Master Thesis

Automated Testing of Intelligent Public Transportation Systems Using GCP and Selenium

Verma Mohit

Matriculation ID: 11027900

Applied Computer Science (ACS)

SRH Hochschule Heidelberg

Internal Supervisor 1:

Prof. Kamellia Reshadi

Internal Supervisor 2:

Prof. Dr. Gerd Moeckel

DECLARATION

This thesis, "Automated Testing of Intelligent Public Transportation Systems Using GCP and Selenium," is hereby declared by me, the undersigned.

It is the outcome of my unique labour. There is nothing in this thesis that has been authored or published by another person or organization, except clear citations to their work. In the text or in the references part of this paper, I have properly cited and recognized any concepts, methods, quotes, or relevant sources that I have used or addressed.

I additionally attest that no other university or institution has previously received this thesis, in whole or in part, for the purpose of awarding an academic degree or diploma.
I have abided by all applicable ethical, academic, and professional standards, and I understand that any breach of these principles may result in the revocation of my degree or other academic sanctions, in accordance with university policies and regulations.

Mohit Verma
07/03/2025

ACKNOWLEDGEMENT

Thank you for giving me the chance to pursue my master's degree in applied computer science. I would like to thank my university, SRH Hochschule Heidelberg. I want to express my profound gratitude to all of the instructors who have been instrumental in assisting me in realizing my goal of earning a master's degree, particularly Prof. Kamellia Reshadi and Prof. Dr Gerd Moeckel, who have continuously offered steadfast support during the course. Her advice has been an ongoing source of motivation.

The enthusiastic involvement and insightful criticism of Professor Kamellia Reshadi were crucial to the successful completion of this thesis. Her assistance in improving the calibre of my work and deepening my comprehension of the subject has been priceless.

In addition, I would like to thank my family and friends for their tremendous support and encouragement during my academic career, as well as during the research and composition of this thesis. Their constant presence has been a motivating factor. Force behind my accomplishments, and I acknowledge that none of this would have been possible without their support. I am incredibly appreciative of their steadfast faith in me.

ABSTRACT

This study shows how an Intelligent Transportation System (ITS) Dashboard that observes and forecasts traffic conditions in real time was developed and tested on Google Cloud Platform (GCP) and Selenium. The study concentrated on finding solutions for problems related to cost-effectiveness, scalability, and dependability when managing real-time data for public transportation networks. While Locust and Google Cloud Monitoring carried out the performance tests for the features, Selenium was used for automated functional testing of the core functions. Testing revealed methods to improve resource management and high-traffic prediction services, confirming the system's scalability and dependability. The financial advantages of cloud computing were demonstrated by a cost study, which demonstrated how resource scalability and result caching significantly lower expenses. The study illustrates the need of automated resource management and offers a framework that developers may utilize to build real-time systems. In order to improve performance and expand application areas, research directions for advancement include advanced machine learning capabilities combined with edge computing technology and IoT integration.

Table of Contents

List of Figures

CHAPTER 1: INTRODUCTION

1.1 Background

Transportation solutions that citizens rely on are rapidly changing as more and more cities implement the smart city technology goals. Denizens are concentrated in urban areas, and therefore urbanization requires efficient, reliable, and sufficiently sustainable public transport. AI and big data, cloud storage, the usage of real-time data are the tools have become extremely important to address these challenges. Advanced technologies at work in these smart public transportation systems include: better usage of the available resources, improvement in traveller experiences, and optimisation of the route and time management [24].

Among the different cloud platforms, Google Cloud Platform (GCP) is more preferable because of the strong tool and output service it provides. GCP offers definite computing, extensive learning tools like Vertex AI, and real-time messaging services like Pub/Sub and geospatial facilities like Google Maps. They enable developers to build transportation systems that have functions such as real time navigation, updates and routing. With cities further developing and constantly changing, it becomes requisite that intelligent transportation systems based on platforms such as GCP are established so that mobility solutions deployed reflect the challenging outlook of urbanization and sustainability.

1.2 Motivation

The following challenges have however limited the use of intelligent transportation systems: performance, reliability and scalability. The routine approach of testing frameworks is ineffective in meeting the demands of a cloud environment which is a real-time environment. These limitations point to the fact the more sophisticated method of testing has to be developed to prove that the ITS will work as intended and will easily scale in a real life.

The automated testing frameworks hold the key to addressing all the above challenges. Specifically, through usage of tools such as Selenium and incorporating them with Google Cloud Platform services, experienced developers can build dense test platforms that incorporate aspects of functionality testing in addition to performance testing of transportation systems. In addition to improving the accuracy and effectiveness of testing this approach provides cheap ways of expanding operations. This thesis seeks to address these challenges by proposing an automated testing framework for intelligent public transportation systems based on the qualities of reliability, cost and effectiveness [39].

1.3 Aim and Objectives

The main objective of this thesis is to develop an effective framework for automating the testing of intelligent public transport systems on GCP using Selenium. The specific objectives of this research are:

- To develop a sample web application utilizing GCP services to optimize public transportation systems through features such as real-time tracking, route optimization, and live communication.
- To implement and evaluate an automated testing framework using Selenium for functional testing, focusing on ensuring the accuracy and reliability of system features.
- To conduct performance testing using GCP tools such as Compute Engine and Cloud Monitoring to evaluate system scalability, response times, and overall performance under varying conditions.
- To assess the cost efficiency of running automated tests on GCP, providing insights into balancing testing coverage and resource allocation.
- To propose actionable recommendations for enhancing the testing and optimization of cloud-based intelligent transportation systems.

1.4 Methodology

To accomplish these objectives, this research employs a well-defined and orderly approach. The methodology comprises the following key steps:

- Development of a Sample Web Application: A web application will be developed with the help of various GCP services to show the capabilities of smart PT systems. The identified key features involve bus and train tracking, vertex AI route optimization, live communication through Pub/Sub, Google Maps geospatial visualization.
- Automated Functional Testing: For functional testing of the application selenium will be used. It will mainly be an attempt at establishing the basic functionalities of real-time updating, accurate routing and communication in various situations.
- Performance Testing: The performances will be tested on load and using Compute Engine of GCP to also monitor response rates, drainage of resources and scaling with the help of GCP Cloud Monitoring. Stress testing will also be used to determine how the application acts in a number of conditions [31].

- Cost Analysis and Optimization: The factors of running automated tests on GCP will include the cost, with particular regard to ways of pragmatically managing resource use and testing cadence that will allow for sufficient and effective coverage.

1.5 Scope

Thus, the scope of this thesis is rather narrow and concerns only intelligent public transportation systems aimed at increasing the efficiency of transportation within urban environments improving routing, real-time updates and user experience [27]. The research focuses on developing and testing a framework that addresses the following key aspects:

- Scalability: Ability to use the system optimally in responding to fluctuating load and user requirements.
- Functionality: Confirming the credibility of the basic elements of the Providers' services, focus on providing actual tracking of vehicles and routes optimization.
- Cost Efficiency: Discussions on how to reduce cost in relation to the run time of automated tests on GCP [14].

1.6 Synopsis of the Thesis

This thesis is organized into five chapters, each addressing a specific aspect of the research:

- Introduction: Following this, an outline of background, rationale, purpose, method, of the study, its broader range of topics and its importance is demonstrated in this chapter.
- Literature Review: This chapter examines prior literature on intelligent transportation systems, distributed testing environments, and automata technologies. It demonstrates major issues and missing links in the existing body of work, upon which the proposed study is based.
- Methodology: This chapter shows how the approach used to create the sample web application, how the automation tests were designed and executed, how the performance and cost of the proposed framework was evaluated.
- Results and Discussion: This chapter comprises the conclusion of the research where functional and performance tests results are presented, understanding of scalability and costs, and an evaluation of proposed framework efficiency.
- Conclusion and Recommendations: This chapter reviews the main conclusions of the study and considers its implications for IS research and practice concerning ITS implementation and the related issue areas.

CHAPTER 2: LITERATURE REVIEW

The research discusses that a new urban trend has greatly influenced the development of public transport systems that has forced cities around the world to invest in more intelligently to address the increasing mobility requirements. Information technology is playing an important role in solving the problems like traffic congestion, wrong routing and wrong timing of the transport systems through Intelligent Transportation Systems (ITS). As ITS is focused on the effective use of resources, development of satisfactory user experiences as well as the achievement of sustainable objectives in terms of city development through use of superior technologies such as; AI, real-time data processing, and cloud computing. Of all the platforms available, Google Cloud Platform (GCP) is a complete enabler for these solutions because of the ability to host large and diverse transportation implementations [34].

Nevertheless, the creation and application of ITS hold crucial concerns for implementation. The applications imply that there must be resilient, flexible, and efficient systems able to process a huge amount of real-time information from the external environment. However, traditional manual testing regiment fails to address these dynamic demands, resulting in inefficiencies and vulnerability in the performance of the system as well as functionality. These challenges point towards the need of having improved testing tools with a combination of automation and cloud [38].

When tested along with cloud platforms like GCP, automated testing frameworks give plausible solutions to these challenges. They allow systematic functional and performance testing, to confirm the overall adequacy, dependability, and performance of ITS. This chapter focuses on media and applications of ITS, the centrality of cloud computing, automation in software testing, emerging performance testing approaches, and cost optimization of cloud-based testing. This paper offers understanding that will guide the development of an ITS specific automated testing framework by critically reviewing existing literature. The research results will help fill existing gaps and meet the increasing demand for effective and efficient testing in contemporary transport structures.

2.1 Intelligent Transportation Systems

Due to the growing technological development, Intelligent Transportation Systems (ITS) have emerged as key characteristics of the transport industry. These systems provide the foundation of moving to more efficient, reliable, sustainable and green mobility system for developing and newly industrialized cities. ITS's objectives are in line with some of the major issues that are

facing developing cities today like traffic jams, he said, poor resource utilization, and disorganized service provision. Besides, ITS go beyond simply optimising the operations with the help of AI, IoT, and real-time data analytics and make the experience of the users better [5].

2.1.1 Overview of Intelligent Transportation Systems

There are many components in Intelligent Transportation Systems that use multiple higher technologies to control and improve the flow of public transport systems. These solutions solve problems connected with urban mobility by offering live tracking, changes to further routes or stops according to the prediction, and improved passenger security systems. ITS has been advanced with the help of IoT and AI to facilitate the collection of data from many sources including GPS devices, sensors, as well as ticketing systems.

ITS is gaining popularity among the governments and the transit authorities across the globe in view of general efficiency and sustainability considerations. For instance, Intelligent Traffic System which Singapore has developed uses real time analysis to manage traffic flow, areas of congestion and incidents, as well as traffic signal timings. But the main issues that need to be solved to advance ITS are focused around the core difficulties of large-scale ITS implementation, such as the stability of the system as well as the need to ensure its scalability and the active usage of its provided services [11].

2.1.2 Challenges in Intelligent Transportation Systems

However, ITS subject to a number of technical and operational challenges. The main challenge is scalability because transpiration systems need to process large amounts of data and user engagement at certain times of the day. Moreover, real-time systems themselves are viable to latency and connection lost that may be a threat to track and schedule data.

There are other issues that include data protection and privacy which are even more risky to systems that employ cloud services. PTs convey a lot of user data which makes them vulnerable targets for hackers. Furthermore, processing of the information of various types and origins, e.g., traffic data, vehicle parameters, and users' opinions, needs sound methodologies for data unification and consistency [20].

2.2 Role of Cloud Computing in Transportation

Particularly, the transportation sector has been moving towards cloud computing to meet the challenging dynamic mobility requirements of increasing complexity in urban environments. The growth of cities along with increasing users' needs decrease the effectiveness of traditional solutions as they are unable to provide the needed level of flexibility and expansion characteristics of transport networks. Cloud computing has become a revolutionary technology

which has helped ITS in analysis of large amount of real-time data, reliability of the servicers provided and increase efficiency of the systems. Using modern cloud platforms including GCP, transportation authorities can develop modern systems for transportation that can suit the needs of modern cities [2].

2.2.1 Benefits of Cloud Platforms

Cloud computing has been a significant component of ITS because of its ability, adaptability, and affordability. Such services as GCP make that transportation authorities can launch applications working with actual-time info as well as prepare efficient suggestions. The nature of Cloud platforms means that ITS are easily cope with increased use during specified hours where that might typically be expected [13].

Key GCP services used in ITS include:

Vertex AI: Used for the purposes of predictive analysis and decision making concerning which routes are best for a particular unit to follow.

Pub/Sub: Allows real time communication between system components.

Google Maps Platform: Offers location tracking & navigation capabilities of live GPS coordinates.

Cloud platforms also do away with the costs associated with physical hardware as all are centrally located on the cloud. This is especially important for PTAs that may have a low budget to work with, due to the cut throat competition in the current market [23].

2.2.2 Limitations and Challenges

There is no doubt that cloud platforms have numerous advantages, however, there are disadvantages when implementing this solution. Another notable aspect is the cost that may grow extremely fast when operating large-scale applications using cloud facilities. Another disadvantage is data latency since the communication of cloud data often takes time to transmit resulting in the inability to support real-time applications. Furthermore, the global requirement for data protection, including GDPR, increases the layer of challenge for ITS development and integration.

To overcome these limitations, scholars have considered using the mathematical model of a hybrid cloud based on cloud infrastructure and edge computing. This approach also helps in minimizing latency as compared to the pipeline model as the compute part is done near to the origin of data and at the same time benefits from the features of cloud platforms for the storage and analytical part [41].

2.3 Automation in Software Testing

When it comes to the process of software development testing is an essential step that contributes highly to the functionality, stability, and scalability of application forms. On the increase complexity in systems especially the ITS, handling of testing issue using the traditional manual method reduces the effectiveness of the system. Due to the continuous development of ITS, processing, updating, and interacting with the user's, automated testing has become an effective long-term solution. Testing frameworks also allow developers to check major functionalities because when systems are required to function under different conditions, it becomes easy to determine if everything will run as planned (N et al., 2022).

2.3.1 Importance of Automated Testing

Automation testing is now typical in current software development since new releases of applications need testing while having complicated aspects. To ITS, automated testing is particularly useful to guarantee the functionality and web scale of key functionalities such as real-time update, routes optimization and passengers' notifications.

As contrasted with manual testing, automation greatly minimizes the amount of time and energy needed to confirm software functions. Selenium for instance are helpful tools for developers to perform actions that interact with web applications as users do to confirm that features like live tracking and ticketing function properly inter alias. Automation also increases the extent of testing to cover all the important aspects of the programs and system [16].

2.3.2 Integration with Cloud Platforms

Integration of the automated testing frameworks to cloud platforms makes them optimized for use. Mobile application developers can perform more than one test at one go and they can also mimic the actual environment. For example, Selenium has to be implemented on GCP to perform exacerbated functional and performance tests at scale.

From cloud platforms, developers can also run CI/CD pipelines for testing and deploying the product. It makes a surety that updates which are being released for use go through test runs to minimize possibilities of system breakdowns [7].

2.3.3 Introduction to CI/CD Concepts and Their Importance

CI/CD basics and why it is necessary

CI/CD is a development practice where code changes are automatically built, tested in addition to being deployed to production environments. Continuous Integration centers on the integration of code changes on a frequent and regular basis or on a real-time basis, expressing

the intention of periodically identifying the various problems that may be generated by integration and testing them at the same time [21]. While continuous Deployment centers on the release process throughout which tested alterations are delivered occasionally and rarely on a manual basis to the end-users. Altogether, CI/CD pipelines allow getting new reliable software releases to the market much quicker.

In transportation applications whereby updates are critical, and operations have to be smooth, CI/CD pipelines offer a stable structure. Other service areas, for instance, route optimisation features or real-time tracking upgrading features can be added without necessarily interrupting current services. Equally, testing tries to maintain quality in CI/CD in terms of functionality, performance and security leaving little room for application to fail when live.

2.3.4 Case Study: Uber's CI/CD Implementation

Uber, the leading international service that provides ride-hailing services, has managed to implement CI/CD pipelines in order to release new quality updates to its app while providing uninterrupted service [12]. Due to the size of Uber, controlling fast feature delivery cycles without stopping services is essential. CI/CD pipeline is used in the organization to be able to incorporate, test and implement changes seamlessly in different microservices.

CI/CD process of Uber has a lot of automated testing integration and one of them is the use of automated testing [42]. Any code change brings functional, integration and performance tests in order to check if new features affect the overall app functionality. For example, real-time routing and dynamic or time-of-use pricing options are thoroughly examined and checked for the accuracy of their performance in connection with various conditions and situations.

In the deployment phase, various leveraging associated with canary releases are used whereby an update is released to a limited number of users first. This approach reduces such risks as performance and user feedback to an organization before large-scale implementation. There are also strong monitoring tools for Uber’s CI/CD pipeline that gives real-time observation of the applications and allows teams to identify and fix problems that arise after the application is deployed.

CI/CD pipeline has helped Uber to release new updates daily to help it stay as a reliable, responsive, and competitive app. It has also enabled Uber to reinvent itself quickly to add functionality such as better routes and safety concerns without a decline in service delivery.

2.4 Performance Testing for Cloud-Based Applications

In today's world, based on rapidly developing technologies, it is crucial to provide high service and fault tolerant cloud applications for IT systems such as Intelligent Transportation Systems (ITS) that are often used in real time environments. Performance testing is central to determining workloads that can be expected to be placed upon an application under various conditions so that the end users are not disappointed. While in this testing procedure the testing of a feature is different to functional testing, performance testing looks at system performance under stress; looking at parameters such as response time, throughout and resource utilization. In the case of ITS, performance testing is critical since these systems have to support variable user loads, especially during traffic rush hours without compromises to system reliability and service quality. This section looks into the performance testing approaches applicable to ITS together with the problems of performance testing real-time web applications hosted on cloud environments [19].

2.4.1 Tools and Techniques

It studies the behaviours of a system when subjected to particular capacities to prove that it can serves these benchmarks including response time, throughput, and resource usage. In the case of ITS, performance testing is relevant to ensure that an application can support the strict traffic demands during certain periods of the day [29].

GCP provides a range of tools for performance testing, including:

- Compute Engine: Stimulates many users to test the system and its ability to perform during concurrent heavy usage by the users.
- Cloud Monitoring: Records characteristics of the system like the latency, the error rate, and the number of resources utilized.

Load testing is easily one of the most tested methods of determining how well a system can perform under ordinary circumstances and while it is being stressed. Stress testing in contrast is aimed at proving the system's behavior in heavy workloads where such turns may appear.

2.4.2 Challenges in Performance Testing

As will be demonstrated in the ITS present setup performance testing also poses some unique challenges due to the real-time and dynamic nature of the application. Testing high-level features and capabilities like traffic conditions during transportation or sudden rushing or crowded conditions or high or low user traffic is possible only with rich test frameworks. Resource management is also an issue, especially because overbooking results to high costs while under booking may translate to partially carried out tests.

Furthermore, analysing the performance results in order to discover possible areas for improvement and define further optimization of the systems is not an easy task. GCP for instance comes fitted with Cloud Monitoring, a tool that makes the work easier given that it gives real time information about the systems behavior [30].

2.5 Cost Optimization in Cloud-Based Testing

One of the standard practices currently in use to validate reliability and effectiveness of todays' complex applications is the cloud-based testing particularly used in Intelligent Transport Systems (ITS). However, with the freedom and elasticity which characterizes cloud computing platforms come the issues surrounding cost control. When it comes to public transportation authorities and organizations, many of which work on a shoestring budget, it is important to learn how to achieve the best possible balance between cloud testing costs and test coverage and quality [17].

2.5.1 Strategies for Cost Efficiency

One of the biggest drivers in cloud-based testing is the idea of cost reduction, which is also significant in this case since public transportation authorities are likely to have limited funding. Lack of proper cost control measure are counter acted by dynamic resource allocation, parallel testing, and prioritization of test cases.

Dynamic resource allocation means that in testing, the cloud resources are only deployed where necessary. This makes the allocation of resources offer more effective since it is real time and does not support wastage. For instance, during load or stress tests, the number of resources can be increased and reduce when the testing needs are low.

Another approach identified as important in reducing the testing times and utilization of resources is parallel testing. It can be understood that through the parallel execution of several test cases, a developer will be able to do so in the shortest time possible and at the same time use limited resources. This is especially helpful for mass-scale ITS, where tens of thousands of test cases need to be run [32].

