Ατομική Διπλωματική Εργασία

# IMPLEMENTATION AND TECHNICAL EVALUATION OF LOW-CODE APPLICATIONS

**Markos Fikardos**

## UNIVERSITY OF CYPRUS

## DEPARTMENT OF COMPUTER SCIENCE

**May 2025**

# UNIVERSITY OF CYPRUS

## DEPARTMENT OF COMPUTER SCIENCE


# Implementation and Technical Evaluation of Low-Code Applications


**Markos Fikardos**

Prof. Dimitris Zeinalipour



This individual thesis was submitted in partial fulfillment of the requirements for the bachelor's degree in computer science, Department of Computer Science, University of Cyprus

May 2025

# Acknowledgments

I would like to extend my deepest gratitude to my supervisor, Prof. Dimitris Zeinalipour, for entrusting me with this thesis and for his invaluable guidance, continuous support, and constant knowledge-sharing, both during the preparation of this work and throughout my studies at the Department of Computer Science, University of Cyprus.

I would also like to thank my classmates and colleagues at the University of Cyprus for their time and insightful feedback, particularly during testing and discussions that helped refine this work. Their perspectives are greatly appreciated.

Finally, I would like to thank Aris Azhar, and the broader Google AppSheet developer community for their publicly shared knowledge and willingness to answer questions. These resources were instrumental in understanding the development process in Google AppSheet.

# Abstract

In recent years, Low-code and No-code development platforms have gained significant traction as a new approach to software development—particularly in the context of cloud-native solutions. These platforms promise faster development, reduced technical overhead, and increased accessibility for both technical and non-technical users. However, questions remain regarding their technical viability and scalability, especially in real-world use cases with limited resources and minimal infrastructure. This undergraduate thesis will evaluate the validity of these claims, by implementing two real-world applications built entirely within Googles Ecosystem. Those applications are “CheapFuelCY”, a fuel price tracking app that utilizes the publicly available Government API to display the best prices for the current day and catalogues them in order to display trends in pricing, and “UCY Maintenance”, a maintenance request system for university infrastructure that utilizes AI in order to help with understanding description semantics to categorize reported issues and generate summaries. These applications were developed using AppSheet for the No-Code frontend, Google Sheets as the database, and Google App Scripts for auxiliary logic and automations. The evaluation is divided into two primary areas: technical performance and developer experience. Performance is assessed through metrics such as response time and concurrency handling under varying network conditions, using benchmarks from Google Chrome DevTools. The evaluation of developer experience is based upon the views of an undergraduate student in Computer science, whereby the personal experience of developing within these platforms are discussed, taking into account aspects such as its versatility for rapid prototyping, and using examples, evaluating whether or not a non-technological user could feasibly implement applications without too much difficulty. Ultimately, the thesis aims to present an informed perspective on whether No-Code platforms like AppSheet can offer a viable performant alternative to traditional coding environments, and to what extent they fulfil the promise of accessibility and efficiency.

**Table of contents**

# Chapter 1 Introduction



## 1.1 Motivation

In recent years, Low-code and No-code development platforms have emerged as transformative technologies within the software development field. These platforms and frameworks enable users from all backgrounds, ranging from professional developers to business analysts and even non-technical individuals to create applications and automate processes with minimal to no traditional coding.

This rise in popularity can be found from articles and papers all the way back in 2021. One such source is a press release by Gartner that which states that application development would shift to application assembly and integration; and that by 2025, 70% of new applications developed by organizations would use low-code or no-code technologies, up from less than 25% in 2020 [1]. Gartner also estimated in 2021 that the market value for Low-code Application Platforms (LCAP) would rise to upwards of 12,351 million dollars by 2024.

No-code and low-code development has rapidly gained traction across industries due to its ability to accelerate digital transformation, reduce dependency on specialized programming skills, and empower a new class of developers dubbed “Citizen developers”. According to research, the market for No-code and Low-code platforms has experienced exponential growth, with arge software vendors such as IBM, Google, Microsoft, and Oracle now incorporating No-code solutions into their portfolio [2].

The rising popularity of Low-code and No-code also happened in parallel to the increasing reliance on cloud computing has further accelerated the adoption of No-code platforms. Cloud infrastructure provides scalable, on-demand computing resources and streamlined service integration, which serve as foundational building blocks for No-code application development. By abstracting away infrastructure management, cloud platforms enable developers to focus

on logic, interface, and data handling, making the development process faster, cheaper, and more maintainable.

## 1.2 Problem Statement

In today's rapidly evolving digital environment, organizations and individuals are frequently required to develop and deploy software solutions under severe time constraints. The COVID-19 pandemic, for example, revealed how urgent and unpredictable situations demand adaptable, fast, and reliable technological responses. Traditional software development methodologies which often involve long development cycles and high technical overhead are not ideal for providing immediate solutions.

Modern applications are increasingly expected to be mobile-friendly, cross-platform, cloud-native, scalable, and support AI functionalities. They must be capable of being delivered through agile development processes, with minimal infrastructure setup and rapid iteration. These requirements are not only rooted in user demand, but also businesses, education, and government bodies.

One case study example for this is VGATE, a web-based conference platform built during the COVID-19 pandemic using Low-code technologies and Google Sheets as its backend. VGATE was developed in order to provide an online conference platform during the pandemic, so that scheduled events could continue as scheduled. Developed in a matter of weeks, VGATE demonstrated how cloud-based No-code/Low-code platforms could be leveraged to deliver fully functional, production-ready applications at critical times.

The problem this thesis addresses is the growing gap between the demand for fast, scalable, intelligent applications and the limited technical evaluation of No-code platforms' ability to deliver on these demands. While these platforms are widely praised for their accessibility and ease of use, few studies assess their performance, flexibility, and practical boundaries when used to develop and deploy real-world solutions, especially in resource-constrained or time-sensitive contexts.

## 1.3 Contribution

This thesis presents a practical and technical evaluation of No-code/Low-code development platforms through the design, implementation, and benchmarking of two functional applications built entirely within Google's ecosystem, with the use of Google AppSheet, Google App Scripts, and Google Sheets.

More specifically, the thesis evaluates both the technical performance and the development experience of this development methodology via the implementation of two applications. The applications implemented are ‘CheapFuelCY’, an informatic fuel pricing app that pulls data from government sources in order to provide daily fuel pricing to users and display analytics of fuel pricing trends, and ‘UCY Maintenance’, an interactive application where users submit reports regarding issues with the university, and with the use of an LLM, categorizes the issues based on their descriptions so that technicians can volunteer to fix the issues and submit verification of their resolution, and view AI generated weekly summaries of the issues that occurred over the past week.

The technical evaluation is based on the measured response time metrics under various conditions and taking into account various parameters, as well as evaluating if the implemented system to handle concurrency in Google Sheets is sufficient.

The evaluation of development is based on both the time it takes to develop a given application, as well as which areas is no code development flexible, and where it is not.

Finally, within the context of this development methodology, a work-around that handles concurrency issues within Google Sheets is implemented (Specifically in ‘UCY Maintenance’) by combining isolated Web-hooks and isolated log files that are unique to each user of the application

In this way, the thesis offers a grounded technical evaluation of No-code/FaaS development in a cloud-native setting, highlighting both the strengths and practical limitations of the approach, and contributing insights that can guide future use, adoption, and research in this rapidly growing field.

### 1.4 Synoptic Description

The thesis is made up of 7 chapters

Chapter 1 presents the motivation, the problem statement, and the contributions of the thesis, as well as the synoptic description of the approach

Chapter 2 lists concepts and terminologies related to no code, as well as mentions to known existing Low-code platforms and related studies on the topic of No-code and Low-code development

Chapter 3 defines the technological background, which includes the various services provided by Google’s Cloud Ecosystem, as well as quota and performance metrics that are important to take into account when implementing applications using aforementioned services.

Chapter 4 is dedicated to the detailed implementation of application A named ‘CheapFuelCY’. This detailed description includes the method in which each layer (Data, Automation, Front-end) were implemented

Chapter 5 is dedicated to the detailed implementation of application B named ‘UCY Maintenance’. This detailed description includes the method in which each layer (Data, Automation, Front-end) were implemented, as well as a detailed description of the solution to managing concurrency and race conditions within Google Sheets

Chapter 6 is made up of both the definition of the technical evaluation and the personal evaluation, with benchmark results and comments.

Chapter 7 concludes the paper, making final statements regarding the evaluation of Low-code development, and a description for future work.

# Chapter 2 Background & Related work



## 2.1 Background

In this subsection we will elaborate on the definition of No-Code and Low-Code development, as well as some concepts that are closely tied to them. This is to build a better understanding of these terminologies henceforth.

### 2.1.1 No-Code

No-Code development platforms are software tools that allow users to build applications without writing any traditional programming code. These platforms aim to simplify software development by enabling individuals with little to no technical background to create solutions using visual interfaces, prebuilt components, drag-and-drop tools, and declarative logic.
The core idea behind No-code development is abstraction. Complex backend operations such as database calls, authentications, API integrations, and general coding syntax are hidden behind user-friendly interfaces. This abstraction allows designing, building, and testing custom solutions without involving IT.
One of the primary advantages of No-Code platforms is speed. The visual development process significantly accelerates the creation, testing, and deployment of applications, making it ideal for rapid prototyping and iterative development. This quality allows for organizations to

experiment with application ideas far more quickly and reduce reliance on traditional development teams. This helps shift technical resources to more complex or mission-critical projects.

It's important to note that the term “No-code” is a misnomer in the sense that the tools provided by No-code platforms are still fundamentally segments of code. The difference is that users are not required to write the code but rather assemble it in a way that will meet the requirements of the end product.

However, No-code has its limitations. Due to the prebuilt nature of No-Code platforms, developers using these systems are limited in how they can customize their applications. Additionally, being confined to a predetermined set of operations restricts the usage of more dynamic solutions to problems. This may make then unsuitable for more complex, enterprise-grade software solutions.

### 2.1.2 No-Code vs Low-Code

Low-code platforms share a lot of common features with No-code platforms that are aimed at accelerating the process of developing and deploying an application, however they do differ in some key aspects. Both use visual interfaces and prebuilt components, however Low-Code platforms allow users to implement custom scripting, which gives it room to be more flexible with how it tackles certain implementations. Ultimately, they share the same goal, that is to abstract and simplify the development process.

### 2.1.3 Functions-as-a-Service (FaaS)

Functions-as-a-Service (FaaS) is a cloud computing model that allows developers to deploy individual functions or blocks of code to the cloud without managing the underlying server infrastructure. [3] It is an important element of the wider serverless architecture paradigm, which focuses on creating a layer of abstraction between the developer and the server provisioning, scaling, and management. Instead of deploying entire applications or services, FaaS allows developers to create functions without having to personally set up infrastructure to run them. These functions are executed in response to specific events, such as HTTP requests, database triggers, or file uploads. This event-driven execution approach means that code is only run when triggered and allows for automatic scaling based on demand. This makes FaaS a very desirable model for workloads that are infrequent, and variable, as the resources that need to be allocated to process a function can be allocated dynamically. Pricing is handled in a similar manner, as billing is based on actual execution time rather than reserved capacity.

The most prominent FaaS providers include AWS Lambda, Google Cloud Functions, Azure Functions, and IBM Cloud Functions.

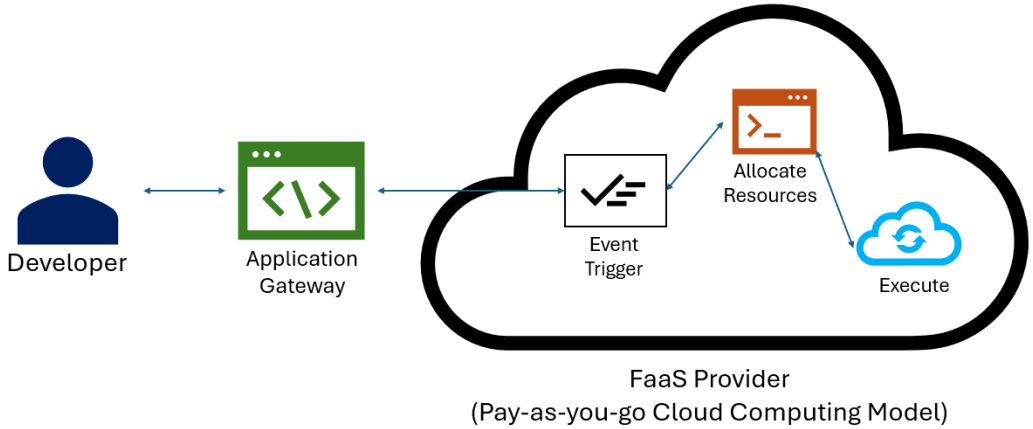



*Figure 2.1 Diagram that displays FaaS Topology*

In the context of No-code and Low-Code development, FaaS can empower these platforms, bypassing limitations by offering a way to create more customized behavior. For example, a No-Code app might trigger a cloud function to handle more complex logic, external API calls, or set up automatic data processing set to a specific schedule. In some instances, a No-Code platform could itself be entirely a FaaS, in the sense that a cloud provider both provides the building blocks to create a No-Code application and provides the infrastructure to host the application.

### 2.1.4 Web Hooks

Webhooks are a web development mechanism that enable event-driven communication between services, by allowing one system to send real-time data to another as soon as a specific event occurs. They are often described as "reverse API's", as instead of polling or actively requesting data from a service, a webhook waits for the data to be pushed to it.
On a technical level, a webhook is a user-defined HTTP callback. It is triggered by an event that occurs in a source system, which sends an HTTP POST request to a predefined URL. This URL is the listener, or receiving endpoint, usually a server or cloud function prepared for handling the incoming data.

## 2.2 Growing Appeal of No-Code Development

Since the start of the decade, the concern that a major shortage in qualified software developers capable of taking positions in companies has been proving to be more and more true. In a study

from 2021, they noted that prior to the COVID-19 pandemic, U.S colleges and universities awarded more than 136,000 bachelors, masters, and PhD degrees in computer and information science, and ever since the pandemic, the American Council on Education reported a 43% decline in pandemic-era enrolments; which is expected to result in approximately 11,700 fewer graduates in these fields [1].

In parallel to this, the U.S. Bureau of Labor Statistics projected a 22% growth in software developer jobs from 2019 to 2029 (around 316,000 software development positions). Notably, this surge does not extend to lower-skilled computer programming roles, which typically involve routine or maintenance tasks, such as writing boilerplate code, simple scripting or working on legacy systems. These roles are expected to see a 9% decline over the same time period. The combination of declining academic output and rising industry demand suggests that a significant portion of software development roles will remain unfilled, creating a growing talent gap in the field of software development. Companies are projected to not be able to meet the demands of manpower in the ever-expanding digital era.

With the wide adoption of large language models (LLMs) such as ChatGPT and Gemini, some have begged the question whether these technologies could fill the gap in the software development workforce by automating programming tasks, potentially eliminating the need for as many human developers. This idea has been widespread, even sparking anxiety among current and prospective Computer and Information Science students, who fear that widespread use of LLMs might render their degrees obsolete.

While this approach may have immediate positive results for businesses, by slashing their employment and only keeping on board a few qualified individuals, this could have negative implications in the long term. This notion overlooks a fundamental truth about how expertise in software engineering is cultivated. Software development is not a static skill but a profession that depends on continuous learning and experiential growth. Most developers begin their careers as junior developers, gradually acquiring the technical and problem-solving skills necessary to become senior developers over time.

If companies increasingly rely on LLMs to fulfil their workforce gap, they risk undermining this natural progression. Junior developers will have fewer opportunities to learn through real-world experiences. This in turn would create a future in which the number of senior developers drops, as the next generation is never given a chance to mature. Rather than solving the issue, it may make it worse.

Finally, while LLMs may be powerful, they are not a true substitution for human developers, as they lack contextual judgment, systems of reasoning, and true understanding of business requirements. Thus, while LLMs can be used to assist developers, relying on them as replacements risks trading short-term efficiency for long-term fragility.

This growing imbalance between the supply of qualified software developers and the increasing demand for digital solutions have motivated those in the field to look for alternative methods of developing software, to hopefully circumvent the issue. As such, the majority of research into No-code and Low-code solutions has been primarily to evaluate its accessibility.

### 2.3 Evaluating No-Code Efficiency

The primary selling point for No-code has been as a productivity booster, reducing the time and technical knowledge required for software development. However, research has evaluated whether these solutions can really live up to expectation. Studies have been made to measure the practical efficiency of No-Code tools in real-world scenarios, focusing on key aspects such as production time saved, usability, and training needed.

One study conducted in 2023 [4] investigated the efficiency of No-Code platforms through controlled experiments and user feedback analysis. The paper investigated these aspects by implementing their own No-Code development platform and assigning tasks to participants in order to evaluate their performance and to receive feedback on the development experience. The group of participants was made up of individuals from 21-29 years old, half of which had a background in Computer science, and the other without.

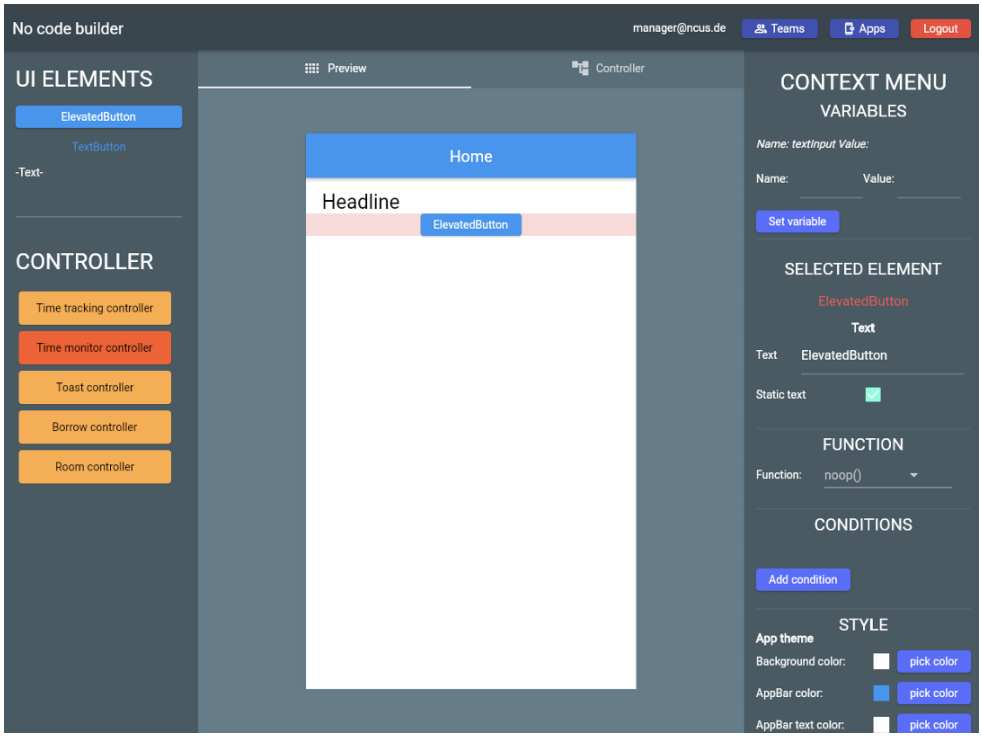



*Figure 2.2: No-Code User interface implemented to run approachability [4]*

The study found that No-Code tools reduced the time required for prototyping and initial deployment of basic applications, and that people without prior programming experience can

develop small applications as good and almost as fast as persons with prior experience, although the margin of variance amongst those without experience was a lot higher. A defining factor regarding what determined the speed at which unexperienced participants could complete the task assigned was how familiar they were with technology. For instance, the weakest performing participant elaborated during the questionnaire that they were almost completely unfamiliar with technology, stating that technology was not present during their daily routine.

