CS
COC251
F221332

AI-Powered Mobile App for Formula 1 Qualifying and Strategy Predictions

by

Daniel Lawrence

Supervisor: Dr. Hui Fang

Department of Computer Science
Loughborough University

May/June 2025

Abstract

Mobile technology, especially artificial intelligence, is quickly changing how we process and use data across nearly all industries. Formula 1 is a highly competitive and technology-focused sport, creating an ideal case study for predictive analytics applications.

This project details designing, developing, and evaluating an Android mobile application that predicts Formula 1 qualifying lap times and various race strategies using machine learning techniques. The application utilises regression models to forecast lap times based on track selection and weather conditions, and generates optimal and alternate race strategies with associated confidence scores based on environmental inputs.

Requirements were initially gathered through my knowledge of Formula 1, informal feedback, and discussions with close friends who had a strong interest in motorsport. This helped me identify the key features and user needs for this application. Development focused heavily on performance, model accuracy, and user-friendly designs, ensuring the app delivered near-realistic predictions for the race environments.

Challenges included limited access to official Formula 1 telemetry data, necessitating the use of publicly available APIS and the development of custom data preprocessing pipelines. Additionally, variability in race conditions, the influx of rookie drivers during the 2025 season, and early driver changes such as car swaps impacted model stability and generalisation, presenting further challenges during model training.

Testing demonstrated that the qualifying lap time model could predict the lap times and grid positions for all 20 drivers with an average accuracy within 1% of real-world data. Quantitative evaluation showed that the model achieved an RMSE score of 0.275 seconds, a MAE of 0.230 seconds and an R^2 score of 0.899, showing a strong predictive performance. The race strategy model successfully generated optimal and alternate pit stop strategies, providing associated confidence scores based on track and weather inputs. These results show the ability of artificial intelligence and how it can enhance the world of motorsport, with future improvements such as integration with live race data.

Acknowledgements

Completing this project would not have been possible without the expertise and support of my supervisor, Dr. Hui Fang. I would like to thank him for his help over the past year.

I would also like to thank the Computer Science department at Loughborough University for providing the academic foundation and resources that have been so important to me throughout my studies.

Finally, I want to thank my friends and family for their encouragement and support during this project and my academic experience. Their motivation helped me persevere through the many challenges I faced during development.

Contents

1 Introduction

1.1 Motivation

Formula 1 is the pinnacle of motorsport, combining the latest technology, near-perfect engineering, and human skill, and to succeed in Formula 1, you must excel in each aspect. The success in Formula 1 is often determined by ultra-thin margins, with qualifying sessions and race strategies decided by differences of mere hundredths of a second. The sport is forever evolving, as has the role of data analytics, with teams now gathering and processing vast amounts of telemetry and environmental data to optimise their car and gain every competitive advantage.

However, the ability to accurately predict qualifying lap times and optimal race strategies remains reserved mainly for organisations with significant resources, proprietary data access, and dedicated analytics departments. Real-time predictions are extremely challenging due to the complexity of integrating track conditions, weather forecasts, car setups, and driver performance.

Artificial intelligence is forever evolving, especially machine learning regression models, which will improve a team's performance as they get smarter. AI systems can learn hidden patterns within large datasets and provide predictive insights with minimal human intervention, offering a fast, but more importantly, scalable method for forecasting race outcomes.

This project explores whether an Android mobile application, powered by machine learning models, can accurately predict Formula 1 qualifying lap times and recommend effective race strategies based on track and weather inputs. Doing so aims to demonstrate the potential of making high-quality predictive analytics accessible from elite motorsport teams and contribute to the ever-growing combination of AI and sports technology.

1.2 Project Aims

This project aims to design, develop, and evaluate an Android mobile application that uses machine learning techniques to predict Formula 1 qualifying lap times and generate race strategies based on user-selected track and weather inputs.

The qualifying prediction system will allow users to select a specific track. After that, the model will output a full grid of twenty drivers, displaying each driver's predicted qualifying time, team name, and starting position. Users can also click on any driver to expand a detailed view, revealing sector times, speed trap performance, average braking force, throttle usage, and RMP metrics. This will provide a detailed breakdown of each qualifying session simulation, offering fascinating insights beyond just lap times and predictions.

The race strategy system will enable users to input key environmental variables such as air temperature, track temperature, and rainfall conditions. Based on these inputs, the system will generate the optimal pit stop strategy for each grid position. Users can expand any result to view three alternate strategies tailored for that position, providing flexibility depending on race conditions and potential events like safety cars or rain.

The project aims to show the feasibility of using machine learning models—specifically regression algorithms—to provide realistic insights that are usually only available to professional race teams and their engineers. The final deliverable will be a user-friendly mobile application that allows users to simulate custom race scenarios, analyse strategies, and better understand the complexity behind Formula 1 performance and race planning.

1.3 Objectives

1.3.1 Main Objectives

1. Research machine learning techniques for time prediction and strategic decision-making in motorsport applications.
2. Develop a regression-based qualifying model that predicts the lap times and starting grid positions for all twenty Formula 1 drivers based on track selection.
3. Implement an expandable driver interface that allows users to view detailed telemetry data for each driver/grid position, including sector times, etc.
4. Develop a race strategy generation system that predicts the optimal pit stop strategy for each grid position based on air temperature, track temperature, and rainfall conditions.
5. Implement an expandable strategy interface that shows three alternate strategies per grid position and confidence scores for each recommendation.
6. Develop an Android mobile application with a user-friendly interface that allows users to use the qualifying and strategy prediction models.
7. Evaluate predictive model performance using metrics such as Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE).
8. Carry out extensive testing to verify system accuracy, user experience quality, mobile application stability, and overall reliability.

1.3.2 Secondary Objectives

1. Implement a secure login and registration system using Firebase Authentication to manage user accounts within the mobile application.
2. Develop backend database functionality using Firebase Firestore to manage user data and preferences.
3. Integrate 3D model visualisations to simulate Formula 1 car models to provide an interactive graphical interface
4. Integrate live API data to dynamically pull in and display current Formula 1 driver and constructor standings, and display them within the application
5. Create a scalable architecture within the mobile application, allowing for future expansions
6. Implement professional UI/UX design principles, allowing the app to be visually appealing, responsive, and intuitive across Android devices.
7. Clear documentation of challenges encountered during development, including data collecting limitations, modal generalisation difficulties across tracks, and performance optimisation on mobile platforms.

2 Literature Review

Finding detailed public research on Formula 1 performance prediction is difficult because of the sport's highly secretive and competitive nature. Teams treat their telemetry data, car setups, and race strategies as confidential, making it very unlikely for any successful prediction model to be shared outside their organisation. As a result, limited academic work focuses specifically on qualifying lap time prediction or race strategy simulations. To overcome this, the literature review will look at research in related areas, including AI and data analytics in motorsport, machine learning for time predictions, and race strategy optimisation. The combination of knowledge from these areas provides a strong foundation for the design and development of the application.

2.1 Artificial Intelligence and Data Analytics in Motorsport

Formula 1 teams increasingly rely on artificial intelligence and machine learning to enhance every aspect of performance, from car design to real-time race decisions. AI systems analyse millions of data points generated by sensors all over the cars, including tyre wear, engine temperature, fuel consumption and aerodynamic efficiency. Tools such as predictive maintenance systems help detect issues before they cause failures. In addition, AI models assist in simulating different race strategies under various conditions, helping teams prepare for countless potential scenarios before race day. According to Forbes (Mar 2023), teams like Mercedes and Red Bull Racing now employ hundreds of data scientists and AI engineers to keep a competitive advantage.

Every decision made during a race weekend, from car setup changes to pit stop calls, is backed by a complex data analytics layer. Real-time telemetry allows teams to compare their cars' performance against their laps and competitors. Based on historical data, AI algorithms help spot performance anomalies, suggest optimal tyre changes, simulate the perfect lap time, and predict the likelihood of on-track incidents like safety cars. Nowadays, modern Formula 1 teams run millions of simulations during race weekends to find the statistically most likely winning strategy, demonstrating how central data-driven decision making has become.

Although technology is evolving rapidly, access to detailed Formula 1 data is extremely limited to the public and academic researchers. Teams treat telemetry and strategic data as confidential intellectual property. Licensing agreements, sponsorship contracts, and the need to protect their competitive advantage mean that much of the raw data remains inaccessible. Most public data sources, such as APIS offering timing data, lack the depth of data needed for professional-grade predictive modelling. Because of this, public projects and researchers often need to make assumptions, rely on simplified datasets, or simulate scenarios using estimated parameters.

2.2 Machine Learning Techniques for Predicting Lap Times

Predicting lap times in Formula 1 is a continuous output problem; therefore, using regression models makes the most sense. The goal is not to predict a category or a label, but to expect a continuous number – the time it takes a driver to complete a lap. One of the simplest ways to approach this is through Linear Regression, where the model tries to fit a straight line between the input features and the output time. Although simple, it can work well if the data has a strong linear relationship, but Formula 1 (and motorsport) is rarely that simple.

More complex models like Decision Trees and Random Forests can better handle non-linear relationships. Decision trees split the data into small chunks based on feature thresholds, allowing the model to capture the non-linear effect of things like tyre compound or track temperature. Random Forest takes this idea further by building many trees and averaging their outputs, which tends to improve accuracy and prevent overfitting.

Finally, Neural Networks are also an option, especially when the relationships between features and lap time are very complex. A well-designed Neural Network can spot hidden patterns across multiple factors, such as how a tyre interacts with air temperature and fuel load. However, Neural Networks need more data and fine-tuning to perform well compared to trees or linear models. Simpler models like regression trees are often preferred for mobile applications as they balance speed and model accuracy, which is crucial for running on mobile devices without enormous computing power.

The best predictions start with good input data. In motorsport, raw telemetry alone isn't enough. The model needs clear and relevant features that impact lap times. Feature engineering transforms this raw data into meaningful features that machine learning models can use to make predictions.

The features for predicting qualifying laps must capture environmental factors and car/driver performance. Key inputs could include the track selected, tyre compound choice (e.g. soft), sector times, and speed traps. During these qualifying sessions, air temperature, humidity, and rainfall play a massive role in lap variability for weather-based features.

Normalisation is also imperative. Features like speed traps and sector times exist on very different scales. Therefore, normalising the inputs ensures that no feature unfairly dominates the learning process. Care also needs to be taken to avoid leaking future information into the training process—for example, using race day information when predicting qualifying results would create data leakage and artificially improve model scores.

Adding useful synthetic features also helps. For example, things like average sector speed (sector time/sector length) and changes in tyre life across stints were introduced to add predictive power to the model without needing access to the team-level confidential data.

Once a model is trained, it is essential to know how it performs. In Formula 1, tiny differences in lap time can mean a massive difference in qualifying position, so the evaluation metrics must be chosen carefully.

Root Mean Square Error (RMSE) is one of the best metrics for lap time prediction because it penalises larger mistakes more than smaller ones. If a model predicts one driver to be 1.5 seconds off, RMSE will reflect that heavily, which is more critical as significant errors are much worse in this context. A model that predicts within a few tenths on average would be considered good Formula 1 standards.

Mean Absolute Error (MAE) is also helpful as it gives a simple, easy-to-understand measure of average error. Unlike RMSE, it doesn't punish big mistakes as harshly, but it's still a good way to summarise how close the model is.

R^2 (R-squared score) shows how much of the variation in actual lap times can be explained by the model's predictions. A perfect model would have a R^2 score of 1.0, but in motorsport, where unpredictable factors always exist, a strong R2 score will realistically be between 0.85 and 0.95, depending on the track and data quality. These three metrics together provide a complete picture of model performance, helping make realistic decisions whether the predictions are reliable enough for real-world usage.'

2.3 Race Strategy Prediction and Optimisation

Regarding race strategy in Formula 1, many factors need to be considered, and they are constantly changing. Tyre degradation is one of the biggest ones. Every compound wears at a different rate depending on the track surface, weather conditions, and how aggressively the driver pushes on the track. On top of that, track temperature plays a massive role in tyre performance, with hotter tracks leading to faster tyre wear and colder tracks reducing grip levels.

Another critical factor is pit lane loss – the time a driver loses when pitting. On some tracks like Italy (Monza), the pit lane is very short, making pit stops cheaper regarding race time. In other places like Singapore, the pit lane loss is much higher, forcing teams to rethink their race strategy completely. Safety car deployments and Virtual Safety Cars (VSCs) are also huge factors because they can change the time penalty of a pit stop in real time, and teams need to react instantly.

The weather is also unpredictable. A sudden wet track can change the best-laid plans, forcing a driver to switch from dry tyres to intermediates or full wets. Race strategy isn't just about choosing the fastest plan at the start—it's about adapting when the race throws surprises.

In past Grand Prixs, race strategies were mainly based on experience and manual simulation work, but that has changed quickly with AI and significant data advancements. Teams now run hundreds of thousands of race simulations before and during race weekends to find the optimal strategies. This includes Monte Carlo simulations, which run thousands of race scenarios based on small random changes in tyre wear rates, pit stops, and lap times.

Optimisation algorithms are also used to plan the best pit stop windows. Linear programming, for example, can balance the back-and-forth fight between pit stop losses and tyre degradation performance over a race distance. Some more experimental research uses Reinforcement Learning, where an AI agent learns by running simulated races repeatedly, constantly tweaking pit stop timings and tyre choices to find better strategies.

Complete reinforcement learning isn't practical in mobile and lightweight predictive systems because of the heavy computation needed. Instead, heuristic-based models—more straightforward rule-based logic with predictive outputs—are used. These methods still allow flexibility (e.g., showing multiple alternate strategies) without needing massive servers running thousands of simulations live.

Predicting race strategies in real time is one of the hardest things to do because the situation on track changes so fast. For example, even if there is a single crash, bringing out a safety car can destroy the "best" strategies. Rain can arrive unexpectedly and force teams to pit immediately for different tyres. Drivers can push too hard and burn out tyres faster than model predictions.

Because of this, any real-time strategy AI needs to be accurate and flexible. It's more important for the model to give a range of good options (primary strategies plus alternatives) rather than locking into a single plan. Real-world strategy engineers in Formula 1 constantly update their models every few laps based on new data (tyre temperatures, weather radars, track evolution, etc.).

2.4 Mobile Application Development for Predictive Systems

When developing any mobile application, the first thing that matters is keeping the user experience simple, straightforward, and responsive. In predictive mobile applications, where users request large amounts of data, they need quick answers with minimal confusion. Ensuring fast response times and clean UI flows is critical for keeping users engaged.

Responsiveness is key because predictive models can take up much of the main thread, introducing minor delays when running calculations if not dealt with properly. Minimising load times and using clear loading indicators if necessary helps keep users engaged. Another critical factor is how the results are presented. Results need to be explained clearly. Instead of overwhelming users with raw numbers, it's best to use structured layouts like cards, tables, or expandable views so the users can easily dive into detail if they want to.

Accessibility across devices is also essential. Predictive apps must perform smoothly on high-end phones and older devices, meaning you must be careful when using a lot of memory.

Android Studio is the standard platform for developing modern Android applications. It provides all the tools to design interfaces, manage backend integrations, and integrate machine learning models. This software is widely used due to its strong support for native Android development and newer libraries like Jetpack Compose.

TensorFlow Lite is a standard library for handling machine learning models on the device. It allows trained models to run well on mobile hardware, which is essential for keeping prediction response times low. This is especially important for predictive applications where real-time responses are expected.

Firebase services, including Firebase Authentication and Firestore, are also widely adopted in mobile application development. Firebase Authentication provides secure login systems that are easy to integrate, while Firestore offers scalable cloud databases for storing user preferences or application data. These services are usually chosen to reduce the time and complexity of setting up secure backend systems.

In sports applications, APIs are frequently connected to pull in live external data such as standings, race timings, or weather updates. Efficient API integration and clever caching techniques ensure that data is delivered quickly without overloading the device or network.

2.5 Real-Time Data Integration and User Interaction

In mobile applications that rely on predictive models or live data, integrating real-time information is essential for keeping the app useful and engaging. Especially in sports apps, users expect up-to-date standings, timings, and race information without delays. This requires pulling live data from external APIs and updating the app's interface without disturbing the user experience.

Real-time data integration typically involves API calls to services that provide the latest race data or standings. Handling these APIs properly is vital because network connections can be unpredictable. To avoid freezing the app when data is being fetched, background threads or asynchronous methods allow the app to continue running smoothly while waiting for external data.

Caching strategies also play a key role—apps often temporarily store API responses on the device to reduce repeated calls and offer fast offline access. For predictive systems, caching means users can still view recent predictions or standings even if their internet connection becomes unstable during use. In contrast, cache expiry must be appropriately managed, so users aren't shown outdated data for too long.

Another essential feature is efficient data refreshing. Some apps use manual refreshing (e.g., the pull-to-refresh gesture), while others fetch updated data automatically at regular intervals. When new data arrives, the app must update the display dynamically without requiring a full page reload. This helps maintain a clean and professional user experience.

User authentication systems like Firebase Authentication must also handle real-time session management. For example, suppose a user's authentication token expires while using the app. In that case, the system should handle reauthentication in the background or smoothly redirect the user to another page without crashing the app. Security must be maintained without compromising the most important thing: the user experience.

Building real-time interaction into predictive applications isn't just about fetching live data; it's about how the data is delivered, updated, and displayed to the user quickly, smoothly, and reliably.

2.6 Summary of Literature

This literature review has highlighted how artificial intelligence, data analytics, and machine learning are used across Formula 1 to improve the sport. It showed that while public access to high-quality telemetry and race data remains restricted, reliable predictions are still possible by focusing on key features such as sector times. Regression models, particularly decision trees and ensemble methods, have been identified as strong techniques for predicting qualifying laps due to their balance between accuracy and speed.

Regarding race strategy, the review explored how computational approaches like Monte Carlo simulations and optimisation algorithms allow teams to plan flexible race strategies, adapting to unpredictable race events like safety cars and changing weather. Real-time data integration techniques, including efficient API handling, caching, and user authentication, were also reviewed as essential for building mobile applications that deliver live updates and data without sacrificing responsiveness.

The research in these areas provides a strong technical foundation for designing and implementing a mobile application capable of accurately predicting Formula 1 qualifying performances and race strategies while maintaining a smooth and engaging user experience across different mobile devices.

3 Methodology and System Design

3.1 System Overview and Architecture

The system is designed to deliver Formula 1 qualifying and race strategy predictions using a native Android application that integrates machine learning models directly through embedded Python scripts. These models take in user-defined inputs such as track, weather, and tyre conditions and return predictions that are processed and displayed in the app. The application uses Chaquopy to run Python within the Android environment to support this, removing the need to fully convert models to another format. This approach ensured flexibility and scalability and made it easier to update/tune models without redeveloping core functionalities. Firebase is used for user authentication and cloud storage of user data, and the app connects to the Ergast API to pull real-time information such as drivers/constructors' standings. By combining this live data integration with machine learning execution and persistent cloud storage, the system creates a clean, data-driven experience optimised for mobile use.

3.1.1 Component Breakdown

The system architecture is divided into four main components: the mobile frontend, the embedded Python ML models, the backend cloud services, and live data integration via the Ergast API.

- Mobile Frontend: Built using Android Studio with Jetpack Compose after being designed in Figma, the mobile interface is lightweight and responsive. It features clean layouts, scrollable cards, expandable panels that allow users to interact with qualifying and strategy predictions, and high-quality and interactive 3D car models. Navigation is kept simple and intuitive, with user satisfaction the primary focus of the entire design.
- Machine Learning Models (via Chaquopy): All models were developed and trained in Python using Scikit-learn. Rather than exporting to TensorFlow Lite, these models are executed directly through embedded Python scripts using Chaquopy. This allows the core machine learning logic to remain in Python, which makes debugging and future changes more efficient. The qualifying predictions use regression-based modelling, while the strategy simulation includes logic-driven outputs with confidence scoring based on input conditions.
- Backend (Firebase Services): Firebase Authentication manages secure login and session handling. Firestore serves as the backend database, allowing the app to store information such as user preferences. This keeps the app lightweight while supporting user-specific data retrieval and storage in the cloud.
- Live Data Integration (Ergast API): The app fetches the live drivers' and constructors' standings using the Ergast API. The data is retrieved on app startup or refresh and cached locally to ensure offline availability. This integration ensures that the prediction context stays up to date with the current Formula 1 season.

3.1.2 System Diagram

The system is designed around a simple and intuitive user flow, ensuring users can access all core features with minimal steps. After logging in or registering through Firebase Authentication, the user is taken to the home screen, which displays real-time Formula 1 driver and constructor standings by pulling data from the Ergast API. From there, users can navigate to the simulations section, where three options are available: Qualifying Predictions, Race Strategy Predictions, and 3D Car Visualisation.

In the Qualifying module, users select a track, and the embedded Python model processes this input to return predicted lap times for all 20 drivers. Each driver entry can be expanded to view more detailed sector data and additional telemetry estimates.

In the Race Strategy module, users select a track and input weather parameters, and the system returns the best strategy for each driver across the grid. Each driver card can also be expanded to view alternate strategy options and their associated confidence levels.

The 3D Car Visualisation section allows users to select a Formula 1 team and view a high-quality, interactive 3D model of the team's current car alongside core team statistics.

Finally, users can securely log out via the settings section, ending their session cleanly through Firebase.

The diagram below outlines the system architecture, highlighting how the Android frontend interacts with the prediction engine, which connects to backend services.