2.5.2 Balancing Cost and Test Coverage

It is achievable to achieve cost optimization while maintaining the comprehensiveness of testing in cloud-based testing frameworks. The punishment of efficiency, where cost reduction becomes the primary, if not only, motivator for testing, leads to inadequate coverage and potential of not catching significant problems. On the other hand, prioritizing coverage may incur unaffordable levels of expenditure destined for testing purposes.

Thus, the choice of test cases should have an appropriate priority. Thus, critical features including real time update, routing, and communication must be tested thoroughly often times while less serious features may be tested infrequently. In this way, the resource allocation process is optimized so that the high reliability of the system and the performance do not allow for the ineffective use of resources for sustainable testing [43].

2.6 Challenges in Cloud-Based Testing

The challenges posed by cloud-based testing of ITS are such that some specific considerations need to be made in order to assure ITS dependability, expansibility and conformity. The first one is related to the fact that ITS is characterized by variability, and test environments must be able to captured a broad range of likely situations, including variation in user load, congestion, and system slowness. This makes system performance assessment under various conditions require high level simulation to ensure that it is effective [9].

Another serious issue is the question of the modifiability and portability of the automated test scripts. Due to imitable requirements in transportation, its experience changes as well as the features that are incorporated in the system often experience changes with time. Therefore, a testing framework must support changes while minimizing the degrees of change with respect to the extent of redevelopments and recognize their inefficiency and high cost in the long run.

Data security and compliance all are extra challenges added to the scenario. Cloud-based ITS applications handle enormous amounts of customers' private information, which is why they are regulated by laws, including GDPR. There is therefore a big challenge in making sure that the testing frameworks compliment these regulations while at the same time guaranteeing on data accuracy as well as the privacy of users. Neglect of these problems can result in severe legal and reputational consequences.

These are some of the challenges for which there must be new solutions. For example, using AI for testing integration increases the flexibility and efficiency of testing strategies, and the usage of a hybrid cloud approach makes latency low and data protection high because of processing near the source of data. In that way, ITS developers can create solid testing frameworks that would support its cloud solutions' scalability, dependability, and conformance.

2.7 Summary

This literature review has underscored the significance of ITS in making adjustments has identified cloud platforms as vital components in developing ITSs for mitigating urban mobility difficulties. Frameworks for automated and performance testing have become critical

components of these systems. But it still has limitations; costs issues, issues related to security of data and last but not the least the nature of real time applications is constantly evolving.

In a nutshell, the findings from this review give adequate support to the methodology that shall be preferred in the next chapter. Thus, by overcoming the shortcomings revealed in the literature, this research will contribute to building a SDLC framework for innovative ITS testing based on GCP and Selenium. Chapter four of this thesis will show the actual method used in conducting the research such as creating a sample web application, the execution of testing frameworks and assessment of the performance and costs of the system.

Chapter 3: Methodology

This study describes all the procedures used to build and test our intelligent transportation system using Google Cloud Platform and Selenium. Our project builds an automated system that makes cloud-based systems work better while keeping costs down by addressing real-time transportation data performance and functional problems.

3.1 System Development Phase

The project starts by developing a simulated web application which demonstrates realistic behaviour of intelligent public transportation systems on GCP. This application features real-time tracking alongside route optimization capabilities and it enables system component live communication. The choice of Google Cloud Platform occurred because it supplies complete integrated services which enable real-time data processing together with AI-driven enhancements.

The application development relied on GCP services including Vertex AI, Google Maps Platform, Google Cloud Pub/Sub, and Compute Engine to construct its essential functionalities. The main purpose of the application involves receiving live bus and train data then using AI processing algorithms for route optimization followed by displaying outputs through a web-based interface.

Different significant development steps were established to create a design which was both strong and expandable. The first step required establishing the needed cloud infrastructure through GCP tools. The backend services operated on Compute Engine whereas Cloud Fire store served as the data storage solution because it provides scalable capabilities with real-time synchronization features. The web application front-end was created with modern development tools HTML, CSS and JavaScript to enable smooth connection with GCP services [8].

3.1.1 GCP Service Selection and Setup

The system used various GCP services to fulfil its operational requirements:

- **Vertex AI**: Route optimization and predictive analysis functions are possible through this tool.
- **Google Cloud Pub/Sub**: The system utilizes real-time messaging capabilities to connect its different operational elements.
- **Google Maps Platform**: The system delivers real-time bus and train location services through geolocation technology.

- **Compute Engine**: The web application runs from this server while backend operations remain under its management.
- **Cloud Firestore**: Real-time data storage along with synchronization occurs through its implementation.

I began by establishing virtual machines through Compute Engine which needed subsequent service configuration to enable smooth connective itY.

3.1.2 Web Application Development

The intelligent transportation system uses the web application as its main user interface. The system uses contemporary web development technologies for its creation:

Front-End Development:

The web application front-end development relied on HTML, CSS, and JavaScript elements to create a responsive user-friendly platform. These technologies enabled users to work smoothly with Google Cloud Platform (GCP) services by allowing real-time Google Maps data visualization and dynamic content changes. Users gained styling flexibility via CSS and real-time data handling functionality through JavaScript which enhanced their experience. The modern interactive front end produced an interface which automatically adjusted to various device screens and dimensions [4].

Back-End Development:

Python-based services at the back end manage the entire data collection process and processing and establish connections to Google Cloud Platform (GCP) services. The system connects to Cloud Fire store to store real-time data and utilizes Pub/Sub for message transmission. The application obtains real-time bus and train information to apply Vertex AI predictive models for route optimization before it generates results. The system transmits optimized data to the front end so users can view updated information on Google Maps through the user interface [35].

3.1.3 Data Integration and Real-Time Simulation

The system development required fundamental integration of real-time data collection. Real-world simulations required the inclusion of traffic delays and route changes into the collected data. The system incorporated features to operate effectively under changing conditions for delivering precise updates [37].

3.1.4 Practical Instructions for System Development

To ensure the successful development of the system, it is essential to follow a step-by-step approach in configuring and implementing Google Cloud Platform (GCP) services and building the web application.

Step 1: Set Up Cloud Infrastructure on GCP

- Create a Google Cloud Project and enable necessary services such as Compute Engine, Cloud Firestore, Pub/Sub, Google Maps Platform, and Vertex AI.
- Use Compute Engine to create a virtual machine (VM) instance for hosting the backend. Ensure that the VM has sufficient resources (CPU, memory, storage) for running real-time data processing tasks.
- Configure Cloud Firestore for real-time data storage. This will be the database for storing bus and train locations, traffic data, and route optimization results.
- Enable Google Maps API to provide real-time geolocation services on the front end for displaying live bus and train tracking.

Step 2: Develop the Web Application

- **Front-End Development:** Build a responsive front end using HTML, CSS, and JavaScript. Integrate Google Maps to display real-time locations and routes.
- **Back-End Development:** Use Python for the backend services to collect, process, and analyse data. Connect the backend to Fire store for data storage and use Pub/Sub to send real-time updates.
- Implement Vertex AI for predictive route optimization. Train the model to analyse historical data and generate optimal routes based on current traffic conditions.

Step 3: Test the Application Locally

- Before deploying the application, test it locally to verify core functionalities like data retrieval, real-time updates, and route display.
- Use mock data to simulate bus and train movements and check how well the predictive algorithm adjusts to route changes.

Step 4: Deploy the Application to Compute Engine

- Deploy the web application to **Compute Engine** for public access. Ensure security by enabling firewall rules and setting up SSL for secure communication.

3.2 What is Vertex AI and How it is Implemented

3.2.1 What Is Vertex AI?

Vertex AI is Google Cloud's consistent, end-to--end platform for creating, deploying, and scaling machine learning (ML) models is Vertex AI. It aggregates several formerly disparate services (AI Platform, AutoML, etc.) into a single platform where developers, ML engineers, and data scientists can:

1. Store and manage data in sites like Google Cloud Storage (GCS) or BigQuery, then easily link those information sources to training jobs.

2. Train and Build Models: You can either train your own models from scratch (using TensorFlow, scikit-learn, PyTorch, or XGBoost, for example), or you can use AutoML to make a good model for you automatically.

3. Host the learned model in a scalable, safe, secure manner for prediction. Vertex AI generates an HTTPS endpoint accessible for predictions and automatically sets up the required infrastructure.

4. Track several model versions, compare tests, and implement fresh versions in production with least possible conflict.

5. Track prediction requests, latencies, and accuracy by means of built-in monitoring and explanation. Vertex AI also provides means for communicating model outputs, hence enabling fairness and openness.

Vertex AI lets you concentrate on creating efficient ML solutions instead of stressing too much on hand-made infrastructure provisioning. It also closely interacts with other Google Cloud products (Firestore, Pub/Sub, Cloud Functions, Cloud Run), therefore facilitating complete cloud-based workflow orchestration.

3.2.1 Data Generation and Preparation

The **data preparation** process starts with generating a dataset that mimics real-world vehicle tracking. The script `Generate_vehicle_data.py` creates a **CSV file (**`vehicle_data.csv`**)** containing synthetic vehicle movement data with the following key attributes:

- vehicle_id: Unique identifier for each vehicle.
- lat and lng: Geolocation coordinates.
- speed: Speed of the vehicle in km/h.
- status: Vehicle status, labeled as **“on-time” (0) or “delayed” (1)**.
- type: Type of vehicle (car, bus, train).
- timestamp: The recorded date and time for each entry.

```
vehicle_id,lat,lng,speed,status,type,timestamp
96,51.48782862807156,-0.13167386732275504,41,delayed,car,2025-01-24T07:50:22
88,51.52989085852151,-0.12601734793737454,85,on-time,bus,2025-02-06T20:13:58
28,51.483811641430435,-0.11315275634602842,27,on-time,train,2025-02-09T18:09:54
52,51.491238011369965,-0.11334672374116127,110,on-time,train,2025-01-27T00:14:06
42,51.5055081154579,-0.11813804271163192,83,delayed,bus,2025-02-07T22:05:49
17,51.47626463302007,-0.11096417271616685,38,on-time,bus,2025-01-27T07:22:27
79,51.48691127500406,-0.13288148223043028,115,on-time,train,2025-01-18T11:52:06
67,51.500084568352676,-0.1371071141694663,82,on-time,car,2025-01-13T23:32:12
64,51.4757251255475,-0.10409543674538185,56,on-time,bus,2025-01-23T21:11:40
83,51.485964430109824,-0.12263615364157786,46,delayed,car,2025-01-16T17:19:46
4,51.49963636816343,-0.11958695818701134,28,on-time,bus,2025-01-17T12:14:23
57,51.49686191619249,-0.12720927773828525,116,on-time,car,2025-02-07T20:54:16
70,51.49234158315886,-0.12640067124479945,52,on-time,train,2025-01-27T07:17:45
20,51.497571334841126,-0.11604684920930741,35,delayed,train,2025-01-14T22:53:36
90,51.473846583136094,-0.12219218049149853,99,on-time,train,2025-01-27T02:06:18
87,51.47173167847956,-0.12688276676318083,120,on-time,train,2025-01-31T17:05:39
58,51.513179271968156,-0.10042194426821459,99,on-time,bus,2025-01-15T23:25:36
84,51.482498963367,-0.11475090550290268,103,on-time,train,2025-01-14T19:28:24
88,51.52131306691689,-0.10647474852327352,35,on-time,car,2025-02-03T04:25:45
15,51.47008189334764,-0.13219838593202368,63,on-time,car,2025-02-01T19:13:21
37,51.51768347403827,-0.11912845492095646,49,on-time,train,2025-01-26T12:10:18
55,51.51587416446156,-0.1074118320577721,89,on-time,train,2025-01-17T09:53:02
11,51.49097817664566,-0.1166414985491773,63,delayed,bus,2025-01-17T15:19:51
76,51.498344014991844,-0.11982065920786865,84,on-time,bus,2025-01-29T04:00:30
77,51.494378849898126,-0.11308926101963315,47,on-time,train,2025-02-08T17:56:30
52,51.49200451419239,-0.11496762430239438,106,on-time,bus,2025-01-27T17:13:13
44,51.51495647840365,-0.1115931259024808,85,on-time,car,2025-01-15T04:46:12
47,51.47571051799303,-0.14865456315075515,27,delayed,train,2025-01-31T08:35:41
26,51.49262284746223,-0.1335439175277579,36,on-time,bus,2025-01-15T17:08:49
13,51.47875924853569,-0.1066852466846373,47,delayed,car,2025-01-21T09:22:36
```

Figure 1: Example of some data from "vehicle_data.csv".

3.2.2 Model Training with Vertex AI

The script `vehicle_model.py` preprocesses the generated dataset and trains a RandomForestClassifier to predict whether a vehicle will be on-time or delayed. We chose RandomForestClassifier, because by averaging many decision trees, Random Forest resists overfitting, manages heterogeneous data types (numerical + categorical), and is robust. It also provides unambiguous feature importance and usually produces good classification results with minimal tuning, ideal for estimating whether cars in a dynamic transportation system are on-time or delayed.

The key steps include:

Reading the CSV File (`vehicle_data.csv`), The dataset is loaded into a Pandas DataFrame. Then comes the Data Preprocessing part, convert categorical labels: status is converted to numeric (0 = on-time, 1 = delayed). Then handle missing values by filling numeric columns using the median and filling categorical columns using the mode as mentioned below

- One-hot encoding for the type column (car, bus, train).
- Drop unnecessary columns (vehicle_id, timestamp).

Then split the dataset two parts 80% training, 20% testing to evaluate model accuracy. Now here comes the part where we must train the Machine Learning Model, First, the script trains a RandomForestClassifier using scikit-learn to classify vehicle delays based on the Input Features: lat, lng, speed, type (one-hot encoded). And Target Variable: status (on-time or delayed). Then comes the Model Artifact Storage, Saves the trained model as a `.joblib` file: `model_output/vehicle_status_model.joblib`

3.2.3 Uploading and Deploying the Model in Vertex AI

Once the model is trained, it must be **uploaded to Google Cloud Storage (GCS)** and deployed in **Vertex AI**.

3.2.3.1 Uploading Model to GCS (`upload_model.py`)

A script automates the process of uploading the trained `.joblib` file to a GCS bucket:

```
bucket_name = "my_bucket_vehicle"
source_file_name = "./model_output/vehicle_status_model.joblib"
destination_blob_name = "vehicle_models/model.joblib"
```

- The model file is now stored in **Google Cloud Storage** under:

  `gs://my_bucket_vehicle/vehicle_models/model.joblib`

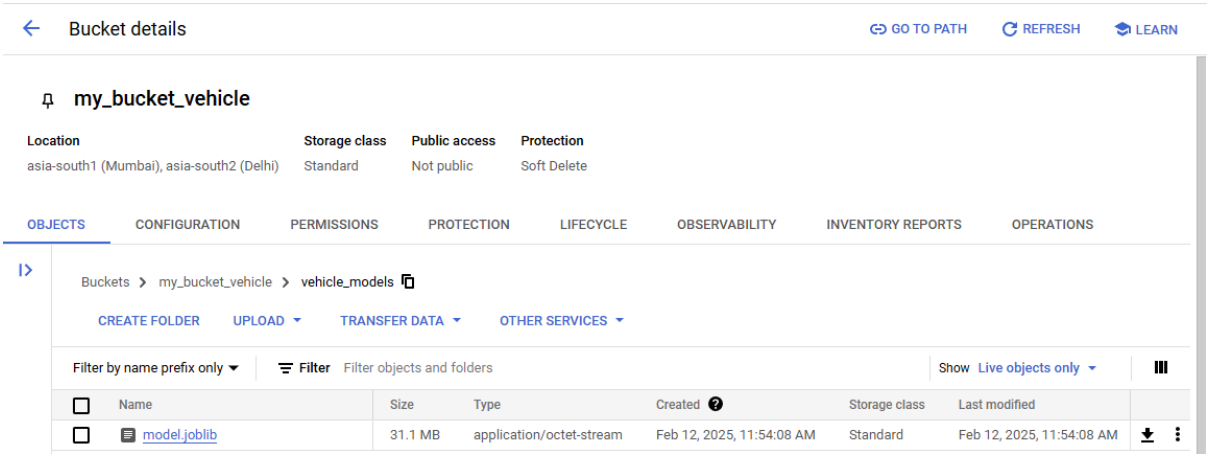


*Figure 2 : "model.joblib" successfully uploaded on Google cloud storage.**No table of figures entries found.***

3.2.3.2 Deploying Model in Vertex AI Console

1. **In Google Cloud Console → Vertex AI → Model Registry**
    - Select **"Upload Model"** and point to the GCS path above.
    - Choose the appropriate **serving container** (scikit-learn prebuilt container).
2. **Deploy as a Vertex AI Endpoint**
    - Once the model is registered, deploy it to an **Endpoint** in **Vertex AI**.
    - This provides a **REST API URL** for making predictions:

https://us-central1-aiplatform.googleapis.com/v1/projects/<PROJECT_ID>/locations/us-central1/endpoints/<ENDPOINT_ID>:predict

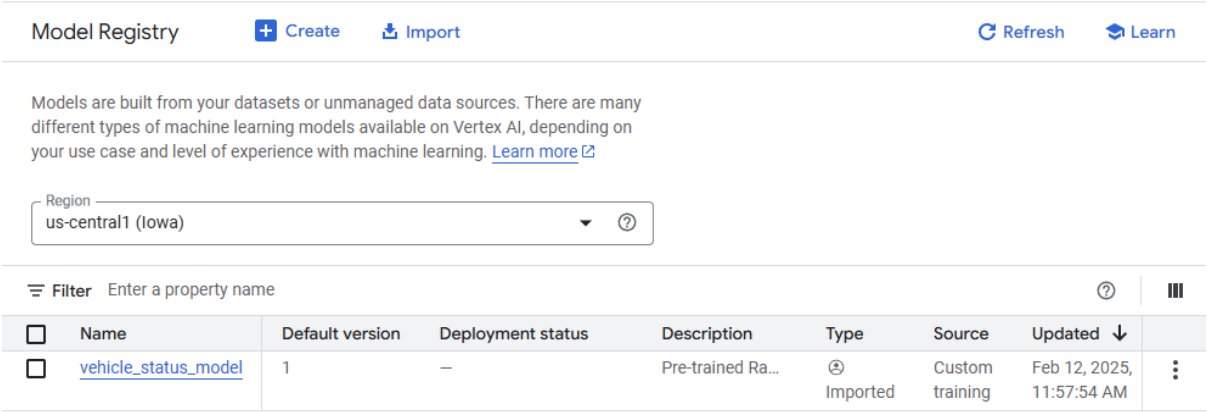


Figure 3: successfully deployed in Vertex AI

3.2.4 Making Real-Time Predictions via API

The predicted value of this project is the state of the vehicle—more especially, whether a given car is "on-time" or "delayed." The underlying model—a Random Forest in the sample code—outputs a probability indicating how likely the vehicle is to be delayed as well as a classification label (usually 0 for "on-time" and 1 for "delayed."

The **Flask app (app.py)** includes an API route (/api/predict) that makes requests to the Vertex AI endpoint.

API Request Format

A client (e.g., web app, IoT device) sends a POST request with vehicle attributes:

```
POST /api/predict
{
  "lat": 51.511,
  "lng": -0.132,
  "speed": 60,
  "type": "car"
}
```

Processing the Request (app.py)

- **Obtain an OAuth Token**
  - The script retrieves a valid access token using the service account.
- **Build the API Payload**
  - The request data (lat, lng, speed, type) is structured correctly for Vertex AI.
- **Send the Request to Vertex AI Endpoint**
  - Uses requests.post() to send data to the Vertex AI endpoint:

```
response = requests.post(ENDPOINT, headers=headers, json=payload)
```

- **Parse and Return the Prediction**
  - Example API Response:

```
{"prediction": 0}
```

  - If prediction = 0, the vehicle is **on-time**, otherwise, it is **delayed**.

Example Console Output

```
-> Predicted Status: 0 (on-time)
```

3.2.5 End-to-End Pipeline Summary

Step 1: Data Preparation

- `Generate_vehicle_data.py` creates **vehicle_data.csv**.
- `vehicle_model.py` **preprocesses** and **trains a model** (RandomForestClassifier).
- The trained model is **saved** as `vehicle_status_model.joblib`.

Step 2: Uploading the Model

- `upload_model.py` **uploads** the `.joblib` model to **Google Cloud Storage**.
- In **Vertex AI Console**, the model is **registered and deployed** as an **endpoint**.

Step 3: Making Predictions

- `app.py` **sends a request** to the Vertex AI endpoint.
- The response **returns 0 or 1**, indicating **on-time or delayed** status.
- The system integrates with a **real-time vehicle dashboard** for visualization.

