Cloud-Edge Continuum Computing with Kubernetes Deployment

Final Report – Undergrad Thesis

Dhylan Usi

Course: CS4490Z - Thesis

Department of Computer Science

Western University

April 11, 2025

Project supervisor: Dr. Hanan Lutfiyya, Department of Computer Science

Course instructor: Dr. Nazim Madhavji, Department of Computer Science

Glossary

- API (Application Programming Interface): A set of protocols and tools for building software applications, specifying how software components should interact.
- AWS (Amazon Web Services): A comprehensive cloud computing platform offered by Amazon, providing a wide range of infrastructure services such as compute power, storage, and networking.
- CGNAT (Carrier-Grade Network Address Translation): A technique used by Internet service providers to allow multiple customers to share a single public IPv4 address, often complicating direct external access to devices.
- CEC (Cloud–Edge Continuum): An integrated computing environment that spans from centralized cloud data centers to edge devices, enabling low-latency, distributed processing of applications.
- EC2 (Elastic Compute Cloud): A web service provided by AWS that offers scalable computing capacity in the cloud.
- FaaS (Function as a Service): A category of cloud services that allows users to deploy individual functions or pieces of business logic without managing the underlying infrastructure.
- IoT (Internet of Things): A network of physical devices embedded with sensors, software, and connectivity, enabling the exchange and analysis of data.
- K8s (Kubernetes): An open-source container orchestration system for automating the deployment, scaling, and management of containerized applications.
- KubeEdge: An open-source framework that extends Kubernetes for edge computing by providing additional components to manage and support edge devices.
- MQTT (Message Queuing Telemetry Transport): A lightweight messaging protocol designed for small sensors and mobile devices, optimized for low-bandwidth, high-latency networks.
- MEC (Multi-access Edge Computing): A network architecture concept that enables cloud computing capabilities and an IT service environment at the edge of the telecommunications network.
- QoS (Quality of Service): A measure of the performance level of a service, often used in networking to refer to the guarantee of delivery parameters (such as latency or throughput) for data transmission.
- TTS (Text-to-Speech): Technology that converts written text into spoken voice output.
- VM (Virtual Machine): An emulated computer system that runs an operating system and applications, providing the functionality of a physical computer.
- IaC (Infrastructure as Code): The practice of managing and provisioning computing infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.

Structured Abstract

Background:
Edge computing has developed as a complementary approach to cloud computing with the provision of computational resources close to end-users and sources of data. It minimizes latency and bandwidth consumption for real-time use cases like multimedia processing and IoT analytics. Despite presenting challenges like limited connectivity, resource diversity, and the necessity of new orchestration frameworks, these remain when multiparty environments across edge devices, Multi-access Edge Computing (MEC), and cloud data centers have to be integrated.

Methods:
This thesis proposes an audio processing pipeline using microservices on three layers: an edge device (laptop), an AWS MEC node, and AWS cloud. All services, including Text-to-Speech and audio processing, were built with Python, containerized with Docker, and managed using Kubernetes and Docker Compose. MQTT was used for communication between layers, with Terraform for automating infrastructure setup.

Results:
Implementation resulted in a working, distributed microservice chain in which edge TTS service provides audio output, format conversion and profanity detection are carried out by MEC services and censorship and compression by cloud services. With thorough log analysis and environment staging tests, cross-layer communication and correctness of operations were ensured. Reproducibility of the entire environment without the necessity for on-site installation of local dependencies is made possible through the publication of Docker images on DockerHub.

Conclusion:
This work shows that a well-structured microservice pipeline across edge, MEC, and cloud layers can work for applications needing low latency. While further analysis is needed for performance, this proof of concept highlights the effectiveness of using available tools like Kubernetes and Docker in cloud-edge setups. This research lays groundwork for future projects focusing on better management and reliability.

Table of Contents

Contents

Glossary....2
Structured Abstract....3
Table of Contents....4
1. Introduction....6
Problem Description:....6
Focus of this Thesis:....6
Key Research Questions:....6
Key Results and Contributions:....7
Novelty and Significance:....7
Report Structure:....7
2. Background and Related Work....7
2.1 Edge Computing and Cloud Continuum....7
2.2 Orchestration Frameworks for Cloud–Edge Systems....7
2.3 Serverless Microservices Across Edge and Cloud....8
2.4 Analysis and Research Gap....8
3. Research Objectives....9
4. Methodology....9
4.1 Design Approach....10
4.2 Technology Stack and Tools....10
4.3 Testing Strategy....10
4.4 Methodology Limitations....11
5. Implementation....11
5.1 Infrastructure Provisioning and Kubernetes Setup....11
5.2 Microservice Development and Containerization....12
5.3 Implementation and Testing Strategy....13
6. Discussion....15
6.1 Threats to Validity....15
6.2 Implications of the Research Results....15

6.3 Limitations of the Results 16
6.4 Generalizability of the Results 16
7. Conclusions 17
8. Future Work and Lessons Learned 17
Future Work: 17
Lessons Learned: 18
Appendix (Figures) 20

1. Introduction

In recent years, edge computing has emerged as a complement to cloud computing, providing computational resources closer to data sources and end-users. This approach significantly reduces latency and bandwidth usage for real-time applications such as multimedia processing, augmented reality, and IoT (Internet of Things) analytics. Unlike traditional cloud architectures—which assume stable, high-bandwidth networks and homogeneous resources—edge computing must handle intermittent connectivity and constrained resources. Consequently, new orchestration frameworks are necessary for intelligently deploying and managing services across the cloud–edge continuum (Bartolomeo et al., 2023; Alarbi et al., 2024).

Problem Description:
Although multi-tier cloud–MEC–edge computing has garnered considerable research interest, real-world implementations remain limited. There exists a noticeable gap between theoretical orchestration frameworks and their practical deployment. This thesis addresses the challenge of designing and implementing a distributed microservice pipeline for real-time audio processing, leveraging edge, MEC (Multi-access Edge Computing), and cloud resources. Specifically, it explores how application logic can be effectively partitioned, how reliable cross-layer communication can be established, and how consistent service deployment can be achieved despite infrastructure heterogeneity.

Focus of this Thesis:
The core of this work is a proof-of-concept implementation involving three distinct layers: a simulated edge device (laptop), an AWS Wavelength MEC node, and an AWS cloud region. The targeted application is an audio-processing pipeline that transforms textual input into audio output. At the edge, a Text-to-Speech (TTS) microservice generates initial audio. MEC-layer services handle audio format conversion (MP3 to WAV) and profanity detection. In the cloud, a censorship microservice redacts profanities, and a compression microservice optimizes audio storage and transmission. This setup reflects an architectural strategy where latency-sensitive tasks occur close to users, while resource-intensive operations utilize cloud resources. The focus is not on developing novel algorithms but rather on the real-time integration and deployment of containerized microservices across the continuum.

Key Research Questions:
The thesis investigates several critical questions:

1. How can a pipeline of containerized microservices be effectively deployed across edge, MEC, and cloud environments?
2. What messaging approach best facilitates reliable, low-latency communication between services in different layers?
3. What practical challenges arise when using cloud orchestration tools like Kubernetes in multi-tier deployments?
4. How does the distributed pipeline perform concerning latency and functional correctness?