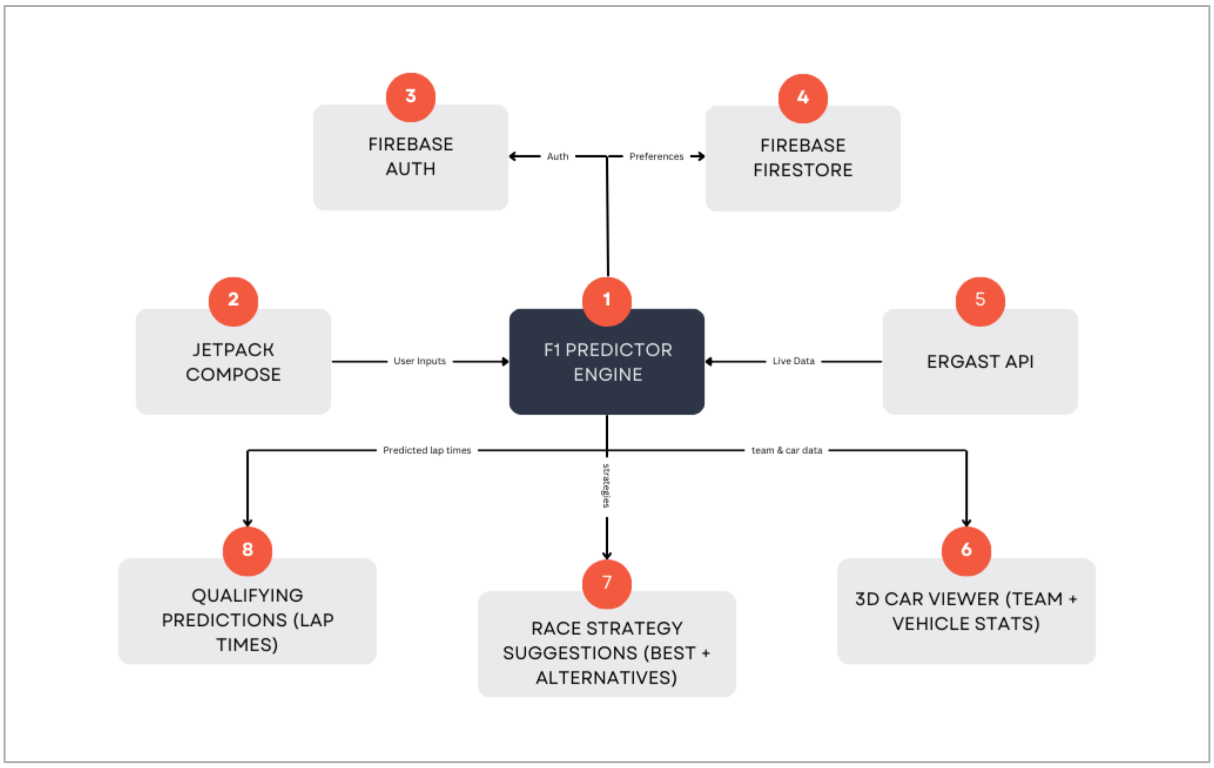


Figure 1: System Design Diagram

3.2 Qualifying Lap Time Prediction System

The qualifying lap time prediction system is designed to forecast the fastest possible lap times for all twenty drivers on a selected track. The system outputs a full qualifying grid using machine learning regression models trained on historical performance data and environmental values, providing detailed predictions for each driver. Telemetry data, such as sector times, throttle usage, and braking force, is also predicted, giving users more insights into the simulation.

3.2.1 Input Features

The input features selected were carefully chosen to capture the most critical factors affecting qualifying performance while still being lightweight enough for efficient on-device predictions. Table 1 outlines the core input features selected for the qualifying lap prediction time system:

Feature Name	Description
Track	The selected Formula 1 circuit.
Year	The season year to align the track and car evolution.
Driver ID	Unique identifier for each driver.
Team ID	Corresponding constructor associated with the driver.
Baseline Sector Times	Average sector times for each track based on historical pole laps.
Delta Sector Times	Differences between observed sector times and baseline sectors.
Average Sector Performance	Mean sector times for each team-year pairing.
Environmental Values	Including weather parameters such as air temperature, track temperature, rainfall intensity, wind speed, and humidity.
Tyre Compound	Type of tyre used during qualifying attempt (e.g. Soft).

Table 1: Input Features

These features were engineered to simulate driver- and track-specific performance variations while accounting for external conditions.

The historical qualifying data used for training were gathered from publicly available Formula 1 datasets. Due to limitations of complete telemetry data, sources such as Ergest and Fast F1 API were used to extract driver lap times and sector performances. Additional feature engineering was applied to simulate telemetry variables like throttle usage and braking force based on available track-specific and weather-adjusted assumptions. Despite the limited access to real team telemetry, this approach provided sufficient data diversity for building a realistic model.

3.2.2 Feature Engineering and Normalisation

Several feature engineering steps were performed to enhance the model's accuracy before training. The table below shows the key feature engineering techniques and data preprocessing steps to prepare the input dataset for model training:

Technique	**Description**
Baseline Correction	Sector time baselines were subtracted from observed data to calculate relative sector performance (delta times).
Team-Year Aggregation	Average sector performances were computed per team and year to factor in team competitiveness progression.
Synthetic Features	Derived variables such as minimum and maximum sector speeds, average throttle and brake pressures, and average RMPs were generated.
Handling Missing Data	Features with missing values were filled using column means to prevent bias and data leakage.
Normalisation	All numeric features were standardised using z-score normalisation (mean = 0, standard deviation = 1) via a StandardScaler to ensure model stability and prevent certain features from dominating during training.
Baseline Correction	Sector time baselines were subtracted from observed data to calculate relative sector performance (delta times).
Team-Year Aggregation	Average sector performances were computed per team and year to factor in team competitiveness progression.
Synthetic Features	Derived variables such as minimum and maximum sector speeds, average throttle and brake pressures, and average RMPs were generated.
Handling Missing Data	Features with missing values were filled using column means to prevent bias and data leakage.

Table 2: Summary of Feature Engineering and Preprocessing Techniques

This preprocessing ensured the model could generalise well across all scenarios without being skewed by feature scaling issues.

3.2.3 Model Design and Selection

Before the final model selection, multiple approaches, including linear regression and Decision Trees, were tested. Random Forest Regressors consistently outperformed these models regarding RMSE and R^2 scores during cross-validation, which is why that selection took place.

Given the complexity of lap time predictions and the need for mobile efficiency, an ensemble learning method was selected over neural networks. After benchmarking different approaches, a Random Forest Regressor was chosen for the main lap time prediction task, supported by Random Forest models for telemetry outputs.

The model was tuned through GridSearchCV, exploring hyperparameters such as:

- Number of estimators (100-1000 trees)
- Maximum tree depth (8-20)
- Minimum samples per split and leaf
- Bootstrap aggregation settings

Cross-validation ensured robustness against overfitting, and additional models were trained for telemetry metrics like sector speed and throttle usage to provide users with detailed qualifying breakdowns.

The final models were stored as .joblib files and loaded dynamically through Chaquopy at runtime within the mobile app.

3.2.4 Model Evaluation

The trained Random Forest models were evaluated using real qualifying data from 2025, achieving strong predictive performance. Table 3 presents the evaluation metrics achieved by the trained lap time prediction model, highlighting the system's strong predictive performance:

Metric	**Value**
Root Mean Squared Error (RMSE)	0.275 seconds
Mean Absolute Error (MAE)	0.230 seconds
R^2 Score	0.899

Table 3: Evaluation Metrics for the Qualifying Lap Time Prediction Model

Predictions for lap times were consistently within 1% of real-world observed data across multiple tracks and weather conditions. The driver ordering by lap times (grid positions) was accurate in most simulations, closely replicating real qualifying sessions.

These results demonstrate that the model effectively captures the essential dynamics of qualifying sessions without access to whole team telemetry data.

3.2.5 On-Device Inference

The system uses Chaquopy to execute Python scripts directly without the Android application. The Random Forest models and preprocessing scalers are bundled with the app and dynamically loaded during prediction requests.

This allows:

- Flexibility for future model updates without format conversion.
- Near real-time inference speeds, with full qualifying grid predictions in less than 1-2 seconds in mid-range mobile devices.
- Consistency between desktop model training and mobile inference outputs.

By executing predictions locally, the app avoids network dependency on core functionality, ensures predictions are delivered, and maintains a responsive user experience.

3.3 Race Strategy Prediction System

The race strategy prediction system recommends optimal and alternate pit stop strategies for each driver based on a selected track and environmental conditions. Using a combination of machine learning classification models and engineered logic, the system generates a variety of race strategies that adapt to varying features such as track and tyre characteristics. Each suggested that a confidence score accompany the plan to give users clarity on the reliability of the prediction.

3.3.1 Input Features

The strategy model considers various key features that influence race strategy. Table 4 details the input features for the race strategy prediction model, covering variables influencing pit stop decisions and tyre performance:

Feature Name	Description
Track	The selected Formula 1 circuit
Start Position	The driver’s grid position
Average Stint length	Calculated based on typical race data for each track
Temperature Range	Standard deviation between the air and track temperature
Air Temperature	Measured at race start
Track Temperature	Surface temperature of the circuit
Rainfall Intensity	Binary classification of dry or wet race conditions
Track Speed Index	Estimated based on average lap speeds
Track Layout Type	Categorical encoding of track as high-speed, medium-speed, or low-speed
Overtaking Difficulty	Categorical encoding based on track characteristics
Number of Pit Stops	Maximum planned pit stops based on a track's profile

Table 4: Input Features used for Race Strategy Prediction

These features were chosen to simulate how real-world race engineers consider numerous factors when planning strategies.

3.3.2 Strategy Modelling Approaches

Several classification models were trialled during model development, including decision trees and gradient boosting. Random Forest Classifiers achieved the best balance between accuracy and speed, making them the most suitable choice for mobile development.

The system uses a hybrid modelling approach to generate race strategies. Two Random Forest Classifiers are trained separately for dry and wet races. They predict the most probable pit stop sequences based on the input conditions. The dry race model is restricted to dry compound strategies, while the wet model includes intermediate and full wet stints where applicable. Input features such as Track Layout and Overtaking Difficulty are label-encoded, and additional features like expected tyre degradation and stint difficulty are engineered to improve prediction reliability. The models are optimised using GridSearchCV to tune hyperparameters, including tree depth, number of estimators, and minimum split sizes. After the initial prediction, a filtering process removes unrealistic strategies, such as those that suggest dry tyres at race start during wet confirmed conditions, unless a mixed forecast is detected. Once filtered, the strategy probabilities are ranked, and the top three options are returned, giving the user a primary recommendation and alternate plans. This maintains the flexibility of machine learning while enforcing race logic, allowing the model to adapt to Formula 1’s unpredictable elements.

3.3.3 Output and Confidence Handling

The race strategy system outputs a primary recommendation alongside up to three alternate pit stop plans, each ranked by predicted confidence. Each strategy includes a pit stop sequence, such as “S-S-M”, and a confidence score derived from the model’s probability estimates. These scores allow users to quickly assess how strongly the model favours a particular plan under race conditions. Alternate strategies provide backup options that range from more conservative approaches to riskier alternatives. This enables greater flexibility for handling unpredictable race scenarios. In wet races, the system prioritises strategies that begin on intermediate or wet tyres unless mixed or drying conditions are detected; in this case, dry tyre sequences may still be suggested. For example, a driver starting in position 10 during heavy rain might receive a recommended "W-W-S" (wet-wet-soft) strategy with a confidence of 0.85, followed by alternates like I-W-S at 0.62 and W-S-M at 0.58. This output gives users realistic outputs that mirror how real Formula 1 race engineers evaluate multiple plans based on changing track conditions.

3.4 Mobile Application Design

The mobile application focused on a strong user experience and responsiveness, while maintaining technical performance. The goal was to create an interface that allows users to easily interact with the prediction systems while preserving simplicity and performance. Android Studio was the main development environment, and the application was built using Jetpack Compose. Design prototypes were initially created using Figma to ensure user flows were intuitive and the layout could support future features.

3.4.1 UI/UX Principles

The application design follows Google's Material Design 3 standards, providing a clean and professional appearance across all screen designs. A monochromatic theme using soft greys, whites, and race-themed accent colours was selected to increase readability and maintain visual consistency. Expandable cards, scrollable lists, and modular layouts were used across the app to allow simple navigation between core functionalities. Maintaining minimal user taps was a key feature, and a clear hierarchy using dynamic sizing and spacing was used.

3.4.2 Key Screens

The app's core navigation flows through a range of main screens. Table 5 outlines these screens implemented within the mobile application and shows how users interact with the simulations and visualisations:

Screen Name	**Purpose**
Login/Register	A simple, focused authentication flow built on Firebase Authentication. Both screen
Home Screen	After logging in, users are greeted with a landing page featuring current race information, live drivers' standings, constructors ' points, and access to simulation models. Key race details such as the next event, dates, and championship standings are fetched live and displayed.
Qualifying Simulation Screen	Users select a track from the vertical card scroller. Upon selection, the app simulated predicted qualifying times for all 20 drivers, showing lap times and positions. Expandable cards for each driver allow users to inspect detailed sector performance and driving statistics.
Strategy Simulation Screen	Users input current track and weather conditions, and the app predicts the optimal race strategies for every driver on the grid. Results are given in expandable format, displaying the recommended pit stop sequences, alternate methods, and confidence levels.
3D Modelling Screen	A dedicated section where users can view interactive 3D models of Formula 1 cars by selecting a team. This feature includes the ability to rotate and inspect cars in 3D space and displays team stats such as championship wins and best race finishes alongside the model.
Settings	It offers options for managing preferences, toggling between live and simulation modes, adjusting visual accessibility options, and logging out of the application.

Screen Name	Purpose
Login/Register	A simple, focused authentication flow built on Firebase Authentication. Both screen
Home Screen	After logging in, users are greeted with a landing page featuring current race information, live drivers' standings, constructors ' points, and access to simulation models. Key race details such as the next event, dates, and championship standings are fetched live and displayed.
Qualifying Simulation Screen	Users select a track from the vertical card scroller. Upon selection, the app simulated predicted qualifying times for all 20 drivers, showing lap times and positions. Expandable cards for each driver allow users to inspect detailed sector performance and driving statistics.
Strategy Simulation Screen	Users input current track and weather conditions, and the app predicts the optimal race strategies for every driver on the grid. Results are given in expandable format, displaying the recommended pit stop sequences, alternate methods, and confidence levels.
3D Modelling Screen	A dedicated section where users can view interactive 3D models of Formula 1 cars by selecting a team. This feature includes the ability to rotate and inspect cars in 3D space and displays team stats such as championship wins and best race finishes alongside the model.
Settings	It offers options for managing preferences, toggling between live and simulation modes, adjusting visual accessibility options, and logging out of the application.

Table 5: Key Screens used for Mobile Application Design

Navigation between all screens is handled using Jetpack Navigation.

3.4.3 API Integration and Visualisations

The application connects to the Ergast API to retrieve real-world Formula 1 data. API requests are managed asynchronously using Kotlin coroutines, preventing UI blocking during data fetches. Data caching mechanisms were implemented to limit redundant requests, reduce network load, and improve app responsiveness during repeated access.

To add further engagement beyond pure data predictions, an optional 3D car visualisation feature was developed. Using glTF assets optimised for mobile devices, 3D models for each team were integrated through the SceneView library. Users can select a team from a scrollable horizontal menu and view highly detailed representations of their 2025 Formula 1 cars. Alongside the model, dynamic text fields present information such as the team's history and championship statistics. This module is entirely optional and does not affect core application performance.

4 Plan

The planning phase was developed following the initial research and design work completed earlier in the project, including background research outlined in the Literature Review and the system overview detailed in Section 3.1. The goals and overall scope of the project were established early and are summarised in Section 1.2. As the project progressed, the plan was adjusted to accommodate the additional complexity introduced by the machine learning model and the mobile application integration and 3D visualisation. Knowledge gained during the research phase was used to shape the project timeline, prioritise critical development tasks, and ensure that sufficient time was allocated to high-impact areas, such as model validation.

4.1 Functional Requirements

Table 6 outlines the core functional requirements identified for the Formula 1 prediction application, detailing the essential features and system capabilities implemented to meet the project objectives:

Number	Requirement
1	The system must predict qualifying lap times for all 20 drivers based on track selection and environmental inputs.
2	The system must generate optimal and non-optimal race strategies for each grid position, with confidence scores.
3	Users must be able to input air temperature, track temperature, and rainfall conditions.
4	The system must display live Formula 1 driver and constructor standings via API integration.
5	Users must be able to register, log in, and save preferences using a secure authentication system.
6	Users must be able to view an interactive 3D model of a selected Formula 1 team car.
7	Predictions must be presented in an expandable, user-friendly mobile layout.

Table 6: Functional Requirements of the Formula 1 Prediction Application

4.2 Non-Functional Requirements

Table 7 summarises the key non-functional requirements identified for the system, focusing on factors such as performance and security that must be achieved during development:

Number	Requirement
1	Predictions must load within reasonable time frames on a mid-range Android device.
2	The application must be usable offline once data has been initially retrieved.
3	The system must maintain secure handling of user data via Firebase Authentication.
4	The application must follow clean, responsive mobile UI design practices.
5	Predictions and outputs must remain consistent across different devices.
6	Integration with APIs must not block or slow down the user interface during data fetches.

Table 7: Non-Functional Requirements of the Formula 1 Prediction Application

4.3 Development Methodology

The development of the system followed a structured, milestone-based approach that aligned with the key phases of the project lifecycle. Despite not following a formal Agile or Waterfall model, the methodology was guided by an iterative and flexible structure, allowing continuous progress and refinement. The initial plan was shaped by early research into project requirements, then adjusted as the technical challenges started to appear when dealing with the machine learning models.

Development was split into five logical phases: research, model development, mobile application build, integration, and testing. The project began with an in-depth research phase covering topics such as machine learning in motorsport. This phase directly informed the model selection and input feature design for the qualifying and strategy systems. Once the system design was finished, model training and evaluation were prioritised early to ensure the predictions were reliable and accurate before any user-facing integration occurred.

Mobile development followed a modular, screen-by-screen approach using Jetpack Compose. Each screen was initially implemented with placeholder content, which was then dynamically linked to live API responses or embedded prediction logic. Chaquopy allowed the machine learning models to be embedded directly within the mobile application.

The original system plan and requirements were revisited and refined throughout the project in response to technical constraints. As implementation progressed, features were adjusted or added based on how well the models performed and how effectively they integrated into the mobile app. For example, the initial requirements were expanded to include synthetic feature generation after limitations in real telemetry data became clear, and prediction loading times influenced how the app handled Chaquopy interaction. These iterative updates to the plan ensured that the core functionality remained the focus point, while still allowing the system to grow, leading to the inclusion of enhancements such as 3D car visualisation.

4.4 Project Planning and Gantt Chart

The project was planned in phases, starting with research and requirements analysis and then developing, integrating, and testing. The timeline involved during the build process as new features such as 3D visualisation and live data integration were added. Figure 2 below shows the key stages and when each task was actively worked on:

	Dec	Jan	Feb	Mar	Apr
1) Research and Literature Review	▨	▨			
2) Requirements Gathering and Planning		▨	▨		
3) Data Collection for Race Strategy AI			▨		
4) Initial Model Development			▨	▨	
5) Integration with Race Simulations				▨	
6) Model Testing and Validation				▨	
7) Final Changes and Refinements					▨
8) Documentation and Submission					▨

Figure 2: Gantt Chart of Project Phases

Table 8 briefly describes each phase in the project timeline, as outlined in the Gantt chart:

Phase	**Description**
Research and Literature Review	Conducted background research into relevant machine learning techniques and F1 systems.
Requirements Gathering and Planning	Defined system objectives, functional and non-functional requirements, and initial timelines.
Data Collection for Race Strategy AI	Collected and structured historical and environmental data to support simulations.
Initial Model Development	Trained machine learning models for qualifying and strategy predictions using Scikit-learn.
Integration with Race Simulations	Linked prediction models to the mobile app using Chaquopy and integrated API data sources.
Model Testing and Validation	Evaluated model performance, tested outputs and refined accuracy.

Phase	Description
Final Changes and Refinements	Polished the UI, improved responsiveness, and finalised systems behaviour.
Documentation and Submission	Completed the report, combining everything.

Table 8: Summary of Project Timeline Phases

4.5 System Limitations and Risk Considerations

This system relies entirely on public data and cannot access official Formula 1 telemetry or car setup information. Key driver telemetry data, such as throttle usage, was estimated using engineering features based on historical lap trends and environmental inputs to work around this. While this approach was practical for general simulation, the accuracy of these values is still limited compared to real-world data. As a result, there is a ceiling to how precise and realistic the predictions can be, particularly for outlier conditions like changing weather or unusual track evolution.

The models assume optimal race and qualifying conditions. For example, the lap time predictions are based on clean flying laps, with no interruptions, such as traffic or yellow flags. Similarly, the strategy predictions treat the race as either fully dry or thoroughly wet. This limits their flexibility in real-world scenarios where races can shift between conditions or involve unexpected events like safety cars. Although the models produce practical and realistic outputs for simulation, they do not currently adjust in real-time or respond to live race changes.

Running the models on-device using Chaquopy introduced performance considerations. On mid-range devices, results were delivered in under two seconds, but lower-end devices may face delays or struggle with memory usage. These risks were minimised by avoiding heavyweight libraries and keeping the models lightweight, but the risk remains for users on outdated hardware.

