with interactive language biases*, 2023. arXiv: 2309.16953 [eess.AS]. [Online]. Available: https://arxiv.org/abs/2309.16953.

[2] AI Singapore, *Speech lab*. [Online]. Available: https://aisingapore.org/aiproducts/speech-lab/ Accessed: 21/01/2025.

[3] O. Chia, "Ai masters singlish in key breakthrough to serve healthcare and patients' needs," *The Straits Times*, 14th Nov. 2024. [Online]. Available: https://www.straitstimes.com/singapore/ai-masters-singlish-in-key-breakthrough-to-serve-healthcare-and-patients-needs Accessed: 21/01/2025.

[4] I. Liew, "Plan to use ai to help emergency call operators," *The Straits Times*, 10th Jul. 2018. [Online]. Available: https://www.straitstimes.com/singapore/plan-to-use-ai-to-help-emergency-call-operators Accessed: 21/01/2025.

[5] Y. Song, "Deploying speech recognition system using kubernetes cluster - infrastructure as code with terraform and terragrunt," B.Eng. dissertation, Nanyang Technol. Univ., Singapore, 2023. [Online]. Available: https://hdl.handle.net/10356/165869.

[6] K. S. Lee, "Deploying asr system for scalability and robustness on aws," B.Eng. dissertation, Nanyang Technol. Univ., Singapore, 2022. [Online]. Available: https://hdl.handle.net/10356/156701.

[7] T. Putra, "Deploying automatic speech recognition system for scalability, reliability, and security with kubernetes," B.Eng. dissertation, Nanyang Technol. Univ., Singapore, 2023. [Online]. Available: https://hdl.handle.net/10356/171933.

[8] P. Kookarinrat and Y. Temtanapat, "Design and implementation of a decentralized message bus for microservices," in *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, 2016, pp. 1–6. DOI: 10.1109/JCSSE.2016.7748869.

[9] S. Newman, *Building Microservices, 2nd Edition*. O'Reilly Media, 2021.

[10] L. De Lauretis, "From monolithic architecture to microservices architecture," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019, pp. 93–96. DOI: 10.1109/ISSREW.2019.00050.

[11] gRPC, *About grpc*. [Online]. Available: https://grpc.io/about/ Accessed: 22/01/2025.

[12] M. Niswar, R. Safruddin, A. Bustamin and I. Aswad, "Performance evaluation of microservices communication with rest, graphql, and grpc," *International Journal of Electronics and Telecommunications*, pp. 429–436, Jun. 2024. DOI: 10.24425/ijet.2024.149562.

[13] Google LLC, *Protocol buffers*. [Online]. Available: https://protobuf.dev/ Accessed: 22/01/2025.

[14] Amazon Web Services, *What's the difference between docker and a vm?* [Online]. Available: https://aws.amazon.com/compare/the-difference-between-docker-vm/ Accessed: 22/01/2025.

[15] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014. DOI: 10.1109/MCC.2014.51.

[16] Red Hat, *What is kubernetes?* [Online]. Available: https://www.redhat.com/en/topics/containers/what-is-kubernetes Accessed: 22/01/2025.

[17] B. Burns, B. Grant, D. Oppenheimer, E. Brewer and J. Wilkes, "Borg, omega, and kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016. [Online]. Available: http://queue.acm.org/detail.cfm?id=2898444.

[18] VMware, *What is kubernetes?* [Online]. Available: https://www.vmware.com/topics/kubernetes#kubernetes-features Accessed: 22/01/2025.

[19] Red Hat, *What is helm?* [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-helm Accessed: 22/01/2025.

[20] Helm, *Charts*. [Online]. Available: https://helm.sh/docs/topics/charts/ Accessed: 22/01/2025.

[21] Amazon Web Services, *Aws cloud services*. [Online]. Available: https://aws.amazon.com/products/ Accessed: 22/01/2025.

[22] Amazon Web Services, *What is amazon vpc?* [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html Accessed: 22/01/2025.

[23] M. Haken, *Improving performance and reducing cost using availability zone affinity*. [Online]. Available: https://aws.amazon.com/blogs/architecture/improving-performance-and-reducing-cost-using-availability-zone-affinity/ Accessed: 22/01/2025.