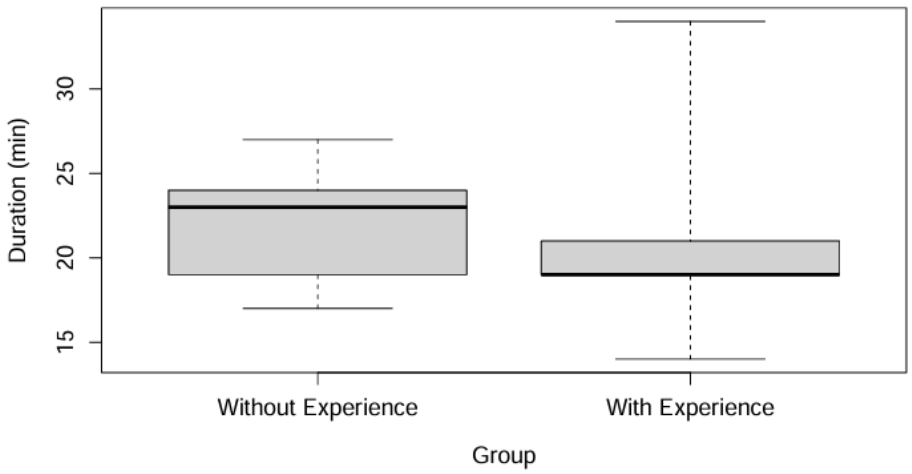



*Figure 2.3 Block plot that shows development time for experienced and non-experienced volunteers [4]*

The research also emphasized critical limitations with their approach. While their No-Code user interface excelled at rapid application development (RAD), they lacked other aspects of development, such as customization, complex business logic, and performance optimization.

| ID | LCDPs | Languages/Frameworks |
|---|---|---|
| P1 | – | – |
| P2 | – | – |
| P3 | Scratch (briefly) | Java, Python, C, C++ |
| P4 | – | JS, Cypress, C, Java |
| P5 | – | Java, Python, C++, Android Studio |
| P6 | – | – |
| P7 | – | – |
| P8 | – | – |
| P9 | Wix, Webflow | Unity, Python psychopy |
| P10 | Scratch | Java, TypeScript, JavaScript, NodeJS, JavaFX, Unity |

| ID | Success | Duration | # Questions |
|---|---|---|---|
| P1 | 92% | 24 min | 2 |
| P2 | 100% | 23 min | 5 |
| P6 | 100% | 17 min | 0 |
| P7 | 100% | 27 min | 2 |
| P8 | 100% | 19 min | 3 |
| P3 | 100% | 34 min | 2 |
| P4 | 100% | 14 min | 0 |
| P5 | 100% | 19 min | 1 |
| P9 | 100% | 21 min | 4 |
| P10 | 100% | 19 min | 2 |

*Figure 2.4: Results of Low-Code development times, with each participants' experience in programming listed [4]*

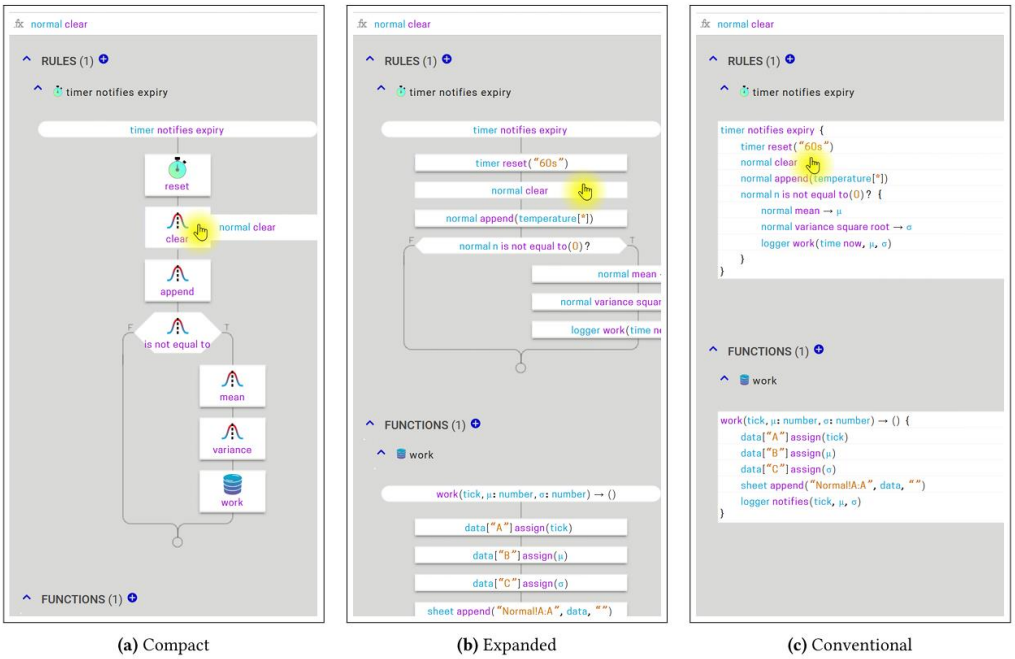
## 2.4 Addressing Modularity

Another study explored the modularity and extensibility of No-Code platforms by introducing their own No-Code development tool named TOO (Things Object-Oriented) as an alternative framework [5] to address the inherent limitations in flexibility and reusability found in No-Code solutions and bring them up to the same level as traditional coding.

The authors argue that existing No-Code platforms suffer from limited abstractions, lacking the flexibility of general-purpose programming languages. To overcome this limitation, they present a minimalist, multi-factored approach that creates a balance between the simplicity of No-Code frameworks, and the expressiveness of traditional solutions.

TOO enables users to construct applications entirely through flowcharts using mouse actions, eliminating the need to write any code. This visual approach allows for the creation of complex logic structures in an intuitive manner. The flowchart based programming also gives users a better understanding of what process or action should lead into what.

However, despite the user-friendly interface, effective use of TOO still required a foundational understanding of programming concepts such as control flow, data structures, and modular design. This requirement echoes the ongoing challenges that are found in other No-Code solutions, as the more freedom you give to developers, the harder it becomes to utilize these platforms as an individual without a technical background in software development.




**(a)** Compact **(b)** Expanded **(c)** Conventional

*Figure 2.5 User Interface of TOO (Things Object Orientated) [5]*

These findings align with the limitations observed in widely used No-Code platforms, including Google AppSheet, which was used as a testbed in this thesis. AppSheet simplifies application development by allowing users to build functional apps without writing code, but

its limitations become apparent when handling complex workflows and large-scale data operations. For instance, while AppSheet provides a user-friendly interface for defining business logic through declarative expressions, it lacks the depth and flexibility required for highly customized solutions. Furthermore, its scalability constraints make it less suitable for enterprise-level applications that demand high performance and robustness.

These challenges are not unique to Google AppSheet but are representative of broader limitations faced by No-Code tools. Research has identified common trade-offs in No-Code development, including:

- **Limited Customization:** Many No-Code platforms offer predefined components, which restrict the ability to tailor applications beyond a certain level.
- **Scalability Issues**: As applications grow, performance bottlenecks and architectural constraints emerge, often requiring migration to traditional development frameworks.
- **Maintainability Concerns:** While No-Code solutions facilitate rapid deployment, they often lead to technical debt due to lack of standardized development practices and version control mechanisms.

With all of that in mind, No-Code platforms continue to evolve, with ongoing research focusing on hybrid approaches that blend No-Code and Low-Code methodologies to strike a balance between usability and power. Future advancements in AI-assisted development, automation, and improved extensibility frameworks may further enhance the effectiveness of No-Code tools, but as of now, their efficiency remains highly context-dependent. The promise of No-Code as a universal software development solution is tempered by the reality that for more sophisticated and scalable applications, some level of coding expertise remains indispensable.

### 2.6 Vendor Lock-In and Platform Dependency in No-Code Solutions

One of the significant concerns with No-Code platforms is the heavy reliance on specific vendors/companies. Research on No-Code adoption has identified that once developers start using a particular platform, transitioning away from it becomes difficult due to proprietary frameworks, ecosystem dependencies, and the lack of standardized export options [6].This issue is exacerbated by the fact that most No-Code platforms do not provide full access to underlying source code, making it challenging to migrate applications to other platforms or traditional development environments without extensive rework.

This aligns with the challenges encountered in this thesis, where the use of a single No-Code ecosystem led to increased reliance on a particular infrastructure, potentially limiting the portability of developed applications. For instance, platform-specific workflows, custom data

storage solutions, and exclusive APIs often create significant barriers when attempting to move applications to alternative environments. Additionally, pricing models in No-Code platforms frequently reinforce vendor lock-in by offering lower-cost entry points but charging significantly higher fees for advanced features, integrations, or exporting functionalities.

Silva & Avelino [6] conducted a comparative study of the most popular No-Code platforms, such as OutSystems (Low-Code) and Bubble (No-Code), testing their efficiency in real-world application development. Their study found that while both platforms enhanced productivity by reducing the need for traditional coding, they introduced significant restrictions on customization, integration, and scalability. In particular, OutSystems provided more flexibility through its low-code approach, allowing some level of custom scripting.

| Benefits | Frequency |
|---|---|
| Reduced development time | 31 |
| Increased productivity | 29 |
| Accessibility to a wider audience Cost | 18 |
| reduction | 12 |
| Flexibility | 6 |
| Security | 5 |
| Personalization | 4 |

| Limitations | Frequency |
|---|---|
| Supplier dependency | 11 |
| High cost | 7 |
| Integration complexity | 7 |
| Resource limitation | 5 |
| Data limitation | 4 |
| Low customization | 4 |
| Low scalability | 3 |

*Figure 2.6 Evaluation of No-code/Low-code platform results [6]*

Beyond these findings, other studies have also highlighted the risks associated with vendor dependency in No-Code development. For example, Zhaohang Yan [7] analyzed the long-term viability of applications built on leading No-Code platforms and found that businesses often face challenges when trying to scale their applications beyond the capabilities of their chosen platform.

These challenges raise concerns about the long-term sustainability of No-Code solutions for enterprise applications, particularly those requiring high levels of customization or integration with external systems. While No-Code tools remain a valuable resource for rapid prototyping and citizen development, organizations must carefully evaluate their long-term technology strategy to avoid costly migration efforts or the risk of being constrained by platform limitations.

# Chapter 3 Technological background



To test the effectiveness of this approach to software development, a decision needed to be made as to which vendor to stick with. The choice of vendor would in turn decide both what functionalities were at our disposal, as well as the rate limitations and architecture the applications would be based on. For the sake of this paper, the choice was made to use services within the Google Workspace ecosystem.

### 3.1 Technology Usage

An application developed through this approach is made up of 3 layers. More specifically, the technologies that were used are: Google Sheets, the free to use online spreadsheet collaboration, which acts as a substitute to a traditional database commonly found in data driven applications; Google App Scripts, which is a modified version of JavaScript provided by Google which can be used both to create standalone plugins for chromium, but also as project dependant automations that run on Google Cloud Server; and Google AppSheet, a tool provided by Google to create frontend User Interfaces for your data without the need for any code.

### 3.1.1 Google Sheets

Google Sheets is a cloud-based spreadsheet application that provides real-time collaboration and data management capabilities. It was originally developed as part of Google Suite, a catalogue of productivity tools, and its primary goal was to allow users to access, edit, and share data simultaneously. Unlike traditional spreadsheet applications like Microsoft Excel, Google Sheets operates entirely on the cloud, eliminating the need for storage of local copies and manual synchronisation of files.

Google Sheets leverages Google's distributed cloud storage infrastructure, which ensures high availability, redundancy, and durability. Data stored in Google Sheets is not confined to a single machine but is instead distributed across Google's global network of data centres. Google organizes its infrastructure using a warehouse-scale computing model, where storage and computing resources are divided into physical enclosures, grouped into clusters, and further integrated into large-scale data centres. This architecture is designed for both performance and fault tolerance. Google uses a multi-layered approach to storage, primarily relying on their Colossus file system, which handles large-scale distributed storage, breaking files into smaller chunks and distributing them across multiple physical machines within a cluster* [8]

Each spreadsheet file in Google Sheets is stored as replicated data chunks across multiple servers. This data replication mechanism ensures durability, fault tolerance, and fast recovery in case of hardware failures.

One key aspect of Google Sheets is its integration with other Google Services, such as with Google App Scripts and Google AppSheet. These integrations give it more flexible usage. With the high durability and availability of Google sheets, and the tight integration with Google Services, we can utilize Google Sheets as a sort of pseudo database that powers the backend data layer of any application we choose to create. We can emulate a traditional database by treating an entire Google Sheet file as its own database, where the individual sheets within the file act as a table.

### 3.1.2 Google App Scripts

Google Apps Script is a cloud-based development platform for creating automations that integrate with Google Workspace. Based on modern JavaScript, Apps Script provides built-in libraries that enable straightforward interaction with Google services, allowing developers to create custom functions, process data, and automate workflows.

Unlike traditional scripting environments that require local execution and dedicated server infrastructure, Google App Scripts operates entirely within Google's cloud. Scripts are written

and executed in a web-based editor, eliminating the need for manual deployment and maintenance. Since the execution environment is fully managed by Google, developers do not need to configure servers, handle authentication manually, or allocate computing resources.
Google App Scripts can be deployed either bound to another Google Workspace file (Doc, Sheet) or as a standalone executable. At surface level, Document bound scripts are easier to work with as they provide easier ways to reference said bound file. Upon further inspection, standalone scripts can be deployed as web-apps and hooks, which proves useful both for sharing functions amongst other developers, and for creating webhooks and APIs.
Google App scripts deep integration with Google Sheets enables the automation of data retrieval, transformation, and storage. This allows for the Google Sheet database to have a functional backend that handles event-driven workflows, Data validation and processing, and API-mediated interactions with external services.

### 3.1.3 Google AppSheet

Google AppSheet is Google's rapid no-code application development platform designed to enable users who are unfamiliar with software development to build and deploy fully functional applications. AppSheet allows businesses and individuals to create applications that leverage structured data from various sources, including local storage, cloud storage solutions (e.g., Google Sheets, Excel, SQL databases), and enterprise systems.
At its core, AppSheet provides a visual development environment to help customers create a user interface for their data so that the users can interact with data in a more intuitive manner. It has built-in premade views for developers to choose from to best display their data. It features a set of pre-built view types (e.g., tables, forms, charts, and dashboards) that help developers structure their app interfaces effectively. Notably, AppSheet integrates machine learning (ML) models to analyse the underlying dataset and suggest optimal view configurations. This ML-driven approach simplifies the development process by recommending appropriate UI components, reducing the need for manual adjustments.
Additionally, AppSheet has built-in conditional formatting, workflows, and automation tools, allowing users to define business rules, and trigger actions. AppSheet developers can define custom business rules that trigger specific actions, such as sending notifications, modifying records, or invoking external services such as scripts or webhooks.

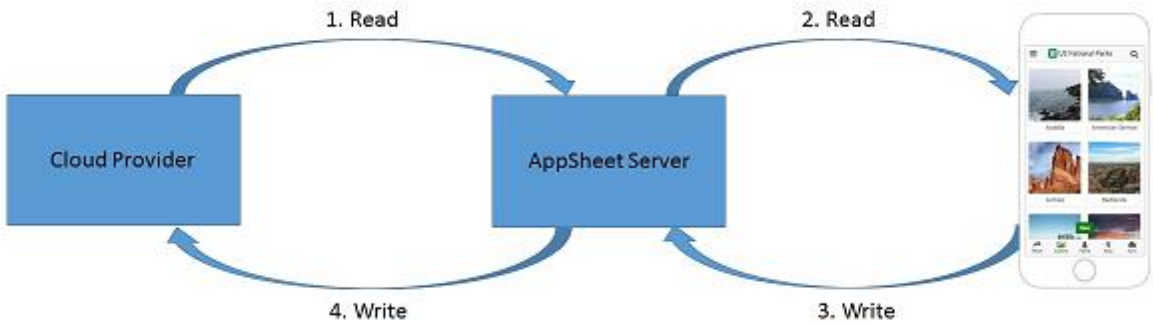



*Figure 3.1 Diagram that displays communication flow between clients, AppSheet, and Storage provider [9]*

Beyond its development interface, AppSheet also functions as a managed backend platform. Applications built with AppSheet do not directly interact with a given data source but rather communicates through AppSheet's own cloud infrastructure. This server layer processes data access, generates virtual columns and data slices, executes business logic and workflows, and handles predictions from integrated machine learning models. All AppSheet applications run in their own environment, and during its runtime, can utilize server-side caching in order to boost performance of data retrievals. This classifies AppSheet itself as a Software-as-a-Service (SaaS) architecture. This server element ensures that applications both remain active, as well as simplifies the deployment process of applications.

### 3.2 Performance Considerations

Understanding the expected performance of all technologies used in our implementation helps with figuring out where the points of bottlenecking will lie, as well as for figuring out workarounds for these performance metrics. Note that since all our technologies interact with each other, we can expect to see the weakest common denominator to act as a bottleneck for our applications' overall performance.

#### 3.2.1 Sheets Performance

Google Sheet performance metrics are split into two parts, those being its raw storage capabilities, and the throughput of the sheet.

A single Google sheet file on its own has a maximum storage capacity of 10 million cells. However, the individual sheets within the file are limited to 18,278 columns (which we can interpret as unique traits each table entry has) and 1,000,000 rows. These limits are in line with other spreadsheet applications such as Microsoft Excel, as well as CSV (Comma Separated Values) file formats. [9]

Since Google AppSheet relies on the Google Sheet API [10], quota metrics from Google Workspace documentation are listed. The documentation states that AppSheet loosely following the same per minute per user quota for accessing the Google Sheets API, with an undocumented quota increase that scales based on the demand within the current moment. [10] The per user quota calculation is based on the individual app creator's account, as all processes run on Google Sheets is based on the current activity and quota limits of the individual who owns the project. While Sheets API has no hard limit for how many API request can be performed in a day, users might experience performance degradation due to different processing components not controlled by Sheets, such as other Google Cloud services.
In the documentation for Google sheets, the developers make recommendations on payload sizes to avoid bottlenecks. Google recommends keeping API request payloads under 2MB for optimal speed. The Sheets API also imposes minute-based rate limits, such as a maximum of 300 read request per minute per project. If the limit is exceeded, additional requests are throttled, returning a 429 status code to inform the callee that they have exceeded the number of requests allowed within that minute. [10]
All Sheets requests follow the rule of atomicity, commonly referred to in databases as "All or nothing". It implies that if any action within a request is not valid, then the entire update is marked as unsuccessful, and none of the changes are applied. Google has also included a time to live (TTL) for all API requests to avoid traffic on their end. When Sheets processes a request for more than 180 seconds, the request returns a timeout error.

The following table details the request limits. Note that these limitations only apply for the current minute of time. As mentioned prior, if you surpass these quotas, they will not lock you out for the current day. They are simply there to reduce the amount of spam requests to the API, and you should be able to resume usage of the API after a minute

| | Read requests | Write requests |
|---|---|---|
| Per minute per project | 300 | 60 |
| Per minute per user per project | 300 | 60 |

*Table 3.1: Google Sheets API Quotas provided by Google Workspace Documentation*

Secondly, but equally as important, whilst Google Sheets may deal with concurrency across multiple cells by using a simple merge algorithm to make sure that all unique data elements stay up to date, we have issues when handling changes to the same cell.

In cases where we are dealing with concurrency issues on the same data element, Google Sheet opts to use a LWW (Last Write wins) approach, meaning that whatever data was pushed last will always be the truth. In the context of Google Sheets in its intended purpose, there would be no issue, as users would be able to notice conflicts in data during input. However, consider the following case scenario:

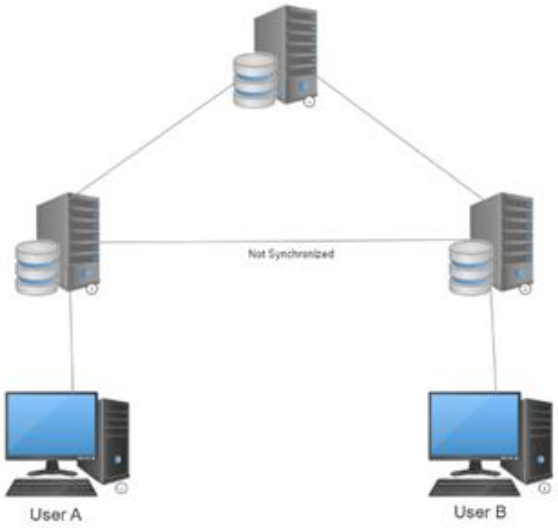



*Figure 3.2: Topology showcasing a plausible case scenario where desync exists between two copies of a file that two clients are trying to modify*

In this scenario, let us assume that two users (A and B) are trying to update data in the Google Sheet. Due to the way that Googles Cloud Architecture works, there is a very likely chance that the two clients are accessing different replicas of the same Google Sheet. In this scenario, we assume that data is being sent at a point in time where the two replicas are not in sync, which is a small time frame.

In this scenario, we will very likely have some destructive concurrency issues. Since Google Sheets operates on a distributed system where replicas of the document may not be immediately synchronized, two users—A and B—could end up modifying the same data at the same time, but on different replicas. When their changes are eventually synced, there is a risk that one user's update may overwrite or conflict with the other's, leading to data inconsistency or loss.

This issue becomes even more apparent when the rate limitations of Google App Scripts are considered. If multiple users are updating the sheet concurrently, especially with frequent triggers like On-Open scripts, it may lead to some actions being dropped due to execution quotas being exceeded. Combined with the possibility of replicas being out of sync, this results in a significant risk of data loss, overwrites, or other inconsistencies.

### 3.2.2 App Scripts Performance

According to the Google App Scripts Quota Documentation [11] , the following key performance constraints apply to Apps Scripts executions:

| | Free User | Google Workspace |
|---|---|---|
| Script Runtime | 6min / execution | 6min / execution |
| Custom Function runtime | 30 sec / execution | 30 sec /execution |
| Simultaneous executions / User | 30 / user | 30 / user |
| Simultaneous executions / Script | 1,000 | 1,000 |

*Table 3.2: Runtime quotas for Google App Scripts. Provided by Google Workspace Documentation. As of 2025, the free and premium quotas are identical. [11]*

In terms of Memory and Payload Constraints, the maximum size of script properties and data transfers is limited to 50 MB per operation. This limitation is fine for loading in text data as there doesn't seem to be a feasible real-world example of needed to exceed this limit. Additionally, image uploads are handled by Google Drive storage, and do not need to pass through Google app scripts, thus not affecting the memory buffer.