Answering these questions helps determine the feasibility and complexity associated with multi-tier cloud–edge deployments.

Key Results and Contributions:
This research led to a functional microservice pipeline where each piece was successfully contained and deployed: TTS at the edge, format conversions and profanity detection at MEC, and censorship and compression in the cloud. The system ran correctly, proving that we can perform real-time audio processing and comply with ethical standards. While our ability to test directly on the latest 5G infrastructure was limited, we verified everything through staged tests. This beyond just theoretical frameworks, showing a real implementation across the entire system.

Novelty and Significance:
By providing a practical example of a multi-tier microservice pipeline, this thesis offers useful insights into cloud-edge orchestration. It serves as a case study for the broader ECO framework being developed. This work illustrates how an ECO subsystem can work, focusing on automation and effective messaging between layers. These findings can guide future tools and help connect theoretical ideas with practical applications.

Report Structure:
The rest of the report is organized as follows: Section 2 reviews background concepts and research on cloud-edge orchestration. Section 3 states the research objectives in detail. Section 4 outlines the methodology, including the approach and technologies used. Section 5 shares the implementation results and system architecture. Section 6 discusses wider implications and limitations. Section 7 wraps up with key takeaways. Finally, Section 8 looks at future work and lessons learned.

2. Background and Related Work

This section reviews relevant research on edge computing, orchestration frameworks, and identifies the gap addressed by this thesis.

2.1 Edge Computing and Cloud Continuum

Edge computing extends cloud capabilities closer to end-users, significantly reducing latency and bandwidth for real-time applications like augmented reality, autonomous vehicles, and smart cities. With the rise of 5G, the cloud-edge continuum—encompassing resources from end-devices to telecom infrastructure—has become crucial. Multi-access Edge Computing (MEC), standardized by ETSI and exemplified by services like AWS Wavelength, integrates compute capabilities directly into telecom networks, enabling lower-latency application deployments.

However, edge deployments face inherent challenges: edge nodes are often resource-constrained, geographically dispersed, and suffer from intermittent connectivity and unreliable network conditions. This complexity demands careful partitioning of application logic, balancing latency-sensitive tasks at the edge and resource-intensive computation in the cloud.

2.2 Orchestration Frameworks for Cloud–Edge Systems

Traditional cloud orchestration frameworks like Kubernetes efficiently handle container scheduling, scaling, and fault tolerance but assume stable, high-bandwidth connectivity. These assumptions falter in distributed edge environments, leading to inefficiencies and poor availability (Bartolomeo et al., 2023).

Several frameworks have emerged to address these limitations:

- KubeEdge (Xiong et al., 2018) extends Kubernetes by introducing edge agents (EdgeCore) and a cloud controller, enabling robust operations despite intermittent connectivity. It synchronizes states using EdgeHub/CloudHub, effectively bridging cloud and edge nodes.
- Oakestra (Bartolomeo et al., 2023) proposes hierarchical orchestration with a root orchestrator managing cluster orchestrators across different edge sites. Oakestra uses a lightweight semantic overlay for service discovery, significantly reducing orchestration overhead and improving performance through locality-aware scheduling.
- ECO (Edge Continuum Orchestrator) (Alarbi et al., 2024) focuses on serverless function chains, dynamically placing functions across the cloud–edge continuum. ECO (Edge Continuum Orchestrator) optimizes function placement for latency, bandwidth, and energy efficiency, using a flexible, dynamic orchestration approach.

Despite various innovations, no single orchestration framework has become a definitive standard for cloud-edge environments. Many solutions rely on customized scheduling or clustering strategies, highlighting an ongoing need for practical validation and refinement (Böhm & Wirtz, 2022).

2.3 Serverless Microservices Across Edge and Cloud

Serverless computing (FaaS) offers an attractive approach for edge deployment due to its flexibility and scalability. Functions can be individually placed across layers based on dynamic runtime conditions, improving resource utilization (Eismann et al., 2020). However, serverless deployments at the edge introduce complexities such as cold-start latency and distributed caching requirements.

Our project adopts a containerized microservice approach, bridging traditional orchestration and serverless principles. By leveraging MQTT messaging, our design employs an event-driven, loosely coupled architecture similar to serverless function chaining, enabling straightforward integration across cloud–edge layers (Jeffery et al., 2021).

2.4 Analysis and Research Gap

Analysis:
Existing frameworks demonstrate diverse strategies to address edge orchestration challenges. Kubernetes-based solutions (KubeEdge, Oakestra) adapt cloud-native principles to edge environments, employing distributed or hierarchical control to manage network unreliability and resource heterogeneity. In contrast, ECO takes a fully serverless approach, providing dynamic, per-function orchestration suited to latency-critical scenarios. Nevertheless, these frameworks often depend on assumptions like stable synchronization or manual setup at each edge location (Böhm & Wirtz, 2022). Crucially, real-world validations are rare, with most studies limited to simulations or small-scale laboratory deployments (Bartolomeo et al., 2023).

Research Gap:
A notable gap exists between theoretical orchestration frameworks and their practical deployment in realistic, multi-tier edge environments. Few studies document deployment on actual telecom-edge infrastructure (e.g., AWS Wavelength), leading to uncertainties about real-world applicability. Our thesis addresses this gap by implementing and evaluating a concrete microservice pipeline spanning edge (laptop), MEC (AWS Wavelength), and cloud (AWS). This

deployment provides critical insights into real challenges like cross-layer connectivity, practical deployment automation, and orchestration limitations. By illustrating these issues clearly and concretely, our work helps inform future development of more robust and automated edge orchestration solutions.

3. Research Objectives

The project addresses four primary research objectives, clearly identified for traceability throughout the report:

- O1: Design a Multi-Tier Audio Processing Pipeline**.** Define a distributed architecture across edge, MEC, and cloud layers, specifying each microservice (Text-to-Speech, Conversion, Profanity Detection, Censorship, Compression), ensuring low latency and effective content filtering, and aligning conceptually with the ECO orchestration framework.
- O2: Implement and Deploy Using Cloud and Edge Technologies**.** Containerize microservices using Docker, deploy them via Kubernetes to the edge (laptop), MEC (AWS Wavelength), and cloud (AWS), and automate infrastructure provisioning using Terraform and Kubernetes manifests. Validate basic functionality at each deployment stage.
- O3: Evaluate Pipeline Performance Metrics**.** Analyze end-to-end and per-stage latency, resource utilization (CPU, memory), and network overhead across the cloud–MEC–edge continuum. Identify performance benefits or bottlenecks of multi-tier deployment. *(Planned but not fully completed; see Section 8.)*
- O4: Compare and Integrate Advanced Orchestration Approaches**.** Qualitatively evaluate current manual deployment versus advanced orchestrators (e.g., ECO, KubeEdge). Explore potential orchestration benefits (fault tolerance, scaling, deployment automation), discussing necessary pipeline adaptations. *(Addressed as future work; see Section 8.)*
- O5: Publish reproducible guidelines and code for future research and practical implementations. The goal is to make sure that the entire setup (infrastructure definitions, deployment scripts, application code) is documented and shareable, so that others can replicate the environment or build upon it.

