Assignment 2
Cluster and Cloud Computing
COMP90024
Repo: https://gitlab.unimelb.edu.au/DBBL/cloudcompa2
Video: https://www.youtube.com/watch?v=K7WnZXc1VW8

The University of Melbourne
Dillon Blake *1524907* Andrea Delahaye *1424289*
Yue Peng *958289* Jeff Phan *1577799*
Alistair Wilcox *212544*

May 2024

Contents

1 Introduction **4**

2 Architecture **5**
2.1 Infrastructure Design . . . 6
2.1.1 Bastion . . . 6
2.1.2 Kubernetes . . . 6
2.1.3 Network . . . 6
2.2 Service Deployment . . . 7
2.2.1 Elasticsearch . . . 7
2.2.2 Kibana . . . 7
2.2.3 Kafka . . . 7
2.2.4 Kafbat UI . . . 7
2.2.5 Fission . . . 8
2.2.6 KEDA . . . 9
2.2.7 Jupyter Notebook . . . 9
2.3 DevOps . . . 9
2.3.1 Git Workflow . . . 9
2.3.2 CI/CD . . . 10
2.3.3 Datadog . . . 11
2.3.4 Data Backup (AWS S3) . . . 11

3 Design **12**

3.1 Overall Diagram . . . 12
3.2 Fetching Static Data . . . 12
3.2.1 Crashes . . . 12
3.2.2 Crimes . . . 13
3.2.3 Weather . . . 13
3.3 Fetching Live Weather . . . 13
3.3.1 Current Observations . . . 15
3.3.2 Hourly Weather Collection . . . 16
3.4 Fetching Air Quality . . . 17
3.5 API . . . 18
3.5.1 Stations API . . . 19
3.5.2 Closest Station . . . 19
3.5.3 Weather API . . . 19
3.5.4 Live Weather API . . . 19
3.5.5 Hourly Weather API . . . 19
3.5.6 Crash API . . . 19
3.5.7 Crime API . . . 20
3.5.8 Air Quality API . . . 20
3.5.9 Stream API . . . 20
3.5.10 Models API . . . 20
3.5.11 EPA Data Collection . . . 20
3.5.12 API Tests . . . 21

4 Instructions **21**
4.1 Air Quality Models . . . 21
4.2 Crashes Models . . . 21
4.3 Crimes Models . . . 22

5 Results of the Data Analysis **22**
5.1 Air Quality vs Weather . . . 22
5.2 Crashes vs Weather . . . 23
5.3 Crimes vs Weather . . . 26

6 Discussions **28**
6.1 Single Point of Failure / Cluster Resilience . . . 28
6.2 MRC Latency Issue . . . 28
6.2.1 Issue Localization . . . 28
6.2.2 Final Solution . . . 29
6.2.3 Suggestion . . . 29
6.3 Other Limitations . . . 30
6.3.1 MRC Limited functionalities . . . 30
6.3.2 Limited CI/CD Pipeline . . . 30
6.4 Future Improvements . . . 30
6.4.1 Error Handling in Fission MQTrigger Functions . . . 30
6.5 ElasticSearch Machine Learning features . . . 31
6.6 Kafka . . . 31

6.7 CI/CD 31

7 Team and Roles **31**

7.1 Roles 31

7.1.1 Dillon Blake 31

7.1.2 Andrea Delahaye 31

7.1.3 Yue Peng 32

7.1.4 Jeff Phan 32

7.1.5 Alistair Wilcox 32

7.2 Strengths and Weaknesses of the Group 32

7.2.1 Strengths 32

7.2.2 Weaknesses 33

8 Conclusion **33**

1 Introduction

Is it safe to go outside?[2] This is a question that humans have asked for hundreds of years. Thanks to developments in cloud computing and statistical analysis, we may be one step closer to answering this question. In this paper, we detail an application developed on the Melbourne Research Cloud to collect and analyze geospatial data in Australia. Specifically, we explored the ability of weather data to predict air quality, car crashes, and crime. This system utilized a mix of static and streamed datasets that were uploaded to an instance of Elasticsearch deployed on a Kubernetes cluster. Serverless functions were used both to harvest live data and to serve stored data to users. To allow users to interact with the system, a suite of Jupyter Notebooks were created to provide an interface. The application also utilises technologies like automated tests and version control to ensure the stability and extendability of the system.

2 Architecture

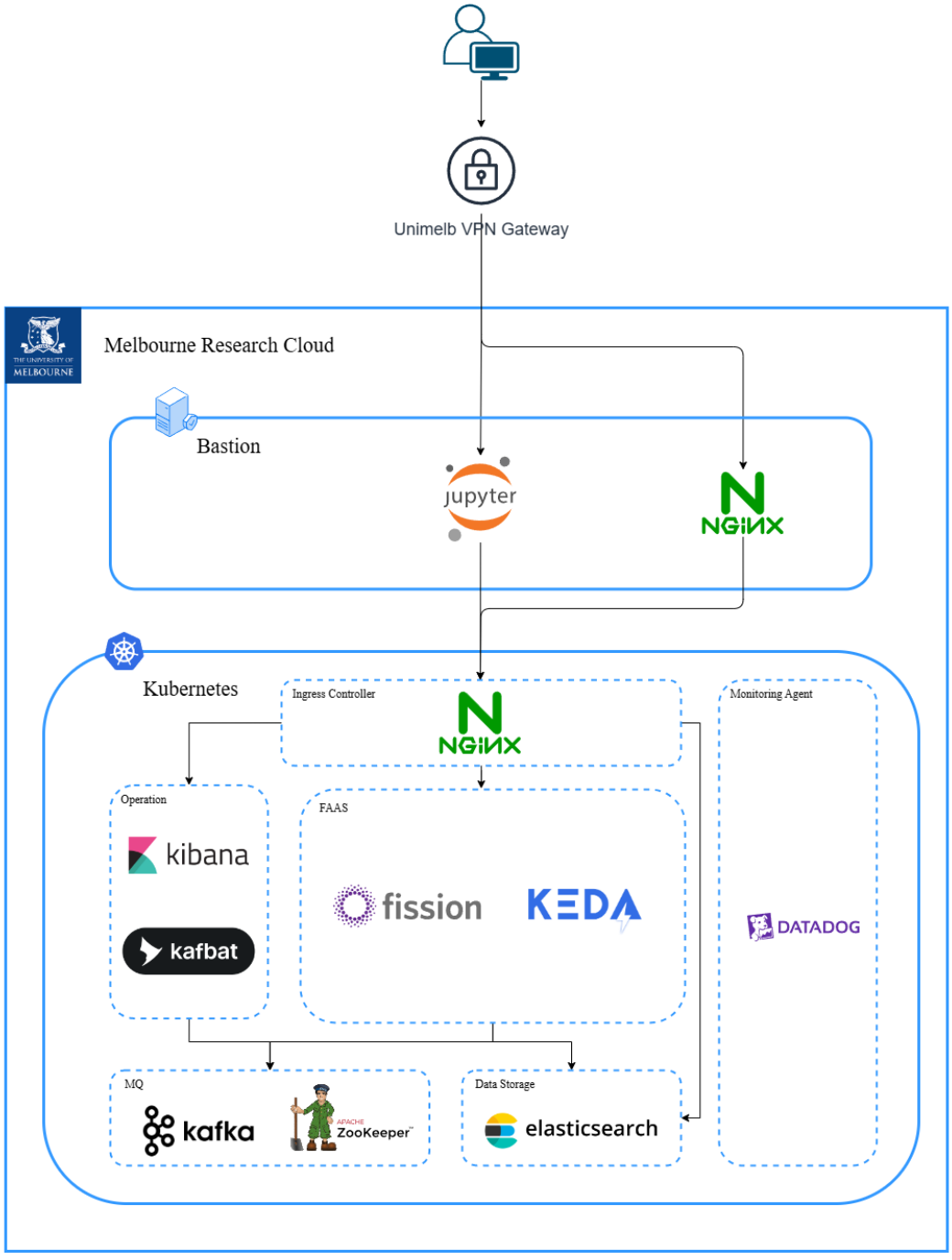


Figure 1: Tech stack

2.1 Infrastructure Design

2.1.1 Bastion

The bastion operates as a gateway to our system, permitting exclusively authorised users from the University of Melbourne VPN/network to connect to our Kubernetes cluster. By ensuring all connections to cluster nodes go through a controlled entry point, it minimizes the attack risks. This was where we deployed all of our code from. Additionally, it logs all access attempts and actions taken, providing a record for monitoring and deployment. This setup enhances security while also simplifying the management and maintenance of the K8s cluster.

2.1.2 Kubernetes

Our Kubernetes cluster is composed of one master node and four worker nodes, forming a scalable environment for managing containerized applications. The master node is responsible for maintaining the cluster's state, managing workloads, and coordinating tasks across the worker nodes through key components such as the API server, scheduler, and controller manager. The three worker nodes execute the actual workloads and run the applications within containers, managed by the kubelet and kube-proxy services. This setup ensures efficient distribution and orchestration of resources, providing high availability and fault tolerance. Kubernetes was also useful for managing our sensitive information like API keys and passwords. We utilised the `secrets` feature to store these values securely. With Kubernetes, we are able to scale and manage the lifecycle of our sytem in a controlled and consistent manner.

2.1.3 Network

In the Kubernetes cluster, a service provides a stable network endpoint to access a logical set of pods. It also brings the convenience in load balancing, decoupling and service discovery. Therefore we use services to facilitate reliable, scalable, and flexible network communication between different components of our application in the cluster.

However, we also need to expose some services to external networks outside of the cluster. One way is by using port-forwarding, which is not recommended in production environment. The other way is to setup an ingress. We use ingress-Nginx in the cluster as the ingress controller, which creates Nginx service in the cluster to route HTTP and HTTPS traffic. Initially we tried to deploy the Nginx service in "loadBalancer" mode but failed with no available quota for an external load balancer. Instead, it is deployed in "nodeport" mode so that the cluster can open specific ports on each node to accept external access. Ingress rules by host are configured for each service that we want to expose, and for HTTPS services like Elasticsearch we enable SSL pass-through of ingress-Nginx.

Now we allow external access to the cluster such as access from bastion, but we still can not access the cluster from our local PC because the cluster and bastion are in a dedicated network. The bastion resides in both the cluster network and UoM internal network, so it is the only gateway to the cluster. In order to link the cluster to UoM internal network, a Nginx reverse proxy is setup on the bastion server. HTTP requests to a service are firstly sent to the reverse proxy port on the bastion, and then forwarded to the node port defined in ingress-Nginx, and finally, routed to the service inside cluster by the host header of the request. Again, to enable SSL pass-through on Nginx, TCP stream proxy is required when configuring the rules. Last but not the least, setting up firewall in operation system and rules in the secruity group block out hostile attack to our server.

2.2 Service Deployment

2.2.1 Elasticsearch

Elasticsearch is a highly scalable search and analytics engine that excels in handling large volumes of unstructured data. Elasticsearch's distributed architecture ensures fault tolerance and high availability, making it an ideal choice for our system. Moreover, Elasticsearch is useful in geographic searches by providing powerful geospatial capabilities, allowing users to store, index, and query location-based data. It supports various geospatial queries, such as filtering results within a specific radius, bounding box, or polygon, and sorting results by distance. These features make Elasticsearch ideal for our geographic data analysis tasks, enabling precise and fast retrieval of spatial information.

In our deployment, we use a dual-master architecture and set one replica for all indices, making the two master nodes backups for each other to ensure high availability and prevent data loss. Additionally, to prevent resource contention from upstream services (Fission functions) during bulk data import scenarios, we use node selectors to allocate two dedicated nodes each for Elasticsearch and Fission functions. This ensures that resource usage does not conflict, thereby improving system performance. Additionally, to prevent the JVM memory insufficiency observed during data backup and recovery, it is necessary to increase the default JVM heap size through the Helm chart values. Finally, we enabled TLS communication on Elasticsearch to prevent man-in-the-middle attacks during external access to the cluster.

2.2.2 Kibana

Kibana is a data visualization and exploration tool integrated with Elasticsearch. We have mainly used it as a user interface for our Elasticsearch database. It can also be used to configure index patterns, monitor index health, manage index templates, as well as monitor snapshots.

In our deployment, we use the helm chart bitnami/elasticsearch which provides a bundle of both Elasticsearch and Kibana. With minimal configuration we can setup Kibana with a TLS connection to Elasticsearch.

2.2.3 Kafka

To serve our data streaming functionalities, we set up Kafka managed alongside Zookeeper to ensure coordinated and reliable operations. Zookeeper as the coordinator handles critical functions such as broker leader election, configuration management, and cluster state tracking, ensuring Kafka's high availability and fault tolerance. By leveraging Kafka with Zookeeper, we not only reduce the latency when dealing with large volumes of data but also enhance the efficiency of error handling.