URL Fetch and External Calls follow their own quota and are relevant when the usage of APIs is in play. Currently, the Google documents state that a standard workplace account is permitted 20,000 calls per day, with a cap of 50MB payload per call. [11]

These constraints reflect Google's architectural assumption that Apps Script is primarily intended for lightweight automation, data manipulation, and user-driven workflows and not large-scale data processing or heavy database emulation. [12]

To validate the practical impact of these documented limitations, a series of performance benchmarks were conducted.

Another benchmark was conducted to validate the behaviour of Google App Scripts during periods of high traffic.

- With 30 simultaneous script executions, 3 increments failed
- With 60 simultaneous executions, 4 increments were lost.
- With 90 simultaneous executions, 6 increments were lost

| Sample Size | Intended Rate | Final rate | Loss% |
|---|---|---|---|
| 30 | 30 | 27 | 10% |
| 60 | 60 | 56 | 6.67% |
| 90 | 90 | 84 | 6.67% |

*Table 3.3: Benchmark results of concurrent script behaviour*

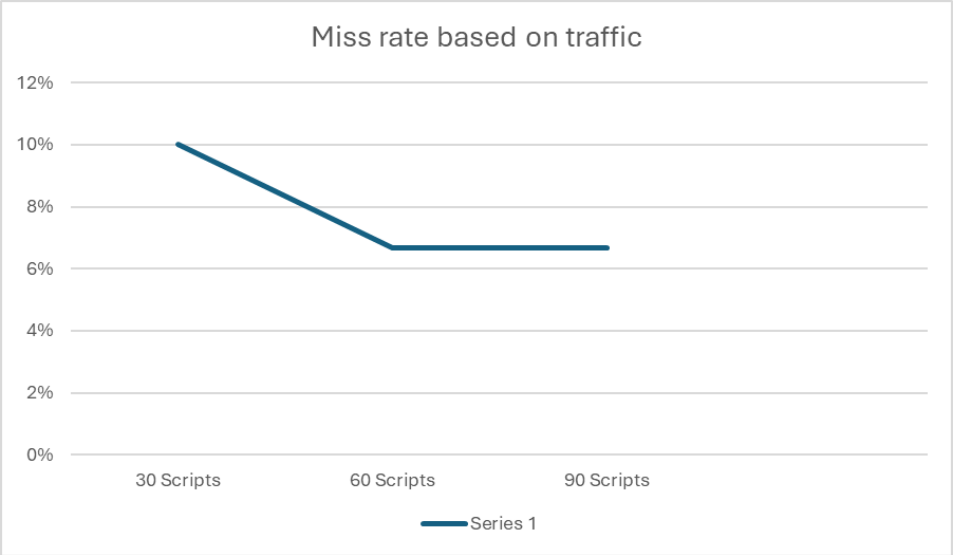



*Figure 3.3: Graphed performance of Google App Script using percentage of misses as evaluation*

These results highlight both the limitations of relying on Google Sheets for scenarios requiring high concurrency or real-time updates, and the unique behaviour of Google App Scripts during periods of high traffic. The decrease in miss-rate can be attributed to the fact that scripts that scripts either refuse to start due to the quota limitation of App Scripts, or due to the poor atomicity of Google Sheet cells.

### 3.3 Pricing

The cost associated with the use of no-code development platform varies based on a number of factors, such as the size of your business, what prices you can negotiate, and whether or not you need access to most services provided by the vendor rather than just a few. In the context of this paper, since this is an independent study, there is no bargaining power to negotiate for any pricing reductions, thus the pricing calculations done henceforth are based on Google's advertised plans and deployment costs.

#### 3.3.1 Google Workspace

When considering Google Workspace, the primary pricing decision revolves around whether a developer requires the additional capabilities offered by paid Workspace plans. Google offers a range of services for free (e.g., App Scripts, Sheets, Cloud storage) however there are a few services exclusively available to individuals who own a Google Workspace account, such as access to Google's Gemini AI, and Google BigQuery. In the context of deploying applications:

- Increased quota for URL Fetch calls: This includes any API
- Increased storage capacity for Google Drive

- Free subscription to Google AppSheet core
- Increase daily runtime limit for App Scripts (Up to 6 hours a day)

### 3.3.2 AppSheet Deployment

While Google Workspace costs are relatively modest, AppSheet deployment costs represent a far more significant consideration in the economics of low-code application development. The subscription model is on a per user basis. Google AppSheet charges users a fixed rate per user that is allowed to use the application, the amount of which is based on the subscription tier. These tiers dictate what functionalities can be used, such as Automations, AppSheet Server caching, Delta sync, On-device encryption, and Security filters. The basic subscription does not offer any of the features, which essentially makes it a much weaker product, almost making the core subscription a necessity if a developers wish to deploy an application with good performance metrics and security.

There is also an enterprise plan that enhances the application even further, which provides machine learning extensions, advanced data types (connections to OpenAPI, BigQuery connector), advanced authentication, and advanced user and data management. These functionalities are not available for testing within the free prototyping version of AppSheet, but they are worth noting. However, they fall outside the scope of this study.

| X | Starter | Core | Enterprise Plus |
|---|---|---|---|
| Price | 5$/user/month | 10$/user/month | Varies |
| Basic App and Automation features | YES | YES | YES |
| Standard Authentication Providers | YES | YES | YES |
| Advanced App and Automation Features | NO | YES | YES |
| Application Security Controls | NO | YES | YES |
| Machine Learning | NO | NO | YES |
| Advanced Data | NO | NO | YES |
| Advanced Authentication | NO | NO | YES |
| Advanced User & Data Management | NO | NO | YES |

*Table 3.4 Showcases tiers of AppSheet, listing their price and the features provided*

If a developer wishes to instead create a public application that is free to the public without requiring authentication or login, the pricing model changes, and they are only required to pay for a monthly Publisher Pro Subscription, which includes same services and functionalities found in the Core subscription, as well as a 50-euro charge for every application the developer

deploys. This brings the monthly pricing for the deployment of a single application down to 58 euros a month, which is incredibly economical all things considered.

### 3.4 Security

The security model of No-code development must be evaluated with the broader context of cloud computing systems in mind. In the context of this paper, since we exclusively use Google provided services, that means the security model that the applications created follow the shared responsibility model proposed by Google [13], with characteristics closely aligned to the Function-as-a-Service model. This implies that any concern for security all the way from the hardware level to the network security, infrastructure, and deployment are handled by Google's Cloud Architecture.

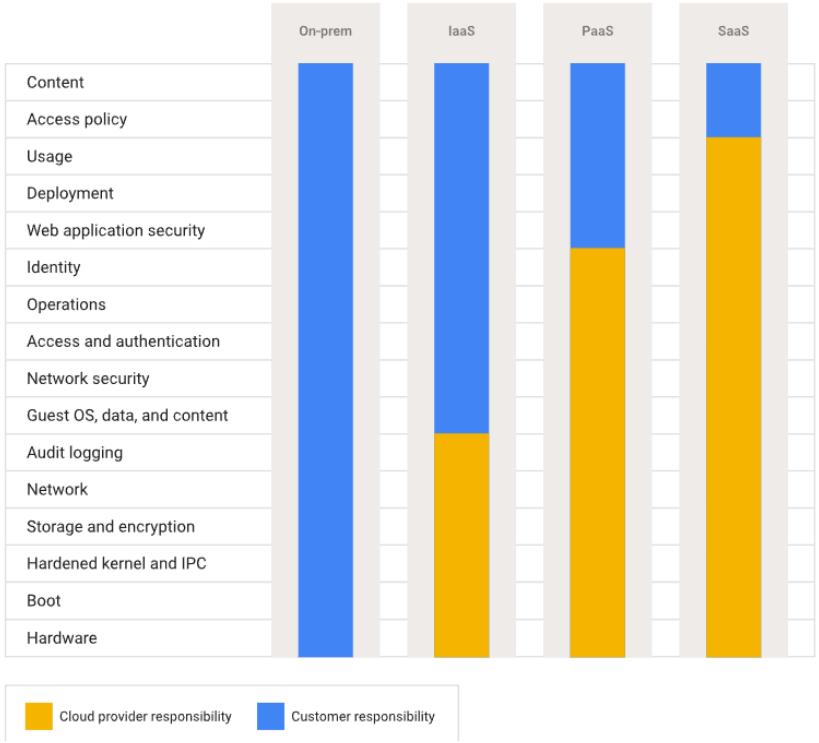



*Figure 3.4: diagram shows the cloud services and defines how responsibilities are shared between the cloud provider and customer. [13]*

In traditional models such as Infrastructure-as-a-Service (IaaS), developers must manage operating systems, networks, and application security. However, due to how Google's services abstract these layers entirely, this means that developers using these tools only need to concern themselves with Application Logic and Data Governance.

This shared responsibility is named by Google as the "Shared fate, Shared responsibility" model, implies that while Google provides secure-by-default infrastructure and services (e.g.,

encrypted data, DDoS protection, authentications systems), developers are responsible for correct configuration of access controls, secure handling of data, and business logic definition through the tools provided (e.g., Google Sheets, Google App Scripts).

As for the level of security provided by Google; Google, being one of the largest and most established technology companies, operates under strict global compliance frameworks and holds numerous security certifications, including ISO/IEC 27001, SOC 1/2/3, FedRAMP, HIPAA, and GDPR. These attest to its adherence to industry standards for data protection, risk management, and operational security, providing a strong foundation for applications built entirely within its ecosystem

# Chapter 4 Implementation A: CheapFuelCY



## 4.1 Data Management and Storage Implementation

The backend of the application is the core data processing layer, responsible for retrieving, structuring, and maintaining fuel price data. In contrast to the frontend, the backend requires a higher level of technical complexity, including data extraction from API responses, following well established database structuring rules, and coding automations in Google App Scripts. As such, familiarity with scripting languages and basic database table structuring is required, however not to the same extent as traditionally.

### 4.1.1 Google Sheets Database

To manage and store fuel pricing data, a structured Google Sheet database was implemented. This database serves as the backbone of the CheapFuelCY application, facilitating data retrieval, analysis, and app functionality. The database consists of multiple sheets, each serving a distinct purpose in data management.

The core of the database comprises five sheets corresponding to different fuel types, as provided by the government API. These sheets are refreshed daily and do not accumulate historical data. Each sheet follows a standardized format to ensure consistency and usability:

| City | District | Address | Fuel Price | Date Modified | Coordinates |
|---|---|---|---|---|---|

*Table 4.1: Elements of each entry in fuel data*

The majority of this data is directly displayed to users, as it represents essential fuel pricing information. However, specific fields, such as Date Modified and Coordinates, serve additional backend functionalities. The Date Modified field allows the application to flag outdated or unreliable prices. Meanwhile, the Coordinates field is utilized for location-based services, such as generating Google Maps links for navigation and enabling the map view within the app.

To accommodate large API responses, an additional set of sheets, labelled raw_data_1-5, is utilized to store the unprocessed JSON data retrieved from the government API. This implementation was necessary due to Google Apps Script's local variable limitations, which prevent the storage of long strings within script execution. Alternatively, a system could be developed to store this raw data in Google Drive instead of a sheet, offering another approach to handling large API responses. The raw data stored in these sheets undergoes processing via Google Apps Script before being utilized by the main application.

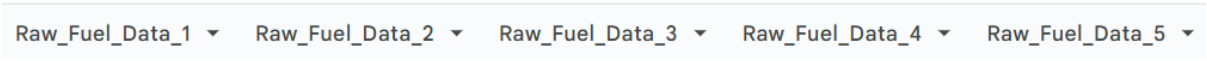



*Figure 4.1: Representation of each sheet used to store raw Json data*

To track fuel price trends over time, additional sheets labelled U95, U98, D, HD, K History store historical pricing data. These sheets maintain records of average fuel prices per city, along with standard deviation calculations. This enables trend analysis and monitoring of price fluctuations. Data collection for these sheets commenced at the initial deployment of the application.

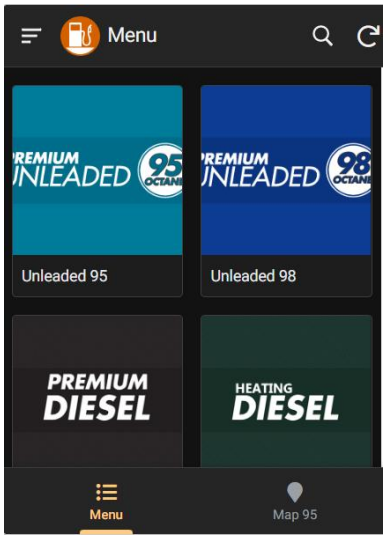
To optimize storage and facilitate long-term analytics, a future implementation may involve limiting records within these sheets or migrating older records (e.g., past one year) to separate archival sheets for yearly analytics segmentation.

| A | B | C | D | E | F | G | H | I | J | K |
|---|---|---|---|---|---|---|---|---|---|---|
| Date | PafosAVG | PafosDIV | LimasolAVG | LimasolDIV | LarnakaAVG | LarnakaDIV | LeukosiaAVG | LeukosiaDIV | AmmoxostoAVG | AmmoxostoDIV |
| 11/03/2025 | 1.445 | 0.011 | 1.43 | 0.021 | 1.431 | 0.016 | 1.419 | 0.03 | 1.434 | 0.009 |
| 14/03/2025 | 1.441 | 0.011 | 1.426 | 0.022 | 1.426 | 0.017 | 1.415 | 0.031 | 1.432 | 0.011 |
| 15/03/2025 | 1.432 | 0.018 | 1.416 | 0.026 | 1.41 | 0.026 | 1.404 | 0.035 | 1.414 | 0.019 |
| 16/03/2025 | 1.424 | 0.02 | 1.412 | 0.026 | 1.403 | 0.025 | 1.4 | 0.032 | 1.41 | 0.018 |
| 17/03/2025 | 1.424 | 0.02 | 1.412 | 0.026 | 1.403 | 0.025 | 1.4 | 0.032 | 1.41 | 0.018 |
| 18/03/2025 | 1.423 | 0.02 | 1.41 | 0.026 | 1.401 | 0.025 | 1.396 | 0.03 | 1.404 | 0.017 |

*Table 4-2 Representation of data entries for fuel analytics/history*

Two additional sheets serve functional purposes within the application:

**Dashboard**: The Dashboard sheet is utilized for enhancing the app's user interface. This sheet stores images that are assigned to views within the application, enabling a more visually intuitive menu. By integrating this sheet with AppSheet's UI components, stored images are displayed as selectable icons, improving user experience. These images are stored in the Drive associated with the AppSheet Project, and are accessed by calling the directory they are located




| UID | Name | Image |
|---|---|---|
| 04cf60a5 | Unleaded 95 | Menu_Images/04cf60a5.Image.214207.png |
| 37c265e1 | Unleaded 98 | Menu_Images/37c265e1.Image.214233.png |
| 4416bf80 | Diesel | Menu_Images/4416bf80.Image.214342.png |
| 960f23ce | Heating Diesel | Menu_Images/960f23ce.Image.204015.png |
| 6250bc5c | Test | Menu_Images/6250bc5c.Image.222646.jpg |

*Figure 4.1: Example of image storage in Sheets, along with visual representation*

**Daily Cheapest Prices Sheet**:

The "Daily_cheapest_prices" sheet facilitates an automation feature within AppSheet. This sheet stores records of the lowest fuel prices for each city, fuel type, and date recorded. When significant price drops are detected between consecutive days, AppSheet's automation feature is triggered to send notifications to users. This system ensures that users are promptly informed of notable changes in fuel prices, enhancing the application's value and usability.

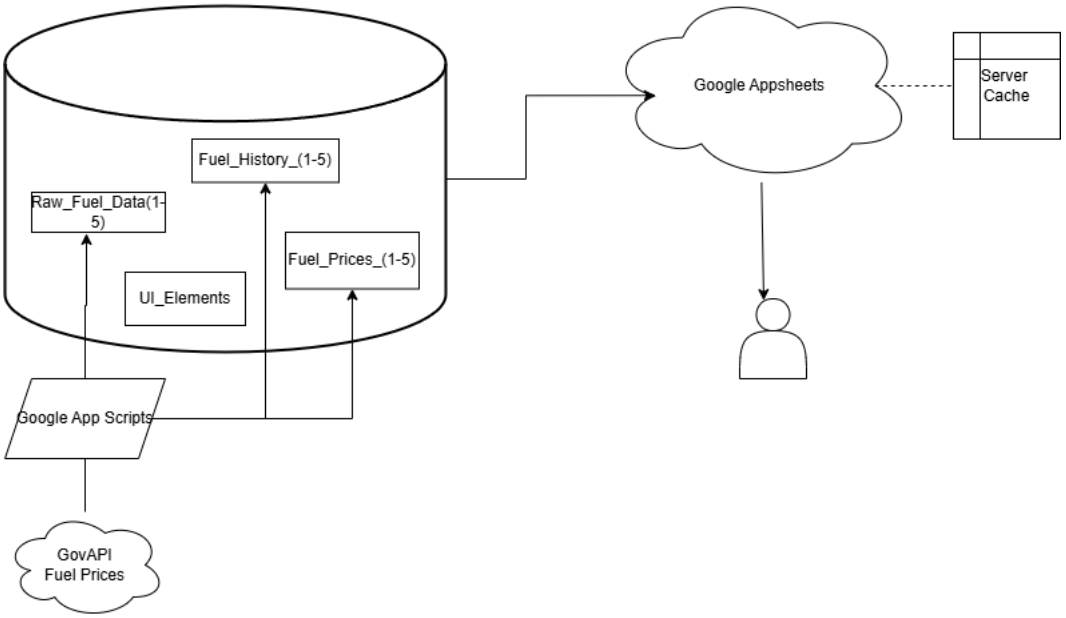



*Figure 4.2: Topology of data transfer path*

### 4.1.2 Google App Scripts & API

The Google App Script running in the background of the CheapFuelCY system is responsible for data retrieval from the government API (GovAPI), data cleaning, and automated calculations that support various in-app features. These scripts ensure that the data stored in the Google Sheet database is structured properly and enriched with necessary computations for enhanced functionality.

**Data Retrieval and Processing:**

The GovAPI requires an initial request to prepare the dataset before it can be retrieved. When a request for a specific fuel type is made, a corresponding request code is generated. This code must be stored and used later to fetch the requested data. To minimize dependence on Google Sheet storage, the request codes are saved in an environment variable within the Google App Script project itself, keeping them independent from the spreadsheet.

Once the prepared data is available, the script retrieves it from the GovAPI and stores it in the designated raw data sheets. Immediately following this retrieval, the data is processed and extracted into the respective structured data sheets. This ensures that the core fuel pricing data is properly formatted and readily accessible for application use.

Additionally, some changes had to be made to the data from the Government API as not all entries follow a uniform format. An example of this would be the values provided for coordinates. In a notable number of entries, the units of measurement differed to what was usable by Google Maps and thus were not able to be simply appended to a Google Maps link.

**Automated Calculation and Additional Scripts:**

Beyond data retrieval, additional scripts handle various automated computations essential for app functionality. One such script calculates daily averages and standard deviations of fuel prices for each city, supporting analytic features within the application. Another script tracks the lowest recorded fuel prices between consecutive days to identify significant price drops. These calculations provide valuable insights and facilitate automation processes within the CheapFuelCY system.

By integrating these scripts, the application maintains an efficient and automated workflow, ensuring current-day data availability, structured storage, and analytical support for enhanced user experience.

**Scheduled triggers for automations:**

Google App Scripts includes an automation feature that enables functions to be executed without manual intervention. These automations can be assigned triggers, allowing them to run based on specific actions, dates, or time-based schedules. Given that fuel data updates daily, scheduled triggers were implemented to execute necessary processes in the morning hours.

Due to the constraints of Google App Scripts' scheduling options, specific time slots had to be strategically selected to prevent overlapping executions. A critical consideration was the data retrieval process, which required an initial request to the GovAPI before data could be fetched. Since the preparation of this data could take longer than the maximum runtime allowed by Google App Scripts, it was not feasible to complete the entire process within a single automation. Instead, separate scheduled triggers were configured: one to initiate the data preparation request and another to retrieve and process the data once it became available.

Once the primary dataset was successfully retrieved and stored, subsequent scripts could be executed concurrently without conflict. These scripts primarily read from the same dataset but wrote results to different sections of the Google Sheet database, ensuring integration of automated calculations and data processing.