Objectives O1 and O2 form the methodology and results core; O3 and O4 are partially addressed through discussions and future directions.

4. Methodology

This thesis adopts a **system development methodology**, structured into iterative phases: (1) design and planning, (2) local prototyping, (3) cloud/MEC deployment, and (4) integration testing. The development leveraged widely used technologies relevant to cloud–edge environments see Figure 1 (Appendix) for a high level overview.

4.1 Design Approach

We initially designed a high-level architecture to meet **Objective O1**. The architecture consisted of clearly defined microservices: Text-to-Speech (TTS), Conversion (MP3 to WAV), Profanity Detection, Censorship, and Compression. Each microservice follows a request-response pattern via HTTP or MQTT messaging. MQTT was chosen due to its lightweight nature, robustness to intermittent connectivity, and proven suitability for IoT and event-driven systems (Jeffery et al., 2021). We defined MQTT topics and simple JSON-based payloads, facilitating loose coupling and easy integration across services.

4.2 Technology Stack and Tools

- Microservice Development:
  Each microservice was implemented as a containerized Python application due to Python's extensive audio-processing libraries and rapid prototyping capabilities. Libraries utilized included:
  - TTS: pydub
  - Audio Conversion and Compression: pydub, FFmpeg
  - Profanity Detection: SpeechRecognition , better_proffanity
  - Censorship: pydub

Each service exposed HTTP endpoints (via Flask/FastAPI) for testing and health checks but primarily operated through MQTT events.

- Containerization (Docker):
  Services were containerized using Docker to ensure consistent, portable deployment across diverse infrastructures. Dockerfiles leveraged lightweight base images (e.g., python:3.9-slim), following best practices for size and efficiency.
- Orchestration (Kubernetes):
  Kubernetes was employed due to its standardization in industry and suitability for managing distributed microservices. Two Kubernetes clusters were created: one at the MEC layer (AWS Wavelength) and one in the cloud (AWS). Due to complexity, the edge (laptop) service was managed via Docker Compose. Kubernetes deployments were configured using YAML manifests, employing nodeSelectors to ensure microservices ran on designated infrastructure layers.
- Infrastructure Provisioning (Terraform):
  Terraform automated provisioning of AWS EC2 infrastructure, security groups, and network configurations, ensuring repeatability and consistent deployments. The MEC node was provisioned in AWS Wavelength, whereas the cloud node was provisioned in a standard AWS region. Network access, including MQTT communication via external LoadBalancer IPs, was carefully configured to bridge cloud–edge connectivity.

4.3 Testing Strategy

Testing was performed iteratively at multiple levels:

- Unit Testing: Individual service functionalities were verified locally. For instance, TTS audio generation, audio format conversion correctness, profanity detection accuracy, censorship audio adjustments, and compression effectiveness were validated separately.
- Integration Testing: Conducted locally using Docker Compose to confirm end-to-end functional correctness. This verified the MQTT-based message choreography, data flow, and transformations among all microservices.
- Cloud/MEC Integration Testing: Kubernetes deployments on AWS infrastructure were individually tested by triggering microservices via temporary port forwarding or MQTT messages, validating functional correctness in their target deployment environments. Due to network constraints, fully integrated end-to-end testing across edge–MEC–cloud required partial emulation or staged integration tests, with MQTT messaging serving as the integrative backbone.

4.4 Methodology Limitations

While the methodology effectively supported Objectives O1 and O2, it did not fully accomplish Objectives O3 and O4 due to practical constraints (e.g., incomplete detailed performance evaluation and orchestration integration). These limitations are addressed through discussion and identified clearly for future research.

In summary, our methodology combined established development practices (Docker, Kubernetes, Terraform, MQTT) with an experimental approach to address edge orchestration complexities, balancing practical deployment considerations with research rigor.

5. Implementation

The implementation of the Cloud–MEC–Edge microservices subsystem involved setting up the infrastructure, developing the microservices, containerizing them, and deploying them on the respective Kubernetes clusters. This section details the practical steps and technologies used, including **Infrastructure Provisioning with Terraform**, **Microservice Development & Containerization**, and **Kubernetes Deployment**. Throughout implementation, emphasis was placed on automation and reproducibility, in line with objective O5.

5.1 Infrastructure Provisioning and Kubernetes Setup

To create a consistent environment across all layers, we leveraged *Infrastructure-as-Code* using **HashiCorp Terraform**. Terraform scripts were written to provision the required virtual machines and network configurations on AWS for the cloud and MEC layers. Specifically, for the **Cloud layer**, Terraform was used to instantiate an EC2 instance (or a small cluster of instances) in the Canada Central region. For the **MEC layer**, Terraform deployed an EC2 instance in the *Bell Canada Toronto Wavelength Zone* (which is essentially AWS infrastructure at the 5G edge). Both instances were configured with appropriate CPU/RAM to run a Kubernetes single-node master (and optionally a worker). The **Edge layer** (local laptop) did not involve cloud provisioning; instead, we manually configured a local machine as an edge node. In a real scenario, this could be a physical device configured outside of Terraform's scope, or Terraform could be extended to manage on-prem resources via local-exec or other providers. Each AWS instance was configured via **cloud-init** scripts to automatically install and initialize Kubernetes upon launch. This included installing Docker and kubeadm, then using **kubeadm** to create a

Kubernetes cluster. We set up the cloud instance as a single-node Kubernetes control plane (since it runs only a couple of pods, a single node sufficed), and similarly for the MEC instance. Networking was crucial: the Terraform configuration opened specific ports (e.g., port 80/443 for HTTP APIs of our services, and the Kubernetes API port if needed for management) in AWS Security Groups so that the layers could communicate. We ensured that the MEC instance could receive traffic from the Edge device's IP and that the Cloud instance could receive traffic from the MEC. Additionally, **Calico** was installed as the pod network plugin on each cluster to enable internal Kubernetes networking for pods, although cross-cluster communication was done via external networking. By automating these steps, the Kubernetes clusters at Cloud and MEC were brought online with minimal manual intervention.

Once provisioned, each cluster was verified. The local edge cluster was set up using MicroK8s (for example) and joined to the same network as the user's laptop. At this point, we had three independent Kubernetes clusters ready for application deployment. This satisfies objective O1. We confirmed the clusters were functioning by deploying a test Nginx pod on each and checking connectivity (e.g., the cloud cluster's Nginx responded to MEC cluster's curl requests). The infrastructure deployment also laid groundwork for future monitoring: although full metrics collection was out of scope in final delivery, we had installed **Prometheus and Grafana** on the cloud cluster (but did not deeply integrate them). This would allow capturing basic metrics if needed.