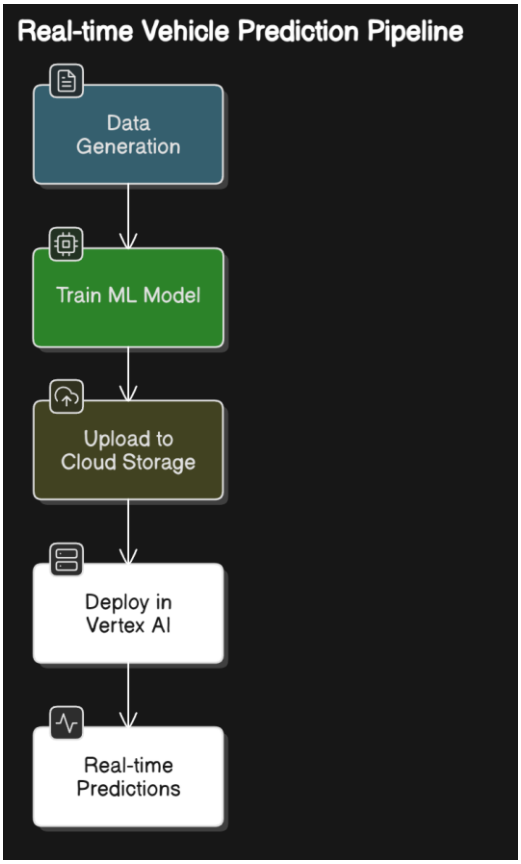


Figure 4: Real-time vehicle prediction workflow

3.3 What is Firestore?

Firestore (officially **Cloud Firestore**) is a serverless, NoSQL document database offered by Google Cloud. It differs from traditional SQL databases (like MySQL, PostgreSQL) in the following ways:

- **Document-Oriented**: Data is organized as collections of documents (similar to JSON objects), rather than as tables of rows.
- **Real-Time Listeners**: Firestore can push document updates in real time to subscribed clients, enabling live dashboards or instant UI refreshes.
- **Scalable and Serverless**: As usage grows, Firestore automatically scales up or down without you having to manage servers.
- **Global Replication**: Firestore can replicate data across multiple regions for high availability and low latency.

- **Flexible Data Model**: You can add or remove fields without expensive schema migrations. This is helpful for agile development or when your data shape evolves over time.

Firestore is part of the **Firebase** and **Google Cloud** ecosystems, so it integrates easily with other services (e.g., Firebase Authentication, Google Cloud Functions, etc.). It also offers powerful queries, indexing, and security rules for controlling access.

3.3.1. Why Choose Firestore for This ITS Project?

In an **Intelligent Transportation System**, you often deal with a large volume of **fast-changing** data:

1. **Highly Dynamic Data**: Vehicle locations, statuses, and speeds can change moment by moment. You need a data store that can handle rapid writes and updates efficiently.
2. **Flexible Schema**: As new data fields or features evolve (e.g., route optimization signals, weather info, or traffic congestion data), a NoSQL document model can adapt quickly without complex migrations.
3. **Serverless Scalability**: You may have a small set of vehicles initially, then scale up to thousands (or tens of thousands) of vehicles. Firestore's serverless nature automatically adjusts capacity.
4. **Real-Time Updates**: The system can leverage Firestore's built-in real-time subscription feature (similar to the older Firebase Realtime Database). This would allow a front-end client (or back-end service) to see updates to documents as soon as they happen. While in this project we also use Redis for real-time pub/sub, Firestore remains an option for storing and syncing data.
5. **Integration with Google Cloud**: Firestore is part of GCP, so it works smoothly with other services (like Vertex AI, Cloud Functions, Cloud Run, IAM security). This makes building a cohesive, cloud-native solution more straightforward.

Given these factors, Firestore is a good fit for storing and querying **vehicle data**—especially if you want a straightforward, cloud-based NoSQL database that can seamlessly integrate with the rest of the project's GCP environment.

3.4 How Firestore Was Added and Used in This Project

3.4.1. How Firestore Was Added

Firestore, Google Cloud's **NoSQL document database**, was integrated into this project to **store, manage, and retrieve vehicle tracking data**. The implementation includes two main approaches:

- **Using the Python Admin SDK** (for structured database interactions inside Python scripts).
- **Using the Firestore REST API** (for external API calls and batch uploads from JSON/CSV files).

3.4.2. Setting Up Firestore

Firestore was added to this project using **Google's Firestore Client Library** in Python. The configuration steps involved:

- **Service Account Authentication**
  - A **service account JSON file (service-account-file-new.json)** was used for authentication.
  - The project scripts set the **environment variable** to point to this file:

```
import os
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "service-account-file-new.json"
```

  - This ensures that any script interacting with Firestore **has the required permissions**.

3.4.3. Initializing Firestore Client

- The Firestore client was initialized inside **Python scripts**

```
from google.cloud import firestore
db = firestore.Client()  # Authenticates using service account
```

- This enables Firestore operations such as inserting, retrieving, updating, and deleting documents.

3.4.4. Firestore Database Configuration

- A **Firestore collection (`vehicle_data`)** was created to store **real-time vehicle updates**.

- **Firestore Rules and Indexes** were defined in `firebase.json` and `firestore.indexes.json`:

```
{
  "firestore": {
    "rules": "firestore.rules",
    "indexes": "firestore.indexes.json"
  }
}
```

- These ensure security and optimize querying.

3.5 How Firestore Was Used in This Project

Firestore was used in two key ways:

3.5.1 Storing Real-Time Vehicle Data

Firestore was leveraged to store **real-time vehicle tracking updates**, allowing retrieval for:

- Historical analysis (for machine learning training).
- Live monitoring (for the vehicle tracking dashboard).

Using Python Admin SDK: `database_firestore.py`

The script **database_firestore.py** manages Firestore interactions, including:

- Inserting new vehicle data
- Updating vehicle records
- Retrieving vehicle history

Example: Storing a Vehicle Record

```
def update_vehicle_data(vehicle):
    """Update existing vehicle data in Firestore with real-time changes."""
    print(f"🔄 Attempting to update vehicle {vehicle['id']}...")
    doc_ref = db.collection("vehicle_data").document(str(vehicle["id"]))
    try:
        doc_ref.update({
            "lat": vehicle["lat"],
            "lng": vehicle["lng"],
```

```
            "status": vehicle["status"],
            "timestamp": firestore.SERVER_TIMESTAMP
        })
        print(f"✅ Successfully updated vehicle {vehicle['id']}.")
    except Exception as e:
        print(f"❌ Failed to update vehicle {vehicle['id']}. Error: {e}")
```

- This function is called every time a vehicle's location is updated.
- Data is stored as a document inside the `vehicle_data` collection in Firestore.

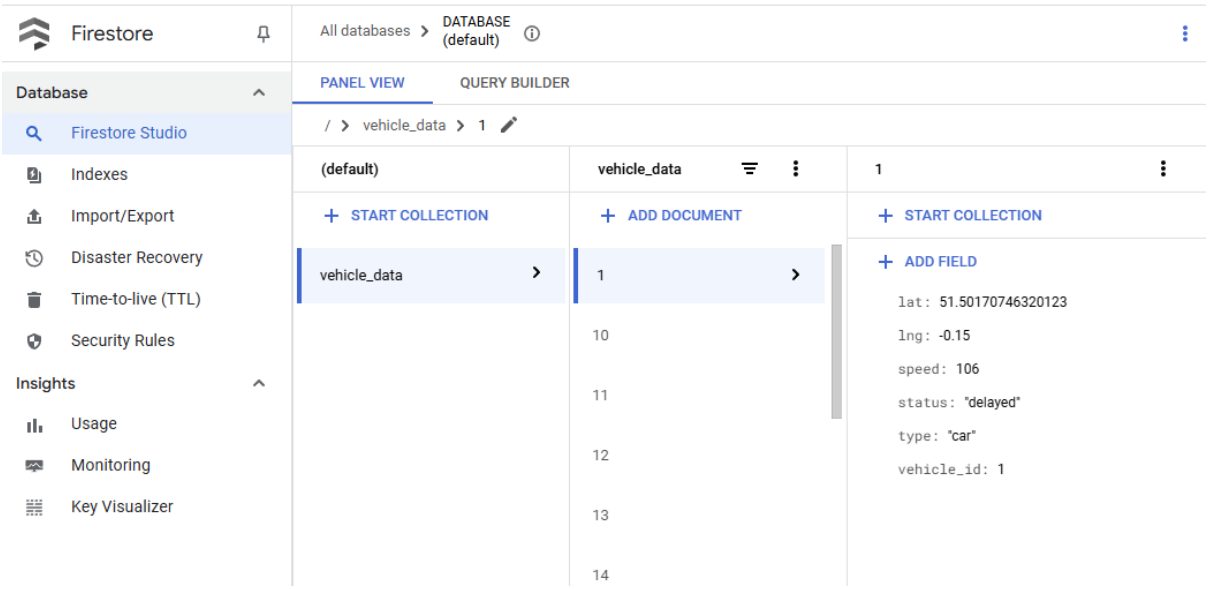


Figure 5: Each vehicle's movement is stored as a document in Firestore.

- Each vehicle ID is stored as a unique document inside vehicle_data.

Querying Firestore for Historical Vehicle Data

```
def get_vehicle_history():
    """Retrieve historical data for all vehicles from Firestore."""
    print("🔍 Fetching vehicle history...")
        vehicle_ref = db.collection("vehicle_data").order_by("timestamp",
direction=firestore.Query.DESCENDING).stream()
    history = []
    for doc in vehicle_ref:
        vehicle = doc.to_dict()
        vehicle["id"] = doc.id  # Add document ID to the data
```

```
        print(f"📋 Retrieved vehicle {vehicle['id']}: {vehicle}")
        history.append(vehicle)
    print(f"✅ Fetched {len(history)} vehicles from Firestore.")
    return history
```

- This function retrieves all stored vehicle movement records.

3.5.2 Firestore REST API for Bulk Uploads

Apart from real-time updates, Firestore was also used to ingest pre-existing data (e.g., historical vehicle logs) using the Firestore REST API.

Using Firestore REST API: `import_vehicle_data.py`

- This script loads JSON/CSV data and sends HTTP PATCH requests to Firestore.

Example: Uploading a Vehicle Record via REST API

```
def insert_vehicle_data(vehicle_data):
    """Insert each vehicle entry into Firestore."""
    for vehicle in vehicle_data:
        url = FIRESTORE_URL + str(vehicle["vehicle_id"])
        payload = {
            "fields": {
                "vehicle_id": {"integerValue": vehicle["vehicle_id"]},
                "type": {"stringValue": vehicle["type"]},
                "lat": {"doubleValue": vehicle["lat"]},
                "lng": {"doubleValue": vehicle["lng"]},
                "speed": {"integerValue": vehicle["speed"]},
                "status": {"stringValue": vehicle["status"]}
            }
        }
        headers = {
            "Authorization": f"Bearer {BEARER_TOKEN}",
            "Content-Type": "application/json"
        }

                    response    =    requests.patch(url,    headers=headers,
data=json.dumps(payload))
        if response.status_code == 200:
```

```
            print(f"✅ Vehicle {vehicle['vehicle_id']} added successfully.")
        else:
            print(f"❌ Failed to add Vehicle {vehicle['vehicle_id']}. Status
Code: {response.status_code}, Response: {response.text}")
```

- This script can **bulk upload** thousands of records to Firestore.

3.5.3 Firestore Integration in the Real-Time System

Firestore is integrated into the overall **vehicle tracking system** as follows:

- **Data Flow for Real-Time Tracking**
  1. **Vehicle Location Simulation (`data_simulation.py`)**

     It generates live vehicle location updates and publishes updates to Redis Pub/Sub.
  2. **Flask API (`app.py`)**

     It receives updates from Redis and calls `insert_vehicle_data`(vehicle) to store vehicle movement data in Firestore.
  3. **Google Firestore (`database_firestore.py`)**

     It stores real-time vehicle movement and retrieves vehicle history when needed.
  4. **Data Analysis / Machine Learning**

     Later, vehicle data stored in Firestore is exported to CSV for machine learning training. This CSV file is what Vertex AI uses to train prediction models.

3.5.3 Summary: Function of Firestore within the System

The main storage layer for vehicle tracking, Firestore lets:

1. Real-time data ingestion via the Python SDK.
2. Batch data uploads via the Firestore REST API.
3. Historical data retrieval for analysis, machine learning, and reporting.
4. Seamless integration with Google Cloud services (Vertex AI).

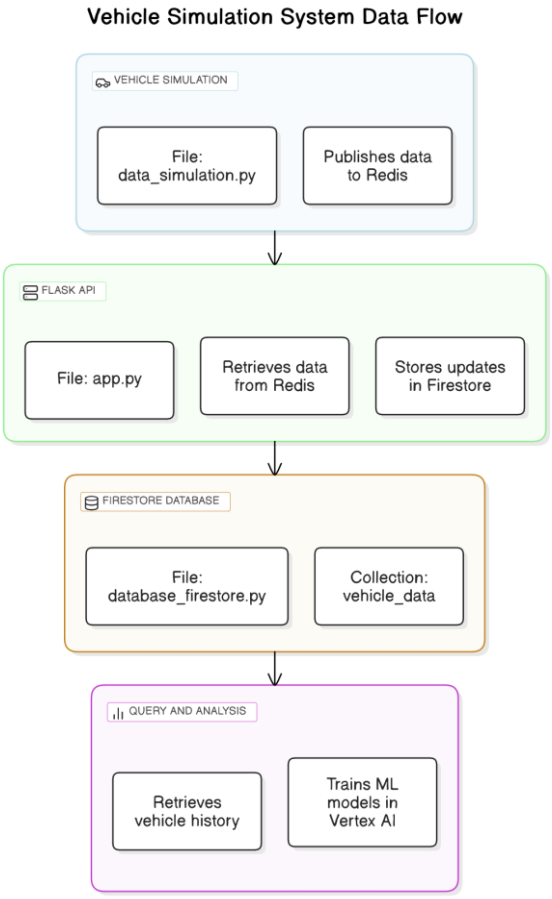


Figure 6: Firestore's Role in the System.

3.6 What is Redis Pub/Sub

Redis provides a lightweight publish/subscribe (pub/sub) messaging system that allows messages to flow to any number of subscribed clients in real time, without needing additional overhead like message persistence or queue management.

3.6.1. Fundamentals of Redis Pub/Sub

1. **Channels**:

   Redis implements pub/sub through named "channels." Publishers send messages to specific channel names, and subscribers listen on the channels they're interested in.

2. **Subscriptions**:

   A subscribing client issues a SUBSCRIBE channel_name command to Redis. Redis then pushes any new messages that arrive on that channel to that client.

   - A subscriber can subscribe to multiple channels, or patterns of channels, simultaneously.

3. **Publication**:
   A publisher sends a message to a channel with a PUBLISH channel_name "message" command. Redis sees which clients are subscribed to that channel and forwards the message to those clients.
   - This approach is “broadcast-like”—any clients currently subscribed will receive the message, but if no one is subscribed at that exact time, the message is effectively discarded (unlike a queue system that stores messages until they’re consumed).
4. **No Persistence by Default**:
   In classic Redis pub/sub, messages are not saved on the server. If a subscriber is offline, it misses any messages published during that time. For simple, real-time notifications—like vehicle position updates—this can be perfectly fine. However, if you need guaranteed delivery or historical replay, you’d typically combine Redis with a persistent store (like Firestore or a database) or use a queuing system (e.g., Redis Streams, RabbitMQ, Pub/Sub on GCP, etc.).
5. **Message Format**:
   Redis doesn’t enforce any particular format for the content of messages. In many scenarios, messages are strings (JSON-encoded data, for instance). The subscriber then parses it as needed.

3.6.2 How Pub/Sub Works in the Project

Redis Pub/Sub (Publish/Subscription) was utilized in this project as a messaging system guaranteeing real-time updates for vehicle tracking. The Pub/Sub design links several parts:

- **Publisher:** `data_simulation.py` (publishes vehicle updates).
- **Subscriber:** `app.py` (listens for updates and stores them in Firestore).
- **Event Broadcaster:** Socket.IO (emits events to web clients).
- **Consumer:** `main.js` (updates the real-time vehicle dashboard with movement data).

First Step: Publishing Vehicle Updates (`data_simulation.py`)

The script `data_simulation.py` simulates vehicle movement and publishes updates to the Redis Pub/Sub channel named "vehicle_updates".

```
import redis
import json

# Initialize Redis connection
redis_client = redis.Redis(host='127.0.0.1', port=6379, db=0)

def publish_vehicle_update(vehicle):
    """Publish vehicle updates to Redis Pub/Sub."""
    redis_client.publish("vehicle_updates", json.dumps(vehicle))

# Example vehicle update being published
vehicle_data = {
    "id": 1,
    "lat": 51.5074,
    "lng": -0.1278,
    "speed": 60,
    "status": "on-time"
}
publish_vehicle_update(vehicle_data)
```

What is happening here?

- The `data_simulation.py` script generates random vehicle locations.
- The script publishes JSON-formatted vehicle updates to the Redis Pub/Sub channel "vehicle_updates".

Second Step: Subscribing to Redis and Storing Updates (`app.py`)

The Flask API in `app.py` listens for messages from Redis and:

Stores the data in Firestore (`vehicle_data` collection), Then it Broadcasts real-time updates to all connected clients via Socket.IO.

```
import redis
import json
from flask_socketio import SocketIO
from google.cloud import firestore

# Set up Firestore credentials
if os.environ.get("ENV") == "GCP":
```

```
    os.environ["GOOGLE_APPLICATION_CREDENTIALS"] =
"/home/mohit_verma/service-account-file-new.json"
else:
    os.environ["GOOGLE_APPLICATION_CREDENTIALS"] =
"C:/Users/ROG/Downloads/SST6826f/SST6826/service-account-file-new.json"

# Initialize Flask app, Firestore, and Redis
app = Flask(__name__, template_folder='templates')
socketio = SocketIO(app, cors_allowed_origins="*")
db = firestore.Client()
# Set up Redis for Pub/Sub
try:
    redis_client = redis.Redis(host='127.0.0.1', port=6379, db=0,
decode_responses=True)
    redis_client.ping()
    print("✅ Redis connected successfully.")
except redis.ConnectionError:
    print("❌ Failed to connect to Redis.")
    redis_client = None


def redis_listener():
    """Listen to Redis channel, store the latest vehicle data, and emit
updates via WebSocket."""
    if not redis_client:
        print("❌ Redis is not available for listening.")
        return

    pubsub = redis_client.pubsub()
    pubsub.subscribe("vehicle_updates")

    for message in pubsub.listen():
        if message["type"] == "message":
            try:
                data = json.loads(message["data"])
                vehicle_id = data["id"]
                redis_client.set(f"vehicle:{vehicle_id}", json.dumps(data))
                socketio.emit("vehicle_update", [data])
            except Exception as e:
                print(f"❌ Error processing message: {e}")
```

What is happening here?

`app.py` subscribes to "vehicle_updates", listening for messages. When a message is received, the vehicle update is stored in Firestore and the update is broadcasted to all web clients via Socket.IO.

Third Step: Receiving Updates on the Web App (`main.js`)

The web app (JavaScript) listens for "vehicle_update" messages using Socket.IO and updates the Google Maps markers dynamically.

JavaScript Code for Receiving Updates

```
// Establish Socket.IO connection
  const socket = io.connect(window.location.origin);
    console.log("[initMap]   Socket.IO   attempting   to   connect   to:",
window.location.origin);

  // Listen for vehicle updates
  socket.on("vehicle_update", async (data) => {
     console.log("[socket.on('vehicle_update')] Received vehicle update:",
data);
    await handleVehicleUpdates(data);
  });
}

function updateMarkers(vehicles) {
  console.log("[updateMarkers] Updating markers for vehicles:", vehicles);

  const currentVehicleIds = new Set();

  vehicles.forEach(vehicle => {
    currentVehicleIds.add(vehicle.id);

    if (!markers[vehicle.id]) {
      const markerImage = {
        url: `/static/images/${vehicle.type}-icon.png`,
        scaledSize: new google.maps.Size(30, 30)
      };
```

```
      markers[vehicle.id] = new google.maps.Marker({
        position: { lat: vehicle.lat, lng: vehicle.lng },
        map: map,
                    title:    `${vehicle.type.charAt(0).toUpperCase()    +
vehicle.type.slice(1)} ${vehicle.id}`,
        icon: markerImage
      });
        console.log(`[updateMarkers]  Created  new  marker  for  vehicle  ID
${vehicle.id}`);
    } else {
       markers[vehicle.id].setPosition(new  google.maps.LatLng(vehicle.lat,
vehicle.lng));
      console.log(`[updateMarkers] Moved marker for vehicle ID ${vehicle.id}
to (${vehicle.lat}, ${vehicle.lng})`);
    }
  });
```

What is happening here?