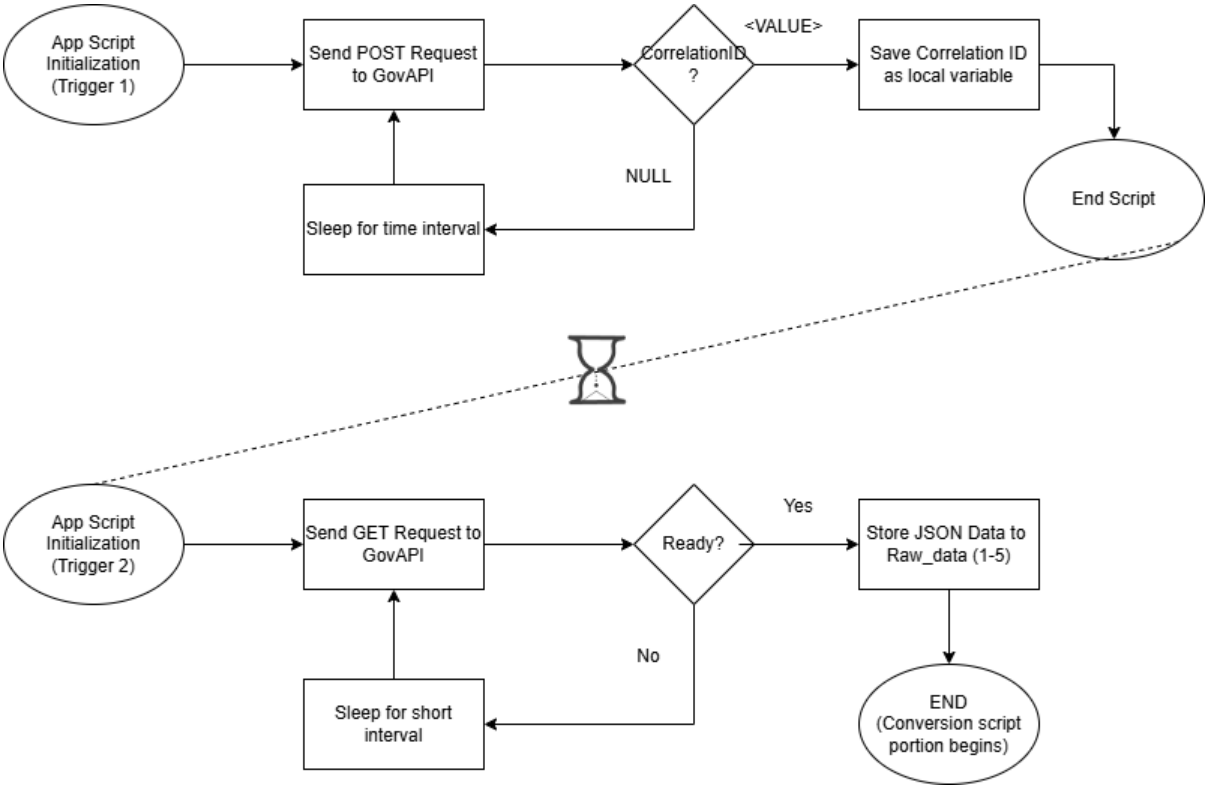



*Figure 4.3: Flowchart logic of acquiring data from Government API*

By implementing a structured scheduling system, the CheapFuelCY application maintains efficient automation, preventing execution failures due to runtime limitations while ensuring timely data updates and analysis.

## 4.2 AppSheet Development

AppSheet development involves both the implementation of the User interface as well as the creation of data slices, security filters, and automations that enhance a given application's functionality. The user interface development involves allocating a table or data slice to a given view and then selecting a premade view type to be display that data.

### 4.2.1 Configuring App and Data Import

Since CheapFuelCY is designed solely as an informational application that disseminates public data without requiring user input, it qualifies for deployment as a public AppSheet application. This decision has distinct advantages and limitations, which are as follows:

**Positives:**

- By disabling login requirements, the application can be accessed immediately by any user without a Google account. While this inherently reduces security, the nature of the app mitigates this risk: the data presented is publicly sourced from official government APIs, and no sensitive or user-specific data is collected, stored, or transmitted
- Public AppSheet applications are billed at a fixed rate of 50 per month per app, regardless of the number of users. This model is particularly cost-effective for apps with large or unpredictable user bases. For applications with a user vase exceeding roughly 10 active users, the pricing becomes more cost efficient.
- All AppSheet applications are also available on browser. As such, they can be shared with a simple URL, making them suitable for mass distribution through websites, social media, or QR codes, and accessible even by anonymous users.

**Negatives:**

- Limited usage of Security Filters as there is no user identification. Security Filters are typically used to restrict data visibility on a per-user basis, which can also improve performance by limiting the volume of data synced to each client. Without user log in, the only way to utilize Security filters is with User settings, which AppSheet only supports up to 10 per application.
- The absence of user accounts prevents detailed tracking of user activity or usage analytics. While basic app usage can still be measured at an aggregate level, individual user behaviour cannot be logged or analysed.

Additionally, due to the nature of CheapFuelCY, that being an informatic application with a low update rate (once per day), it makes great use of AppSheet's server caching feature. While the primary data source continues to be the Google Sheet containing the fuel data, AppSheet stores cached copies of the data on its servers to reduce direct access to the Google Sheet, improving performance and minimizing traffic.

AppSheet natively integrates with Google Sheets as part of the Google ecosystem, ensuring seamless data import. The required tables were loaded directly into AppSheet without complications. However, to optimize performance and reduce unnecessary data processing, certain tables, such as the raw data sheets, were deliberately excluded from AppSheet's data structure.

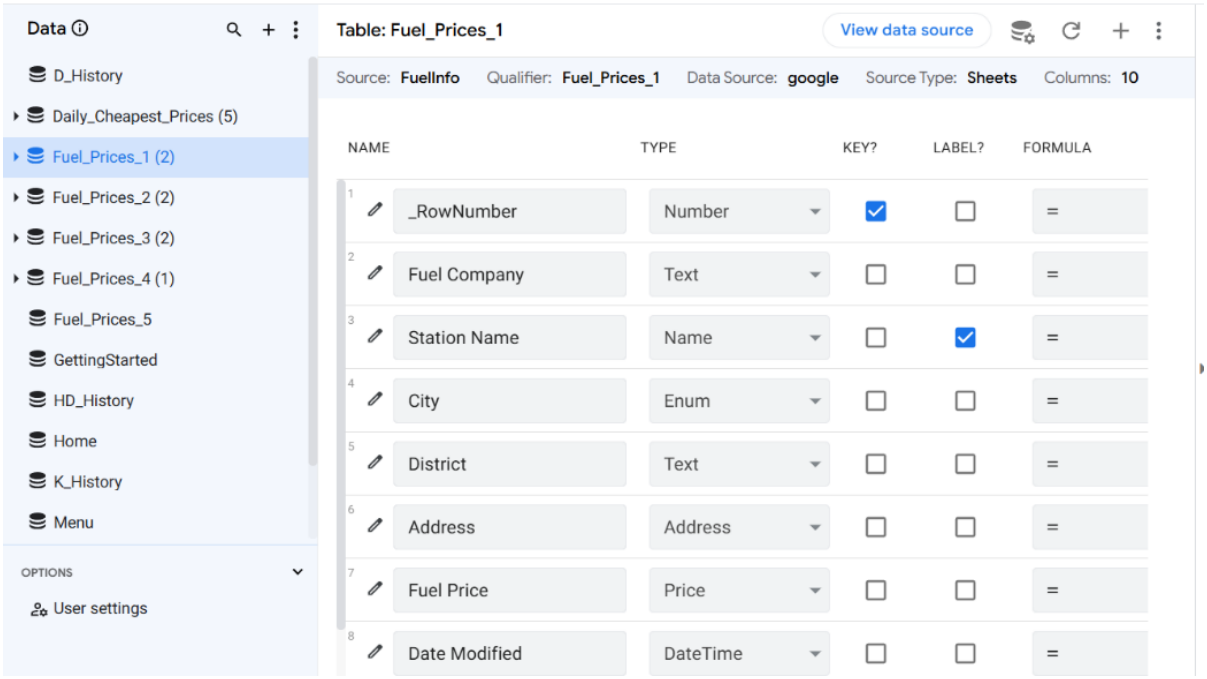



*Figure 4.4 Screen shot of Data import screen in Google AppSheet*

The data structures within AppSheet mirror those of the Google Sheets, with additional virtual columns introduced to enhance functionality. These virtual columns dynamically generate supplementary data on AppSheet server processes in the cloud and are separate from the other layers, such as Google Maps links using stored coordinates and ranking systems for petrol stations based on fuel prices. These enhancements improve user experience by enabling quick access to navigation and price-based sorting within the application.

### 4.2.2 Generating Data Slices

To better utilize the processing and cache provided by Google AppSheet servers, usage of Data Slices is greatly incentivised. This approach enables more efficient distribution of processing responsibilities between Google App Scripts and the AppSheet server. In context of CheapFuelCY, Data slices are generated from the Fuel prices catalogue using the Declarative Expression Language (DEL)[1] provided.

[1] Declarative Expression Language: A declarative, functional, domain-specific expression language used for business logic inside AppSheet. Functions in a similar manor to SQL Queries, using Relational Algebra

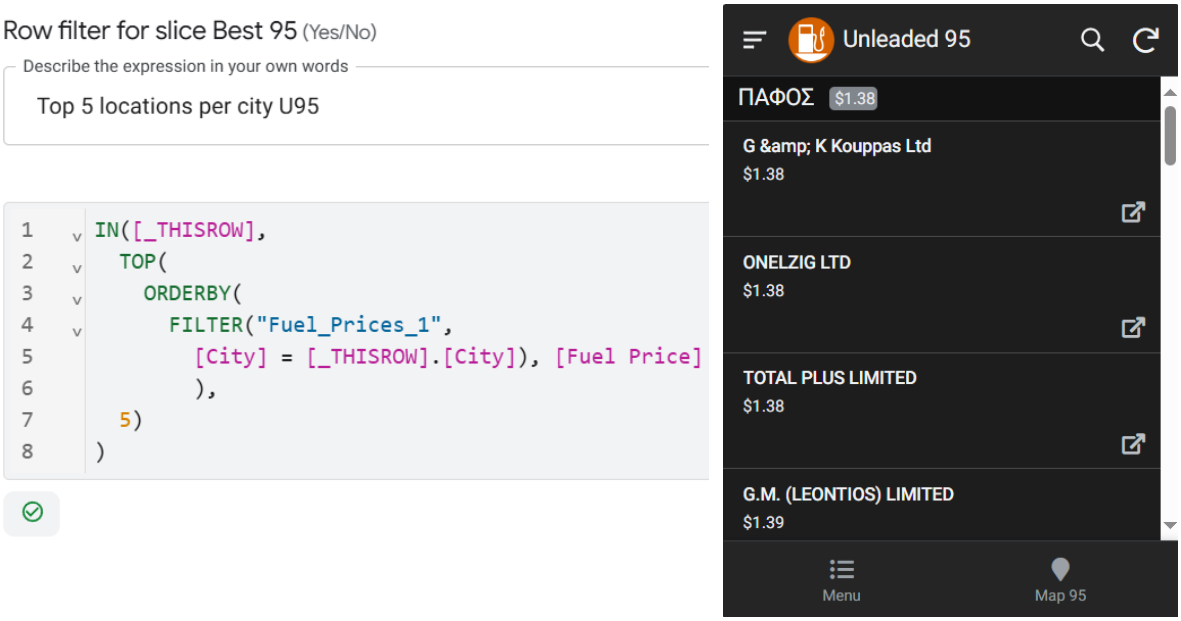



*Figure 4.5 Filter logic applied to data table in order to generate data slice, along with screenshot of results*

Due to high number of entries listed for each fuel type, data slices were created for each fuel type, only including the top 5 cheapest locations per city. Although the choice made for how many entries should be listed (i.e. five) is arbitrary, the design philosophy remains the same, as minimizing the number of entries displayed to a user impacts the user experience positively by making the content more digestible.

The other set of Data slices is dedicated to location-aware filtering, which tailers fuel station visibility based on their proximity to the user. To improve usability and avoid visual clutter on the map interface, a custom data slice is used to filter fuel stations within a defined radius of the user's position. This is implemented using AppSheet's built in DISTANCE() function, which calculates the straight-line distance between two target coordinates. Using the coordinates of the user and the coordinates of the stations, we can generated dynamic data slices, by displaying those within the user-defined radius, which can be modified in User settings.

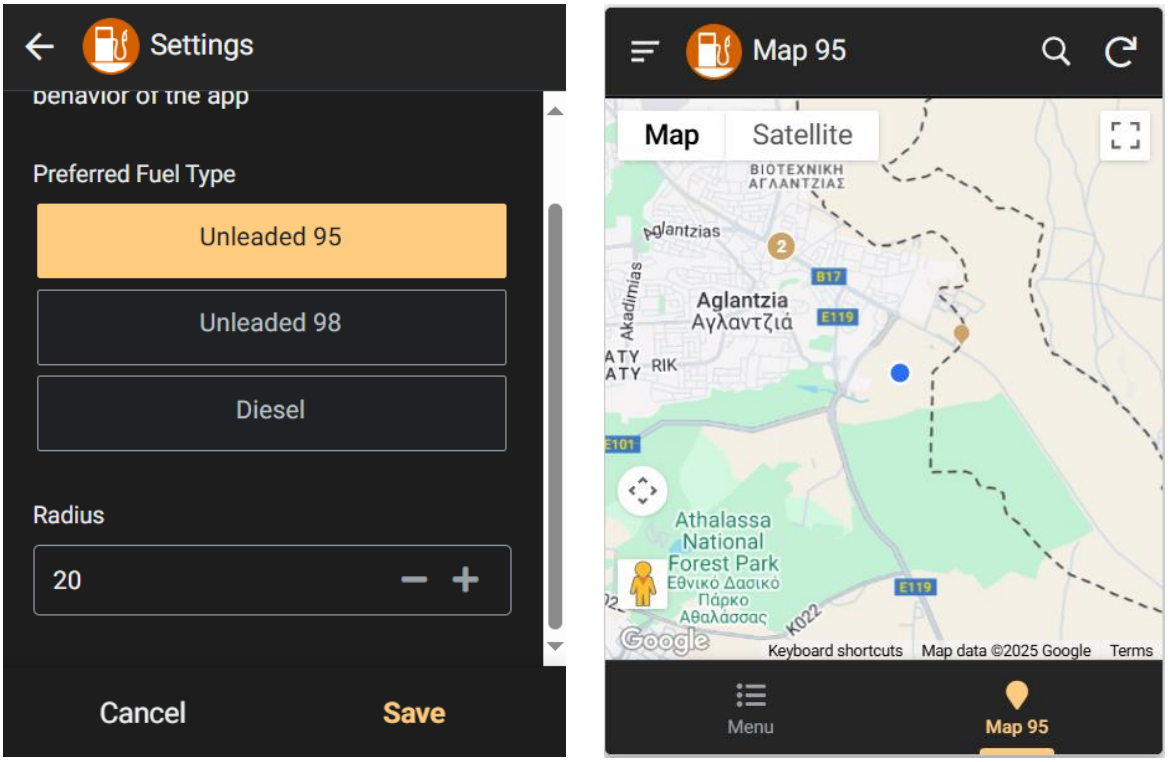



*Figure 4.6 Screenshot of User Options view showcasing the variables, along with another screenshot showing the effect of radius variable*

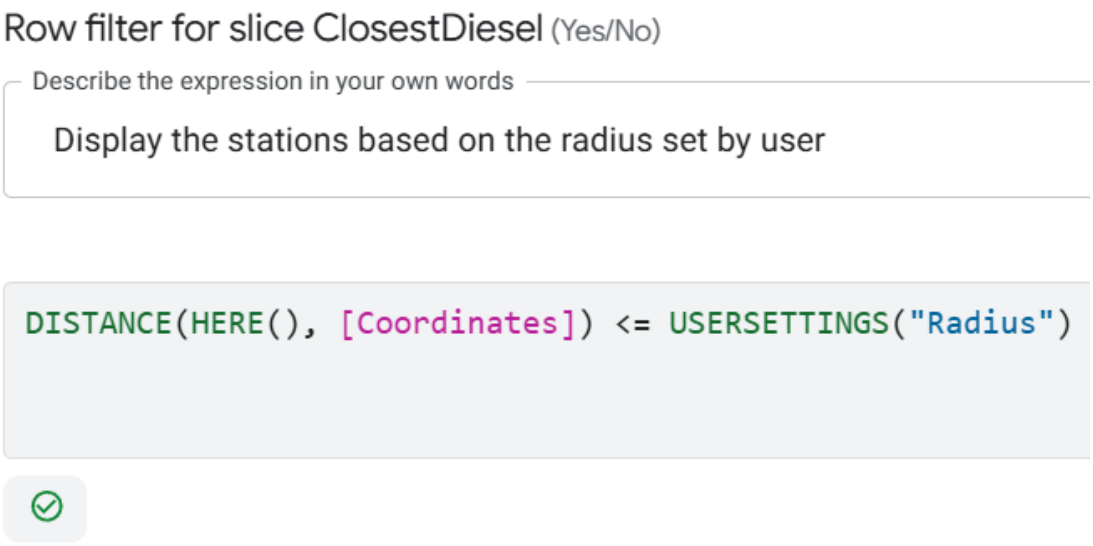



*Figure 4.7 Screenshot of data slice generated to include only stations within proximity*

### 4.2.1 View Construction and Interface design

With all the data sources and filters in place, the views can be created. Each view in AppSheet is directly linked to one and only one data source (Sheet, Slice), and as such, all choices made in regard to data were deliberate in order to make the creation of views as straightforward as possible.

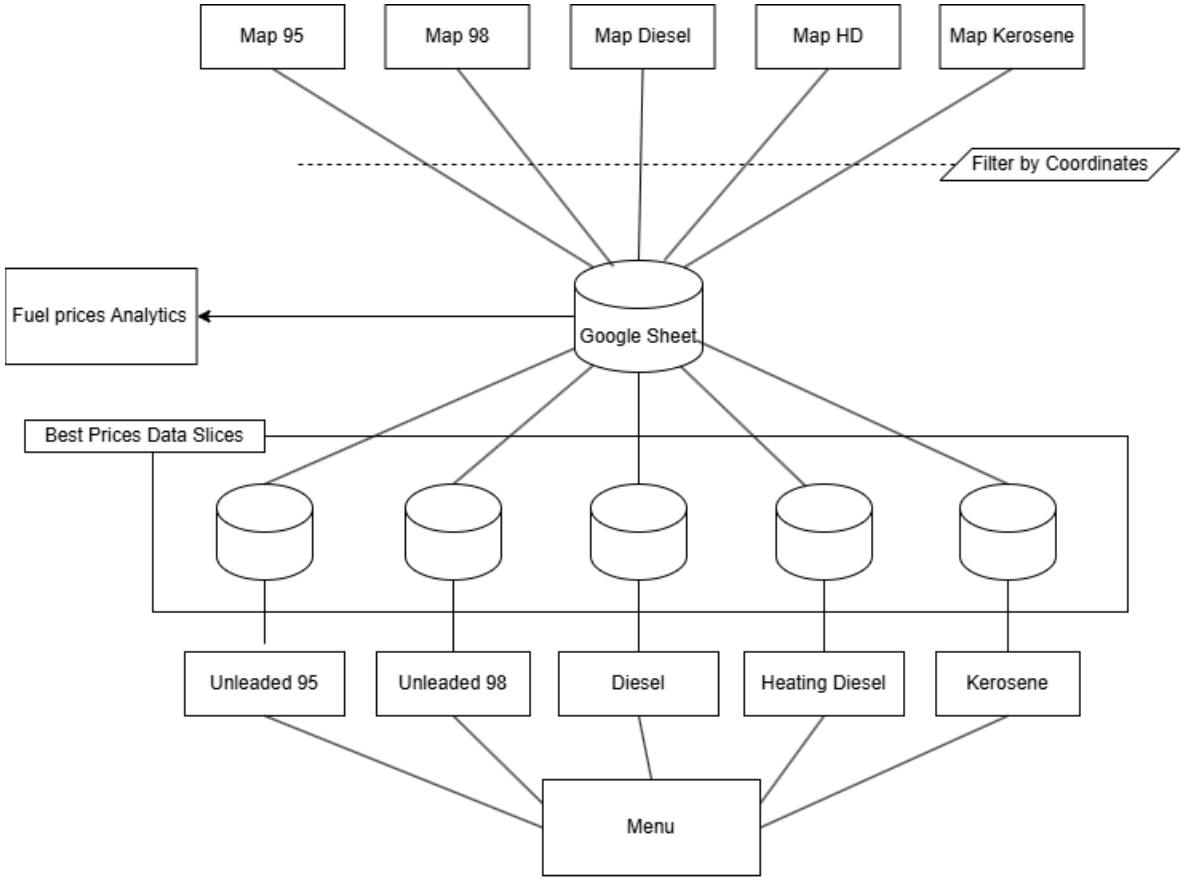



*Figure 4.8 Diagram of data tables, slices, and to what views they are assigned to*

- **Map Views (per fuel type):**

  A dedicated map view was created for each fuel type. These views use the proximity-based data slices to display only nearby stations, relying on AppSheet's native integration with Google Maps to visualize geographic location. The user's fuel preference, defined in the User Settings, determines which fuel map is made accessible, in order to reduce menu clutter. Each map pin allows the user to read more detailed information regarding the station, and allows for them to open the station location in the original Google Maps application that will generate a routing from their location to the station.