Live data integration, such as driver and constructor standings, depends on external APIs. Some app parts may fail or show outdated results if those APIs go offline or return malformed data. Local caching was implemented to reduce this risk, but the app still assumes the availability of stable internet during key interactions. There is also a risk that users might over-trust the results, particularly in strategy suggestions. To address this, the app displays multiple strategy options with confidence scores and indicates that all outputs are simulations.

4.6 Testing and Evaluation Approach

Both machine learning models were tested using real-world race data from the 2025 current season. The lap times prediction model was evaluated using RMSE, MAE, and R^2. It achieved a root mean squared error of 0.275 seconds, a mean absolute error of 0.230 seconds, and an R^2 score of 0.899, showing strong alignment between predicted and actual lap times. Grid order predictions were consistent with actual qualifying results, and the model reliably captured performance differences across teams and weather conditions. The race strategy model was validated against known pit stop trends from past races, and the top predicted strategies often aligned with real-world outcomes, particularly under dry conditions.

All core features, such as model loading and prediction triggers, were tested on physical mobile application devices and emulators. The app remained responsive and stable during standard use, with no critical issues. The qualifying and strategy simulations loaded under two sections on a mid-range device (Galaxy Tab A9), meeting the performance expectations for real-time interaction.

Cross-device compatibility was tested by running the app on different screen sizes to ensure layout consistency and response UI behaviour. The app also handled network issues correctly. When APIs were unreachable, cached data was shown without crashing the app. This ensured that users could still view data even in unstable network conditions.

User interface elements such as expandable cards and navigation transitions were tested for usability and smoothness. Special attention was given to keeping the required taps low and maintaining clarity across screens. No dead links, navigation bugs, or layout breaks were encountered during testing. These tests confirmed that the system performs well in prediction accuracy and provides a reliable and user-friendly experience on mobile/tablet.

5 Implementation and System Setup

This section explains how the system was built, covering the development environment, data processing pipelines, machine learning models, and the mobile app. Every stage is covered step by step, from acquiring raw input data to real-time predictions for qualifying laps and race strategies. Each component has been implemented with the three fundamental features of performance, accuracy, and mobile responsiveness. Where needed, the workaround was applied to overcome restrictions like the lack of official telemetry data, and both models were optimised to run entirely on the device using Chaquopy. The implementation also covers content such as 3D visualisation and live data integration, and how the user interacts with the system from start to finish.

5.1 Development Environment

The system was developed across two main environments: one for building and training the machine learning models, and another for designing and developing the mobile application. Tooling choices were based on factors such as flexibility and performance.

5.1.1 Machine Learning Development

The qualifying and race strategy models were developed entirely in Python 3.8 (due to Chaquopy compatibility), using VS Code as the primary IDE. This environment provided complete control over data preprocessing and feature engineering processes. Table 9 summarises the core Python libraries used throughout the development phase:

Library	**Purpose**
Scikit-learn	Regression/classification models (Random Forest Regressor/Classifiers) were used for GridSearchCV to tune model hyperparameters.
Pandas/Numpy	Data manipulation and transformation.
Joblib	Exporting trained models and scalers for mobile integration
Matplotlib/Seaborn	Debugging and plotting internal evaluation metrics during development
Openpyxl	Reading .xlsx files as input data during model training.

Table 9: Core Python libraries used.

All model experimentation, tuning (via GridSearchCV), and synthetic feature creation were handled in this environment. Version control of both the qualifying and race strategy models via Git, with local branches used to separate experimental model iterations.

The machine learning development pipeline was CPU-based due to the lightweight chosen models. GPU acceleration was unnecessary because the priority was fast inference on-device rather than model training throughput.

5.1.2 Mobile Application Development

The mobile application was built in Android Studio (Kotlin), using Jetpack Compose as the UI framework. Compose was chosen for its clean state management and flexibility in creating dynamic and data-driven layouts such as scrollable prediction results and expandable panels.

Initial UI wireframes and interaction flows were created in Figma. These mock-ups informed the final Jetpack Compose layouts, which were implemented using the following main components:

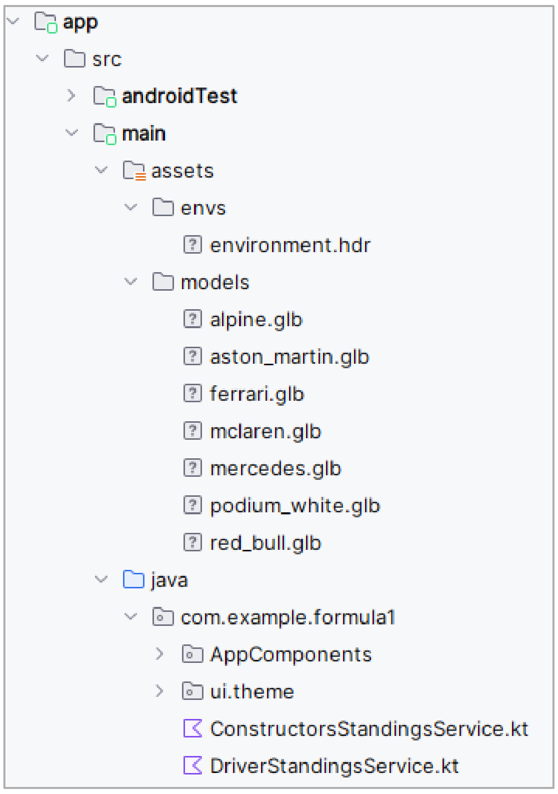
- LazyColumn is used to render scrollable lists for prediction cards.
- Card layouts for encapsulating driver or strategy outputs.
- Navigation components for screen transitions between simulation modules.
- ViewModel classes to manage asynchronous model inference and API data retrieval.

Both prediction systems were developed in Python and bundled inside the app using Chaquopy, which allows .py scripts and .joblib models to run on-device via the embedded Python interpreter. This removed the need to convert the models into TensorFlow or other mobile formats.

The application is structured into modular Kotlin packages for clarity and maintainability:

- Screens/ - Jetpack Compose UI screens for Qualifying, Strategy, Login, Home, etc.
- Predictors/ - Kotlin activity classes, such as SimulateQualifyingScreenActivity.kt, handle the interaction with embedded Python models via Chaquopy. These classes are stored within the com.example.formula1 package.
- Models/ - Kotlin data class definitions for mapping model results.
- Chaquopy/ - Embedded .py model files and assets, loaded at runtime.

Prediction results are returned as CSV from Python, decoded into Kotlin objects, and then observed through MutableState variables that trigger UI recomposition on completion. This flow ensured non-blocking prediction and responsive feedback in the UI layer. Figure 3 shows a snippet of the structure inside Android Studio, including the embedded Python scripts, Excel data sources, and GLB model files for the 3D visualisation:



```
app
  src
    androidTest
    main
      assets
        envs
          environment.hdr
        models
          alpine.glb
          aston_martin.glb
          ferrari.glb
          mclaren.glb
          mercedes.glb
          podium_white.glb
          red_bull.glb
      java
        com.example.formula1
          AppComponents
          ui.theme
          ConstructorsStandingsService.kt
          DriverStandingsService.kt
```

Figure 3: Project directory structure example within Android Studio.

5.1.3 Integration Tools and Libraries

Chaquopy was configured with a clean Python entry point for each model to integrate the machine learning models into the mobile app. On the Kotlin side, wrapper classes manage the interaction by passing input parameters to Python, which writes predictions as CSV files. These files are then read in Kotlin and parsed into structured data classes. This allows complete control over model execution while maintaining a consistent Kotlin interface. The integration structure is as follows:

- SimulateQualifyingScreenActivity.kt – handles all calls to the qualifying model, formats input features, receives sector deltas and lap times.
- SimulateStrategyScreenActivity.kt – manages race condition input, strategy inference, and output confidence ranking.
- Chaquopy.PyObject.callAttr() – used to invoke Python methods from Kotlin directly

Python writes returned outputs as CSV-formatted strings, which are parsed in Kotlin and mapped into structured data classes. All model execution is handled asynchronously to avoid blocking the main thread. Snippet 1 shows how the race strategy output is structured after being returned from the embedded Python model. This format allows for clean binding to the UI layer in Jetpack Compose.

```
data class StrategyPredictionClass(
    val bestStrategy: String,
    val bestConfidence: Double,
    val numPitStops: Int,
    val alternatives: List<AlternativeStrategy>
)

data class AlternativeStrategy(
    val strategy: String,
    val confidence: Double
)
```

Snippet 1: Data class for handling race strategy prediction output in Kotlin

The external libraries and APIs used across the app were:

- Firebase authentication is used for secure login and account management.
- Firebase Firestore is used to store user preferences and app settings.
- Ergast API (via Retrofit), to get live F1 data such as race calendars and driver standings.
- Glide is used for efficient image loading for the constructor logos and visual elements.

Snippet 2 shows how the qualifying prediction model is triggered from Kotlin using Chaquopy. The Python module qualifying.py is called via callAttr() and passes in the selected track number. The resulting output is a CSV-formatted string containing information such as sector times, which is then parsed and used to update the UI.

```
val python = Python.getInstance()
val pyObjectResult = python.getModule("qualifying")
    .callAttr("main", trackNumber)
val rawCsv = pyObjectResult.toString()
```

Snippet 2: Kotlin-Python integration for triggering the qualifying prediction model and retrieving CSV output.

5.1.4 Cross-Platform Testing

The application was tested across multiple devices and configurations to ensure consistent performance and model inference reliability under varied conditions. Table 10 shows the range of physical and virtual devices used to test the application across different performance tiers:

Device	Type	Purpose
Galaxy Tab A9	Physical Device	Primary development and testing device; used for real-world validation.
Pixel 7 Emulator	Emulator (High)	Simulated top-end performance for UX responsiveness testing.
Galaxy A52 Emulator	Emulator (Mid)	Simulated lower-spec performance under memory/CPU constraints.

Table 10: Devices used for cross-platform testing and performance validation.

Compatibility testing covered Android versions 11 through 14, with a focus on: model loading time, screen rendering speed, layout responsiveness under low memory conditions, and API fallback behaviour in poor network conditions. Testing confirmed that both machine learning models executed in under 2 seconds on target hardware, keeping interactions within an acceptable UX latency threshold.

5.1.5 Version Control and Project Management

Version control was handled using Git, with clear commit messages and branches grouped by feature (e.g. qualifying model). A private GitHub repository was also used for backups. In addition, milestone checkpoints were tagged using Git tags. Project management and task tracking were handled using Notion, with a Kanban board to organise to-dos and feature progress.

5.2 Data Acquisition and Preprocessing

This section explains how data was sourced, structured and processed for use in the qualifying and race strategy models. Due to the lack of official Formula 1 telemetry, the datasets were assembled using a combination of public APIs and manually structured Excel files. All preprocessing was implemented in Python 3.8 using pandas, Openpyxl, and scikit-learn.

5.2.1 Data Sources

Two datasets were used to support the qualifying model:

- HistoricalData.xlsx – contains qualifying session data across previous sessions.
- CurrentData.xlsx – mirrors the same structure, but contains updated entries for the 2025 grid.

Both datasets include features such as driver IDs, team mappings, sector times, weather conditions, and telemetry data such as throttle and RPM. The only distinction is that CurrentData.xlsx is filled with driver and team data only for the current season. This allows the system to simulate a full qualifying session on any selected track for the 2025 grid.

HistoricalData.xlsx was used exclusively during model training, while CurrentData.xlsx is used during inference to run live simulations in the app. The structure of each row represents a driver-track entry. This separation ensures the model doesn't leak future data into its training process. It supports the fast generation of full-grid qualifying outputs when users select a track inside the app. Weather-related fields such as air temperature, track temperature, wind speed, humidity, and 'is wet' were manually entered using race weekend summaries and official session reports. Additional data was cross-referenced using the FastF1 API to verify data. Table 11 shows the features used in the qualifying datasets (HistoricalData.xlsx and CurrentData.xlsx).

Feature Name	Description
Year	Season year of the session
Track	Numeric ID representing the selected track
Driver ID	Encoded identifier for the driver
Team ID	Encoded identifier for the constructor team
BaselineS1, S2, S3	Pole lap sectors are used as a reference for delta calculations.
DeltaS1, S2, S3	Time difference between the driver and baseline per sector
Average Throttle	Estimated average throttle for the lab
Average Brake	Estimated breaking force based on speed and compound
Average RPM	Simulated RPM based on average lap speed and track length
Sector Speed X	Speed at speed traps
Compound	Encoded tyre compound (e.g. Soft = 1)
Race Number	Race order in the calendar (e.g. 1 = 1st race of season)

Table 11: Features in HistoricalData.xlsx and CurrentData.xlsx

The strategy model used a separate file:

- StrategyData.xlsx contains, for example, year, track, average stint length, start position, and strategy.

This file was used to train the Random Forest classifier for race strategy recommendations.

All Excel files are located under the python/ directory of the Android Studio project. Additional data was cross-referenced for validation using public sources like the FastF1 API. Table 12 shows the features used in the strategy dataset (StrategyData.xlsx), covering start positions, pit strategies, position changes, and track-specific race conditions.

Feature Name	Description
Year	Season year of race recorded.
Track	Encoded track ID
Average Stint Length	Average number of laps per tyre stint for that race
Temp Range	Range of temp during the race
Start Position	Starting grid slot for the driver
Finish Position	Final classified race position
Position Change	Positional gain/loss from start to finish
Strategy	Actual pit strategy used (e.g. “M-S”)
Number Of Pit Stops	Number of pit stops in the strategy.
Air Temp	Ambient air temperature (°C) at race start
Track Temp	Surface temperature at race start (°C)
Is Wet?	Boolean indicating if race was wet (True/False)
Track Speed	Average speed of the track (kph)
Track Type	Encoded label (e.g. “high-speed”)

Feature Name	Description
Overtaking Difficulty	Encoded difficulty of overtaking on track (e.g. "low")

Table 12: Features used in StrategyData.xlsx

5.2.2 Dataset Preprocessing and Cleaning

Missing numeric values were filled using column means to prepare the datasets for training, and categorical fields were encoded using label or one-hot encoding. Abnormal or incomplete rows were manually removed from the Excel sheets where needed. Validation was performed using standard cross-validation without time-based splits, as the data structure did not include session grouping or chronological order.

5.2.3 Feature Engineering

Due to the lack of official telemetry APIS or direct team data, feature generation was built entirely from scratch using race data, derived variables, and engineering telemetry approximations. Each model uses a tailored set of features aligned with its objective – lap time prediction or strategy recommendation.

Qualifying Model

Feature engineering for qualifying was implemented inside qualifying.py. The preprocess_data() and fit_models() functions construct all model inputs.

Baseline Sector Times

Snippet 3 shows how baseline sector times are injected into the dataset using task-specific mappings. Each track is associated with historical Q3 reference times for all three sectors, sourced from the previous Formula 1 season. These values are mapped using the track's unique identifier and applied to the dataset to establish a consistent performance benchmark across all circuits.

```
df['Baseline_S1'] = df['Track'].map(lambda t:
track_sector_baselines[t][0])
df['Baseline_S2'] = df['Track'].map(lambda t:
track_sector_baselines[t][1])
df['Baseline_S3'] = df['Track'].map(lambda t:
track_sector_baselines[t][2])
```

Snippet 3: Track ID mapping injects Baseline sector times into the dataset.

Performance Deltas

Snippet 4 calculates performance deltas by comparing each driver's sector times to the track baselines. These deltas calculate how much faster or slower a driver performs in each sector relative to the benchmark, serving as a crucial input feature for the regression model.

```
df['Delta_S1'] = df['Sector1Time'] - df['Baseline_S1']
df['Delta_S2'] = df['Sector2Time'] - df['Baseline_S2']
df['Delta_S3'] = df['Sector3Time'] - df['Baseline_S3']
```

Snippet 4: Sector deltas derived from driver times vs baseline tracks

Team-Year Sector Averages

Snippet 5 outlines how team-year sector averages are computed and merged into the dataset to reflect the car's relative performance. By calculating the average sector times for each team across different seasons, the model can better consider the drivers' performances and variations in a team's competitiveness.

```
for sec in ['1','2','3']:
    avg = (
        df.groupby(['Year','Team'])[f'Sector{sec}Time']
          .mean()
          .reset_index()
          .rename(columns={f'Sector{sec}Time':f'TeamAvg_S{sec}'})
    )
    df = df.merge(avg, on=['Year','Team'], how='left')
```

Snippet 5: Aggregated team-sector averages added for driver context

Session ID Grouping

Snippet 6 illustrates how a unique SessionID is generated by combining the year and race number. This identifier ensured accurate grouping and alignment of data points from the same race session.

```
all_data['SessionID'] = (
    all_data['Year'].astype(str) + "_" +
    all_data['RaceNo'].astype(str)
)
```

Snippet 6: Creating unique session identifiers for grouping

Snippet 7 presents the training of auxiliary regression models to estimate additional telemetry outputs. These models predict features such as throttle usage, enabling more advanced visualisation and insight within the app's UI when real telemetry data is unavailable.

```
for col in EXTRA_COLS:
    mask = all_data[col].notna()
    if mask.sum() < 30:
        continue
    X_sub = scaler.transform(all_data.loc[mask, FEATURES])
    y_sub = all_data.loc[mask, col]
    model = RandomForestRegressor(n_estimators=120, max_depth=10,
random_state=42)
    model.fit(X_sub, y_sub)
    extra_models[col] = model
```

Snippet 7: Training auxiliary regressors for throttle, RPM, brake, and speed trap features

Strategy Model

Snippet 8 shows how categorical strategy features are encoded for use in the strategy model. The train_model() method transforms race-specific attributes such as track type into numeric labels to enable practical model training and pattern recognition.

```
full_df['TrackTypeEncoded'] =
track_type_encoder.fit_transform(full_df['TrackType'])
full_df['OvertakingDifficultyEncoded'] =
overtaking_diff_encoder.fit_transform(full_df['OvertakingDifficulty'])
full_df['StrategyEncoded'] =
strategy_encoder.fit_transform(full_df['Strategy'])
```

Snippet 8: Label encoding for strategy model categorical figures.

Snippet 9 demonstrates how low-frequency strategy classes are filtered out to reduce class imbalances. By removing strategies with fewer than five occurrences, the model is trained on more representative data, improving the generalisation and reducing the risk of overfitting to rare or outlier strategies.

```
strategy_counts = full_df['Strategy'].value_counts()
common_strategies = strategy_counts[strategy_counts >= 5].index
full_df = full_df[full_df['Strategy'].isin(common_strategies)]
```

Snippet 9: Filtering low-frequency strategy classes

Snippet 10 illustrates how the dataset is split into dry and wet race subsets for separate model training. This separation is based on the IsWet flag and tyre compound indicators, ensuring that strategy predictions account for different race conditions.

```
dry_df = full_df[(full_df['IsWet'] == 0) &
(~full_df['Strategy'].str.contains('I|W'))]
wet_df = full_df[(full_df['IsWet'] == 1) &
(full_df['Strategy'].str.contains('W|I'))]
```

Snippet 10: Splitting data into dry and wet training subsets

5.2.4 Data Scaling and Model Preparation

Before training, all numerical input features were scaled using StandardScaler to ensure fair weight and distribution and consistent inference behaviour on-device. This was done for both models (qualifying and race strategy). Snippet 11 shows how the scaler is fitted to the training data, applied to the inputs, and saved for use during prediction.

```
scaler = StandardScaler().fit(X_tr)
X_scaled = scaler.transform(X_tr)
joblib.dump(scaler, "scaler_strategy.joblib")
```

Snippet 11: Scaling and exporting the fitted StandardScaler used in the qualifying model training

In qualifying.py, the scaler and trained Random Forest Regression models were saved as .joblib files and loaded dynamically during inference through Chaquopy. Additional models for auxiliary features were saved in a bundled dictionary and exported alongside the main model. Snippet 12 shows how the strategy models were trained for wet and dry conditions using a Pipeline, with hyperparameter tuning via GridSearchCV.

```
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', RandomForestClassifier(random_state=42,
class_weight='balanced'))
])

grid = GridSearchCV(pipeline, param_grid=param_grid, cv=3, n_jobs=-1,
scoring='accuracy')
grid.fit(X_train, y_train)
```

Snippet 12: Defining and training the strategy model pipeline with scaling and classification steps

In strategy.py, two models were trained and saved – one for dry races and one for wet. Each used a Pipeline containing both the scaler and classifier. After training, the best model and encoders were stored in a dictionary and cached using joblib.

5.2.5 Summary of Processing Pipeline

Figure 4 shows the complete data pipeline. Raw Excel files are pre-processed and passed through feature engineering scripts before training. Models and scales are saved as .joblib files and then loaded in-app using Chaquopy for fully offline inference. No external APIs are required at runtime.