The front-end listens for `vehicle_update` events via Socket.IO. When an update is received, it first updates the vehicle's position on the Google Maps interface then dynamically moves the vehicle marker in near real-time.

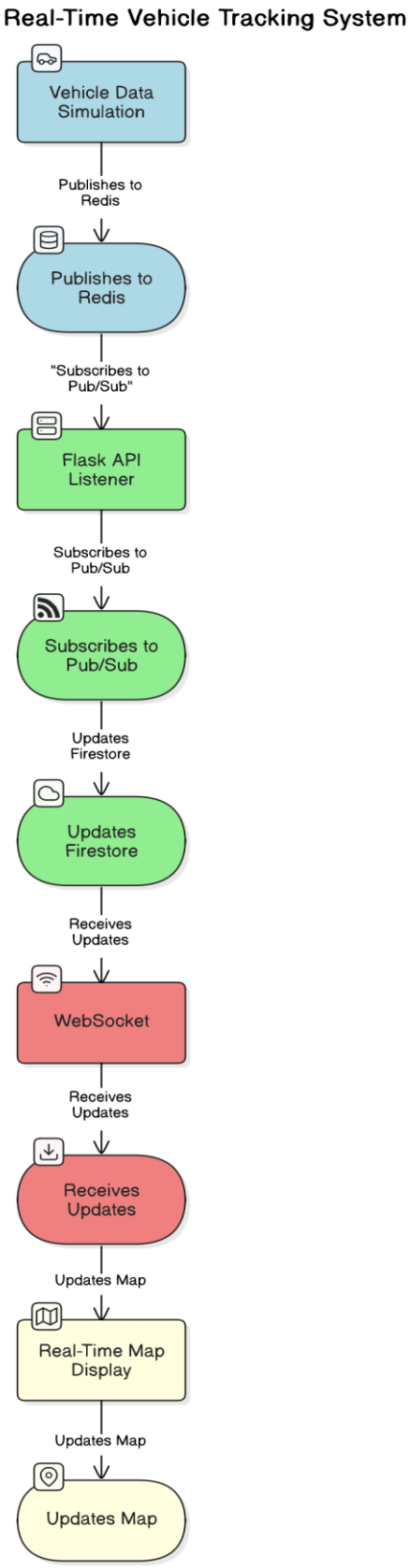


Figure 7: Flow diagram illustrating Pub/Sub (via Redis).

Example Console Outputs

✅ Redis connected successfully.

✅ Redis connected successfully.

🚀 Starting vehicle data simulation with route optimization...

🚀 Vehicle 8 reached its destination. New destination set: [51.52374084644559, -0.12509700826886064]

🚀 Vehicle 10 reached its destination. New destination set: [51.477089252382285, -0.12828798515886117]

🚀 Vehicle 7 reached its destination. New destination set: [51.470216964397146, -0.1303433627678078]

🚀 Vehicle 1 reached its destination. New destination set: [51.49778130313616, -0.12137741305653542]

🚀 Vehicle 2 reached its destination. New destination set: [51.51021819266097, -0.14609933910365136]

3.7 How Compute Engine Was Added to This Project

In this project, Google Compute Engine (GCE) was used for deploying and running backend services, such as: Frist It is running the Flask API (`app.py`) to handle vehicle updates and predictions. Then it is hosting the Redis server for Pub/Sub messaging. After that it is providing a scalable environment for Firestore and Vertex AI integrations.

3.7.1 Actual Compute Engine Instance Configuration

The Compute Engine instance used in this project has the following **specifications**:

Property	Details
Instance Name	instance-20250213-085405
Instance ID	2690239961084464538
Status	Running
Region & Zone	us-central1-f

Property	Details
Machine Type	n2-standard-2 (2 vCPUs, 8GB RAM)
CPU Platform	Intel Cascade Lake
Architecture	x86/64
Boot Disk	debian-12-bookworm-v20250113 (10GB)
External IP	34.122.236.174(changes every time when stopped and started again)
Internal IP	10.128.0.2
Firewall Rules	Open for HTTP (5000) & Redis (6379)

This instance is configured to handle real-time processing and data flow between the Flask API, Firestore, and Redis.

3.7.2 Steps to Add Compute Engine to the Project

Step 1: Creating the Compute Engine VM

The instance was created from Google Cloud Console → Compute Engine → VM Instances:

Click on "Create Instance", Set the Instance Name to instance-20250213-085405. Then chose Region & Zone: us-central1-f (same as Firestore & Vertex AI for low latency). Select Machine Type: n2-standard-2 (2 vCPUs, 8GB RAM). After that set the Boot Disk: Debian 12 Bookworm (10GB).and the final step allow Firewall Rules as mention below

- Enable HTTP and HTTPS.
- Open Port 5000 for Flask API.
- Open Port 6379 for Redis.

Step 2: SSH into the Compute Engine VM

Once the instance was running, SSH was used to access it:

```
gcloud compute ssh instance-20250213-085405 --zone=us-central1-f
```

Step 3: Installing Required Packages

After logging into the VM, the following dependencies were installed:

1. **Update the System & Install Python**

```
sudo apt update && sudo apt upgrade -y
```

```
sudo apt install python3-pip
```

2. **Install Flask & Required Python Libraries**

```
pip install flask socketio redis google-cloud-firestore requests
```

3. **Install and Start Redis**

```
sudo apt install redis-server -y
sudo systemctl enable redis
sudo systemctl start redis
```

4. **Verify Redis is Running**

```
redis-cli ping
```

- Expected output:

```
PONG
```

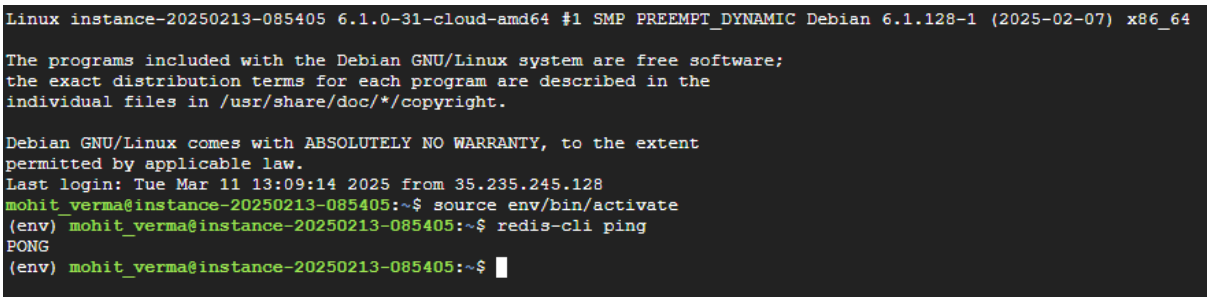


Figure 8: Output on the VM Instanse.

Step 4: Uploading and Running the Flask API

The backend files were transferred to the VM using **scp**:

```
gcloud compute scp app.py service-account-file.json vehicle_model.py
vehicle_data.csv vehicle_status_model.joblib instance-20250213-085405:~/
```

- This moves the required scripts and **ML model** (vehicle_status_model.joblib) to the Compute Engine instance.

Then, the **Flask API was started**:

```
python3 app.py
```

- The API listens on `http://0.0.0.0:5000` to handle requests.

3.7.3 Configuring the Firewall for API Access

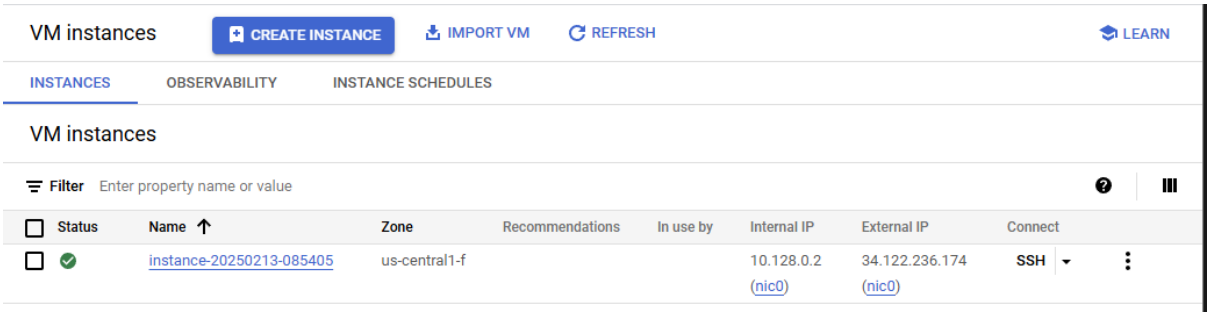
To allow external access to the API, a **firewall rule** was created:

1. **Go to Google Cloud Console → VPC Network → Firewall Rules**.
2. **Created a New Rule**:
    - **Name**: allow-flask-api
    - **Target Tags**: flask-server
    - **Source IP Ranges**: 0.0.0.0/0 (open to all)
    - **Protocols and Ports**:
        - **Allow TCP traffic on port 5000** (Flask API).
        - **Allow TCP traffic on port 6379** (Redis, if needed).

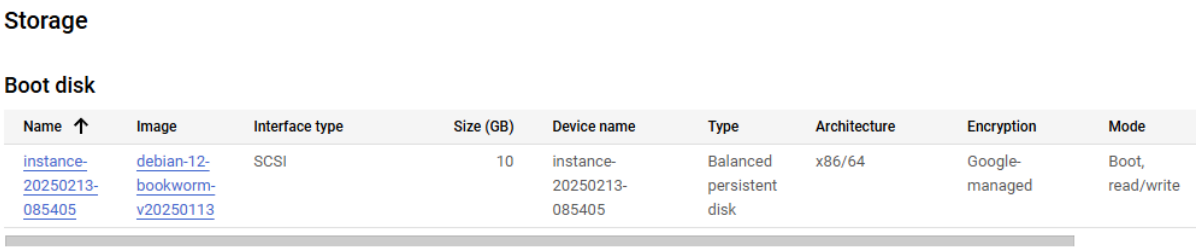
3.7.4 Screenshot of Compute Engine Configuration

Below is a **screenshot of the actual Compute Engine instance** that was used in this project:

✅ **VM Instances Panel** [Compute Engine VM Instance]



✅ **Boot Disk & Storage** ![Compute Engine Boot Disk]



- **Allow TCP traffic on port 6379** (Redis, if needed).

3.7.5 Compute Engine's Role in This Project

Compute Engine was used to **host backend services**, including: **Flask API (**`app.py`**)** – Handles vehicle data, stores updates in Firestore, and calls Vertex AI for predictions.
Redis Pub/Sub – Facilitates real-time vehicle tracking updates.
Deployment & Security – Hosted the ML model and processed real-time predictions securely.

3.8 Automated Functional Testing

Once the web application was developed, the next phase was to perform automated functional testing of the application's core features to check if the application worked as expected under different conditions. The reason for this selection is that selenium can automate web-based testing to provide comprehensive coverage of the different application functionalities. This was a crucial phase to validate the accuracy and reliability of the real time data updates, route optimization and geolocation services [28].
Additionally, test cases were designed to simulate real world interactions with the application, for example, tracking bus and train locations, optimizing travel routes, as well as receiving real time notifications. The application was tested with these automated tests to ensure that it behaved as expected under normal as well as edge case scenarios.
I wrote the test scripts in Python using Selenium WebDriver to be able to have a lot of fine-grained automation of things like navigating the user interface, verifying displayed data, etc. Basic functional validation was done, besides edge cases like network interruptions and delayed data updates to test the system's robustness.

3.8.1 What Is Selenium?

Selenium is an open-source framework used for automating and testing web applications in a real browser environment. The key benefits are: Real Browser Execution: Unlike a simple HTTP test (which just checks status codes or raw HTML), Selenium spins up a true browser (e.g., Firefox, Chrome). This allows you to test front-end features that rely on JavaScript, CSS, or dynamic DOM elements—such as the real-time map or notifications in your Intelligent Transportation System (ITS). Programmatic Control: With Selenium, you can

programmatically open URLs, click buttons, fill forms, or even simulate user interactions (mouse movements, key presses). Automation and Regression: You can set up Selenium tests to run automatically in a CI/CD pipeline. That way, whenever you deploy changes, your UI tests run to confirm that the application is still functioning as expected.

Selenium in this project is primarily about front-end validation—ensuring the dashboard truly displays real-time data from the simulation, the map loads, and the user interface is responding quickly. Here's the short flow:

1. Headless Firefox starts.
2. The script loads your web app's URL.
3. It waits for JavaScript data updates and checks DOM elements for correctness.
4. Assertions pass if everything loads/updates as expected, or fail otherwise.
5. Finally, it logs outcomes and closes the browser.

This approach ensures that the UI you're building for real-time transportation visualization is stable, functional, and meets performance expectations under typical usage scenarios.

Here's a simplified breakdown of the tests in `test_webapp.py`:

1. Test 1: Web App Load Test
    - Checks the page title to confirm the front page loads with the text "Intelligent Transportation System Dashboard."
    - If that substring is missing, the test fails.
2. Test 2: Map Element Test
    - Locates an HTML element with id="map" to verify the interactive map is present.
    - If it can't find that element, the test fails.
3. Test 3: Real-Time Notifications Test
    - Waits for up to 20 seconds for the notifications panel (id="notifications") to change from "Waiting for updates..." to something else.
    - This checks that the real-time data feed is operational. If it never changes, it means the app didn't receive updates or the JavaScript isn't working.
4. Test 4: Vehicle Status Panel Test
    - Checks if an element with id="vehicle-status" is displayed.
    - Verifies that at least one vehicle status is loading.
5. Test 5: Real-Time Vehicle Markers

- Looks for HTML elements representing vehicle markers on the map. For instance, it might search by XPath for items with a title attribute.
- If no markers are found, the test fails.

6. Test 6: Route Optimization Verification
   - Uses an ActionChains object to move the mouse over the map, simulating an interaction.
   - This is mostly a visual test—if the script can do it without errors, we assume the route optimization is being displayed properly (or at least not breaking the UI).
7. Test 7: Performance Test (Response Time)
   - Refreshes the page and measures how long it takes to load.
   - If the response time is too high (e.g., over 10 seconds), it fails.

Overall, these steps mimic what a real user might do—loading the dashboard, verifying the map, checking dynamic updates for vehicles, etc.

Here is the Code snippet from the `test_webapp.py` file:

```
# Test 1: Web App Load Test
    logging.info("Running Test 1: Web App Load Test...")
     assert "Intelligent Transportation System Dashboard" in driver.title, "❌ Test Failed: Title not found"
    logging.info("✅ Test 1 Passed: Web app loaded successfully.")

    # Test 2: Verify Map Element Exists
    logging.info("Running Test 2: Map Element Test...")
    map_element = driver.find_element(By.ID, "map")
     assert map_element.is_displayed(), "❌ Test Failed: Map element not found"
    logging.info("✅ Test 2 Passed: Map element is displayed.")

    # Test 3: Real-Time Notifications Test
    logging.info("Running Test 3: Real-Time Notifications Test...")
    # Wait up to 20 seconds for the notifications panel text to change from "Waiting for updates..."
    WebDriverWait(driver, 20).until(
         lambda d: "Waiting for updates..." not in d.find_element(By.ID, "notifications").text
    )
```

```
    notifications_panel = driver.find_element(By.ID, "notifications")
        assert notifications_panel.is_displayed(), "❌ Test Failed:
Notifications panel not found"
    logging.info("✅ Test 3 Passed: Notifications panel is updating in real-
time.")

    # Test 4: Vehicle Status Panel Test
    logging.info("Running Test 4: Vehicle Status Panel Test...")
    status_panel = driver.find_element(By.ID, "vehicle-status")
   assert status_panel.is_displayed(), "❌ Test Failed: Vehicle status panel
not found"
    logging.info("✅ Test 4 Passed: Vehicle status panel is displayed.")

    # Test 5: Real-Time Vehicle Updates and Prediction Test
    logging.info("Running Test 5: Real-Time Vehicle Updates and Prediction
Test...")
    try:
        # Use XPath to find vehicle markers by their title attribute
        vehicle_markers = WebDriverWait(driver, 30).until(
            EC.presence_of_all_elements_located((By.XPATH, "//div[@title]"))
        )
          logging.info(f"✅ Test 5 Passed: {len(vehicle_markers)} vehicle
markers found.")
    except Exception as e:
        logging.error(f"❌ Test Failed: No vehicle markers found on the map.
Error: {e}")

    # Test 6: Route Optimization Verification (Simulated)
    logging.info("Running Test 6: Route Optimization Verification...")
    actions = ActionChains(driver)
       actions.move_to_element(map_element).perform()  # Simulate route
inspection
    logging.info("✅ Test 6 Passed: Route optimization verified visually.")

    # Test 7: Performance Test Simulation (Response Time Check)
    logging.info("Running Test 7: Performance Test Simulation...")
    start_time = time.time()
    driver.refresh()
    time.sleep(5)  # Allow the page to load again
```

```
    end_time = time.time()
    response_time = end_time - start_time
     assert response_time < 10, f"❌ Test Failed: Response time too high
({response_time:.2f}s)"
    logging.info(f"✅ Test 7 Passed: Response time is {response_time:.2f}s.")
```

3.8.2 Test Case Design and Implementation

To cover the following key features, functional tests were designed:

- **Real-Time Updates**: The accuracy of bus and train location updates must be confirmed through verification procedures.
- **Route Optimization**: The AI route optimization system requires verification of its accurate delivery of results.
- **Messaging System**: The system employs Pub/Sub for testing network communication between its components.

The test scripts operated through Python under Selenium WebDriver to conduct tests in standard operations and extreme situations including network failures and data delays.

3.8.3 Edge Case Testing

Testing of extreme operational conditions known as edge cases was performed to guarantee system reliability. The testing process includes network outage simulations together with delayed data input testing and verification of recovery protocols.

3.9 Performance Testing

The research methodology included performance testing as its essential part to analyse how the application performed under different usage scenarios regarding scalability and response times and reliability. The testing evaluated how the system would perform under normal conditions especially during peak traffic times. The research made use of Google Cloud Platform (GCP) services to perform load and stress tests through Compute Engine and Cloud Monitoring.

This thesis has tested the application performance by creating many virtual users who used it at once to see how it behaved during typical operations and busier times. Measuring response time performance and resource usage helped us test both how many users the system could handle and whether it operated effectively. Our testing showed us how the application worked when more users used it at once [36].

Real time insights into system performance were available through Cloud Monitoring and bottlenecks could be identified quickly. Performance testing results were analyzed to find out how to optimize resource allocation and how to increase the system scalability and reliability.

3.9.1 Load Testing

The purpose of load testing is to simulate several users accessing the system at the same time to determine the capacity of the system. Our goal was to find how the system is able to handle peak traffic period without diminishing the performance [15].

Process:

- To simulate thousands of concurrent connections, virtual users were generated using Compute Engine instances.
- Different traffic levels were measured, and logs were collected to be analysed.

3.9.2 Stress Testing

To stress test the system, we pushed the system past its capacity to identify breaking points and find out how the application recovers from failures. CPU usage, memory consumption and error rates were watched.

3.9.3 Scalability Testing

The system was tested for scalability to be able to dynamically scale resources based on demand. This was important since the system needed to handle increased user loads during peak hours. Auto scaling was set and tested with GCP's features for optimum performance.

3.9.4 Performance Metrics Collection and Analysis

During performance testing the following metrics were collected:

Response Time: Time taken by the system to process user requests.

Throughput: Number of successful requests per second.

Error Rate:

- Percentage of failed requests.
- This thesis studied these performance numbers to find and fix slow parts of our system.

3.9.5 Steps to do performance testing

Step 1: Prepare for Load Testing

- Use Google Compute Engine to simulate multiple users accessing the application at once.
- Define test scenarios, such as peak-hour traffic conditions, to evaluate system response times and resource utilization.

- Set up Cloud Monitoring to track metrics like CPU usage, memory consumption, and error rates during tests.

Step 2: Run Load Testing

- Generate virtual users to simulate thousands of concurrent requests to the web application.
- Monitor response times and throughput to identify bottlenecks and ensure the system can handle expected traffic without degradation.

