COMP90024: Cluster and Cloud Computing

Assignment 2 Report

Video Link: https://www.youtube.com/watch?v=XYiFBetXico
Gitlab Repo Link: https://gitlab.unimelb.edu.au/yeeshen/comp90024_team_81

Team 81

Full Name - Student ID

Adam McMillan - 1393533

Ryan Kuang - 1547320

Tim Shen - 1673715

Yili Liu - 883012

Yuting Cai – 1492060

May 21, 2025

1. Introduction

1.1. Background Information

Cloud computing has emerged as a transformative technology, offering scalable, on-demand resources that enable rapid deployment for large-scale data processing and storage. NeCTAR Research Cloud is an open-source cloud computing platform powered by OpenStack that provides flexibility and cost-efficiency compared to commercial providers like AWS or Azure [i]. OpenStack's modular architecture supports customised deployments, eliminating vendor lock-in and fostering a community-driven ecosystem ideal for academic research [ii].

The platform integrates Kubernetes for robust container orchestration, dynamically scaling containerised applications to manage fluctuating workloads. Fission, a serverless framework, enables event-driven, scalable data processing, supporting a decoupled architecture where data harvesting, processing, and storage operate independently for enhanced modularity and fault tolerance. ElasticSearch facilitates rapid indexing and full-text querying of datasets, delivering near-real-time analytics on unstructured social media data. A Jupyter Notebook front-end, interfaced via a ReSTful API, provides interactive visualisation of trends and insights, aligning with the project's analytical objectives.

1.2. Objective

This project aims to develop a big data analytics platform on the NeCTAR Research Cloud to process social media data from platforms like Mastodon, BlueSky, and Reddit, focusing on Australian-related content. The primary objective is to analyse public sentiment on Australian social media regarding the popularity of global versus local artists, using data harvested from Spotify's top listeners.

By leveraging the decoupled architecture, the system ensures resilience and efficiency in handling large social media datasets, with components scaling independently to meet demand. The platform delivers a scalable, fault-tolerant solution, deriving insights into Australian music preferences through advanced analytics and visualisation. This implementation demonstrates the efficacy of open-source cloud technologies in academic research settings, offering a cost-effective and flexible alternative to commercial cloud solutions.

1.3. Scenario Overview

This project focuses on analysing Australian social media discourse surrounding popular musical artists, particularly comparing local versus international acts. Leveraging Spotify's publicly available rankings of top artists by monthly listeners, we narrow our scope to the global top 25 and Australian top 10 artists. This dataset serves as a reference point for evaluating how public sentiment and attention on social media platforms correlate with global music popularity metrics.

By integrating data from Mastodon, and Reddit, the system tracks two primary indicators: the volume of social media discussions (post counts over time) and the sentiment expressed in those posts (positive, neutral, or negative trends). These are analysed in contrast to the artists' standing in the Spotify ranking, revealing which artists attract attention primarily for their musical achievements, and which are more prominent due to their online persona, media presence, or controversies.

This scenario is academically significant as it not only highlights differences in local versus global reception but also explores the digital visibility of Australian talent. By identifying artists who are widely discussed or well-received on domestic platforms but underrepresented globally, the system can provide insights into gaps in global recognition. This may support future research or initiatives aimed at promoting local artists, contributing to broader discussions about cultural export, national identity in media, and algorithmic influence on music discovery.

2. System Architecture and Design

2.1. System Design Overview

The system was architected following distributed system principles, with functionality modularized across distinct layers. Each service was designed for horizontal scalability and fault tolerance, ensuring that individual components can operate independently and recover gracefully from failures.

To optimize both performance and resource efficiency, services were configured to scale dynamically based on incoming traffic. Autoscaling mechanisms adjust the number of pods in response to ingress load, allowing the system to maintain responsiveness under high demand while minimizing resource usage during idle periods. This elasticity is particularly crucial for deployments in public cloud environments, where cost efficiency is a key concern. By enabling automatic scaling up and down, we ensure the system remains both resilient and economically sustainable.

2.2. Melbourne Research Cloud

The Melbourne Research Cloud (MRC) provides free, on-demand computing resources to researchers at the University of Melbourne and affiliated institutions. It offers functionality comparable to commercial cloud platforms such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform. With access to nearly 20,000 virtual cores and a variety of virtual machine configurations, MRC also supports specialist resources, including GPGPUs, private networking, load balancing, and DNS services. [ii]

By leveraging MRC, we avoided the overhead of provisioning hardware and configuring foundational infrastructure. This allowed us to focus directly on the design and implementation of our cloud-native Kubernetes environment and the development of scalable, resilient software architecture.

2.3. Cloud Deployment

The sentiment analysis system was deployed on Melbourne Research Cloud, utilising its Kubernetes-based infrastructure in order to support a scalable, cloud-native architecture. Deployment involved orchestrating multiple services and components, including data ingestion/queueing, a machine learning component, and visualisation tools for debugging and monitoring.

The deployment process began with setting up the data storage and processing infrastructure. A Redis instance was provisioned using Helm, with RedisInsight configured for monitoring and management. Elasticsearch was deployed also using Helm and served as the primary database for storing harvested data, while Kibana provided an interactive dashboard for visualising incoming data and monitoring ongoing sentiment analysis.

The machine learning component, which was powered by a HuggingFace model, was deployed in a persistent volume using Kubernetes Jobs. To support this, various scripts and analysis pipelines were applied using CronJobs and ConfigMaps, enabling scheduled processing of social media data from Reddit and Mastodon. The sentiment analysis jobs pushed results to the configured Elasticsearch indexes.

The FastAPI backend was containerised using Docker and deployed to the Kubernetes cluster by using Helm. The API offered a variety of endpoints that allowed for querying of mention counts and sentiment data. Fission functions were used for serverless task execution based on event-triggers, and KEDA was integrated to dynamically scale workloads depending on queue lengths in Redis.

Monitoring and logging of the application were handled through Prometheus and Grafana, with the Blackbox Exporter configured to probe and observe service health.

The following figure 1 provides a visualisation of the resources utilised in MRC:

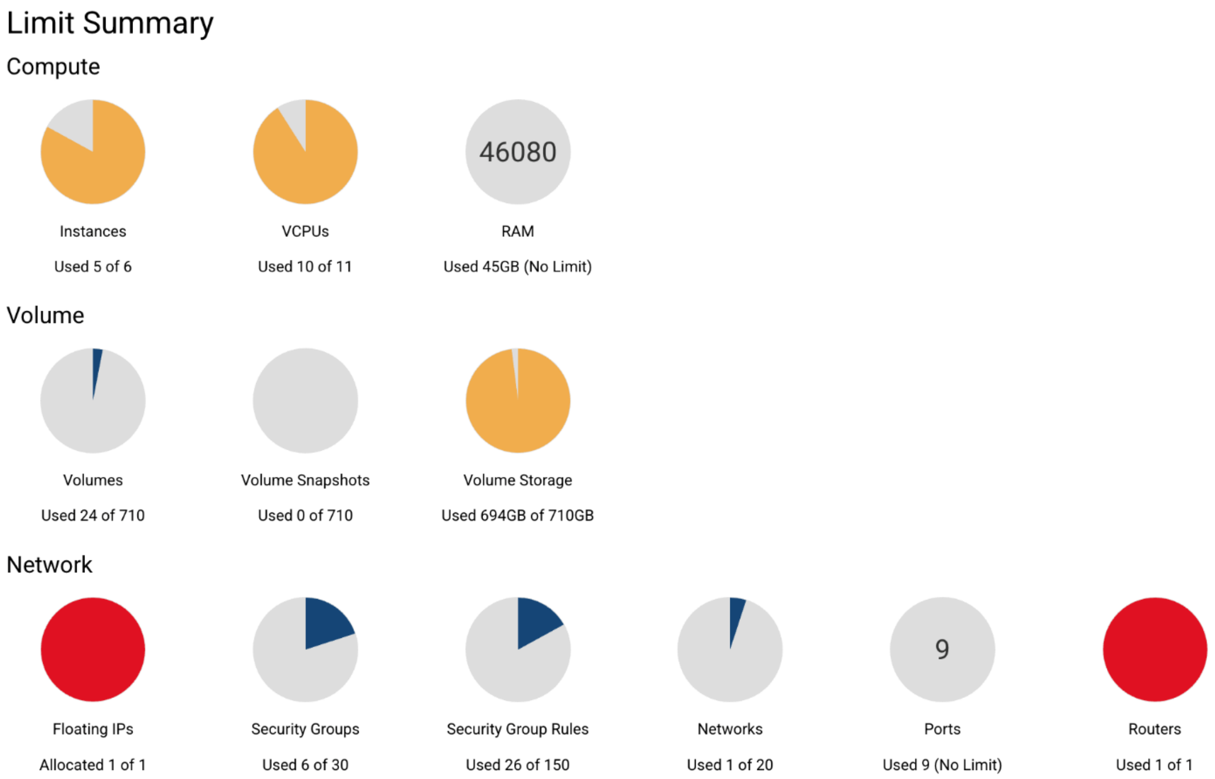


Figure 1. Overview of resource utilization in MRC (Additional Node is utilized.)

2.4. Advantage and Disadvantages of MRC

Advantages

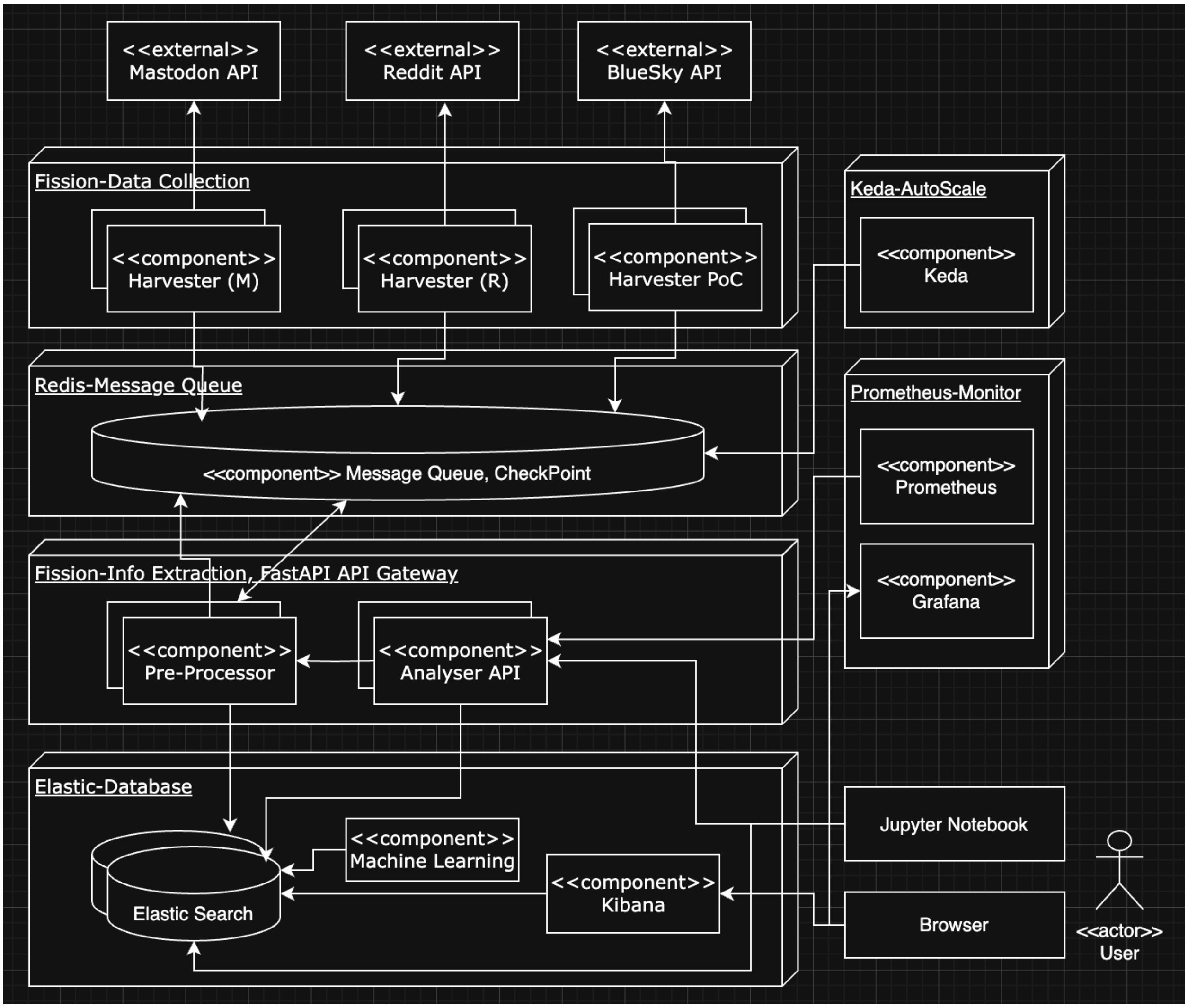
- **Full Infrastructure Control with Low Cost Overhead** - Users have full control over virtual machines, networking, and storage. This level of access allows for complete customization and is ideal for those with a solid understanding of cloud infrastructure—without the high usage costs of commercial platforms.
- **Academic-Driven Customization** - MRC is purpose-built for research and academic use. It can be tailored to meet the specific computational and data needs of researchers across disciplines.
- **Data Security and Compliance** - All data is hosted within the University of Melbourne's data center. This ensures data sovereignty, reduces exposure to third-party risk, and supports compliance with institutional and research data policies.

Disadvantages

- **Limited Availability and Scalability** - MRC operates within a single availability zone, which limits its suitability for applications requiring high availability, geographic redundancy, or global-scale deployment.

- **Operational and Maintenance Requirements** - As a university-managed infrastructure, ongoing hardware maintenance and upgrades are necessary. This introduces operational overhead.
- **Requires Infrastructure Knowledge** - MRC functions as an Infrastructure as a Service (IaaS). Users are responsible for configuring and managing their environments, which requires familiarity with system administration, networking, and cloud concepts. Unlike Platform as a Service (PaaS) solutions, MRC does not offer pre-configured runtime environments or serverless capabilities.

3. Application Stack



<<external>> Mastodon API
<<external>> Reddit API
<<external>> BlueSky API
Fission-Data Collection
<<component>> Harvester (M)
<<component>> Harvester (R)
<<component>> Harvester PoC
Keda-AutoScale
<<component>> Keda
Redis-Message Queue
<<component>> Message Queue, CheckPoint
Prometheus-Monitor
<<component>> Prometheus
<<component>> Grafana
Fission-Info Extraction, FastAPI API Gateway
<<component>> Pre-Processor
<<component>> Analyser API
Elastic-Database
<<component>> Machine Learning
Elastic Search
<<component>> Kibana
Jupyter Notebook
Browser
<<actor>> User

Figure 2. High Level Architecture

3.1. Backend

3.1.1. Fission – Data Collection, Message Queue, Info Extraction Tiers

Our backend system was built upon Fission, which is a serverless framework for Kubernetes. Fission allowed us to deploy backend logic as individual functions that could be triggered by events such as timers and triggers, as well as by HTTP requests where applicable. This modular approach helped us build a scalable and flexible harvesting and processing pipeline with clearly defined stages and clean interfaces.