2.2.4 Kafbat UI

For managing our Kafka infrastructure within the Kubernetes cluster, we utilise Kafbat UI, which provides a user-friendly interface for monitoring and managing Kafka clusters. We have used it to create new topics, send test message to a topic and even reset the lag of a consumer group to resend messages. For the monitoring, it allows us to visualise message flows, inspect consumer groups, and monitor lag, ensuring efficient data streaming and processing. We can use it to troubleshoot and optimise our Kafka operations, making it easier to maintain the reliability and performance of our data pipelines. By integrating Kafbat UI with our Datadog monitoring setup, we gain a comprehensive view of both system and application metrics, enabling us to manage our Kafka infrastructure within the Kubernetes environment.

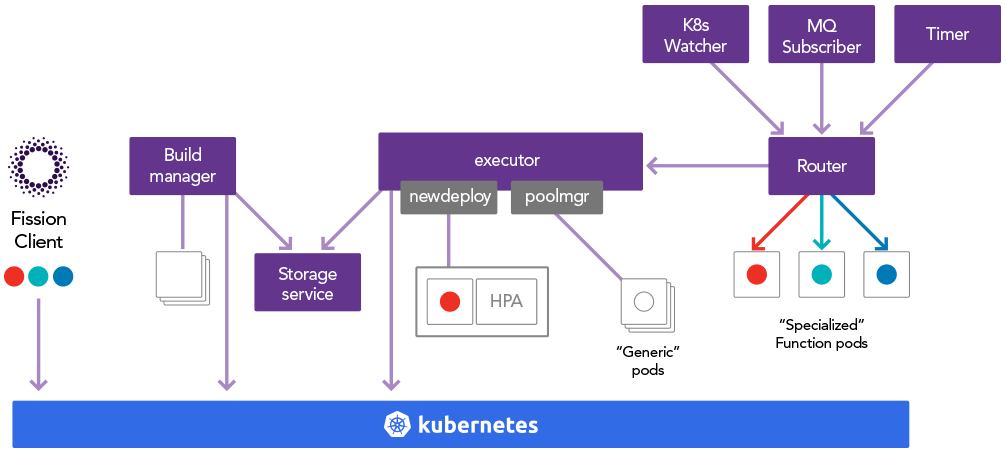


Figure 2: Fission Architecture

2.2.5 Fission

Fission is essential to our system as it enables us to run serverless functions and act as our API server for the front-end users. With this setup, we built and deployed event-driven and timer-driven applications without the complexity of managing underlying infrastructure. Fission functions are automatically scaled and managed by Kubernetes, providing a streamlined development process. Events such as HTTP requests from the front-end, timers, or Kafka messages trigger these functions. Debugging was also made easy by integrating the Python logging library to record status messages from the Fission functions.

Fission supports two executor modes: "pool manager" and "new deploy." The pool manager mode maintains a pool of pre-warmed pods. When handling new requests, the Fission executor uses a technique called "pod specialisation" which mounts the volume of requested function code to a pre-warmed pod and labels the pod to separate it from the pool deployment. This mode is suitable for rapid function execution by reusing these containers, thus reducing cold start latency. In contrast, the new deploy mode creates a new deployment and a HPA (Horizontal Pod Autoscaler) for each function execution, offering better isolation and resource management at the cost of increased resources. These modes allow Fission to balance between quick response times and resource efficiency depending on the application's needs.

In our deployment, we use "pool manager" mode for our function pods because their life spans are short and requires low latency, and we set the concurrency prudently to avoid too many pods created at once. While for the triggers pods, we use "new deploy" mode to ensure that each function execution occurs in a fresh, isolated environment and leverage HPA to scale instances up or down, optimizing resource usage and maintaining performance under varying loads. We set the minimal scale of all triggers to one to avoid cold start time and also the maximal scale to prevent resource preemption.

A custom environment(python-es) is built and used by our function pods. The custom environment includes a docker image with additional PyPI packages based on Fission's official Python image. Using this environment allows us to import modules like elasticsearch, scikit-learn, openai and so on in single file functions. Once we need to add new requirements, we only need to build a new docker image and apply it to the environment spec.

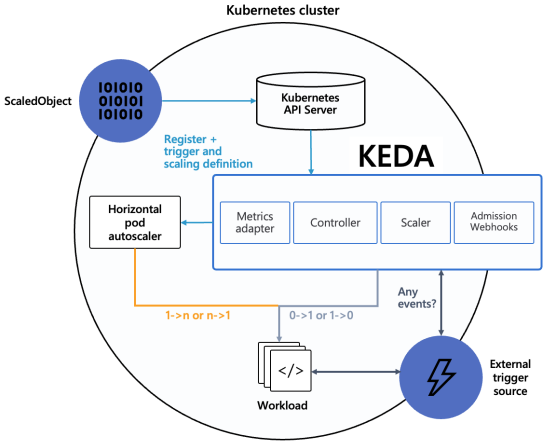


Figure 3: Keda Architecture

2.2.6 KEDA

We utilised KEDA as it is useful for auto-scaling Kafka consumers and facilitates event-driven scaling based on Kafka topic lag. With KEDA, we ensure that consumer instances scale dynamically according to the actual workload, optimizing resource utilization and reducing costs.

Fission needs to work with KEDA to create a message queue trigger. Fission executors first create a new ScaledObject to specify the details of the Kafka consumer. Secondly, the KEDA controller will create the deployment and HPA of the Kafka consumer once it monitors a creation of a ScaledObject. KEDA metrics adapter is responsible for providing the metrics of Kafka topics to the HPA so that HPA can calculate how many pods we need. After consumers are created, messages are sent to Fission router by the consumers and finally routed to the specialised function pod.

2.2.7 Jupyter Notebook

For our frontend, we utilised Jupyter Notebooks. These notebooks allow a simple interface to run Python code while also interacting with UI components like dropdowns and interactive diagrams. Because we are dealing with geospatial data, we decided to utilise interactive maps as well to let to users pick locations. Our bastion hosts the Jupyter Notebook server at
http://172.26.135.52:8888/lab/workspaces/auto-J/tree/cloudcompa2 so they can accessed on the UniMelb network.

2.3 DevOps

2.3.1 Git Workflow

For version control, we chose to utilise GitLab for its strong privacy controls. GitLab acts as the storage service for our git repositories while the git software managed our code. In development, we did our work on separate branches as to avoid pushing untested/invalid code to the master branch. We tried to keep the branch names logical to keep track of what features they corresponded to. Once we felt a branch was ready, we then opened a merge request with the master branch. GitLab was helpful in providing an intuitive interface for examining changes and resolving any conflicts that may have occurred on merge.

2.3.2 CI/CD

To facilitate our development process, a continuous integration and continuous deployment pipeline was created using GitLab. There are countless ways this could have been implemented, but for this project we focused on the basic phases of testing and deployment. GitLab utilises workers called runners to run the pipeline tasks. GitLab provides runners for basic functionality, but to keep it simple we chose to use our bastion as the worker using its shell. To do this, the GitLab Runner client is installed on the bastion to run in the background and accept jobs from the GitLab scheduler.

The pipeline is configured so that every time something in changed on the master branch, the CI/CD workflow begins as defined in the `.gitlab-ci.yml` file. First, various Makefiles are utilised to reinstall the Fission functions on the system. There is also a Makefile in the backend directory that uses pydoc to convert comments in our python files into HTML documentation. A Makefile in the doc directory is then used to deploy a Fission function with a httptrigger to serve the html documentation. The documentation can be accessed with the following links:

- `http://172.26.135.52:9090/docs/api.html` for the REST API
- `http://172.26.135.52:9090/docs/fission.html` for the Fission harvester functions
- `http://172.26.135.52:9090/docs/data_functions.html` for the static data functions

It is important to note that the documentation server does open some security issues as it does not currently check permissions for the files it serves. Next, tests are ran on key functions, namely the REST API. This is done using the unittest framework in Python where all tests are located in the test directory. A typical CI/CD pipeline runs tests before deployment. However, since many of our tests interact with Fission functions we needed to deploy the code before testing. To accommodate for this, the testing phase checks whether or not the tests pass. If there are any failures, it automatically rolls back to the last commit on the master branch and rebuilds. The combination of the build and test phases ensures that the system maintains a reliable working state.

2.3.3 Datadog

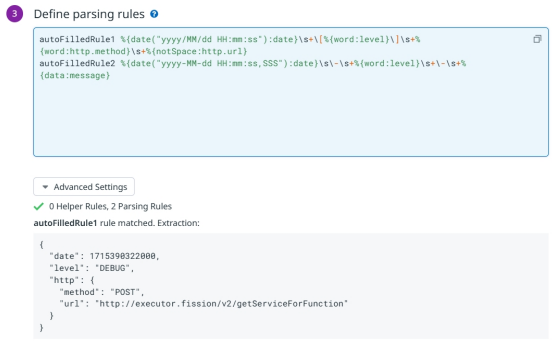


Figure 4: Log parser pipeline

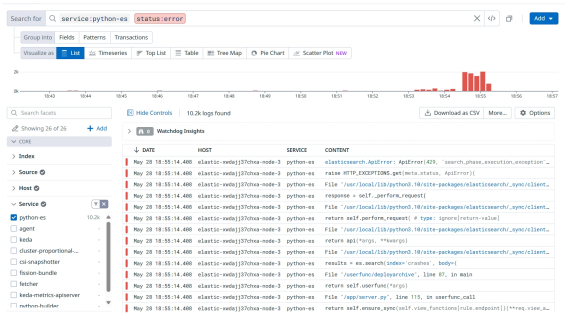


Figure 5: Log search

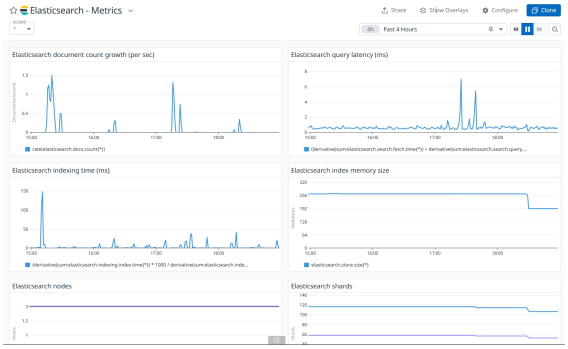


Figure 6: Elasticsearch metrics

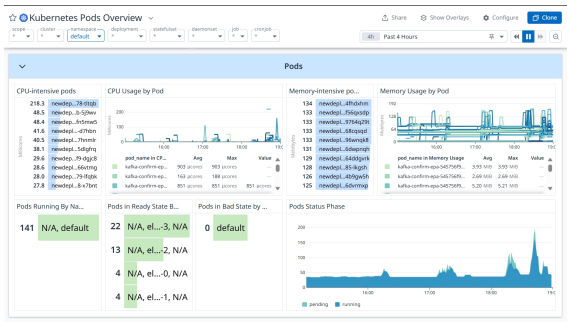


Figure 7: Pods metrics

To enhance monitoring within our Kubernetes cluster, we have integrated Datadog. Datadog provides comprehensive real-time visibility into the performance and health of our cluster by collecting and visualizing metrics, logs, and traces from all nodes and containers. This integration allows us to monitor key performance indicators, detect anomalies, and troubleshoot issues with greater efficiency. The screenshots in figures 4, 5, 6, and 7 show some examples of the data that Datadog provided us to help in debugging. This was especially helpful when we encountered issues with the MRC as discussed section 6.

2.3.4 Data Backup (AWS S3)

In order to guarantee the data durability and facilitate disaster recovery, we employ AWS S3 to keep snapshots of our Elasticsearch indices. This was extremely helpful as we worked though issues with the MRC. There were times that we had to resintall Elasticsearch so AWS S3 saved us many hours of reuploading data.

To create a snapshot on AWS S3 we first need to create an AWS IAM user with sufficient privilege to operate S3. And then the most tricky part is to inject the access ID and secret key to Elasticsearch keystorage. The first attempt to inject the secrets by logging into the Elasticsearch pod and execute the binary exacutable *elasticsearch-keystore* failed because the secrets need to be injected before bootstrapping Elasticsearch. We then tried init-container but it also failed because the modification on container layer is isolated. The final solution is by placing scripts inside *docker-entrypoint-initdb.d* where the container engine ensures any scripts will be executed when the container starts. Luckily, bitnami/elasticsearch provides directly chart values for that, and the chart even supports passing Secrets in order to hide AWS secret key. We setup the link to the S3 bucket, define a backup policy and manually trigger the backup if necessary. The only thing needs to be

aware of during data restoration is to increase the JVM heap size, because during large data throughput the defualt heap size of 128MB is too small and OutOfMemoryError will arise.