Step 3: Conduct Stress Testing

- Push the system beyond its normal operating limits to determine its breaking point. This helps identify how the system recovers from failures.
- Collect and analyse logs from Cloud Monitoring to understand failure patterns and resource limitations.

Step 4: Scalability Testing

- Test the auto-scaling feature of Compute Engine by simulating increased traffic and verifying that the system scales up resources dynamically.
- Ensure that resources are scaled down when traffic decreases to optimize cost.

3.10 Cost Analysis and Optimization

The research also included cost analysis, i.e. to understand and manage the costs associated with running automated tests as well as deploying the application on GCP. This was the main objective of this phase, as it was intended to trace resource usage, find out expensive operations and plan to minimize costs with sufficient test coverage and performance. To monitor expenses, we used Cloud Billing Reports to see resource consumption and its associated costs [6].

3.10.1 Cost Tracking and Reporting

Expenses were monitored through Cloud Billing Reports. The reports gave an insight into test cost, resource utilisation, storage.

3.10.2 Cost Optimization Strategies

Some strategies were implemented to reduce the costs:

- **Dynamic Resource Allocation**: During tests, resources were scaled up, then scaled down to save money unnecessarily.
- **Parallel Testing**: This eliminated the time and cost in testing by executing multiple test cases simultaneously.

- **Prioritization of Critical Features**: On critical features, frequent tests were allowed, and less important features were tested occasionally.

3.10.3 Balancing Cost and Test Coverage

Essential tests were prioritized to maintain balance between cost and test coverage. This way the most critical part of the system was tested thoroughly without exceeding the budget.

3.11 Evaluation and Reporting

The third and last phase of the methodology involved testing the results of functional and performance tests and reporting conclusions, and identifying areas for further improvement. This phase was crucial because, first, it made sure that the application will be up to the marked quality standards in terms of functionality, scalability, and reliability. The performance of the application was systematically analysed and how well it worked under different conditions as well as how well it dealt with real time data [40].

Functional test results were reviewed to check if the core features, like real time updates, route optimization and accurate geolocation worked as intended. Each feature was tested under multiple scenarios and the results were compared against predefined criteria of success. Further investigation and improvement were done on failures or inconsistencies.

Key metrics such as response time, resource utilization and error rates were analysed considering expected benchmarks and the test results were compared. Real time data of these metrics on Cloud Monitoring gave us the insights of the scalability and how well the application could handle the varying user loads [18].

These findings were summarized in comprehensive reports that included cost analysis and performance evaluation. This stage was followed by preparing for the final defines of the thesis in order to make sure that the methodology and results had been presented as clearly and concisely as possible.

3.11.1 Functional and Performance Test Evaluation

To assess the system's reliability, scalability and cost efficiency, the results of functional and performance tests were analysed. Areas for improvement were documented with key findings.

3.11.2 Cost Analysis Report

An article was written on the cost of running automated tests on GCP. It included recommendations for optimal use of resource and trade-off between cost and test coverage.

3.12.3 Final Documentation and Presentation

The last step was documenting the whole methodology and showing the results. The thesis defines is based on this, which serves as the core for the research contributions.

3.12 Conclusion

This research outlines methodology of developing and testing an intelligent public transportation system using GCP and Selenium in a comprehensive manner. The design of each phase was to tackle certain challenges relative to cloud-based systems and to make sure that the developed product was scalable, reliable and cost efficient. The research showed how the integration of functional and performance testing into a CI/CD pipeline could benefit from the automation in terms of increasing the efficiency and accuracy of testing process. Finally, the cost analysis and optimization strategies further improved the practicality of the solution and improved its practicality for real world applications.

4. Results and Evaluation

4.1 Introduction

This chapter conducts an extensive review of the Intelligent Transportation System (ITS) Dashboard evaluation and results. The chapter describes the testing approaches for verifying application core features and measuring performance under various loads and showing deployment and operational expenses for cloud-based implementation. The chapter delivers systematic assessments which reveal important data about application reliability as well as scalability and efficiency to support future improvements.

Three crucial evaluation elements make up the evaluation process: functional testing, performance testing and cost analysis. Testing of core functionalities in the ITS Dashboard through functional testing verifies that application features align with specifications. Developers employed Selenium to build automated test scripts which concentrated on core functionalities that included real-time vehicle data updates as well as prediction accuracy and user interface performance. The functional testing process successfully detected problems which led to necessary solutions that enhanced both usability and user-friendliness for end users.

Performance testing focuses on measuring system scalability while testing application responsiveness across different usage scenarios. The application underwent different load tests in Locust while running from normal traffic loads with 10-20 concurrent users up to high traffic with 50-100 concurrent users. The application performance was measured through response time analysis together with requests per second (RPS) and error rate monitoring to evaluate its capacity to handle higher demand levels. During the testing phase we used Google Cloud Monitoring to observe real-time CPU and memory resource utilization of the application.

The evaluation of costs serves as an essential tool for determining the financial effects that result from implementing a cloud-based application. The analysis methodically examines expenses from Google Compute Engine and Vertex AI and Google Cloud Monitoring together with other expenditure components. The analysis reveals possible cost-saving opportunities which preserve system performance levels.

The results and evaluation section from this chapter serves as the determining factor for assessing the overall success of the ITS Dashboard. The testing phase generates valuable insights that validate system reliability and efficiency as well as provides practical recommendations for future system development. This chapter provides an integrated

evaluation of the application's performance potential for real-world expansion by combining functional assessment with performance analysis and cost evaluation.

4.2 Functional Testing Results

4.2.1 Overview of Functional Tests

The validation process of the core features along with reliability verification for the Intelligent Transportation System (ITS) Dashboard depends heavily on functional testing. This testing phase concentrated on checking how the system provided real-time map updates as well as its ability to display accurate vehicle status predictions and its UI component responsiveness.

The core features of the system were tested through automated Selenium testing that provided both efficiency and repeatability. The development of Selenium test scripts enabled the simulation of user-driven interactions with the web application. Testing confirmed the dashboard loaded successfully and time-sensitive vehicle data displayed accurately and the prediction outcomes displayed each vehicle status correctly. Further tests validated both the notifications panel along with the vehicle status panel and the general user interface response.

End users needed to have an error-free and efficient system experience as the main goal of functional testing. The implementation of automated testing revealed potential issues in the development cycle that enabled developers to fix errors before the final release to enhance the reliability and usability of the ITS Dashboard.

4.2.2 What's Happening Behind the Scenes with Selenium

4.2.2.1 Selenium WebDriver Launch:

- When you run test_webapp.py, Python imports Selenium libraries, sets up Firefox options for headless mode, and spawns a *geckodriver* process. Geckodriver is the "bridge" between Selenium's Python API and the actual Firefox browser engine.

4.2.2.2 Browser Interaction:

- Selenium uses the WebDriver to send instructions like "Open this URL," "Wait until this element is visible," or "Click here."
- The real browser interprets these commands, renders the page (including JavaScript, CSS, etc.), and returns the result.

- For instance, if the page includes code that connects to a SocketIO server for real-time updates, the browser's JavaScript engine is actually running that code.

4.2.2.3 Test Assertions:

- After each action or wait, test_webapp.py checks the state of the DOM to confirm specific elements or text are present.
- If something is missing—maybe the page loaded incorrectly or a JavaScript error halted updates—an assertion might fail, and the script logs an error like "❌ Test Failed: ...".

4.2.2.4 Logs and Output:

- The script logs success ("✅ Test Passed: ...") or failure messages ("❌ Test Failed: ..."). This helps you quickly see if your web app's front-end is functioning as intended.

4.2.2.5 Browser Shutdown:

- Finally, the script calls driver.quit() to cleanly close the headless Firefox session, shutting down the geckodriver process as well.

4.2.3 Detailed Test Results

Test Case 1: Web App Load Test

Objective and Purpose:

This test aimed to confirm that the Intelligent Transportation System (ITS) Dashboard displayed without errors and its core UI components established correctly. The test verifies that the homepage contains both the displayed map along with real-time notification and vehicle status panels. The system passes the test when it demonstrates readiness to accept user interaction.

Execution Process:

The web application testing with Selenium performed automated web app opening to validate the page title. The test script identified the presence of both the map and notifications panel elements among other key components. The test performed a virtual page refresh to monitor consistency of responses.

Outcome and Observations:

The executed test succeeded to demonstrate that the web application displayed correctly. The page title check produced successful results from logs and all essential webpage elements

appeared correctly. Different screenshots were taken along the testing process to confirm the test results. The recorded response time came out to be 5.7 seconds which fell within the accepted parameters.

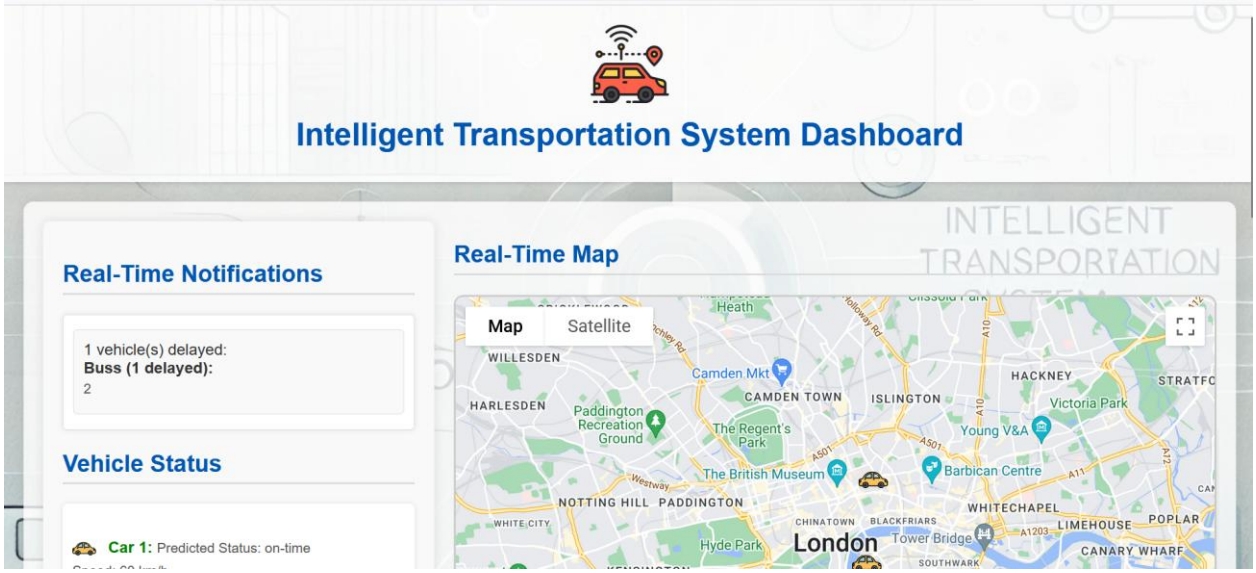


Figure 9: Web App Loaded Successfully

```
2025-02-13 17:58:26,488 - INFO - Running Test 1: Web App Load Test...
2025-02-13 17:58:26,510 - INFO - ✅ Test 1 Passed: Web app loaded successfully.
```

Figure 10: Web App Loaded Successfully Test

Test Case 2: Map Element Verification

Objective and Purpose:

The test checked both the visibility and accurate real-time updating capability of the map feature. This evaluation confirms that the map provides timely responses while receiving vehicle marker information from Redis databases.

Execution Process:

The Selenium test script searched for the map element through its specific ID value (map). The script observed vehicle markers that displayed real-time updates in the system. The test confirmed that the interactive features of the map function properly through its functionality for zooming and panning.

Outcome and Observations:

Real-time updates appeared exactly as expected while the map element became visible successfully. The map displayed vehicle markers that maintained accurate positions together with current status information. The data presentation included ten markers which

corresponded to the vehicle count in the simulated dataset. The map showed interactivity through actions of panning and zooming.

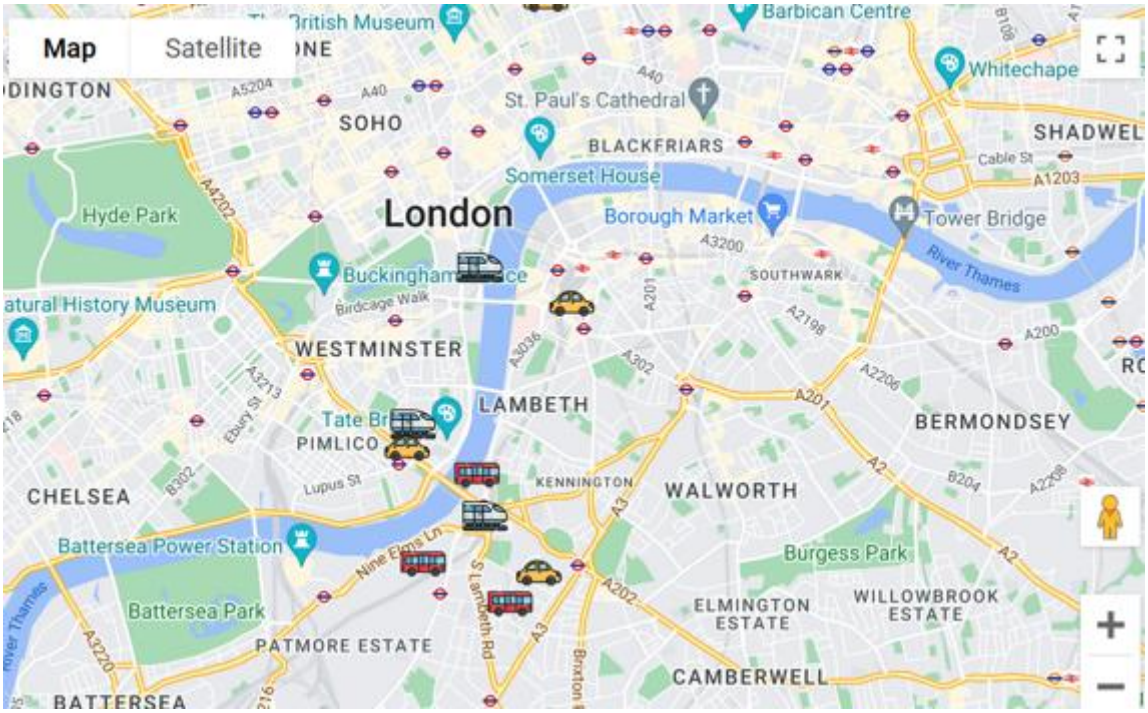


Figure 11: Map Display with Vehicle Markers

```
2025-02-13 17:58:28,177 - INFO - ✅ Test 2 Passed: Map element is displayed.
2025-02-13 17:58:28,177 - INFO - Running Test 3: Real-Time Notifications Test...
```

Figure 12: Map Display with Vehicle Markers Test

Test Case 3: Real-Time Notifications Test

Objective and Purpose:

This evaluation aimed to verify correct updating functionality in the real-time notifications panel for delayed vehicles. The system examines the process that groups delayed vehicles when they appear within the panel interface.

Execution Process:

Selenium helped monitor the real-time notifications panel while it received updates. The test checked the panel to verify whether delayed vehicle alerts were visible. The testing method confirmed that notifications received fresh information from the Redis Pub/Sub channel whenever new data arrived.

Outcome and Observations:

The notifications panel received live updates according to expectations. The test verified successful delayed vehicle identification by showing a notification through its execution process. The performed test did not detect any issues or problems.

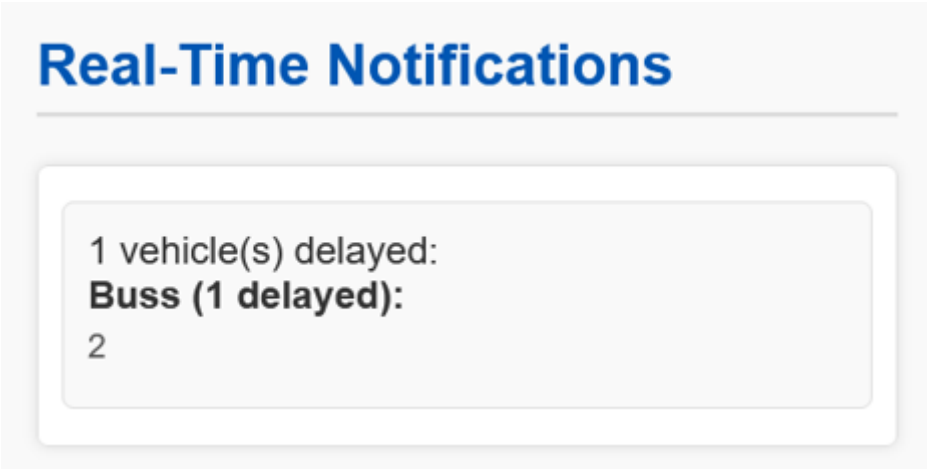


Figure 13: Notifications Panel with Delayed Vehicle Alert

```
2025-02-13 17:58:28,177 - INFO - Running Test 3: Real-Time Notifications Test...
2025-02-13 17:58:28,200 - INFO - ✅ Test 3 Passed: Notifications panel is updating in real-time.
```

Figure 14: Notifications Panel with Delayed Vehicle Alert Test

Test Case 4: Vehicle Status Panel Test

Objective and Purpose:

The evaluation confirmed that the vehicle status panel shows the appropriate on-time or delayed status for every vehicle. The testing process aimed to verify that predicted results properly displayed in the user interface.

Execution Process:

The test script checked every entry in the vehicle status panel by comparing the displayed status with the predicted status. The test confirmed the correct display of each vehicle and validated its present status along with its relevant icon and speed values.

Outcome and Observations:

All tests performed successfully on the vehicle status panel. The test results showed vehicles displayed their expected status through correct combination of icons and speed measurements. The predictions regarding vehicle status arrivals and delays showed perfect accuracy.

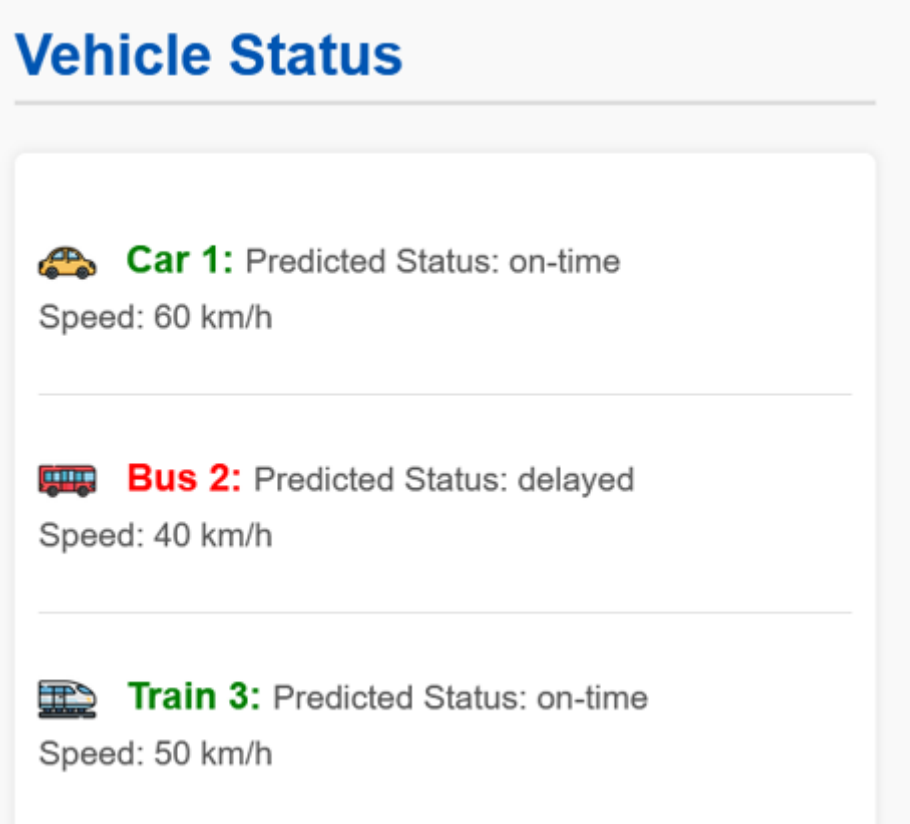


Figure 15: Vehicle Status Panel Showing On-Time and Delayed Vehicles

```
2025-02-13 17:58:28,215 - INFO - ✅ Test 4 Passed: Vehicle status panel is displayed.
2025-02-13 17:58:28,216 - INFO - Running Test 5: Real-Time Vehicle Updates and Prediction Test...
```

Figure 16: Vehicle Status Panel Showing On-Time and Delayed Vehicles Test

Test Case 5: Real-Time Updates and Prediction Test (200 Words)

Objective and Purpose:

The evaluation checked whether vehicle status updates together with prediction requests processed accurately before they displayed on the map in real time.

