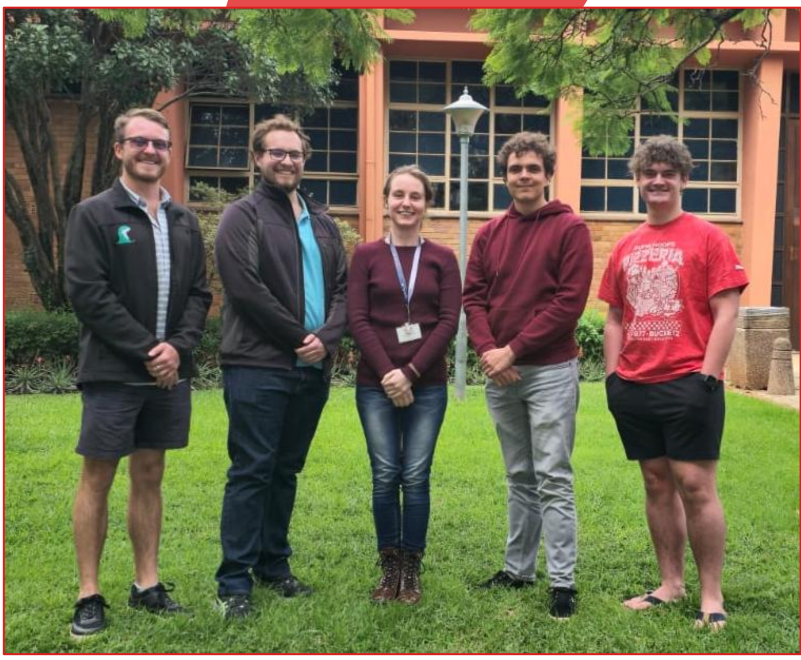
SuperLap Racing Line Optimization System

EPI-USE



Quintessential

Amber Ann Werner [u21457752]

Milan Kruger [u04948123]

Qwinton Knocklein [u21669849]

Sean van der Merwe [u22583387]

Simon van der Merwe [u04576617]

Contents

Introduction 1

User Characteristics 2

User Stories 4

Use Case viii

Service Contracts 7

Track Image Processing 7

Racing Line Optimization 7

AI Training Service 7

Visualization Service 7

User Account Management *(optional)* 8

Lap Time Comparison 8

Requirements 9

Technology Requirements xiii

Functional Requirements 9

Architectural Requirements 11

Quality Requirements xvii

Architectural Pattern xix

Design Patterns 14

Constraints xxiii

Diagrams and Models 21

Architecture Diagram 21

Class Diagram 22

Domain Model 23

Deployment Diagram 24

Manuals 25

Installation Manual 25
Technical Installation Manual 26
User Manual 27
Specifications and Standards 29
Machine Learning Specification 29
API Documentation 30
Coding Standards 31
Testing Policy 35
Testing Scope & Levels 35
Testing Types & Frequency 35
Entry & Exit Criteria 35
Contribution of Teammates 38
Project Manager 38
Amber Werner 38
Backend Developers 38
Qwinton Knocklein 38
Sean van der Merwe 38
Front End Developers 38
Simon van der Merwe 38
Milan Kruger 38
Appendix: Old Versions of SRS iii
Version 1 [26/05/2025] iii

Introduction

There is a growing need for accessible, data-driven training tools in motorsports, especially among students, amateur riders, and enthusiasts who lack access to expensive telemetry systems or real-world testing environments. SuperLap Racing Line Optimization System addresses this need by providing an AI-powered platform that helps superbike riders identify the fastest possible racing line on a racetrack.

The project aims to develop a Reinforcement Learning and Computer Vision-based system that analyses a top-down image of a racetrack, simulates thousands of optimal pathing scenarios, and overlays the ideal racing line on the map. Designed with usability and precision in mind, SuperLap focuses on delivering accurate, performance-enhancing insights in a visually intuitive format, supporting smarter race training without the traditional barriers of cost or access.

