# Austin Traffic Incidents API Report

Ashton Cole, Kelechi Emeruwa, and Carel Soney

26 April 2023

## ABSTRACT

We designed a REST API to process and distribute real-time traffic data from the greater Austin metroplex. Our product adds value to the end user by implementing data filtering and visualization functions for a raw dataset of over 320,000 entries. This project demonstrates our ability to deploy a RESTful API for an arbitrary data set in Python leveraging several libraries like Flask, Redis, GeoPY, and HotQueue. It also demonstrates a popular software engineering paradigm of containerized, loosely coupled systems, facilitated by Docker and Kubernetes. In conclusion, our application proves our ability to create portable, reliable and scalable software for a variety of engineering applications.

# 1 INTRODUCTION

The discipline of civil engineering is responsible for designing the infrastructure that sustains our daily lives, encompassing residential, commercial, and transportation facilities. It is unique among the fields of engineering in that it directly impacts the quality-of-life of ordinary people on a day-to-day basis. Traffic management is a significant subset of this field, one that frequently elicits complaints from commuters. Access to traffic data provides civil engineers with the information necessary to design traffic infrastructure that is both safer and more efficient. Moreover, such data is valuable for lay consumers seeking to avoid congested or hazardous routes. With these considerations, we chose live traffic incident data for the Austin area to be our source data set.

Publicly available on the city's website, the raw JSON file contains over 320,000 records of traffic incidents in the Austin area since 2017. The size of the data makes it difficult to navigate. This presented an opportunity for our team. We decided to design a RESTful API to automate accessing and filtering through the records. Clients would leverage this tool to access only what they need from the set. In addition, we provide a few useful data visualizations for engineers and consumers, including a bar plot and maps. More details about the development and functionality of these tools will be described in subsequent sections.

# 2 PROJECT OUTLINE

At the outset of this project, we were required to consider a variety of coding aspects and assess how they could be effectively integrated into our project. We engaged in considerable deliberation with regard to the project that would most optimally leverage our combined coding experience and lead to the most successful outcome. Ultimately, our team arrived at the consensus that a dataset rooted in Austin, which had the potential for practical, real-world applications, would provide the most stimulating and engaging coding experience.

## 2.1 Dataset

The initial phase of our project entailed the selection of an appropriate dataset. Following careful evaluation, we determined that the real-time updated dataset that tracks traffic incidents in the Austin area would best align with our project objectives. This dataset possesses inherent links to the field of civil engineering and, furthermore, allows us to identify the safest and most hazardous locations within Austin.

The dataset comprises multiple entries, each of which is accompanied by incident categorization, timestamp, coordinates, street address, publication date, and incident status. Here is a snippet of the data:


```
{
"traffic_report_id":"EBF5283D22437BB974F4248BA0EC4E90F26D32CB_1520152566000"
"published_date":"2018-03-04T08:36:00.000Z",
"issue_reported":"LOOSE LIVESTOCK",
"location":"(30.256997,-97.611818)",
"latitude":"30.256997",
"longitude":"-97.611818",
"address":"N Fm 973 Rd & Fm 969 Rd",
"traffic_report_status":"ARCHIVED",
"traffic_report_status_date_time":"1970-01-18T14:16:00.000Z"
}
```


Although the dataset appears to be relatively simple, it presents the opportunity to develop a meaningful API. In addition to facilitating straightforward data access, endpoints can be incorporated to distill current and historical data into meaningful statistics or visualizations. For instance, consumers may wish to access a live traffic map of active incidents, while researchers and engineers may desire a comprehensive map of all incidents to identify hotspots for specific traffic incidents or a time-series graph. All of these visualizations can be generated through the use of the Matplotlib python library and returned to the user via HTTP.

## 2.2 Beginnings

Following the selection of the dataset, the next step in our project was to effectively allocate tasks among the three team members. The responsibilities of programming the routes, generating plots, and producing this report were delegated to the most competent individual on the team. Drawing on the experience gained from our previous ISS Tracker project, the development of initial routes was a relatively straightforward process. However, we sought to challenge ourselves by brainstorming more distinctive routes that were uniquely suited to the current dataset. Given our prior experience in the COE 332 course, we did not need to undertake extensive research on programming concepts. Most of the necessary skills had already been covered in the class. Nonetheless, we found online resources regarding plotting to be particularly useful in addressing any incidental queries that arose.