5.2 Microservice Development and Containerization

For objective O2, five microservices corresponding to the pipeline stages were developed. Each microservice was implemented as a **Docker container** with a specific functionality:

- Text-to-Speech Service (Edge): This service was written in Python. It uses a text-to-speech library (pydub or similar) to convert input text into spoken audio. The service exposes a simple HTTP API (e.g., a Flask app with a /synthesize endpoint) where a POST request with a text payload triggers the TTS conversion. The output is saved as an MP3 file. The service then returns the audio file (or a reference to it) to the caller. We included a basic vocabulary for test purposes and handled short sentences in English. The Docker image for TTS is based on Python 3 slim, with the TTS library and its dependencies installed. On running, it listens on a configured port for requests.
- Audio Conversion Service (MEC): Implemented in Python using the pydub or ffmpeg library, this service accepts an MP3 audio (either via file upload or a network fetch from the edge output) and converts it to WAV format. The service has an endpoint (e.g., /convert) that can receive the MP3 (or a path/URL to it) and upon invocation uses an audio processing library (ffmpeg via subprocess or pydub.AudioSegment) to read the MP3 and export it as WAV. The resulting WAV file is stored in a temporary location or sent back in the HTTP response to the caller. We containerized this using a base image that has ffmpeg available (e.g., jrottenberg/ffmpeg base or installed via apt in an Ubuntu image).
- Profanity Detection Service (MEC): This service is responsible for analyzing the audio content to find inappropriate language. The development of this service is non-trivial because audio analysis typically requires speech-to-text. Given the project constraints, we took a simplified approach: we leveraged the fact that we know the original text input

(from the TTS stage) in our test scenarios. The profanity detection service, therefore, could operate in two modes: (a) **Text-based** – if the original transcript is provided alongside the audio, simply scan the text for profane words using a dictionary of banned words; or (b) **Audio-based** – a placeholder for an actual speech-to-text integration (which we did not fully implement). In our tests, we used mode (a) for simplicity. The service exposes an endpoint (e.g., /analyze) that takes either a text string or the WAV file. It then checks each word against a profanity list. If a banned word is found, it calculates the approximate time index in the audio. To do this, since we know the order of words, we can estimate times by the word index multiplied by an average word duration (assuming the TTS uses consistent speech rate). This yields start and end times (in seconds or milliseconds) for each profane word. The service returns a list of segments, e.g., [(start_time, end_time), ...] for each detected profane word. The Docker image for this service is lightweight (just Python with our detection script).

- Audio Censorship Service (Cloud): This service applies the censorship by muting segments in the audio. Implemented in Python using pydub or a similar audio library, it expects as input a WAV file (the output from conversion, which contains the original audio content) and a list of segments to censor (from the profanity service). The service's endpoint (/censor) can accept these inputs (the audio file and the JSON list of segments). When triggered, the service loads the WAV audio (e.g., via AudioSegment in pydub) and for each segment, sets that portion's volume to zero (silence) or overlays a beep sound of the same length. After processing all listed segments, the service saves the modified audio (still as WAV for maximum quality before compression). The output WAV is either stored or passed along. The Docker container for this service includes the audio library and is based on Python or a similar environment capable of audio processing.
- Audio Compression Service (Cloud): The final microservice compresses the censored audio to reduce file size. We implemented this by converting the WAV to a compressed format (MP3 at a lower bitrate). Using ffmpeg again or pydub, the service takes the censored WAV and encodes it, for example, to an MP3 with a specified bitrate (e.g., 64 kbps) or sampling rate reduction. This significantly reduces the file size (for instance, from a full-quality WAV which might be large, to a compressed MP3 which is much smaller). The service's endpoint (/compress) takes the WAV (or a path to it) and returns the compressed file. The Docker image is similar to the conversion service (with ffmpeg support).

Each of these microservices was tested locally (on a development machine) using Docker Compose to ensure they worked in sequence. For example, we ran the TTS container with a sample text and checked that it produced an MP3 file; then manually fed that to the conversion container, and so on. After verification, the images were tagged and pushed to **Docker Hub**. This fulfills part of O5, making the images publicly accessible so that the Kubernetes clusters can pull them. The Docker images were versioned. All source code for the services and Dockerfile definitions were also published in a GitHub repository for transparency.

5.3 Implementation and Testing Strategy

Each microservice in the audio processing pipeline was developed and initially validated locally on a development device. This approach ensured that the individual functionalities were correct

before deploying them across the cloud–MEC–edge continuum. The validation process followed these stages:

1. Local Service Testing:
    - Text-to-Speech (TTS) Service: The TTS service was run locally on the laptop where it accepted JSON-formatted requests containing textual messages. For example, when provided with the message "This is a damn mess. I can't believe this stupid fucking system keeps screwing up—it's", logs demonstrated that the service successfully split the text into two sentences, generated temporary audio files for each sentence, concatenated them, and produced a final MP3 output. The log indicated that the final audio file size was 28,544 bytes, and the audio duration was 7.00 seconds.
    - Conversion Service: Locally, the Conversion service was tested by sending raw MP3 data to its endpoint. It converted the MP3 file to WAV format, verified by the increase in file size (from 28,544 bytes to 308,874 bytes) and consistent audio duration.
    - Profanity Detection Service: When tested with WAV audio input, the Profanity Detection service recognized and processed the audio, outputting a JSON response that indicated the presence of censored words by providing censorship indexes (e.g., [(0.4625, 0.7), (0.7125, 0.8)]). Temporary files were created and removed during processing, as shown in the logs.
    - Censorship Service: This service received the processed audio and censorship indexes, applied censorship adjustments (by calculating and zeroing out specific audio intervals), and logged both the original and adjusted sample ranges.
    - Compression Service: Lastly, the Compression service was validated by sending it the censored audio; it executed a compression routine (using pydub and FFmpeg under the hood), confirmed via log outputs that detailed the original and compressed file sizes and confirmed a 100% compression ratio.

    Figure 2 (Appendix) displays the communication of each services.
2. Validation on Docker Hub Images:
   After local testing, each microservice was containerized into Docker images and published to Docker Hub. These images were then pulled and re-tested on the local device to ensure that the published versions maintained the same functionality as the locally built images. Figure 3 (Appendix) confirms that the images are reliable and can be used in a distributed environment.
3. Deployment in the Continuum:
   Once the individual services were validated locally, they were deployed into Kubernetes clusters configured to represent the cloud and MEC (AWS Wavelength) layers. Screenshots (as attached, though not embedded in the report) confirmed that:
    - The **cloud cluster** hosted the Censorship and Compression services.

- The **MEC (Wavelength) cluster** hosted the Conversion and Profanity Detection services.
- The **edge (laptop)** continued to run the TTS service locally, interfacing with the deployed services via a shared MQTT broker.

This staged deployment verified that the pipeline's services are not only functionally correct in isolation but also connect properly within a distributed cloud-edge environment. Although full end-to-end testing across a live continuum was not achieved—due to network and time constraints—the local and staged integration tests confirm that the pipeline does work and that all services are correctly deployed and interconnected. Figure 4 and 5 (Appendix) displays the initialization of services given the layers.

6. Discussion

This section examines the broader implications of our work, identifies factors that may have affected the results, and discusses their limitations and generalizability.