[24] Amazon Web Services, *What is amazon ec2?* [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html Accessed: 22/01/2025.

[25] Amazon Web Services, *Amazon ec2 instance types*. [Online]. Available: https://aws.amazon.com/ec2/instance-types/ Accessed: 22/01/2025.

[26] Amazon Web Services, *Amazon ec2 on-demand pricing*. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/ Accessed: 22/01/2025.

[27] Amazon Web Services, *What is iam?* [Online]. Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html Accessed: 22/01/2025.

[28] Amazon Web Services, *How iam works*. [Online]. Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/intro-structure.html Accessed: 22/01/2025.

[29] Amazon Web Services, *What is amazon eks?* [Online]. Available: https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html Accessed: 22/01/2025.

[30] Amazon Web Services, *Deploy amazon eks clusters across cloud and on-premises environments*. [Online]. Available: https://docs.aws.amazon.com/eks/latest/userguide/eks-deployment-options.html Accessed: 22/01/2025.

[31] Amazon Web Services, *What is amazon elastic container registry?* [Online]. Available: https://docs.aws.amazon.com/AmazonECR/latest/userguide/what-is-ecr.html Accessed: 22/01/2025.

[32] Amazon Web Services, *Identity and access management for amazon elastic container registry*. [Online]. Available: https://docs.aws.amazon.com/AmazonECR/latest/userguide/security-iam.html Accessed: 22/01/2025.

[33] Amazon Web Services, *Scan images for software vulnerabilities in amazon ecr*. [Online]. Available: https://docs.aws.amazon.com/AmazonECR/latest/userguide/image-scanning.html Accessed: 22/01/2025.

[34] Amazon Web Services, *Automate the cleanup of images by using lifecycle policies in amazon ecr*. [Online]. Available: https://docs.aws.amazon.com/AmazonECR/latest/userguide/LifecyclePolicies.html Accessed: 22/01/2025.

[35] Amazon Web Services, *What is amazon elastic file system?* [Online]. Available: https://docs.aws.amazon.com/efs/latest/ug/whatisefs.html Accessed: 22/01/2025.

[36] Amazon Web Services, *How amazon efs works*. [Online]. Available: https://docs.aws.amazon.com/efs/latest/ug/how-it-works.html Accessed: 22/01/2025.

[37] Amazon Web Services, *Amazon efs performance*. [Online]. Available: https://docs.aws.amazon.com/efs/latest/ug/performance.html Accessed: 22/01/2025.

[38] Amazon Web Services, *What is infrastructure as code?* [Online]. Available: https://aws.amazon.com/what-is/iac/ Accessed: 22/01/2025.

[39] HashiCorp, *What is terraform?* [Online]. Available: https://developer.hashicorp.com/terraform/intro Accessed: 22/01/2024.

[40] HashiCorp, *Terraform language documentation*. [Online]. Available: https://developer.hashicorp.com/terraform/language/ Accessed: 22/01/2025.

[41] Y. Qiu *et al.*, "Simplifying cloud management with cloudless computing," in *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, ser. HotNets '23, Cambridge, MA, USA: Association for Computing Machinery, 2023, pp. 95–101, ISBN: 9798400704154. DOI: 10.1145/3626111.3628206. [Online]. Available: https://doi-org.remotexs.ntu.edu.sg/10.1145/3626111.3628206.

[42] HashiCorp, *Use cases*. [Online]. Available: https://developer.hashicorp.com/terraform/intro/use-cases Accessed: 22/01/2025.

[43] IBM, *What is redis?* [Online]. Available: https://www.ibm.com/think/topics/redis Accessed: 22/01/2025.

[44] Amazon Web Services, *What is a message queue?* [Online]. Available: https://aws.amazon.com/message-queue/ Accessed: 22/01/2025.

[45] G. Fu, Y. Zhang and G. Yu, "A fair comparison of message queuing systems," *IEEE Access*, vol. 9, pp. 421–432, 2020.