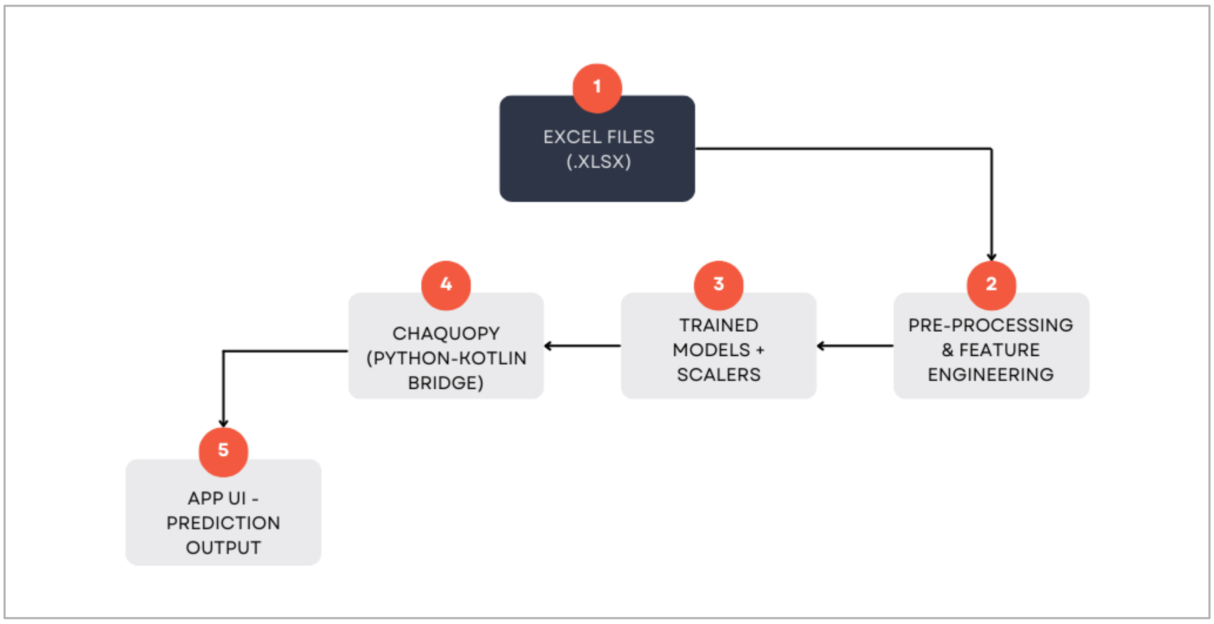

1
EXCEL FILES (.XLSX)
2
PRE-PROCESSING & FEATURE ENGINEERING
3
TRAINED MODELS + SCALERS
4
CHAQUOPY (PYTHON-KOTLIN BRIDGE)
5
APP UI - PREDICTION OUTPUT

Figure 4: Complete pipeline diagram

5.3 Qualifying Lap Time Prediction System

This system generates a qualifying simulation for the current Formula 1 grid on any selected track. It returns data such as predicted sector times, full lap times, and simulated telemetry for all 20 drivers using trained models and pre-processed data.

5.3.1 Prediction Pipeline

When a user selects a track, the app calls the predict_qualifying_results() inside qualifying.py. This function loads the latest model and scaler, extracts current driver/team data from the CurrentData.xlsx sheet, and builds feature rows for predictions. Snippet 13 shows how the feature rows are built from the current grid to prepare input for the model.

```
rows = [{
    'Year': 2025,
    'Driver': r['Driver'],
    'Team': r['Team'],
    'Track': track_number,
    'TeamAvg_S1': r['TeamAvg_S1'],
    'TeamAvg_S2': r['TeamAvg_S2'],
    'TeamAvg_S3': r['TeamAvg_S3']
} for _, r in latest.iterrows()]
```

13: Feature row construction from the current grid for inference.

5.3.2 Lap Time Prediction

The base model predicts sector deltas relative to the track's reference pole time. These deltas are added to static baselines to form final sector and lap times. Snippet 14 shows how the predicted deltas are applied to baseline times to compute final sector and lap values.

```
deltas = base_rf.predict(Xs_inf)
b1, b2, b3 = track_sector_baselines[track_number]
secs = deltas + np.array([b1, b2, b3])
laps = secs.sum(axis=1)
```

Snippet 14: Constructing complete laps by combining model deltas with baseline times

The output includes all sector times, lap time, and a computed Position column based on the lap time ranking. Drivers are then mapped to team name labels using the ID dictionaries.

5.3.3 Simulated Telemetry

To improve the UI experience, telemetry-like outputs (e.g. throttle usage) are generated using auxiliary regressors trained on available data. If a feature is missing, the system returns to a known average for that driver. Snippet 15 shows how auxiliary regressors predict telemetry values or fall back to defaults.

```
for col in EXTRA_COLS:
    if col in extra_models:
        out[col] = extra_models[col].predict(Xs_inf)
    else:
        out[col] = [_fallback_latest(drv, col) for drv in
out['Driver']]
```

Snippet 15: Generating telemetry fields using auxiliary regression models

These metrics are displayed in expandable driver cards inside the app, allowing users to explore detailed driver performance.

5.3.4 Output Formatting and Sorting

Snippet 16 shows the final formatting step where predicted results are sorted by lap time and key output fields are organised. The resulting data frame includes driver and team names, individual sector times, total lap time, and other telemetry metrics, presented in race order to mirror a realistic qualifying session.

```
ordered_cols = ['Position', 'DriverName', 'TeamName',
                'Sector1Time', 'Sector2Time', 'Sector3Time'] +
EXTRA_COLS + ['LapTime']
return
out[ordered_cols].sort_values('Position').reset_index(drop=True)
```

Snippet 16: Result formatting, sorting and column ordering for UI display.

5.4 Race Strategy Prediction Engine

The engine functions as a decision-making tool that emulates the dynamic strategy planning seen in modern Formula 1. It uses historical race data and environmental inputs to recommend optimal tyre and pit stop strategies for dry and wet conditions.

5.4.1 Model Design and Structure

Two separate Random Forest classifiers were trained using structured data from StrategyData.xlsx. These models were designed to handle key race parameters such as stint dynamics and weather conditions. Rather than relying solely on static heuristics, the engine captures the probabilistic nature of race-day outcomes by learning from past event patterns.

Feature vectors are constructed using encoded categorical variables (e.g. Overtaking Difficulty) and numerical values (e.g. Average Stint Length).

Only strategies with sufficient support (> 4 samples) were retained to improve generalisation during training. A separate pipeline is maintained for wet races, recognising the need to account for intermediate and full wet tyre scenarios.

5.4.2 Inference and Application

During execution, the model dynamically adapts to user inputs such as track number and weather. Relevant contextual data is fetched using a predefined mapping, and the input vector is constructed accordingly. Below shows a core snippet showing how feature preparation and prediction occur during inference:

```
# Encode categorical features
track_type_encoded = track_type_enc.transform([track_type])[0]
overtaking_difficulty_encoded =
overtaking_enc.transform([overtaking_diff])[0]

# Final feature vector for prediction
features = [
    track, start_position, stint_length, temp_range,
    air_temp, track_temp, is_wet, track_speed,
    track_type_encoded, overtaking_difficulty_encoded, num_pit_stops
]

predicted_strategy = model.predict([features])
```

Snippet 17: Preparing real-time feature vector for race strategy inference

This setup allows the engine to simulate strategies under various hypothetical race conditions or reflect outcomes of a live race.

5.4.3 Deployment and Integration

Models are trained offline in Python and serialised using joblib. Inference is performed on-device within the Android application. This enables predictions to be made without network access, supporting real-time usability. By combining historical learning with current telemetry-like input, this engine offers high-confidence suggestions (e.g. "M-H" with 86% confidence) and alternate fallback or adaptive planning strategies.

5.5 3D Visualisation Module

The 3D Visualisation Module enhances the user experience by providing a dynamic and interactive interface for exploring Formula 1 car models and constructor information. While not part of the core analytics, this component plays a vital role in engagement and immersion.

5.5.1 Structure and Assets

Each constructor has a .glb model stored under assets/models/ (e.g. red_bull.glb). A single podium model (podium_white.glb), generated from Blender, is also included to anchor the scene visually. Lighting is provided by a high-dynamic-range (HDR) environment map under assets/envs/environment.hdr.

The UI lets users scroll through constructors and load the corresponding car model into the scene. On tap, a modal appears with construction statistics, such as team principles and the number of podiums.

5.5.2 Scene Configuration

Snippet 18 shows how 3D models and environment are loaded into a SceneView for real-time rendering in the Compose layout. A central node anchors the camera orbit around the selected car, with interactions through gestures enabling rotation. Models are dynamically loaded based on user selection, and the environment is loaded using an HDR background to enhance realism.

```
val model = modelLoader.createModelInstance("models/red_bull.glb")
val environment =
environmentLoader.createHDREnvironment("envs/environment.hdr")
```

Snippet 18: SceneView model and environment loading

Performance and Behaviour

Assets are lightweight and fully offline. SceneView handles rendering efficiently on mid-tier Android hardware, averaging around 30 FPS even with gestures and lighting enabled. Only one model is kept in memory to ensure smooth performance.

5.6 Mobile Application Architecture

The mobile application uses a single-activity architecture managed entirely with Jetpack Compose. Navigation is handled without a NavController. Instead, routing between screens is based on managing state (selectedPage) inside a Scaffold and conditionally rendering Composable screens.

5.6.1 Navigation and Structure

Snippet 19 outlines how screen navigation is managed using a selected page state variable and Jetpack Compose's Scaffold layout. The bottom navigation bar dynamically updates with the current screen based on user input, and is rendered conditionally based on app states like isQualifyingActive. This structure keeps navigation logic centralised within the MainScreen() composable.

```
var selectedPage by remember { mutableStateOf("Home") }

Scaffold(
    bottomBar = {
        if (!isQualifyingActive) {
            NavigationBar(
                selectedItem = selectedPage,
                onClick = { item → selectedPage = item }
            )
        }
    }
) { paddingValues →
    Box(Modifier.padding(paddingValues)) {
        when (selectedPage) {
            "Home" → HomeScreen(...)
            "Simulate" → SimulateHomeScreen(setQualifyingActive = {
isQualifyingActive = it })
            "Settings" → SettingsScreen()
        }
    }
}
```

Snippet 19: Code that handles all navigation.

There is no NavGraph. Screens like Qualifying, Strategy, and 3D Modelling are embedded directly into the logic of MainScreen() and toggle via flags and mutable states.

5.6.2 Screen Management

Each central section of the app (e.g. Qualifying Simulation) is represented as a top-level composable. Flags such as isQualifyingActive suppress or enable certain UI elements, such as hiding the navbar during full-screen simulations. For example:

- SimulateHomeScreen() handles both strategy and qualifying entry points
- Qualifying mode activates full-screen layout, hiding the bottom navigation bar
- The navigation bar only reappears when the user exits simulation mode

5.7 Cloud and Live Data Integration

The mobile app integrates live data and cloud services using Firebase and the Ergast API. These services support features such as user account management and access to up-to-date Formula 1 standings without requiring any custom backend infrastructure.

5.7.1 Firebase Authentication and Cloud Services

Snippet 20 shows how user logout is handled using Firebase Authentication. The app supports email/password and Google Sign-In, with session persistence enabling easy access across launches. On logout, the user's session is cleared using Firebase's signOut() method, redirecting them to the login screen on the next launch.

```
FirebaseAuth.getInstance().signOut()
```

Snippet 20: User logout function

No multi-factor authentication is used. Google Sign-In was implemented and is fully functional via the standard Firebase identity provider integration.

Firebase was also implemented and tested to store user preferences such as favourite constructor/driver. This functionality worked but was commented out during final development. Local state was deemed sufficient for this app version, and remote persistence didn't improve the experience meaningfully.

5.7.2 Live Race Data via Ergast API

Snippet 21 shows the Retrofit base URL configuration used to access live race data from the Ergast API. This setup enables the app to retrieve up-to-date information: driver standings, constructor rankings, and race schedules. API calls are executed within a viewModelScope using Dispatchers.IO, with responses parsed through Gson to integrate within the UI.

```
Retrofit.Builder()
    .baseUrl("https://api.jolpi.ca/ergast/")
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

Snippet 21: Base URL setup

Responses are handled safely with fallback defaults. If an API call fails or returns null, the app shows cached or empty content without crashing.

5.7.3 Offline Support & Caching

To improve responsiveness and avoid blank screens when offline:

- API responses are cached in memory and optionally on disk
- Prediction models and all required data files are bundled in the APK
- 3D assets (GLBs, environment maps) are stored in /assets/

This ensures the app can still function offline – users can view old results and run predictions without an internet connection.

6 Testing and Evaluation

This section shows how each central feature of the system was tested and evaluated for features such as correctness and reliability. The goal was not only to confirm whether each component worked unreal ideal conditions, but to assess how it behaves under circumstances such as failures and invalid inputs. Testing was performed on physical devices (Galaxy Tab A9) and Android emulators, with varying performance profiles and network conditions. Each feature was tested in isolation as part of the whole system. Tests were based on realistic user behaviour and common edge cases, ensuring all features performed reliably across practical use cases.

6.1 Firebase Authentication

Authentication was tested across all user flows, including registration, email/password login, Google Sign-In, logout, and session persistence. The aim was to ensure that authentication logic worked and failed gracefully and predictably in edge cases. Tests were performed using a mix of valid, invalid, and missing credentials. FirebaseAuth exceptions and duplicate registration attempts were monitored closely via log outputs (using Logcat).

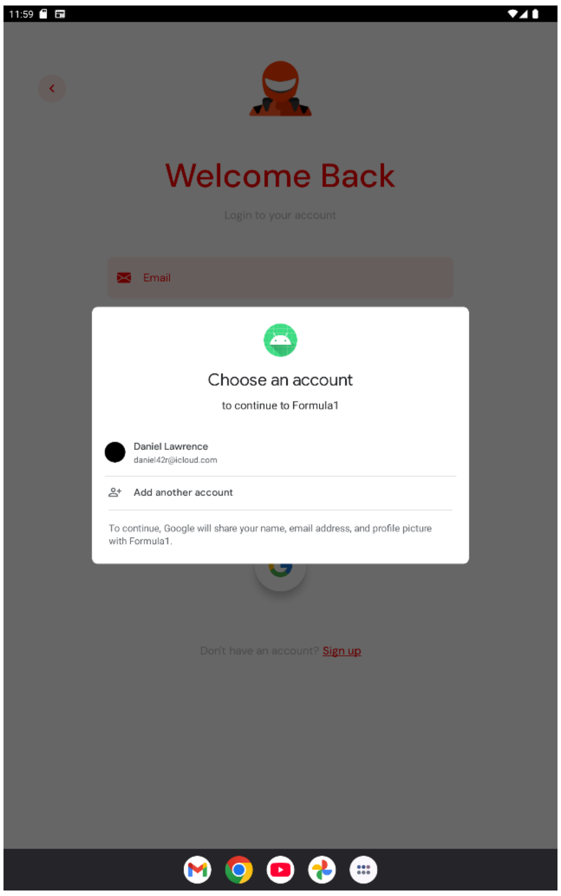
On valid input, registration was completed successfully, and the user was returned to the main screen. Weak password inputs triggered validation errors before network calls were made. Repeated registration with the same email surfaced proper feedback tied to Firebase's FireBaseAuthUserCollisionException. Login behaviour was similarly solid: incorrect passwords were rejected, and appropriate error feedback was rendered without crashing or UI freezing. Session persistence was tested by restarting the app, which correctly resumed the previous session without requiring re-login.

Google Sign-In was tested using both successful authentication and user-initiated cancellations. Account switching was also verified. The app responded correctly to all cases and did not leak session state. Offline behaviour was tested by disconnecting the network and attempting login and registration. These actions failed silently, and the UI remained stable.

6.1.1 Results Summary

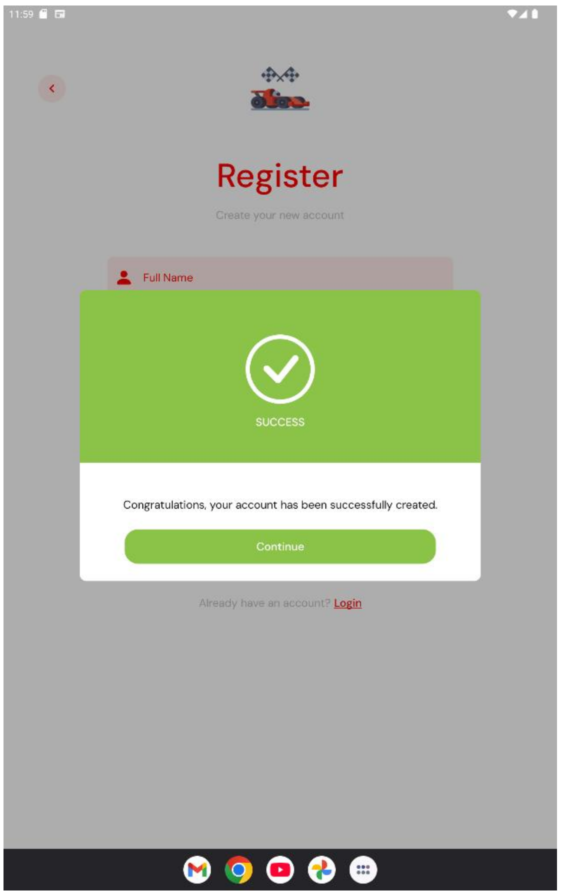
Scenario	Expected Behaviour	Outcome
Valid registration	New account created; success prompt displayed	✓ Pass
Weak password / invalid password / invalid email	Field-level errors shown, blocked submission	✓ Pass
Duplicate registration	Firebase exception caught, inline feedback shown	✓ Pass
Valid login	Auth state updated; app navigated to home	✓ Pass
Wrong password	Login blocked, feedback shown	✓ Pass
Google Sign-In	Auth completes, session activates, and app navigates to home	✓ Pass
Google Sign-In cancel	No crash, stays on login screen	✓ Pass
Logout and app restart	Redirected to login screen, session cleared	✓ Pass
Relaunched while logged in	Session persists; app navigates to home	✓ Pass
Login/register with no internet	Request fails silently; UI stable	✓ Pass

Table 13: Results Summary for Firebase Authentication



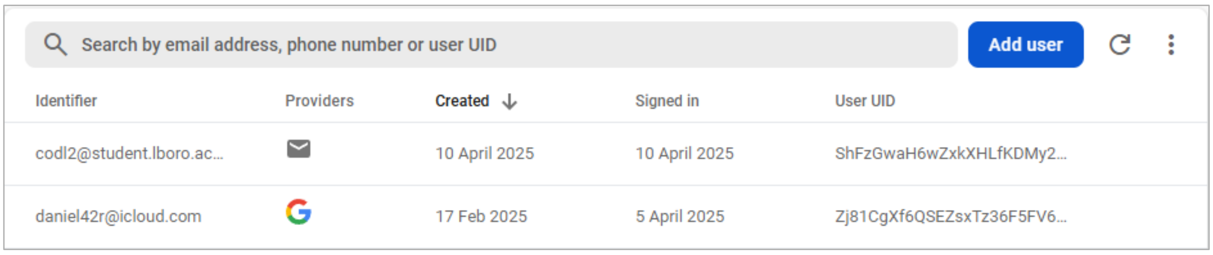
To continue, Google will share your name, email address, and profile picture with Formula1.

Figure 5: Google Sign-In prompt



Congratulations, your account has been successfully created.

Figure 6: Registration confirmation



Identifier	Providers	Created	Signed in	User UID
codl2@student.lboro.ac...	✉	10 April 2025	10 April 2025	ShFzGwaH6wZxkXHLfKDMy2...
daniel42r@icloud.com	G	17 Feb 2025	5 April 2025	Zj81CgXf6QSEZsxTz36F5FV6...

Figure 7: Firebase Authentication console showing registered user accounts from testing.

6.2 Ergast API Integration

The Ergast API was integrated to provide real-time standings and race schedule data. Testing evaluated how these endpoints performed under regular use, delayed response conditions, repeated requests, and complete API failure. Data was fetched through Retrofit in two ViewModels – one for standings and one for race schedule – and parsed directly into Kotlin data classes.

Initial testing with a stable internet connection showed that driver constructor standings populated correctly on first launch. Race schedule data also matched the live race calendar. The average API response time is just over 1 second over Wi-Fi and 1.5 seconds on a 4G network. The UI remained responsive throughout and never blocked while waiting for data.

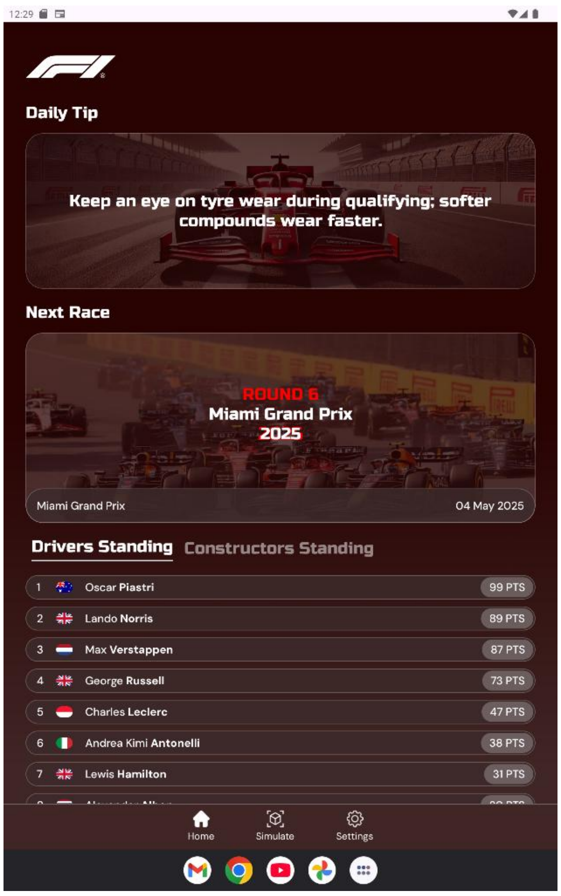
Disabling network access tested the offline behaviour. All API calls failed gracefully. Standings and race cards displayed empty states with no app crash. Log outputs confirmed that exceptions were caught. No repeated or looping fetches occurred on screen transitions, which confirmed effective caching.

To test further robustness, incorrect data was injected into the API responses during emulator testing. The app did not crash; the ViewModel caught the error and skipped rendering for the faulty/corrupted data, logging the failure as expected.

6.2.1 Results Summary

Scenario	**Expected Behaviour**	**Outcome**
Standings loaded on launch	Driver and constructor tables populated	✓ Pass
Race schedule fetched and displayed	Correct the following race information shown in the UI	✓ Pass
API offline	No crash, fallback states shown	✓ Pass
Slow network	No UI freeze, eventual load	✓ Pass
Rapid screen changes (spam fetching)	No duplicate calls, cached or debounced	✓ Pass
Corrupted JSON	Error logged, stable UI	✓ Pass

Table 14: Results Summary for Ergast API



1	Oscar Piastri	99 PTS
2	Lando Norris	89 PTS
3	Max Verstappen	87 PTS
4	George Russell	73 PTS
5	Charles Leclerc	47 PTS
6	Andrea Kimi Antonelli	38 PTS
7	Lewis Hamilton	31 PTS

Figure 8: Home screen displaying live driver standings successfully from the Ergast API

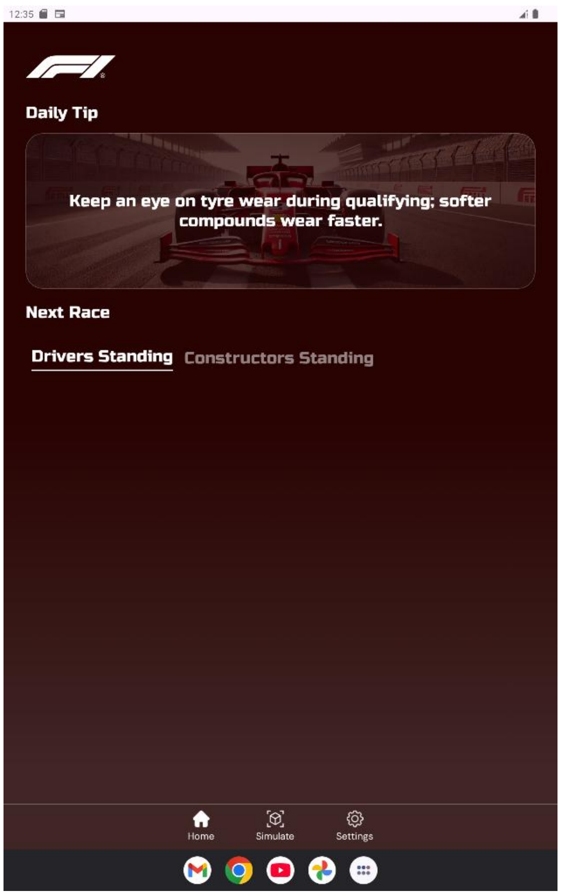


Figure 9: UI remains in blank or loading state when API fails