- **Menu View (Navigation hub):**

  A central menu was implemented as an intermediate interface, functioning as a navigation dashboard. It presents users with visual icons and labels for each category. These buttons cause the application to open another reference view, with the data associated with that category

- **Best Fuel Prices View:**

These views display the data slice for best prices per city. Stations are grouped by city, with each group displaying a sorted list of the most affordable fuel options. Group headers use aggregation expressions to show the lowest price available per city.

- **Analytics View (Trends over time):**
  To allow for price trend analysis, a series of chart-based views were made using the historical fuel data gathered over time. Each line on the line graph represents a city and their average fuel price per day.

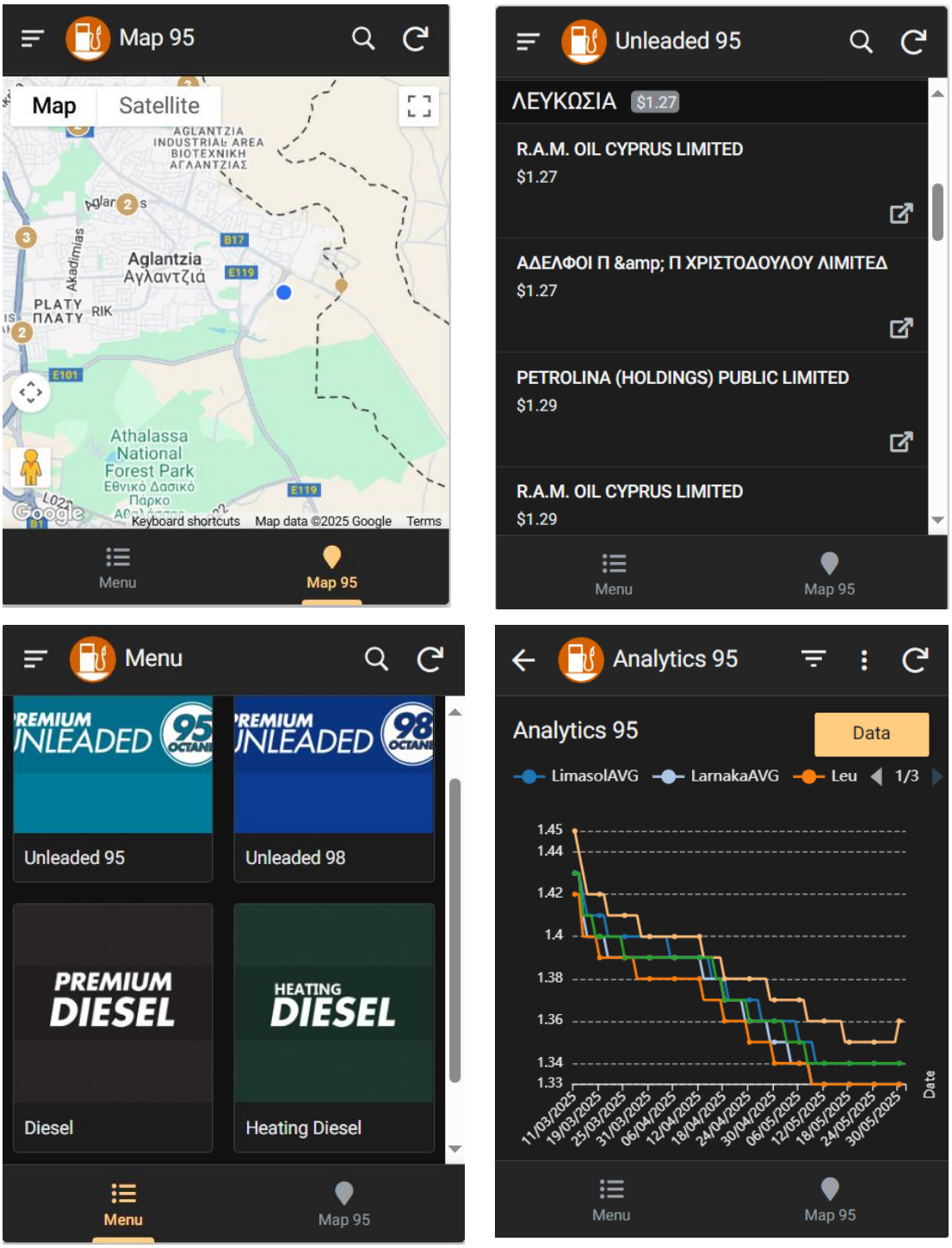



*Figure 4.9 Screenshot compilation of the views aforementioned. In order*

# Chapter 5 Implementation B: UCY Maintenance



## 5.1 Backend Architecture

For the backend implementation of "UCY Maintenance", some major decisions needed to be made as to how and where to isolate data. Due to the desired functionalities of the application, the Google Sheet would have to manage user input, and have methods of checking and handling concurrency, in the event that data gets lost during input. As such, the backend architecture of "UCY Maintenance" is primarily divided into two sections, the first of which being the data sheet, which would store the data that users send via the AppSheet front end, and the second section is responsible for logging any action an individual performs within the app, so that concurrency handling could occur. To aid this, isolated webhooks were deployed, detached from the Google Sheets that would log the data sent by Google AppSheet automations.

### 5.1.1 Google Sheets Database

The database architecture implemented for this system is divided into two separate Google Sheet files, each serving a distinct purpose. The first file, referred to as the UCY Maintenance Sheet, is responsible for storing all user-facing and operational data. The second file, named the Logger Sheet, serves as a logging and audit system used for maintaining historical records and tracking user activity.

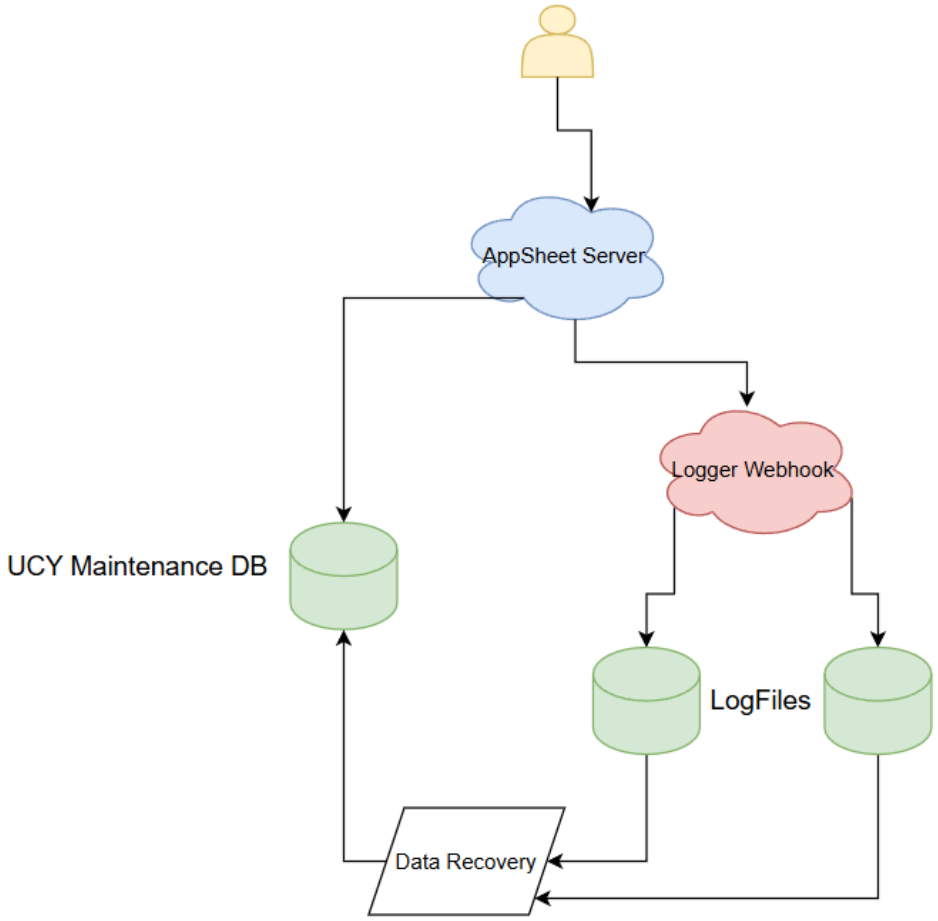



*Figure 5.1 Diagram that shows the topology of the databases and how each one is accessed*

**UCY Maintenance Sheet:**

The UCY Maintenance Sheet is comprised of 7 tables, which are the following:

1. **Report-Log**: This table is responsible for storing all newly submitted maintenance reports. Each entry includes comprehensive metadata such as the reporting user's email and name, their geographical coordinates (LatLong), a textual description of the observed issue, a reference to any submitted photo evidence, and a timestamp of submission. Additionally, each row contains a Boolean field named Verified, which is initially set to FALSE. This field is used as a flag to indicate whether a report has been processed by the automated classification system. The Google Apps Script bound to this database scans the Report-Log for unverified entries in order to trigger the necessary automation for classification and storage.

- **User-Manager**: This table maintains a registry of all users permitted to interact with the application. It contains user-specific information such as identifiers, email addresses, and assigned roles. This table is referenced during app initialization to determine user permissions and to control access to which views and data the individual has.

- **On-Campus and Server:** These two tables are used to store maintenance issues after they have been verified and classified by the large language model (LLM). The classification separates reports into either on-campus physical maintenance issues or server/network-related problems. These tables contain similar metadata to the Report-Log, but also include additional elements such as Priority, Status, Assigned, and Resolved Timestamp. Once an issue is verified and classified, it is appended to the appropriate table. When a technician agrees to resolve a reported issue, their email is recorded in the Assigned column. Upon resolution, the technician updates the Status field to "Resolved", which in turn automatically records the current timestamp under the Resolved Timestamp column.
- **Weekly Report:** This table is dedicated to storing weekly AI-generated summaries of maintenance activity. These reports are produced by aggregating data from unresolved issues and any new issues submitted within the past seven days. The purpose of this table is to provide stakeholders with a high-level overview of system trends, urgent matters, and performance over a weekly cycle
- **Menu:** The Menu table contains user interface metadata required by AppSheet to generate the navigational structure of the application. It serves as a way to inform the AppSheet server where UI elements are stored within Google Drive
- **AI-Calls:** The AICalls table is simply used to catalogue all API activity of the application.

**Log-File Sheet:**

The Log-File Sheets are distinct Google Sheets that have no predefined set of tables. The number of tables is based on the number of users the application has been granted to. Within each user log file, their actions are recorded, with all the information that they submit. This data is used in order to detect possible conflicts that may exist in the database, and if anything is marked as suspicious, an App Script that is bound to the Log-File database accesses the original UCY Maintenance database and checks the Report-Log file. In the event that a conflict did occur, it will attempt to automatically fix the issue, as well as inform the administrator of the application of the event, so that they may manually verify that everything is in order.

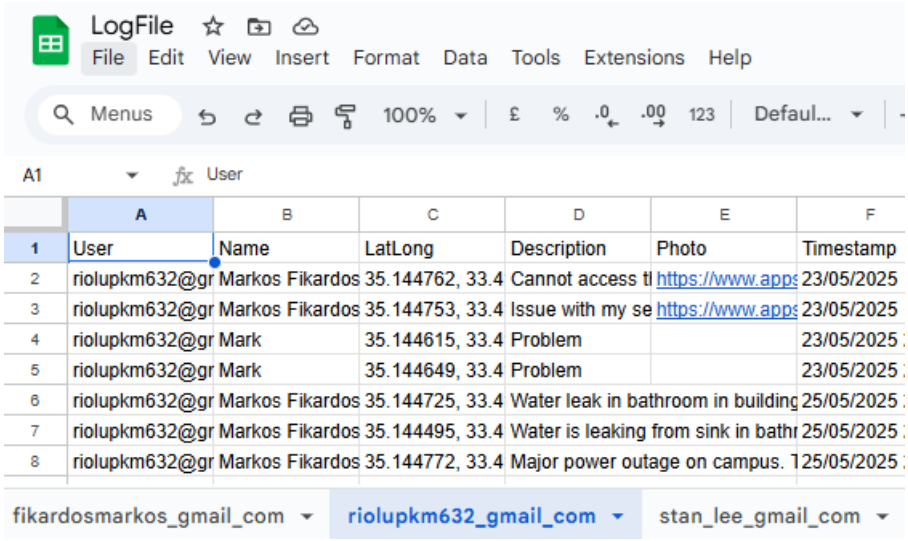



*Figure 5.2 Screenshot of the log file format in Sheets*

### 5.2.1. Google App Scripts + Webhook

The Google App Scripts serve as the most crucial element in the implementation of "UCY Maintenance", as it is responsible for both managing the flow of data, as well as for incorporating LLM with the use of APIs and for monitoring the state of the database. In order for "UCY Maintenance" to function, a total of four (4) distinct App Scripts needed to be created, each of which is isolated and deployed differently. The contents of these scripts can be found in the appendix section.

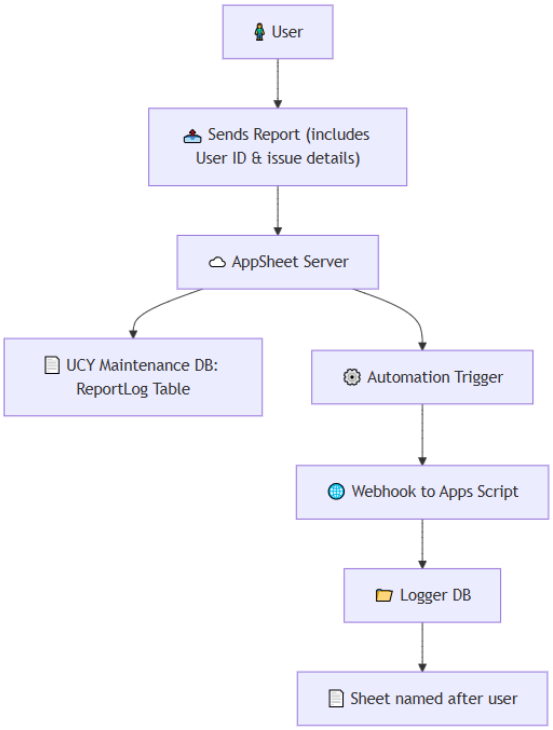



*Figure 5.3 Flowchart of triggered event that invokes the webhook logger*

**I) Backend-UCY:**

Backend-UCY is a script that is bound to the UCY Maintenance database. It is composed of three Google Script files that exclusively manage the data that exists within the UCY Maintenance database.

1. **Process.gs:** This is an automation that triggers every time a new report is submitted to the database. By scanning the Report-Log table for all entries that are unverified, it identifies which entries have yet to be processed. It is responsible for taking all of the reports that were saved into the Report-Log file and sending their descriptions to the Groq API, which contains a multitude of Large language models available for free on a limited quota. The model used for this implementation is 'llama3-8b-8192', and uses a customized prompt in order to control its behaviour so that content that the LLM outputs remains consistent. If for some reason the LLM disobeys its established system prompt, and the data it outputs is not fit for usage, the Process.gs script will simply refuse the answer it provided, and await for another attempt at processing the specific report. If the output of the LLM is in the correct structure, the script extracts the classification and the priority of the report and uses them to decide where to store the processed report, which is either in the ‘on-Campus’ table, or the ‘Server’ table.
2. **CleanUp.gs:** This script ensures that the Report-Log table does not overflow with data, by scanning the Report-Log table every day in order to locate entries that have already been verified. If they have been verified, that implies that they have already been stored to their respective table. As such, removing them is fine. Note that this does not apply for any report that has not been processed.
3. **AnalyzeData.gs:** This automation is responsible for analysing the existing data from the past week of reports, and any other issue report that has yet to be resolved. It compiles all of the entries that meet the criteria from both the ‘on-Campus’ table and the ‘Server’ table. Once it compiles all of the data, it sends it over to the Groq API to be analysed by a LLM using a different prompt. It sends over an engineered prompt, that requests the LLM to structure its message by listing the total number of reports that occurred in the last week, as well as listing the total number of resolved and unresolved issues. After that, it breaks down those issues based on their assigned priority, lists any issues that occurred recently, and generates a short, professional summary regarding the trends, urgent concerns, and patterns it recognizes within all of the reports. It then stores the summary into the ‘Weekly-Report’ table, which members of Maintenance and IT can read it within the “UCY Maintenance” app. The specific prompt that was

used for this implementation can be found in the appendices section at the end of the thesis.

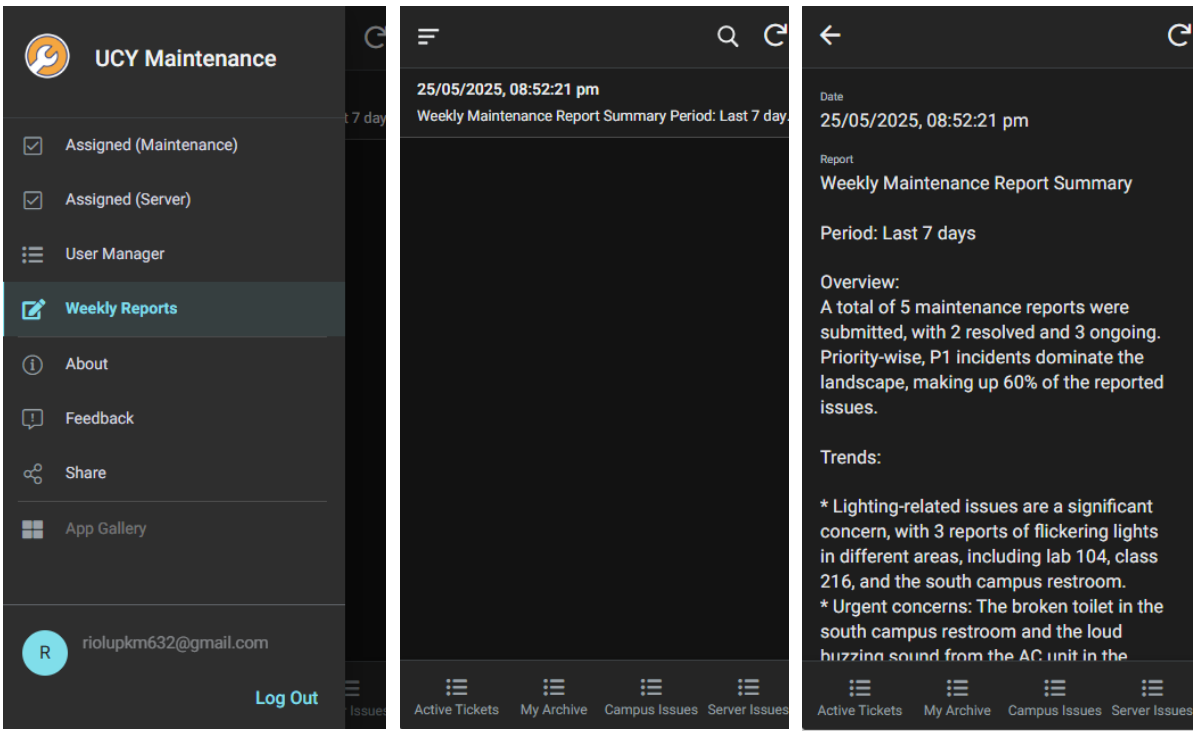
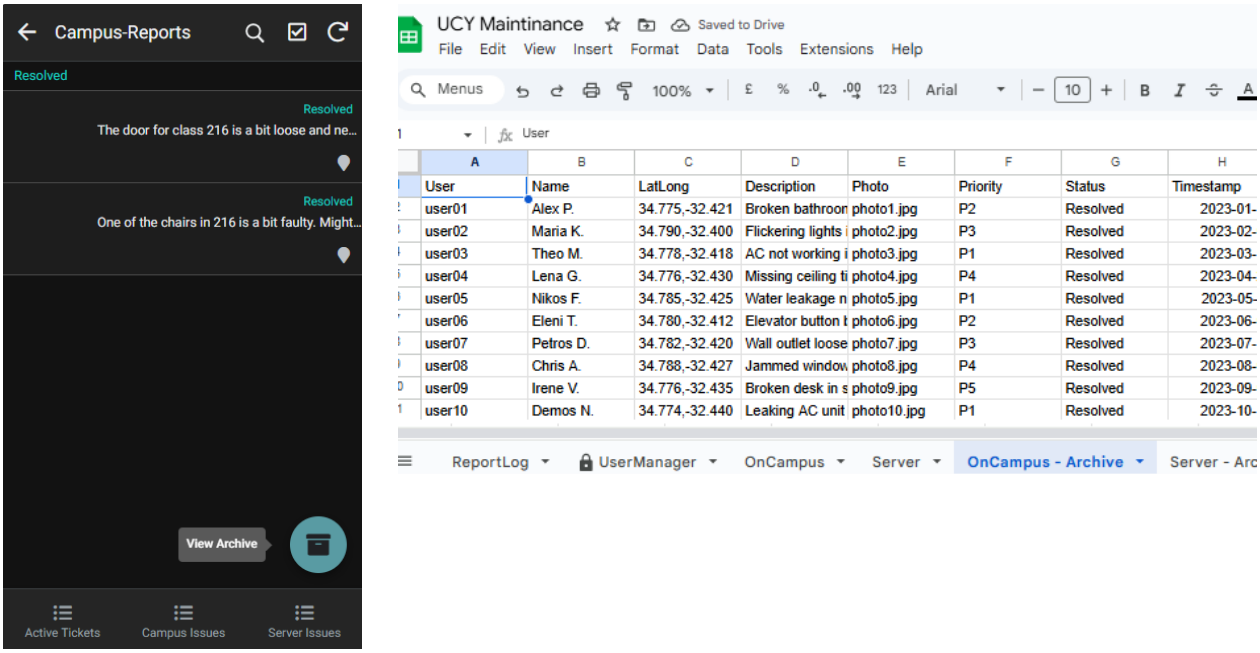