[46] Amazon Web Services, *What is apache kafka?* [Online]. Available: https://aws.amazon.com/what-is/apache-kafka/ Accessed: 22/01/2025.

[47] Apache Software Foundation, *Documentation*. [Online]. Available: https://kafka.apache.org/documentation/ Accessed: 22/01/2025.

[48] Broadcom Inc., *Which protocols does rabbitmq support?* [Online]. Available: https://www.rabbitmq.com/docs/protocols Accessed: 22/01/2025.

[49] L. Johansson, *Part 1: Rabbitmq for beginners - what is rabbitmq?* [Online]. Available: https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html Accessed: 22/01/2025.

[50] PubNub, *What is rabbitmq?* [Online]. Available: https://www.pubnub.com/guides/rabbitmq/ Accessed: 22/01/2025.

[51] P. Dobbelaere and K. S. Esmaili, "Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper," Jun. 2017, pp. 227–238. DOI: 10.1145/3093742.3093908.

[52] Broadcom Inc., *Negative acknowledgements*. [Online]. Available: https://www.rabbitmq.com/docs/nack Accessed: 22/01/2025.

[53] D. Wang, X. Wang and S. Lv, "An overview of end-to-end automatic speech recognition," *Symmetry*, vol. 11, no. 8, p. 1018, 2019.

[54] S. Gunja, *What is chaos engineering?* [Online]. Available: https://www.dynatrace.com/news/blog/what-is-chaos-engineering/ Accessed: 22/01/2025.

[55] Chaos Mesh, *Basic features*. [Online]. Available: https://chaos-mesh.org/docs/basic-features/ Accessed: 22/01/2025.

[56] Chaos Mesh, *Chaos mesh overview*. [Online]. Available: https://chaos-mesh.org/docs/ Accessed: 22/01/2025.

[57] Amazon Web Services, *What is aws fault injection service?* [Online]. Available: https://docs.aws.amazon.com/fis/latest/userguide/what-is.html Accessed: 22/01/2025.

[58] Tornado, *Tornado web server*. [Online]. Available: https://www.tornadoweb.org/en/stable/.

[59] D. Povey *et al.*, "The kaldi speech recognition toolkit," in *IEEE 2011 workshop on automatic speech recognition and understanding*, IEEE Signal Processing Society, 2011.

[60] Amazon Web Services, *What is amazon s3?* [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html Accessed: 26/01/2025.

[61] Amazon Web Services, *What is amazon dynamodb?* [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html Accessed: 26/01/2025.

Appendix A

Relevant Code Snippets

Code A.1: Example Code Documentation

```
def _start_task(self, task_queue, instance_id=1):
    """
    Start the task and set the state of the worker to BUSY.

    Args:
        task_queue (str): Task queue to be processed by the worker.
        instance_id (int): Instance ID of the task.
    """
```

Code A.2: Worker Manager Server gRPC Proto File

```
syntax = "proto3";

package worker_manager;

service WorkerManagerService {
    rpc AllocateWorker (AllocateWorkerRequest) returns
        (AllocateWorkerResponse);
    rpc SendHeartbeat (SendHeartbeatRequest) returns
        (SendHeartbeatResponse);
}
```

```
message AllocateWorkerRequest {
    string model_name = 1;
    string task_queue = 2;
}

message AllocateWorkerResponse {
    bool success = 1;
    string worker_name = 2;
    string message = 3;
}

message SendHeartbeatRequest {
    string worker_name = 1;
    string model_name = 2;
    bool final = 3;
}

message SendHeartbeatResponse {
    bool success = 1;
    string message = 2;
}
```

Code A.3: Terraform Plan Output

```
Terraform used the selected providers to generate the following
    execution plan. Resource actions are indicated with the following
    symbols:
+ create
<= read (data resources)

Terraform will perform the following actions:

    # module.eks.aws_eks_addon.efs_csi_driver will be created
```

```
    + resource "aws_eks_addon" "efs_csi_driver" {
        + addon_name        = "aws-efs-csi-driver"
        + addon_version     = "v2.1.3-eksbuild.1"
        + arn               = (known after apply)
        + cluster_name      = "ed-fyp-eks-cluster"
        + configuration_values = (known after apply)
        + created_at        = (known after apply)
        + id                = (known after apply)
        + modified_at       = (known after apply)
        + tags_all          = {
            + "Environment" = "Development"
            + "Owner"     = "Edmund"
            + "Terraform" = "True"
        }

... (output truncated) ...

Plan: 37 to add, 0 to change, 0 to destroy.
```