# 3 KEY TECHNOLOGIES

## 3.1 Flask

Flask is a popular Python library that is widely used in the development of web servers. In particular, it is used for Microservices, a collection of loosely-coupled, independent services that can be created and deployed separately (Maguire). One of the key advantages of Flask is its small size and ease of use, which makes it a suitable framework for the development of REST APIs. In addition, Flask is known for its robustness and ability to handle high levels of traffic, rendering it an ideal choice for web applications (Allen). However, note that since Flask is not a standard Python library, it must be installed manually.

Like any web-connected application, a Flask application needs to use a particular port to send and receive HTTP packets. The port is associated with the machine or container, so it is written after the IP address. For example, 127.0.0.1:5000 would send a request to the local host through port 5000. By default, Flask uses port 5000. This is not the default port for web requests, so when accessing the API, this port must be specified after the address. What is important to note is that only one application can use a port on a machine at any given time, and some of those ports are reserved for other activities.

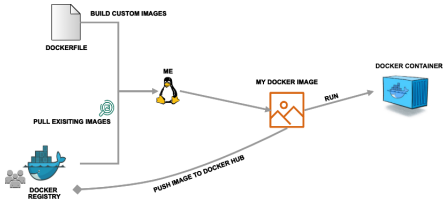
To make requests to the Flask service in Linux, the `curl` command can be used. The command facilitates sending the request and downloading the results with the HTTP protocol. The appropriate syntax is to specify the IP address and port number, followed by an HTTP request method. For example, `curl 'http://localhost:5000/incidents' -X GET` makes a `GET` request to the `/incidents` endpoint at localhost through port 5000.

Our Flask API endpoints are defined in a Python script. Each endpoint is implemented with a Python function that takes input and returns output. These functions are linked to the intended endpoint using the `@app.route` decorator function. Essentially, when our Python script runs, it listens for incoming HTTP requests to port 5000. When a request arrives, the imported Flask code directs it, using the decorator function, to the correct Python function. The returned value is then processed and sent as a HTTP request back to the client.

## 3.2 Docker

Docker is a tool which facilitates containerizing software. This means packaging programs and all of their dependencies into isolated units. This allows applications to be deployed in various environments without compatibility issues. These containers can be executed as isolated processes on the same machine, providing a lightweight alternative to traditional virtualization methods.

Figure 1 – Overview of Docker




**Source: Allen 2023**

As seen in Figure 1, Docker containers are derived from Docker images. An image is like a fixed template for creating new containers, providing the source code and specifying necessary

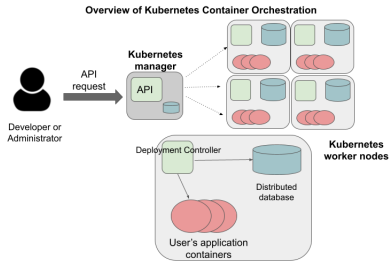
dependencies. This provides an easy means of distributing software in a standardized, portable format.

## 3.3 Kubernetes

Kubernetes (k8s) is a container orchestration system that supports Docker containers, as well as other container systems. Its key advantages lie in its robust features for modern, distributed systems management, as well as its wide adoption across multiple cloud providers, such as Amazon, Google, IBM, and Microsoft.

Kubernetes functions as a distributed system of software components that operate within a cluster of one or more machines. These machines are categorized as either 'manager' or 'worker' nodes, and user interaction with these "k8s" occurs via Kubernetes API requests. These requests may involve tasks such as requesting that two pods of a certain image are running to handle high loads, or shutting down a pod once it is no longer needed. A pod is a Kubernetes abstraction much like a container, except that a pod can encapsulate more than one container. Upon receiving the command, k8s schedules the new pod to run on the worker nodes. Finally, the API and deployment controllers ensure that the actual state matches the desired state.

Figure 2 – Overview of Kubernetes (k8s)




**Source: Allen 2023**

There are a few common types of pods that were used to orchestrate our application. Deployments manage the deployment of pods and keep them operational. They ensure that a desired number of containers remain active at all times. If an error causes one container to crash, the Deployment ensures that it is quickly replaced, minimizing interruptions for the client.

Services facilitate communication with pods. If a Deployment creates multiple identical copies of the same container, which one should be used? The Service coordinates even job allocation among the pods, so that incoming requests to the Flask API would be directed to copies of the software that are not in use.