3 Design

3.1 Overall Diagram

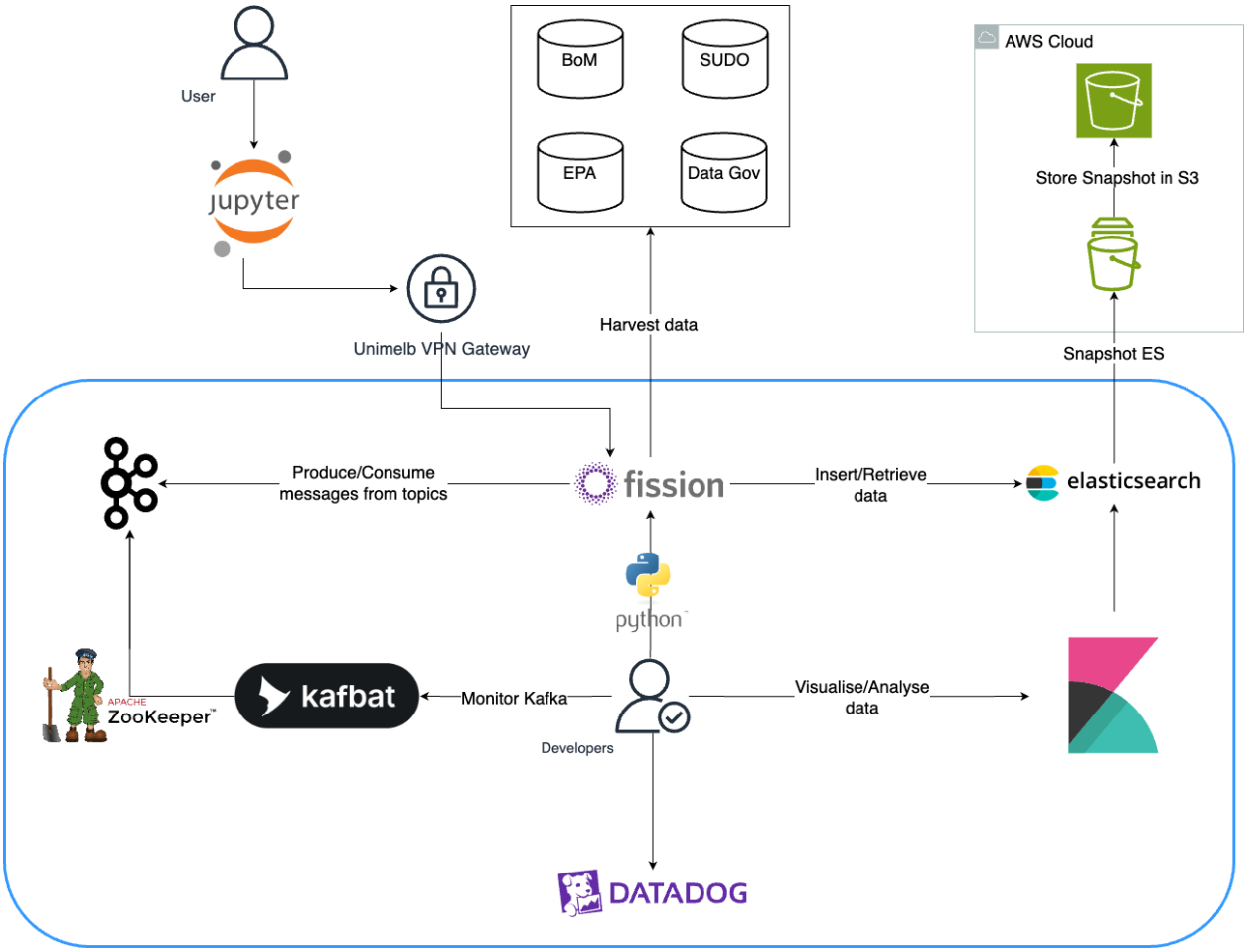


Figure 8: Flowchart

3.2 Fetching Static Data

3.2.1 Crashes

For our analysis, we chose to collect the data from a **SUDO** () database called *Data frame TAS DSG - Tasmania Crash Statistics (Point) 2010-2020.* The data frame lists and provides information on all road crashes that occurred in Tasmania between 2010 and 2020. To save space on our Elasticsearch database, we chose to collect only the following attributes: `crash_date` stored as a date type, `light_condition` stored as a keyword, `location` as a geo_point, and `severity` stored as a byte. When the value of severity is missing, we have set it to be -1 . For traceability purposes we have kept and documented the code used to upload the data from local csv files to Elasticsearch, it could be useful in case we had to re-upload the data or redeploy

Elasticsearch.

3.2.2 Crimes

In order to spotlight possible correlations between the weather and the the number of offences, we chose to collect the data from an official government database () called *Crime Statistics*. The data is composed of 10 data frames (one for each year) which count the number of offences per day by category, for the whole of Australia, over the last 10 years. The offence categories are organised into 3 levels of description. We collected and uploaded the following attributes to Elasticsearch: `description_1`, `description_2`, `postcode`, `suburb` all stored as keywords; `description_3` which is the most granular type stored as a text (as the number of characters in not bounded); `offence_count` as a float and `reported_date` stored as a date. For traceability purposes we have kept and documented the code used to upload the data from local csv files to Elasticsearch, it could be useful in case we had to re-upload the data or redeploy Elasticsearch.

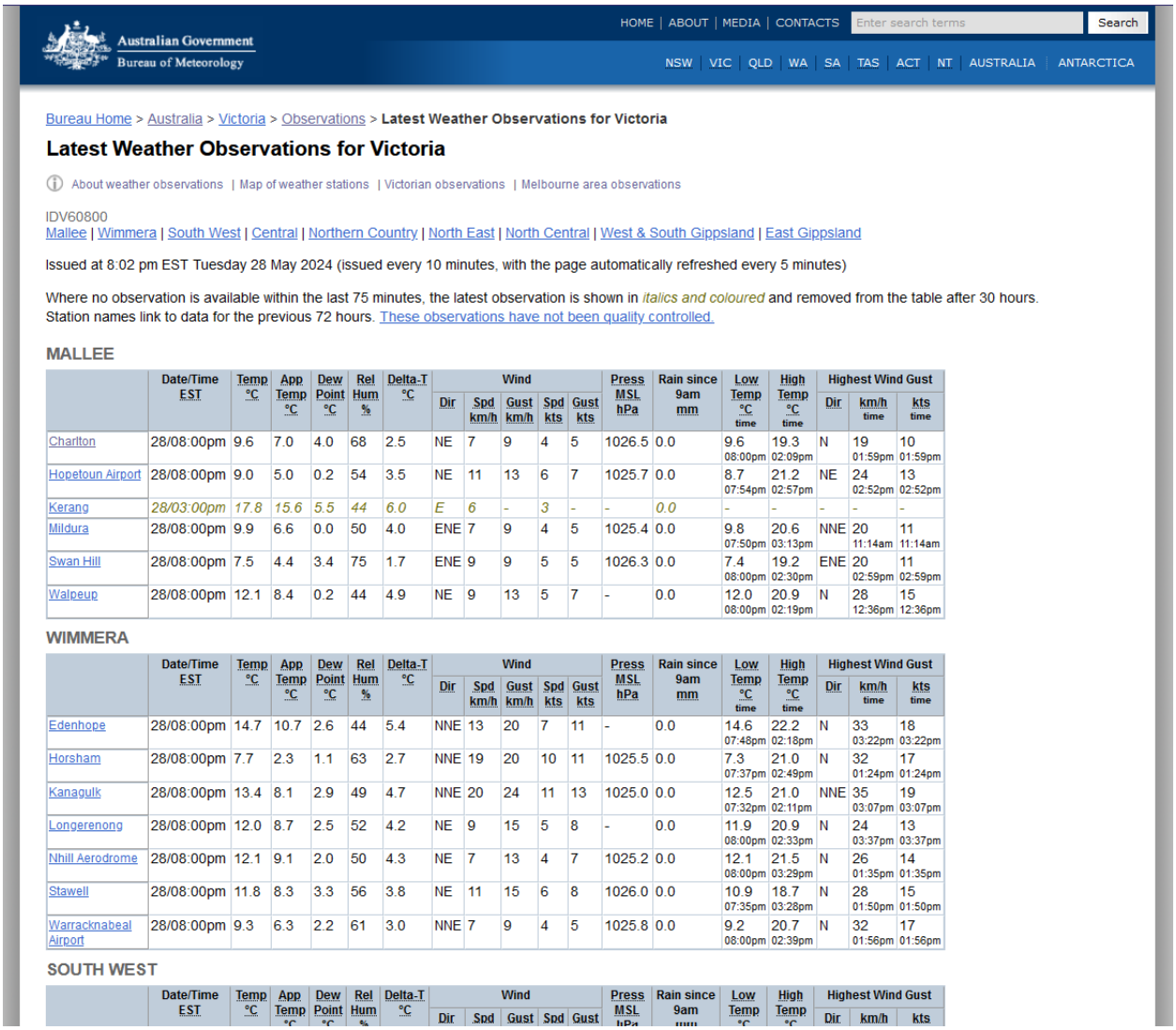
3.2.3 Weather

Data Source: For our project, we decided to use the weather across our three scenarios and the Australian Bureau of Meteorology (BOM) provided a simple interface to access this data. They provides extensive range of historical/live weather data, including temperature, precipitation, wind speed, and other meteorological variables. BOM offers weather data through their FTP server.
Data Collection Process: We develop a serverless Fission Python script to connect to the FTP server, navigate through the directory structure, and download the required CSV files. The extracted data including key variables such as temperature, humidity, rainfall, and wind speed is then converted as JSON messages to send to Kafka.
Data Storage: Fission Message Queue Trigger consumes the incoming messages from subscribed Kafka topic and then trigger the Fission function to preprocess the data before inserting it into Elasticsearch.

3.3 Fetching Live Weather

Two pipelines were established to retrieve weather data from the BOM: 1. Latest observations from a weather station 2. Hourly observations collected from all weather stations in Australia.

Although the BOM does not provide a free API service to request live data it does provide them through HTML and JSON eg , for each observation station for the previous 72 hours. These data are issued every 10 minutes, although the recency of the actual observations varies by station, with some metropolitan areas providing updates every 10 minutes, and other stations every hour.



Australian Government
Bureau of Meteorology

HOME | ABOUT | MEDIA | CONTACTS | Enter search terms | Search

NSW | VIC | QLD | WA | SA | TAS | ACT | NT | AUSTRALIA | ANTARCTICA

Bureau Home > Australia > Victoria > Observations > **Latest Weather Observations for Victoria**

Latest Weather Observations for Victoria

About weather observations | Map of weather stations | Victorian observations | Melbourne area observations

IDV60800

Mallee | Wimmera | South West | Central | Northern Country | North East | North Central | West & South Gippsland | East Gippsland

Issued at 8:02 pm EST Tuesday 28 May 2024 (issued every 10 minutes, with the page automatically refreshed every 5 minutes)

Where no observation is available within the last 75 minutes, the latest observation is shown in *italics and coloured* and removed from the table after 30 hours.
Station names link to data for the previous 72 hours. These observations have not been quality controlled.