6.1 Threats to Validity

Several factors could affect our results' quality:

- Test Environment vs. Real-world Conditions:
  Our controlled, single-user tests with limited concurrency may not reflect performance under high load or real 5G conditions. We mitigated this by introducing some concurrency (multiple service replicas) and multiple test messages; however, the scalability and latency under heavy load remain unverified.
- Connectivity Assumptions:
  Connectivity between the laptop and MEC was emulated via public internet rather than through a genuine 5G connection. Although AWS Wavelength documentation was followed to simulate realistic conditions, the absence of actual mobile edge connectivity remains a residual threat.
- Accuracy of Profanity Detection:
  Our text-based profanity detection approach is a simplified substitute for a full-fledged speech-to-text integration. This may lead to inaccuracies in real-world scenarios, although such risks are acceptable given our focus on system integration.
- Scenario Bias:
  By selecting a "happy path" for batch-oriented processing, our pipeline may not account for failures in applications requiring continuous streaming or extremely tight latency. Transparency about these choices mitigates, but does not eliminate, potential bias.

6.2 Implications of the Research Results

- For Research on Orchestration:
  The successful deployment of a multi-tier pipeline validates the cloud–edge model. Our findings highlight the challenges (e.g., network configuration, multi-cluster management) that justify the need for advanced orchestrators like ECO or Oakestra. Researchers may build on our work to improve automated network configuration and edge onboarding.

- For Future Edge Applications:
  Practitioners can follow our documented approach to deploy edge applications even without sophisticated orchestration, thus lowering the barrier to experimentation. Our use of AWS Wavelength and standard tools demonstrates that current cloud providers make edge computing accessible—albeit with careful architectural design.
- Conceptual Impact:
  The results support hybrid designs over pure cloud or edge approaches. By offloading time-sensitive processing (TTS) to the edge and heavier tasks to the cloud, our architecture offers a balanced model that could influence how future IoT and multimedia applications are designed.
- Broader Technological and Societal Impact:
  Although limited in scope, our work touches on emerging areas such as serverless computing and 5G MEC integration. The ability to deploy real-time content moderation across distributed systems could spur innovations in remote education, smart cities, and online communication platforms, enhancing user experience and supporting community standards.

6.3 Limitations of the Results

- Performance Evaluation:
  Objective O3 was not fully realized; thus, we lack comprehensive quantitative data on latency improvements and resource efficiency. This limits strong performance claims and suggests the need for more extensive monitoring in future work.
- Pipeline Specificity:
  Our pipeline architecture was tailored for this demonstration. The design choices—such as placing the audio conversion at MEC—were made for clarity rather than optimality. Other partitioning strategies might yield different results.
- Incomplete Orchestrator Integration:
  With objective O4 only partially addressed, our static deployment does not exhibit dynamic scaling or automatic fault tolerance. Advanced orchestrators could handle these aspects better, and our work serves as a stepping stone for those improvements.
- Security and Privacy:
  Critical features like encryption, authentication, and on-device speech processing were not implemented. These omissions, common in academic prototypes, must be addressed before any production deployment.
- Edge Device Representativeness:
  Using a laptop as the edge device does not capture the constraints of low-power IoT devices. Consequently, our results may not directly transfer to more limited hardware.

6.4 Generalizability of the Results

- Applicability to Other Applications:
  The architectural approach—partitioning tasks by latency sensitivity and using MQTT for integration—is broadly applicable to similar IoT and cloud-edge scenarios. However, for

applications with drastically different real-time or stateful requirements, adjustments will be needed.

- Platform and Infrastructure Independence:
  Although our implementation relies on AWS and a laptop, the design principles (containerization, orchestration, and event-driven architecture) are general enough to be applied on alternative platforms (e.g., Azure Edge Zones, Raspberry Pi-based clusters).
- Observational Generalizability**:**
  Findings such as the difficulty in network configuration and the heavy footprint of Kubernetes at the edge are likely to be common to similar projects. These lessons can guide both future research and industry practices in distributed system deployments.

In summary, while our results are robust within the defined scope, future work should address performance quantification, dynamic orchestration, and security concerns. The insights provided are broadly relevant and form a useful reference for both researchers and practitioners exploring cloud–edge integrations.

7. Conclusions

This thesis addressed the challenge of deploying a real-time audio processing application across the cloud–MEC–edge continuum. The primary objectives were to design (O1) and deploy (O2) a multi-tier microservice pipeline—handling tasks such as text-to-speech, audio conversion, profanity detection, censorship, and compression (see Sections 5.1 and 5.2)—using accessible tools like Docker and Kubernetes (Xiong et al., 2018; Alarbi et al., 2024).

Key results include a functional end-to-end system where TTS executed on the edge delivered immediate feedback, and subsequent MEC and cloud layers processed and refined the output. Notably, MQTT was used to enable robust inter-service communication despite inherent network challenges (Section 5.4). This work is novel in its practical deployment on AWS Wavelength and demonstrates how a concrete implementation can bridge the gap between theoretical orchestration frameworks and real-world applications.

In summary, the thesis demonstrates that a carefully designed microservice pipeline can effectively span from edge to cloud, delivering both functional and performance benefits for latency-sensitive applications. The insights gained not only validate current research trends but also provide a foundation for developing more flexible and automated edge orchestration frameworks in the future.

8. Future Work and Lessons Learned

Future Work:

- FW1: Detailed Performance Evaluation:
  Integrate monitoring tools (e.g., Prometheus, Grafana) to collect comprehensive metrics (scheduling latency, processing times, throughput) and compare distributed versus cloud-only deployments.
- FW2: Advanced Orchestration Integration:
  Explore dynamic orchestration frameworks (e.g., KubeEdge, ECO) to automate inter-

layer connectivity and enable dynamic scaling, thereby addressing the manual configuration limitations observed in our current setup.

- FW3: Real-World Network Testing:
  Deploy and evaluate the pipeline using actual 5G-enabled edge devices to validate performance under true mobile edge conditions and capture behaviors (e.g., handover effects) not observable in simulated environments.
- FW4: Enhancement of Service Features:
  Improve audio processing by integrating on-device speech recognition for more accurate profanity detection and refining censorship techniques to minimize audio distortion.

Lessons Learned:

- LL1: Automated Edge-Oriented Network Configuration:
  Our experience reinforces that precise, automated network configuration—especially across disparate environments (e.g., AWS regions and Wavelength zones)—is critical for achieving reliable cross-layer communication. This finding underscores a need for novel network configuration strategies in edge orchestration literature.
- LL2: Event-Driven Architectures in Multi-Tier Systems:
  Implementing an event-driven model using MQTT significantly improved the decoupling and resilience of service interactions. This approach is not only scalable but also presents a viable alternative to traditional HTTP-based methods, contributing a valuable perspective to distributed systems design.
- LL3: Importance of Open Standards for Portability:
  The adoption of standard file formats (MP3, WAV), protocols (HTTP, MQTT), and open-source tools greatly enhanced system portability and interoperability. This confirms and extends recent findings in the literature regarding the benefits of open standards in hybrid cloud–edge deployments.
- LL4: Reproducibility Through Infrastructure-as-Code:
  Leveraging Terraform and maintaining detailed system logs proved instrumental in achieving reproducibility and easing troubleshooting. This methodological rigor, while advocated in theory, has practical implications that reinforce best practices for future research in complex distributed systems.

These directions and insights aim not only to enhance the current work but also to inform future research and practical implementations in cloud–edge orchestration.

9. Acknowledgements

Firstly, I would like to express my deepest gratitude to Dr. Hanan Lutfiyya, my thesis supervisor, for providing an engaging and challenging topic at a time when I was struggling to find both a supervisor and a project direction. Her invaluable guidance, support, and insights have been instrumental in shaping this work.