Code A.4: Terraform Configuration for Creating EFS CSI Driver

```
module "eks" {
    source = "terraform-aws-modules/eks/aws"
    version = "~> 20.31"

    cluster_name  = var.cluster_name
    cluster_version = "1.31"

    vpc_id    = var.vpc_id
    subnet_ids = var.subnet_ids

    cluster_endpoint_private_access     = true
    cluster_endpoint_public_access      = true
    enable_cluster_creator_admin_permissions = true
```

```
    cluster_compute_config = {
    enabled   = true
    node_pools = ["general-purpose"]
    }
}

resource "aws_eks_addon" "efs_csi_driver" {
    cluster_name = module.eks.cluster_name
    addon_name   = "aws-efs-csi-driver"
    addon_version = "v2.1.3-eksbuild.1"
}

resource "aws_iam_role" "efs_csi_role" {
    name = "EKS_EFS_CSI_DriverRole"
    assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
        {
        Action = [
            "sts:AssumeRoleWithWebIdentity",
        ]
        Principal = {
            Federated =
                "arn:aws:iam::${data.aws_caller_identity.current.account_id}:
                oidc-provider/${module.eks.cluster_oidc_issuer_url}"
        }
        Effect = "Allow"
        Condition = {
            StringLike = {
            "${module.eks.cluster_oidc_issuer_url}:sub" =
                "system:serviceaccount:kube-system:efs-csi-*",
            "${module.eks.cluster_oidc_issuer_url}:aud" =
```

```
            "sts.amazonaws.com"
          }
        }
        },
    ]
    })
}

resource "aws_iam_role_policy_attachment"
    "efs_csi_driver_policy_attachment" {
    role      = aws_iam_role.efs_csi_role.name
    policy_arn =
        "arn:aws:iam::aws:policy/service-role/AmazonEFSCSIDriverPolicy"
}

data "aws_caller_identity" "current" {}
```

Code A.5: Terraform Configuration for Setting Up EC2 Instance

```
# Generate new private key
resource "tls_private_key" "my_key" {
    algorithm = "RSA"
}

# Generate a key-pair with above key
resource "aws_key_pair" "key-pair" {
    key_name  = "${var.owner}-key"
    public_key = tls_private_key.my_key.public_key_openssh
}

# Saving Key Pair for ssh login for Client if needed
resource "null_resource" "save_key_pair" {
    provisioner "local-exec" {
    command = "echo '${tls_private_key.my_key.private_key_pem}' >
```

```
        '${aws_key_pair.key-pair.key_name}'.pem && chmod 400
        '${aws_key_pair.key-pair.key_name}'.pem"
    }
}

# create ec2 resource for mounting model
resource "aws_instance" "ec2-instance" {
    ami                     = "ami-0e48a8a6b7dc1d30b"
    instance_type           = "t2.micro"
    key_name                = aws_key_pair.key-pair.key_name
    subnet_id               = var.public_subnet_ids[0]
    vpc_security_group_ids  = [var.security_group_nfs_ssh]
    associate_public_ip_address = true

    tags = {
    Name = "${var.owner}-model-transfer"
    }
}
```

Code A.6: Kubernetes Configuration for Setting Up Dashboard Dependencies

```
apiVersion: v1
kind: ServiceAccount
metadata:
    name: admin-user
    namespace: kubernetes-dashboard
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
    name: admin-user
roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: ClusterRole
```

```
    name: cluster-admin
subjects:
    - kind: ServiceAccount
    name: admin-user
    namespace: kubernetes-dashboard
---
apiVersion: v1
kind: Secret
metadata:
    name: admin-user
    namespace: kubernetes-dashboard
    annotations:
    kubernetes.io/service-account.name: "admin-user"
type: kubernetes.io/service-account-token
```