MALLEE

	Date/Time EST	Temp °C	App Temp °C	Dew Point °C	Rel Hum %	Delta-T °C	Wind					Press MSL hPa	Rain since 9am mm	Low Temp °C time	High Temp °C time	Highest Wind Gust		
							Dir	Spd km/h	Gust km/h	Spd kts	Gust kts					Dir	km/h time	kts time
Charlton	28/08:00pm	9.6	7.0	4.0	68	2.5	NE	7	9	4	5	1026.5	0.0	9.6 08:00pm	19.3 02:09pm	N	19 01:59pm	10 01:59pm
Hopetoun Airport	28/08:00pm	9.0	5.0	0.2	54	3.5	NE	11	13	6	7	1025.7	0.0	8.7 07:54pm	21.2 02:57pm	NE	24 02:52pm	13 02:52pm
Kerang	*28/03:00pm*	*17.8*	*15.6*	*5.5*	*44*	*6.0*	*E*	*6*	*-*	*3*	*-*	*-*	*0.0*	-	-	-	-	-
Mildura	28/08:00pm	9.9	6.6	0.0	50	4.0	ENE	7	9	4	5	1025.4	0.0	9.8 07:50pm	20.6 03:13pm	NNE	20 11:14am	11 11:14am
Swan Hill	28/08:00pm	7.5	4.4	3.4	75	1.7	ENE	9	9	5	5	1026.3	0.0	7.4 08:00pm	19.2 02:30pm	ENE	20 02:59pm	11 02:59pm
Walpeup	28/08:00pm	12.1	8.4	0.2	44	4.9	NE	9	13	5	7	-	0.0	12.0 08:00pm	20.9 02:19pm	N	28 12:36pm	15 12:36pm

WIMMERA

	Date/Time EST	Temp °C	App Temp °C	Dew Point °C	Rel Hum %	Delta-T °C	Wind					Press MSL hPa	Rain since 9am mm	Low Temp °C time	High Temp °C time	Highest Wind Gust		
							Dir	Spd km/h	Gust km/h	Spd kts	Gust kts					Dir	km/h time	kts time
Edenhope	28/08:00pm	14.7	10.7	2.6	44	5.4	NNE	13	20	7	11	-	0.0	14.6 07:48pm	22.2 02:18pm	N	33 03:22pm	18 03:22pm
Horsham	28/08:00pm	7.7	2.3	1.1	63	2.7	NNE	19	20	10	11	1025.5	0.0	7.3 07:37pm	21.0 02:49pm	N	32 01:24pm	17 01:24pm
Kanagulk	28/08:00pm	13.4	8.1	2.9	49	4.7	NNE	20	24	11	13	1025.0	0.0	12.5 07:32pm	21.0 02:11pm	NNE	35 03:07pm	19 03:07pm
Longerenong	28/08:00pm	12.0	8.7	2.5	52	4.2	NE	9	15	5	8	-	0.0	11.9 08:00pm	20.9 02:33pm	N	24 03:37pm	13 03:37pm
Nhill Aerodrome	28/08:00pm	12.1	9.1	2.0	50	4.3	NE	7	13	4	7	1025.2	0.0	12.1 08:00pm	21.5 03:29pm	N	26 01:35pm	14 01:35pm
Stawell	28/08:00pm	11.8	8.3	3.3	56	3.8	NE	11	15	6	8	1026.0	0.0	10.9 07:35pm	18.7 03:28pm	N	28 01:50pm	15 01:50pm
Warracknabeal Airport	28/08:00pm	9.3	6.3	2.2	61	3.0	NNE	7	9	4	5	1025.8	0.0	9.2 08:00pm	20.7 02:39pm	N	32 01:56pm	17 01:56pm

SOUTH WEST

	Date/Time EST	Temp °C	App Temp °C	Dew Point °C	Rel Hum %	Delta-T °C	Wind					Press MSL hPa	Rain since 9am mm	Low Temp °C	High Temp °C	Highest Wind Gust		
							Dir	Spd	Gust	Spd	Gust					Dir	km/h	kts

Figure 9: Example of observation stations by region

Both pipelines utilise the JSON endpoints supplied by BOM to retrieve observation data. To support the retrieval of data from these endpoints a lookup table was compiled that included important observation station metadata, such as its longitude and latitude, as well as the JSON URL to request the data. This lookup was put together by scraping both the html pages for the station as well as the data supplied through the JSON endpoint as neither include all the required information. State and region were included to support batching of requests. An example of regions in Victoria is shown above in Figure 9, which shows the observation stations located in the regions of Malle and Wimmera. A comparison was made with the prior weather station data obtained through the FTP service, however there were significant differences found. Some of these differences included different ID conventions, names and changes over time (eg portable stations). As a result this lookup was added as a new index in Elasticsearch as ‘current_bom_stations‘.

3.3.1 Current Observations

A RESTful endpoint to request the most recent observations for a specified station was created to support analytics. An overview of the pipeline is described in Figure 10. This pipeline was designed to be lightweight and flexible to multiple use cases. It allows the user to request data using one of three parameter combinations: station name, station ID or latitude and longitude. If a user supplies the station name or id, this is searched against the current_bom_stations lookup in Elasticsearch; if there is a matching record the JSON URL for the station is returned and a request is made to BOM for the latest data. Alternatively if a user provides geocoordinates, the function sends a GET request to the /station API to determine which station is the closest, which in turn is used to request data.

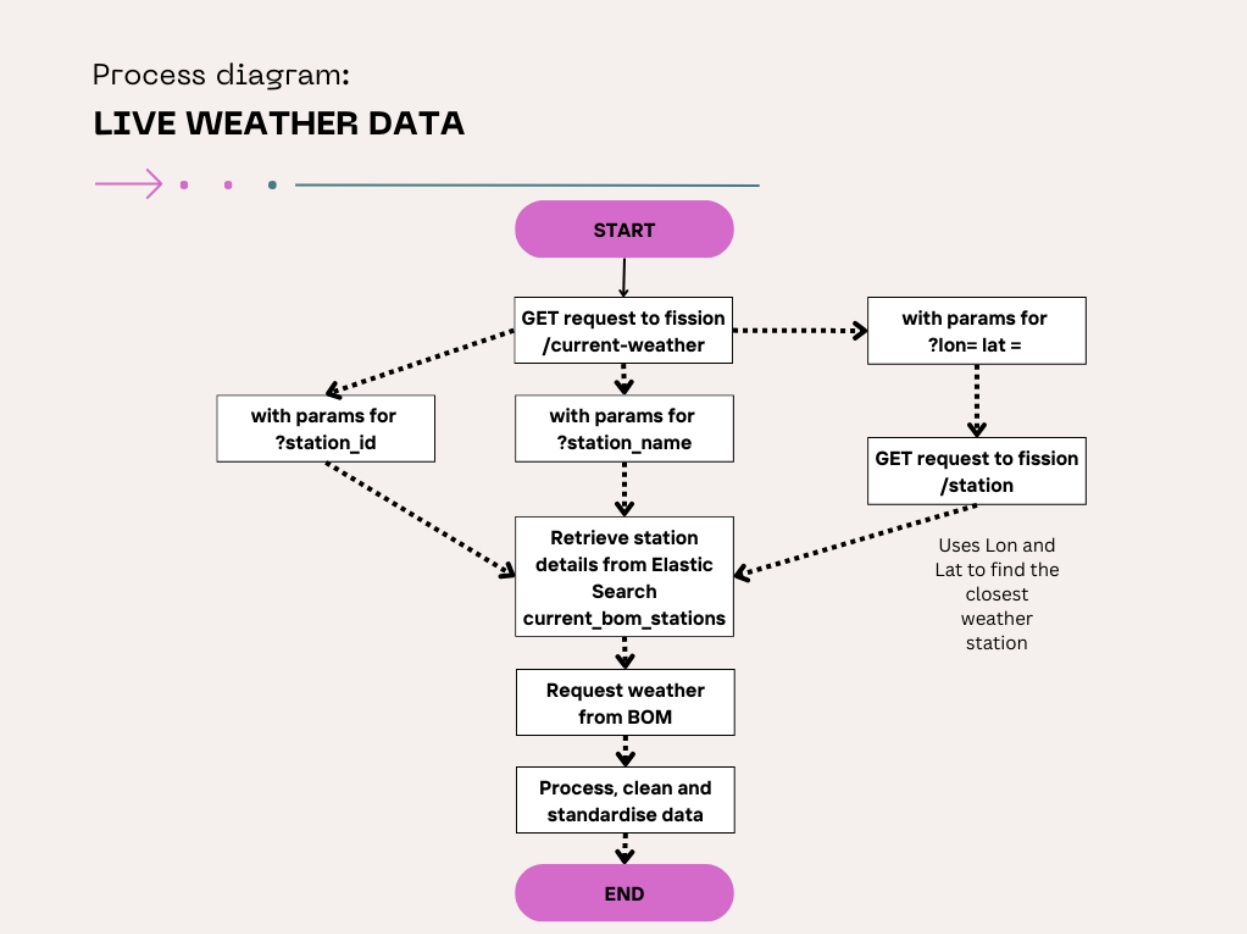


Figure 10: Diagram of the live weather pipeline

Once the data is retrieved from BOM, a cleaning function is applied to standardise the data before returning to the the user. A decision was made in this process to combine the retrieval and cleaning within the one monolith function, rather than to separate into different services. This is because the data retrieval and processing requirement is small, and the additional network overhead would likely slow down performance for end users.

There were some challenges designing the pipeline to handle different types of network failures. The tenacity package was utilised to handle one off or short term errors by ensuring a number of retries, along with a

back-off which helped to minimise errors returned to users due to these issues.

3.3.2 Hourly Weather Collection

The hourly weather pipeline utilises a combination of different fission functions and endpoints to collect and ingest weather observation data from the BOM. An overview of the pipeline is described in Figure 11. A fission timer is scheduled to trigger the 'hourly-weather-obs' function each hour. This function looks up a list of states and regions, and sends a GET request to the fetch-weather-obs endpoint for each region. As there are nearly 800 observation stations in Australia, This design decision was made to break up the total number of requests whilst also trying to balance the network overhead within our system.

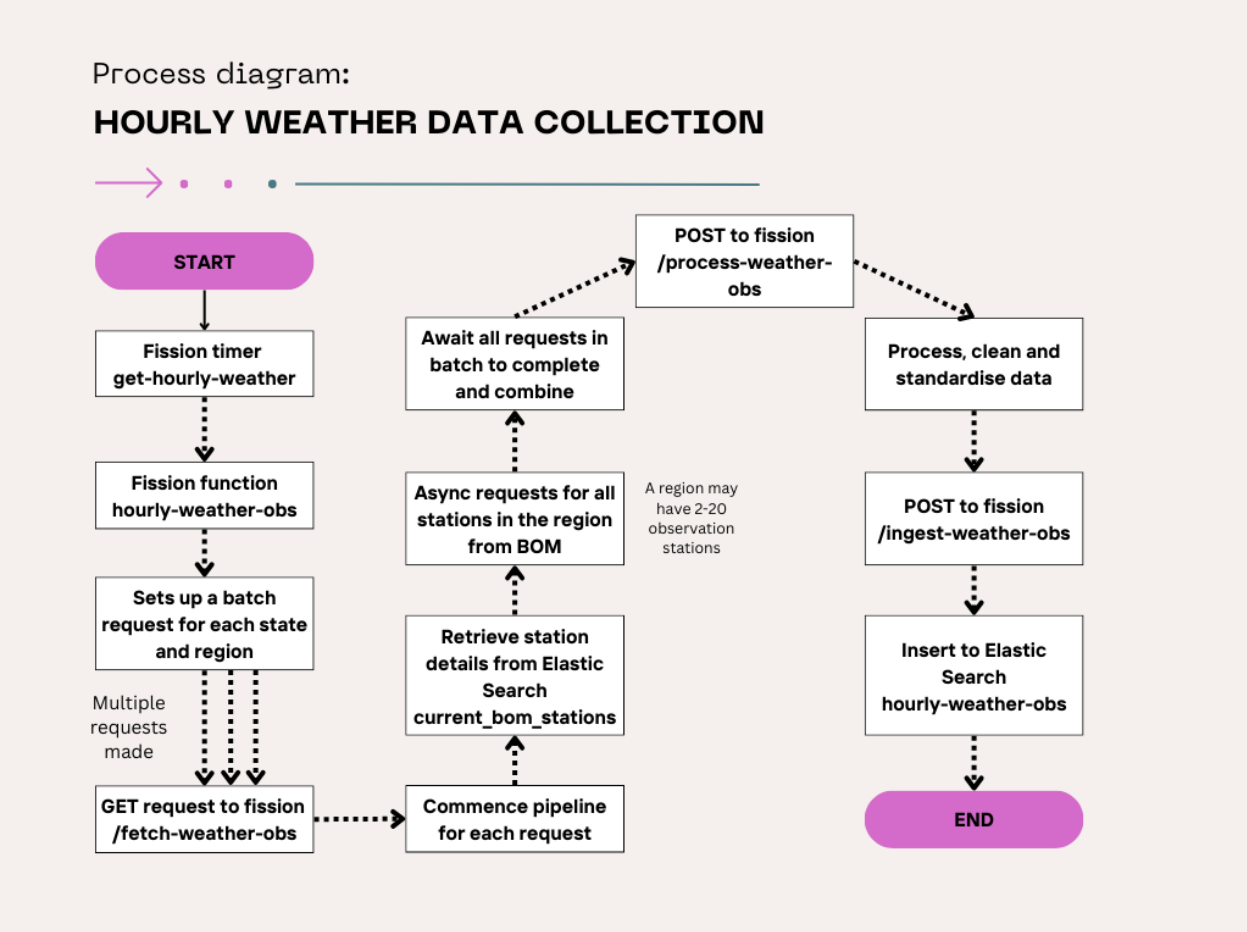


Figure 11: Diagram of the hourly weather collection pipeline

For each region the relevant station details were extracted from our lookup in Elasticsearch. Then observation data for all stations in the region where requested asyncrounously using the aiohttp and asycnio packages, and the results awaited on and combined into a single structure. As with the live weather, the tenacity package was also utilised to help handle network failures. Special handling was included to allow for when one or more stations in a region could not return data, in these cases the observation data for the relevant station was set to missing and the rest of the function was allowed to continue without raising an error.