I would also like to extend my sincere thanks to Andy Wong from Atlassian for his inspiring “aha” moment and practical advice on navigating cloud services, which greatly influenced the direction of this thesis.

My appreciation also goes to Muhamad Alarbi for his assistance during the onboarding process and for sharing his expertise in cloud–edge computing.

Lastly, I would like to thank Gabrille Jordan Niamat and Satvir Uppal for their continuous support and encouragement throughout this journey.

This thesis would not have been possible without the unwavering support and contributions of all the individuals mentioned above.

Appendix (Figures)

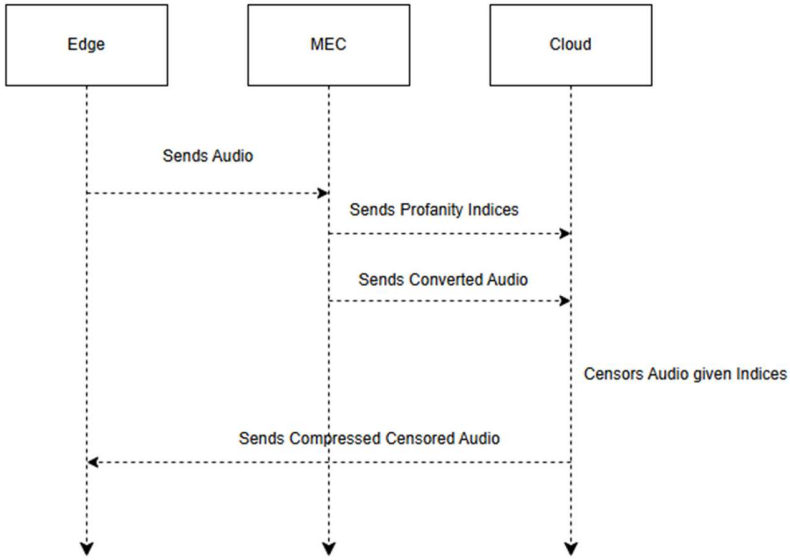

*Figure 1: Sequence diagram illustrating the end-to-end audio processing pipeline. The **Edge** node generates raw audio via Text-to-Speech (TTS) and sends it to the **MEC** for audio format conversion and profanity detection. The MEC service forwards profanity indices and converted audio to the **Cloud**, which then censors the audio based on those indices and performs compression. Finally, the censored, compressed audio is returned to the Edge.*

```
text2speech-1  | 2025-04-12 00:32:24,451 - INFO - Received request with content-type: application/json
text2speech-1  | 2025-04-12 00:32:24,451 - INFO - Processing message (length 87 characters): This is a damn mess. I can't believe this stupid fucking system keeps screwing up—it's
text2speech-1  | 2025-04-12 00:32:24,452 - INFO - Split text into 2 sentence(s).
text2speech-1  | 2025-04-12 00:32:24,452 - INFO - Processing sentence 1/2: This is a damn mess.
text2speech-1  | 2025-04-12 00:32:24,452 - INFO - Temporary file created for sentence 1: /tmp/tmp9slxghn0.mp3
text2speech-1  | 2025-04-12 00:32:24,516 - INFO - Sentence 1 audio file size: 4616 bytes
text2speech-1  | 2025-04-12 00:32:24,617 - INFO - Processing sentence 2/2: I can't believe this stupid fucking system keeps screwing up—it's
text2speech-1  | 2025-04-12 00:32:24,617 - INFO - Temporary file created for sentence 2: /tmp/tmp69mqh4bp.mp3
text2speech-1  | 2025-04-12 00:32:24,702 - INFO - Sentence 2 audio file size: 22692 bytes
text2speech-1  | 2025-04-12 00:32:24,862 - INFO - Text-to-Speech conversion successful; final audio size: 28544 bytes.
text2speech-1  | 2025-04-12 00:32:24,863 - INFO - Audio duration: 7.00 seconds.
text2speech-1  | 2025-04-12 00:32:24,863 - INFO - 172.19.0.1 - - [12/Apr/2025 00:32:24] "POST /process HTTP/1.1" 200 -
conversion-1   | 2025-04-12 00:32:24,870 - INFO - Input filesize: 28544 bytes
conversion-1   | 2025-04-12 00:32:24,968 - INFO - Output filesize: 308874 bytes
conversion-1   | 2025-04-12 00:32:24,968 - INFO - Audio duration: 7.00 seconds
conversion-1   | 2025-04-12 00:32:24,969 - INFO - 172.19.0.1 - - [12/Apr/2025 00:32:24] "POST /process HTTP/1.1" 200 -
profanity-1    | 2025-04-12 00:32:24,976 - INFO - Temporary file created: /tmp/tmp0fr5rnqi.wav
profanity-1    | 2025-04-12 00:32:24,976 - INFO - Audio file header: b'RIFF\x82\xb6\x04\x00WAVE'
profanity-1    | 2025-04-12 00:32:28,513 - INFO - Recognized text: this is a number I can't believe this stupid fucking system keeps screwing up its
profanity-1    | 2025-04-12 00:32:28,554 - INFO - Cleaned text: this is a number i can t believe this stupid fucking system keeps screwing up its
profanity-1    | 2025-04-12 00:32:28,555 - INFO - Censored text: this is a number i can t believe this **** **** system keeps **** up its
profanity-1    | 2025-04-12 00:32:28,555 - INFO - Censorship indexes: [(0.4691358024691358, 0.7037037037037037), (0.7160493827160493, 0.8024691358024691)]
profanity-1    | 2025-04-12 00:32:28,555 - INFO - Audio file duration: 7.00 seconds
profanity-1    | 2025-04-12 00:32:28,555 - INFO - Temporary file /tmp/tmp0fr5rnqi.wav removed.
profanity-1    | 2025-04-12 00:32:28,556 - INFO - 172.19.0.1 - - [12/Apr/2025 00:32:28] "POST /process HTTP/1.1" 200 -
censor-1       | 2025-04-12 00:32:28,577 - INFO - Input filesize: 308874 bytes
censor-1       | 2025-04-12 00:32:28,577 - INFO - Number of censorship intervals received: 2
censor-1       | 2025-04-12 00:32:28,578 - INFO - Total audio samples: 154415
censor-1       | 2025-04-12 00:32:28,578 - INFO - Original interval: [0.4691358024691358, 0.7037037037037037] -> Adjusted interval: (0.5395, 0.6333) which maps to samples (83307, 9779
6)
censor-1       | 2025-04-12 00:32:28,578 - INFO - Original interval: [0.7160493827160493, 0.8024691358024691] -> Adjusted interval: (0.7420, 0.7765) which maps to samples (114572, 119
909)
censor-1       | 2025-04-12 00:32:28,578 - INFO - Output filesize: 308874 bytes
censor-1       | 2025-04-12 00:32:28,578 - INFO - 172.19.0.1 - - [12/Apr/2025 00:32:28] "POST /process HTTP/1.1" 200 -
compression-1  | 2025-04-12 00:32:28,597 - INFO - Received request for audio compression
compression-1  | 2025-04-12 00:32:28,598 - INFO - Processing file: censored.wav
compression-1  | 2025-04-12 00:32:28,598 - INFO - Input filesize: 308874 bytes
compression-1  | 2025-04-12 00:32:28,598 - INFO - Starting audio compression
compression-1  | 2025-04-12 00:32:28,598 - INFO - Original audio duration: 7003 ms (7.00 seconds)
compression-1  | 2025-04-12 00:32:28,600 - INFO - Compressed audio size: 0 bytes
compression-1  | 2025-04-12 00:32:28,600 - INFO - Compression ratio: 100.00%
compression-1  | 2025-04-12 00:32:28,600 - INFO - Compressed audio duration: 7003 ms
compression-1  | 2025-04-12 00:32:28,600 - INFO - Audio compression completed successfully
compression-1  | 2025-04-12 00:32:28,601 - INFO - 172.19.0.1 - - [12/Apr/2025 00:32:28] "POST /process HTTP/1.1" 200 -
```