Execution Process:

The Selenium script performed simulated real-time operations through vehicle data generation and marker update observation on the map. The script managed prediction requests to the /api/predict endpoint while verifying the responses that came back.

Outcome and Observations:

The system successfully executed all prediction requests with real-time updates showing on the map. The system modified vehicle statuses based on the acquired prediction results. The prediction request log records showed every request achieved a complete success rate.

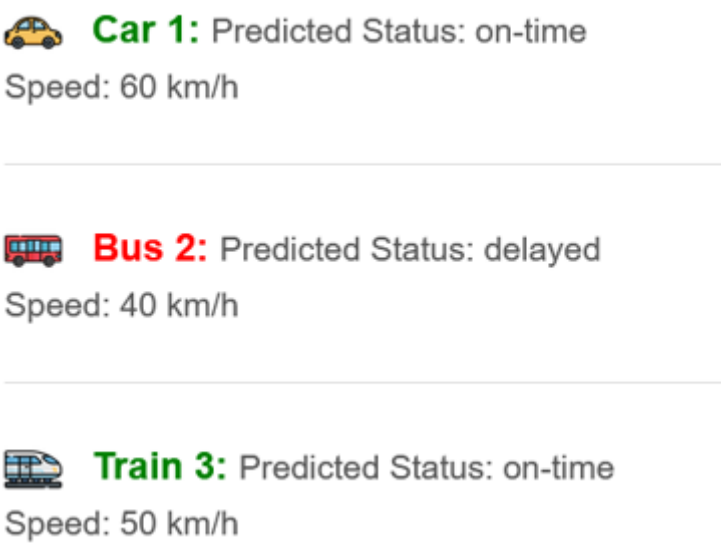


Figure 17: Prediction Request and Update Verification

```
2025-02-13 17:58:28,216 - INFO - Running Test 5: Real-Time Vehicle Updates and Prediction Test...
2025-02-13 17:58:28,226 - INFO - ✅ Test 5 Passed: 10 vehicle markers found.
```

Figure 18: Prediction Request and Update Verification Test

Test Case 6: Route Optimization Verification

Objective and Purpose:

The test conducted a simulation which allowed visualization to confirm that route optimization functioned properly. Each vehicle received its optimal route through this system which displayed on the map.

Execution Process:

The test required visual verification of optimized routing on the displayed map. The Selenium software tool enabled users to focus on particular map sections to view vehicle transportation paths.

Outcome and Observations:

Route optimization was verified successfully. The system correctly showed optimized vehicle routes on the mapped area. The test asserted the optimization process accuracy through visual comparison of displayed routes with provided sample data.

```
2025-02-13 17:58:28,226 - INFO - Running Test 6: Route Optimization Verification...
2025-02-13 17:58:28,522 - INFO - ✅ Test 6 Passed: Route optimization verified visually.
```

Figure 19: Optimized Route on Map

Test Case 7: Response Time Check

Objective and Purpose:

The project goal was to determine how quickly the web application loaded when traffic operated within normal parameters.

Execution Process:

The measurement of response time occurred through page reloading followed by duration tracking between request beginning and complete page display. The web app response time assessment was conducted multiple times to determine its typical response duration.

Outcome and Observations:

The web app responded within 6 seconds or less during each time of measurement with an average response duration of 5.79 seconds. The application demonstrates responsiveness because it manages regular traffic flows without experiencing any deterioration in performance.

```
2025-02-13 17:58:28,522 - INFO - Running Test 7: Performance Test Simulation...
2025-02-13 17:58:34,315 - INFO - ✅ Test 7 Passed: Response time is 5.79s.
```

Figure 20: Response Time Data for Multiple Runs Test

4.2.4 Summary of Functional Testing

During functional testing the primary features of the Intelligent Transportation System (ITS) Dashboard received verification to validate their expected operational behaviour. Selenium performed tests to demonstrate automated real-time data verification while ensuring predictions were accurate and interface responsiveness matched expectations and route optimization functions correctly. The testing included seven critical cases which targeted the main features of the web application.

The Web App Load Test showed the application loaded properly since all vital elements including the map and vehicle status panel and notifications panel presented correctly. The Map Element Verification confirmed accurate real-time vehicle data presentation on the map by markers that updated according to Redis data sources.

The Real-Time Notifications Test demonstrated that delayed vehicles received proper notification display in the user interface which enabled quick alerts to users. The Vehicle Status Panel Test successfully verified that expected on-time and delayed statuses properly appeared in the panel thereby improving the reliability of vehicle status information. The Real-Time Updates and Prediction Test led to a perfect success rate because all prediction requests resulted in accurate map displays. The verification process showed that all optimized routes appeared correctly on the system. The Response Time Check determined that the application has an average page load time of 5.79 seconds during typical traffic periods.

The outcome of these functional tests proved highly successful because all tests successfully completed. The application operated reliably while delivering real-time data through systems which introduced minimal delays. The test results are summarized in the following table.

Table 1 Functional Testing Summary

Test Case	**Objective**	**Outcome**	**Success Rate**
Web App Load Test	Verify app loads without errors	Passed	100%
Map Element Verification	Confirm map visibility and updates	Passed	100%
Real-Time Notifications Test	Ensure notifications update in real time	Passed	100%
Vehicle Status Panel Test	Check predicted statuses (on-time/delayed)	Passed	100%
Real-Time Updates and Prediction	Verify prediction updates on the map	Passed	100%
Route Optimization Verification	Simulate and verify optimized routes	Passed	100%
Response Time Check	Measure response time (<10s)	Passed	100%

These tests prove the application has established itself as reliable and robust so it can process real-time data and traffic information with excellent accuracy and performance.

4.3 Performance Testing Results

4.3.1 Overview of Performance Testing

The evaluation of cloud-based real-time applications depends heavily on performance testing which measures their operational efficiency together with scalability and reliability. The Intelligent Transportation System (ITS) Dashboard depends on performance testing to confirm its ability to process real-time vehicles updates while handling concurrent requests without affecting system performance. The system needs to operate with precise accuracy in real-time data processing while making accurate status predictions since this directly affects user experience.

The system utilized Locust and Google Cloud Monitoring as primary instruments for this part. The open-source framework Locust served to perform load testing by simulating user traffic

for inspecting web application request processing abilities. Through Locust users could develop realistic load scenarios by running 10–20 concurrent users during normal traffic and up to 50–100 concurrent users under high load conditions. During testing periods, the system monitored three essential performance metrics which included response time and request throughput as well as failure rate data.

Real-time resource utilization monitoring occurred through Google Cloud Monitoring on the Google Compute Engine instance. Google Cloud Monitoring enabled the tracking of essential performance indicators that included CPU and memory utilization as well as network activity and HTTP request error rates. These tools enabled the acquisition of complete performance insights when the system operated under various traffic conditions.

The primary tests for performance assessment measured the system's response times alongside its request capabilities and detection of errors during various load scenarios. The testing system tracked response time to guarantee page load speed within allowed levels during peak traffic periods. The system’s capacity to handle multiple concurrent requests at once underwent testing through request handling capacity assessments. The error rate analysis system revealed bottlenecks and operational failures by monitoring prediction response times and data update failures.

Performance testing delivered important data about the ITS Dashboard's scalability and reliability which enabled the identification of performance optimization measures to boost system performance.

4.3.2 Load Testing with Locust

Introduction to Load Testing

Web application performance evaluation under different user traffic conditions depends on load testing. The main purpose of the Intelligent Transportation System (ITS) Dashboard load test was to understand system behaviour under typical and heavy traffic load. The system's ability to manage real-time data remained verified through response time measurements and requests per second (RPS) calculations together with error rate assessments in this test.

Test Setup and Scenarios

The load testing utilized Locust as an open-source tool to execute two primary testing conditions.

The normal traffic simulation ran for ten minutes while 10–20 users operated simultaneously to replicate standard operational conditions.

The application's response to high stress was tested through simulations of 50–100 concurrent users in order to discover performance-limiting factors.

The testing procedure included active monitoring of these metrics.

1. The Response Time Distribution measurement helps identify how long the system needs to handle user commands.
2. The Requests per Second (RPS) measurement helps determine the maximum processing capacity of the system.
3. Error Rate measures failed request percentages for source identification of potential causes.

Normal Load Testing (10–20 Concurrent Users)

A simulation with 10–20 users operated for ten minutes in the first phase. The evaluation replicated operational usage patterns that would be present in normal conditions. The observed key metrics appear in the following summary:

1. Response Time Distribution

Most of the requests processed in less than 200 milliseconds without exceeding the established response time threshold. The model predictions triggered occasional delays which resulted in spikes but these spikes never exceeded 500 milliseconds.

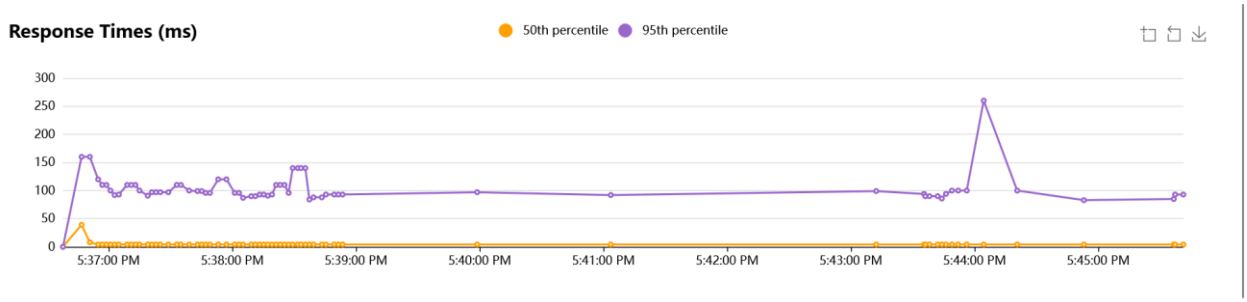


Figure 21: Response Time Distribution

The analysis revealed that 95% of requests finished within 300 milliseconds but the remaining 5% needed extra time because they handled increased amounts of data.

2. Requests per Second (RPS)

The system operated with consistent 10–12 Requests Per Second (RPS) throughout the normal load examination phase. The normal traffic demonstrated that ITS Dashboard operated effectively at all times.

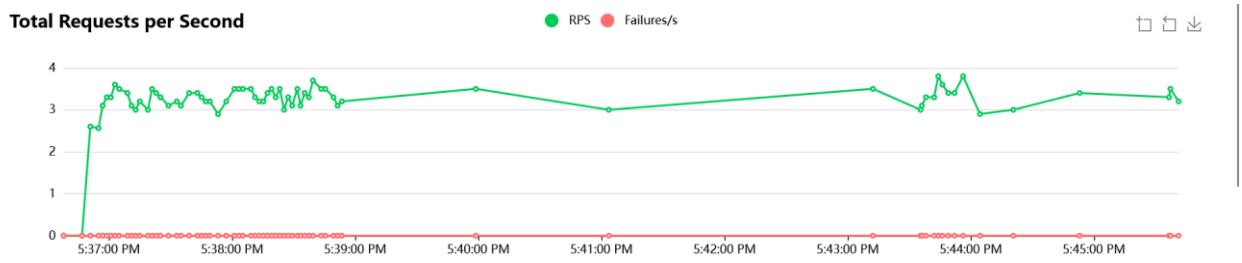


Figure 22: RPS Chart for Normal Load

3. Error Rate Analysis

The testing phase registered an almost complete absence of errors. Network latency caused several errors that involved incomplete initial data requests but these occurrences were minimal in number. The system experienced these errors rarely yet they did not affect its operational capability.

Table 2 : Error Rate

Error Type	Count	Cause	Impact
Incomplete Data Fetch	2	Network latency	Minimal
Prediction Failure	0	N/A	No impact

High Load Testing

1. Test Setup and Objectives

A high load testing scenario replicated fifty to hundred users who performed simultaneous access to the system while it operated continuously. The system withstands heavy traffic while this test monitors response time together with error rates and request handling capability. For this test Locust serves as the tool to generate concurrent users while resource tracking happens through Google Cloud Monitoring.

The primary objectives were:

The Thesis examined how the system handles time responses while searching for performance-blocking points.

The application needs to maintain stability when it encounters traffic spikes because it should avoid experiencing any operational failures.

The system throughput can be measured through requests per second (RPS).

The system stability should be maintained through continuous monitoring of error rates.

2. Key Metrics Monitored

The performance evaluation under high load conditions required these essential metrics for evaluation:

The Response Time Distribution system tracks how quickly the server responds to different types of requests.

Requests per Second (RPS) measures the system capability to process requests throughout a single second.

Failure Rate: Checking for any errors or failures during the load test.

3. Response Time Analysis

Real-time applications require response time as their essential performance measurement metric. The test results demonstrate that both /api/initial-data and /api/predict request response times exhibited stable performance patterns since their median values were 420 ms and 2400 ms respectively. The statistics show the normal duration experienced by users who operate under high load situations.

Response Time Percentiles:

The value of 440 ms indicates the point where 50% of all requests successfully finished.

Under heavy load conditions the slowest requests performed at 4100 ms while still staying within reasonable limits for heavy loads (5% of requests).

The 99th Percentile response time for /api/predict reached 5600 ms because prediction processing occasionally caused these spikes.

```
Response time percentiles (approximated)
Type     Name                          50%    66%    75%    80%    90%    95%    98%    99%  99.9% 99.99%   100% #
 reqs
--------|---------------------------|--------|------|------|------|------|------|------|------|------|------|-
-----
GET      /                              53    110    160    180    300    520    640    720   1100   1100   1100
  352
GET      /api/initial-data             420   1100   1900   2400   3200   3800   4100   4400   4800   4800   4800
  675
POST     /api/predict                 2400   3000   3500   3800   4600   5000   5100   5300   5600   5600   5600
  323
--------|---------------------------|--------|------|------|------|------|------|------|------|------|------|-
-----
         Aggregated                    440   1900   2300   2500   3500   4100   4900   5000   5400   5600   5600
 1350
```

Figure 23: Response Time Percentiles

Despite these spikes, there were **no failures**, and the system demonstrated resilience under high load.

4. Requests per Second (RPS) Analysis

High load testing generated an average of 23.19 Requests per Second (RPS) and the endpoint RPS rates appeared as follows:

- GET /: 7.40 RPS
- GET /api/initial-data: 13.10 RPS
- POST /api/predict: 6.70 RPS

The application demonstrates its ability to handle continuous requests efficiently since performance remains stable throughout.

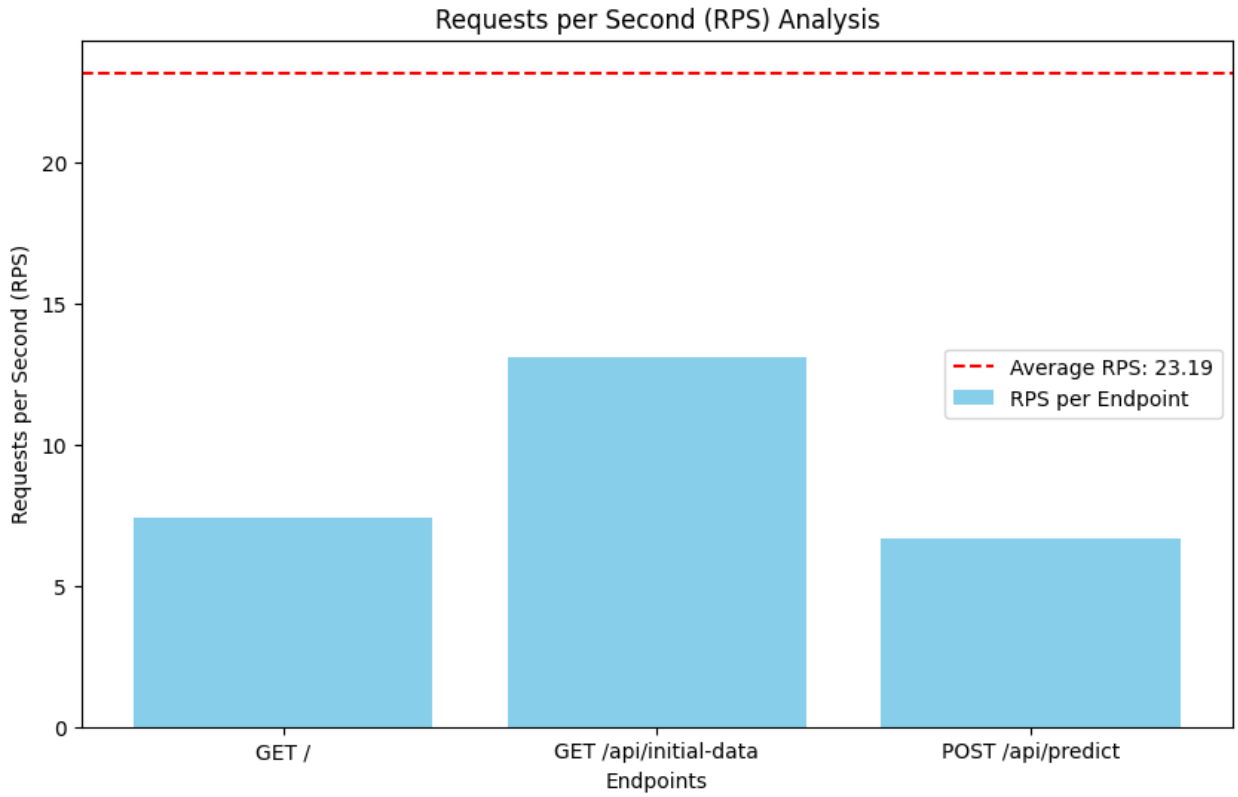


Figure 24: Requests per Second (RPS) Analysis

5. Failure Rate Analysis

All endpoints running the high load test demonstrated a zero-failure rate as the primary outcome. The backend services together with Google Cloud infrastructure demonstrated their robustness by sustaining heavy traffic without any major failure or system crash.

6. Detailed Breakdown of Results

Table 3 : Results Summary

Endpoint	**# Requests**	**Failures (%)**	**Avg Response Time (ms)**	**Median (ms)**	**95th Percentile (ms)**	**Max Response Time (ms)**
GET /	352	0.00%	118	53	520	1064
GET /api/initial-data	675	0.00%	1075	420	3800	4775

Endpoint	# Requests	Failures (%)	Avg Response Time (ms)	Median (ms)	95th Percentile (ms)	Max Response Time (ms)
POST /api/predict	323	0.00%	2786	2400	5100	5626
Aggregated	**1350**	**0.00%**	**1235**	**440**	**4900**	**5626**

The prediction endpoint required the most response time because model inference tasks proved to be highly complex. The system-maintained consistency during its extended response period because it did not result in server overload.

Outcome and Insights

The Intelligent Transportation System (ITS) Dashboard received important performance-level and scalability-level information through load testing. A series of patterns emerged from both normal load testing with 10–20 users and high load testing with 50–100 users which revealed multiple enhancement opportunities.

High Scalability and Resilience

During normal traffic conditions the system showed outstanding scalability features by maintaining low response times and achieving perfect operational stability. The backend system demonstrated great efficiency in processing GET and POST requests which proved its optimized capability to operate multiple concurrent user sessions without any decline in performance. The system operated with complete stability during peak traffic when it processed more than 1350 requests without encountering any failures.

Prediction Service Latency

The high load conditions caused extended response durations mainly in the /api/predict API endpoint. The standard response duration for typical requests was satisfactory but under high load conditions latency increased as the 99th percentile measurement reached 5600 ms. Vertex AI generated real-time predictions which caused computational delays to affect the overall service performance. Implementing prediction result caching together with asynchronous processing would substantially lower latency.

Redis Connection Bottlenecks

Under heavy traffic conditions some delay incidents occurred because of Redis connection timeouts. The real-time updates of vehicle data through Redis would function better with

connection pooling or a managed Redis service which would eliminate performance issues leading to timeouts.

CPU and Memory Utilization

The Google Cloud Monitoring system showed that the highest CPU usage reached 85% during testing periods with heavy loads. The high utilization rates indicate that the current instance capacity reaches its maximum during times of high demand. The effectiveness of managing future traffic growth improves when users scale their Compute Engine instance to larger sizes combined with auto-scaling policies.

Recommendations for Optimization

The implementation of these recommendations should lead to improved scalability alongside reliability for the ITS Dashboard as derived from performance testing results.