Once all station data for the region was retrieved, the data was sent through as a POST request to the

/process-weather-obs endpoint. Following cleaning, these data were sent as A POST request to the /ingest-weather-obs endpoint. The processed data is then inserted into Elasticsearch in the hourly-weather-obs index. This service-oriented architecture leverages fission scalability and minimises and isolates points of failure in the pipeline.

There were a number of challenges encountered with this pipeline which evolved from an early monolithic function to a service-oriented architecture. To begin with, there were a number of JSON lookup files required to drive the process, this necessitated creating a package for the fission functions rather than just a single python script. Our initial python environment was not configured to build multi-file packages and a new python environment needed to be created to facilitate this. Another significant challenge was managing number of requests to BOM to prevent IP blocking or other limiting. At various times in the project requests to BOM would fail, returning 403 forbidden, indicating that restrictions had been placed on our IP, this may have been due to other projects also making multiple requests. As part of the batching process a wait time was included in between each batch to ensure that our overall requests per minute were reasonable.

3.4 Fetching Air Quality

To conduct our study on air quality, we utilised data provided daily by the **Environment Protection Authority of Victoria** (EPA). The dataset includes measurements of various particles and gases in the air (such as NO_2, $PM10$, or the number of particles per cubic meter), averaged over periods ranging from 20 to 40 hours, along with the associated locations of the measurement stations, which are all situated within the state of Victoria.

After creating an account on the official API, we are able to request all data produced by the measurement stations over the last 48 hours. Given the fixed number of stations and the limit on the number of measurements per day per station for each particle/gas, the total number of documents requested does not exceed 400, allowing us to make a single API request to the **EPA**.

To prevent the loss of the retrieved data in case of processing errors, we store the latter in a message within the *airquality* Kafka topic, appending a uniquely generated *message_id*. This message serves as a buffer, enabling us to have repeated access to the data in the event of an error, and it can also serve as a log of all data fetched from the EPA API.

To handle potential errors during the buffer message upload, we catch any exceptions returned by the `KafkaProducer.send()` method. If no exceptions are caught, a confirmation message is sent, including the same *message_id* as the buffer message and a 'Success' status.

Subsequently, a second Fission function is triggered upon the upload of the confirmation message. This function retrieves the most recent messages from both the *airquality* and *airquality-uploaded* topics, verifying that the buffer and confirmation messages share the same *message_id* and that the confirmation message indicates success. Without the confirmation message, an upload error to Kafka would result in the retrieval of the last message in the buffer, hence the data from the day before, which would harm the integrity of our data.

We observed that the EPA data was sometimes messy, and given the overlapping time ranges of the measurements, it is possible to encounter duplicate measurements for the same location and particle/gas (when one time range is included into the other). To address this, we retrieve recent data from Elasticsearch (ES)—specifically, data ending after the start of the oldest data retrieved from the EPA API. And then we compare the time ranges for each particle/gas and location to ensure the data is not already present in our

database, discarding any duplicates.

Finally, we convert the data types to ensure compatibility with the index mapping of our *airquality* index in ES. The processed data is then uploaded, and a new confirmation message with 'Reset' status is produced to prevent future duplicate uploads into Elasticsearch.

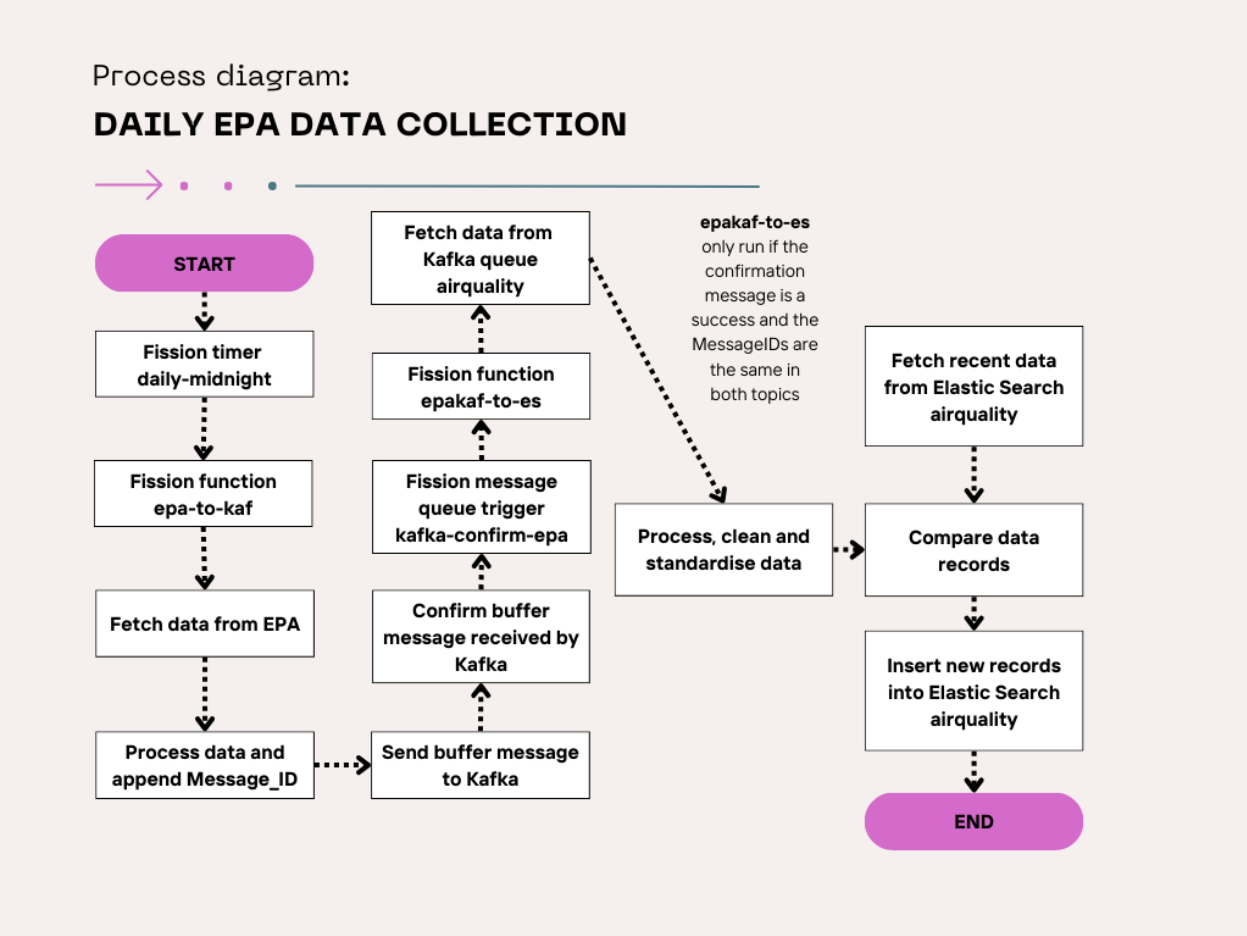


Figure 12: Diagram of the EPA live data collection process

We encountered errors with the data types we initially chose in the Elasticsearch index mapping. Thus we made a script to download all the data locally to a csv file. And another one to upload the same data back to Elasticsearch. We kept both scripts and documented them for traceability purposes.

3.5 API

The base endpoint for the RESTFUL API is http://172.26.135.52:9090. Each endpoint is a Fission function that uses a Flask request. The Fission function is attached to a httptrigger and is mapped to a url so it can be called using the HTTP protocol. Only GET requests are accepted. The API endpoints contain error handling to improve the user experience. If will check for bad parameters and return a 400 error in this case. An example of bad parameters could be a radius that is a string not a number. Any internal errors with Elasticsearch are also caught and a 500 error is returned.

3.5.1 Stations API

`/stations`

This API endpoint simply returns a list of all the stations used for the historical weather data collection. There are no parameters.

3.5.2 Closest Station

`/stations/LONGITUDE/LATITUDE`

This endpoint is used to find the closest station to a longitude and latitude coordinate. This function utilises the most recent list of stations that is used for daily weather data. This is because stations have moved since the historical weather data was collected.

3.5.3 Weather API

`/weather/STATION_ID/START_YEAR/END_YEAR`

This endpoint is used to get all the weather reported historically from a station in the time frame of the start year to end year. Internally, an Elasticsearch query is made to the weather index for the station ID filtered by the range of years.

3.5.4 Live Weather API

`/current-weather` with the parameters of `?lon=LONGITUDE&lat=LATITUDE`, `?name=STATION_NAME`, or `?id=ID`

This function is used to get the current weather from a weather station. For flexibility, the user must pass a station id, station name, or station coordinates. The data is returned as a dictionary to the user.

3.5.5 Hourly Weather API

`/fetch-weather-obs` GET with the parameters of `?state=STATE&region=REGION`

`/process-weather-obs` POST

`/ingest-weather-obs` POST

This series of endpoints enables a data pipeline to extract weather data for weather stations within a specific region, and to process, clean and insert these data into Elasticsearch.

3.5.6 Crash API

`/crashes/STATION_ID/SIZE/RADIUS_KM`

This endpoint allows users to get all the crashes that occurred in a given radius in km of a weather station. The user must specify the station id number, the radius, and the number of results to return in each batch. The function returns the status, data, and a token to be used to fetch the next batch of data. Internally, the token is actually a scroll id used by Elasticsearch. This token can be passed to the stream endpoint to get the next data, the crashes endpoint is only meant to start the transaction. If the token is set to "END", that means there is no more data.

3.5.7 Crime API

`/crime/STATION_ID/SIZE/RADIUS_KM`

This function behaves very similarly to the crashes endpoint, but except it fetches crime data not crash data. Again, it returns a token that can be passed to the stream API to continue the transaction. As with the crash API, this endpoint is only used to start a data fetch.

3.5.8 Air Quality API

`/epa/STATION_ID/SIZE/RADIUS_KM`

This endpoint is again similar to the crash and crime functions. It accepts the station id, batch size, and radius of search to fetch epa data. Currently, epa data is only streamed for Victoria, but this API provides a flexible interface that can return data for the rest of Australia as well. Again, it starts a transaction and returns a token that can be passed to the stream function.

3.5.9 Stream API

`/stream/TOKEN`

As mentioned, this endpoint is used to continue a transaction for crash, crime, and epa data. A token is passed in which is actually an Elasticsearch scroll id. This is used to fetch the next batch of data. This API returns another token to continue the pagination. Once there is no more data, the token is set to "END"

3.5.10 Models API

The models API is used to get predictions from the different models we have trained during our analysis. For performance and speed we have chosen to directly store the trained models in a Fission package **models-API-pack** based on our environment **python-es**. Models are stored as pickle files, and the addition of a new model is very easy as all is needed is the addition of the file to the **models_API** folder and then call `make update`.

To make a prediction, the user simply has to get request from the following URL:
`http://172.26.135.52:9090/models/MODELNAME?1.1,2.2,3.3`
With MODELNAME being the name of the model (stored as **MODELNAME.pkl** in the Fission package), and the float values of the predictors separated by commas as parameters. Here for example, a prediction is requested with [1.1, 2.2, 3.3] as values for the predictors.

If the user wants more information on the model than simply predictions, they can request the coefficients and intercept in the case of a linear regression, by not specifying any parameters:
`http://172.26.135.52:9090/models/MODELNAME`
Here the response of the request will be the coefficients and intercept of the MODELNAME model.

3.5.11 EPA Data Collection

We implemented tests for the collection of EPA data to ensure resilience against unexpected missing keys or data types, in order to maintain execution for valid data.

Additionally, we developed tests for comparing the time ranges between the data retrieved from the EPA and our pre-existing data in Elasticsearch.

3.5.12 API Tests

We implemented tests for our API functions to ensure they respond as expected to requests with incorrect parameter counts, invalid parameters, or valid parameter sets. Indeed, we designed custom error responses to help users identify and resolve issues with their specific requests more efficiently.

4 Instructions

As touched on, the frontend was designed with a set of Jupyter Notebooks that can be accessed in the frontend directory of the repository or at on the UniMelb network. If prompted for a password, use "cloudcomp". If you choose to run the files locally, simply run `jupyter notebook` in the frontend directory to open the interface. The notebooks can then be broken down by scenario.