As a data-based API, our program needs to store large quantities of data in between requests. Considering that it takes several minutes to download the traffic data from the source, it would be wasteful to re-download it every time a new request is made. It would also be inefficient to store the data in the Python script's runtime memory, and even if it weren't, the data would be confined to one container, and lost when that container was shut down. Persistent Volume Claim objects, also referred to as PVC, reserve memory resources to save data in the long term.

## 3.4 Queues

Concurrent computing refers to the ability of a computer system to perform multiple tasks simultaneously without compromising the accuracy of the program. One effective approach to developing concurrent programs involves the utilization of queue data structures.

A queue is an ordered list of items. Items can be added by enqueueing them, which appends them. They can be accessed and removed by dequeueing them, which removes the item from the front. This is helpful for concurrent computing. Multiple copies of a program can read the same queue for job allocations. Once a copy takes a job, it removes it from the list, preventing conflicts and inefficiencies. To facilitate the development of concurrent programs, the Python programming language provides a queue library known as Hotqueue, which stores queue items in a Redis database.

# 4 CODING

## 4.1 Purpose

The objective of this project was to create a Representational State Transfer (REST) Application Programming Interface (API) for the designated data set. The API allows clients to execute the four fundamental CRUD operations on data: Create, Read, Update, and Delete. Additionally, users can submit job requests for intensive analysis tasks like plotting, which are delegated to "worker" containers and retrievable with a separate API request.

## 4.2 Orchestration

The application creates a Pod, Service, and Persistent Volume Claim for a Redis database. This database stores all of the records pulled from the web, as well as a HotQueue of client-submitted jobs and metadata for each of those jobs. It also deploys multiple pods of the Flask application and worker nodes. The Flask application functions as a front end, interpreting requests and returning results. Multiple copies of this pod can handle API requests concurrently. The worker nodes execute more intensive tasks, like plotting, freeing up front-end pods to receive more requests. These nodes communicate with the front-end pods via the Redis database. Another Service object allocates requests to the front end, and finally an Ingress accepts incoming HTTP requests from the web. The whole application is deployed on a Kubernetes cluster, with access made available to external parties via a public URL.

Some features of the code include defensive programming as a snippet is shown below: `return message_payload("ERROR: No incident exists with the id: {id}", False, 404), 404`. A specific feature of our code includes a function, `def get_query_params() -> dict:` that handles all the query parameters for easier implementation.

## 4.3 Routes

The following is a list of all the routes in this project:

`/`: returns a welcome message

`/help`: returns description of each route

`/incidents`: returns data depending on the method called

Possible Methods:

1. POST: initializes data 2. GET: returns data 3. DELETE: deletes data

`/incidents/epochs`: returns list of epochs

`/incidents/epochs/<epoch>`: returns list of all incidents of a given incident type

`/incidents/ids`: returns list of incident identification numbers

`/incidents/ids/<id>`: returns incident by specified ID

`/incidents/issues`: returns list of incident types

`/incidents/published-range`: returns earliest and latest published incident dates

`/incidents/coordinates-range`: returns maximum and minimum coordinates

`/incidents/updated-range`: returns updated incident

`/jobs`:

Possible Methods:

1. GET: retrieves all current, future, and past jobs

2. DELETE: clears job record

`/jobs/plot/heatmap`:

Possible Methods:

1. POST: creates a new job to generate a plot

2. GET: retrieves heatmap plot

`/jobs/dotmap`:

Possible Methods:

1. POST: creates a new job to generate a dot map

2. GET: retrieves dotmap

`/jobs/timeseries`:

Possible Methods:

1. POST: creates a new job to generate a time series

2. GET: retrieves timeseries

`/jobs/plot`:

Possible Methods:

1. POST: creates a new job for plot

2. GET: retrieves all plots

`/jobs/incidents`: returns all historical, current, and pending jobs for incidents

`jobs/plot/<jid>`: returns plot for specific job id

# 5 DISCUSSION

## 5.1 Ethics

Engineering software applications with online data raises ethical concerns related to privacy, confidentiality, ownership, and bias.

Regarding privacy and confidentiality, software engineers must ensure that the data they use is legally and ethically obtained. It must not publicly reveal any sensitive or personal information that could harm other individuals or organizations. If sensitive information is included, engineers should implement adequate security measures to protect the data from unauthorized access. Our data set does not contain personal data. Incidents are anonymous, and details about each are obscured under discrete fixed categories. Thus, we conclude that we have fulfilled our privacy and confidentiality obligations.