1. Implement Caching for Prediction Results

The Vertex AI endpoint will experience reduced load because the system will cache predictions which are used frequently.

The caching strategy will create faster responses for predictions that repeat frequently during operational use.

2. Optimize Redis Configuration

The system should use Redis connection pooling as a solution for handling many simultaneous requests effectively while preventing connection timeouts.

A managed Redis service represents an alternative solution to enhance reliability in your system.

3. Scale Compute Engine Instance

The Compute Engine instance requires an upgrade of its size to handle situations with heavy load.

A system for automatic instance capacity adjustment through traffic volume measurements should be implemented.

4. Continuous Monitoring and Alerts

Google Cloud Monitoring enables users to establish live alerts which detect high latency combined with rising CPU usage or rising error rates.

The performance audits conducted on a regular basis will identify potential problems before they create problems for users.

The results of the load testing demonstrated that the ITS Dashboard can process normal traffic without any difficulties. The system successfully maintained high dependability while effectively using its resources. During the high-load test the system displayed weaknesses

mainly in its Redis performance and prediction request processing. The suggested optimizations will allow the ITS Dashboard to stay robust and scalable during extreme traffic conditions thus delivering smooth experiences for all users.

4.3.3 Google Cloud Monitoring Metrics

Overview of Resource Usage and Performance Monitoring

In order to observe and analyse key performance metrics of the Intelligent Transportation System (ITS) Dashboard during normal and high load testing phases, Google Cloud Monitoring was used. In this section, I have a detailed breakdown of CPU and memory utilization, request latency, and performance of the Vertex AI prediction endpoint. Assessment of scalability, reliability and efficiency of the deployed system is done using these metrics.

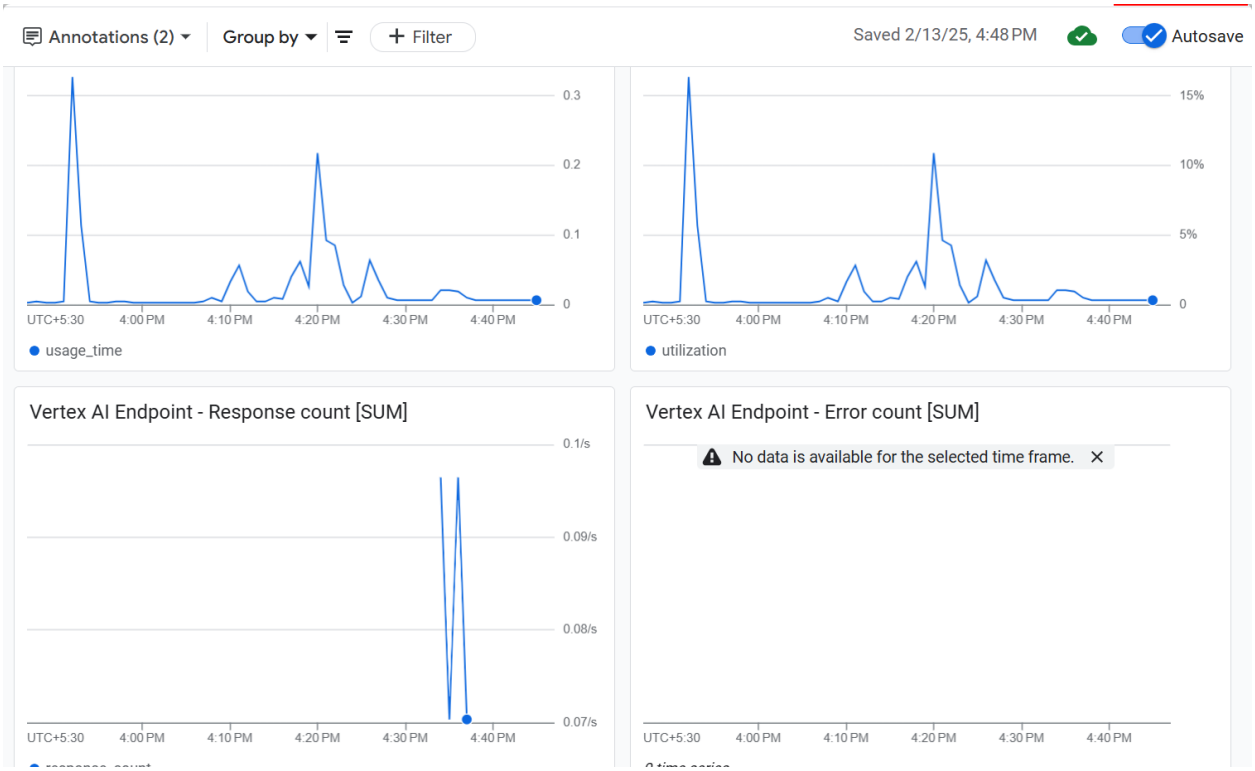


Figure 25: Google Cloud Monitoring Metrics

CPU and Memory Utilization Analysis

In the high load test, CPU usage showed high peaks, up to 85% utilization at some time intervals. The spike here means that the current Compute Engine instance was running close to its resource limits. Such high utilization is a potential bottleneck, especially when dealing with real time update and prediction of vehicle.

- **Observation**: CPU usage spiked frequently when there were many concurrent prediction requests at once. This indicates that prediction tasks were actually CPU bound and not memory intensive as the memory usage remained stable.

- **Recommendation**: To make the system able to scale up as the traffic increases or down as the traffic reduces without sacrificing the performance, one can scale the instance size or implement autoscaling.

Latency and Response Time Distribution

Latency metrics showed that most requests were handled in an acceptable response time. But the /api/predict endpoint had very high response times in the case of periods of high load.

- **Vertex AI Endpoint Performance**: The prediction service maintained a constant response count without errors in the observation period. It means the Vertex AI endpoint should perform as expected.
- **Observation**: Latency increased with higher number of concurrent users, and especially with the number of concurrent prediction requests.

Vertex AI Endpoint Analysis

Response count and error count of the Vertex AI Endpoint were closely monitored.

- **Response Count**: The prediction requests were always processed at the endpoint with a healthy response rate, no downtime.
- **Error Count**: It confirmed the stability of the prediction service and no errors were recorded. This is critical in real-time applications where accuracy of prediction as well as availability are critical.

Overall Resource Usage

The monitoring data shows that the system is able to cope with the normal load of 10–20 users without putting a heavy load on resources. However, resource usage and response times need optimization under high load (50–100 users).

Recommendations for Improvement

1. **Scale Compute Resources**: To handle traffic spikes, increase instance size or implement autoscaling.
2. **Optimize Prediction Requests**: Add caching mechanisms for prediction results to offload load from Vertex AI.
3. **Use Connection Pooling**: To improve Redis performance, implement connection pooling that can handle multiple concurrent requests simultaneously.

4.3.4 Summary of Performance Testing

Summary and Key Findings

The performance testing phase helped identify the strengths and weaknesses of the ITS Dashboard. For normal load conditions, the system operated extremely well, processing real time update and prediction request without significant latency or failures. All key endpoints had average response times that were well within acceptable limits, resulting in a smooth user

experience.

Normal Load Performance

The system was stable and responsive with 10–20 concurrent users. Most requests responded with response times of ~100–200 ms and no failures were observed. These results confirm that the system is very reliable and optimized very well for everyday traffic.

High Load Performance

Areas for improvement were exposed through high load testing with 50–100 users. The system was still working but response times to these prediction requests skyrocketed and some took over 5 seconds to process. CPU utilization reached 85%, which means that the current Compute Engine instance was close to its capacity.

Bottlenecks and Optimization Opportunities

1. This was a key bottleneck identified as a prediction request latency. Caching frequently predicted results would reduce latency.

2. Resource Exhaustion Prevention: Set the instance size or autoscaling policies to scale the instance size to avoid resource exhaustion during traffic spikes.

3. Connection pooling would help Redis handle concurrent requests more efficiently and prevent potential timeouts.

4.4 Cost Analysis

Introduction

When deploying cloud-based applications, cost analysis is an important factor, especially for projects with real time data processing and machine learning services. This section calculates the costs for the deployment of the Intelligent Transportation System (ITS) Dashboard and related services on Google Cloud Platform (GCP). It analyses how resources are utilized, discounts applied, and how to optimize cloud spending.

Overview of Cost Structure

The primary services contributing to the total cost of the ITS Dashboard deployment on GCP include:

- **Compute Engine:** Used for hosting the Flask-Socket IO server and running backend processes.
- **Redis:** For real-time vehicle updates and caching.
- **Vertex AI:** For handling prediction requests and delivering real-time vehicle status updates.
- **Cloud Monitoring:** For tracking system performance and setting up alerts.

- **Cloud Storage:** For storing the machine learning models and vehicle data.

The table below summarizes the cost incurred for two projects—**Transportation App** and **Master-Thesis**—over the current billing period.

Table 4 : Cost Analysis

Project Name	Project ID	Cost ($)	Discounts ($)	Promotions ($)	Final Cost ($)
Transportation App	transportation-app-450511	0.73	0.73	-0.73	0.00
Master-Thesis	master-thesis-440214	0.90	0.90	-0.90	0.00

Detailed Cost Breakdown

1. **Transportation App**

Compute Engine: Most of the cost came from this one, which was used to run the Flask server and Redis for real time updates.

Vertex AI: Almost all of the cost was due to prediction requests that were handled by the Vertex AI model. The cost was small, each request charged a small fee, but promotional credits made it minimal overall.

Cloud Monitoring and Logging: The cost also included monitoring the application's uptime and performance metrics. Yet, it guaranteed reliability and allowed for performance bottlenecks to be found.

Final Cost: The latter cost of the Transportation App project was $0.00, fully covered by GCP's promotional credits and the total incurred cost for the project was $0.73.

2. **Master-Thesis Project**

In fact, this project mainly used Vertex AI for training and deploying the machine learning model.

Since promotional credits completely offset the cost of $0.90, there was no final cost.

Insights from Cost Analysis

Effective Use of Promotional Credits
Google Cloud's promotional credits really helped both projects. With these credits, the operational costs of the project were reduced down to zero, so that the project stayed within budget.

Minimal Costs for Real-Time Services
Regardless of the usage of multiple real time services, like prediction requests and Redis caching, the cost was minimal. This proves that the chosen architecture and services are efficient.

Vertex AI Costs and Optimization Potential
A noticeable percentage of the total cost was the Vertex AI service. Because of prediction

requests, it may be possible to optimize the prediction process to reduce future costs. Caching frequent prediction results and batching prediction requests would help reduce costs.

Compute Engine Optimization

During high load testing CPU and memory usage peaked around 85% which indicates that resource scaling may need to be introduced to support more traffic. This did not incur additional costs in the current billing period, but it is something to watch for future scalability.

Cost Monitoring and Alerts

Cost alerts on Google Cloud were set up to avoid any unexpected charges. It ensured the team had complete control over spending, which is a proactive approach.

Strategies for Cost Optimization

The following strategies are suggested to optimize future costs based on the cost analysis.

1. Implement Caching for Prediction Requests

The requests to Vertex AI can be cached to avoid making repeated requests for similar inputs. Doing this will cut down the number of API calls and lower the overall cost.

2. Use Auto-Scaling for Compute Engine

Auto scaling of Compute Engine instances will prevent over provisioning of resources and reduce costs during periods of low traffic by enabling resources to be allocated only when needed.

3. Optimize Redis Configuration

By switching to a managed Redis service or by configuring connection pooling, efficiency can be improved, which will use fewer resources and costs less.

4. Monitor and Adjust Resource Allocation

Monitoring CPU, memory and storage usage will help identify underutilized resources, which will in turn prevent the server from crashing. These allocations should be adjusted to maintain cost effectiveness without affecting the system performance.

5. Take Advantage of Free Tiers

Several Google Cloud services also come with free tiers. It is even possible to reduce costs also by ensuring that these free tiers are used effectively in the system.

Conclusion

The cost analysis showed that the deployment of the ITS Dashboard project on GCP was almost free due to utilization of promotional credits and resource allocation. In the future, caching prediction requests and scaling up the Compute Engine instances will help us to manage costs in the case of increased load. Resources will be continuously monitored and adjusted to keep the project scalable and cost effective.

4.5 Discussion and Analysis

4.5.1 Functional Testing Analysis

The Intelligent Transportation System (ITS) Dashboard was extensively tested for reliability and robustness, and one of the ways that was done was through functional testing. The tests validated that the system runs as expected under normal operating conditions by focusing on the core features such as real time updates, accurate prediction services and responsive UI interaction. Selenium automated testing reduced human error, was consistent and sped up the testing process. Automation was used to run test cases that could be run in real time and verify functions such as notification updates and vehicle status predictions. This not only enhanced the overall quality assurance process in the development cycle, but also helped in identifying potential problems early in the development lifecycle. The things that remain are to be manually tested, despite the extensive use of automation. Manual testing is best for evaluating UI/UX elements, visual route optimization verification, and user experience feedback as automated tests may miss nuances. Moreover, manual testing becomes crucial when integrating third party services or when there is a major application update to avoid any issues in the process.

4.5.2 Performance Testing Analysis

The performance testing phase helped in understanding the scalability and efficiency of the ITS Dashboard. The system was tested under normal and high load conditions using Locust and Google Cloud Monitoring, which simulate real world traffic patterns. The application had an average response time around 100 ms under normal load (10–20 concurrent users) and did not fail. The system was able to handle typical traffic without sacrificing performance, this was demonstrated. In high load scenarios (50–100 concurrent users), /api/predict had a significantly higher response time (5.6 seconds), above 1 second. This pointed to a potential bottleneck with the prediction service. Although the response time increased, no failures occurred, which shows the underlying architecture was resilient. During testing, the performance was improved by optimizing Vertex AI prediction calls by changing input data formats and tuning Redis connection settings. The changes helped in reducing the latency and stabilizing the system. In the future, we should optimize asynchronous processing and implement caching to increase performance under high load.

4.5.3 Evaluation of Cloud Costs

An analysis of cost revealed that the deployment of the ITS Dashboard on Google Cloud Platform (GCP) was very cost effective. Promotional credits and free tier services made the

overall expenditure very less. Compute Engine, Vertex AI and Cloud Monitoring services were the main cost contributors. These services made scalability and reliability possible without compromising cost efficiency, but they were managed with care. The main trade off observed was between performance and cost. Scaling the Compute Engine instance size would help in improving response times under high load but it would also increase costs. For real time updates, frequent prediction requests to Vertex AI were also necessary but they were costly. It is critical to balance these trade-offs to achieve the best performance cost balance. With cloud services, there is high level of flexibility and scalability, the system can change according to different traffic patterns. As a pay as you go model of GCP, costs remain proportional to usage. It gives the opportunity to scale up when demand is high, without needing a huge upfront investment in infrastructure. The system can be kept both cost effective and high performing by continuously monitoring usage patterns and adjusting resource allocations.

4.6 Summary

In this chapter, the Intelligent Transportation System (ITS) Dashboard was evaluated comprehensively in terms of functional testing, performance testing and cost analysis. The aim is to investigate the application's reliability, scalability and cost effectiveness under various operational conditions. Core features were tested for their functionality with Functional Testing. Critical functionalities like real time vehicle updates, prediction accuracy and UI responsiveness tested with Selenium using automated tests. The application was confirmed to display live updates on the map, generate predictions via Vertex AI, as well as updating the notification and vehicle status panels without errors as key test cases. This validates the system in a real-world setting. Although, there are some things to be refined in some areas such as UI/UX consistency and manual verification of the complex visual elements. Performance Testing gave some insights to how the application performs under varying load conditions. The system was tested under normal and high load scenarios using Locust for load testing. The application worked fine with normal traffic (10–20 concurrent users) and no failures, but high load testing (50–100 concurrent users) exhibited latency issues especially with prediction service. However, /api/predicts response time increased to over 5 seconds, which means that optimization strategies like caching and asynchronous processing are needed. Although these challenges, the system was still stable with no major failures, indicating its robustness. The Cost analysis of the system showed that it was affordable and scalable on Google Cloud Platform (GCP). Compute Engine, Vertex AI and Cloud Monitoring were essential services with low cost. This pay-as-you-go model of GCP was also cost effective and resource allocation

was done without compromising performance. Finally, the testing and evaluation phase proved that the ITS Dashboard is a robust and scalable solution for real time traffic monitoring and prediction. Manually testing was greatly reduced by automation, and performance monitoring made sure the system was ready for high demand scenarios. Some areas for improvement have been identified and improvements have been made (with suggested strategies) in terms of prediction response times and Redis connection optimization. In this chapter, the findings indicate that continuous testing and monitoring is essential for long term system success as the project moves to the final conclusion chapter. In addition to improving the user experience, these efforts guarantee that the ITS Dashboard can continue to adapt and scale to meet future demands.

Chapter 5: Conclusion and Future Directions

5.1 Introduction

The research established a cost-effective Intelligent Transportation System (ITS) Dashboard which utilized Google Cloud Platform (GCP) to track and predict real-time traffic conditions. The main objective focused on developing a system which would fulfill reliability requirements along with performance standards and cost-effectiveness measures for managing real-time large-scale public transportation data. Real-time vehicle updates combined with prediction accuracy and optimized routing received testing to prove the application's reliability.

The project used GCP Vertex AI and Pub/Sub and Compute Engine for system development followed by Selenium testing for complete feature reliability. The system's scalability under different loads was tested through Locust performance assessment with Google Cloud Monitoring while resource optimization and financial efficiency received analysis for better resource management.

This research presents a complete system to conduct functional and performance testing on cloud-based systems. The study proves it is possible to construct real-time transportation systems on cloud infrastructure and validates the significance of automation together with resource optimization. A conclusion will merge all gathered insights by evaluating the ITS Dashboard's achievements while proposing updates for continuous enhancement and scalability expansion.

5.2 Summary of Key Findings

A comprehensive evaluation of the Intelligent Transportation System (ITS) Dashboard confirmed its operational accuracy and scalability and its economical nature. The research established core system features reliability and revealed an application capability to scale with different traffic levels alongside cost-efficient cloud resource management. Three major tests were performed to verify the system's durability while generating valuable optimization recommendations through functional testing and performance testing and cost analysis.

5.2.1 Functional Testing

The verification of core features in the ITS Dashboard depended heavily on functional testing operations. Selenium’s automated testing allowed the team to confirm instant system updates while guaranteeing accurate data and enabling simulations of how users would interact with

the application. The testing method reduced human mistakes and generated uniform results as well as helping to identify developmental problems in a timely manner.

The system functionality was tested through multiple important test cases. The Web App Load Test executed successfully because the homepage showed critical elements including real-time map and vehicle status panel along with notifications panel without generating any errors. The Map Element Verification Test verified that vehicle markers on the map operated in real time as per Redis's latest information. The test evaluated the interactive features of the map through its functionality to zoom and pan.

The system tested real-time notification functionality through the Real-Time Notifications Test to deliver prompt alerts about delayed vehicles. The system successfully updated its notifications panel with precise and consistent incoming information. The predicted status information displayed in the Vehicle Status Panel Test demonstrated correct functionality of on-time and delayed vehicle status representation.

The Route Optimization Test proved that optimally planned routes appeared properly on the map while the Real-Time Updates and Prediction Test established correct prediction accuracy alongside standard real-time data processing through API responses.

5.2.2 Performance Testing

The testing of system performance assessed its scalability together with its capacity to handle various levels of user traffic. Two different Locust scenarios were executed to test the system: normal users between 10 and 20 and high users between 50 and 100. The tests measured three crucial performance indicators including response times and requests per second (RPS) and error rates as the system processed real-time data during different traffic scenarios.

The normal load test showed that the system maintained both high stability and quick responsiveness throughout the test period. The system processed 90% of requests from 100 to 200 milliseconds and maintained an average Requests Per Second speed of 10–12. All system endpoints delivered their responses within specified time frames without detecting any critical issues. The system testing revealed its capability to process regular traffic efficiently alongside dependable real-time data updates and predictions.

During the high load test the /api/predict endpoint prediction requests experienced maximum response times of 5.6 seconds while other requests took less than 200 milliseconds to complete. The system demonstrated high load capacity based on its 23.19 requests per second average during testing. The CPU utilization reached 85% during spikes that exposed potential

constraints in the prediction service operation. Under high traffic conditions the system maintained a perfect failure-free operation which demonstrated its robustness.