4.1 Air Quality Models

The user facing dashboard for the air quality models is called `EPA_Dashboard.ipynb`. There are only two code blocks that must be run in this notebook. The first imports required packages and loads helper functions. The second code block creates a map for the user to select a location. A user can click on a location to drop a marker on the map. To change locations, the user can then drag the marker. The service will then fetch the current weather at the station closest to the selected point. It will then use this weather to predict the amount of particles in the air by calling the models API. Then, if data is available from the last year, epa metrics near that station will be loaded. Currently, only locations inside Victoria support this and not all areas of Victoria are in the 50 km range of an EPA station. This radius was chosen arbitrarily.

The system will notify the user if data was not found, but if it was a scatter plot will be shown for the different weather metric vs particle concentrations in the air. Currently, the temperature, humidity, rain, and wind speed are considered. A dropdown allows the user to toggle between independent variables. Under this, a correlation matrix is displayed with all of the variables to examine their interactions.

There is also a notebook called `epa_model_trainer.ipynb` that is used to train the epa model. This is a backend interface that was used to train the model, but it does display a similar scatter plot and correlation matrix. However, it uses all the EPA data, not just for one specific weather station. It also saves a pickled linear regression model to the current directory. All that needs to be done is that all cells are ran in order and the model and graphs will be created.

4.2 Crashes Models

For this model, we also offer two Jupyter Notebooks. The first one is called `weathervscrashmodel.ipynb` which is for users interested in how data was retrieved and cleaned, Exploratory Data Analysis (EDA) process and model training. First, it collects all weather stations with crashes around within a given user input radius. After that, based on *crash_date* and *Station ID*, it retrieves the relevant weather conditions. Then, the cleaning process and EDA is conducted to prepare for model training step. The trained models are then saved as *pkl* file and pushed to the backend to serve the Fission *models* API.

The second Jupyter Notebook, `weathervscrash_peer.ipynb` is for interactive mode with a map, a slider to choose radius, and a marker to pick the preferred location. The user should decide the radius before moving the marker. The heatmap of crashes for the closest weather station and trendline plot will show up. Depending on the radius size and the amount of crashes data pulled from the backend, the waiting time

might vary. The user will be notified with the specific message if an error happens. The final code cell in this notebook is to make a prediction based on current weather of the chosen location.

All of the code cells need to be run one after the other. Please raise a Git Issue if you find any further errors.

4.3 Crimes Models

The user facing dashboard for the offences vs. weather models is called `Crime_Dashbord.ipynb`. The user runs the blocks of code one by one and chooses a location and a radius to conduct his study. Then gets several plots for the trends of the offences that happened in his area during the last year. Then an analysis, via a linear regression, is conducted to determine if indeed in his area there is a correlation between the weather predictors and the offences. Finally the user can predict the number of offences against persons that will happen today in the whole state of Victoria. The prediction calls, via the API, a pre-trained model on the data from the whole state over the last 10 years.

5 Results of the Data Analysis

5.1 Air Quality vs Weather

Using the data obtained from the EPA, we were able to examine the relationship between weather and air quality. Specifically, we chose to focus on particle concentrations in the air measured in micrograms per cubic meter. For predictors, temperature, wind speed, rain, and humidity were used. To train the model, first a list of all the stations for historical weather was obtained. Then, for each station all the EPA data within 50 km was collected using the REST API. The data was then cleaned to remove repeat rows. This was done because sometimes the 50 km search radii around weather stations overlapped. To avoid duplicates, only one weather station may correspond to one EPA station. Data formats were also fixed and data not related to particle concentrations was removed.

Next, the weather data for each weather station that had nearby EPA data was collected from the REST API. The historical data only had maximums and minimums for the temperature and humidity so these two values were then averaged to get an overall value. Weather data with temperatures less than -50 degrees celsius were then removed as this is indicative of invalid readings. For wind speed, humidity, and rainfall, any row with values less than 0 were dropped for the same reason. Finally, the two data sets were inner joined on their dates and station ids. With this data, a linear regression model was then fit for the particle values versus the temperature, rain, wind speed, and rain.

As shown in figure 13, the linear regression model was not the strongest approach to estimate particle concentrations in the air. The most significant correlation was with wind speed at -0.17. This makes sense because higher wind may blow particles away, however this correlation coefficient is still close to 0. Temperature, humidity, and rain had correlation coefficients of -0.05, 0.03, and 0.05 respectively. Since these values are close to 0, it is shown that they are not significant predictors of particle concentrations in the air. Further, it should be noted that a downside of the linear regression model is that negative values can be predicted which don't make sense in the context of particles. In the future, it would be interesting to explore more models but the linear regression model provided a simple demonstration of how inferences could be made on the data.

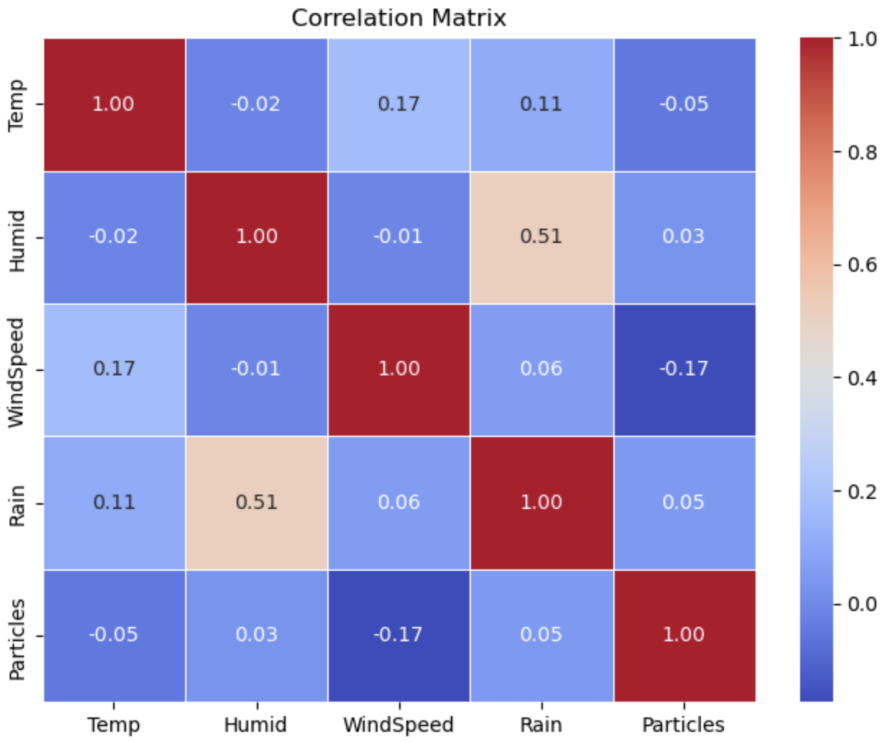
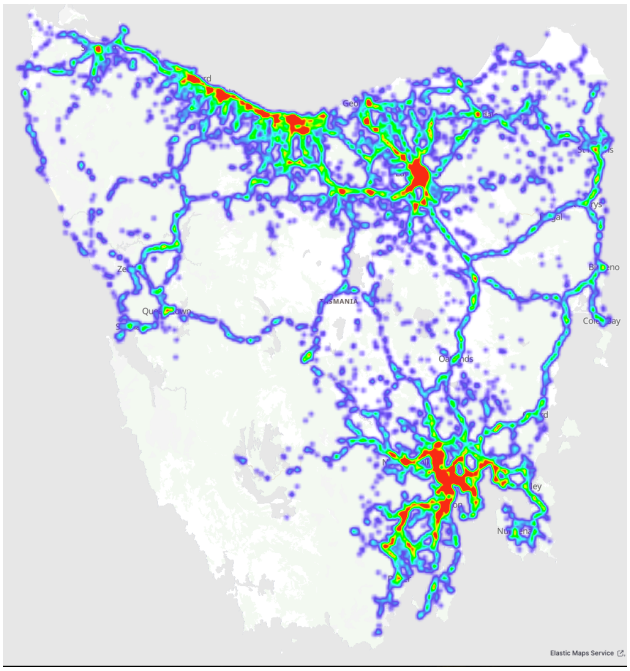


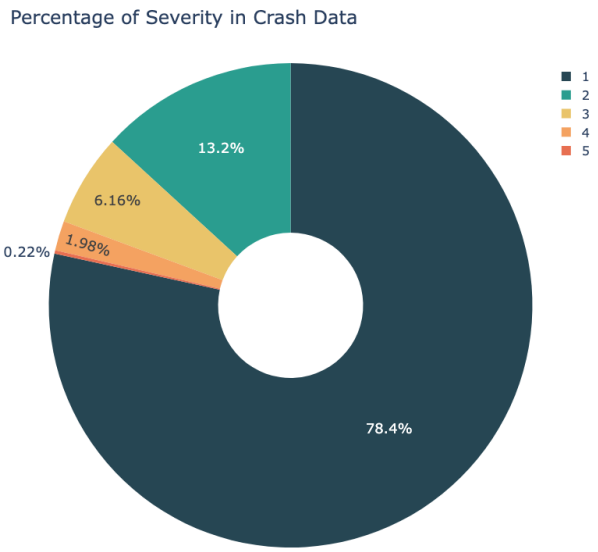
Figure 13: Correlation matrix for temperature, humidity, wind speed, rain, and particle concentration

5.2 Crashes vs Weather

Scenario description: Our project investigates the relationship between weather conditions and traffic crashes. By analyzing historical weather data and crash records, we aim to identify correlations between variables like temperature, humidity, wind speed, and UV index and crash occurrences. These insights will help develop predictive models and enhance road safety strategies.



(a) Heat map of crashes in Tasmania



(b) The percentage of crash severity in Tasmania

Figure 14: Visualization of crash data in Tasmania

The heatmap of Figure 14a shows that the crash density is high in large cities, such as Hobart and Launceston, and along highways which indicates that these are candidate points for targeted safety measures. Crash frequencies are relatively moderate on secondary roads and in smaller towns, potentially indicating that infrastructure improvements could have an impact. Areas of lower population density result in fewer crashes, indicative of either superior existing safety or simply less traffic.

Figure 14b further illustrates that within strong data on crash severity, nearly four out of five crashes are minor while few serious crashes occur. Although most crashes are mild in magnitude, the fact that moderate and serious crashes account for high proportions means that specific safety interventions should be developed to decrease all crash incidence in order to improve road safety.