Fission functions formed the core of our backend pipeline, each responsible for a specific stage in the data workflow. The function architecture can be broadly grouped into the following components:

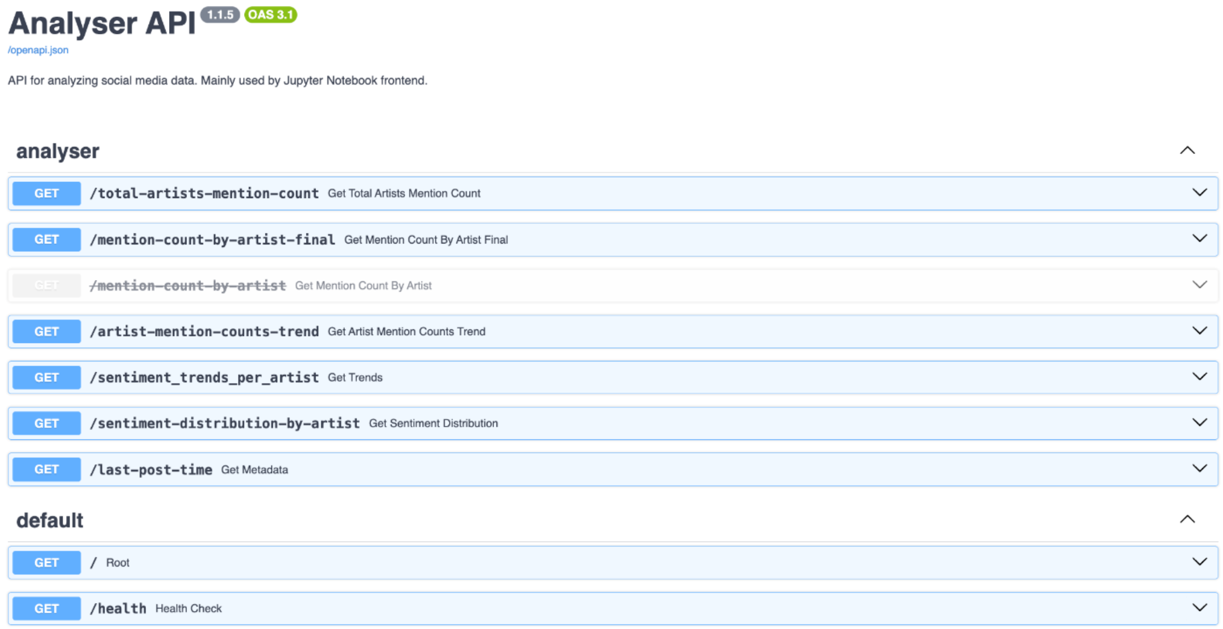
- **Harvesters:** Python-based functions that collect data from Reddit and Mastodon. These were deployed with timer triggers to run at regular intervals (e.g., every 5 seconds), ensuring continuous data collection. Each harvested item was pushed into a Redis queue for further processing.
- **Preprocessors:** Functions that pull items from the Redis queue and index them into Elasticsearch. We implemented a mix of timer vs trigger driven approaches for Mastodon and Reddit preprocessors respectively.
- **Analysis Functions:**
    - A sentiment analysis function enriched posts with sentiment scores. (see Machine Learning section for more implementation details)
- A keyword digger function searched for configured keywords in post content and stored matched results in dedicated Elastic indexes
- **API Function:** An HTTP-triggered function exposed a minimal API for querying backend data. This used Fission's route mechanism to map URLs to function calls.

All Fission components – functions, routes, timers, triggers, packages – were deployed with the assistance of spec files, in order to facilitate idempotency and simplify our CI/CD pipeline configuration.

3.1.2 Analyzer Restful API

Our API backend was developed using FastAPI, a modern, high-performance web framework that supports asynchronous programming and automatic generation of interactive API documentation. FastAPI natively supports OpenAPI, which allowed us to produce real-time, auto-generated documentation in HTML. This feature significantly improved the clarity and usability of our API interface.

By sharing the OpenAPI specification early, team members were able to work in parallel—frontend developers could begin integrating with the documented endpoints while backend developers focused on implementation. This parallel workflow accelerated development and enhanced crossteam communication. The consistent API specification also served as a reliable contract between components, reducing integration errors and improving productivity. Below is a screenshot demonstrating a portion of the auto-generated API documentation.



Analyser API 1.1.5 OAS 3.1

/openapi.json

API for analyzing social media data. Mainly used by Jupyter Notebook frontend.

analyser

GET `/total-artists-mention-count` Get Total Artists Mention Count

GET `/mention-count-by-artist-final` Get Mention Count By Artist Final

GET ~~`/mention-count-by-artist`~~ Get Mention Count By Artist

GET `/artist-mention-counts-trend` Get Artist Mention Counts Trend

GET `/sentiment_trends_per_artist` Get Trends

GET `/sentiment-distribution-by-artist` Get Sentiment Distribution

GET `/last-post-time` Get Metadata

default

GET `/` Root

GET `/health` Health Check

Figure 3. Analyser API Documentation

3.1.3. Elastic Search – Database Tier

We utilized Elasticsearch as the primary data store for all collected content from Mastodon and Reddit. The data pipeline was designed such that the Fission Preprocessing Tier handled data normalization and transformation before storing it into Elasticsearch via the official Elasticsearch client.

To ensure scalability, availability, and fault tolerance, we configured Elasticsearch with two primary shards and one replica per shard, distributed across three Elasticsearch nodes. This setup supports quorum-based consensus, improves read/write reliability, and enables horizontal scaling to accommodate growing data volumes from the preprocessing pods.

For data exploration and visualization, we used Kibana, which provided powerful tools to query, analyze, and visualize social media trends. Using Elasticsearch’s Domain-Specific Language (DSL), we constructed and tested queries within Kibana’s DevTools to extract insights and identify emerging topics from the ingested datasets.

3.2. Frontend

The Jupyter Notebook connects to several API endpoints, some were designed by our team, such as "mention-count-by-artist-final", while others were provided by ElasticSearch, such as "_search".

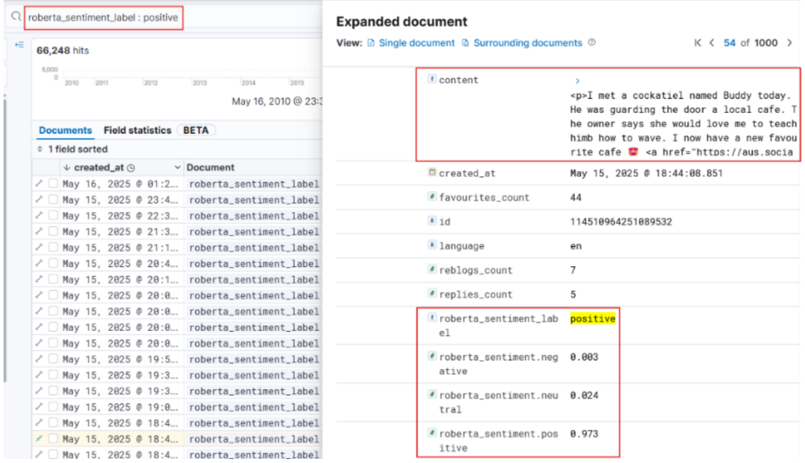
Whether to design customised API depends on the specific data we want to query. Data are processed, aggregated, and visualised interactively to help interpret the scenarios.

3.3. Machine Learning Sentiment Analysis Model

3.3.1. ML Model Description

In this project, a machine learning model is used to analyse content and assigns sentiment labels and scores across three categories: negative, neutral, and positive. The model, **cardiffnlp/twitter-roberta-base-sentiment-latest**, is hosted on **HuggingFace** and built on the **RoBERTa** architecture—an optimized version of BERT (Bidirectional Encoder Representations from Transformers).

Specifically fine-tuned for sentiment analysis on Twitter data, so this model excels at analyzing social media content. Beyond simply classifying post content into different sentiment categories, it assigns precise confidence scores to each sentiment class. As illustrated in Figure 4, a positive statement might receive tags such as "positive" along with numerical confidence values: "positive-score: 0.973", "neutral-score: 0.024", and "negative-score: 0.003". This detailed scoring system makes the model particularly valuable for accurately analyzing sentiment patterns in content from Mastodon and Reddit throughout this project.



roberta_sentiment_label : positive

66,248 hits

created_at	Document
May 16, 2025 @ 01:2...	roberta_sentiment_label
May 15, 2025 @ 23:4...	roberta_sentiment_label
May 15, 2025 @ 22:3...	roberta_sentiment_label
May 15, 2025 @ 21:3...	roberta_sentiment_label
May 15, 2025 @ 21:1...	roberta_sentiment_label
May 15, 2025 @ 20:4...	roberta_sentiment_label
May 15, 2025 @ 20:1...	roberta_sentiment_label
May 15, 2025 @ 20:0...	roberta_sentiment_label
May 15, 2025 @ 20:0...	roberta_sentiment_label
May 15, 2025 @ 20:0...	roberta_sentiment_label
May 15, 2025 @ 20:0...	roberta_sentiment_label
May 15, 2025 @ 19:5...	roberta_sentiment_label
May 15, 2025 @ 19:3...	roberta_sentiment_label
May 15, 2025 @ 19:3...	roberta_sentiment_label
May 15, 2025 @ 19:0...	roberta_sentiment_label
May 15, 2025 @ 18:4...	roberta_sentiment_label
May 15, 2025 @ 18:4...	roberta_sentiment_label
May 15, 2025 @ 18:4...	roberta_sentiment_label

content	<p>I met a cockatiel named Buddy today. He was guarding the door a local cafe. The owner says she would love me to teach himb how to wave. I now have a new favourite cafe 😻 <a href="https://aus.socia
created_at	May 15, 2025 @ 18:44:08.851
favourites_count	44
id	114510964251089532
language	en
reblogs_count	7
replies_count	5
roberta_sentiment_label	positive
roberta_sentiment.negative	0.003
roberta_sentiment.neutral	0.024
roberta_sentiment.positive	0.973

Figure 4. Example of Sentiment Label and Score Results in Kibana

3.3.2. Sentiment Analysis System Design and Structure

There are two sentiment analysis systems that are designed to analyse sentiment score depending on different amount of data in each index. Some indices contain a large volume of data—over 400,000 records. However, other indices contain significantly fewer records, typically around 10,000 or less. To address CPU usage limitations and optimize ML model performance, two distinct deployment architectures have been designed to accommodate these varying data workloads.

Figure 5 illustrates a system that leverages Kubernetes for orchestration and parallel processing to efficiently handle large volumes of data. This system is specifically designed to process a single

Elasticsearch index—in this case, "reddit-comments-prod"—containing numerous text entries. On the other hand, Figure 6 presents the second sentiment analysis system, which adopts a different architectural approach. Instead of parallel processing a single large index, it performs sequential processing across multiple smaller indices.

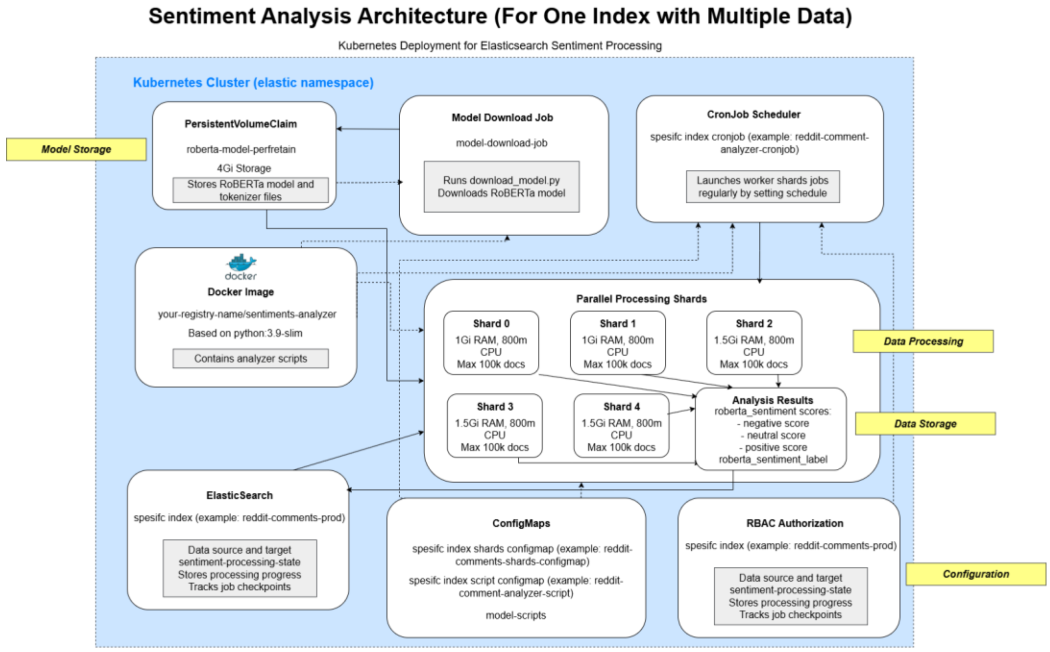

Sentiment Analysis Architecture (For One Index with Multiple Data)
Kubernetes Deployment for Elasticsearch Sentiment Processing
Kubernetes Cluster (elastic namespace)
Model Storage
PersistentVolumeClaim
roberta-model-perfretain
4Gi Storage
Stores RoBERTa model and tokenizer files
Model Download Job
model-download-job
Runs download_model.py Downloads RoBERTa model
CronJob Scheduler
spesifc index cronjob (example: reddit-comment-analyzer-cronjob)
Launches worker shards jobs regularly by setting schedule
docker
Docker Image
your-registry-name/sentiments-analyzer
Based on python:3.9-slim
Contains analyzer scripts
Parallel Processing Shards
Shard 0
1Gi RAM, 800m CPU
Max 100k docs
Shard 1
1Gi RAM, 800m CPU
Max 100k docs
Shard 2
1.5Gi RAM, 800m CPU
Max 100k docs
Shard 3
1.5Gi RAM, 800m CPU
Max 100k docs
Shard 4
1.5Gi RAM, 800m CPU
Max 100k docs
Analysis Results
roberta_sentiment scores:
- negative score
- neutral score
- positive score
roberta_sentiment_label
Data Processing
Data Storage
ElasticSearch
spesifc index (example: reddit-comments-prod)
Data source and target
sentiment-processing-state
Stores processing progress
Tracks job checkpoints
ConfigMaps
spesifc index shards configmap (example: reddit-comments-shards-configmap)
spesifc index script configmap (example: reddit-comment-analyzer-script)
model-scripts
RBAC Authorization
spesifc index (example: reddit-comments-prod)
Data source and target
sentiment-processing-state
Stores processing progress
Tracks job checkpoints
Configuration

Figure 5. System leveraging Kubernetes for orchestration and parallel processing of a single Elasticsearch index with large volumes of text data

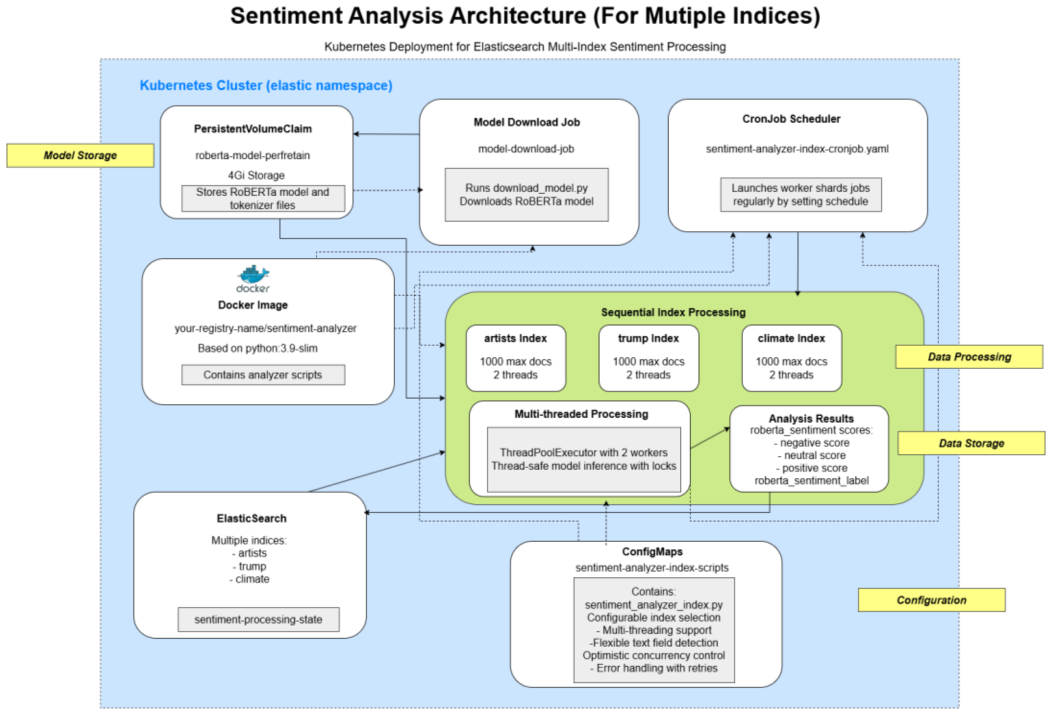

Sentiment Analysis Architecture (For Mutiple Indices)
Kubernetes Deployment for Elasticsearch Multi-Index Sentiment Processing
Kubernetes Cluster (elastic namespace)
Model Storage
PersistentVolumeClaim
roberta-model-perfretain
4Gi Storage
Stores RoBERTa model and tokenizer files
Model Download Job
model-download-job
Runs download_model.py Downloads RoBERTa model
CronJob Scheduler
sentiment-analyzer-index-cronjob.yaml
Launches worker shards jobs regularly by setting schedule
docker
Docker Image
your-registry-name/sentiment-analyzer
Based on python:3.9-slim
Contains analyzer scripts
Sequential Index Processing
artists Index
1000 max docs
2 threads
trump Index
1000 max docs
2 threads
climate Index
1000 max docs
2 threads
Multi-threaded Processing
ThreadPoolExecutor with 2 workers
Thread-safe model inference with locks
Analysis Results
roberta_sentiment scores:
- negative score
- neutral score
- positive score
roberta_sentiment_label
Data Processing
Data Storage
ElasticSearch
Multiple indices:
- artists
- trump
- climate
sentiment-processing-state
ConfigMaps
sentiment-analyzer-index-scripts
Contains:
sentiment_analyzer_index.py
Configurable index selection
- Multi-threading support
-Flexible text field detection
Optimistic concurrency control
- Error handling with retries
Configuration