*Figure 5.4 Screenshot of access to AI generated summaries*

**Archive.gs**: This script functions as a programmed button within the Google Sheet database. In the event that the number of entries within the On-Campus and Server tables reaches the point in which there is an impact on performance, an administrator can use the function in order to take all resolved issues that are over a year old and transfer them into their respective "Archive" tables. These tables do not get sent to the AppSheet process, and to view these entries, a user must use the action that is located within the Archive view in order to open the spreadsheet.Figure 5Figure 6Figure 7Figure 8




| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| | User | Name | LatLong | Description | Photo | Priority | Status | Timestamp |
| | user01 | Alex P. | 34.775,-32.421 | Broken bathroon | photo1.jpg | P2 | Resolved | 2023-01- |
| | user02 | Maria K. | 34.790,-32.400 | Flickering lights | photo2.jpg | P3 | Resolved | 2023-02- |
| | user03 | Theo M. | 34.778,-32.418 | AC not working i | photo3.jpg | P1 | Resolved | 2023-03- |
| | user04 | Lena G. | 34.776,-32.430 | Missing ceiling ti | photo4.jpg | P4 | Resolved | 2023-04- |
| | user05 | Nikos F. | 34.785,-32.425 | Water leakage n | photo5.jpg | P1 | Resolved | 2023-05- |
| | user06 | Eleni T. | 34.780,-32.412 | Elevator button t | photo6.jpg | P2 | Resolved | 2023-06- |
| | user07 | Petros D. | 34.782,-32.420 | Wall outlet loose | photo7.jpg | P3 | Resolved | 2023-07- |
| | user08 | Chris A. | 34.788,-32.427 | Jammed window | photo8.jpg | P4 | Resolved | 2023-08- |
| | user09 | Irene V. | 34.776,-32.435 | Broken desk in s | photo9.jpg | P5 | Resolved | 2023-09- |
| | user10 | Demos N. | 34.774,-32.440 | Leaking AC unit | photo10.jpg | P1 | Resolved | 2023-10- |

*Figure 5.5: Screenshot of Archive view and action to view older records within Sheets. The action opens the link to the sheet*

**II) Log-File-Backend:**

The Log-File-Backend script that is bound to the Log-File database. It is responsible for routinely analysing the logs of each user from each user from the past our of operations and cross matching them in order to locate logs which were inserted around the same time. This in turn marks them as “suspicious”, in the sense that if any data entry were to cause concurrency issues, it would be those within the same time frame. Upon detecting suspicious log entries, the Log-File-Backend accesses the “UCY Maintenance” database to investigate. In order to reassure that nothing interrupts the process, it locks the database. It is one of the only instances where locks are utilized, as locking the state of the database can cause performance loss. If everything is in order, and all the data that the suspicious logs represent exist within the database, then it simply unlocks the database. However, if a loss of data is detected, it attempts to automatically insert the data that was missing; after which is then sends an email to the administrator of the database notifying them that a data loss was detected.

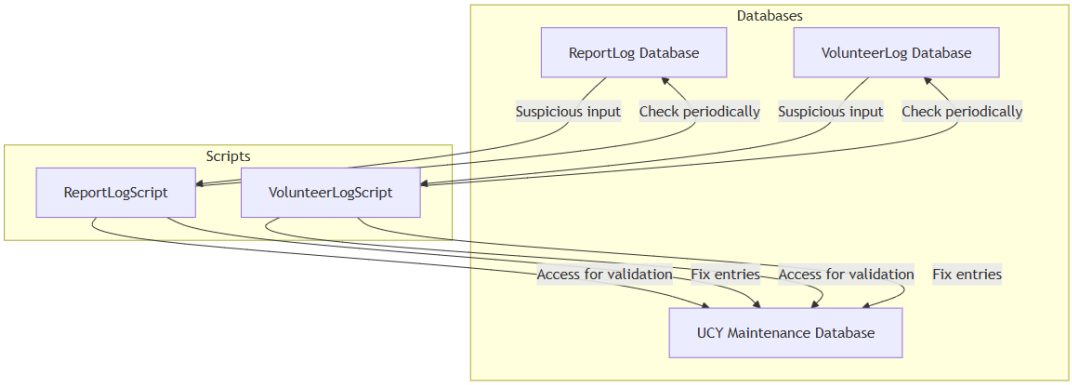



*Figure 5.6: Flowchart that shows the actions and events that the LogFiles follow in order to handle concurrency*

**III) Logger (Webhook):**

The Logger script is the only App script to be deployed entirely independently from any sort of database. It is instead deployed as a webhook, and acts as an intermediary between the "UCY Maintenance" frontend and the Log-File. The idea is that whenever a user inserts a new report into the Report-Log table within the "UCY Maintenance" database, the AppSheet server asynchronously triggers an automation that takes the same data that was inputted, and sends a POST request to the webhook. Once invoked, the webhook takes this data and saves it to a table that is entirely unique to the user that submitted the data to the Report-Log table. Through this methodology, logs can be stored in an isolated manner. Since every instance of a webhook will only access the unique table associated with the user, it can lock the table before inserting the data, without compromising the performance of other webhooks that are attempting to write log entries for other users.

### 5.2.2. Frontend Development

For the implementation of the frontend for 'UCY Maintenance', careful consideration must be given to which pieces of data is delivered to the end user. This is done in consideration for both security and performance. Necessary prebuilt actions must be implemented in order to constrict the level of data a user can manipulate.

### 5.2.3. Data slices and Security filters

When a user initializes the application, it pulls the data from the AppSheet server, which first employs security filters in order to determine what data should be sent over. In the best interest of confidentiality between regular users, and to reduce bloating within the app, multiple

security filters were implemented in order to restrict what data is sent based on the users role that is listed within the User-Manager table within the database. The roles that were implemented are User, Maintenance, IT, and Admin. The role of each user is stored in the User-manager table, and the AppSheet server performs a lookup to determine the permissions that a given user has.

Basic users are only allowed to load data that they inserted into the database. Since a users only function is to report any issues they may come across within the university campus, they do not need access to reports that are submitted by other users. Thus the majority of data is hidden by most security filters, only being allowed to view entries that contain their user email.

Maintenance and IT are given access to more data. While they are unable to view the entire Report-Log, they do have free access to view any entries within the 'on-Campus' and 'Server' that have yet to be assigned. Their roll in combination with their User Email allows them to view an archive of all issues that they have resolved within the 'My Archive

The admin role has access to all data.

Unique data slices were created for the 'on-Campus' and 'Server' tables in order to create the views. They apply simple filters so that views that include either their own personal assignments or problems that have yet to find a volunteer.

### 5.3 Mapping views and Data

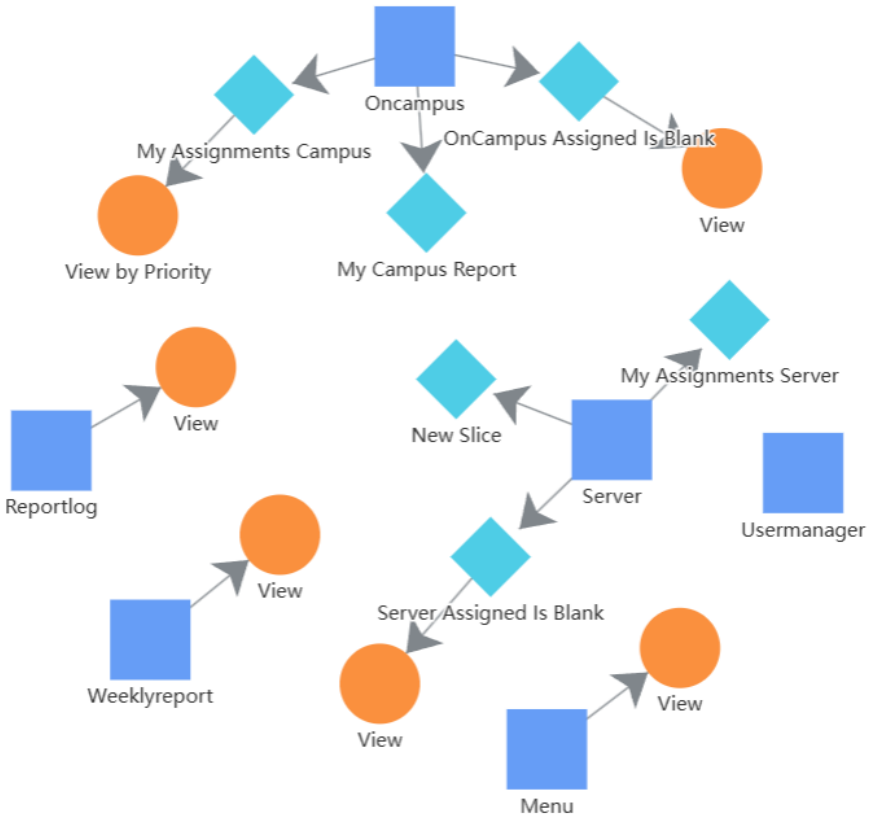



*Figure 5.7 Relation Graph of Views and Data for UCY Maintenance*

The implementation of the views is based on the Data slices listed in Chapter 5.2.3, with the purpose of breaking down the entire dataset into more digestible views. Figure 5.6 shows the uses of these data slices in order to limit views to personal reports, personal assignments, and available active tickets for maintenance to volunteer for.

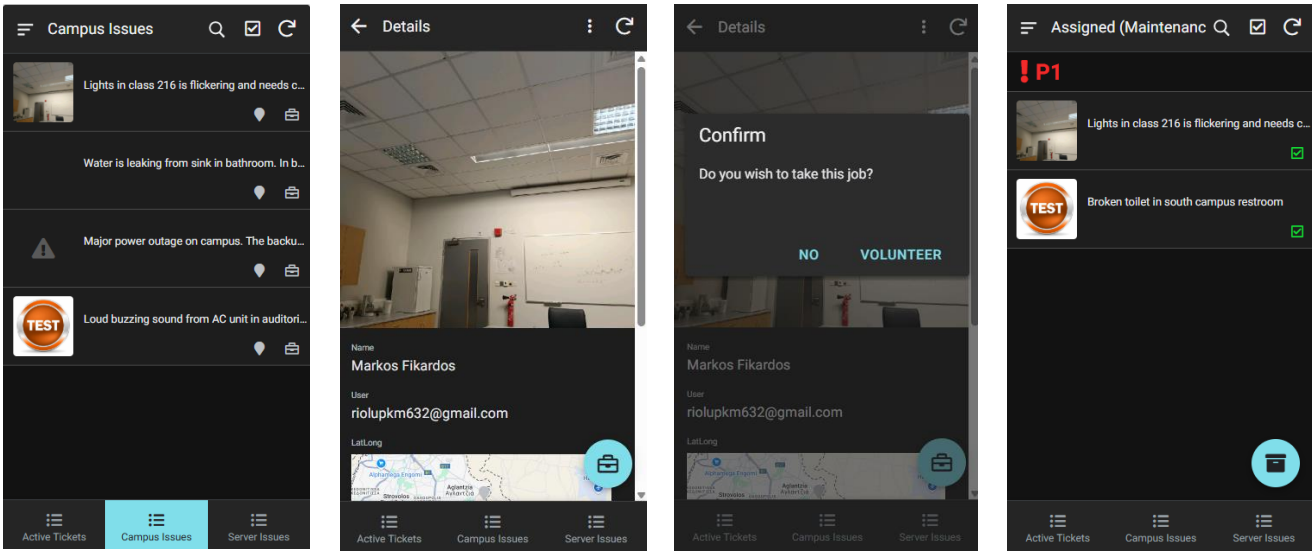



*Figure 5.8 Screenshots showing individual views, which order you access these views in order to volunteer for a job and to view it in your tasks view*

The process of separating is also important for defining which actions are available to the user. For example, since assigning a task to yourself and verifying the completion of a task both pertain to the same set of data, if not separated by views, it would result in some unintended app behavior, such as staff of maintenance being able to mark an issue as resolved without first assigning the task to themselves.

## 5.4. Automations

Automations are deployed to track when specific events occur within the app. When triggered, they perform a fixed set of operations that you are allowed to order. These operations include basic automations such as modifying values, sending emails, and calling App Scripts and Webhooks. For the implementation of 'UCY Maintenance', four (4) automations were created both to add functionality. The most important of the two is the Log-Bot, which monitors the Report-Log table in the AppSheet server, and when triggered, takes the input that caused the instance and proceeds to take that data and send it over to the Log-File webhook.

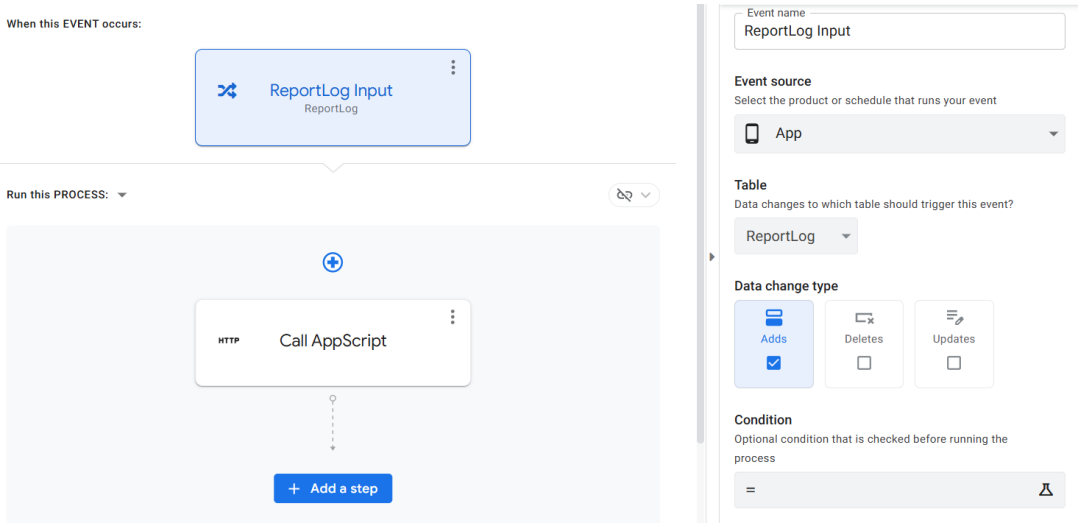



*Figure 5.9 Screenshot of Automation creation view in AppSheet*

**onCampus & Server Automations:** These are two automations that monitor the onCampus and Server tables respectively. Whenever a member of maintenance updates an entry to signify that they are volunteering to work on the reported issue, the automation triggers and prepares data to be sent to the distinct log file. This data is used by the log files respective backend scripts.

**Email-Bot:** When a new report is added to the ReportLog table, the bot triggers and send a structured email to all staff of maintenance to inform them that a new issue has been reported, with the details of the report listed in the email

**Report Log-Bot:** Since logs for inputting a report store different variables in comparison to logs for tracking task assignments, another automation was created that documents the data that the user reported into the LogFile backend.

# Chapter 6 Evaluation



In this chapter, we will evaluate both the technical performance our two case studies: "CheapFuelCY" and "UCY Maintenance"; as well as the development experience of building applications using AppSheet, Sheets, and App Scripts. This includes a detailed analysis of app speed under various conditions, such as different network configurations, caching states, and filtering strategies. This is all in order to assess how efficiently the platform delivers data to the end user in various scenarios.

As an undergraduate student in Computer Science, I bring a foundational understanding of programming and system architecture to this evaluation. One of my goals is to determine whether AppSheet is genuinely accessible for complete beginners with zero background in software development, or whether its true value lies in boosting the productivity of users who already have technical experience.

## 6.1 Technical Benchmark Methodology

The two primary metrics used to evaluate the technical performance of our applications are response time and concurrency handling. Response time refers to the speed at which the application retrieves and displays data to the user under varying networking and caching conditions. Concurrency handling evaluates how the system behaves when faced with simultaneous user input within a short time frame.

To assess the baseline response time, we will focus on "CheapFuelCY", as it is a read-only application that fetches and displays data. Its behavior allows us to isolate the performance metrics under different configurations and loading strategies, as well as evaluating to what degree Google AppSheet's 'Server caching' feature improves performance. With the baseline performance established, focus will then be shifted to "UCY Maintenance", to review how its response time fluctuates depending on the implementation of Security Filters.

**Response time metrics (via Google DevTools):** With the usage of Google Chrome DevTools, it is possible to force no caching for the website which simulates a real world scenario of not having the data loaded on your device and thus reflecting the data retrieval time.

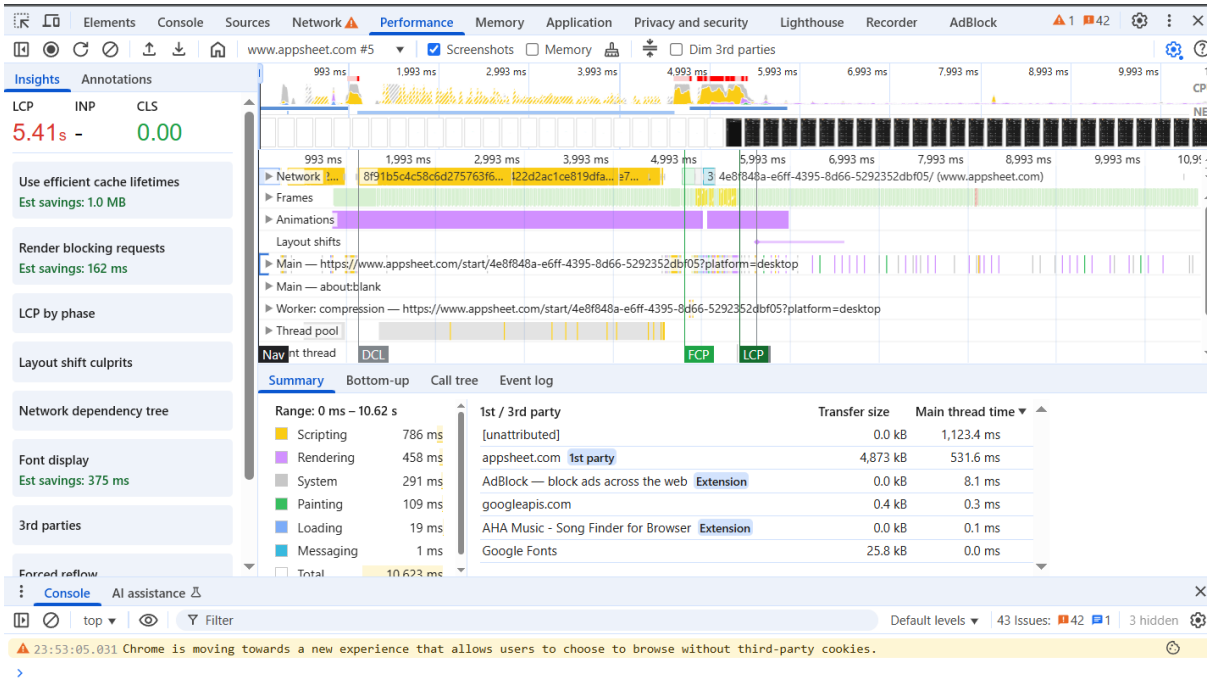



*Figure 6.1 Screenshot of Google DevTools that display response metrics for evaluation*

The following metrics are used

1. **Largest Contentful Paint (LCP):** LCP measures the time it takes for the largest visible content element (typically a prominent image, table, or text block) to fully render within the viewport. It is considered a core indicator of perceived load speed to the end user. This metric does not imply that all the data is loaded, rather it conveys the idea that the website has finished loading to the user, thus creating the illusion of faster performance. A good LCP value is recommended to be withing 2.5 seconds, while anything above 4 seconds is considered poor.
2. **First Contentful Paint (FCP):** FCP marks the point at which the first piece of DOM Content is rendered on the screen. It gives insight into how quickly the user sees any visual feedback after initiating the application load / browser load. A good FGCP value is recommended to be within 1.8 seconds, while above 3 seconds is considered poor.
3. **DOMContentLoaded (DCL):** DCL records the time at which the initial HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and subframes to finish loading. This metric indicates when the application

content becomes structurally ready for interaction and further script execution. A good DCL value is recommended to be withing 1 second, while anything above 2.5 seconds is considered poor.

**Network Conditions:** To evaluate application performance under various network conditions, tests were run using the University of Cyprus's Wi-Fi network, which provided a stable connection with measured speeds of approximately 230 Mbps download and 210 Mbps upload. This environment served as a baseline for assessing application responsiveness.