Ownership refers to the use of copyrighted or proprietary data without permission. Beyond being unethical, this can spur serious legal and financial repercussions. Therefore, software engineers must ensure that they have the right to use the data and abide by any licensing or usage restrictions. Our data is sourced from the City of Austin's official website, where it is made free to the public. Likewise, the map image used for plotting comes from OpenStreetMaps, an online service which provides free street maps for use by the public. Finally, our API is not monetized, nor does it claim credit where it is not due. Thus, we conclude that we have fulfilled our ownership obligation.

Finally, biases in data sets should be considered, as they can lead to unfair or discriminatory results. Software engineers should mitigate bias by seeking diverse, appropriately-sampled data sets, and being mindful of the flaws and limitations data sets may have. Our source data set is in turn sourced from a live RSS feed of recent incident reports posted by Austin and Travis County law enforcement agencies. The set appears to be comprehensive and faithful of traffic incidents. It does not include any personal or protected-class-type information that would icrease the likelihood of unintentional harm from bias. Thus, we conclude that we have fulfilled our bias obligation.

In summary, software engineers must prioritize ethical considerations in their work. They must honor privacy, confidentiality, ownership, and mitigate bias. We have taken reasonable, good-faith actions and considerations to conform to appropriate data handling and processing practices, and ensure that our product complies with legal and ethical guidelines.

## 5.2 Future Improvements

Although we executed this project to the best of our abilities, we found many opportunities to expand and improve it. For instance, the initialization of the data to our program takes an extended amount of time due to the size of the source data set. We would like to conduct further investigations to optimize the performance of the program.

Another challenge we would like to solve is programming our code to automatically retrieve recent data. Our source is live, updated every five minutes, but users need to manually retrieve the data to obtain up-to-date information. Out-of-date data hurts the functionality of our API. Posting the data is also very time-consuming. It would help to have an automated process to update the data in the background, without disrupting access to the API.

Additionally, our API has imperfect data. Some values like dates and coordinates are far out of range, while the incident categories are poorly formatted. Currently, the most egregious

data points are excluded from analysis, but it would be better to perform proper data cleaning on the source. This would also add significant value for the client.

Currently, we are utilizing a static Mercator-projected image of Austin from OpenStreetMaps to construct our plots. The small latitude and longitude range plotted allow for a very close approximation of a Cartesian projection, but the plot is technically inaccurate. Furthermore, users are unable to zoom in on a specific coordinate due to image quality preservation restrictions. We were not able to use geographic plotting libraries because of techincal difficulties involving C-based dependencies. In the future, we would like to learn how to include the necessary dependencies and produce more customizable plots.

The JSON parameters we use are currently restrictive, requiring a start and end date for each job. We would be interested in re-implementing the code that handles these parameters to accept generalized dictionaries. This would enable expanded customizations for each endpoint. For instance, plots could be divided into categories by `issue_reported` or filtered to a single issue. This would let civil engineers to conduct more thorough analyses to identify problematic incident sites.

Finally, we would design a user-friendly front-end with features catered to lay consumers. Although it would not necessarily possess the full extent of functionalities of the primary API, it would provide key tools for commuters, such as a live map or a historical map identifying hazardous sites. It would be catered to users inexperienced with APIs, expanding our client base and increasing our positive impact.

# 6 CONCLUSION

In conclusion, this software engineering project has been a significant accomplishment, with the successful development and implementation of Flask and Redis components in Docker and Kubernetes. The project team leveraged various software development tools and technologies, such as the various libraries of Python, defensive programming, and testing tools, to design and implement the software application.

The team was able to effectively manage the project scope, timelines, and resources, delivering a high-quality product that meets all design requirements. While this project is considered complete, the project team recognizes ample opportunity for additional features and functionality to enhance the software's capabilities.

Overall, this software engineering project has been a challenging but rewarding experience, and the project team is proud of the results achieved. The successful completion of this project has demonstrated the team's expertise in software development and their ability to work collaboratively to achieve a common goal.

# 7 REFERENCES

Allen, William, et al. “COE 332: Software Engineering and Design.” COE 332: Software Engineering  Design - COE 332: Software Engineering  Design Documentation, 2023, https://coe-332-sp23.readthedocs.io/en/latest/

Transportation, Austin. “Real Time Traffic Incident Report.” Open Data, https://data.austintexas.gov/resource/dx9v-zd7x.json

Maguire, Jamie. “Microservices Architecture Diagram Examples.” DevTeam.Space, 30 Mar. 2023, https://www.devteam.space/blog/microservice-architecture-examples-and-diagram/