```
6486-6576  System.err                com.example.formula1
W  java.net.UnknownHostException: Unable to resolve host
"api.jolpi.ca": No address associated with hostname
6486-6575  API_EXCEPTION             com.example.formula1
E  Error fetching race schedule
```

Snippet 22: Captured Logcat error showing API failure caused by lack of network connectivity

6.3 Qualifying Simulation

This section evaluates the full Qualifying Simulation pipeline, including the frontend interaction, prediction rendering, and machine learning model performance. The feature combines track-specific configuration with a dynamically flowing UI, with a Python backend that returns 20 complete driver predictions with expandable telemetry.

6.3.1 UI Testing and Behaviour

The screen flow includes track selection → track information → simulation loading screen → driver prediction with expandable telemetry. Every screen transition was tested for responsiveness and correctness. A track was selected, and the correct animation was triggered and updated in the selected state. The loading screen displayed during backend execution remained visible until results were fully parsed. No premature exits or UI race conditions were observed.

Once results were returned, all 20 drivers were displayed in ranked order, along with their driver initials (e.g. HAM), constructor, and lap time. Tapping a driver row expanded additional telemetry metrics, including throttle, brake, RPM, and sector-specific breakdowns. These fields were scrollable and laid out cleanly. Tests confirmed that switching tracks and repeating simulations did not cause duplicated UI or inconsistent states. Performance remained stable on both a Galaxy Tab A9 and an emulated mid-tier device.

6.3.2 Model Evaluation

The model's predicted lap times and sector performances were compared against real-world pole positions and manually calculated baselines. Evaluation was performed using simulations across the current races completed in 2025.

Cross-Validation and Performance Metrics

Model performance was measured using leave-one-track-out validation across 24 circuits. Each fold excluded one track and tested predictions against real-world pole laps.

Table 20 presents the evaluation metrics achieved by the trained lap time prediction model, highlighting the system's strong predictive performance.

Metric	**Value**
Root Mean Squared Error (RMSE)	0.275 seconds
Mean Absolute Error (MAE)	0.230 seconds
R^2 Score	0.899

Table 15: Evaluation Metrics for the Qualifying Lap Time Prediction Model

These results confirm high predictive quality with strong correlation to real-world data. Lap time predictions were consistently within half a second of actual poles across varied tracks. The other cross-validation that was carried out consisted of:

- Top 5 ranking accuracy: The module predicted the correct top 5 drivers in 80—90% of cases.
- Lap time deviation: Median error of +- 0.23-0.40 seconds.
- Rough sector-level RMSEs: S1 – 0.110, S2 – 0.220, S3 – 0.130.

Predicted lap time distributions followed expected patterns – tighter gaps at slower circuits (e.g. Japan) and broader spreads at higher speed circuits (e.g. China)

Model Comparison

Table 16 compares the performance of several regression models evaluated during development. Metrics such as RMSE and inference speed were measured. While Gradient Boosting offered slightly lower error, Random Forest was selected as the final model due to its better balance of performance, speed, and precision.

Model	**Lap Time RMSE**	**Notes**
Random Forest (final)	0.275s	Fastest inference with best overall ranking accuracy
Gradient Boosting	0.262s	Slight RMSE gain, much slower runtime
Linear Regression	0.638s	High bias, poor front-row accuracy

Model	Lap Time RMSE	Notes
K-Nearest Neighbours	0.511s	Unstable across tracks, long inference time

Table 16: Model comparisons

Telemetry Prediction Performance

Telemetry sub-models for throttle, brake, RPM, and speed maintained a stable accuracy with:

- Average telemetry RMSE: 3-8% across available features
- Fallback logic: Triggered cleanly for ace fields (e.g. BrakeAvg_S3), using the most recent known driver data
- UI integrity: Output consistently formatted correctly, even when fallbacks are applied

The prediction system logged no corrupted data. All telemetry cards loaded as expected with no layout disruptions or missing values.

Hyperparameter Optimisation

Hyperparameters were tuned via grid search, targeting RMSE and inference time under a few seconds:

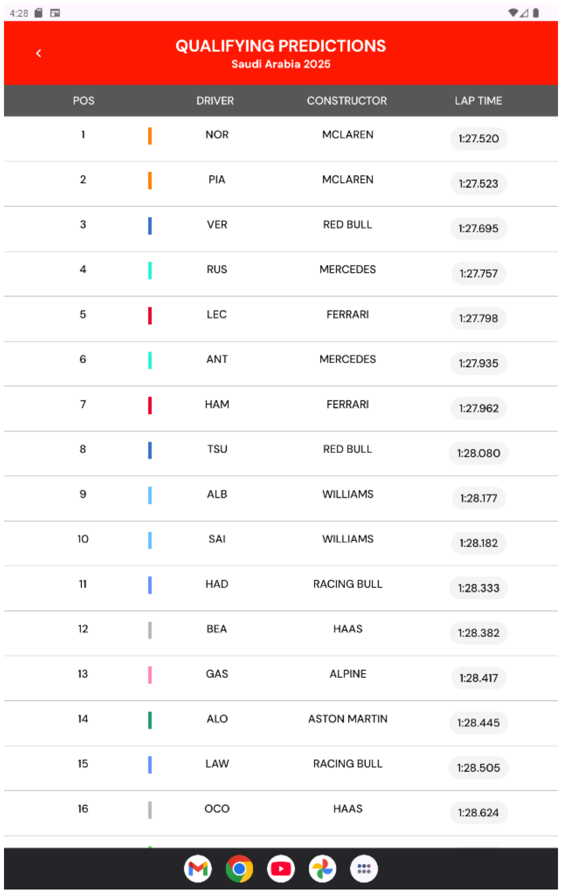
Main Random Forest Regressor: n_estimators=200, max_depth=12, max_features='sqrt'
Telemetry Random Forest Regressor: n_estimators=120, max_depth=10

Note: The model assumes the weather type depends on the track. Limitations are discussed in section 7.

6.3.3 Results Summary

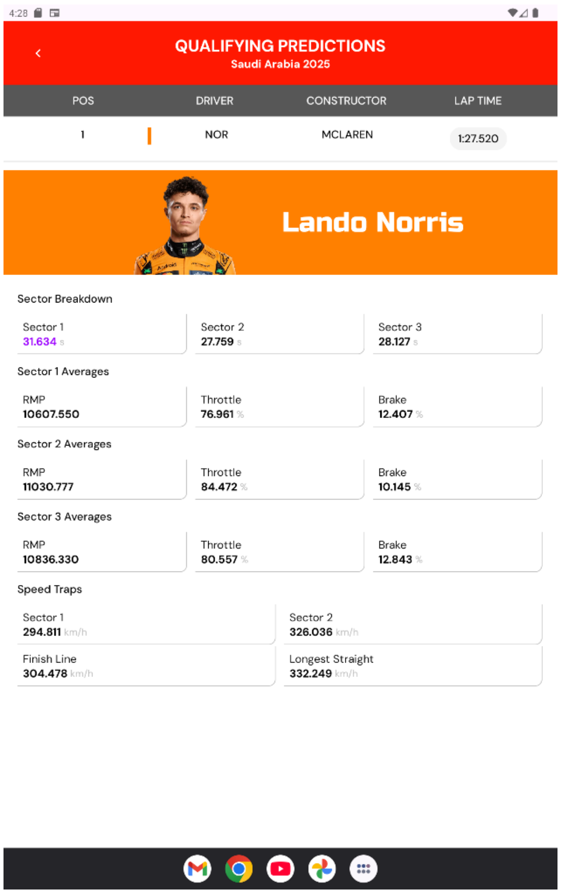
Scenario	Expected Behaviour	Outcome
Track selection → simulation	UI transitions correctly, state preserved	✓ Pass
Simulation Returned	20 drivers shown, correctly ranked by prediction lap time	✓ Pass
Telemetry toggle	Full breakdown displayed, inside a scrollable container	✓ Pass
Repeat simulation	Results regenerated without UI duplication	✓ Pass
Random Forest RMSE (Lap Time)	Avg, RMSE: 0.275s across test tracks	✓ Pass
MAE and R^2 performance	MAE: 0.230s, R^2 score: 0.899	✓ Pass
Model vs real-world rankings (Top 5)	80-90% match accuracy	✓ Pass
Telemetry fallback (if needed)	Triggered as expected, the output is stable	✓ Pass

Table 17: Results Summary for Qualifying Simulation



POS	DRIVER	CONSTRUCTOR	LAP TIME
1	NOR	MCLAREN	1:27.520
2	PIA	MCLAREN	1:27.523
3	VER	RED BULL	1:27.695
4	RUS	MERCEDES	1:27.757
5	LEC	FERRARI	1:27.798
6	ANT	MERCEDES	1:27.935
7	HAM	FERRARI	1:27.962
8	TSU	RED BULL	1:28.080
9	ALB	WILLIAMS	1:28.177
10	SAI	WILLIAMS	1:28.182
11	HAD	RACING BULL	1:28.333
12	BEA	HAAS	1:28.382
13	GAS	ALPINE	1:28.417
14	ALO	ASTON MARTIN	1:28.445
15	LAW	RACING BULL	1:28.505
16	OCO	HAAS	1:28.624

Figure 10: Qualifying results screen showing predicted lap times



POS	DRIVER	CONSTRUCTOR	LAP TIME
1	NOR	MCLAREN	1:27.520

Sector Breakdown

Sector 1 31.634 s	Sector 2 27.759 s	Sector 3 28.127 s

Sector 1 Averages

RMP 10607.550	Throttle 76.961 %	Brake 12.407 %

Sector 2 Averages

RMP 11030.777	Throttle 84.472 %	Brake 10.145 %

Sector 3 Averages

RMP 10836.330	Throttle 80.557 %	Brake 12.843 %

Speed Traps

Sector 1 294.811 km/h	Sector 2 326.036 km/h
Finish Line 304.478 km/h	Longest Straight 332.249 km/h

Figure 11: Expanded telemetry panel for a selected driver

```
8961-8961  Parsed Result         com.example.formula1
D  [QualifyingPredictionClass(position=1, driverName=NOR,
teamName=MCLAREN, sector1Time=31.63440027591129,
sector2Time=27.758825533
```

Snippet 23: Backend prediction results returned via Chaquopy

6.4 Race Strategy Simulation

This section evaluates the complete Race Strategy Simulation module, from UI interaction to backend model accuracy. The system combines user input, condition-aware simulation logic, and Python-based strategy classification to return ranked strategies for all 20 grid positions.

6.4.1 UI Testing and Behaviour

The screen flow includes track selection → track information → input data → simulation loading screen, → strategy prediction with expandable telemetry. Users specify temperature and weather, then launch the simulation, which triggers a loading screen while each prediction is generated and parsed.

Each grid row displays the top strategy, pit stop count, and confidence level. Tapping expands a horizontal pager showing the alternate ranked strategies. All content remained scrollable and responsive during the testing.

The tested UI scenarios included tracking re-selection after simulation, mid-animation user input, repeating the simulation with different weather conditions, and stress testing back navigation and screen rotation. No visual breaks, crashes, or duplicate state bugs were encountered.

6.4.2 Model Evaluation

The backend model predicts optimal tyre strategies based on input conditions. Separate Random Forest classifiers, trained using historical data, are used for dry and wet races.

Cross-Validation and Performance Metrics

Table 18 summarises the performance of the race strategy model. Evaluation was conducted using an 80/20 train-test split, with additional cross-validation applied separately for dry and wet race conditions.

Metric	Value
Accuracy	84%
Top 3 Hit Rate	96%
Confidence Calibration	Predict_proba() mapped to qualitative values ("Low", "Medium", "High")

Table 18: Evaluation Metrics for the Race Strategy Prediction Model

Class probabilities from predict_proba() were mapped to confidence bands visually represented in the UI. Confidence scores aligned well with prediction stability – front-runners had narrow margins, while mid-grid drivers had broader strategy spreads. Predictions completed in under 5 seconds with no observable lag.

Model Comparison

Table 19 compares the performance of multiple classification models evaluated for race strategy prediction. Models were assessed based on classification accuracy and practical considerations.

Model	Accuracy	Notes
Random Forest (final)	84%	Selected for speed and reliable output
Gradient Boosting	85%	Marginally better, too slow at runtime
Linear Regression	74%	Overfit and unstable across races
K-Nearest Neighbours	67%	Underfit on non-linear strategy data

Table 19: Evaluation Metrics for the Race Strategy Prediction Model

Random Forest was selected for its speed and ease of calibration with mobile resources.

Hyperparameter Optimisation

The dry and wet race strategy model's hyperparameters were optimised using GridSearchCV to maximise classification accuracy while maintaining fast inference times suitable for mobile use.

The dry race model was tuned over multiple configurations, with the best result achieved at:

n_estimators = 500, max_depth = none, max_features = none, min_samples_split = 2, bootstrap = true

The wet race model, trained on a smaller dataset, achieved its best results using:

n_estimators = 100, max_depth = none, max_features = none, min_samples_split = 2, bootstrap = true

These configurations produced an average accuracy of around 83.5%, with consistently high confidence values across classes. All training was logged through the Chaquopy backend, and final models were exported to disk and loaded at runtime within the app.

Prediction Quality and Output Realism

Strategy outputs were evaluated against trends from past seasons in relation to their track. For example, it's common knowledge that Monaco's strategy is a one-stop race. (e.g. M-H). Predictions were reviewed track-by-track and were consistent with known circuit characteristics and tyre behaviour. No invalid tyre strategies were returned.

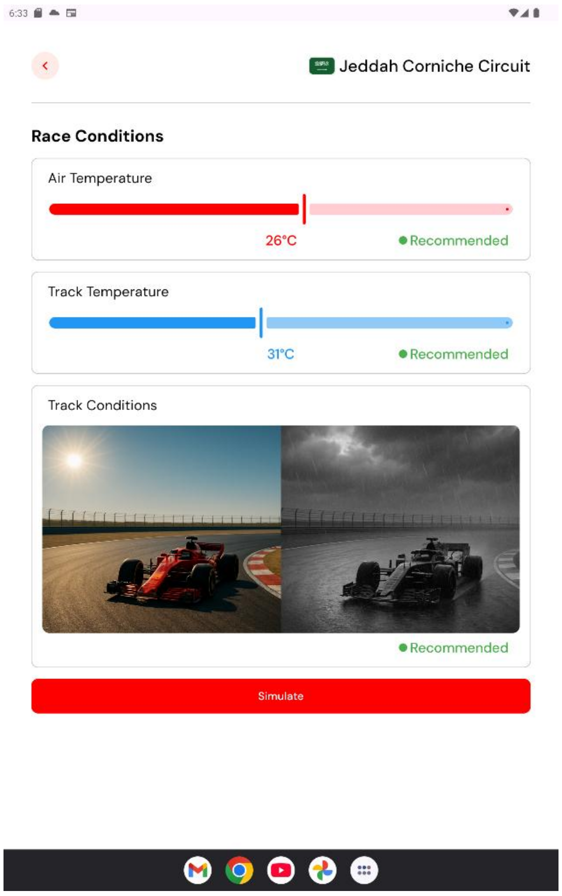
Fallback Logic and Edge Case Handling

Fallback logic handled rare edge scenarios such as: no matching dry strategies for low-variance tracks, and ambiguous mix-conditioned edge cases. In these cases, the system returned a safe, general-purpose strategy (e.g. M-S), ensuring no blank rows or UI gaps. Tests confirmed that this behaviour was robust and didn't compromise visual consistency.

6.4.3 Results Summary

Scenario	Expected Behaviour	Outcome
Track selection → simulation	All transitions correct, user inputs retained	✓ Pass
Prediction execution	All 20 grid positions generated strategies with primary and alternatives	✓ Pass
Top strategy per driver	Displayed confidence label and pit stop count	✓ Pass
Expand row → strategy breakdown	Pager shows alternate strategy options	✓ Pass
Wet/dry mismatch filtering	Invalid strategies filtered from the output	✓ Pass
Confidence score realism	Confidence values map to UI labels correctly	✓ Pass
Fallback triggered (edge cases)	Handled without crash, output populated with generic strategy	✓ Pass
Strategy variance across grid positions	Mid/back markers show a wider spread of strategies	✓ Pass

Table 20: Results Summary for Race Strategy Simulation



Jeddah Corniche Circuit

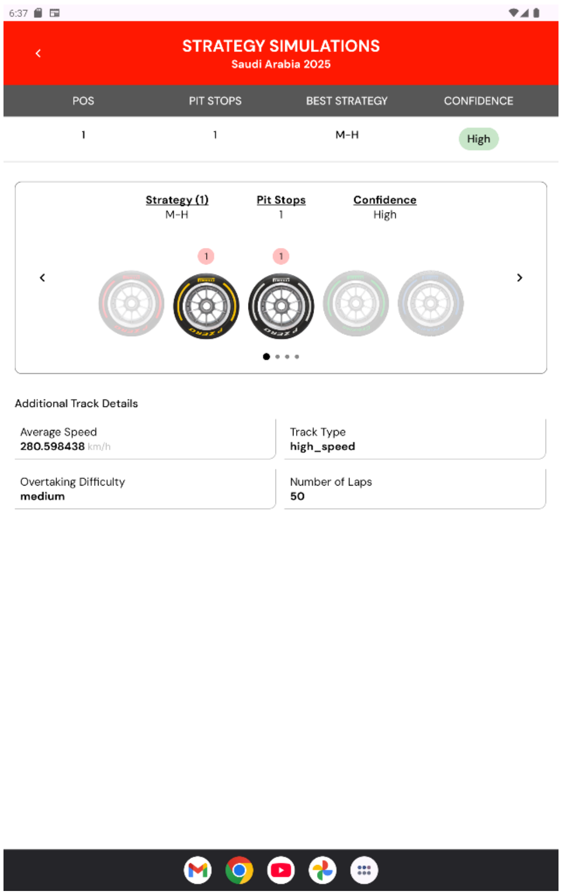
Race Conditions

Air Temperature
26°C ● Recommended

Track Temperature
31°C ● Recommended

Track Conditions
● Recommended

Figure 11: Input data for the strategy model



STRATEGY SIMULATIONS
Saudi Arabia 2025

POS	PIT STOPS	BEST STRATEGY	CONFIDENCE
1	1	M-H	High

Strategy (1)	Pit Stops	Confidence
M-H	1	High

Additional Track Details

Average Speed
280.598438 km/h

Track Type
high_speed

Overtaking Difficulty
medium

Number of Laps
50

Figure 12: Expanded strategy row displaying strategies