To simulate real-world mobile network scenarios, additional tests were performed using Google DevTools to throttle network speeds to emulate:

1. **Boundless (unthrottled):** Reflecting access from established Wi-Fi networks
2. **4G(Fast):** Representing typical modern mobile network conditions
3. **3G:** Emulating slower, legacy mobile network environments.

**AppSheet Server Caching Mechanism:** AppSheet employs a server-side caching of back-end data. Whenever a user initializes an application, the request is sent to AppSheet server, and if the data does not exist in its cache, it will pull it from the backend so that it may generated virtual columns and process which chunks of data the requesting user is permitted to have. This creates two different scenarios:

1. **Cold boot:** The data does not locally exist on AppSheet server, thus it must be pulled from the cloud source. This results in a slower response time
2. **Warm Cache:** The data exists in AppSheet's server cache, and does not have to request the data from the cloud source, increasing response time.

AppSheet offers an optional "Server Caching" feature as part of its Core subscription plan. When enabled, this feature allows AppSheet to store your data on its own servers rather than making calls to the cloud provider. This can greatly improve performance during cold loads, as data now lives closer to the AppSheet endpoint.

It's important to note that while the "Server Caching" functionality can improve cold load performance, there should be no discernible difference in performance during warm loads, since in both cases, the AppSheet server already leverages the data stored in its cache without re-fetching.

**Preloaded Assets:** When a user accesses the application for the first time, assets such as elements and scripts must be downloaded so that the interface can be assembled. This adds a bit of additional latency. As such, we take into account both the response time for first time users, and returning users which should already have some of the assets loaded on their device.

**Evaluating the effect of Security Filters:** Since security filters directly limit what data is sent from the AppSheet server to the end user in order to retain data privacy, this also has an effect on performance; by limiting the amount of data that gets sent to the end user. To test Security filters under extreme scenarios, a large number of AI-Generated entries will be inserted into the database manually via the Google Sheet backend. Tests will follow the same methodology of measuring for response time, with the expected outcome being that larger throughputs of data will require more time.

### 6.2 Developer Experience Evaluation Methodology

The evaluation of the Developer Experience is conducted from the perspective of a technically literate user, and it focuses on the following aspects:

**Development Complexity:** The ease or difficulty of building functional applications using AppSheet is assessed by referencing specific implementation examples. These examples are used to highlight the level of technical knowledge required to achieve certain software requirements. Where appropriate, tasks are analysed in terms of the implicit understanding needed of programming fundamentals, data modelling, or cloud architecture

**Function Scope and Limitations:** This dimension focuses on assessing what types of applications can realistically be developed using AppSheet as the primary frontend. Through the development of two case-study applications, platform flexibility, extensibility, and architectural limitations are reviewed.

### 6.3 Technical Benchmark Results

Since all of the conditional parameters exist at the same time, it's difficult to isolate the impact on performance each element has. As such, when evaluating the effects, on a specific variable, we need to also take into account what other variables are in play in order to effectively determine our conclusions.

**A) Network speeds:** Network speeds persistently have a substantial effect on all core metrics. When considering scenarios where the 'CheapFuelCY' exists as a process within the AppSheet server (warm), a substantial increase in response time is measured, increasing the LCP time by around 3x its unbound counterpart. Any other characteristics that affect the response time while the server cache is warm is reflected in its 4G response time counterpart. However, when observing these metrics while the application process is cold, we can see that for 4G connections, it becomes the primary bottleneck along side the cold boot. This consistency

occurs because the key bottlenecks such as the initial server response, asset download, and data fetching are dominated by the limited and stable throughput imposed by network throttling.

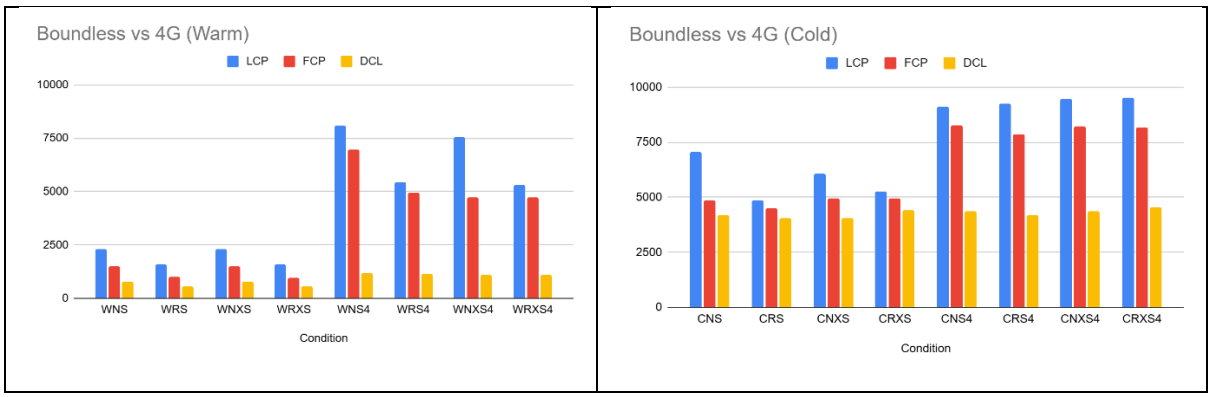



*Figure 6.2 Results of Response time benchmarks, separated by state of AppSheet process (Warm / Cold)*

As for 3G connections, the level at which it bottlenecks the transaction is so severe that no other element matters anymore, consistently displaying response times of over a minute. As such, due to time constraints and how there wouldn't be much meaningful data from measuring with 3G throttling, a few metrics were gathered under optimal conditions to reflect the estimated response time for 3G.

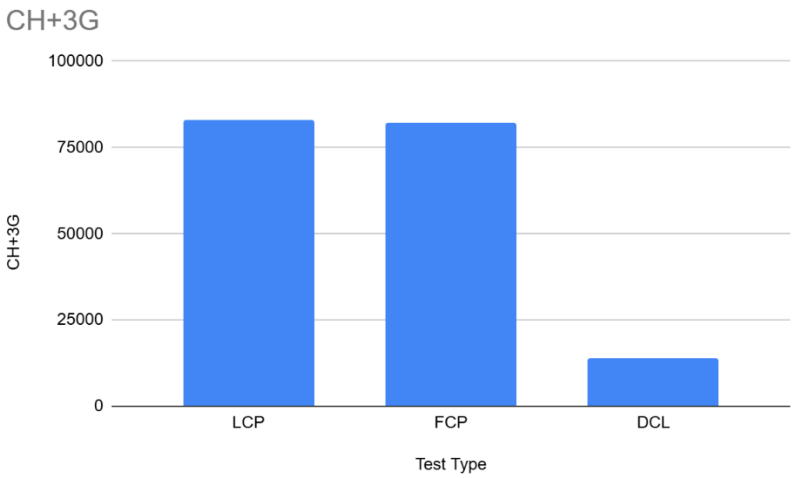



*Figure 6.3 Results of 3G throttling benchmarks. Only performed on parameters warm with cache*

**B)Warm vs Cold access:** While the process exists within the AppSheet server (warm), we can observe response times of around 1.5 – 2.3 seconds, with returning users making up the faster times recorded due to the fact that they have a portion of the assets preloaded on to their device, and new users requiring slightly longer as to fetch the elements from the AppSheet server. One thing important to note is the fact that while the process is warm, there is no discernable difference between having AppSheet's 'Server Caching' feature enabled. This is due to the fact

that while the process is active, AppSheet already has all of the elements from the database within its cache, which means that it doesn't have to fetch the elements of the database, regardless of where they are stored.

In stark contrast, the response time measured by users while the process was not active on the AppSheet server is both noticeably slower, as well as reflecting the effect that AppSheet's 'Server caching' has on response time. While overall performance is twice as slow in comparison to already active processes, without 'Server Caching', the LCP time increases by around 500-800 milliseconds. This increase can be attributed to the fact that without 'Server caching', the data lives further away from the process, thus the AppSheet server must call the Sheets API to retrieve the data before sending it to the user.

Finally, regarding DCL, a marginal increase in its recorded time can be observed during cold starts. The observed delay in DOMContentLoaded timing during cold starts of AppSheet applications can be attributed to the fact that during cold starts, the AppSheet server must first initialize the app's runtime environment. This process includes spinning up the backend process for the app instance, allocating resources such as memory and ports, and the execution of any startup logic. Only after these steps can the server begin streaming structured HTML and scripts to the client, which directly effects when the browser reaches the DOMContentLoaded milestone.

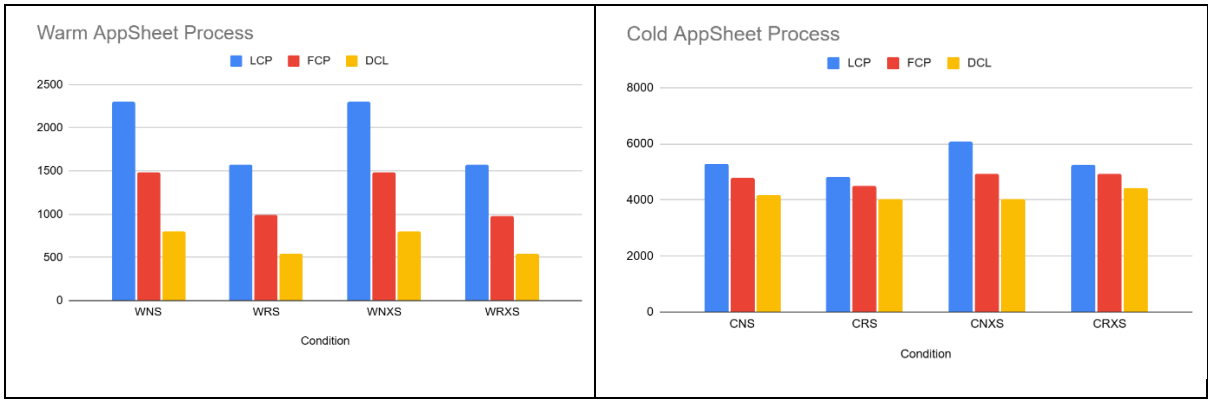



*Figure 6.4 Benchmarks results grouped by warm cold, excluding 4g*

**C) Security Filters:** To evaluate the true impact of Security Filters, the tests are conducted with the AppSheet instance already existant within the AppSheet server. By making modifications to the number of entries within a table and then running a benchmark, we evaluate the time it takes for the AppSheet server to update with all the new AI-Generated entries and then how long it takes for the AppSheet server to deliver the content to the client.

The benchmarks reveal that Security Filter do in fact help with response time, particularly with the Largest Content Paintful metric. With the use of Security Filters, we can observe that the response time across all different number of rows remains constant, reflecting similar

performance times to the baseline we established with the set of benchmarks for ‘CheapFuelCY’. When Security filters are not deployed, the rensponse time gradually increases based on the size of the payload. It should be noted that while on average the average response time would increase along with the payload size, there was a great variance detected. Some measured resposne times went as high as 2.2 seconds, while others went as low as 1.6 seconds for LCP.

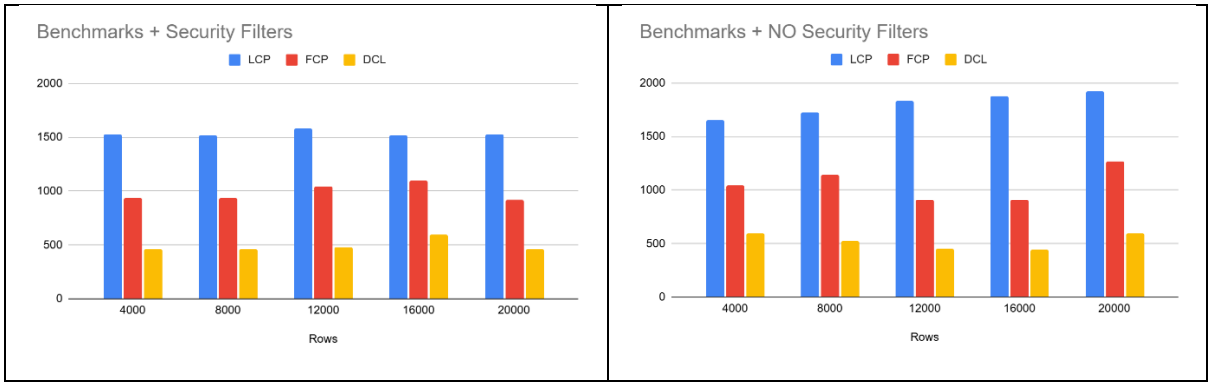



*Figure 6.5 Results of Benchmarks evaluating the effect of Security Filters on performance*

## 6.3 Review of Development Process

**Development Complexity:**

The development experience using AppSheet and Google’s ecosystem proved generally positive, particularly for rapid prototyping and structured data-driven applications. Over the course of the study, multiple prototype applications were developed, though only a subset reached full implementation. Notably, the ‘UCY Maintenance’ application, which required careful consideration of concurrency handling, was developed in a relatively short timeframe, demonstrating the speed and approachability of the platform.

AppSheet’s machine learning–assisted interface, which attempts to predict the developer's intent when configuring app behaviour, proved helpful, although only after sufficient contextual clues were established. In the early stages of app setup, these recommendations were often inaccurate or misaligned with the actual design goals.

Certain development patterns within AppSheet were unconventional but manageable. For example, building a custom navigation menu required creating a dedicated data table to store button entries, which were then linked to actions that redirected users to specific views. While this was not inherently complex, it did require a mental shift in thinking, particularly for developers accustomed to traditional UI frameworks.

To fully unlock AppSheet’s potential, proficiency in Google Apps Script is essential. Although AppSheet handles many frontend and simple logic tasks, more advanced operations, such as

automated events, API integrations, or complex conditional workflows, must be implemented in Apps Script. For example,

**Flexibility:**

Despite its strengths in accessibility and rapid deployment, AppSheet imposes several significant limitations that constrain its use in more complex or dynamic scenarios. The most constricting is the inability to perform local or client-side computation. All logic must be processed via cloud-based services, which increases backend load and limits responsiveness for computation-heavy tasks.

A practical illustration of this limitation arose during the development of the 'CheapFuelCY' application. An initial design idea included an algorithm to rank fuel stations based on both their prices and geographic proximity to the user instead of simply listing them in descending order. However, because the app was publicly accessible and did not enforce user authentication, it was not possible to generate personalized experiences, since the data could not be stored on the local client's device, and there isn't a way to store personalized content due to having no way of identifying individual users and which data was calculated based on their parameters. Furthermore, AppSheet lacked native capabilities to execute custom algorithms dynamically within the app interface, and did not support views that allowed As a result, this feature was ultimately discarded due to the platform's technical constraints.

These examples reflect a broader pattern: AppSheet is highly effective for structured, data-driven applications with moderate complexity, but it becomes increasingly restrictive when attempting to build context-aware, personalized, or computationally intensive functionality.

# Chapter 7 Conclusion and Future work



## 7.1 Conclusions

This study set out to evaluate the technical viability and development experience of No-code platforms, specifically within the Google ecosystem, through the implementation of two real-world applications. Based on the results, it can be concluded that AppSheet, when used alongside Google Sheets and Google Apps Script, offers a compelling solution for rapid application prototyping and small to medium scale deployments, with the capability of supporting large deployments by switching out the Google Sheets and App Script layers with enterprise technologies.

While AppSheet may not be fully accessible to completely non-technical users, it significantly accelerates the development process for technically literate individuals or small teams. Its visual interface, tight integration with cloud services, and minimal setup requirements make it particularly well-suited for small businesses or local organizations with limited IT resources. In these contexts, AppSheet provides an efficient path to digital transformation without the overhead of traditional software development or DevOps infrastructure.

Furthermore, the platform demonstrated strong performance metrics in terms of response time and availability, with no significant latency issues under standard usage conditions. Combined with its low operational cost, which is typically bundled under Google Workspace subscriptions, AppSheet offers an economically viable option for organizations that need lightweight, reliable, and maintainable solutions.

In summary, AppSheet proves to be a solid candidate for rapid prototyping, internal tools, or production-ready applications in environments with modest complexity. Its limitations in flexibility and extensibility suggest that it is best viewed as a productivity enhancer for developers, and as a viable method for quick implementation when faced with sudden requirements for digital solutions.

## 7.2 Limitations

A common hurdle that was encountered during this thesis was the limited access to online resources, due to the fact that all of the work did not involve any sort of funding. The digital world is not the same as it was a decade ago during the start of the Web 2.0, where tech firms would provide API's and cloud services at no cost. With the focus shifting over to profit, there aren't many resources that provide their functions and software free of charge. This limited both what was possible to implement in my applications, as well as hindered the application brainstorming process.

## 7.3 Future Work

While this study focused on evaluating No-code development within Google's ecosystem, several promising directions for future research and development emerged throughout the process.

**1. Exploring Hybrid Platforms for Semi-Technical Developers**

Given that this thesis assumes a user base with some technical proficiency, though not professional developer experience, it would be valuable to explore alternative platforms such as OutSystems. These platforms strike a balance between No-code rapid prototyping and traditional development flexibility, offering users the ability to extend application logic with custom JavaScript, APIs, or back-end services. Future studies could assess whether such hybrid platforms provide better scalability and long-term maintainability while retaining the speed and accessibility.

**2. Offloading Logging to External Infrastructure**

One technical bottleneck observed during development involved quota limitations imposed by Google Apps Script, especially in operations involving repeated background writes such as event logging. A viable direction for future implementation would be to offload log handling to an external cloud infrastructure, such as an AWS-based logging service or an enterprise-grade database like PostgreSQL or Firebase. Alternatively, hosting a lightweight custom server specifically for log reception and storage could mitigate the risk of exceeding quotas while improving concurrency and write throughput.

# Bibliography


[1] Gartner, «Gartner,» Gartner, 10 November 2021. [Ηλεκτρονικό]. Available: https://www.gartner.com/en/newsroom/press-releases/2021-11-10-gartner-says-cloud-will-be-the-centerpiece-of-new-digital-experiences. [Πρόσβαση 18 5 2025].

[2] U. F. Alexander C. Bock, «Springer Nature Link,» 5 February 2021. [Ηλεκτρονικό]. Available: https://link.springer.com/chapter/10.1007/978-3-031-16947-2_1. [Πρόσβαση 15 May 2025].

[3] IBM, «ibm,» IBM, 6 May 2025. [Ηλεκτρονικό]. Available: https://www.ibm.com/think/topics/faas. [Πρόσβαση 6 May 2025].

[4] J. K. O. H. Till Guthardt, «ACM Digital Library,» 31 October 2024. [Ηλεκτρονικό]. Available: https://dl.acm.org/doi/pdf/10.1145/3652620.3688332. [Πρόσβαση 6 May 2025].

[5] D. H. L. Assaf Avishahar-Zeira, «ACM Digital Library,» 19 October 2023. [Ηλεκτρονικό]. Available: https://dl.acm.org/doi/10.1145/3622758.3622893. [Πρόσβαση 6 May 2025].

[6] G. A. Juciê Xavier Silva, «ACM Digital Library,» 21 December 2024. [Ηλεκτρονικό]. Available: https://dl.acm.org/doi/10.1145/3701625.3701649. [Πρόσβαση 6 May 2025].

[7] Z. Yan, «arxiv,» 28 December 2021. [Ηλεκτρονικό]. Available: https://arxiv.org/abs/2112.14073. [Πρόσβαση 6 May 2025].

[8] Google Developers, «Colossus under the hood: a peek into Google's scalable storage system,» Google, 2021. [Ηλεκτρονικό]. Available: http://carpex.usal.es/~visusal/rodrigo/documentos/papers/Hildebrand2021.pdf. [Πρόσβαση 6 May 2025].

[9] Google Developers, «Google Drive Help Center,» 2025. [Ηλεκτρονικό]. Available: https://support.google.com/drive/answer/37603?sjid=14062174831409046216-EU. [Πρόσβαση 6 May 2025].

[10] Google Developers, «App Performance: Core Concepts,» Google, 2025. [Ηλεκτρονικό]. Available: https://support.google.com/appsheet/answer/10105761. [Πρόσβαση 6 May 2025].

[11] G. Developers, «App Scripts Service Quotas,» [Ηλεκτρονικό]. Available: https://developers.google.com/apps-script/guides/services/quotas. [Πρόσβαση 6 May 2025].

[12] Google Developers, «Guideline of Publishing Google App Script addons,» Google, 2025. [Ηλεκτρονικό]. Available: https://developers.google.com/workspace/add-ons/how-tos/publish-add-on-overview. [Πρόσβαση 6 May 2025].

[13] Google Developers, «Google Support,» Google, 2025. [Ηλεκτρονικό]. Available: https://cloud.google.com/architecture/framework/security/shared-responsibility-shared-fate. [Πρόσβαση 6 May 2025].

[14] J. M. Travis Breaux, «ACM Digital Library,» 21 6 2021. [Ηλεκτρονικό]. Available: https://dl.acm.org/doi/10.1145/3440753. [Πρόσβαση 6 5 2025].

[15] D. H. L. Assaf Avishahar-Zeira, «ACM Digital Library,» 19 October 2023. [Ηλεκτρονικό]. Available: https://dl.acm.org/doi/10.1145/3622758.3622893. [Πρόσβαση 6 May 2025].