User Characteristics

Amateur & Hobbyist Racers

Characteristics:

- **Skill Level:** Novice to intermediate riders.
- **Goals:** Improve lap times, learn optimal racing lines, understand basic track dynamics.
- **Technical Proficiency:** Basic; comfortable using apps but limited technical knowledge.
- **Usage:**
  - Upload 2D track images from local circuits.
  - Use AI-generated racing lines as visual training aids.
  - Compare different racing lines for self-improvement.
- **Motivation:** Affordable alternative to professional coaching and telemetry systems.

Example: A track-day rider at Kyalami Circuit aiming to shave seconds off lap times.

Motorsport Coaches & Instructors

Characteristics:

- **Skill Level:** Advanced (former or current racers).
- **Goals:** Teach optimal racing strategies using AI-generated insights.
- **Technical Proficiency:** Moderate; knowledgeable in racing physics, less so in AI/ML.
- **Usage:**
  - Validate AI-generated racing lines against personal experience.
  - Generate annotated visual materials for student feedback.

    - Compare multiple rider lines for debriefing sessions.

- **Motivation:** Enhance coaching efficiency with data-backed tools.

Example: A racing school instructor using the system to highlight braking points to students.

Sim Racing Enthusiasts

Characteristics:

- **Skill Level:** Varies from casual to competitive sim racers.
- **Goals:** Optimize virtual race performance in games like Assetto Corsa or Gran Turismo.
- **Technical Proficiency:** High; comfortable with mods, data analysis, and telemetry tools.
- **Usage:**
    - Import in-game 2D track maps for AI analysis.
    - Compare AI-generated lines against in-game telemetry data.
    - Share optimized lines with online sim racing communities.
- **Motivation:** Gain a competitive edge in online and league racing.

Example: An iRacing league competitor seeking the ideal Monza racing line.

Professional Racing Teams (Small/Privateer)

Characteristics:

- **Skill Level:** Expert (professional riders, engineers).
- **Goals:** Fine-tune bike setup and validate racing strategies.
- **Technical Proficiency:** High; familiar with AI, telemetry, and vehicle dynamics.
- **Usage:**
    - Cross-reference AI predictions with real telemetry data where available.

   - Test hypothetical scenarios (e.g: wet vs dry racing lines).
   - Integrate with existing telemetry tools via API if supported.
- **Motivation:** Cost-effective supplement to expensive motorsport analytics solutions.

Example: A privateer Moto3 team optimizing cornering lines with limited budget.

Engineering & Motorsport Students

Characteristics:

- **Skill Level:** Academic learners in racing dynamics and AI.
- **Goals:** Study racing line theory, reinforcement learning applications, and vehicle physics.
- **Technical Proficiency:** Medium; some coding and mathematical background.
- **Usage:**
   - Experiment with different AI models (e.g: DQN, PPO).
   - Validate theoretical models against system simulations.
   - Use 2D track data as accessible inputs for research projects.
- **Motivation:** Research, thesis projects, and hands-on learning.

Example: Mechanical engineering student analysing Suzuka's "S-curves" for a thesis.

User Stories

Core User Stories (Functionality & User Experience)

1. As a rider, I want to upload a top-down 2D image of my racetrack so the system can analyse it for optimal racing line suggestions.
2. As a user, I want to view and customize the uploaded image (zoom, pan, annotate) to better understand the data.
3. As a user, I want to see the AI-generated optimal racing line overlaid on the track to compare it with my own strategy.

4. As a motorsport enthusiast, I want the system to simulate multiple racing lines using reinforcement learning so I can evaluate their performance under different conditions.
5. As a rider, I want to compare my recorded lap times with AI-predicted optimal lap times to identify areas for improvement.
6. As a beginner racer, I want simple, actionable guidance (e.g: “brake here,” “turn in here”) based on the AI racing line to apply during real-world riding.
7. As a user, I want to toggle between different visualization modes (e.g: 2D top-down view) to analyse racing lines effectively.