Figure 6. System Architecture using Sequential Processing across Multiple Elasticsearch Indices

3.3.3. Model Storage Layer

The roberta-model-perfretain **PersistentVolumeClaim (PVC)** serves as the central storage hub for the sentiment analysis model. Allocated with 4Gi of storage, this Kubernetes resource creates a persistent storage space that remains intact regardless of pod lifecycle events. This persistence is important, as the RoBERTa model and its associated tokenizer files are sizable and inefficient to download repeatedly. Also, the PVC uses the "perfretain" storage class, which offers performance-optimized storage with data retention capabilities — ensuring fast model access while maintaining data durability across system restarts or failures. All worker processes across the sentiment analysis system share this model storage, preventing duplication and ensuring consistent analysis results.

Moreover, the **model-download-job** is a Kubernetes Job designed to execute once, setting up the environment for sentiment analysis model. It connects to the HuggingFace model repository and downloads the pre-trained "cardiffnlp/twitter-roberta-base-sentiment-latest" model. Once downloaded, the model is saved directly in the PVC, making it accessible to all worker pods. By decoupling model acquisition from the processing logic, this design allows the system to update or replace the model independently, providing flexibility for upgrades without disrupting the overall workflow.

3.3.4. Data Processing Layer

The two sentiment analysis systems employ different architectures during the data processing phase. The first system, designed to analyze indices with large volumes of data, systematically creates five separate worker shard jobs using YAML configurations stored in **ConfigMaps** (example in figure is reddit-comment-shards-configmap). This level of orchestration creates a reliable processing pipeline that runs without manual intervention, making it suitable for continuously processing new data dynamically stored in Elasticsearch. It leverages parallel processing to efficiently handle the high data volume.

The system employs **five worker shards** (numbered 0-4), each operating independently while coordinating through Elasticsearch to divide the workload. Documents are dynamically assigned to shards based on their position in the result set, using a simple modulo operation (position % total_shards == shard_index). To account for potential processing imbalances, later shards (2–4) are allocated more memory (1.5 Gi compared to 1 Gi for earlier shards). Each shard is configured to process up to 100,000 documents per run, with a batch size of 5,000 to manage memory usage and ensure reliable progress tracking. The shards use scroll context management when querying Elasticsearch, maintaining efficient, stateful queries that can retrieve large volumes of documents without overwhelming the database.

Unlike the previous architecture, which deployed multiple separate worker pods, the second system uses a single container that sequentially processes multiple indices. It adopts a configuration-driven approach, allowing the same script to be reused across different indices without modification. This design enhances adaptability, making it easy to add additional indices as needed. Furthermore, instead of relying on pod-level parallelism processing, this system implements thread-level parallelism within a single process. It uses Python's **ThreadPoolExecutor** to manage concurrent document processing while efficiently sharing system resources.

3.3.5. Data Storage Layer

The sentiment analysis process produces structured results that enhance each original document with sentiment-related metadata. For every processed document, the system generates two key fields: "**roberta_sentiment**"—an object containing three floating-point values representing the negative, neutral, and positive sentiment scores—and "**roberta_sentiment_label**"—a string indicating the overall sentiment classification of the post content. These results are written back to the original documents in Elasticsearch, preserving the original content while adding sentiment insights.

To ensure efficiency, the system uses **Elasticsearch's update API** with **bulk operations**, allowing large volumes of documents to be modified with minimal overhead. It also utilizes the **Elasticsearch's scroll API** for efficient pagination through large datasets, enabling the processing of millions of documents without excessive memory consumption.

3.3.6. Configuration Layer

Three distinct ConfigMaps are created for the first system, externalizing all configuration and code from the container images. This design results in a highly flexible and maintainable architecture:

1. The "**reddit-comment-shards-configmap**" contains the complete YAML definitions for all five worker shard jobs. Storing these as ConfigMaps, rather than hardcoding them, allows job parameters to be modified without rebuilding container images.
2. The "**reddit-comment-analyzer-script**" Holds the Python script responsible for performing sentiment analysis.
3. The "**model-scripts**" ConfigMap Includes the Python script used for downloading and preparing the RoBERTa model. By isolating this functionality, the system can update model acquisition logic independently of the analysis logic.

In addition, the system that processes large indices implements a precise security model using **Kubernetes Role-Based Access Control (RBAC)**. The "sentiment-job-creator" Role grants specific permissions—create, get, list, watch, update, patch, and delete—for Job resources within the "elastic" namespace. This allows the CronJob to dynamically create and manage worker jobs as needed.

In contrast, the second sentiment analysis model does not require RBAC. However, it does use a ConfigMap containing the sentiment_analyzer_index.py script, which defines its sentiment analysis logic.

3.3.7. Docker Image

The system also utilizes a purpose-built **Docker image** ("yucai5/sentiment-analyzer:lightweight")—which provides the execution environment for all processing components. This image plays a critical backup role within the machine learning model architecture. It not only includes all necessary dependencies for model execution but also embeds the essential scripts for model downloading and sentiment analysis directly within the image. This comprehensive packaging strategy ensures robustness by preventing failures caused by missing or incompatible external configuration files. Even if external ConfigMaps encounter issues, the processing pipeline remains operational and reliable.

3.3.8. Resource Efficiency

The model download job operates with moderate resource requirements—between 500m and 1 CPU, 1 to 2 Gi of memory, and 2 to 4 Gi of ephemeral storage. These settings reflect its primarily **I/O-bound nature** during the download phase and the memory-intensive demands of model loading. In contrast, processing jobs use more granular resource configurations. CPU allocations range from 200m to 800m, memory from 512 Mi to 1.5 Gi, and ephemeral storage from 5 to 10 Gi. These resources are strategically distributed across shards, with higher limits assigned to those expected to handle more complex or larger volumes of data.

Additionally, the second architecture reuses the same process and loaded model across multiple indices, eliminating redundant model loading between runs. This approach reduces both memory usage and processing time. The system also limits the thread count to two, based on available CPUs in the cluster, to avoid resource contention and ensure stable performance.

3.3.9. Performance Optimizations

A key optimization in the sentiment analysis model is its parallel processing design, which distributes the analytical workload across five independent processing shards. Each shard runs as a separate Kubernetes Job, with its own isolated processing logic, memory space, and execution flow.

In addition, the system leverages batch processing and the Elasticsearch Scroll API to further enhance performance. These optimizations are applied both during document retrieval and result persistence. Sentiment analysis results are accumulated in batches before being written back to Elasticsearch using the **Bulk API**, significantly reducing network overhead by consolidating multiple updates into a single HTTP request. This design achieves an optimal balance between processing speed and infrastructure load, enabling the system to continuously process large volumes of documents without destabilizing the underlying Elasticsearch service.

3.3.10. Error Handling

The single-index model architecture first attempts to load the sentiment analysis model from persistent storage. If this fails, it automatically falls back to downloading the model directly from

Hugging Face. The system also includes a **checkpointing mechanism** that regularly saves processing state, allowing recovery in the event of a failure. Additionally, it features a **scroll context recovery** mechanism: if the Elasticsearch scroll context is lost—a common issue in long-running queries—the system automatically reinitializes the search to continue processing.

In contrast, the multi-index architecture employs multi-threading with thread synchronization to prevent concurrency issues. It leverages **Elasticsearch's optimistic concurrency control** to resolve potential write conflicts when multiple threads attempt to update the same document. Unlike the single-index model, the multi-index architecture introduces a more comprehensive retry strategy. It retries the entire batch processing with **exponential backoff**, waiting $2^{retry\ count}$ seconds between attempts, specifically to handle bulk update failures. In the case of catastrophic failures, the entire processing attempt can be retried up to three times, also using exponential backoff, ensuring resilience and robustness in high-load or unstable environments.

Overall, the Elasticsearch-based sentiment analysis system demonstrates two distinct architectural approaches, each tailored to different workload patterns. By implementing these complementary designs, the system achieves optimal resource utilization and performance across various sentiment analysis tasks. Whether processing a large volume of data within a single index or analyzing sentiment across multiple specialized datasets—such as indices related to artists, trump, or climate-related topics—the system can be efficiently scaled and adapted to meet the specific demands of each use case.

3.4. Technical Features and Optimizations

3.4.1. KEDA (Kubernetes Event Driven Autoscaling) - Auto Scaling

To achieve scalable and responsive system behavior, we integrated Fission and KEDA into our architecture. Fission was used to manage the data collection tier, efficiently handling large volumes of HTTP requests. It automatically scales out function pods based on incoming traffic, ensuring the system remains responsive under load.

For components outside Fission's management scope—such as Redis—we utilized KEDA. KEDA continuously monitors the Redis queue length, and when the queue size exceeds a threshold of messages, it dynamically scales the Redis deployment up to a maximum of five instances. This adaptive scaling allows the system to handle increased throughput effectively, ensuring that message processing remains timely and reliable.

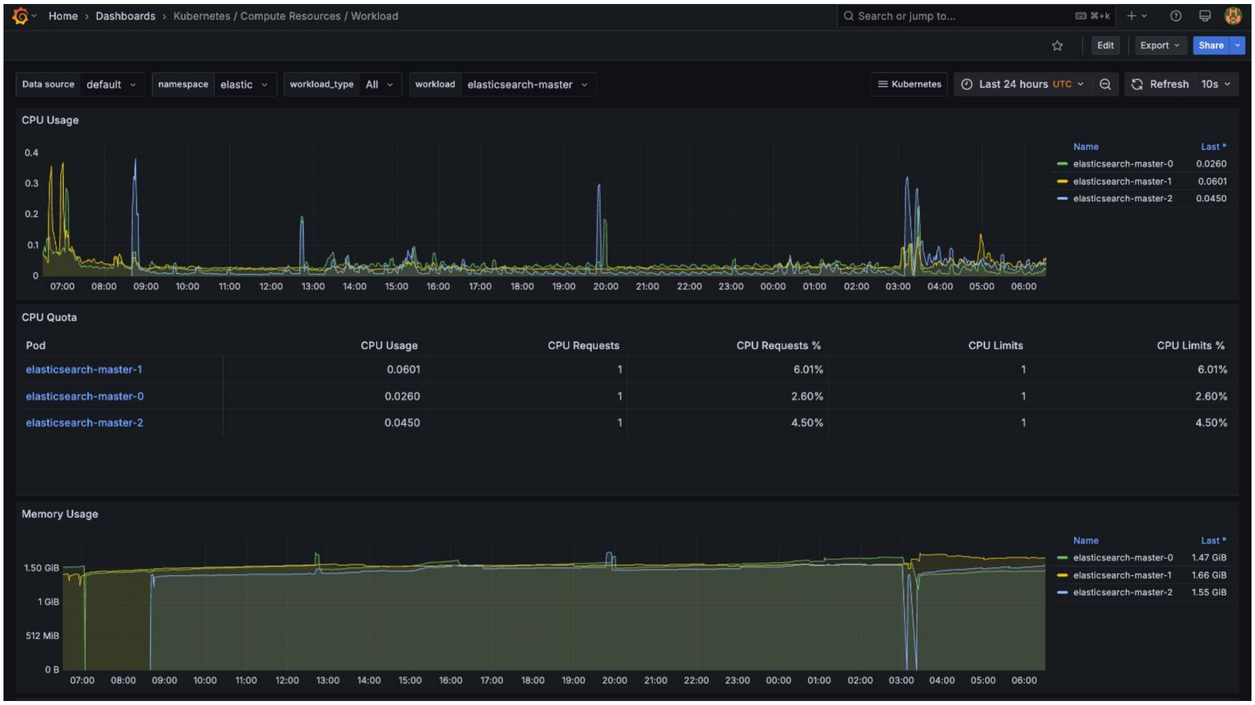
By leveraging both Fission and KEDA, we achieved a robust autoscaling mechanism that enhances performance while maintaining system stability under varying loads.

3.4.2. Prometheus - Observability and Performance Monitoring

To ensure robust observability and operational reliability, we deployed Prometheus and Grafana for comprehensive resource monitoring and health tracking. Prometheus aggregates metrics from various system components, while Grafana renders these metrics into visual dashboards that present real-time trends in CPU, memory, and disk usage across all pods and nodes.

These dashboards provide deep visibility into system behavior, enabling the early detection of performance bottlenecks, service overloads, and potential resource leaks. This observability framework not only facilitates prompt issue resolution but also informs ongoing architectural evaluation and refinement.

For example, during one monitoring session, we observed a significant spike in data ingestion that led to an Elasticsearch pod failure. Based on these insights, we provisioned an additional Kubernetes node via OpenStack to accommodate the load and improve overall system resilience. This incident underscores the critical role of observability tools in guiding proactive system scaling and design optimization.