# Appendix A

Code behind “CheapFuelCY”. API Payloads are hidden to hide personal information (Government portal, API key)

```
//First scheduled function. Will call the sendRequest function five times so that the postman protocol can prepare the data
enviroment.
function requestAllFuelTypes() {
  var fuelTypes = [1, 2, 3, 4, 5]; // Fuel types (Unleaded95 → Kerosene)
  var correlationIDs = {}; // Store CorrelationIDs
  Logger.log("Requesting Correlation IDs for all fuel types...");
  for (var i = 0; i < fuelTypes.length; i++) {
    var fuelType = fuelTypes[i];

    try {
      var correlationID = sendSubmissionRequest(fuelType); // Call API

      if (correlationID) {
        correlationIDs[fuelType] = correlationID;
        Logger.log("Stored Correlation ID for Fuel Type " + fuelType + ": " + correlationID);
      } else {
        Logger.log("Failed to retrieve Correlation ID for Fuel Type " + fuelType);
      }
    } catch (error) {
      Logger.log("Error requesting Fuel Type " + fuelType + ": " + error.message);
    }
    Utilities.sleep(2000); // Prevent API spam by adding a delay
  }
  // Store all correlation IDs in script properties
  PropertiesService.getScriptProperties().setProperty("fuelCorrelationIDs", JSON.stringify(correlationIDs));

  // Verify stored Correlation IDs
  var storedCorrelationIDs = JSON.parse(PropertiesService.getScriptProperties().getProperty("fuelCorrelationIDs") || "{}");

  Logger.log("Verifying stored Correlation IDs...");
  fuelTypes.forEach(fuelType => {
    if (storedCorrelationIDs[fuelType]) {
      Logger.log("Verified: Fuel Type " + fuelType + " -> Correlation ID: " + storedCorrelationIDs[fuelType]);
    } else {
      Logger.log("Missing Correlation ID for Fuel Type " + fuelType + ". Check request logs.");
    }
  });

  Logger.log("All fuel types requested. Ready for polling.");
}
```

```
function sendSubmissionRequest(fuelType) {
  var url = "https://cge.cyprus.gov.cy/gg/submission";
  //API Payload is listed bellow, and is hidden for API Key security. The format can be fouond on gov.cy
  var xmlRequest = `<?xml version="1.0" encoding="utf-8"?>
  <GovTalkMessage xmlns="http://www.govtalk.gov.uk/CM/envelope">⋯
  </GovTalkMessage>`;

  var options = {
    method: "post",
    contentType: "application/xml",
    payload: xmlRequest
  };

  var response = UrlFetchApp.fetch(url, options);
  var responseText = response.getContentText();

  // Extract CorrelationID
  var correlationIdMatch = responseText.match(/<CorrelationID>(.*?)<\/CorrelationID>/);
  if (correlationIdMatch) {
    Logger.log(`Fuel Type ${fuelType}: Correlation ID = ${correlationIdMatch[1]}`);
    return correlationIdMatch[1]; // Return correlation ID
  }

  Logger.log(`Failed to get Correlation ID for Fuel Type ${fuelType}`);
  return null;
}
```

```
function pollForFuelPrices(correlationID, fuelType) {
  var url = "https://cge.cyprus.gov.cy/gg/poll";
//Bellow is another API Payload. Once again hidden for security reasons
  var xmlPollRequest = `<?xml version="1.0" encoding="utf-8"?>
  <GovTalkMessage xmlns="http://www.govtalk.gov.uk/CM/envelope">⋯
  </GovTalkMessage>`;

  var options = {
    method: "post",
    contentType: "text/xml", // Try changing from "application/xml" to "text/xml"
    payload: xmlPollRequest
  };

  try {
    var response = UrlFetchApp.fetch(url, options);
    var responseText = response.getContentText();

    // Log the full response for debugging
    Logger.log(`Full API Response for Fuel Type ${fuelType}:`);
    Logger.log(responseText);

    if (responseText.includes("<Qualifier>response</Qualifier>")) {
      Logger.log(` Fuel price data ready for Fuel Type ${fuelType}`);

      // Create or get sheet
      var sheetName = `Raw_Fuel_Data_${fuelType}`;
      var sheet = SpreadsheetApp.getActiveSpreadsheet().getSheetByName(sheetName);

      if (!sheet) {
        sheet = SpreadsheetApp.getActiveSpreadsheet().insertSheet(sheetName);
      }

      sheet.clear();
      sheet.appendRow(["Fuel Price API Response"]);
      sheet.appendRow([responseText]);

      Logger.log(`Response saved in '${sheetName}'`);

      saveDataToSheet(responseText, fuelType);
      return true;
    } else {
      Logger.log(`Data NOT ready for Fuel Type ${fuelType}. Retrying later...`);
      return false;
```

```
function saveDataToSheet(xmlData, fuelType) {
  if (!xmlData) {
    Logger.log(`No XML data found for Fuel Type ${fuelType}. Skipping...`);
    return;
  }

  // Get the correct raw data sheet name
  var sheetName = `Raw_Fuel_Data_${fuelType}`;
  var sheet = SpreadsheetApp.getActiveSpreadsheet().getSheetByName(sheetName);

  if (!sheet) {
    Logger.log(`No raw data found for Fuel Type ${fuelType}. Run pollForFuelPrices() first.`);
    return;
  }

  //Get the correct structured data sheet name
  var fuelSheetName = `Fuel_Prices_${fuelType}`;
  var fuelSheet = SpreadsheetApp.getActiveSpreadsheet().getSheetByName(fuelSheetName);

  if (!fuelSheet) {
    fuelSheet = SpreadsheetApp.getActiveSpreadsheet().insertSheet(fuelSheetName);
    fuelSheet.appendRow(["Fuel Company", "Station Name", "City", "District", "Address", "Fuel Price", "Date Modified", "Coordinates"]);
  } else {
    fuelSheet.clear();
    fuelSheet.appendRow(["Fuel Company", "Station Name", "City", "District", "Address", "Fuel Price", "Date Modified", "Coordinates"]);
  }

  //Extract station data using regex
  var stations = xmlData.match(/<PetroleumPriceDetails1>[\s\S]*?<\/PetroleumPriceDetails1>/g);
  if (!stations) {
    Logger.log(`No station data found for Fuel Type ${fuelType}`);
    return;
  }
  //Process extracted stations
  var extractedData = stations.map(station => [
    extractXMLTag(station, "fuel_company_name"),
    extractXMLTag(station, "station_name"),
    extractXMLTag(station, "station_city"),
    extractXMLTag(station, "station_district"),
    extractXMLTag(station, "station_address1"),
    extractXMLTag(station, "Fuel_Price"),
    extractXMLTag(station, "price_modification_date"),
    extractXMLTag(station, "map_coordinates")
  ]);

  //Batch insert instead of appending row by row (faster)
  fuelSheet.getRange(fuelSheet.getLastRow() + 1, 1, extractedData.length, extractedData[0].length).setValues(extractedData);

  Logger.log(`Fuel prices saved for Fuel Type ${fuelType}!`);
}
```

```
function convertCoordinates() {
  var ss = SpreadsheetApp.getActiveSpreadsheet();

  // Loop through all fuel price sheets (1-5)
  for (var i = 1; i <= 5; i++) {
    var sheet = ss.getSheetByName("Fuel_Prices_" + i);
    if (!sheet) continue;

    var data = sheet.getDataRange().getValues();
    var updatedData = [];

    // Iterate over each row (skip header)
    for (var r = 1; r < data.length; r++) {
      var rawCoord = data[r][7]; // Assuming coordinates are in the 8th column (index 7)

      if (rawCoord && isDMSFormat(rawCoord)) {
        var convertedCoord = dmsToDecimal(rawCoord);
        data[r][7] = convertedCoord; // Replace with converted value
      }

      updatedData.push(data[r]);
    }

    // Write back updated data
    if (updatedData.length > 0) {
      sheet.getRange(2, 1, updatedData.length, updatedData[0].length).setValues(updatedData);
    }
  }

  Logger.log("Coordinate conversion completed.");
}

// Check if the coordinate is in Degrees, Minutes, Seconds (DMS) format
function isDMSFormat(coord) {
  return /°|'|"/.test(coord);
}
```

# Appendix B

Code for UCY Maintenance implementation. (Process.gs)

```
function processMaintenanceReports() {
    const ss = SpreadsheetApp.getActiveSpreadsheet();
    const reportSheet = ss.getSheetByName('ReportLog');
    const aiLogSheet = ss.getSheetByName('AICalls');
    const onCampusSheet = ss.getSheetByName('OnCampus');
    const serverSheet = ss.getSheetByName('Server');

    const lastRow = reportSheet.getLastRow();
    Logger.log("Starting processMaintenanceReports");

    for (let row = 2; row <= lastRow; row++) {
        const verified = reportSheet.getRange(row, 6).getValue(); // Column F = Verified
        const description = reportSheet.getRange(row, 4).getValue(); // Column D = Description

        if (verified === false && description) {
            Logger.log(`Processing row ${row}: ${description}`);

            // Call the AI classification function
            const aiResponse = classifyMaintenanceIssue(description);
            aiLogSheet.appendRow([new Date(), description, aiResponse || "API failed"]);
            Logger.log(`AI Response for row ${row}: ${aiResponse}`);

            if (aiResponse) {
                const lowerResponse = aiResponse.toLowerCase();
                let category = null;
                let priority = null;

                // Extract classification
                if (lowerResponse.includes("oncampus")) category = "onCampus";
                else if (lowerResponse.includes("server")) category = "server issue";

                // Extract priority (P1–P5)
                const priorityMatch = lowerResponse.match(/p[1-5]/i);
                if (priorityMatch) priority = priorityMatch[0].toUpperCase();

                if (category && priority) {
                    // Get full row (without Verified column)
                    const fullRow = reportSheet.getRange(row, 1, 1, reportSheet.getLastColumn()).getValues()[0];

                    const [user, name, latlong, desc, photo, , timestamp] = fullRow;

                    const entry = [
                        user,
                        name,
                        latlong,
                        desc,
                        photo,
                        priority,
                        "Unresolved", // Status
                        timestamp,
                        "", // Resolved timestamp (blank)
                        ""  // Assigned (blank)
                    ];

                    if (category === "onCampus") {
                        onCampusSheet.appendRow(entry);
                    } else if (category === "server issue") {
                        serverSheet.appendRow(entry);
                    }

                    // Mark ReportLog entry as verified
                    reportSheet.getRange(row, 6).setValue(true); // Column F = Verified
                } else {
                    Logger.log(`Invalid data for row ${row}: category or priority missing`);
                    reportSheet.getRange(row, 6).setValue(false); // AI responded, but data invalid
                }
            } else {
                Logger.log(`API failed for row ${row}`);
                reportSheet.getRange(row, 6).setValue(false); // API failed
            }
        }
    }
    Logger.log("Finished processMaintenanceReports");
}
```

```
// Function to classify maintenance issues using the provided API
function classifyMaintenanceIssue(description) {
    const apiKey = '[redacted]';
    const url = 'https://api.groq.com/openai/v1/chat/completions';

    const messages = [
  {
    role: "system",
    content:
      "You are an AI that classifies user-submitted maintenance issues. You must output only the classification (onCampus or server issue) and the priority level (P1 to P5) based on urgency.\n\n" +
      "Classification rules:\n" +
      "- If the issue concerns physical infrastructure like classrooms, bathrooms, lighting, A/C, electricity, etc., classify it as `onCampus`.\n" +
      "- If the issue concerns digital systems like WiFi, internet access, university websites, or internal servers, classify it as `server issue`.\n\n" +
      "Priority guidelines:\n" +
      "- P1 = Critical: Urgent and blocks operations or safety (e.g., broken main door, total WiFi outage)\n" +
      "- P2 = High: Major disruption to users (e.g., multiple classrooms without lights)\n" +
      "- P3 = Medium: Noticeable issue but not urgent (e.g., flickering light in one room)\n" +
      "- P4 = Low: Minor issue, can be delayed (e.g., loose door handle)\n" +
      "- P5 = Informational or cosmetic only (e.g., scuff marks, minor noise)\n\n" +
      "Return only this exact format:\n" +
      "Category: onCampus or server issue\nPriority: P1/P2/P3/P4/P5\n\n" +
      "Do not return anything else or explanations. Only return the classification and priority in this format."
  },
  {
    role: "user",
    content: description
  }
];


    const payload = {
        model: 'llama3-8b-8192',
        messages: messages
    };

    const options = {
        method: 'POST',
        contentType: 'application/json',
        headers: {
            'Authorization': `Bearer ${apiKey}`
        },
        payload: JSON.stringify(payload),
        muteHttpExceptions: true
    };

    try {
        const response = UrlFetchApp.fetch(url, options);
        Logger.log("API call successful");
        const responseText = response.getContentText();
        Logger.log("AI raw response: " + responseText); // ✅ Log the full raw response

        const json = JSON.parse(responseText);
        return json.choices?.[0]?.message?.content?.trim() || null;

    } catch (error) {
        Logger.log("API error: " + error);
        return null;
    }
}
```

# Appendix C

Code for UCY Maintenance (AnalyzeData.gs)

```
function generateWeeklyMaintenanceReport() {
  const ss = SpreadsheetApp.getActiveSpreadsheet();
  const onCampusSheet = ss.getSheetByName('OnCampus');
  const serverSheet = ss.getSheetByName('Server');
  const reportSheet = ss.getSheetByName('WeeklyReport') || ss.insertSheet('WeeklyReport');

  const oneWeekAgo = new Date();
  oneWeekAgo.setDate(oneWeekAgo.getDate() - 7);

  const data = [...getFilteredData(onCampusSheet, oneWeekAgo), ...getFilteredData(serverSheet, oneWeekAgo)];
  if (data.length === 0) {
    Logger.log("No data to report.");
    return;
  }

  const summaryStats = analyzeData(data);
  const aiPrompt = buildPrompt(summaryStats);
  const aiResponse = callReportAI(aiPrompt);

  if (aiResponse) {
    reportSheet.appendRow([new Date(), aiResponse]);
  } else {
    Logger.log("Failed to generate AI report.");
  }
}
```

```
function getFilteredData(sheet, sinceDate) {
  const allRows = sheet.getDataRange().getValues();
  const headers = allRows[0];
  const timestampIdx = headers.indexOf("Timestamp");
  const statusIdx = headers.indexOf("Status");
  const priorityIdx = headers.indexOf("Priority");
  const descIdx = headers.indexOf("Description");

  const results = [];
  for (let i = 1; i < allRows.length; i++) {
    const row = allRows[i];
    const timestamp = new Date(row[timestampIdx]);
    const status = row[statusIdx];

    if (timestamp >= sinceDate || status.toLowerCase() === "unresolved") {
      results.push({
        timestamp,
        status,
        priority: row[priorityIdx],
        description: row[descIdx]
      });
    }
  }
  return results;
}
```

```
function analyzeData(data) {
  const stats = {
    total: data.length,
    resolved: 0,
    unresolved: 0,
    priorityCount: { P1: 0, P2: 0, P3: 0, P4: 0, P5: 0 },
    recentDescriptions: []
  };

  for (const item of data) {
    if (item.status.toLowerCase() === "resolved") {
      stats.resolved++;
    } else {
      stats.unresolved++;
    }

    const pr = item.priority.toUpperCase();
    if (stats.priorityCount[pr] !== undefined) {
      stats.priorityCount[pr]++;
    }

    stats.recentDescriptions.push(`- [${item.priority}] ${item.description}`);
  }

  return stats;
}
```

```
function buildPrompt(stats) {
  const { total, resolved, unresolved, priorityCount, recentDescriptions } = stats;
  return `You are an AI that generates weekly maintenance reports for a university. Based on the following data, write a concise summary for stakeholders. Prioritize clarity and actionable insight.

Date Range: Last 7 days

Total Reports: ${total}
Resolved: ${resolved}
Unresolved: ${unresolved}

Priority Breakdown:
P1: ${priorityCount.P1}
P2: ${priorityCount.P2}
P3: ${priorityCount.P3}
P4: ${priorityCount.P4}
P5: ${priorityCount.P5}

Recent Issues:
${recentDescriptions.slice(0, 10).join("\n")}

Generate a short professional report summarizing trends, urgent concerns, and patterns.`;
}
function callReportAI(prompt) {
  const apiKey = '[redacted]';
  const url = 'https://api.groq.com/openai/v1/chat/completions';

  const payload = {
    model: 'llama3-8b-8192',
    messages: [
      { role: "system", content: "You are a professional maintenance report generator AI." },
      { role: "user", content: prompt }
    ]
  };

  const options = {
    method: 'POST',
    contentType: 'application/json',
    headers: {
      'Authorization': `Bearer ${apiKey}`
    },
    payload: JSON.stringify(payload),
    muteHttpExceptions: true
  };

  try {
    const response = UrlFetchApp.fetch(url, options);
    const json = JSON.parse(response.getContentText());
    const result = json.choices?.[0]?.message?.content?.trim();
    Logger.log("Weekly Report AI Response: " + result);
    return result;
  } catch (e) {
    Logger.log("AI report generation failed: " + e);
    return null;
  }
}
```

# Appendix D

Code for the backend of LogFile (Auditing)

```
function auditRecentLogs() {
  const reportLogSheetId = "1ui-z89MhQalfMKEMOl3ETPE9yMtNrESN2uLKAQ8lu5w";
  const reportLogSheet = SpreadsheetApp.openById(reportLogSheetId).getSheetByName("ReportLog");
  const reportLogData = reportLogSheet.getDataRange().getValues();
  const reportLogHeaders = reportLogData[0];
  const reportEntryKeys = new Set(
    reportLogData.slice(1).map(row => row[0] + normalizeTimestamp(row[6])) // user + normalized timestamp
  );

  const now = new Date();
  const oneHourAgo = new Date(now.getTime() - 60 * 60 * 1000);
  const activeSS = SpreadsheetApp.getActiveSpreadsheet();
  const userSheets = activeSS.getSheets().filter(s => s.getName() !== "Discrepancies" && s.getName() !== "Control");

  Logger.log(`Checking ${userSheets.length} user sheets...`);

  let allEntries = [];

  // 1. Gather all recent logs from all sheets
  userSheets.forEach(sheet => {
    const data = sheet.getDataRange().getValues();
    const headers = data[0];
    const idx = {
      user: headers.indexOf("User"),
      name: headers.indexOf("Name"),
      latlong: headers.indexOf("LatLong"),
      description: headers.indexOf("Description"),
      photo: headers.indexOf("Photo"),
      timestamp: headers.indexOf("Timestamp")
    };

    for (let i = 1; i < data.length; i++) {
      const rawTimestamp = data[i][idx.timestamp];
      const parsedDate = parseFlexibleTimestamp(rawTimestamp);
      if (!parsedDate || parsedDate < oneHourAgo) continue;
```

```
      allEntries.push({
        sheetName: sheet.getName(),
        data: data[i],
        timestamp: parsedDate,
        user: data[i][idx.user],
        name: data[i][idx.name],
        latlong: data[i][idx.latlong],
        description: data[i][idx.description],
        photo: data[i][idx.photo],
        timestampStr: rawTimestamp
      });
    }
  });

  Logger.log(`Found ${allEntries.length} total recent logs.`);

  // 2. Compare entries for timestamp similarity
  for (let i = 0; i < allEntries.length; i++) {
    for (let j = i + 1; j < allEntries.length; j++) {
      const diff = Math.abs(allEntries[i].timestamp - allEntries[j].timestamp);
      if (diff <= 10000) { // 5 seconds
        const a = allEntries[i];
        const b = allEntries[j];
        Logger.log(`Similar entries detected:\n- ${a.user} @ ${a.timestampStr}\n- ${b.user} @ ${b.timestampStr}`);

        [a, b].forEach(entry => {
          const key = entry.user + normalizeTimestamp(entry.timestampStr);
          if (!reportEntryKeys.has(key)) {
            Logger.log(`→ Inserting missing entry for ${entry.user} at ${entry.timestampStr}`);
            reportLogSheet.appendRow([
              entry.user,
              entry.name,
              entry.latlong,
              entry.description,
              entry.photo,
              false,
              entry.timestampStr
            ]);
          }
        });
      }
    }
  }

  Logger.log("Audit complete.");
}
```

```
// Normalize timestamp to a consistent format key
function normalizeTimestamp(input) {
  const date = parseFlexibleTimestamp(input);
  return date ? date.toISOString() : "";
}

// Parses timestamps in various known formats
function parseFlexibleTimestamp(str) {
  if (str instanceof Date) return str;
  if (!str || typeof str !== "string") return null;

  try {
    // Try standard parsing first
    const d = new Date(str);
    if (!isNaN(d)) return d;

    // Try format: DD/MM/YYYY HH:MM:SS
    const [datePart, timePart] = str.split(" ");
    const [day, month, year] = datePart.split("/").map(Number);
    const [hours, minutes, seconds] = timePart.split(":").map(Number);
    return new Date(year, month - 1, day, hours, minutes, seconds);
  } catch (e) {
    Logger.log(`Timestamp parse failed: ${str}`);
    return null;
  }
}
```