Visualization & Comparison Stories

1. As a racer, I want to switch between user-set and AI-optimized racing lines to choose the best fit for my skill level.
2. As a user, I want to scrub through the lap simulation to analyse critical points like braking zones and apexes.
3. As a coach, I want to export AI-generated racing lines and performance data for offline review and training.

Interface & User Experience Stories

1. As a casual user, I want a guided tutorial on how to interpret AI racing lines and use the app effectively.
2. As a user, I want to switch between light and dark modes for better visibility depending on the time of day.

Backend & Performance Stories

1. As a backend developer, I want the system to efficiently process large track images to reduce wait time for the user.
2. As a power user, I want to configure AI training parameters (e.g: epsilon decay, learning rate) for custom experiments.
3. As a team, we want to store training sessions and model states securely in a database so that progress isn’t lost between runs.

Gamification & Community Stories

1. As a user, I want to share my best lap and AI-optimized strategy with others to compare and compete.
2. As a community member, I want to vote on or comment on AI racing lines that others have shared to collaborate and learn.
3. As a racer, I want leaderboards showing AI lap times vs. user lap times to motivate improvement.

Service Contracts

Track Image Processing

Aspect	Description
Service Name	Track Image Processing
Description	Allows users to upload a top-down image of a racetrack. The system processes and standardizes it for analysis.
Inputs	Image file (JPG/PNG), optional track name or location
Outputs	Normalized track layout data (internal format), confirmation message
Interaction	Frontend sends image via HTTP POST; backend responds with processed track data or error

Racing Line Optimization

Aspect	Description
Service Name	Racing Line Optimization
Description	Calculates optimal racing line based on uploaded track image and racing parameters
Inputs	Track layout data, user skill level (optional), simulation settings
Outputs	Optimal line data (coordinates + speed/brake points), estimated lap time
Interaction	Backend returns optimized racing line as data or overlaid image

AI Training Service

Aspect	Description
Service Name	AI Training Service
Description	Trains reinforcement learning models to simulate different racing strategies on the track
Inputs	Track layout, AI parameters (e.g: learning rate, episodes), training goals
Outputs	Trained model, performance logs, fastest simulated lap time
Interaction	Invoked from backend or developer interface; may take time (async)

Visualization Service

Aspect	Description
Service Name	Visualization of Results

Aspect	Description
Description	Visually simulates laps using 2D/3D track views and overlays AI data on the track
Inputs	Racing line data (AI and/or user), view preferences (2D/3D), playback controls
Outputs	Unity-powered animation/render, scrub controls, brake/acceleration cues
Interaction	Real-time interaction on frontend with data fetched from backend

User Account Management *(optional)*

Aspect	Description
Service Name	User Account Management
Description	Handles user registration, login, and preferences storage
Inputs	Email, password, user profile info
Outputs	Auth tokens, session info, user data
Interaction	API-based login/signup endpoints, token-based authentication for access to services

Lap Time Comparison

Aspect	Description
Service Name	Lap Time Comparison
Description	Compares user-recorded lap times against AI's optimal laps
Inputs	User lap times (manually entered or uploaded), AI lap data
Outputs	Comparison report, performance delta, suggestions for improvement
Interaction	Web interface comparison, downloadable report or visual overlay

REQUIREMENTS

Functional Requirements

R1: Track Image Processing

R1.1: Image Conversion

- The system will convert top-down racetrack images into binary maps for AI analysis.
- The system will load data from saved csv files for comparison.

R1.2: Boundary Detection

- The system will accurately detect and distinguish track boundaries from off-track areas.
- The system will store this information for future use.

R2: Racing Line Optimization

R2.1: Reinforcement Learning

- The system will apply Reinforcement Learning (RL) to simulate and refine racing lines.
- The system will use data saved as .csv files to train the AI.

R2.2: Path Evaluation

- The system will iterate through multiple paths to determine the fastest racing line.