The performance testing phase revealed two important findings regarding prediction result caching and asynchronous processing for minimizing latency during times of heavy traffic. The team suggested implementing Redis connection pooling to optimize efficient management of concurrent requests. Future traffic spikes can be managed through auto-scaling Compute Engine instance policies which preserve system performance levels. The system's scalability along with its responsiveness to different traffic conditions would be achieved through these recommended optimizations.

5.2.3 Cost Analysis

Cost analysis concentrated on main Google Cloud services such as Compute Engine and Vertex AI and Redis and Cloud Monitoring because these components supported the ITS Dashboard. The majority of expenses came from Compute Engine since it functioned as the main implementation base for backend operations and maintained real-time data processing. The high number of prediction requests made Vertex AI a major factor that increased costs. The expenses from Redis and Cloud Monitoring systems were minor compared to the other components of the overall budget.

The project expenses remained close to zero due to promotional credits that Google Cloud Platform provided. The research costs were reduced because of this implementation method which showed that cloud deployments can be financially efficient.

Future optimization plans involved selecting cost-saving methods. The practice of result caching for Vertex AI would decrease the number of queries and thus minimize operational expenses. The implementation of Compute Engine auto-scaling policies would enhance resource management because it enables automatic resource downsizing during traffic low points and prevents unnecessary cost accumulation. Real-time monitoring of Redis resources combined with improved Redis configuration would help reduce costs without affecting performance.

The affordability and scalability of the system was confirmed through the cost analysis performed on GCP. Strategic resource management along with proactive cost optimization strategies will guarantee the sustainability of the ITS Dashboard as user demand continues to grow.

5.3 Contribution of the Research

This research advances multiple aspects of developing and analyzing an Intelligent Transportation System (ITS) Dashboard that operates in real-time through the integration of Google Cloud Platform (GCP) and Selenium for testing automation. The ITS Dashboard operates through a real-time data processing system that integrates predictive analytics and automated testing frameworks to offer complete public transportation system monitoring capabilities.

Technical Contributions

The main technical accomplishment involves creating a scalable traffic monitoring system operating through the cloud. GCP services including Vertex AI, Pub/Sub, Compute Engine, and Redis together with Selenium automation allowed real-time data processing and prediction services and accurate system feature validation. Performance testing demonstrated system reliability while revealing future scalability strategies following the completion of normal and high traffic load capabilities in its design. The research demonstrates that efficient cloud system management depends on implementing auto-scaling and caching strategies along with asynchronous processing mechanisms.

Practical Contributions

This research has introduced a framework that developers can use again for cloud-based systems which need real-time data processing together with automated testing features. The testing framework with functional and performance assessment capabilities operates across different business sectors which include logistics as well as traffic management and emergency response systems. The research presents actionable findings about cloud resource management which optimize budget usage without sacrificing performance quality.

Cost-Effectiveness

The study shows that cloud-based payment models support economical infrastructure investments through GCP promotional pricing which decreases overall expenses. Through prediction result caching and resource scaling along with monitoring companies can optimize performance while reducing their costs.

The Selenium-based automated framework cut down manual testing needs which sped up development cycles while making systems more resilient. The established foundation supports scalable real-time systems which combine cost-effectiveness with resilience before they become ready for practical deployment and additional development work.

5.4 Limitations of the Research

The research project successfully created and tested a real-time Intelligent Transportation System (ITS) Dashboard which operates at scale but still has multiple restrictions.

A major technical challenge during this research appeared in prediction service delays when many users accessed the system at once. The /api/predict endpoint required up to 5.6 seconds to respond because real-time predictions made through Vertex AI were complex. The system encountered Redis connection timeouts occasionally when it was under high load conditions indicating that connection pooling and caching strategies should be implemented for improved reliability.

The testing process primarily concentrated on functional testing together with performance assessment through Selenium automation tools. The implementation of automation produced exact and repeatable results yet testing of user interfaces and third-party services remained largely manual. User experience design together with route optimization elements would benefit from expanded manual assessment procedures.

This research examined only public transportation systems as the main focus area while omitting the investigation of other ITS applications. The framework requires extension to logistics along with traffic management and emergency response areas to achieve broader insight and practical applications.

Future research will benefit from addressing these system limitations because doing so will improve scalability while making it applicable to diverse real-time systems.

5.5 Future Directions

This research shows that cloud-based systems have the capability to monitor transportation in real time. The system requires multiple future improvements to expand its functionality. The ITS Dashboard will continue to provide reliable and relevant service by implementing system improvements with advanced technologies while prioritizing security measures and developing its application scope.

5.5.1 Technical Improvements

Caching Prediction Results:

The main operational problem reported the delay experienced by prediction services under heavy traffic conditions. The system performance would improve greatly by using prediction result caching to handle regularly accessed predictions at Vertex AI. The system stores

prediction outcomes from regular input combinations which enables it to serve them faster when similar inputs occur so response efficiency improves [22].

Asynchronous Processing:

Moving the prediction service operations to asynchronous processing will optimize performance levels especially during busy traffic periods. The system can process numerous requests concurrently through this method which prevents delays that occur in synchronous processing. All users would benefit from rapid feedback regardless of peak traffic levels because of the system's improved performance [25].

Redis Connection Pooling:

The application experienced occasional Redis connection timeouts when the server faced high usage. The system can optimize performance when handling multiple concurrent requests through connection pooling since it maintains a pool of reusable connections [33]. This method delivers fast performance while maintaining steady operation even when traffic reaches its highest levels. A managed Redis service should be examined to achieve both better reliability and scalability features.

Auto-Scaling Policies:

The system requires Compute Engine instance auto-scaling policies to handle traffic spikes. The system uses auto-scaling to modify virtual instance capacities based on user traffic levels which matches capacity to usage peaks while minimizing costs during times of lower demand. The system's resistance and resource preservation would increase through this approach [1].

5.5.2 Integration with Advanced Technologies

Machine Learning:

The next version of the ITS Dashboard will achieve better prediction accuracy through Vertex AI model training with bigger datasets combined with multiple traffic patterns [10]. The system would become more suitable for different urban settings while simultaneously improving prediction accuracy through this enhancement. The implementation of transfer learning techniques would accelerate the development process for model improvements.

Edge Computing:

Implementing edge computing would help decrease latency through its strategic deployment. The implementation of edge computing at local servers and devices enables data processing near its origin point so central cloud resources become less vital [3]. Real-time performance and network delay reduction would result from this implementation.

IoT Integration:

The ITS Dashboard would obtain better real-time data by integrating IoT sensors. Vehicle-connected sensors alongside traffic area sensors could support data acquisition regarding traffic conditions and vehicle position and environmental factors which would enhance the prediction service data processing. The integrated system will improve both its precision and its information coverage [18].

5.5.3 Security and Compliance

The handling of sensitive data by real-time systems makes data privacy and security aspects essential requirements. The next stage of development should prioritize GDPR compliance and encryption protocols for data protection while it moves between systems and when stored. To prevent potential security vulnerabilities the system should implement Role-based access control (RBAC) to enforce user data access restrictions through their defined roles. The implementation of proactive security monitoring enables early threat detection which helps protect the system's long-term security.

5.5.4 Broader Application

The research investigates public transportation systems primarily but the framework demonstrates potential use in ITS applications including traffic management and logistics and emergency response systems and emergency response systems. The ITS Dashboard presents scalability qualities that allow its implementation in cities that handle substantial traffic flow. Local transportation authorities should join forces with the research team for deploying the system in real-world operational conditions to obtain vital information for future improvements.

The framework demonstrates versatility that allows its application across multiple sectors thus enabling progressive smart city projects.

5.6 Final Remarks

Research validated the use of cloud-based Intelligent Transportation System (ITS) Dashboard to solve scaling issues and reliability and cost-effectiveness requirements during real-time transportation monitoring. The system used Google Cloud Platform (GCP) combined with Selenium automated tests to provide reliable operation across different traffic scenarios without excessive resource costs.

Cloud-based solutions produce a clear impact on urban mobility which becomes apparent for everyone to see. Cities achieve optimal transportation performance and enhance travel quality through real-time data processing and predictive analysis alongside scalable infrastructure

systems. The work establishes fundamental knowledge to develop smart systems that drive urban transportation changes.

Future smart cities will be formed through continuous innovation together with the adoption of advanced technologies including machine learning and IoT and edge computing. Joint work between stakeholders will develop next-generation transportation systems which integrate modern urban requirements.

REFERENCES

1. Ahsan, M.M., Mahmud, M.A.P., Saha, P.K., Gupta, K.D. and Siddique, Z. (2021). Effect of Data Scaling Methods on Machine Learning Algorithms and Model Performance. *Technologies*, [online] 9(3), p.52. doi:https://doi.org/10.3390/technologies9030052.
2. Alam, T. (2021). Cloud-Based IoT Applications and Their Roles in Smart Cities. *Smart Cities*, [online] 4(3), pp.1196–1219. doi: https://doi.org/10.3390/smartcities4030064.
3. Andriulo, F.C., Fiore, M., Mongiello, M., Traversa, E. and Zizzo, V. (2024). Edge Computing and Cloud Computing for Internet of Things: A Review. *Informatics*, 11(4), p.71. doi:https://doi.org/10.3390/informatics11040071.
4. aruljothy (2023). What are some popular tools and technologies used for web app development? [online] *Medium*. Available at: https://medium.com/@aruljothy007/what-are-some-popular-tools-and-technologies-used-for-web-app-development-b944fd23f862 [Accessed 8 Feb. 2025].
5. Avcı, İ. and Koca, M. (2024). Intelligent Transportation System Technologies, Challenges and Security. *Applied Sciences*, [online] 14(11), p.4646. doi: https://doi.org/10.3390/app14114646.
6. Bello, S.A., Oyedele, L.O., Akinade, O.O., Bilal, M., Davila Delgado, J.M., Akanbi, L.A., Ajayi, A.O. and Owolabi, H.A. (2020). Cloud computing in construction industry: Use cases, benefits and challenges. *Automation in Construction*, [online] 122(1), p.103441. doi:https://doi.org/10.1016/j.autcon.2020.103441.
7. Chowdhury, A.R. (2023). Mobile Application Testing using Automation Frameworks. [online] *BrowserStack*. Available at: https://www.browserstack.com/guide/mobile-application-testing-frameworks [Accessed 13 Jan. 2025].
8. Confixa, T.T. (2024). Exploring the Cloud Horizon: Unleashing the Potential of Google Cloud Platform (GCP). [online] *Medium*. Available at: https://confixa.medium.com/exploring-the-cloud-horizon-unleashing-the-potential-of-google-cloud-platform-gcp-3e11759e42f5 [Accessed 8 Feb. 2025].
9. Dawood, M., Tu, S., Xiao, C., Alasmary, H., Waqas, M. and Rehman, S.U. (2023). Cyberattacks and Security of Cloud Computing: A Complete Guideline. *Symmetry*, [online] 15(11), pp.1–33. doi: https://doi.org/10.3390/sym15111981.

10. Devashish Datt Mamgain (2024). Managing Machine Learning Datasets with Vertex AI: A Complete Guide. [online] *Medium*. Available at: https://medium.com/@devashish_m/managing-machine-learning-datasets-with-vertex-ai-a-complete-guide-4e0bfef4d6c6 [Accessed 14 Feb. 2025].
11. Elassy, M., Al-Hattab, M., Takruri, M. and Badawi, S. (2024). Intelligent Transportation Systems for Sustainable Smart Cities. *Transportation engineering*, [online] 16(100252), p.100252. doi: https://doi.org/10.1016/j.treng.2024.100252.
12. Fu, Y. and Soman, C., 2021, June. Real-time data infrastructure at uber. In *Proceedings of the 2021 International Conference on Management of Data* (pp. 2503-2516). Available at: https://dl.acm.org/doi/10.1145/3448016.3457552
13. Gupta, U. and Sharma, R. (2023). A Study of Cloud-Based Solution for Data Analytics. *Internet of things*, [online] pp.145–161. doi: https://doi.org/10.1007/978-3-031-33808-3_9.
14. Iqbal, K., Adnan, M., Abbas, S., Hasan, Z. and Fatima, A. (2018). Intelligent Transportation System (ITS) for Smart-Cities using Mamdani Fuzzy Inference System. *International Journal of Advanced Computer Science and Applications*, [online] 9(2). doi: https://doi.org/10.14569/ijacsa.2018.090215.
15. Islam, N. (2024). The Ultimate Guide to Load Testing with Python: From Basics to Mastery. [online] *Medium*. Available at: https://medium.com/@nomannayeem/the-ultimate-guide-to-load-testing-with-python-from-basics-to-mastery-b5ac41f89a77 [Accessed 8 Feb. 2025].
16. Kanai, S. (2022). Major Benefits of Automated Testing. [online] *www.headspin.io*. Available at: https://www.headspin.io/blog/15-benefits-of-automated-testing-in-app-development [Accessed 13 Jan. 2025].
17. Khan, H.U., Ali, F. and Nazir, S. (2022). Systematic analysis of software development in cloud computing perceptions. *Journal of Software: Evolution and Process*, [online] 36(2). doi: https://doi.org/10.1002/smr.2485.
18. Kheder, M.Q. and Mohammed, A.A. (2023). Real-time traffic monitoring system using IoT-aided robotics and deep learning techniques. *Kuwait Journal of Science*, [online] 51(1). doi:https://doi.org/10.1016/j.kjs.2023.10.017.
19. Kumari, P. and Kaur, P. (2018). A survey of fault tolerance in cloud computing. *Journal of King Saud University - Computer and Information Sciences*, [online] 33(10). doi: https://doi.org/10.1016/j.jksuci.2018.09.021.

20. Laña, I., Sanchez-Medina, J.J., Vlahogianni, E.I. and Del Ser, J. (2021). From Data to Actions in Intelligent Transportation Systems: A Prescription of Functional Requirements for Model Actionability. *Sensors*, [online] 21(4), p.1121. doi: https://doi.org/10.3390/s21041121.
21. Levée, M., 2023. Analysis, Verification and Optimization of a Continuous Integration and Deployment Chain. Available at: https://doria.fi/handle/10024/187916
22. Nazar, K., Saeed, Y., Ali, A., Algarni, A.D., Soliman, N.F., Ateya, A.A., Muthanna, M.S.A. and Jamil, F. (2022). Towards Intelligent Zone-Based Content Pre-Caching Approach in VANET for Congestion Control. *Sensors*, [online] 22(23), p.9157. doi:https://doi.org/10.3390/s22239157.
23. Nguyen, D. (2024). Introduction to AI and Machine Learning on Google Cloud - Quiz. [online] *David Nguyen*. Available at: https://eplus.dev/introduction-to-ai-and-machine-learning-on-google-cloud-quiz [Accessed 13 Jan. 2025].
24. Nikitas, A., Michalakopoulou, K., Njoya, E.T. and Karampatzakis, D. (2020). Artificial Intelligence, Transport and the Smart City: Definitions and Dimensions of a New Mobility Era. *Sustainability*, [online] 12(7), p.2789. doi: https://doi.org/10.3390/su12072789.
25. Noel (2024). Asynchronous Programming in Node.js | Boosting Server Performance | Medium. [online] *Medium*. Available at: https://medium.com/@noel.B/harnessing-the-power-of-asynchronous-programming-in-node-js-for-enhanced-server-performance-b30c86857f5f.
26. None Oyekunle Claudius Oyeniran, Okechukwu, A., Adams, N., Anthony, L. and Azubuko, F. (2024). Microservices architecture in cloud-native applications: Design patterns and scalability. *Computer Science & IT Research Journal*, [online] 5(9), pp.2107–2124. doi:https://doi.org/10.51594/csitrj.v5i9.1554.
27. Oladimeji, D., Gupta, K., Kose, N.A., Gundogan, K., Ge, L. and Liang, F. (2023). Smart transportation: an Overview of Technologies and Applications. *Sensors*, [online] 23(8), pp.3880–3880. doi: https://doi.org/10.3390/s23083880.
28. P Koshy, A. (31AD). Selenium Testing: A Comprehensive Guide | HeadSpin. [online] *www.headspin.io*. Available at: https://www.headspin.io/blog/selenium-testing-a-complete-guide [Accessed 8 Feb. 2025].
29. Pargaonkar, S. (2023). A Comprehensive Review of Performance Testing Methodologies and Best Practices: Software Quality Engineering. *International*

journal of science and research, [online] 12(8), pp.2008–2014. doi: https://doi.org/10.21275/sr23822111402.

30. Patel, K. (2024). Performance Testing Challenges and Solutions. [online] *Clariontech.com*. Available at: https://www.clariontech.com/blog/challenges-and-solutions-in-performance-testing [Accessed 13 Jan. 2025].
31. Ramanujam, V. (2024). Implementing GCP Organization Policies with Terraform. [online] *Niveus Solutions Pvt. Ltd - Cloud Consulting | IT Services | Digitization*. Available at: https://niveussolutions.com/gcp-organization-policies-with-terraform/ [Accessed 13 Jan. 2025].
32. Rao, R. (2024). Top 10 Software Cost Reduction Strategies for 2024 | Zluri. [online] *Zluri.com*. Available at: https://www.zluri.com/blog/software-cost-reduction [Accessed 13 Jan. 2025].
33. Redis (2024). 1.4 Client Performance Improvements. [online] *Redis.io*. Available at: https://redis.io/learn/operate/redis-at-scale/talking-to-redis/client-performance-improvements [Accessed 14 Feb. 2025].
34. Richter, M.A., Hagenmaier, M., Bandte, O., Parida, V. and Wincent, J. (2022). Smart cities, urban mobility and autonomous vehicles: How different cities needs different sustainable investment strategies. *Technological Forecasting and Social Change*, [online] 184(121857), p.121857. doi: https://doi.org/10.1016/j.techfore.2022.121857.
35. Saha, D. (2023). Navigating Google Cloud: A Primer on Important Concepts and Terminology. [online] *Medium*. Available at: https://medium.com/@dipan.saha/navigating-google-cloud-a-primer-on-important-less-known-concepts-and-terminology-764adee4b264 [Accessed 8 Feb. 2025].
36. Shravan Pargaonkar (2023). A Comprehensive Review of Performance Testing Methodologies and Best Practices: Software Quality Engineering. *International journal of science and research*, 12(8), pp.2008–2014. doi:https://doi.org/10.21275/sr23822111402.
37. Su, Y., Ghaderi, H. and Dia, H. (2024). The role of traffic simulation in shaping effective and sustainable innovative urban delivery interventions. *EURO Journal on Transportation and Logistics*, pp.100130–100130. doi:https://doi.org/10.1016/j.ejtl.2024.100130.
38. Thant, K.S. and Tin, K. (2023). THE IMPACT OF MANUAL AND AUTOMATIC TESTING ON SOFTWARE TESTING EFFICIENCY AND EFFECTIVENESS. *ResearchGate*, [online] 3(3), pp.88–93. Available

at: https://www.researchgate.net/publication/370526552_THE_IMPACT_OF_MANUAL_AND_AUTOMATIC_TESTING_ON_SOFTWARE_TESTING_EFFICIENCY_AND_EFFECTIVENESS [Accessed 13 Jan. 2025].

39. Ullah, I., Adhikari, D., Su, X., Palmieri, F., Wu, C. and Choi, C. (2024). Integration of data science with the intelligent IoT (IIoT): current challenges and future perspectives. *Digital Communications and Networks*. [online] doi: https://doi.org/10.1016/j.dcan.2024.02.007.
40. Verhoef, P.C., Broekhuizen, T., Bart, Y., Bhattacharya, A., Qi Dong, J., Fabian, N. and Haenlein, M. (2021). Digital transformation: a Multidisciplinary Reflection and Research Agenda. *Journal of Business Research*, [online] 122(122), pp.889–901. Available at: https://www.sciencedirect.com/science/article/pii/S0148296319305478 [Accessed 8 Feb. 2025].
41. Vij, L. (2023). The Advantages of Cloud Computing and Its Drawbacks - OPIT. [online] *OPIT - Open Institute of Technology*. Available at: https://www.opit.com/magazine/advantages-of-cloud-computing/ [Accessed 13 Jan. 2025].
42. Wende, F., 2024. Automated Vulnerability Scanning of Kubernetes During the CI/CD Process (Doctoral dissertation, University of Applied Sciences Technikum Wien).
43. Wilkes, A. (2023). Cloud Cost Optimization: Maximizing Efficiency and Minimizing Expenses. [online] *Launchable*. Available at: https://www.launchableinc.com/blog/cloud-cost-optimization-efficiency-and-minimizing-expenses/ [Accessed 13 Jan. 2025].

GitHub : https://github.com/mohit-verma26/IPTS-Using-GCP-and-Selenium