Data source: default | namespace: elastic | workload_type: All | workload: elasticsearch-master | Last 24 hours UTC | Refresh 10s

CPU Usage

Name	Last *
elasticsearch-master-0	0.0260
elasticsearch-master-1	0.0601
elasticsearch-master-2	0.0450

CPU Quota

Pod	CPU Usage	CPU Requests	CPU Requests %	CPU Limits	CPU Limits %
elasticsearch-master-1	0.0601	1	6.01%	1	6.01%
elasticsearch-master-0	0.0260	1	2.60%	1	2.60%
elasticsearch-master-2	0.0450	1	4.50%	1	4.50%

Memory Usage

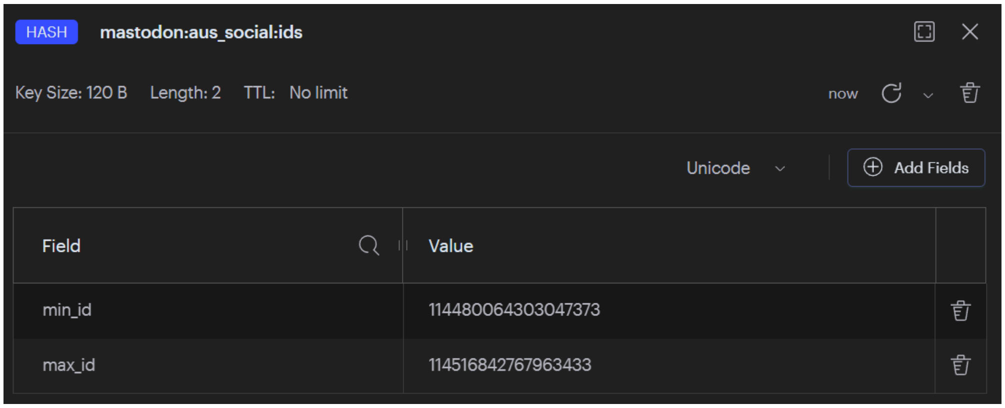
Name	Last *
elasticsearch-master-0	1.47 GiB
elasticsearch-master-1	1.66 GiB
elasticsearch-master-2	1.55 GiB

Figure 7. Kubernetes Resource Observability

4. Data Collection

4.1. Mastodon

We selected Mastodon as our primary data source, leveraging its accessible and well-documented API to facilitate programmatic data collection. Public posts ("toots") can typically be retrieved without API key authentication, simplifying deployment and enhancing system scalability. We targeted prominent Australian instances, such as Mastodon.au and aus.social, which provide concentrated sources of Australian perspectives on current events, news, and societal discourse.



mastodon:aus_social:ids

Key Size: 120 B Length: 2 TTL: No limit

Field	Value
min_id	114480064303047373
max_id	114516842767963433

Figure 8. Checkpoint Design by Utilizing Redis

Our harvester collects public posts from targeted Australian Mastodon servers using the Mastodon.py library's timeline functions. Since the API does not support direct retrieval of posts from a specific time, we implemented a Redis-based checkpoint mechanism to track the harvesting state across multiple Mastodon instances. For each server endpoint, we use a Redis hash to store min_id and max_id parameters, which track the oldest and newest harvested post identifiers. These serve as bidirectional pagination cursors, enabling efficient traversal of the content timeline in our collection pipeline. This state management ensures that our harvester processes retrieve contiguous blocks of posts without gaps. Each successful batch retrieval triggers immediate atomic updates to the checkpoint values, ensuring continuous data flow throughout the pipeline.

Additionally, our Redis checkpointing system provides essential concurrency control that powers our distributed architecture's efficiency. Utilizing Redis watch-multi-exec transactions, we can ensure that each successful batch retrieval triggers immediate atomic updates to the checkpoint values. This transactional approach guarantees that all our concurrent harvester instances operate with consistent state information, preventing race conditions while enabling horizontal scaling of our collection capacity across multiple pods targeting the same Mastodon servers.

The atomic transaction design we developed delivers multiple performance benefits that strengthen our overall architecture. It enables true parallelization of our collection workload without data corruption risks, prevents redundant API calls by ensuring each timeline segment is harvested exactly once, and minimizes unnecessary load on Mastodon servers through our optimized request patterns. This coordination mechanism substantially improves our system fault tolerance by guaranteeing comprehensive post collection with minimal duplication or omission, even when multiple harvester instances operate simultaneously in our distributed cloud environment.

We temporarily store posts in a Redis queue before indexing them in Elasticsearch, which functions as an essential buffer and load balancer in our data pipeline. This intermediate storage approach we implemented decouples our harvesting and indexing processes, allowing each component to operate asynchronously, minimising bottlenecks between the processes. During high-volume collection periods, our Redis queue absorbs traffic spikes that might otherwise overwhelm Elasticsearch, preventing data loss and system instability. Additionally, this architecture provides us with fault tolerance, as our harvested data persists in the queue even if Elasticsearch becomes temporarily unavailable, ensuring complete data processing once our system recovers.

Our system leverages a hybrid execution strategy, combining PoolManager and NewDeploy executors, to dynamically respond to increased processing demands while optimizing resource efficiency. PoolManager maintains a pool of four pre-warmed pods for low latency preprocessing functions, ensuring predictable performance, while NewDeploy enables elastic scaling for harvester functions, automatically adjusting pod counts based on CPU usage to handle traffic spikes without manual intervention. Both executors scale back when demand decreases, with PoolManager reusing idle pods and NewDeploy reducing pod counts, minimizing resource waste. When we deploy a new harvester, it seamlessly integrates with our Redis-based state management system, receiving server endpoints and ID queue parameters through request arguments, as implemented in our harvest functions, and begins contributing to data collection without manual configuration. This hybrid scaling capability allows our system to efficiently respond to emerging events driving increased social media activity and to expand coverage to new social platforms as they become relevant to our research objectives.

4.2. Reddit

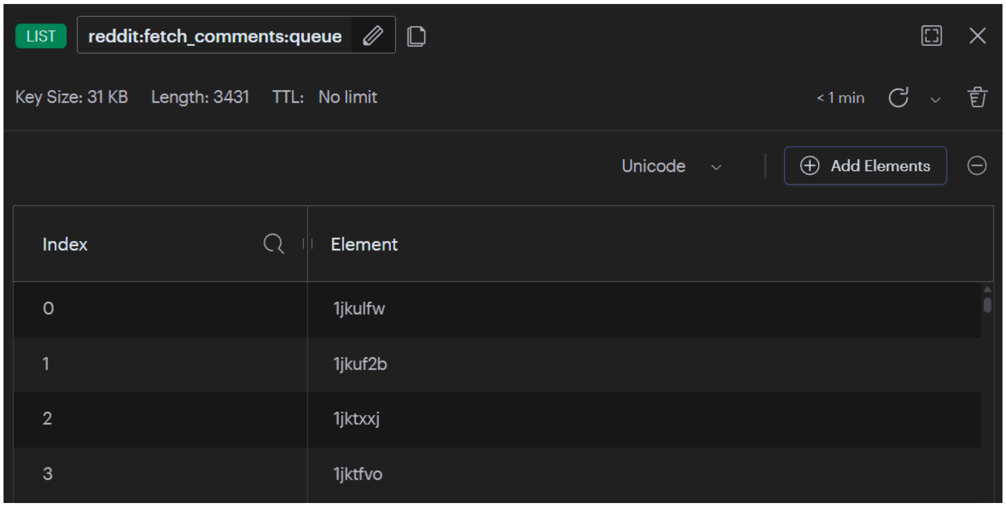
Reddit serves as a key data source for this project, with our focus on Australia-specific subreddits such as r/australia, r/melbourne, r/sydney, r/ausproperty, and r/ausfinance to capture diverse discussions, opinions, and news. Reddit’s free APIs provide access to public data, enabling programmatic collection from these targeted communities.

To manage API rate limits, we employ a pool of multiple API authentication tokens. A function randomly selects a token for each API request, distributing requests across tokens to ensure consistent data flow and minimize downtime due to rate-limiting errors.

Like the Mastodon harvester, the Reddit submission harvester retrieves batches of 100 submissions using min_id and max_id parameters as pagination cursors, updating these IDs after each batch to ensure continuous collection. These submission IDs are then enqueued into a Redis list to trigger comment harvesting. To manage API rate limits, we use a pool of multiple API authentication tokens, with a function randomly selecting a token for each request to distribute load and minimize rate-limiting errors.

Our harvesting strategy however mainly prioritizes comments over submissions to maximize data volume, as Reddit's API restricts submission retrieval to the 1000 most recent entries. Each submission can generate 10 to 200 comments, providing richer perspectives. A Redis queue system manages this process: submission IDs from targeted subreddits are enqueued into a dedicated Redis list, serving as a work queue for a separate comment harvesting process that retrieves comment batches for each submission.

The comment harvester is activated via a message queue trigger tied to the Redis list. When a submission ID is enqueued, it initiates an event-driven workflow, ensuring the harvester only consumes resources when work is available. This loose coupling allows independent scaling of submission and comment harvesting processes. For instance, additional comment harvesters can be deployed if the queue length grows, without affecting submission harvesting. Asynchronous processing enables rapid submission ID enqueuing, while the I/O-bound comment harvesting operates independently, preventing bottlenecks in the ingestion pipeline.



reddit:fetch_comments:queue

Key Size: 31 KB Length: 3431 TTL: No limit

Index	Element
0	1jkulfw
1	1jkuf2b
2	1jktxxj
3	1jktfvo

Figure 9. Posts stored in Redis queue to be preprocessed

The architecture enhances fault tolerance through the Redis queue's persistence. If a comment harvester fails, unprocessed submission IDs remain in the queue, allowing other instances to resume work, ensuring data consistency and preventing loss. Rate limit errors are isolated to individual harvester instances, and affected submission IDs can be re-queued or handled by another instance after the token's rate limit resets, maintaining system continuity.

5. Scenario Analysis

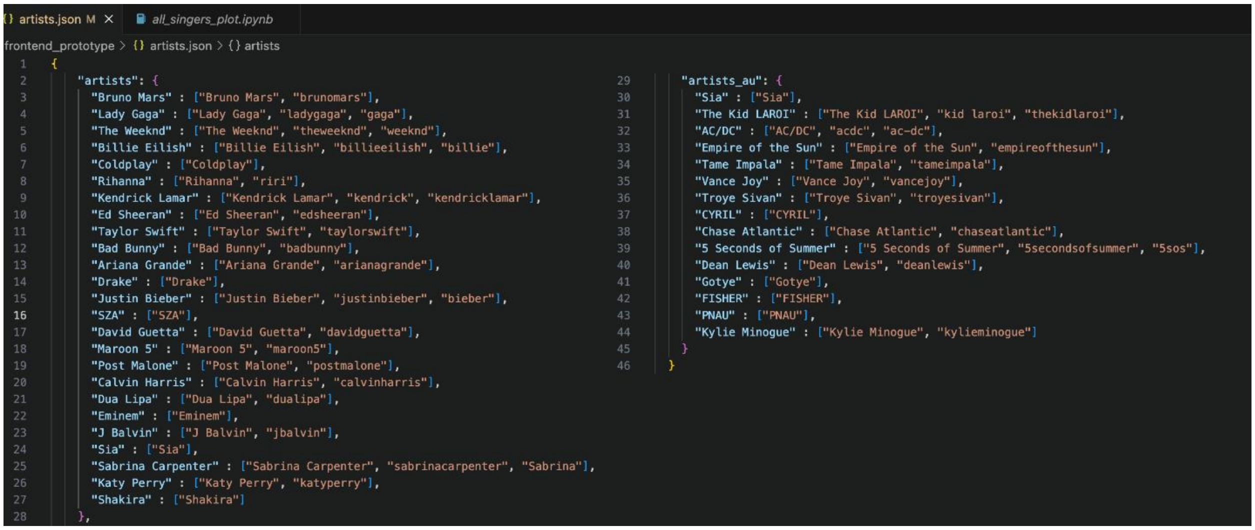
5.1. Scenario 1: Popularity of Pop Artists among Australian Audiences

5.1.1 Define the Scenario and Scope of Research

In this scenario, we compare international and Australian pop singers in terms of popularity. Who is more popular among Australian social media users? Do Australians prefer local artists? And is there a correlation between streaming data and social media discourse?

First, we defined the pool of artists by querying **Spotify's Global Monthly Listener rankings**. We selected the **top 25** artists globally, starting with Bruno Mars and ending with Shakira. We then filtered for artists who were born, or began their music careers in Australia, selecting the **top 15**, starting with Sia and ending with Kylie Minogue.

Next, we attached a list of common aliases to each artist's formal name to handle nicknames, abbreviations, and hashtags. The final scope of artists is structured in nested dictionaries below. Note that Australian artist Sia also appears in the global top 25, and FISHER was removed due to excessive noise. Therefore, the total number of artists in this analysis is **38**, not 40.



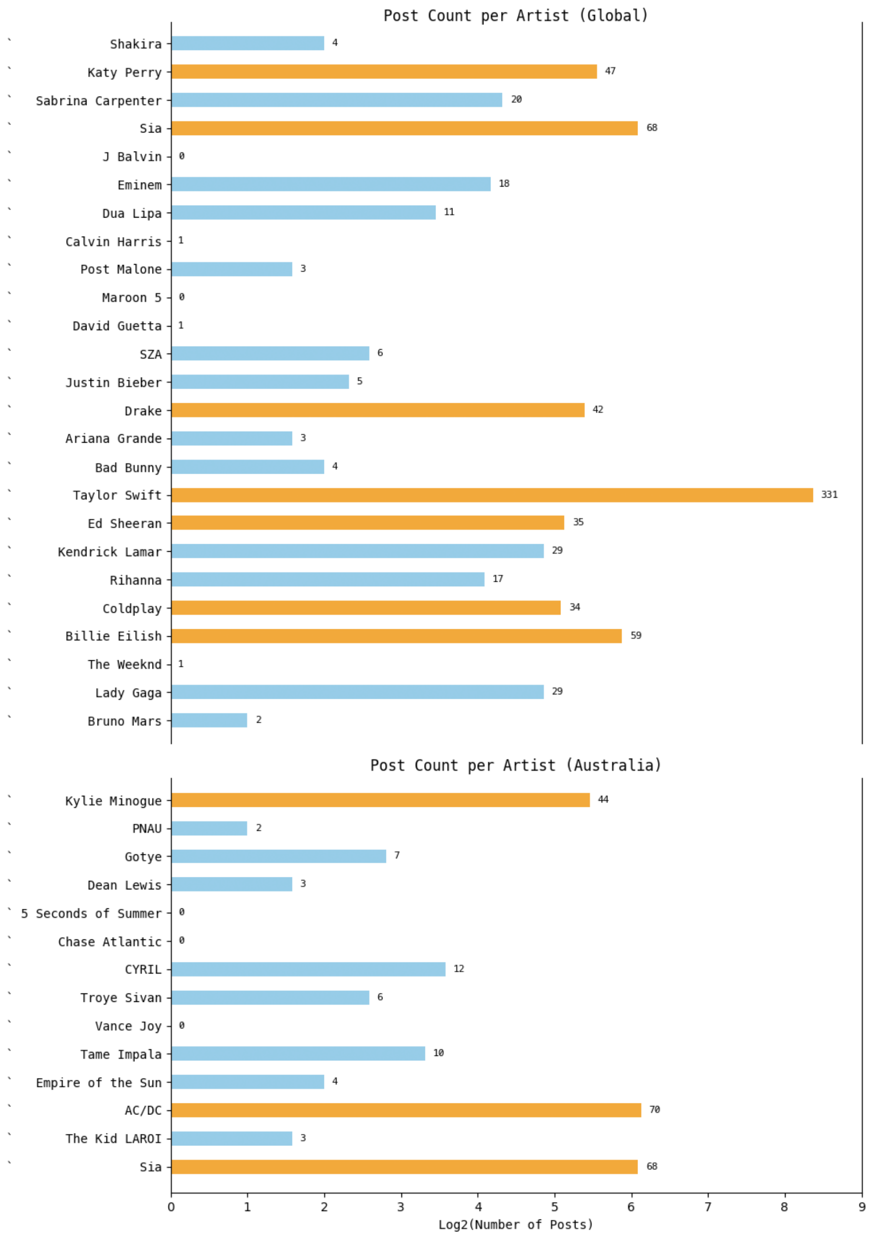
```
{
    "artists": {
      "Bruno Mars" : ["Bruno Mars", "brunomars"],
      "Lady Gaga" : ["Lady Gaga", "ladygaga", "gaga"],
      "The Weeknd" : ["The Weeknd", "theweeknd", "weeknd"],
      "Billie Eilish" : ["Billie Eilish", "billieeilish", "billie"],
      "Coldplay" : ["Coldplay"],
      "Rihanna" : ["Rihanna", "riri"],
      "Kendrick Lamar" : ["Kendrick Lamar", "kendrick", "kendricklamar"],
      "Ed Sheeran" : ["Ed Sheeran", "edsheeran"],
      "Taylor Swift" : ["Taylor Swift", "taylorswift"],
      "Bad Bunny" : ["Bad Bunny", "badbunny"],
      "Ariana Grande" : ["Ariana Grande", "arianagrande"],
      "Drake" : ["Drake"],
      "Justin Bieber" : ["Justin Bieber", "justinbieber", "bieber"],
      "SZA" : ["SZA"],
      "David Guetta" : ["David Guetta", "davidguetta"],
      "Maroon 5" : ["Maroon 5", "maroon5"],
      "Post Malone" : ["Post Malone", "postmalone"],
      "Calvin Harris" : ["Calvin Harris", "calvinharris"],
      "Dua Lipa" : ["Dua Lipa", "dualipa"],
      "Eminem" : ["Eminem"],
      "J Balvin" : ["J Balvin", "jbalvin"],
      "Sia" : ["Sia"],
      "Sabrina Carpenter" : ["Sabrina Carpenter", "sabrinacarpenter", "Sabrina"],
      "Katy Perry" : ["Katy Perry", "katyperry"],
      "Shakira" : ["Shakira"]
    },
    "artists_au": {
      "Sia" : ["Sia"],
      "The Kid LAROI" : ["The Kid LAROI", "kid laroi", "thekidlaroi"],
      "AC/DC" : ["AC/DC", "acdc", "ac-dc"],
      "Empire of the Sun" : ["Empire of the Sun", "empireofthesun"],
      "Tame Impala" : ["Tame Impala", "tameimpala"],
      "Vance Joy" : ["Vance Joy", "vancejoy"],
      "Troye Sivan" : ["Troye Sivan", "troyesivan"],
      "CYRIL" : ["CYRIL"],
      "Chase Atlantic" : ["Chase Atlantic", "chaseatlantic"],
      "5 Seconds of Summer" : ["5 Seconds of Summer", "5secondsofsummer", "5sos"],
      "Dean Lewis" : ["Dean Lewis", "deanlewis"],
      "Gotye" : ["Gotye"],
      "FISHER" : ["FISHER"],
      "PNAU" : ["PNAU"],
      "Kylie Minogue" : ["Kylie Minogue", "kylieminogue"]
    }
}
```

Popularity is measured by **two criteria**: how many posts mention them, and the sentiment of those posts. We look at both the overall data, and how they change over time.