R3: AI Training and Simulation

R3.1: Training Data Input

- The system will train AI agents using simulated or game-based datasets.

R3.2: Physics Modelling

- The system will incorporate physics-based models to ensure realistic performance.

R4: Result Visualization

R4.1: Line Overlay

- The system will overlay the optimized racing line on the track image.

- The system will allow for adjustments to the overlay.

R4.2: Performance Metrics

- The system will display key performance indicators such as estimated lap time and braking zones.

R5: Infrastructure Integration

R5.1: Computation Support

- The system will support GPU-accelerated or equivalent computational resources for efficient RL training.

R5.2: Cloud Compatibility

- The system will optionally integrate with cloud services to allow for scalability and extended computation.

R6: Adaptive AI Strategies

R6.1: Dynamic Track Conditions

- The system will adjust racing lines based on simulated track conditions (e.g: wet/dry surfaces).

R7: Enhanced Visualization & User Interaction

R7.1: Interactive 3D Simulation (Optional)

- The system will provide optional 3D visualization of the track and racing line for enhanced user insight.

R7.2: Dynamic Line Adjustment

- The system will allow users to manually adjust the racing line and re-simulate performance with sliders and input areas.

R7.3: Heatmap of Speed/Acceleration Zones

- The system will generate a speed/acceleration ‘heatmap’ overlay for performance analysis.
- The system will allow users to provide feedback on AI-generated lines for iterative improvement.

Architectural Requirements

Architectural Design Strategy

Architectural Overview

The system will adopt a **microservices-based architecture** and **event driven architecture** to ensure modularity, scalability, and maintainability. Each major functionality – such as image preprocessing, reinforcement learning (RL) training, visualization, and user management – will be encapsulated within its own loosely coupled service. These services will communicate through event-driven mechanisms using technologies such as Kafka or RabbitMQ, enabling asynchronous processing and reactive behaviour across the platform.

This architectural approach is particularly suited to our application's workflow, where user actions (e.g: uploading a track or sharing a lap) trigger a cascade of processing stages. By decoupling components and promoting asynchronous event handling, the system remains scalable and resilient to failure in individual services.

Architectural Patterns

Event-Driven Architecture (EDA)

The system will heavily rely on Event-Driven Architecture to coordinate asynchronous tasks. When users upload new track images, an event will trigger the preprocessing pipeline. Similarly, once RL model training completes, another event will initiate the visualization service to generate optimal racing lines.

Examples of events include:

- `TrackUploaded` → triggers `TriggerPreprocessing`
- `ModelTrainingCompleted` → triggers `GenerateOptimalLine`
- `UserSharesLap` → triggers `UpdateLeaderboard`

This architecture allows components to remain decoupled and scale independently, improving performance and fault tolerance.

Model-View-Controller (MVC)

For user interaction and visualization, especially within Unity and potential web-based frontends, the system will follow the Model-View-Controller (MVC) design pattern:

- **Model**: Represents application data such as track metadata, AI model outputs, and simulation results (stored in MongoDB and PostgreSQL).
- **View**: Consists of Unity-based 3D visualizations and optional web dashboards built using React and Three.js.
- **Controller**: Handles user input, routes it to backend services, and updates the view with the appropriate state changes.

This separation of concerns simplifies UI development and makes the interface more responsive and maintainable.

Core Components and Interactions

The core system components and their interactions are described as follows:

- Track Processing Service
    - Input: Top-down track images (JPEG/PNG).
    - Output: Binary maps and detected boundaries, stored in Redis for fast retrieval.
    - Technology: Python, OpenCV.
- Reinforcement Learning (RL) Training Service
    - Input: Binary maps and physics parameters (e.g: tire grip, bike specs).
    - Output: Optimized racing lines with version control, stored in PostgreSQL.
    - Technology: PyTorch/TensorFlow, Python.
- Simulation Engine
    - Models realistic physics using a simulation library such as PyBullet or a custom engine.
- API Gateway
    - Offers REST and GraphQL endpoints for frontend access and internal coordination.
- Frontend

  - Web-based interface using React and Three.js, with optional desktop client via Electron.
  - Visual rendering through Unity.