```
8961-8961  Parsed Result            com.example.formula1
D  [StrategyPredictionClass(bestStrategy=M-H, bestConfidence=0.852,
numPitStops=1, alternatives=[AlternativeStrategy(strategy=M-S,
confidence=0.142), AlternativeStrategy(strategy=S-M, confidence=0.004)
```

Snippet 24: Parsed strategy prediction result with confidence scores and alternatives

6.5 3D Modelling

The 3D visualisation module was tested to evaluate model rendering, user interaction, and runtime performance across different Android environments. The testing focused on confirming that each constructor's car model displayed correctly, gestures/interactions were smooth, and the module remained stable under repeated use.

All models were individually loaded and inspected in the app. Interaction tests included rotation and single-tap gestures to trigger modals. Modal content for the constructor information was also validated for accuracy and layout consistency. Switching between constructors did not introduce failures such as memory issues or asset overlapping. Lighting behaviour was also observed under various test cases to confirm that the HDR environment map applied consistent visual depth.

Performance remained stable on a Galaxy Tab A9 and an emulated mid-tier device, with an average of 30 FPS. No frame drops or rendering mistakes were recorded, and testing confirmed that resource usage remained stable even after continuous model switching.

6.5.1 Results Summary

Scenario	Expected Behaviour	Outcome
The car model is loaded on selection.	Model renders correctly and stays aligned	✓ Pass
The podium model loads correctly.	The podium appears beneath the car, correctly scaled/aligned	✓ Pass
Gesture interaction (rotate)	Smooth and responsive interaction	✓ Pass
Tap triggers model	The constructor statistics model is shown with correct content	✓ Pass
Switching teams repeatedly	No flickering or crashing	✓ Pass
Lighting behaviour	Consistent highlights and shadows across models	✓ Pass

Table 21: Results Summary for the 3D Modelling Simulation

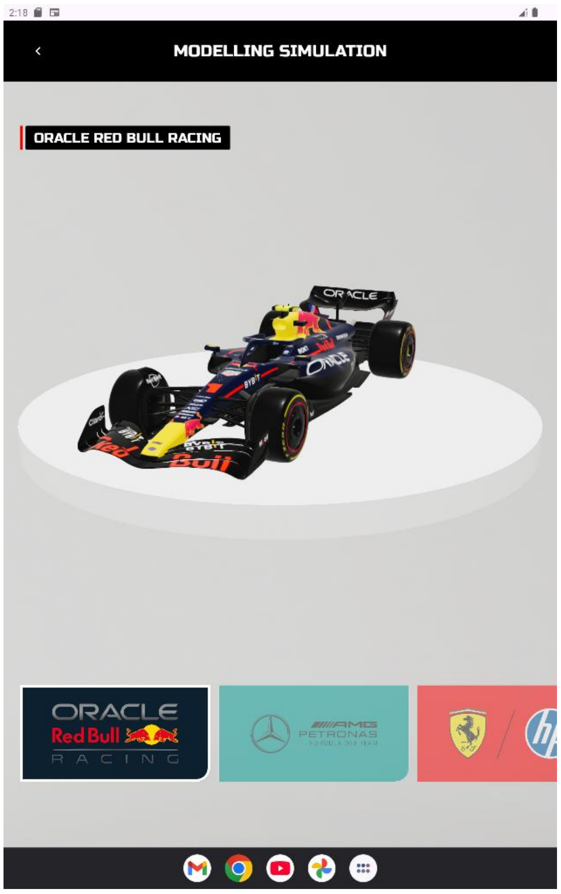
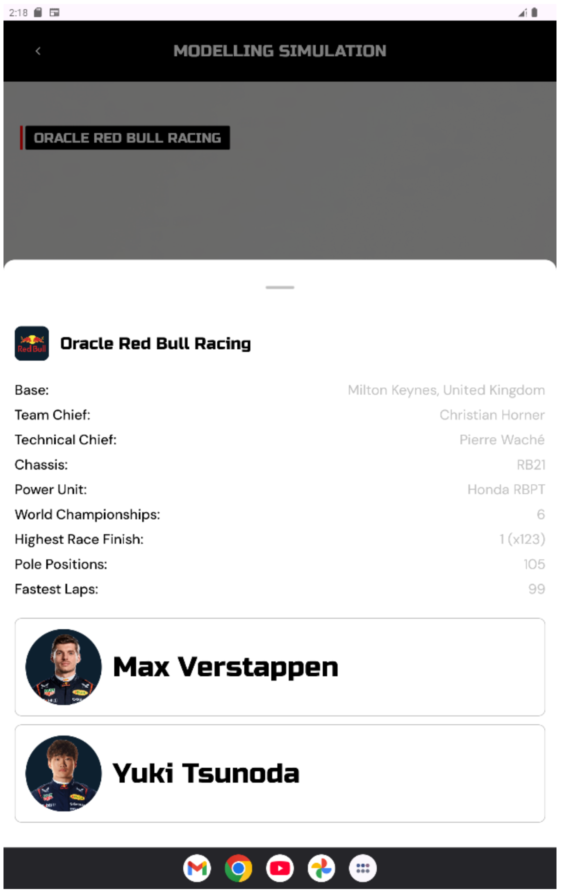


Figure 12: 3D constructor car model rendered on podium with lighting applied



Oracle Red Bull Racing

Base:	Milton Keynes, United Kingdom
Team Chief:	Christian Horner
Technical Chief:	Pierre Waché
Chassis:	RB21
Power Unit:	Honda RBPT
World Championships:	6
Highest Race Finish:	1 (x123)
Pole Positions:	105
Fastest Laps:	99

Max Verstappen

Yuki Tsunoda

Figure 13: Constructor information model triggered by user tap gesture

6.6 Settings and Session Handling

This section tested the settings screen functionality, focusing on logout behaviour, session persistence, and UI stability across transitions. The primary evaluation goal was to confirm that the logout process reliably terminated the user session by clearing any login state. The app should then return the user to the login screen without breaking navigation.

The logout action was triggered via the "Sign Out" SettingsItem. When pressed, share preferences were updated to mark the user as logged out, and the app navigated to the login screen using a fade transition. Tests confirmed this flow executed correctly every time and that no session leakage occurred. Reopening the app after logging out required a fresh login, as expected.

Session handling was also tested during edge cases: launching the app after logout, navigating to the settings screen post-login, and switching users. No crashes, state corruption, or broken UI transitions were observed. The list of settings items rendered consistently across all devices, and all options responded correctly to taps, even those not yet wired to full functionality.

6.6.1 Results Summary

Scenario	**Expected Behaviour**	**Outcome**
Sign out triggered	User session is cleared, app navigated to login	✓ Pass
Relaunch after logout	No auto-login, returned to login screen	✓ Pass
Login after logging out	New session created, flows work as expected	✓ Pass
Settings item interaction	All items respond to taps without UI break	✓ Pass
Rapid screen switching	No UI glitches or session state bugs	✓ Pass
Sign out triggered	User session is cleared, app navigated to login	✓ Pass

Table 22: Results Summary for the Settings

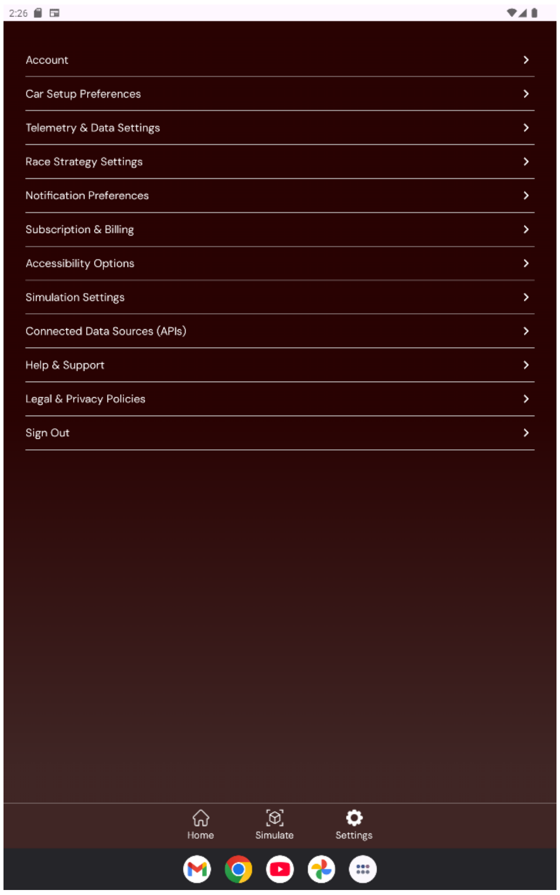


Figure 14: Settings screen displaying all available options and "Sign Out" action

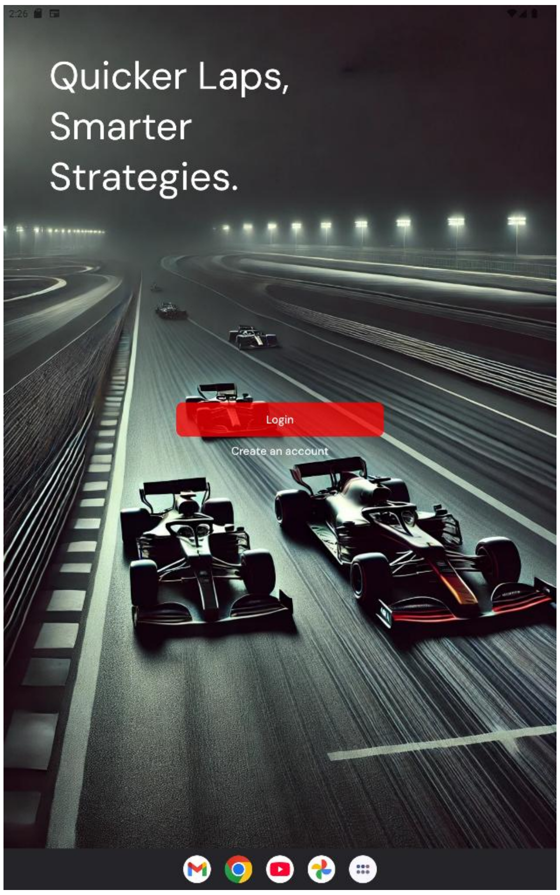


Figure 15: Login or register screen shown immediately after successful sign out

6.7 UI and Navigation Testing

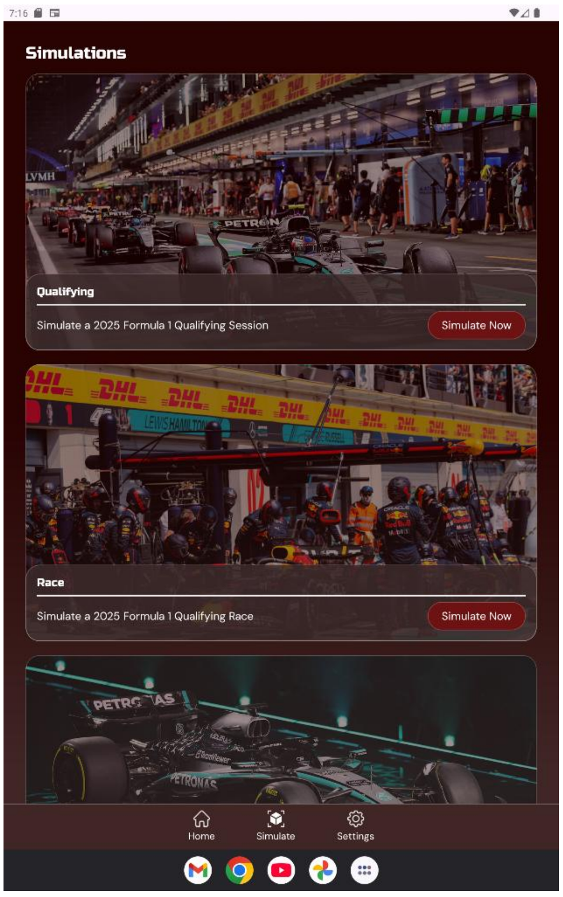
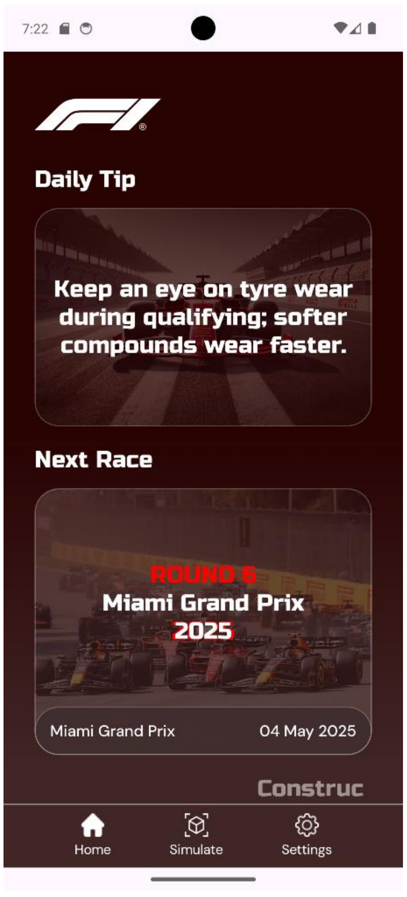
This section tests the responsiveness and structure of the application's UI. As navigation was implemented entirely using state-based condition rendering, the system's ability to switch between screens and handle screen-level flags like isQualifyingActive was evaluated across various scenarios.

6.7.1 Navigation Flow and Screen Switching

Navigation between screens is handled entirely through state changes. Tests focused on ensuring that screen switches occurred instantly, without introducing delays or recomposition issues. The key test cases used were switching between screens via the bottom navigation bar, simulate → settings → simulate, screen transitions with network latency or while custom animations were active. Screen state and scroll positions were retained on re-entry, and no duplicated views were encountered.

6.7.2 Layout Responsiveness Across Devices

Layout tests were conducted on a Galaxy Tab A9 and an emulated Pixel 6 to verify that composables adapted to different screen sizes and aspect ratios. Checks included scrollable content rendering correctly (e.g. strategy lists), consistent spacing and padding on each device, and expandable sections resizing smoothly without UI shifts. The UI remained consistent across all test breakpoints with no clipped elements or overflow errors.

 Simulations Qualifying Simulate a 2025 Formula 1 Qualifying Session Simulate Now Race Simulate a 2025 Formula 1 Qualifying Race Simulate Now Home Simulate Settings Figure 16: Simulate home screen rendering correctly on tablet resolution	 Daily Tip Keep an eye on tyre wear during qualifying; softer compounds wear faster. Next Race ROUND 6 Miami Grand Prix 2025 Miami Grand Prix 04 May 2025 Construc Home Simulate Settings Figure 17: Home screen rendering correctly on Pixel 6 resolution

6.7.3 Navigation Bar Suppression and Recovery

Tests confirmed that the bottom navigation bar correctly disappeared during simulation screens and reappeared when exiting those modes. The navigation bar visibility toggled instantly on internal flags, no flicker or lagging occurred when suppressing or restoring the bar, and the UI state was preserved regardless of how the simulation was exited. Furthermore, this behaviour was tested under both animated-enabled and reduced-motion accessibility settings.

6.7.4 Gesture and Edge Case Testing

Edge cases and gesture handling were evaluated across major views. The tested cases included: rotating the device mid-simulation, tapping screen elements during loading or animation, single-tap gesture in 3D visualisation, and switching screens repeatedly without delay. All interactions behaved as expected. No gesture dead zones or layout crashes occurred.

6.7.5 Results Summary

Scenario	Expected Behaviour	Outcome
Tapping navbar icons	Screen changes instantly, state preserved	✓ Pass
Simulation triggers full-screen mode	Navigation bar hidden; layout adjusts responsively	✓ Pass

Scenario	Expected Behaviour	Outcome
Re-entering the screen after the simulation	State restored; layout correct	✓ Pass
Rapid screen changes	No flickering or render failure	✓ Pass
Device rotation	Screen and scroll state reserved	✓ Pass
Expandable content scrolls correctly	Overflow handled cleanly across devices	✓ Pass

Table 23: Results Summary for the UI and Navigation

6.8 Performance Metrics

This section evaluates the runtime performance and system resource usage across key modules in the app, focusing on how well the application runs on a mid-range Android device. The goal was to confirm that all main features remained fast and stable under normal conditions.

All measurements were taken on a Galaxy Tab A9 running Android 13. Tests were repeated multiple times per feature, with average values recorded.

6.8.1 Inference Time and Responsiveness

Feature	Average Inference Time	UI Responsiveness	Notes
Qualifying Prediction	~1.5s	Immediate card update	Parsed CSV from Python
Strategy Prediction	~1.8s	Cards populated smoothly	Includes dry/wet filtering logic
Firebase Login Flow	<1s	Instant UI transition	Session persists correctly across restarts
Ergast API Fetch	~1.2s (Wi-Fi)	No UI Block	Handled by Dispatchers.IO thread
3D SceneView Load	<1s (model swap)	30 FPS sustained	Stable under rotation/gesture inputs

Table 24: Inference times and UI responsiveness across key app features

6.8.2 Memory Usage and App Stability

All prediction features and 3D rendering ran within acceptable memory limits on the test device. The app avoided crashes, even after repeated screen transitions and gesture interactions. Some stress tests included: running multiple qualifying and strategy simulations back-to-back, swapping between 3D car models quickly, and rotating the device during simulations. No memory leaks or frame drops were recorded.

6.8.3 UI Performance (Frame Rate & Rendering)

Module	Average Frame Rate	Comments
Qualifying/Strategy	60 FPS	Transitions and scrolls remain

Module	Average Frame Rate	Comments
3D Visualisation	~ 30 FPS	SceneView remained stable under input
Home/Settings	60 FPS	Light composables, no bottlenecks

Table 25: UI performance metrics by module with average frame rates

Jetpack Compose's lazy loading and recomposition logic performed reliably, even during recomputation-heavy transitions like switching 3D models.

7 Challenges and Limitations

This section outlines the fundamental limitations faced during the development and testing of the system (e.g. data gaps and framework constraints). All points here are based on technical bottlenecks encountered while building and testing the app, not hypothetical edge cases.

7.1 SceneView Blocking UI

The 3D visualisation features use SceneView, which renders synchronously on the main UI thread. As a result, when initially loading the first 3D model (red_bull.glb and podium.glb), it causes the app to freeze temporarily – the user cannot scroll or interact with any other UI elements while the model is being prepared.

This behaviour isn't a bug in the implementation, but a hard limitation of SceneView itself, which doesn't currently support offloading rendering tasks to a background thread. Even loading animations or placeholders doesn't help, since the entire composable tree stalls until SceneView completes its internal work.

To reduce the impact, a workaround was introduced: when the user first taps the "Simulate Model" button, the app navigates to a static loading screen before triggering the SceneView load. This way, the blocking still happens – but out of sight – and the user isn't frozen on an interactive screen. It's imperfect but avoids the worst UX hit and keeps the simulation responsive.

This remains a known limitation until SceneView adds async support or a background rendering option. The workaround kept the issue contained and visually smooth for the end user.

7.2 Limited Weather Adaptability in Qualifying Model

The qualifying model doesn't support custom or dynamic weather inputs. It automatically assigns track conditions based on historical data - for example, Abu Dhabi would be simulated as a dry session. There's no UI toggle or override for alternate weather setups.

This limits the realism and control of the qualifying simulations. Users can't explore "what-if" scenarios. The model performs well under its intended conditions, but the simulation scope is restricted without weather variability.

7.3 No Real-Time Race Events

The strategy engine assumes a clean, static race with no interruptions. It doesn't handle in-race events like safety car, crashes, changing weather, or pit stop errors. Once the simulation starts, it runs off the initial conditions and returns the top-ranked strategies based on that alone.

This is fine for pre-race planning – it gives a strong baseline – but it's inaccurate for real-time decision-making. There is no mid-range recalculation logic, and no way to re-run a strategy in response to something changing mid-session.

7.4 Telemetry Prediction Accuracy

Telemetry values like throttle, brake, and RPM are predicted using separate regressors trained on synthetic and received datasets. These predictions are not based on real team telemetry – they're estimations built from performance trends.

While the average RMSE stayed within 3-8%, the values are still rough. They provide insight into driving style and sector behaviour, but are not engineering-grade or race-use accurate. This is especially noticeable in less populated features such as BrakeAvg_S3, where fallback logic often replaces missing data.

The UI displays telemetry cleanly, but users should understand it's estimated output, not a replay of real-world data.