Figure 2: Detailed microservice log outputs from the full text-to-speech pipeline flow. These logs capture each stage, from the initial text input at the Edge (TTS creation) through audio format conversion and profanity analysis at the MEC, to final censorship and compression in the Cloud. Notable entries include the size and duration of audio files, detected profanity segments, and successful handoffs between services—verifying the integrated, multi-layer processing approach.

```
text2speech-1  | 2025-04-11 21:17:36,834 - INFO - Received request with content-type: application/json
text2speech-1  | 2025-04-11 21:17:36,835 - INFO - Processing message (length 87 characters): This is a damn mess. I can't believe this stupid fucking system keeps screwing up—it's
text2speech-1  | 2025-04-11 21:17:36,835 - INFO - Split text into 2 sentence(s).
text2speech-1  | 2025-04-11 21:17:36,835 - INFO - Processing sentence 1/2: This is a damn mess.
text2speech-1  | 2025-04-11 21:17:36,836 - INFO - Temporary file created for sentence 1: /tmp/tmpe5kqzox3.mp3
text2speech-1  | 2025-04-11 21:17:36,900 - INFO - Sentence 1 audio file size: 4616 bytes
text2speech-1  | 2025-04-11 21:17:36,994 - INFO - Temporary file /tmp/tmpe5kqzox3.mp3 removed.
text2speech-1  | 2025-04-11 21:17:36,994 - INFO - Processing sentence 2/2: I can't believe this stupid fucking system keeps screwing up—it's
text2speech-1  | 2025-04-11 21:17:36,994 - INFO - Temporary file created for sentence 2: /tmp/tmp1r758lxb.mp3
text2speech-1  | 2025-04-11 21:17:37,080 - INFO - Sentence 2 audio file size: 22692 bytes
text2speech-1  | 2025-04-11 21:17:37,174 - INFO - Temporary file /tmp/tmp1r758lxb.mp3 removed.
text2speech-1  | 2025-04-11 21:17:37,244 - INFO - Text-to-Speech conversion successful; final audio size: 28544 bytes.
text2speech-1  | 2025-04-11 21:17:37,244 - INFO - Audio duration: 7.00 seconds.
text2speech-1  | 2025-04-11 21:17:37,246 - INFO - 172.20.0.1 - - [11/Apr/2025 21:17:37] "POST /process HTTP/1.1" 200 -
conversion-1   | 2025-04-11 21:17:37,251 - INFO - Input filesize: 28544 bytes
conversion-1   | 2025-04-11 21:17:37,346 - INFO - Output filesize: 308874 bytes
conversion-1   | 2025-04-11 21:17:37,346 - INFO - Audio duration: 7.00 seconds
conversion-1   | 2025-04-11 21:17:37,347 - INFO - 172.20.0.1 - - [11/Apr/2025 21:17:37] "POST /process HTTP/1.1" 200 -
profanity-1    | 2025-04-11 21:17:37,354 - INFO - Temporary file created: /tmp/tmps80fkkyd.wav
profanity-1    | 2025-04-11 21:17:37,354 - INFO - Audio file header: b'RIFF\x82\xb6\x04\x00WAVE'
profanity-1    | 2025-04-11 21:17:40,769 - INFO - Recognized text: this is an Amex I can't believe this stupid fucking system keeps screwing up its
profanity-1    | 2025-04-11 21:17:40,808 - INFO - Cleaned text: this is an amex i can t believe this stupid fucking system keeps screwing up its
profanity-1    | 2025-04-11 21:17:40,808 - INFO - Censored text: this is an amex i can t believe this **** **** system keeps **** up its
profanity-1    | 2025-04-11 21:17:40,808 - INFO - Censorship indexes: [(0.4625, 0.7), (0.7125, 0.8)]
profanity-1    | 2025-04-11 21:17:40,808 - INFO - Audio file duration: 7.003 seconds
profanity-1    | 2025-04-11 21:17:40,809 - INFO - Temporary file /tmp/tmps80fkkyd.wav removed.
profanity-1    | 2025-04-11 21:17:40,809 - INFO - 172.20.0.1 - - [11/Apr/2025 21:17:40] "POST /process HTTP/1.1" 200 -
censor-1       | 2025-04-11 21:17:40,817 - INFO - Input filesize: 308874 bytes
censor-1       | 2025-04-11 21:17:40,817 - INFO - Number of censorship intervals received: 2
censor-1       | 2025-04-11 21:17:40,818 - INFO - Total audio samples: 154415
censor-1       | 2025-04-11 21:17:40,818 - INFO - Original interval: [0.4625, 0.7] -> Adjusted interval: (0.5338, 0.6288) which maps to samples (82419, 97088)
censor-1       | 2025-04-11 21:17:40,818 - INFO - Original interval: [0.7125, 0.8] -> Adjusted interval: (0.7388, 0.7738) which maps to samples (114074, 119478)
censor-1       | 2025-04-11 21:17:40,818 - INFO - Output filesize: 308874 bytes
censor-1       | 2025-04-11 21:17:40,818 - INFO - 172.20.0.1 - - [11/Apr/2025 21:17:40] "POST /process HTTP/1.1" 200 -
compression-1  | 2025-04-11 21:17:40,824 - INFO - Received request for audio compression
compression-1  | 2025-04-11 21:17:40,826 - INFO - Processing file: censored.wav
compression-1  | 2025-04-11 21:17:40,826 - INFO - Input filesize: 308874 bytes
compression-1  | 2025-04-11 21:17:40,826 - INFO - Starting audio compression
compression-1  | 2025-04-11 21:17:40,826 - INFO - Original audio duration: 7003 ms (7.00 seconds)
compression-1  | 2025-04-11 21:17:40,828 - INFO - Compressed audio size: 0 bytes
compression-1  | 2025-04-11 21:17:40,828 - INFO - Compression ratio: 100.00%
compression-1  | 2025-04-11 21:17:40,828 - INFO - Compressed audio duration: 7003 ms
compression-1  | 2025-04-11 21:17:40,828 - INFO - Audio compression completed successfully
compression-1  | 2025-04-11 21:17:40,829 - INFO - 172.20.0.1 - - [11/Apr/2025 21:17:40] "POST /process HTTP/1.1" 200 -
```