5.1.2. Plotting Post Counts

The customised API endpoint, "mention-count-by-artist-final", queries the "content" field of all documents in index "artists" using aliases, then computes a union for all aliases associated with each artist. The total number of posts mentioning each artist is plotted using their formal names.

We observe a significant imbalance of social media counts, with some getting no mentions throughout 4 years of dataset, while some had hundreds. To visually mitigate the imbalance, a



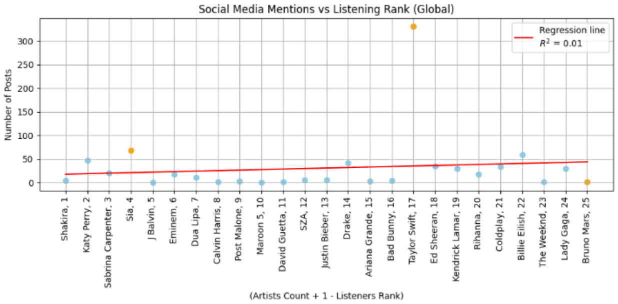
base-2 logarithm is applied to the post counts. Each grid along the x-axis doubles the total count.

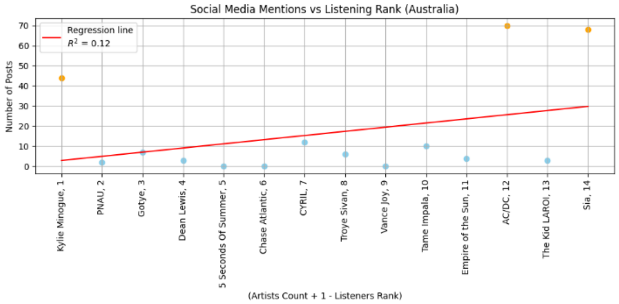
The highest count goes to **Taylor Swift**, with 331 posts mentioning her in texts or in hashtags, at the time of this screenshot.

We can also notice that Australian artists with more mentions are generally **in their 50s to 70s**, the highest being **AC/DC** with 70 mentions; in contrast, popular international singers skew younger, indicating a noticeable **generational gap** of Australian artists with global impact.

Meanwhile, in the international category, the three artists on the bottom (meaning having the most monthly listeners) — Bruno Mars, Lady Gaga, and The Weeknd — have relatively few mentions. This led us to explore **whether streaming data correlates with social media presence**.

We plotted each artist's rank against their social media mention count and performed the **least squares linear regression** to test for correlation. The results are shown below. Note that numbers may **vary** if users rerun the cells, as new documents continue to be added to the database.





The regression results show a positive coefficient between an artist's listening rank and the number of social media posts, with a slope of 1.088 for international artists and 2.068 for Australian artists.

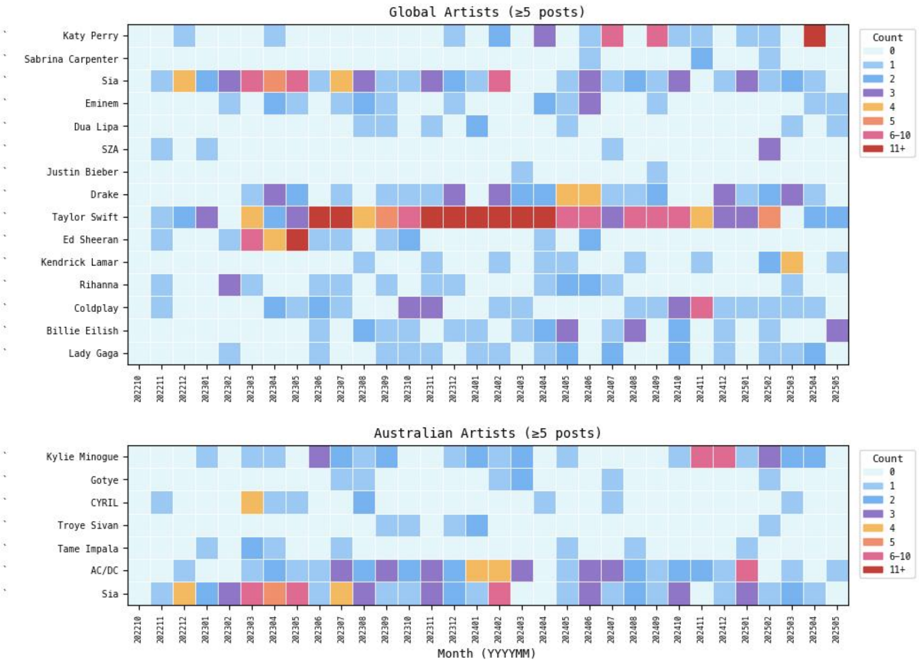
The gentler slope of 1.088 for international artists is influenced by **Sia**, who ranks lower in listening but has a relatively high post count, lifting the left side of the regression line. This assumption could be supported by Sia being marked in orange colour, indicating **having top 3 residuals**. Since the social media data comes from the Australian Mastodon server regardless of whether the artist discussed is local or international, this could reflect **stronger support for domestic artists** within the "international" category.

In contrast, the steeper slope for Australian artists suggests a **clearer alignment** between listening popularity and social media discussion volume among local performers.

5.1.3. Plotting Post Count Trends

Sudden spikes in social media presence often indicate major events related to the artists. To link these spikes to real-world activities, we aggregated monthly mention counts **from the earliest data (Oct 2022) harvested, up to real time**. These values are mapped into colours to create a heatmap.

The hottest area on the plot aligns with **Taylor Swift's Eras Tour** which spans from Mar 2023 to Dec 2024. Another example is Katy Perry, the release of Woman's World in Jul 2024, and her Blue Origin space trip in Apr 2025, are both in sync with warm colours on the plot.



5.1.4. Plotting Overall Sentiment

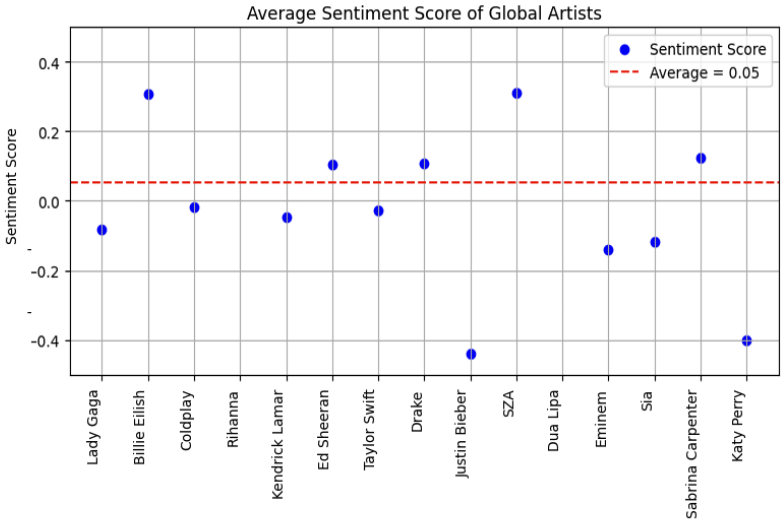
Of course, being discussed does not equate to being welcomed. So how are these artists perceived by Australian social media users?

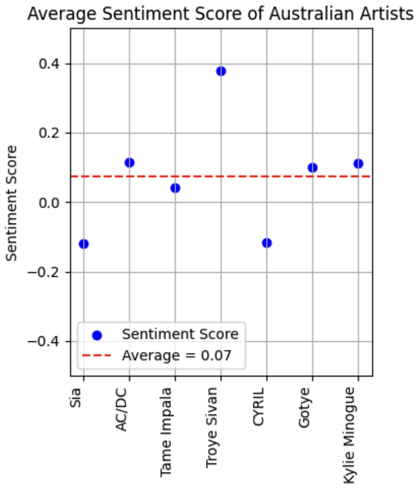
The **RoBERTa** ML model assigns each post a sentiment label, along with the probabilities of it being positive, neutral, or negative. To assist interpretation, these probabilities are converted into a single numerical sentiment score using the formula:

Score = (P(positive) - P(negative)) / (0.5 + P(neutral))

This score captures both **polarity** and **confidence**. Starting with the difference between the probabilities of this post being positive and negative: when a post is labelled neutral (i.e. P(neutral) > 0.5), the numerator is dampened; when it is not labelled neutral, the sentiment is amplified.

The average sentiment scores across the dataset are plotted for each artist, grouped into international and Australian. The top 3 are **Troye Sivan**, **SZA**, **Billie Eilish**, and bottom 3 are Justin Bieber, Katy Perry, Eminem, at the time of this screenshot.



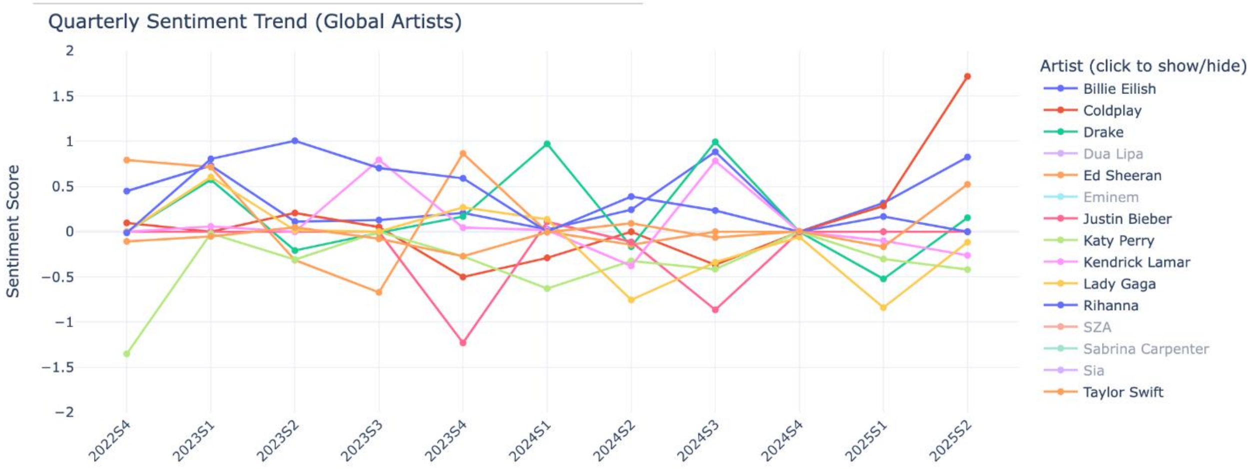


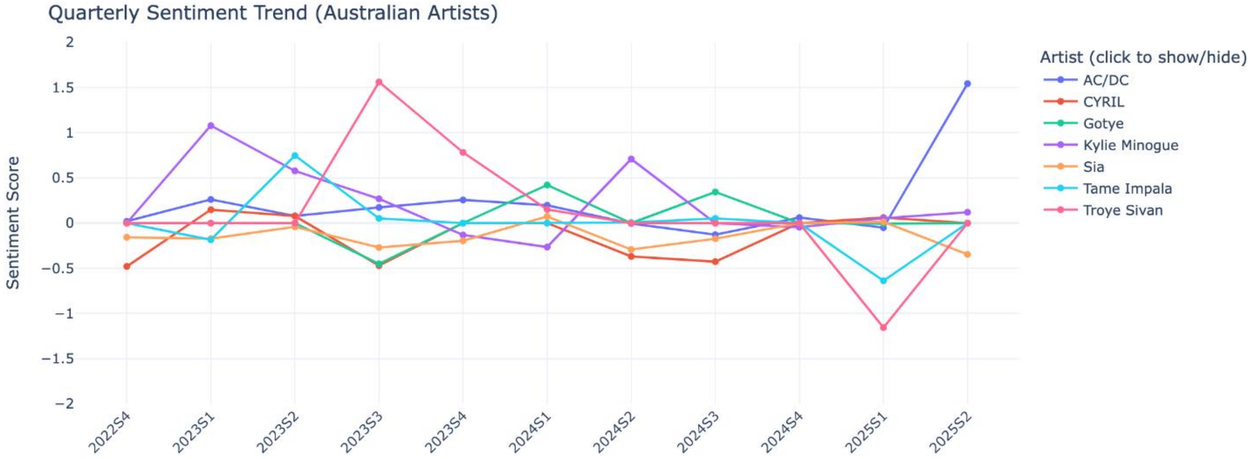
With enough data, this plot may answer whether Australian social media user prefer local artists. The most liked artist being Australian, and the most disliked artists all being non-Australian, may suggest a local preference.

Meanwhile, the average scores for international and Australian artists are 0.05 and 0.07. We use ttest_ind from scipy.stats which performs a **t-test** on the two groups with hypothesis being “the two groups have the same averages”. P-value was 0.839 at the time of t-test, meaning that we **can only be 16.1% sure** that Australian artists are genuinely better received by Australian people.

5.1.5. Plotting Sentiment Trends

How does public sentiment toward each artist evolves over time? To examine this, sentiment scores are aggregated **quarterly** to visualise this fluctuation, plotted using Plotly Express. The plot is interactive in the Jupyter Notebook, and users can click on the legend to toggle each line for comparison purposes.





Public sentiment often fluctuates in response to specific events. For example, Katy Perry's space trip may have attracted public scrutiny and critics. In contrast, Coldplay's "Music of the Spheres" tour, which included performances in Melbourne and Sydney in Nov 2024, appeared to be acclaimed by the Australian audience.

These examples underscore the dynamic nature of public opinion and highlight how real-world events can significantly influence social media sentiment toward celebrities.