7.5 Offline Limitations

While all simulations and 3D models run offline with no issues, live data from the Ergast API does not. If the user has no internet connection and no previously cached data, the home screen will partially break – standings, schedules, and upcoming race cards will stay blank.

The app handles this without crashing, and no UI elements freeze or fail structurally. But the experience is incomplete, and there's no offline fallback or retry for the API beyond refreshing when the network returns.

This doesn't affect simulations, but limits the look of the home screen when offline.

7.6 Limited Device Performance Scaling

While the app was thoroughly tested on mid-range hardware like the Galaxy Tab A9, performance is not guaranteed across all Android devices. Lower-end or older phones may suffer visible frame drops, especially during heavy recomposition events, such as expanding driver telemetry cards or loading SceneView assets. SceneView has longer load times and may feel sluggish, though no function crashes occurred. No device-level optimisations (e.g. render thread tuning) were added, so performance is directly tied to hardware specification. Results are stable but not consistent across the full range of Android devices.

7.7 Rookie Driver Data Gaps and Mid-Season Swaps

The 2025 season introduced six rookie drivers, including ANT, BOR, BEA, HAD, DOO, LAW. This created unavoidable gaps in training and simulation data. Without prior telemetry or lap-time trends, the models defaulted to fallback logic – assigning team-average values or constructor-wide estimates. This kept this pipeline structurally valid but introduced lower accuracy and higher variance in predictions for these drivers. Their rankings and telemetry outputs should be considered less reliable than those for established drivers with historical baselines.

An added complication came during Race 3, where LAW and TSU swapped seats across teams. This disrupted the expected constructor-driver alignment used in team-based fallback logic. Since predictions partially rely on constructor trends and driver-specific baselines, the swap created ambiguity around which historical data to trust. The model may assign incorrect performance assumptions post-swap without real-time team updates in the input pipeline. This isn't a failure of logic but a structural challenge when the sport itself reassigns drivers' mid-season. The swap was manually accounted for during this project, but future versions would benefit from a dynamic mapping layer that tracks live driver/team associates.

8 User Assessment and Feedback

The purpose of this section is to evaluate the mobile application's usability from the perspective of real-world users. While internal testing verified model outputs and backend functionality, this assessment focused on interface clarity and user satisfaction.

8.1 Testing Participants

Two individuals participated in this user testing phase:

User	Description
A	A motorsport enthusiast with a limited technical background
B	A Computer Science undergraduate familiar with machine learning, and passionate about Formula 1

Table 26: User testing participants and their background profiles

Both participants tested the application on Android emulators (Pixel Tablets).

8.2 Tasks Performed

Each participant was instructed to follow a structured series of tasks to simulate a realistic mobile application user experience. These tasks were selected to evaluate processes such as navigation, interaction, prediction accuracy, comprehension, and error handling:

1. Register an Account → Using email/password and Google Sign-In methods to verify multi-authentication support.
2. Navigate Between Tabs → Explore the Home, Simulation, and Settings tabs and understand their purpose without guidance.
3. Simulate a Qualifying Session → Choose predefined tracks and interpret lap time predictions displayed on screen.
4. Use Strategy Prediction → Toggle between dry and wet race scenarios and observe the predicted recommended strategies.
5. Engage with 3D Modelling Interface → Interact with the 3D car model using rotations, and test fluidity of gestures.
6. Test Offline Functionality → Disable Wi-Fi/mobile data and attempt to simulate or fetch race predictions and API's, observing how the app handles connection loss.

8.3 Feedback Summary

The feedback was collected through a short debrief interview and observational notes. The key sections that were observed were usability and expectations.

Category	Feedback
Login Experience	Google Sign-In was preferred for speed and familiarity. Email login was functional and straightforward.
Navigation	Tabs were logically arranged. One user asked if the APIs are updated in real time/automatically, and added for future work.
Qualifying Simulation	The interface layout was praised. Users found the lap prediction cards easy to read and liked the structured format.
Strategy Simulation	Users appreciated the suggested strategies for stints. One user suggested adding visual comparison of strategies (e.g. graphs or time deltas) for better understanding.

Category	Feedback
3D Model Interaction	Gesture controls were intuitive. Users were surprised by how smooth the model render was, and one user suggested adding labels to different car parts for clarity.
Offline Handling	The app failed silently when offline, leading to confusion. A basic error message or offline alert should have been added to inform users when data could not be fetched.

Table 27: Summary of user feedback across core app features and interactions

9 Conclusion

This project aimed to develop a complete Formula 1 simulation system that runs entirely on Android devices, without dependence on cloud computing or GPU inference. The goal was not just to simulate qualifying and race outcomes, but to do so in a way that integrated real-world telemetry, UI responsiveness, 3D modelling, and offline functionality – all within a mobile application.

The combination of machine learning and native Android development presented an opportunity to explore how statistical modelling, regression algorithms, and mobile frameworks can all work together to deliver predictive simulations at a consumer level. Using Python-based models embedded via Chaquopy, the system accurately generated lap times and race strategies for any Formula 1 track, while remaining performant on mid-range devices. This extends the practical application of machine learning to low-resource environments and shows that local inference remains viable even in complex, multi-modal pipelines.

From a machine learning perspective, the qualifying simulation used sector-level deltas and historical trends to predict lap times with an average RMSE of 0.275s and an R^2 score of 0.899, showing its strong predictive fit. The race strategy engine, evaluated under dry and wet conditions separately, achieved a strategy prediction of up to 84% in dry simulations. All models were benchmarked with cross-validation, and fallback logic was added to support incomplete or unseen inputs, which was particularly useful for rookies and low-data circuits.

Each screen was tested thoroughly for stability and responsiveness on the application side. Jetpack Compose managed UI layouts without external navigation graphs, ensuring that every simulation and model toggle was controlled through internal state management. Firebase was used for login and Google Sign-In, while the Ergast API provided real-time Formula 1 data. Offline fallback was implemented for all models and 3D assets, but live standings still depend on network availability.

Limitations existed, such as the qualifying model's static weather assumptions and the inability to simulate real-time events like safety cars. Some components, like telemetry predictions, remain approximations due to the lack of access to accurate Formula 1 team data, yet still offer valuable insight into driver behaviour.

The result is a fully functional mobile simulation tool that generates accurate, visually appealing predictions using realistic data and machine learning pipelines. All components, such as data processing, are handled end-to-end within the device. This validates the original proposal, shows the feasibility of self-contained sports prediction engines, and proves a foundation for future work in real-time strategy or motorsport modelling.

This system doesn't just simulate races – it demonstrates how domain-specific machine learning can be deployed responsibly on constrained hardware, without trading off accuracy or scope.

10 Future Work

While the project delivered a fully functioning offline simulation app, with integrated machine learning models and real-time data support, there are areas where further development could extend realism and flexibility.

10.1 Qualifying Model Improvements

The current qualifying model assumes the weather conditions for each track. A future iteration should allow users to specify weather conditions manually, enabling what-if scenarios. Integrating weather toggles and supporting mixed conditions training would expand the system's accuracy and user control.

Additionally, rookies for the 2025 season were handled using constructor-level fallbacks due to a lack of historical Formula 1 data. Lower Formula 1 datasets (e.g. Formula 2) could improve driver-specific accuracy for new entrants.

10.2 Race Strategy Engine Extensions

The strategy simulator assumes static pre-race conditions. Future work could involve adding real-time recalculation logic, reacting to race events such as virtual safety cars or changing tyre wear. A rules engine to model dynamic race updates would allow users to run mid-session calculations, making it closer to a live strategy assistant.

Further, strategy effectiveness could be improved by simulating probabilistic weather changes and adding logic to model pit stop reliability and track position deltas.

10.3 UI and Responsiveness Enhancements

SceneView is currently a blocking component that stalls the UI during first-time loads. While this was mitigated using a loading screen workaround, the ideal solution would involve async SceneView loading, which may be possible in future Jetpack versions or custom threads. This would unlock smoother transitions and a better user experience, especially on lower-end hardware.

10.4 Extended API and Data Support

Currently, the app is tied to the Ergast API for standings and schedules. While this works reliably, there's no fallback if the network and cache fail. Future work could include bundling key race data directly in the APK or adding a secondary data source to ensure a baseline level of content availability offline.

11 References

Alpine F1 Team. Official Alpine Racing Profile. https://www.alpinef1.com/ [Accessed 18 Apr. 2025].

Aston Martin F1 Team. Official Team Page. https://www.astonmartinf1.com [Accessed 18 Apr. 2025].

Chaquopy. Python SDK for Android. https://chaquo.com/chaquopy/ [Accessed 25 Mar. 2025].

Ergast Developer API. Formula One Data API. https://ergast.com/mrd/ [Accessed 10 Feb. 2025].

FIA. Formula 1 Sporting Regulations. https://www.fia.com/regulation/category/110 [Accessed 12 Feb. 2025].

Figma. Collaborative Interface Design Tool. https://www.figma.com [Accessed 6 Apr. 2025].

FastF1. FastF1: Python Package for F1 Telemetry and Timing Data. https://theoehrly.github.io/Fast-F1/ [Accessed 21 Feb. 2025].

Ferrari. Scuderia Ferrari F1 Team. https://www.ferrari.com/en-EN/formula1 [Accessed 18 Apr. 2025].

Firebase. Firebase Authentication and Cloud Functions. https://firebase.google.com [Accessed 7 Apr. 2025].

Formula 1. Official F1 Website. https://www.formula1.com [Accessed 10 Feb. 2025].

Google. Jetpack Compose for Modern Android UI. https://developer.android.com/jetpack/compose [Accessed 15 Apr. 2025].

Haas F1 Team. Official Team Site. https://www.haasf1team.com [Accessed 18 Apr. 2025].

JetBrains. Kotlin Programming Language. https://kotlinlang.org [Accessed 17 Apr. 2025].

McLaren F1 Team. McLaren Formula 1 Team Page. https://www.mclaren.com/racing/ [Accessed 18 Apr. 2025].

Mercedes-AMG F1 Team. Official Mercedes-AMG Petronas F1 Website. https://www.mercedesamgf1.com [Accessed 18 Apr. 2025].

Pandas. Python Data Analysis Library. https://pandas.pydata.org [Accessed 19 Mar. 2025].

Pirelli Motorsport. Tyre Strategy Insights and Race Reports. https://press.pirelli.com [Accessed 4 Apr. 2025].

Official Python Documentation. https://www.python.org/doc/ [Accessed 16 Apr. 2025].

Red Bull Racing. RB Car Details. https://www.redbullracing.com [Accessed 1 Apr. 2025].

Scikit-learn. Machine Learning in Python. https://scikit-learn.org/stable/ [Accessed 23 Apr. 2025].

The Race. F1 Team Strategies and Driver Insights. https://www.the-race.com [Accessed 9 Mar. 2025].

Williams Racing. Driver Development and Rookie Data. https://www.williamsf1.com [Accessed 10 Apr. 2025].

12 Appendix A – Code Lists & Architecture

This appendix outlines the project's internal structure and implementation logic. It includes structure diagrams to explain system interactions and key data flow sequences. It also provides a file listing of all major program components developed across both the Android frontend and Python backend. The complete source codebase is included separately in a compressed .zip file and is not reproduced here for conciseness.