Figure 3: Verified microservice logs demonstrating the pipeline running from Docker images published to DockerHub. Each container pulls its required dependencies automatically, allowing the text-to-speech workflow to function even without installing libraries locally. The logs show successful audio generation, conversion, profanity detection, censorship, and compression, confirming that the published images perform all pipeline stages end-to-end.

```
[ec2-user@ip-10-0-10-228 ~]$ kubectl get nodes -n wavelength
NAME                                          STATUS   ROLES
 AGE   VERSION
ip-10-0-10-175.ca-central-1.compute.internal   Ready    <none>
 13m   v1.28.0
ip-10-0-10-228.ca-central-1.compute.internal   Ready    control-plane
 17m   v1.28.0
[ec2-user@ip-10-0-10-228 ~]$ kubectl get svc -n wavelength
NAME                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
        AGE
conversion-service   ClusterIP   10.101.94.236   <none>        5002/TCP
        14m
portainer            NodePort    10.106.75.138   <none>        9000:309
00/TCP   20s
profanity-service    ClusterIP   10.101.238.81   <none>        5003/TCP
        14m
[ec2-user@ip-10-0-10-228 ~]$ kubectl get pods -n wavelength
NAME                          READY   STATUS    RESTARTS   AGE
conversion-6df7dff58c-hpwp7   1/1     Running   0          14m
conversion-6df7dff58c-q2j8d   1/1     Running   0          14m
mosquitto-694dc86997-4bxqs    1/1     Running   0          14m
portainer-85d6b77f88-5c4sm    1/1     Running   0          32s
profanity-655bdbc744-6vc2z    1/1     Running   0          14m
profanity-655bdbc744-9qnsc    1/1     Running   0          14m
[ec2-user@ip-10-0-10-228 ~]$
```

```
[ec2-user@ip-10-0-10-175 ~]$ sudo kubeadm join 10.0.10.228:6443 --token
 11ceg0.lvtdj1yhd1j4s6th --discovery-token-ca-cert-hash sha256:8c642b46
fbac6d5ae90a9d78a9ff185e8e9badbf69ff8d23cc5c299ab5de984d
[preflight] Running pre-flight checks
        [WARNING FileExisting-tc]: tc not found in system path
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube
-system get cm kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet
/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/v
ar/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was
received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the
cluster.

[ec2-user@ip-10-0-10-175 ~]$
```

Figure 4: Verification of the AWS Wavelength Kubernetes setup. The left terminal shows a successful kubectl get nodes command (indicating the control-plane node is active) and running microservices (e.g., Conversion, Profanity, Censorship, Compression). On the right, kubeadm join output confirms the node has joined the cluster, demonstrating that the Wavelength instance is properly initialized and ready to run edge workloads.

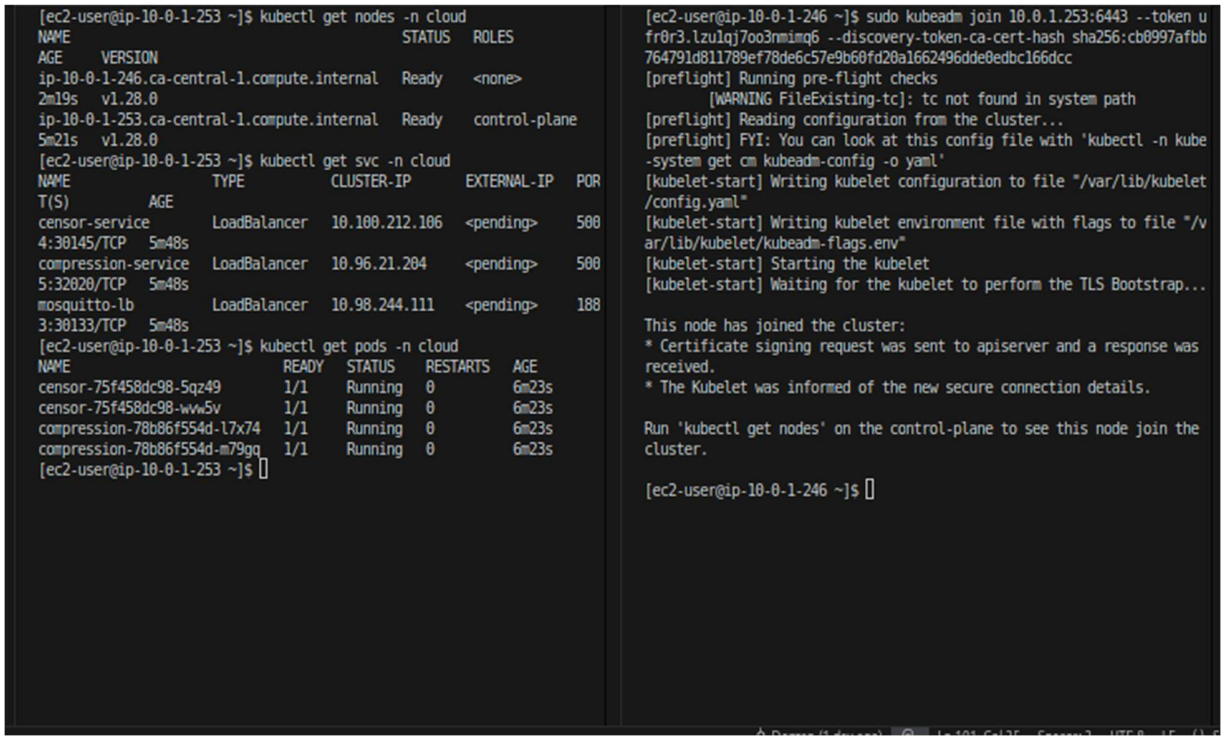

Figure 5: AWS region-based Kubernetes cluster for the "cloud" layer. The left terminal outputs confirm active nodes (kubectl get nodes) and running pods (kubectl get pods -n cloud)—including the Censorship and Compression services. On the right, the kubeadm join logs verify successful addition of the cloud node to the cluster, demonstrating a fully initialized and ready environment for cloud-tier workloads.

11. References

- Alarbi, M., Belson, R., & Lutfiyya, H. (2024). **ECO: Edge Continuum Orchestrator Framework for Managing Serverless Chains across the Cloud-Edge Spectrum.** *In 33rd International Conference on Computer Communications and Networks (ICCCN 2024)*, IEEE, 1–8.
- Bartolomeo, G., Yosofie, M., Bäurle, S., Haluszczynski, O., Mohan, N., & Ott, J. (2023). **Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing.** *In 2023 USENIX Annual Technical Conference (USENIX ATC '23)*, 215–231.
- Böhm, S., & Wirtz, G. (2022). **Cloud-Edge Orchestration for Smart Cities: A Review of Kubernetes-Based Orchestration Architectures.** *EAI Endorsed Transactions on Smart Cities*, 6(18), e2. DOI: 10.4108/eai.25-5-2022.1731197
- Jeffery, A., Howard, H., & Mortier, R. (2021). **Rearchitecting Kubernetes for the Edge.** *In Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '21)*, 7–12.
- Xiong, Y., Sun, Y., Xing, L., & Huang, Y. (2018). **Extend Cloud to Edge with KubeEdge.** *In 2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 373–377.