5.2. Scenario 2: What does Australian social media really think of Trump?

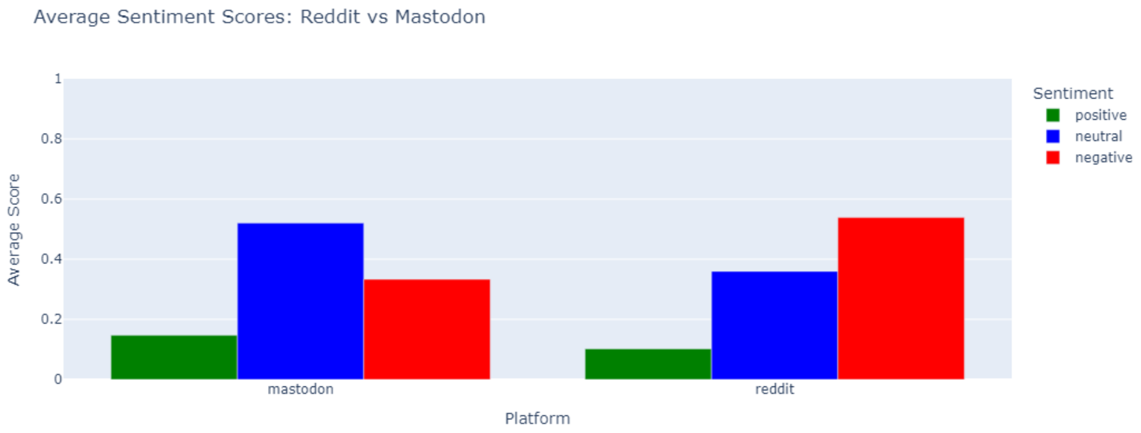
5.2.1. Scenario Introduction

Social media platforms like Mastodon and Reddit offer valuable insights into real-time public sentiment and opinions. Our team collected approximately 5,000 posts from Reddit and 12,000 posts from Mastodon to explore Australian social media users' perceptions of Donald Trump. We aim to examine differences in sentiment between these platforms and track how attitudes have evolved over time. By investigating whether specific platforms reflect distinct sentiments and identifying major events that may have influenced Trump's popularity or disapproval, this study seeks to uncover patterns in public opinion and their implications for broader socio-political trends in Australia.

5.2.2. Analysis of Sentiment between Reddit and Mastodon

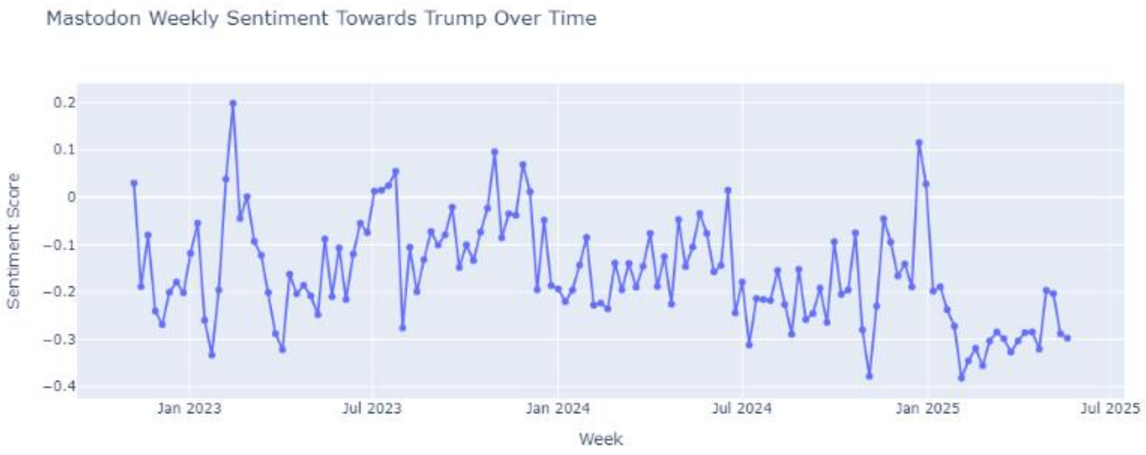
From the bar plot below comparing the average sentiment scores between Reddit and Mastodon users we see that Australian social media users on both Mastodon and Reddit generally dislike Donald Trump, with Reddit showing stronger negative sentiment (0.54) compared to Mastodon (0.33), while positive sentiment scores for both Reddit and Mastodon remain at 0.1 and 0.15 respectively.

This stronger negative sentiment on Reddit may attributed to the recency of our Reddit data since it only includes comments from 2025 which coincides with significant developments like the imposition of U.S. tariffs on Australian goods and influence on Australian politics have been widely criticized by Australian users and linked to broader anxieties about cost of living and economic stability. Mastodon's broader timeframe (2022-2025) likely contributes to its higher neutral sentiment (0.52), reflecting a more balanced perspective over time.

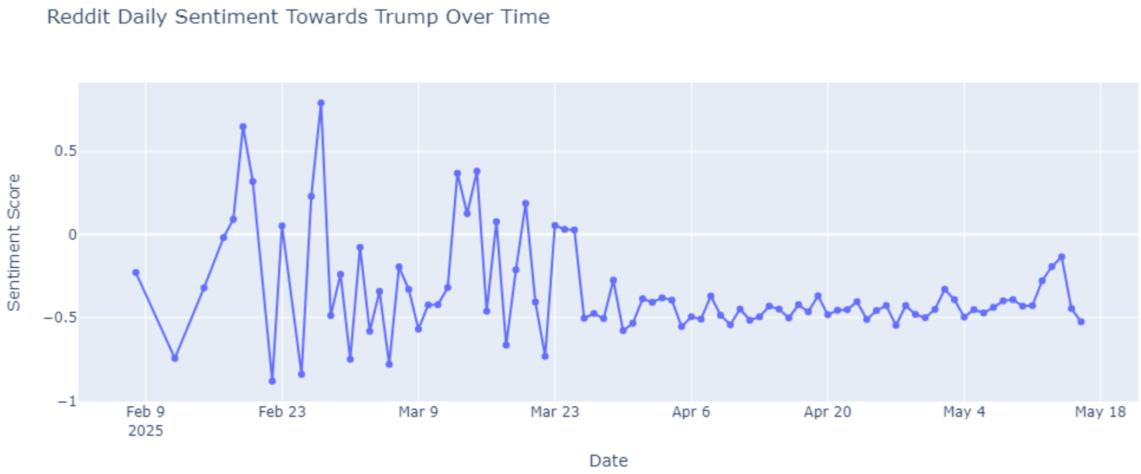


5.2.3. Analysis of Sentiment Over Time

While the aggregate scores illustrate overall sentiment tendencies of the general Reddit/Mastodon user base, they do not fully capture the changes in opinions over time. To further investigate how time plays a role in the fluctuations of Australian social media sentiment towards Trump, we analyze the changes in sentiment score over time. To plot the sentiment scores over time, we first transform the sentiments (positive, neutral, and negative) into a single score by doing (positive – negative). Since the Mastodon posts span a significantly larger time frame compared to the Reddit data, we decided to plot the weekly changes in scores for Mastodon compared to the daily changes for Reddit.



Looking at the Mastodon plot, we see that overall sentiment towards Trump from 2022 to 2025 is negative, with significant peaks on February 13, 2023, driven by his 2024 election campaign, and on December 23, 2024, driven by his decisive November 5, 2024, election victory after a campaign emphasizing economic grievances and immigration control. These peaks were followed by rapid declines due to escalating legal challenges, including his March 30, 2023, indictment for hush money payments and ongoing January 6 investigations, and unpopular policies such as 25% tariffs in February 2025 and mass deportation efforts criticized by April 15, 2025.



The Reddit plot shows a volatile period peaking at 0.8 around mid-February due to his recent inauguration and early policy announcements, but this could be due to low post counts during February. This is followed by a sharp decline to -0.5 by late March as legal challenges and tariff backlash emerged. Sentiment remained mostly negative through April and early May, with a slight uptick around May 4 possibly linked to positive economic data, though it dropped again by May 18, reflecting ongoing criticism of his immigration policies.

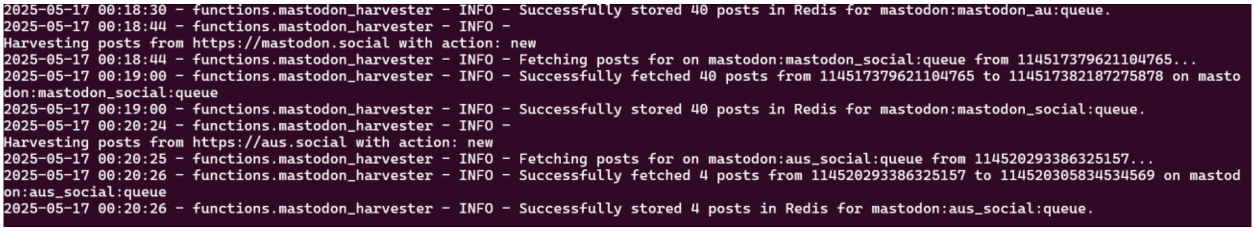
6. Error Handling and Encountered Issues

To ensure the quality and reliability of our software delivery, we employed multiple layers of validation. Static code analysis was conducted using Pylint, helping to enforce coding standards and detect potential issues early in the development cycle. Additionally, unit tests were implemented to verify the correctness of individual components in isolation, while end-to-end (E2E) tests were used to validate the system's behavior from a user perspective. Together, these practices formed a robust testing strategy aimed at maintaining code quality, functionality, and system stability throughout the development lifecycle.

Furthermore, we adopted a Continuous Integration and Continuous Deployment (CI/CD) pipeline to streamline the development workflow. This automation allowed us to integrate code changes frequently, run tests consistently, and deploy features rapidly and reliably. By embedding CI/CD into our process, we enhanced development productivity and ensured that high-quality features could be delivered continuously with minimal manual intervention.

6.1. Fission Function Tests

To ensure the reliability and performance of our cloud-based functions deployed in a Kubernetes environment, we conducted comprehensive Fission function tests. These tests were designed to validate that each function operated as intended, had access to sufficient resources, and could efficiently handle data processing tasks. A key component of our debugging strategy was the implementation of a robust logging framework, which provided detailed insights into function execution and error handling. By leveraging a custom logger class and Fission's built-in logging and testing tools, we systematically monitored and debugged our functions, ensuring seamless operation in the cloud.

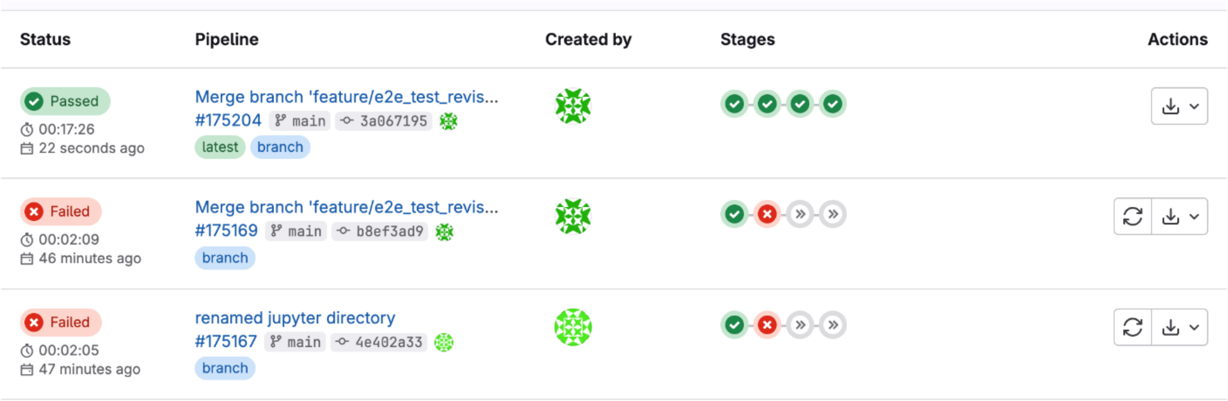


```
2025-05-17 00:18:30 - functions.mastodon_harvester - INFO - Successfully stored 40 posts in Redis for mastodon:mastodon_au:queue.
2025-05-17 00:18:44 - functions.mastodon_harvester - INFO -
Harvesting posts from https://mastodon.social with action: new
2025-05-17 00:18:44 - functions.mastodon_harvester - INFO - Fetching posts for on mastodon:mastodon_social:queue from 114517379621104765...
2025-05-17 00:19:00 - functions.mastodon_harvester - INFO - Successfully fetched 40 posts from 114517379621104765 to 114517382187275878 on masto
don:mastodon_social:queue
2025-05-17 00:19:00 - functions.mastodon_harvester - INFO - Successfully stored 40 posts in Redis for mastodon:mastodon_social:queue.
2025-05-17 00:20:24 - functions.mastodon_harvester - INFO -
Harvesting posts from https://aus.social with action: new
2025-05-17 00:20:25 - functions.mastodon_harvester - INFO - Fetching posts for on mastodon:aus_social:queue from 114520293386325157...
2025-05-17 00:20:26 - functions.mastodon_harvester - INFO - Successfully fetched 4 posts from 114520293386325157 to 114520305834534569 on mastod
on:aus_social:queue
2025-05-17 00:20:26 - functions.mastodon_harvester - INFO - Successfully stored 4 posts in Redis for mastodon:aus_social:queue.
```

We implemented a dedicated logger class, which standardized logging across all functions. We utilized Fission's logging and testing commands to track critical operations, such as monitoring whether posts were successfully indexed into Elasticsearch, verifying whether API rate limits were met, and tracking the number of posts harvested in each batch. This granular logging enabled us to confirm successful data harvesting and post insertion into Elasticsearch or to identify specific points of failure. Additionally, diagnosing issues such as failed Elasticsearch insertions or token verification errors allowed us to quickly pinpoint and resolve problems, enabling targeted code adjustments to prevent errors from cascading.

6.2. E2E Test

The implemented end-to-end (E2E) tests play a critical role in maintaining the reliability of our API Gateway, specifically the Analyser APIs. These tests automatically verify the health and responsiveness of each API endpoint. If any endpoint fails during testing, it indicates a potential issue introduced by the latest merge request. Such failures can directly impact downstream services—for example, preventing the Jupyter Notebook interface from displaying the summarized, analysed result correctly. By integrating E2E checks into our CI/CD pipeline, we ensure that every code merge maintains API integrity, prevents regressions error issues, and upholds the overall quality of the system.



Status	Pipeline	Created by	Stages	Actions
✓ Passed 00:17:26 22 seconds ago	Merge branch 'feature/e2e_test_revis... #175204 main 3a067195 latest branch		✓ ✓ ✓ ✓	
✗ Failed 00:02:09 46 minutes ago	Merge branch 'feature/e2e_test_revis... #175169 main b8ef3ad9 branch		✓ ✗ » »	
✗ Failed 00:02:05 47 minutes ago	renamed jupyter directory #175167 main 4e402a33 branch		✓ ✗ » »	

Figure 9. CI/CD Pipeline including Pylint, Unit-Test, Deploy and E2E Testing

6.3. Challenges

Although we had solid unit and end-to-end tests integrated into the pipeline, there were still instances where unexpected issues occurred. For example, the Fission timer wasn't running as expected. We later discovered that Fission timers don't support intervals in seconds, even though such configurations can be set via Fission commands. We resolved this by digging through pod logs and using trial-and-error approaches.

In another case, additional fields were added to the harvester, but the Elasticsearch mapping wasn't updated accordingly. We resolved this quickly by communicating through our WeChat group and applying fixes as soon as possible.

In a real-world scenario, we could adopt more robust troubleshooting and monitoring practices—for instance, implementing an EFK (Elasticsearch, Fluentd, Kibana) stack for application performance monitoring. These ideas are further discussed in the Future Improvements section below.

6.3.1. Data Collection Issues

Mastodon Harvester

Our initial harvesting strategy for Mastodon posts relied on using tag-based filtering but this yielded poor results due to majority of the posts not having tags assigned to them. We solved this by targeting high-traffic Australian instances (Mastodon.au, aus.social) and harvesting all posts from these instances. By implementing bidirectional pagination with min_id and max_id parameters, we achieved comprehensive timeline collection that overcame the limitations of tag-based filtering.

Managing collection state across multiple concurrent harvesters presented risks of data duplication or gaps. We addressed this by creating a Redis-based checkpoint system that stored pagination markers for each server endpoint. We used Redis's atomic transactions to prevent race conditions when harvesters updated collection states simultaneously. This approach helped maintain collection continuity while allowing us to scale harvester instances horizontally.

Our direct harvesting to indexing pipeline created bottlenecks during high-volume periods, often overwhelming Elasticsearch and risking data loss. We solved this by implementing a Redis queue between harvesting and processing components. This buffer helped absorb traffic spikes and maintained data persistence during downstream outages, significantly improving our system stability.