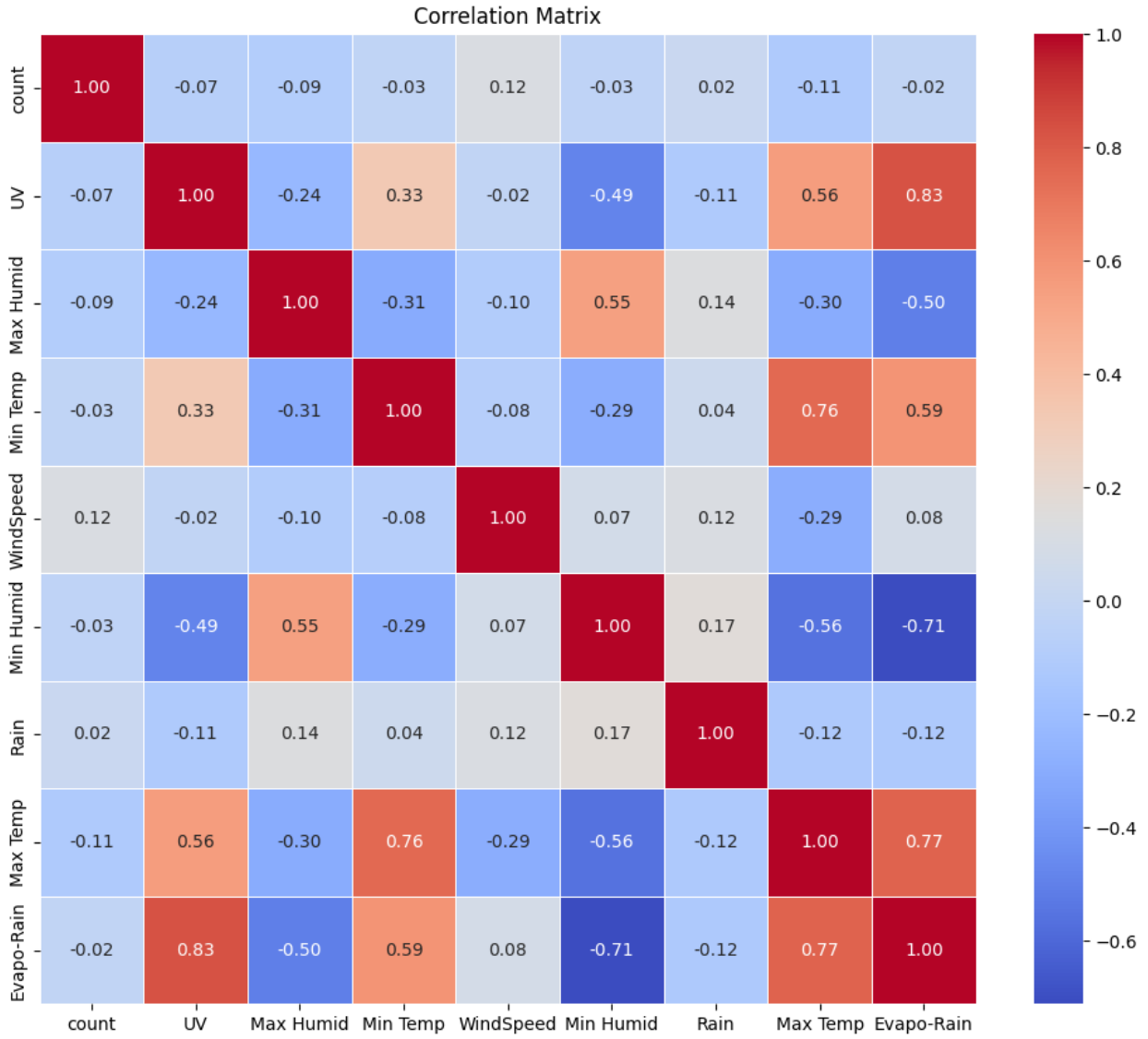


Figure 15: Correlation matrix of crashes frequency vs. weather

The correlation matrix analysis indicates that the weather variables are either very weakly related to the crash counts, or not relevant at all. In particular the UV Index (-0.07), Max Humidity (-0.09) and Max Temp (-0.11) all have weak negative correlations with counts (i.e. slightly less crashes in these conditions). Wind speed (0.12) is more weakly related with the target value, which suggests that higher wind speeds lead to a small increase in crashes. The weather features do not seem to have a direct influence on the number of accidents so we choose not to train any models on this.

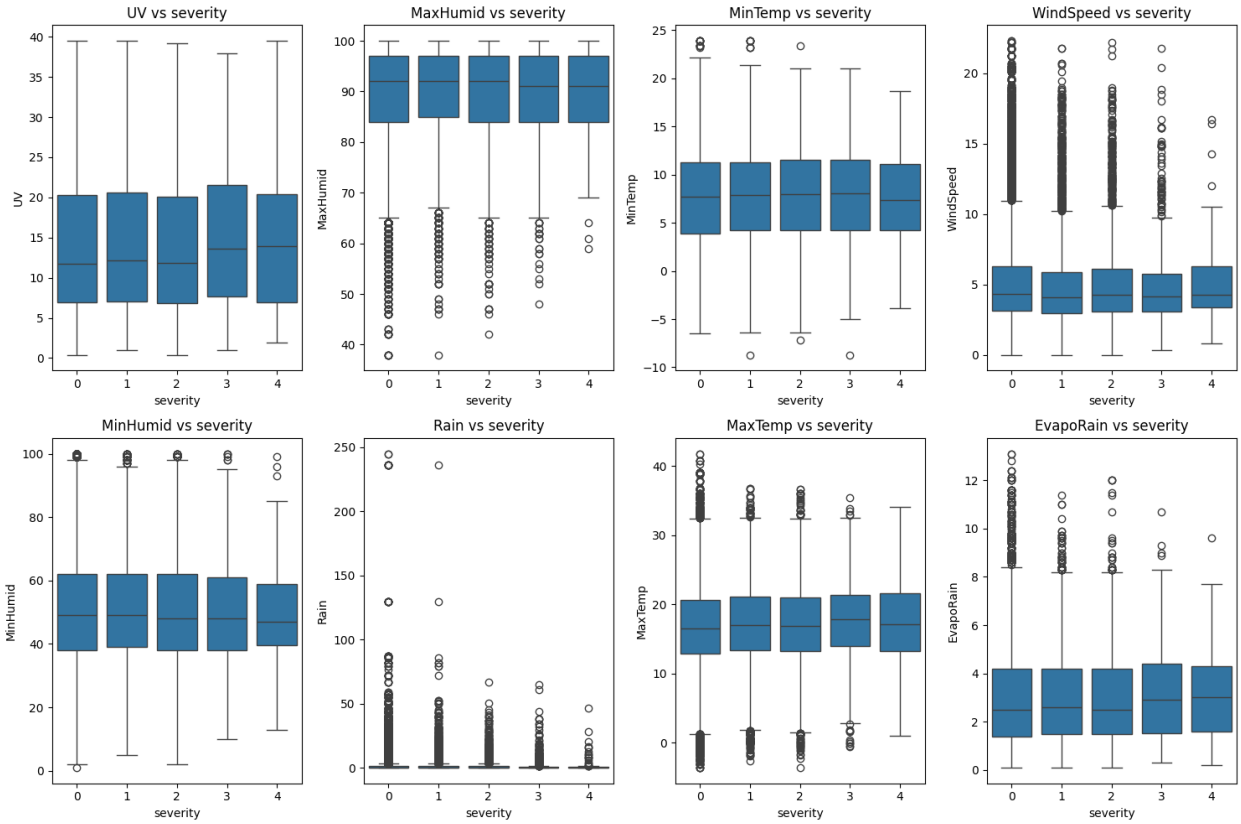


Figure 16: Correlations crashes severity versus weather conditions

The box and whisker plots indicate that some weather variables have the same distribution across crash severity levels, which implies weak visual associations. Nevertheless, Kruskal-Wallis H test results established significant differences in UV index, minimum temperature, wind speed, rain, maximum temperature and evapotranspiration for the different severity levels despite their visual similarity. This tells us that there may not be a strong visual correlation even if statistical analysis revealed notable underlying distinctions pertinent to predicting.

We then used several machine learning algorithms to predict traffic crashes using these weather conditions as inputs features. These algorithms included Logistic Regression, K-Nearest Neighbors (KNN), Decisions Tree and Random Forest classifiers. Of all these models Logistic Regression gave the most accurate prediction with an accuracy of 0.79. Therefore it can be concluded that Logistic Regression is best suited for this dataset.

5.3 Crimes vs Weather

As the second and the third level description are way too granular we focus our study on the number of offences committed per day for specific regions, grouped by the first level description, which can take only two values: 'Offences against the Persons' and 'Offences against Property'. In the state of Victoria, the overall trend of the number of offences against persons or against property seems to be constant with respect to time.

After grouping the offences by their date and their closest weather stations, we can fit a linear regression

model, with the number of offences against persons (resp. against properties) as the response, and the numerical variables of the weather as the predictors.

We observe that the effects of the predictors are very low in general. This was expected as the number of offences per day is already low, and there is no any evident causation link between the weather and the number of offences in an area.

However the coefficient associated with Evapo-Rain is orders of magnitude higher than the rest. When we compute the correlation it is also higher than for the other predictors as well with a value of 0.0735. And we can compute the p-value = 0.009702 to finally reject the null hypothesis that there is no correlation between the answer and the predictor. Indeed the number of points studied being quite large (18 622 to be precise), a weak correlation is sufficient to assert that there is a correlation between the two variables.

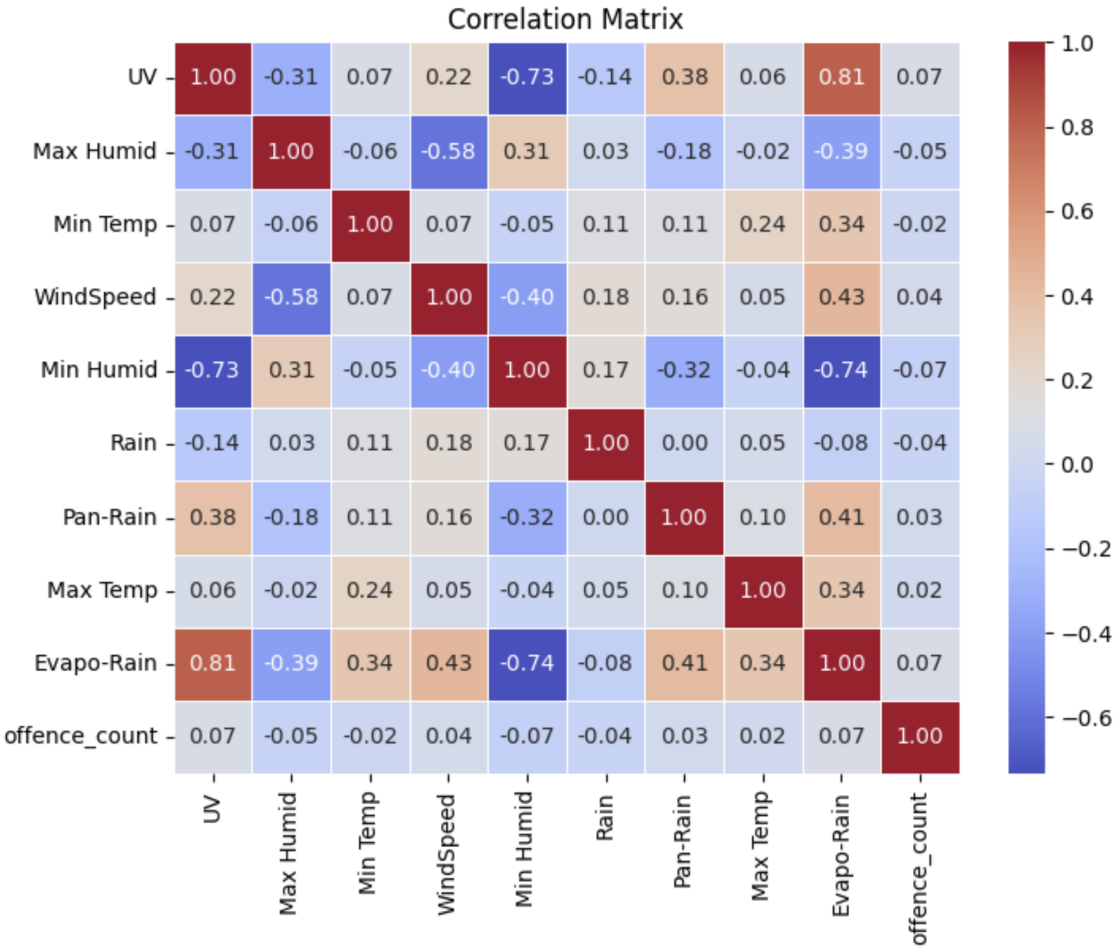


Figure 17: Correlation matrix of crimes against persons frequency vs. weather

We obtain similar results with the offences against properties, which reassures us about the presence of a correlation between Evapo-Rain and the number of offences, and not a statistical coincidence. Intuitively, we can think that after the rain, when the weather is more enjoyable again, more people go out in the streets, thus slightly increasing the number of offences, however further studies would be needed to prove a causation link.

6 Discussions

6.1 Single Point of Failure / Cluster Resilience

All our major services: Elasticsearch, Fission, Kafka are deployed across several nodes. Elasticsearch is in dual master mode and all indexes have at least one replica, so that if one node fails the other node still provides stable service to other services. And additionally we separate the nodes by workload of Elasticsearch and Fission functions in order to prevent resource preemption in one node during large data throughput. Fission functions are designed to be terminated and spawned frequently across different nodes, so it will not cause a long down time if one node fails. However, our Fission internal services like executor, router have no replica. This means if the node where these internal services is down, there will be a longer down time to setup these services on the other node(that happened to us during the MRC issue). Similarly in Kafka we only have one broker node which is also the controller node. In a single point of failure Kafka can recreate its pods on other node and mount the existing persistent volume, but during the recreation Kafka is not accessible.

In terms of our cluster's infrastructure which includes 1 master node and 4 worker nodes, the work load on worker node can be migrated to other nodes in case of single point failure, but if the master node failed all services which need to access Kubernetes API will get into trouble. Services like Elasticsearch will still be alive because it does not rely on the Kubernetes API to monitor the resources in the cluster. But services like Fission executor will fail because it actively queries kube-apiserver to monitor Fission CRDs(Custom Resource Definitions).

To improve resilience and prevent single point of failure, on the cluster's layer we should deploy a multi-master architecture, which includes multiple master nodes and a load balance between the worker nodes and the master nodes to distribute the request to api-server on each master node. Also on the services' layer, we should enforce every major service to have one or more replica to avoid down time.

6.2 MRC Latency Issue

We are one of the groups that affected by the MRC issue. The major behavior of this issue includes everlasting Fission pods restarting and slow responses from kube-apiserver. We were one of the earlier groups to spot and report this issue and we managed to partially fix this issue on our own with staff assistance.