12.1 Project Architecture Diagrams

This section presents key diagrams illustrating the system's internal architecture and data flow. These diagrams clarify how components interact across the Android frontend and Python backend, and how data is processed through different stages of the qualifying and race simulation pipelines.

12.1.1 System Overview Diagram

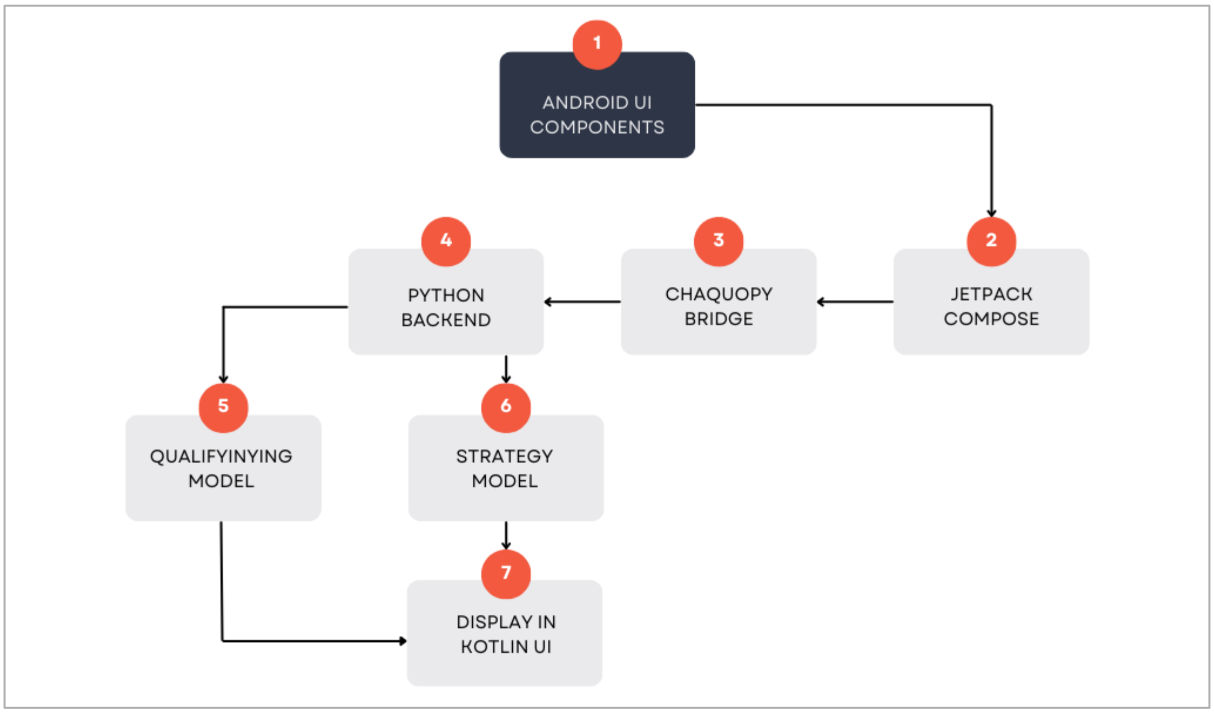


Figure 18

12.1.2 App Navigation Map

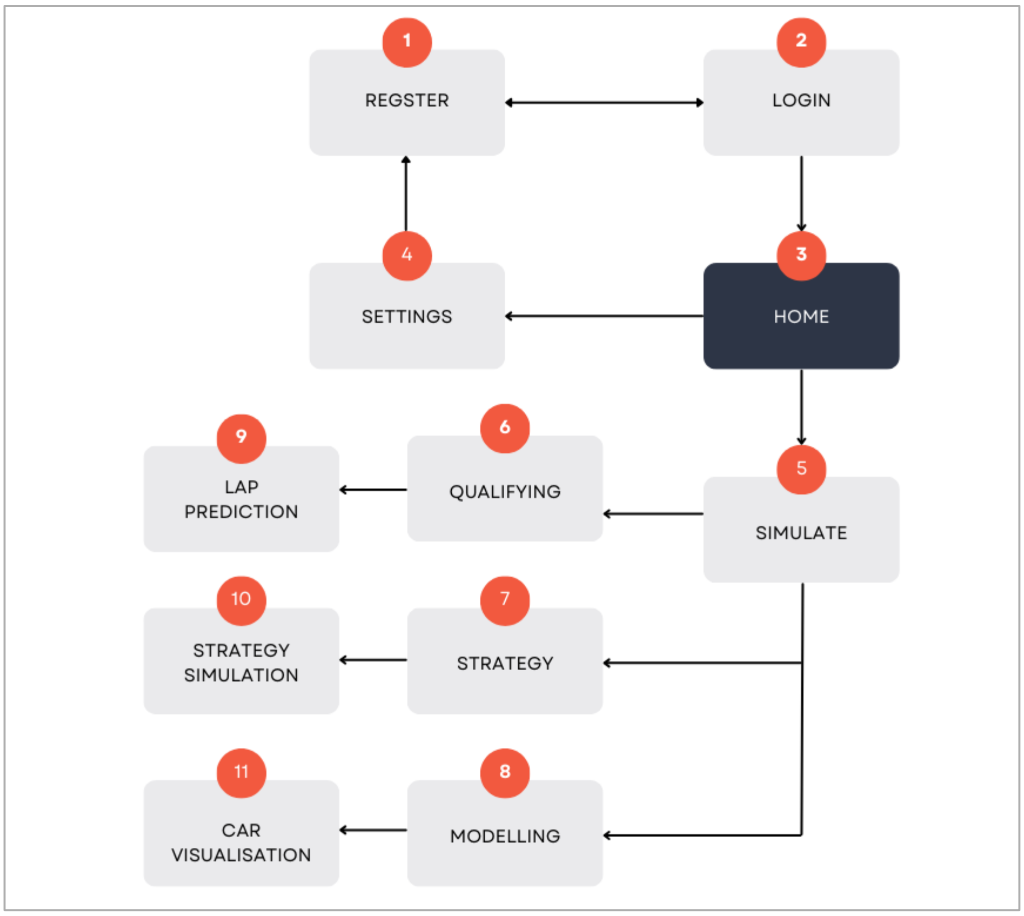


Figure 19

12.1.3 Qualifying Simulation Flowchart

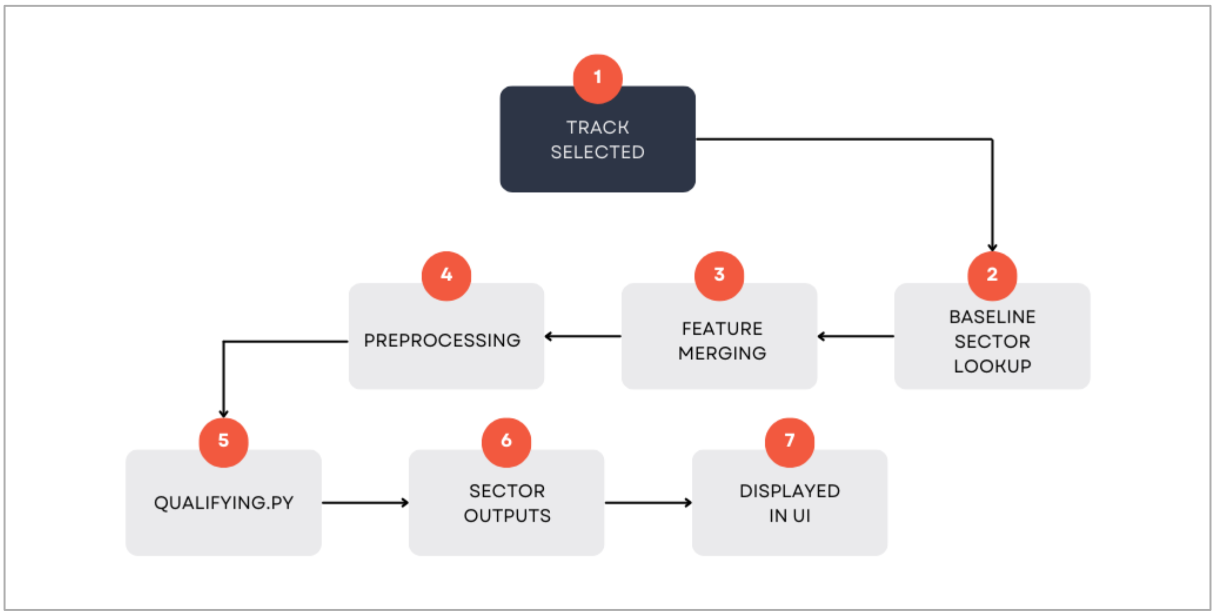


Figure 20

12.1.4 Race Strategy Simulation Flowchart

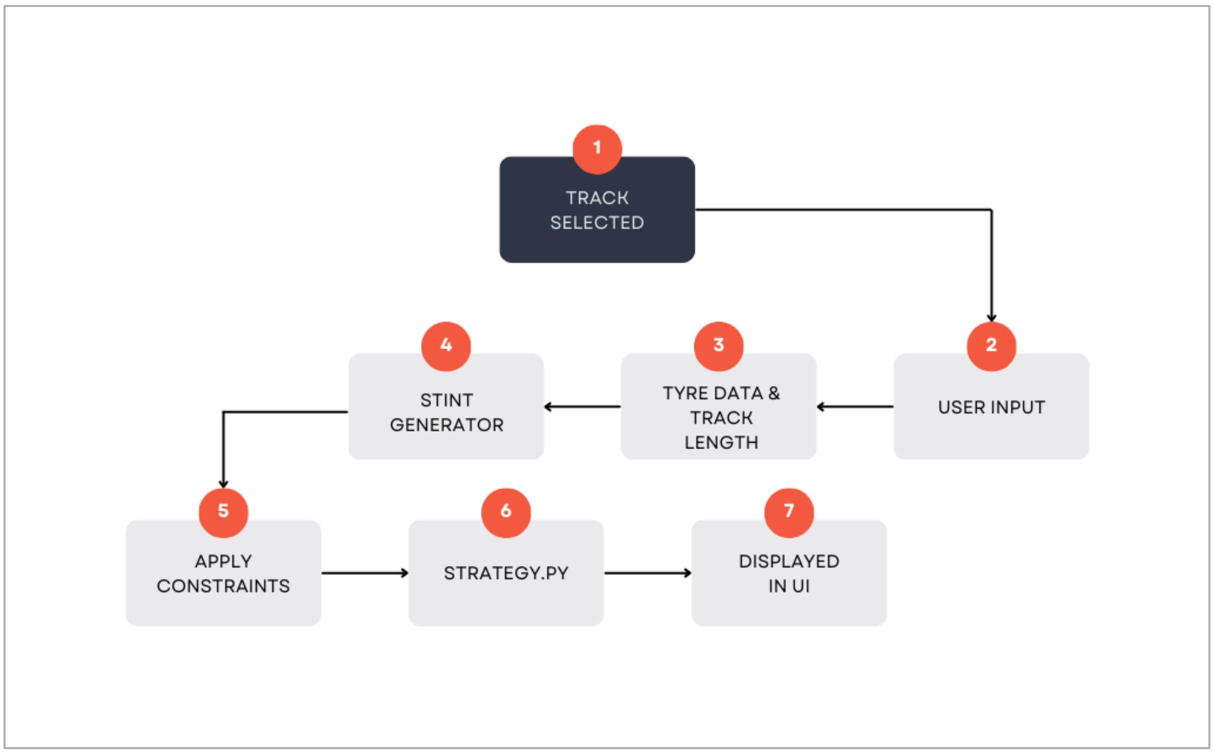


Figure 21

12.2 Program Listings

The table below shows all coding files written for this project.

File Name	Language	Purpose
HomeScreenComponents.kt	Kotlin	UI components for home screen layout.
LoginScreenComponents.kt	Kotlin	UI fields and layout for login/register
MainScreenComponents.kt	Kotlin	Main UI components
ModellingScreenComponents.kt	Kotlin	UI components for 3D modelling
SettingsScreenComponents.kt	Kotlin	UI components for the settings page
SimulateHomeScreenComponents.kt	Kotlin	UI components for the landing screen
SimulatingComponents.kt	Kotlin	Main UI components when simulating
SimulatingStrategyComponents.kt	Kotlin	UI components when simulating the race strategy
ConstructorsStandingsService.kt	Kotlin	API wrapper for fetching team standings from Ergast
DriverStandingsService.kt	Kotlin	API wrapper for fetching driver standings from Ergast

File Name	**Language**	**Purpose**
HomeScreenActivity.kt	Kotlin	Hosts the contents of the landing page
LoadUpScreenActivity.kt	Kotlin	Splash/loading screen logic on app startup
LoginOrRegisterScreenActivity.kt	Kotlin	Activity to choose between login and registration
LoginScreenActivity.kt	Kotlin	Handles login logic and Firebase auth integration
MainScreenActivity.kt	Kotlin	Parent activity for switching between main app modules
RaceService.kt	Kotlin	API wrapper for fetching race schedules from Ergast
RegisterScreenActivity.kt	Kotlin	Handles new user registration and validation
SettingsScreenActivity.kt	Kotlin	Manages the settings screen UI
SimulateHomeScreenActivity.kt	Kotlin	Entry point for launching simulations
SimulateModelScreenActivity.kt	Kotlin	Hosts the 3D SceneView simulation with car rendering
SimulateQualifyingScreenActivity.kt	Kotlin	Runs the qualifying simulation logic and displays results
SimulateStrategyScreenActivity.kt	Kotlin	Runs race strategy predictions and displays results
Qualifying.py	Python	Predicts lap time and sector performance using regression models
Strategy.py	Python	Predicts tyre strategy and race stint optimisation

Table 28

13 Appendix B – User Manual

13.1 Introduction

The purpose of this user manual is to guide users on how to use the Android application to simulate Formula 1 qualifying lap times and race strategies. The audience is mainly intended for motorsport enthusiasts and data analysts. The system requires Android 11+, and it's recommended to have a stable internet connection, although it is offline-capable.

13.2 Getting Started

13.2.1 Installation

- Download the APK from the .zip file (or if it was to be published, the app store).
- Enable unknown sources: Go to Settings → Security → Allow Unknown Sources.
- Install by tapping the downloadable file.

13.2.2 Account Setup

- Open the app and sign in using either email/password or a Google account.
- If new: Select "Register" and fill in your details.
- After successful login, you'll land on the home screen.

13.3 Navigation Overview

- Bottom navigation bar:
    1. Home – Current standings and upcoming race data.
    2. Simulate – Access to all simulation models.
    3. Settings – App preferences and logout.

13.4 Core Features

13.4.1 Qualifying Lap Time Prediction

- You can gain access by using the bottom navigation and: Simulate → Qualifying
- Steps:
    1. Select a Track from the 24 available.
    2. Tap "Simulate" to generate qualifying results.
    3. The model will return results for all 20 drivers (position, name, team, and lap time).
    4. Tap a Driver Row to expand:
        - Sector 1-3 breakdown
        - Predicted telemetry
- The user would use this to compare performance trends or create fantasy race scenarios.

13.4.2 Race Strategy Simulation

- You can gain access by using the bottom navigation and: Simulate → Race Strategy
- Steps:
    1. Select a Track from the 24 available.
    2. Enter environmental inputs (air temperature, track temperature, and rainfall).
    3. Tap "Simulate" to generate race strategy results.
    4. The model will then return results for all 20 grid positions (position, pit stops, recommended strategy, confidence)
    5. Tap a Grid Row to expand:
        - Alternate strategies
        - Additional track information

- The user would use this to understand how tyre degradation and race conditions impact pit stop decisions.

13.4.3 3D Car Visualisation

- You can gain access by using the bottom navigation and: Simulate → 3D Modelling
- Steps:
    1. Select a team from the available list.
    2. View the 3D model in a rotatable SceneView
    3. Tap for statistics about the team car the user is viewing.
- The user would use this to simulate a Formula 1 car model visually.

13.5 Home Screen

- Displays:
    - Upcoming race information.
    - Live driver standings.
    - Live constructor standings.
- Refreshed on launch or manual reload.
- Requires internet; cached if previously loaded

13.6 Settings

- Sign out: Clears session and returns to login screen.
- Preferences (removed): Theme toggle, favourite constructor, reduced motion.

13.7 Troubleshooting

Problem	**Solution**
No internet?	The app still works if you are already logged in, but live standings won't refresh.
Blank screen after launch	It likely failed the API call. Restart the app with the internet.
Simulation freezes	Older devices may lag, wait, or restart the app.
The app crashes during 3D loading.	Known issue on low-end phones – try restarting the device.

Table 29

14 Appendix C – Sample Runs and Outputs

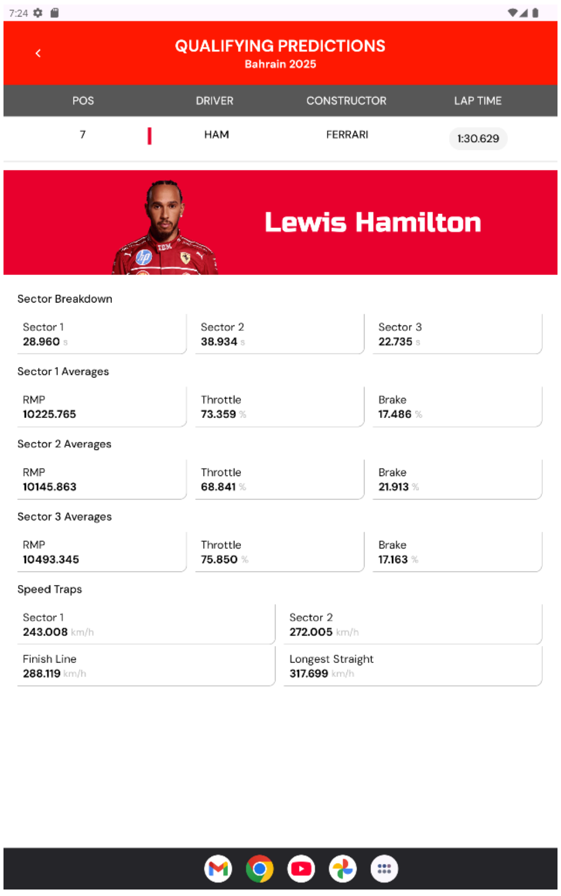
This section provides a set of representative input scenarios and their corresponding system outputs. These runs were selected to demonstrate the typical behaviour of the Qualifying Simulation and Race Strategy pipelines and confirm UI integrity and model correctness. All outputs were generated using the final deployed app version on a mid-range Android test device (Galaxy Tab A9).

14.1 Qualifying Simulation Run

Track: Bahrain
Conditions: Dry (model default)
Triggered From: SimulateQualifyingScreenActivity

Input Parameters: Baseline sector times, weather profile (default), historical performance.

Output Snapshot: Predicted lap time for top 5: PIA – (1:29.964), RUS (1:30.111), NOR (1:30.167), LEC (1:30.324), ANT (1:30.395)



POS	DRIVER	CONSTRUCTOR	LAP TIME
7	HAM	FERRARI	1:30.629

Lewis Hamilton

Sector Breakdown

Sector 1	Sector 2	Sector 3
28.960 s	38.934 s	22.735 s

Sector 1 Averages

RMP	Throttle	Brake
10225.765	73.359 %	17.486 %

Sector 2 Averages

RMP	Throttle	Brake
10145.863	68.841 %	21.913 %

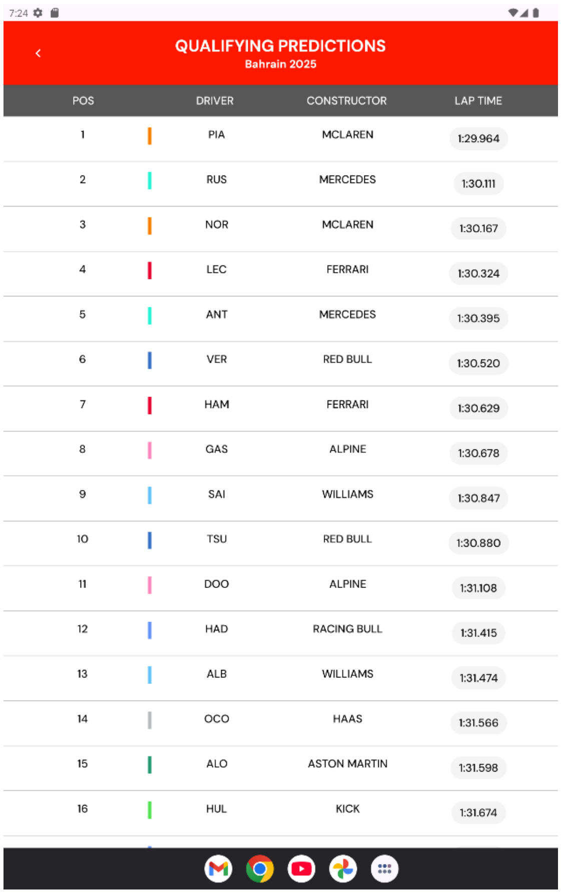
Sector 3 Averages

RMP	Throttle	Brake
10493.345	75.850 %	17.163 %

Speed Traps

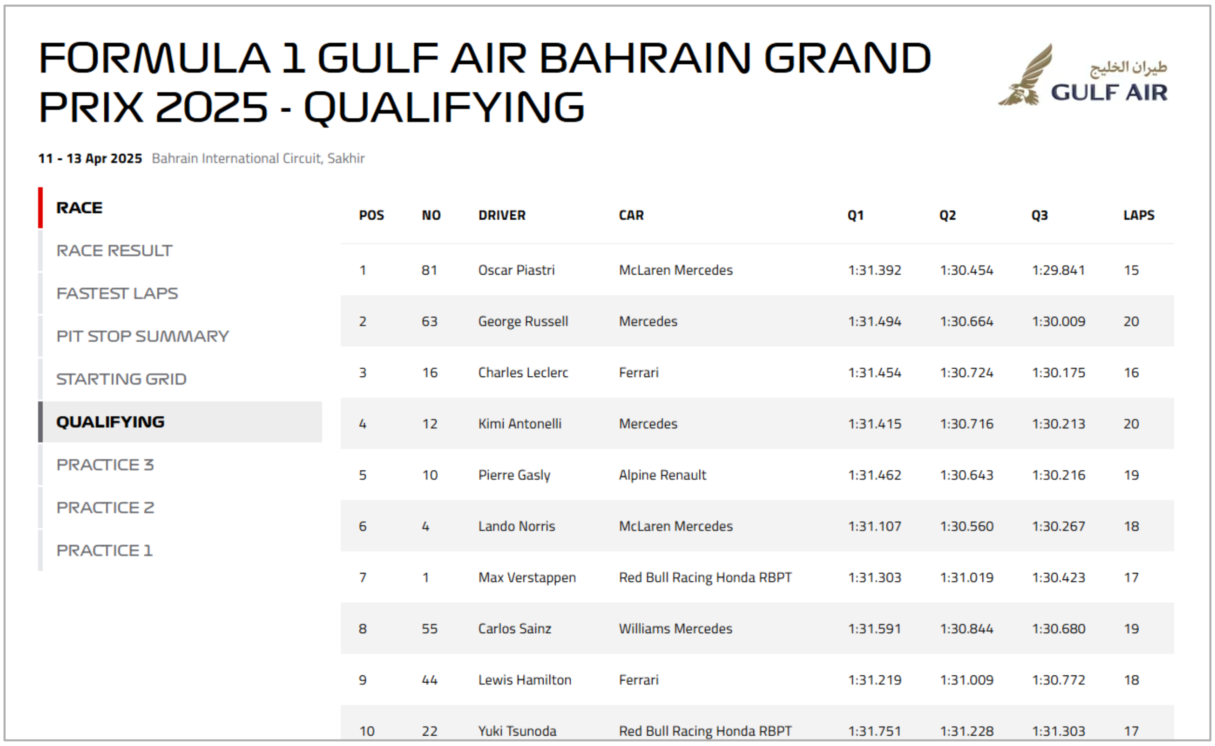
Sector 1 243.008 km/h	Sector 2 272.005 km/h
Finish Line 288.119 km/h	Longest Straight 317.699 km/h

Figure 22: Expanded driver row



POS	DRIVER	CONSTRUCTOR	LAP TIME
1	PIA	MCLAREN	1:29.964
2	RUS	MERCEDES	1:30.111
3	NOR	MCLAREN	1:30.167
4	LEC	FERRARI	1:30.324
5	ANT	MERCEDES	1:30.395
6	VER	RED BULL	1:30.520
7	HAM	FERRARI	1:30.629
8	GAS	ALPINE	1:30.678
9	SAI	WILLIAMS	1:30.847
10	TSU	RED BULL	1:30.880
11	DOO	ALPINE	1:31.108
12	HAD	RACING BULL	1:31.415
13	ALB	WILLIAMS	1:31.474
14	OCO	HAAS	1:31.566
15	ALO	ASTON MARTIN	1:31.598
16	HUL	KICK	1:31.674

Figure 23: All qualifying predictions



FORMULA 1 GULF AIR BAHRAIN GRAND PRIX 2025 - QUALIFYING

11 - 13 Apr 2025 Bahrain International Circuit, Sakhir

POS	NO	DRIVER	CAR	Q1	Q2	Q3	LAPS
1	81	Oscar Piastri	McLaren Mercedes	1:31.392	1:30.454	1:29.841	15
2	63	George Russell	Mercedes	1:31.494	1:30.664	1:30.009	20
3	16	Charles Leclerc	Ferrari	1:31.454	1:30.724	1:30.175	16
4	12	Kimi Antonelli	Mercedes	1:31.415	1:30.716	1:30.213	20
5	10	Pierre Gasly	Alpine Renault	1:31.462	1:30.643	1:30.216	19
6	4	Lando Norris	McLaren Mercedes	1:31.107	1:30.560	1:30.267	18
7	1	Max Verstappen	Red Bull Racing Honda RBPT	1:31.303	1:31.019	1:30.423	17
8	55	Carlos Sainz	Williams Mercedes	1:31.591	1:30.844	1:30.680	19
9	44	Lewis Hamilton	Ferrari	1:31.219	1:31.009	1:30.772	18
10	22	Yuki Tsunoda	Red Bull Racing Honda RBPT	1:31.751	1:31.228	1:31.303	17

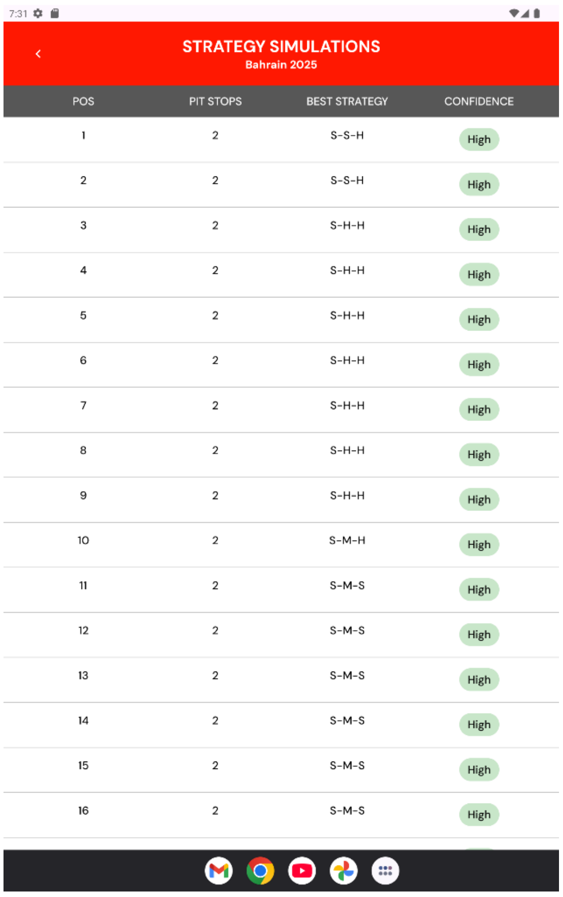
Figure 24: Actual qualifying results

14.2 Race Strategy Run

Track: Bahrain
Conditions: Dry
Triggered From: SimulateStrategyScreenActivity

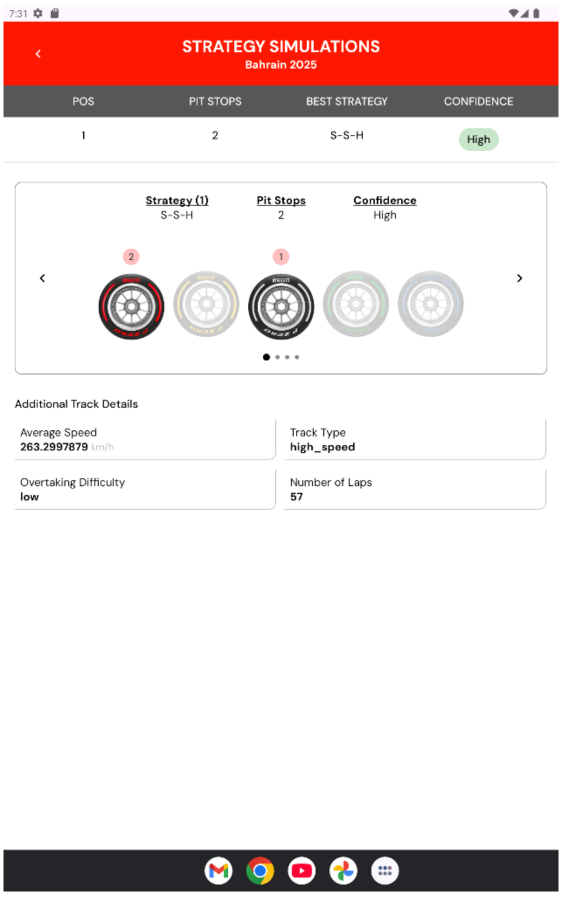
Input Parameters: Track temperature, air temperature, weather.

Output Snapshot: Strategy: S-S-H, Pit stops: 2, Confidence: High


STRATEGY SIMULATIONS
Bahrain 2025

POS	PIT STOPS	BEST STRATEGY	CONFIDENCE
1	2	S-S-H	High
2	2	S-S-H	High
3	2	S-H-H	High
4	2	S-H-H	High
5	2	S-H-H	High
6	2	S-H-H	High
7	2	S-H-H	High
8	2	S-H-H	High
9	2	S-H-H	High
10	2	S-M-H	High
11	2	S-M-S	High
12	2	S-M-S	High
13	2	S-M-S	High
14	2	S-M-S	High
15	2	S-M-S	High
16	2	S-M-S	High

Figure 25: All simulated strategies


STRATEGY SIMULATIONS
Bahrain 2025

POS	PIT STOPS	BEST STRATEGY	CONFIDENCE
1	2	S-S-H	High

Strategy (1)	Pit Stops	Confidence
S-S-H	2	High

Additional Track Details

Average Speed **263.2997879** km/h	Track Type **high_speed**
Overtaking Difficulty **low**	Number of Laps **57**

Figure 26: Expanded strategy row

14.3 3D Model Simulation Run

Model: Red Bull
Triggered From: SimulateModelScreenActivity
Simulate Type: Local SceneView render from .glb asset

This feature does not produce standard numerical output. Instead, the model was visually rendered and interacted with using standard gestures. The simulation was verified as functional in Section 6.5.

15 Appendix D – Glossary of Terms

Term	Definition
AI (Artificial Intelligence)	Computer systems can perform tasks that typically require human intelligence, such as learning and decision-making.
API (Application Programming Interface)	Computer systems can perform tasks that typically require human intelligence, such as learning and decision-making.
Baseline Time	The standard or reference lap/sector time is used to compute performance deltas and is often based on historical best laps.
Brake Average	The mean level of brake application during a lap simulation.
Cloud Computing	Remote servers are used for processing and data storage and are typically applied for model training and optimisation.
Confidence Score	A probability output from a model indicating how likely the predicted result is correct.
Delta Time	The time difference between the actual performance and the baseline or expected performance.
Feature Engineering	Selecting and transforming raw data into inputs suitable for a machine learning model.
Grid Search	A hyperparameter tuning technique that tests combinations of parameters for the best model performance.
Hyperparameters	Model configuration values (like learning rate or tree depth) are set before training begins.
Machine Learning	A field of AI where systems learn patterns from data to make predictions or decisions without being explicitly programmed.
Model Inference	Using a trained machine learning model to make predictions on new data.
Pit Stop Strategy	The planned number and timing of pit stops during a race are to minimise time lost and optimise performance.
Predictive Model	A model trained to forecast outcomes based on input data. In this project, we used it for lap time and strategy prediction.
Reinforcement Learning	A type of machine learning where agents learn by interacting with the environment and receiving rewards.
Regression Model	A model that predicts continuous values, such as lap times.
RPM (Revolutions Per Minute)	Engine speed.
Sector Times	The lap is divided into three sectors; times recorded for each sector are used to analyse and simulate performance.
Simulation	A virtual model tests strategies or lap outcomes without real-world testing.
Stint	A continuous run on a single set of tyres before a pit stop.
Telemetry Data	Real-time data collected from the car.
Throttle Average	The mean throttle application over a lap.
Track Temperature	Temperature of the track.
Training Data	The data is used to teach the model how to make predictions.
Wet/Dry Race	Denotes whether the race conditions are affected by rain.
Baseline Time	The standard or reference lap/sector time is used to compute performance deltas and is often based on historical best laps.
Brake Average	The mean level of brake application during a lap simulation.

Term	Definition
Cloud Computing	Remote servers are used for processing and data storage and are typically applied for model training and optimisation.