As we added additional servers to harvest posts from, we needed our harvester infrastructure to scale dynamically. To manage this, we implemented a Fission-based serverless architecture that automatically scaled harvester functions based on demand. This approach allowed new harvester instances to integrate seamlessly with our state management system, with server endpoints passed through request arguments. Our solution efficiently distributed collection workloads while maintaining consistent state management across different Mastodon instances.

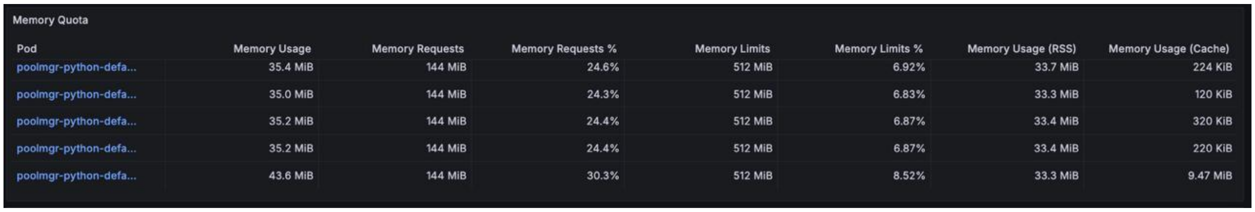
Reddit Harvester

Reddit’s API imposed strict rate limits, hindering parallelized harvesting, while Mastodon’s tag-based filtering yielded inconsistent results due to variable tag usage and limited historical query capabilities. To overcome Reddit’s rate limits, we implemented a pool of API authentication tokens, randomly selecting a token for each request to distribute load and maintain consistent data flow. Before initiating any API request, a function randomly selects a token from this pre-configured batch. This distribution of requests across different tokens significantly mitigates the risk of any single token hitting its rate limit. This contributes to a more efficient and consistent data flow by minimizing downtime attributable to rate-limiting errors.

The second major challenge is that Reddit API's restricts post retrieval to only the 1000 most recent entries. This constraint inherently curtails the volume of historical data collectable directly via submissions. In order to increase the amount of data we can harvest, we strategically pivoted our harvesting approach to prioritise comments. While post numbers are capped, each submission can generate a substantial volume of comments, often ranging from 10 to 200 for popular subreddits. This strategy allows us to maximise the amount of data we can retrieve from Reddit. To manage this approach, a separate Redis queue system was implemented to when new submissions are harvested from targeted subreddits, their unique IDs are enqueued into a dedicated Redis list. This list subsequently acts as a work queue for a distinct comment harvesting process, which methodically harvests batches of comments for each post.

6.3.2. Fission Issues

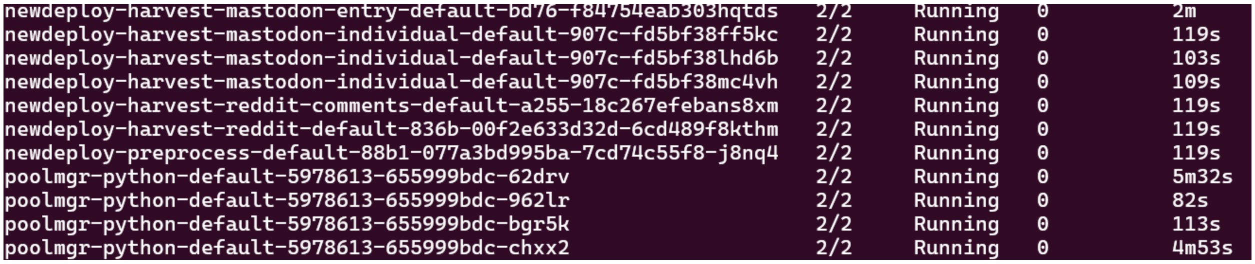
Our initial Fission deployments revealed critical challenges stemming from poor resource management, where some functions were allocated significantly more pods than others causing misallocated or evicted due to resource mismanagement. This problem was particularly when high min scale values for pods and uncapped resources for individual pods triggered rapid scaling cascades during traffic spikes, destabilizing the Kubernetes control plane. For instance, the Reddit comments harvester function, initially deployed without specified resource constraints for its message queue trigger mechanism, exhibited considerable operational instability. During peak load periods, this function instantiated approximately 30 concurrent pods, thereby depleting the cluster's available resource allocation. This excessive resource consumption resulted in the exhaustion of computational capacity reserved for other critical functions, subsequently causing the forced eviction of all managed pods within the pool manager's jurisdiction.



Memory Quota

Pod	Memory Usage	Memory Requests	Memory Requests %	Memory Limits	Memory Limits %	Memory Usage (RSS)	Memory Usage (Cache)
poolmgr-python-defa...	35.4 MiB	144 MiB	24.6%	512 MiB	6.92%	33.7 MiB	224 KiB
poolmgr-python-defa...	35.0 MiB	144 MiB	24.3%	512 MiB	6.83%	33.3 MiB	120 KiB
poolmgr-python-defa...	35.2 MiB	144 MiB	24.4%	512 MiB	6.87%	33.4 MiB	320 KiB
poolmgr-python-defa...	35.2 MiB	144 MiB	24.4%	512 MiB	6.87%	33.4 MiB	220 KiB
poolmgr-python-defa...	43.6 MiB	144 MiB	30.3%	512 MiB	8.52%	33.3 MiB	9.47 MiB

To address resource management challenges, we implemented hybrid approach that uses both the PoolManager and NewDeploy. First, we utilized Grafana to monitor CPU and memory usage patterns, establishing precise resource boundaries for pods and functions. Our PoolManager's pooling mechanism maintains four pre-warmed pods for low-latency preprocessing functions, while NewDeploy's dynamic scaling for harvester functions, prioritizing resource allocation based on workload demands. The scaling is driven by CPU utilization, with thresholds set at 80% and maximum concurrent instances capped at 3–5 per function. For instance, harvester functions like harvest-reddit scale up to 5 pods with limits of 500m CPU and 128Mi memory, while preprocessors use PoolManager with only 64Mi memory limits. We introduced these changes through graduated deployments, testing configurations incrementally to assess performance before full implementation. By leveraging Redis queue depth for workload distribution and enforcing namespace resource quotas, we prevent individual components from monopolizing cluster resources during peak demand, ensuring equitable computing capacity across our social media data collection system.



```
newdeploy-harvest-mastodon-entry-default-bd76-f84754eab303hqtds   2/2     Running   0          2m
newdeploy-harvest-mastodon-individual-default-907c-fd5bf38ff5kc   2/2     Running   0          119s
newdeploy-harvest-mastodon-individual-default-907c-fd5bf38lhd6b   2/2     Running   0          103s
newdeploy-harvest-mastodon-individual-default-907c-fd5bf38mc4vh   2/2     Running   0          109s
newdeploy-harvest-reddit-comments-default-a255-18c267efebans8xm   2/2     Running   0          119s
newdeploy-harvest-reddit-default-836b-00f2e633d32d-6cd489f8kthm   2/2     Running   0          119s
newdeploy-preprocess-default-88b1-077a3bd995ba-7cd74c55f8-j8nq4   2/2     Running   0          119s
poolmgr-python-default-5978613-655999bdc-62drv                    2/2     Running   0          5m32s
poolmgr-python-default-5978613-655999bdc-962lr                    2/2     Running   0          82s
poolmgr-python-default-5978613-655999bdc-bgr5k                    2/2     Running   0          113s
poolmgr-python-default-5978613-655999bdc-chxx2                    2/2     Running   0          4m53s
```

Building on these resource optimization efforts, we further refined our data harvesting infrastructure, which was originally built on time-triggered functions for preprocessing and indexing posts into Elasticsearch. This time-based approach contributed to our resource allocation problems, as functions executed on schedule even when no work was available. The resulting resource shortage left critical functions starved of necessary pods and compute capacity. By transitioning to a message queue trigger that activates only when posts are present in the Redis queue, we were able to eliminate unnecessary function calls and create a responsive pipeline that dynamically accelerates processing during high-volume periods, ensuring optimal resource utilization across our entire Fission deployment.

7. Evaluation and Improvement

7.1. High Availability, Single Point of Failure and Fault Tolerance Evaluation

The system was architected with a layered design approach, separating functionality across distinct tiers to improve modularity and resilience. To mitigate single points of failure and enhance fault tolerance, redundancy and horizontal scalability were incorporated throughout the stack.

At the fission harvester layer, Fission functions were deployed in multiple pods (minimum of two), enabling seamless failover and load distribution. For message queuing, Redis was configured as a clustered setup. This ensures high availability—if one Redis node fails, other nodes can continue processing without service interruption. The preprocessing services that consume messages from Redis were also deployed in a scalable manner, with multiple pods ready to handle load surges or node failures.

Elasticsearch was configured with two primary shards and one replica per shard. This setup ensures data availability and query reliability even in the event of a node failure, as the replicas can immediately take over.

The API service layer was similarly designed for high availability, with multiple pods deployed. This allows the system to route traffic away from any failing instance while maintaining uninterrupted service.

By applying these design principles, redundancy, horizontal scalability, and distributed architecture—we have effectively minimized the risk of single points of failure and significantly improved the system's fault tolerance and operational reliability.

7.2. Future Improvement

Test Effectiveness - While unit tests and end-to-end (E2E) testing were implemented in this project, the absence of system integration testing was a notable gap in our CI/CD pipeline. This shortcoming occasionally led to delays in identifying integration issues across components. Although adopting a test-driven development (TDD) approach could have improved overall reliability, familiarity with TDD practices varied among team members, limiting its application. Despite this, strong team collaboration played a key role in mitigating the impact. Team members consistently demonstrated a shared sense of responsibility, addressing integration issues proactively—even when the problems fell outside their assigned tasks.

Application Troubleshooting – System failures are inevitable, and it's crucial to have centralized access to application logs—or even better, a monitoring system that can proactively alert operators when something goes wrong. Implementing an EFK stack (Elasticsearch, Fluentd, Kibana) would allow us to collect logs from application pods via sidecar containers and stream them directly to Elasticsearch. This setup would significantly improve productivity and reduce debugging time by eliminating the need to manually inspect logs pod by pod.

Security - To further enhance the security of our application, we recommend integrating automated security tools such as SonarQube, WhiteSource (now Mend), or similar solutions into the CI/CD pipeline. These tools can perform static code analysis, vulnerability scanning, and license compliance checks, helping to identify and mitigate potential security risks early in the development cycle.

Incorporating security checks into the pipeline aligns with DevSecOps best practices, ensuring that security becomes an integral part of the software delivery process rather than a post-deployment concern. This is particularly important if the application is ever intended for public release, where compliance with recognized standards such as NIST, OWASP, or CIS Benchmarks would be essential to ensure the system is secure, auditable, and production-ready.

Proactively embedding these practices will not only improve the security posture of the application but also demonstrate maturity in software governance and operational readiness.

8. Teamwork Management

8.1. Team Collaboration and Communication

We held regular extended design and knowledge-sharing meetings twice per week, two hours per each session to foster alignment and encourage technical discussion. Continuous communication was maintained through a dedicated WhatsApp group, ensuring rapid support and collaboration whenever issues arose. Each team member contributed actively, with defined roles and responsibilities outlined in the table below.

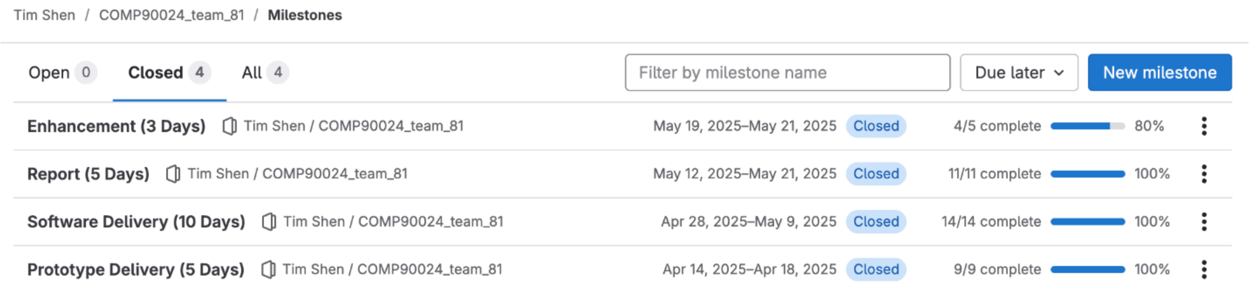
Team Member	Responsibility
Adam McMillan	• Implemented a CI/CD pipeline with test and deployment stages • Configured idempotent Fission package deployment • Coordinated overall integration of backend Fission architecture (including development of clean interfaces between harvester, pre-processer, Elastic, and analyser components) • Introduced easily configurable settings for harvesting, and querying functions to allow for customisable application behaviour based on JSON config • Debugging of Fission function behaviour
Ryan Kuang	• Built and deployed data harvesters for Mastodon and Reddit • Created storage and preprocessing workflows for Elasticsearch indexing • Configured Redis queue system to balance load between harvesting and indexing operations • Set up logging and did debugging for Fission functions • Configured YAML files for Fission functions and event triggers with resource optimization • Explored Trump-related scenarios by aggregated Mastodon and Reddit data from Elasticsearch, and created suitable visualizations
Tim Shen	• Led high-level design of the distributed architecture. • Deployed, configured, operate, troubleshoot, core infrastructure components, including K8s clusters, Elasticsearch, KEDA, Redis, and Prometheus, to support system scalability and observability. • Implemented unit test, end-to-end (E2E) testing CI/CD pipelines and developed key API backend RESTFUL services. • Introduced and applied project management, agile development methodologies, share to help team processes and execution.
Yili Liu	• Explored feasibility of different scenarios, experimented on the ease of harvesting different data sources. • Finalised the scenario with actionable scope, re-indexing ElasticSearch databases for optimised scenario-specific queries via API and Kibana. • Designed the Jupyter Notebook frontend, interacting with database, processing, aggregating, and visualising data to assist narrating the scenarios.

	- Daily maintenance of teamwork (arranging time and room for meetups).
Yuting Cai	- Researching and selecting a suitable sentiment analysis machine learning model - Design the sentiment analysis model structure and deploy it in Kubernetes clusters in effective way - Optimizing the model operation process works successfully - Checking whether specific indices from ElasticSearch that they have sentiment score and labels, which allow our group to do analysis.