6.2.1 Issue Localization

1. In the end of April some group member found the Fission functions were slow or not responding during data ingestion. Yue thought it was because a resource preemption between function pod and the Elasticsearch pod. So he specified nodeSeletor for both to separate their node allocation.
2. In the early May we still experienced a significant down time of Fission function. In the investigation Yue found the pod strimzi-cluster-operator CPU usage to be unusual so he raised the ED post.
3. Later on, we realised Fission was down not because of resource preemption but resulted from the restart of the Fission executor, which recreates every function pods during restarts.
4. Another night, since we observed that the node where Fission function is running on has spikes of disk latency compared with the other nodes, we decided to migrate that node. We created a new worker node, moved workloads from the old node to the new one, and evicted the old node.

5. The situation remained still after the node migration, the new node also had spikes of disk latency. This cause us to realise the problem is on the master node but not the worker node. We use node-shell to log into master node and spotted that more than 80% CPU is spent on I/O wait and the etcd process which acts as the distributed key-value storage of Kubernetes and is in I/O waiting status most of the time. It is the etcd I/O latency that causes all services frequently access kube-apiserver to be unstable. The metrics on Datadog also support our findings. After each high I/O usage there is a restart. Followed after that, our last failed attempt was to reboot the master node

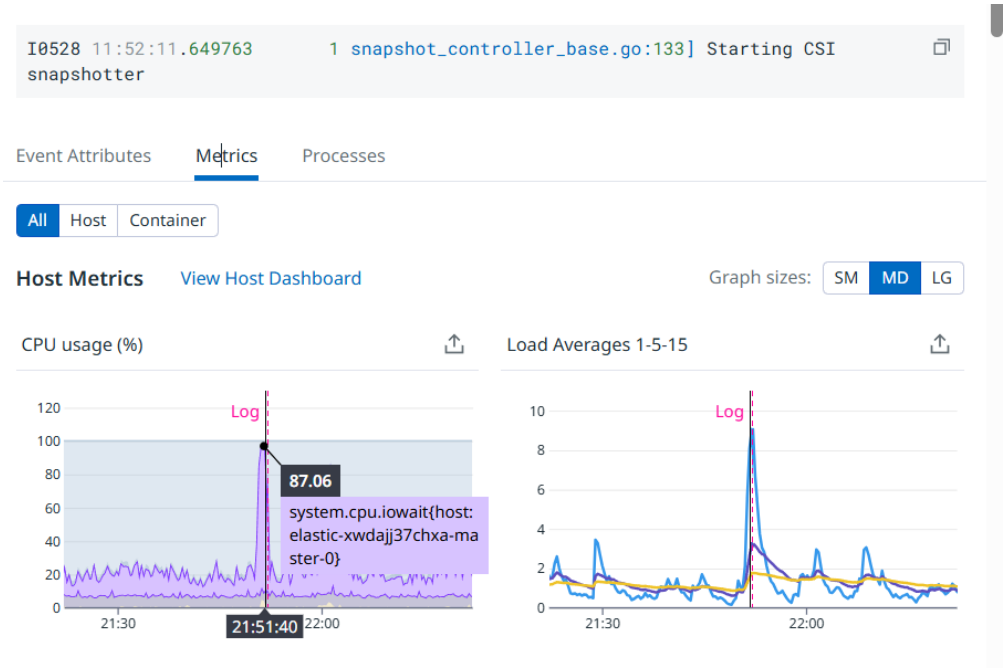


Figure 18: MRC issue

6.2.2 Final Solution

Our final solution is to completely recreate the cluster. Before that we had all our infrastructure deployment and services deployment packed up in makefile scripts and we created snapshots for Elasticsearch data. Therefore, we were quickly back online within one night with all services deployed and data restored. The issue is not entirely fixed since services like cinder still keeps restarting, but our workload services like Fission executor barely restart. The I/O usage on master node dropped from 80% to 40%, and we can still observe some spikes of disk latency. The root problem may not be fixed because it is probably close to the hardware layer. Maybe the saver to our issue is that we use 2 cores in the new cluster instead of 1 core in the old cluster so that we have an extra core free to respond to kube-apiserver requests when the other core is stuck in I/O.

6.2.3 Suggestion

1. Setting up scheduled snapshot policy on every data storage, in case of a cluster failure. 2. Setting up monitoring tools like Datadog. Because MRC does not provide some basic metrics like CPU usage, network throughput makes it very hard to distinguish to incidence.

6.3 Other Limitations

In this section, we delve into the limitations encountered during our coding project and provide suggestions to address these challenges. By discussing the problems we faced, we aim to highlight areas where improvements are necessary and offer practical recommendations for overcoming these issues.

6.3.1 MRC Limited functionalities

Limitation: MRC is quite limited in some basic functionalities compared to well-known public cloud platforms, such as permission management, data metrics monitoring, etc., which are essential for our project. It would be much better if MRC could provide some foundational big data platforms like Hadoop, allowing us to perform large-scale data analysis. Additionally, there are some features that MRC may have but lack detailed guidance, such as setting up an external load balancer for Kubernetes, which AWS provides detailed instructions for. Moreover, there are some functional bugs in the web console. For example, we encountered unknown errors when modifying security groups, whereas the same operations using the CLI have no issues.
Suggestion: To enhance usability and functionality, we recommend that MRC improve its basic features such as permission management and data metrics monitoring. Additionally, detailed guidance on complex tasks would be beneficial, similar to the comprehensive instructions provided by other cloud providers.

6.3.2 Limited CI/CD Pipeline

Limitation: Although helpful in our development, the CI/CD pipeline we employed was quite limited in scope. With our current setup, even if code does not pass tests it still must be temporarily deployed to the production system. This could result in downtime and errors for users. In addition, even though if tests fail commits are rolled back, there is not standardized way to ensure that previous commits also pass tests. A major issue is that currently on rebuild all the Fission functions and triggers are deleted and reinstalled leading to potential downtime.
Suggestion:
A strong solution would be to deploy a dedicated development environment. This system would mirror the tech stack used in production without directly interfering with the live system. This would take additional resources, but would save errors and downtime in production. This is the typical way that code is deployed so it would be an appropriate solution for our system. When code is deployed to the production environment, it would also be beneficial to minimize the amount of Fission functions and triggers that are recreated. An update would be more efficient.

6.4 Future Improvements

This section will explore potential enhancements and new ideas that could be implemented in subsequent iterations of the project. These forward-looking suggestions aim to optimize performance, expand functionality, and increase the overall effectiveness of our solution.

6.4.1 Error Handling in Fission MQTrigger Functions

Error handling in Fission message queue (MQTrigger) functions has not been looked at. First of all, because of shortage of time we have failed to come up with a strong solution that will control and detect any error that can happen during the processing of messages. This exclusion makes our system vulnerable to problems like data loss, inconsistent states and potential system failures when errors are not properly handled.

A new approach would be to use another Fission MQTrigger function which regularly checks whether confirmation messages have returned to normal. If this function detects any discrepancy from what it expects, it would create several new call messages indicating that the relevant MQTrigger functions must be invoked again. Through enabling retries in this manner, we can make serverless functions for Kafka message processing more robust and reliable; hence better error handling as well as system stability.

6.5 ElasticSearch Machine Learning features

At the moment, our pre-trained models are stored in the backend server and using Fission API with Python script to return the prediction. ElasticSearch now offers advanced features that supporting both deployment of pre-trained models and creation of trained model API. **Opportunity**: We can use the trained model pipeline for the training and the predictions and feed those prediction directly to the model API using support features of Elasticsearch. [1] *https://www.elastic.co/guide/en/machine-learning/current/ml-trained-models.html*

6.6 Kafka

When testing functions, there is a risk of interfering with the message destined for production. Thus, there is an opportunity to deploy a development environment with its own Kafka topics for testing **Opportunity**: Improvement over the confirmation message: use messages for error handling, or passing parameter from one function to the next.

6.7 CI/CD

Along with the fixes with current issues in the CI/CD pipeline, there is also room for other improvements. With multiple people working on the same codebase, it would be beneficial to have a way to standardize coding conventions. For this, a linter could be deployed to make sure that code complies to a common format. Standardized code reviews would also be useful to get another opinion on an integration before it is merged.

7 Team and Roles

7.1 Roles

7.1.1 Dillon Blake

- Collection, cleaning, and uploading of the crashes data into ES
- Implementation of the Weather, Crime, Crash and Stream API
- Deployment of the CI/CD pipeline
- Secrets file for the keys and passwords
- Air quality vs weather frontend notebook

7.1.2 Andrea Delahaye

- Collection cleaning and upload of the crimes data into ES
- Implementation of the daily Fission/Kafka/ES pipeline to collect and clean live EPA data

- Implementation of the Models API, with Fission package
- Implementation of all the API tests, and EPA collection tests
- Crime vs weather frontend notebook

7.1.3 Yue Peng

- In charge of the overall server infrastructure
- Deployment of all server infrastructure

7.1.4 Jeff Phan

- Implementation of the Fission/Kafka/ES pipeline to collect and clean past weather data
- Backup ElasticSearch Snapshot with AWS
- Crashes vs weather exploratory data analysis
- Crashes vs weather frontend notebook

7.1.5 Alistair Wilcox

- Implementation of the live weather API
- Implementation of hourly weather pipeline
- Early development of functions to use in frontend notebooks
- AI functions for text and image generation

7.2 Strengths and Weaknesses of the Group

7.2.1 Strengths

The responsiveness of our group on WhatsApp was undoubtedly one of our main strengths. We ensured that no teammate was left without an answer when facing a bug related to another person's code or when they had questions about the functionality of a specific technology.

Our regular meetings (initially three times a week, then every single day towards the end of the allocated period) also helped a lot to have an overview of each team member's tasks at any given moment. This ensured that no one was left stranded or worked functionalities already implemented by another team member.

Another significant strength was the diverse skill set within our team. Each of us added so much insight from such different points that were multi-directional. This increased both the calibre of our answers and contributed to a creative environment in which ambitious new ideas could take root. It made those cutting-edge solutions actionable and it was a speed that came from not only the options available but also our collective ability to adapt quickly to new tools and technologies.

In addition, our team portrayed great communication and conflict resolution. This attitude allowed us to tackle differences or disagreements with a win-win approach — always discussing issues seeking solutions, and not surprised eyes at the problems. Handling issues in this way was a strong foundation for avoiding conflict and provided positively with little to no problems and helped create a very good team environment.

7.2.2 Weaknesses

While we wrote a few basic tests, our testing strategy was not comprehensive and covered only the functionality that we had just built. Occasional bugs, and problems were discovered later in development which delayed and led to more work. Better design for testing could have provided faster feedback on bugs.

8 Conclusion

Despite some significant challenges outside our control due to issues with the MRC, we successfully produced and deployed a range of scalable services and functions to support statistical analysis of real weather, pollution, crime and car accident data from disparate sources.

As a result of the hardware failures within the cluster, we were required to deep-dive into the troubleshooting of issues that prevented us from developing quickly, and progressing the project. We were therefore forced to focus on developing processes that were resilient to failure, and that helped to automate deployment. We discovered the importance of having rich and robust logging that can provide transparency and that supports speedy debugging of issues, as well as utility of automated testing and continuous integration and deployment to minimise manual rework when issues were encountered.

Our analysis and models provide insights into the correlations between weather conditions and various societal metrics. While some predictive models, such as those for air quality, showed weaker correlations, the study of traffic crashes and crimes suggest more significant findings that could inform public safety strategies. However, further research should be performed to collaborate these findings.

There are a number of improvements identified that could be explored in future projects. Enhancing the resilience of the infrastructure by adopting a multi-cloud strategy, refining the CI/CD pipeline, as well as implementing more sophisticated error handling and testing mechanisms. More advanced machine learning models could be explored to improve predictive accuracy and provide richer insights into the data.

In conclusion, this project has demonstrated the effective use of cloud computing, container orchestration, and data analytics to address the complex problem of predicting environmental and societal events based on geospatial data. By leveraging a Kubernetes cluster on the Melbourne Research Cloud, we successfully implemented a robust system that integrates various technologies such as Elasticsearch, Kafka, and Fission. This infrastructure facilitated the collection, processing, and analysis of diverse datasets including weather, air quality, crime, and traffic crash data.

References

[1] Elastic. *Machine Learning in the Elastic Stack*. Accessed: 29-05-2024. URL: `https://www.elastic.co/guide/en/machine-learning/current/ml-trained-models.html`.

[2] DALLE-3 OpenAI. *Is it safe to go outside*. Accessed: 29-05-2024.