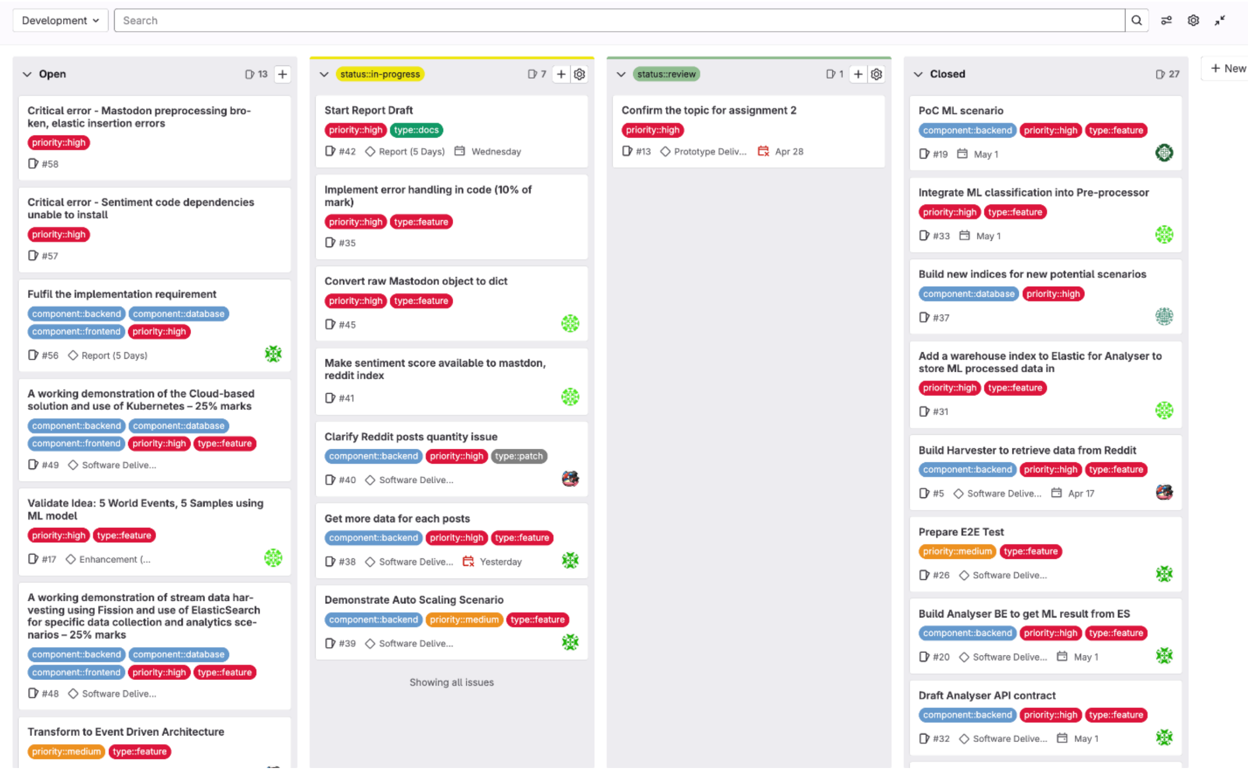
8.2. Project Management

Initially, we explored using JIRA for project management, given its widespread adoption in software development. However, due to budget constraints, we opted for GitLab's integrated project management tools. Task prioritization was managed through a label-based system, enabling clear visibility into team responsibilities and workflow. We adhered to agile practices, including daily virtual stand-ups meetings, twice per week face-to-face 2 hours meeting consistent with Scrum methodology, and structured our work around well-defined milestones to maintain direction and momentum.

To validate feasibility from the user's perspective, we rapidly developed a prototype (bluesky_harvester_poc), which informed the iterative implementation of detailed features. The GitLab Scrum board and milestone tracking facilitated real-time updates on task assignments and priorities, ensuring alignment and progress across the team. Completion of all issue tickets signaled the successful delivery of the project to a high standard.



Enhancement (3 Days)	Tim Shen / COMP90024_team_81	May 19, 2025–May 21, 2025	Closed	4/5 complete	80%
Report (5 Days)	Tim Shen / COMP90024_team_81	May 12, 2025–May 21, 2025	Closed	11/11 complete	100%
Software Delivery (10 Days)	Tim Shen / COMP90024_team_81	Apr 28, 2025–May 9, 2025	Closed	14/14 complete	100%
Prototype Delivery (5 Days)	Tim Shen / COMP90024_team_81	Apr 14, 2025–Apr 18, 2025	Closed	9/9 complete	100%



Open 13

Critical error - Mastodon preprocessing broken, elastic insertion errors
priority::high
#58

Critical error - Sentiment code dependencies unable to install
priority::high
#57

Fulfil the implementation requirement
component::backend component::database component::frontend priority::high
#56 Report (5 Days)

A working demonstration of the Cloud-based solution and use of Kubernetes – 25% marks
component::backend component::database component::frontend priority::high type::feature
#49 Software Delive...

Validate Idea: 5 World Events, 5 Samples using ML model
priority::high type::feature
#17 Enhancement (...

A working demonstration of stream data harvesting using Fission and use of ElasticSearch for specific data collection and analytics scenarios – 25% marks
component::backend component::database component::frontend priority::high type::feature
#48 Software Delive...

Transform to Event Driven Architecture
priority::medium type::feature

status::in-progress 7

Start Report Draft
priority::high type::docs
#42 Report (5 Days) Wednesday

Implement error handling in code (10% of mark)
priority::high type::feature
#35

Convert raw Mastodon object to dict
priority::high type::feature
#45

Make sentiment score available to mastdon, reddit index
#41

Clarify Reddit posts quantity issue
component::backend priority::high type::patch
#40 Software Delive...

Get more data for each posts
component::backend priority::high type::feature
#38 Software Delive... Yesterday

Demonstrate Auto Scaling Scenario
component::backend priority::medium type::feature
#39 Software Delive...

Showing all issues

status::review 1

Confirm the topic for assignment 2
priority::high
#13 Prototype Deliv... Apr 28

Closed 27

PoC ML scenario
component::backend priority::high type::feature
#19 May 1

Integrate ML classification into Pre-processor
priority::high type::feature
#33 May 1

Build new indices for new potential scenarios
component::database priority::high
#37

Add a warehouse index to Elastic for Analyser to store ML processed data in
priority::high type::feature
#31

Build Harvester to retrieve data from Reddit
component::backend priority::high type::feature
#5 Software Delive... Apr 17

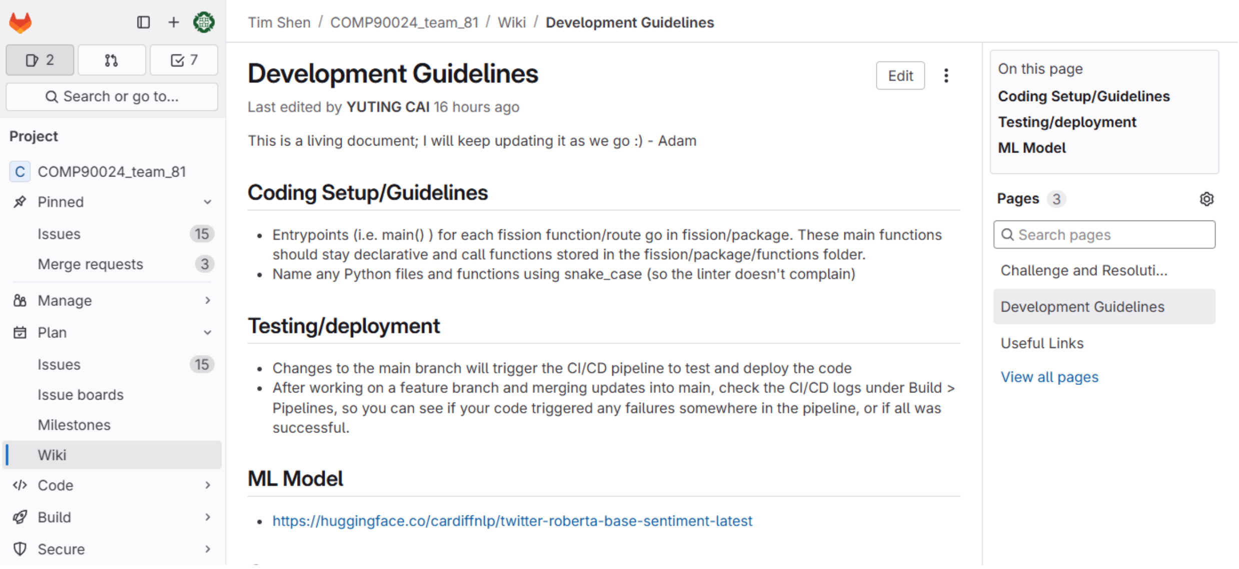
Prepare E2E Test
priority::medium type::feature
#26 Software Delive...

Build Analyser BE to get ML result from ES
component::backend priority::high type::feature
#20 Software Delive... May 1

Draft Analyser API contract
component::backend priority::high type::feature
#32 Software Delive... May 1

Figure 10. Gitlab Issues Board Adopting Agile Development

In addition, we leveraged GitLab's Wiki feature as a central repository for team knowledge. Team members actively documented lessons learned, useful resources, and frequently used shortcuts. This collaborative knowledge base enhanced our shared understanding and significantly improved the efficiency and productivity of Team 81.



Development Guidelines

Last edited by **YUTING CAI** 16 hours ago

This is a living document; I will keep updating it as we go :) - Adam

Coding Setup/Guidelines

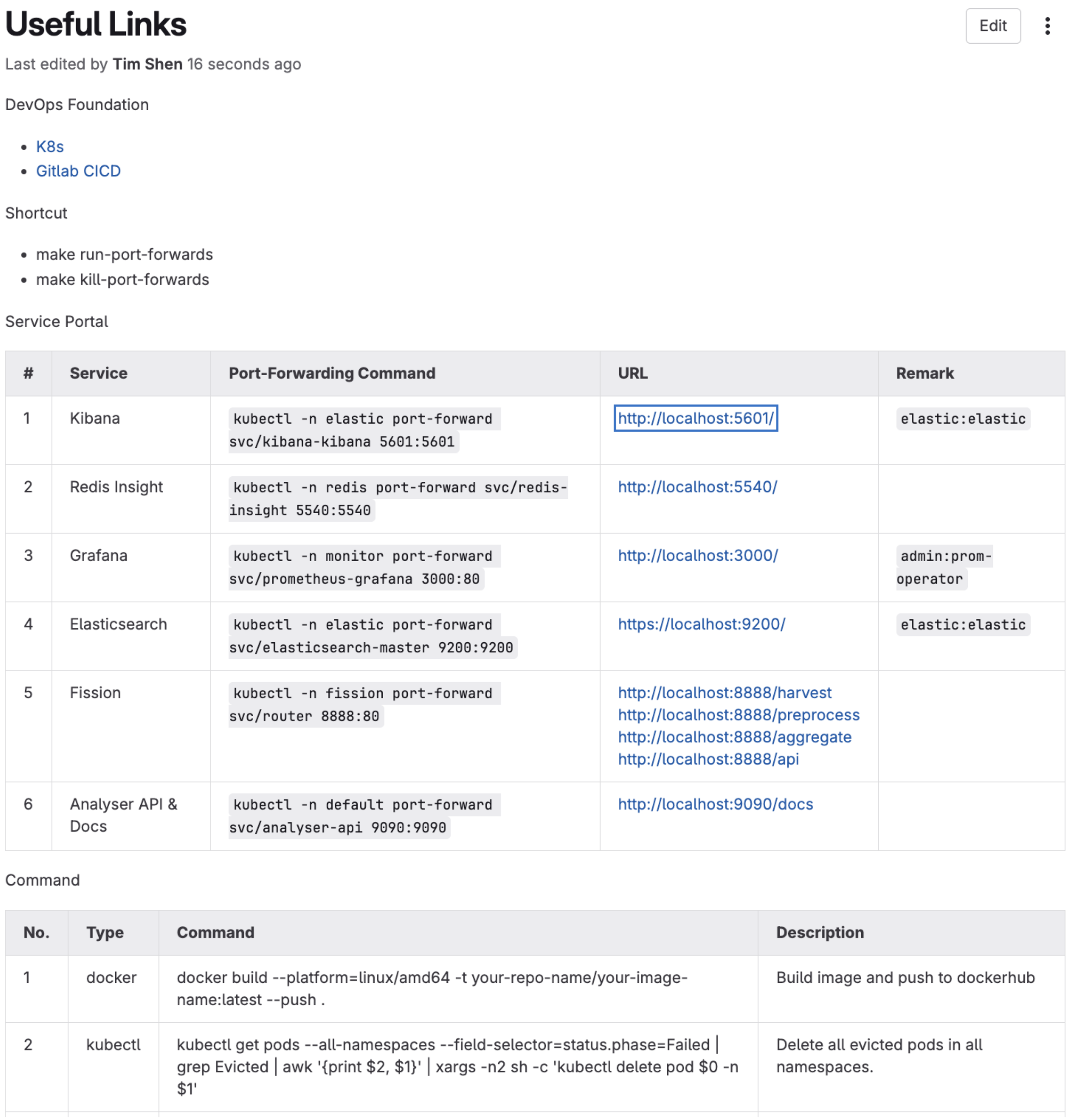
- Entrypoints (i.e. main()) for each fission function/route go in fission/package. These main functions should stay declarative and call functions stored in the fission/package/functions folder.
- Name any Python files and functions using snake_case (so the linter doesn't complain)

Testing/deployment

- Changes to the main branch will trigger the CI/CD pipeline to test and deploy the code
- After working on a feature branch and merging updates into main, check the CI/CD logs under Build > Pipelines, so you can see if your code triggered any failures somewhere in the pipeline, or if all was successful.

ML Model

- https://huggingface.co/cardiffnlp/twitter-roberta-base-sentiment-latest



Useful Links

Last edited by **Tim Shen** 16 seconds ago

DevOps Foundation

- K8s
- Gitlab CICD

Shortcut

- make run-port-forwards
- make kill-port-forwards

Service Portal

#	Service	Port-Forwarding Command	URL	Remark
1	Kibana	`kubectl -n elastic port-forward svc/kibana-kibana 5601:5601`	http://localhost:5601/	`elastic:elastic`
2	Redis Insight	`kubectl -n redis port-forward svc/redis-insight 5540:5540`	http://localhost:5540/	
3	Grafana	`kubectl -n monitor port-forward svc/prometheus-grafana 3000:80`	http://localhost:3000/	`admin:prom-operator`
4	Elasticsearch	`kubectl -n elastic port-forward svc/elasticsearch-master 9200:9200`	https://localhost:9200/	`elastic:elastic`
5	Fission	`kubectl -n fission port-forward svc/router 8888:80`	http://localhost:8888/harvest http://localhost:8888/preprocess http://localhost:8888/aggregate http://localhost:8888/api	
6	Analyser API & Docs	`kubectl -n default port-forward svc/analyser-api 9090:9090`	http://localhost:9090/docs	

Command

No.	Type	Command	Description
1	docker	docker build --platform=linux/amd64 -t your-repo-name/your-image-name:latest --push .	Build image and push to dockerhub
2	kubectl	kubectl get pods --all-namespaces --field-selector=status.phase=Failed | grep Evicted | awk '{print $2, $1}' | xargs -n2 sh -c 'kubectl delete pod $0 -n $1'	Delete all evicted pods in all namespaces.

Figure 11. Knowledge Sharing for Productivity Improvement

9. Conclusion

Throughout this project, we gained substantial hands-on experience with a suite of modern cloud-native technologies. We explored Fission as a Function-as-a-Service (FaaS) platform to build event-driven microservices, and used Helm to simplify Infrastructure-as-a-Service (IaaS) provisioning and management on OpenStack, enabling efficient deployment of Kubernetes clusters. We also integrated a range of open-source tools—Redis, Elasticsearch, Prometheus, and Grafana—to support data processing, storage, monitoring, and observability within a distributed system.

Most importantly, we deepened our practical understanding of cloud computing principles, including horizontal scaling, autoscaling, stateless service design, resilience, and fault tolerance. We applied these concepts to address load management at scale—an essential skill set for operating reliable and performant systems in production-grade environments.

Additionally, we adopted GitLab as an end-to-end DevOps platform, enabling continuous integration, continuous delivery (CI/CD), and collaborative development. This significantly streamlined our workflow and improved code quality through automation and testing.

Beyond the technical outcomes, our team demonstrated strong collaboration and disciplined project management. By following agile methodologies and maintaining clear communication, we delivered a well-architected, scalable, and maintainable solution aligned with industry best practices.

External Links

- README (Installation) : https://gitlab.unimelb.edu.au/yeeshen/comp90024_team_81
- Gitlab Repository URL: https://gitlab.unimelb.edu.au/yeeshen/comp90024_team_81
- YouTube Video URL: https://www.youtube.com/watch?v=XYiFBetXico
- ML model HuggingFace URL: https://huggingface.co/cardiffnlp/twitter-roberta-base-sentiment-latest

[i] Research on OpenStack of open source cloud computing in colleges and universities' computer room. (2017). ResearchGate. Retrieved from https://www.researchgate.net/publication/317554315_Research_on_OpenStack_of_open_source_cloud_computing_in_colleges_and_universities%27_computer_room

[ii] Common Use Cases & Benefits of OpenStack Technology for Universities. (2025). OpenMetal. Retrieved from https://openmetal.io/resources/blog/common-use-cases-benefits-of-openstack-technology-for-universities/