Data Flow Overview:

`User Upload → Track Processing → RL Training → Simulation → Visualization`

Data Management

The system will employ a hybrid data storage strategy:

- Track images and metadata will be stored in AWS S3 (or equivalent blob storage) for cost-efficient scalability.
- Simulation results and racing lines will be stored in MongoDB (for structured queries).
- Training datasets ingested from games or simulators will use Parquet file format for optimized columnar storage and analytics.

Scalability and Performance

RL training will be horizontally scalable using Kubernetes, allowing auto-scaling across GPU-enabled nodes.

During periods of peak usage, image preprocessing workloads will be offloaded to AWS Lambda for efficient resource utilization.

The frontend will leverage CDN caching to serve static assets rapidly and reliably.

Fault Tolerance and Recovery

RL training processes will checkpoint progress every 15 minutes, ensuring minimal data loss in the event of failure.

A replica standby PostgreSQL instance will provide automatic database failover.

User uploads will automatically retry up to 3 times before surfacing an error to the user, increasing resilience to transient issues.

Security Architecture

The system will adopt a zero-trust security model, incorporating the following mechanisms:

- Authentication & Authorization: All API requests will be validated using JWT tokens.
- Network Isolation: Training workloads will run in isolated VPCs for enhanced security.
- Data Encryption:
    - At rest: AES-256 encryption for data in S3 and PostgreSQL.
    - In transit: All communication between services and users will be secured using HTTPS and mTLS.

Deployment and DevOps

The system's infrastructure will be managed using Infrastructure-as-Code (IaC) tools such as Terraform and Ansible. A robust CI/CD pipeline will be implemented using GitHub Actions or Jenkins, enabling:

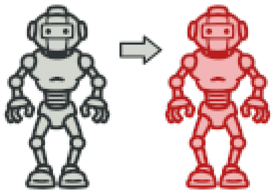
- Unit testing with PyTest and integration testing using Selenium.
- Automated rollback in case of deployment errors, triggered if failure rate exceeds 5% in canary deployments.

Design Patterns

Façade

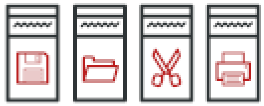
The Façade pattern is used to provide a simplified interface to the complex subsystems within the application. This design allows clients (e.g: frontend components or external APIs) to interact with the system through a unified entry point, hiding the complexity of underlying operations such as track processing, AI training, and data visualization. It promotes loose coupling between components and enhances maintainability by centralizing control logic.

Prototype



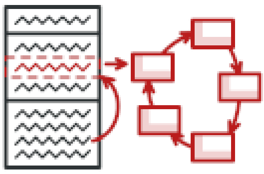
The Prototype pattern is employed to efficiently duplicate existing AI models, track configurations, or lap setups. This is particularly useful when users wish to reuse or slightly modify previously trained models or configurations without reprocessing them from scratch. Deep cloning ensures that replicated objects maintain their own state, avoiding unintended side effects caused by shared references.

Command



The Command pattern encapsulates user actions (such as uploading a track, modifying lap data, or initiating a simulation) as standalone command objects. This abstraction enables queuing, logging, and the ability to implement undo/redo functionalities. By decoupling the invoker from the execution logic, the system gains flexibility in handling user interactions in both the UI and backend workflows.

State



The State pattern allows the system to alter its behaviour dynamically based on its current state. For example, the UI and backend processing logic behave differently depending on whether a track is being uploaded, a model is in training, or results are ready for visualization. This pattern ensures that transitions between states (e.g: Idle → Processing → Completed) are handled cleanly and predictably, improving the system's reliability and user experience.

Architectural Strategies

[Needs to be created]

Architectural Quality Requirements

NF1: Performance Requirements

- **NF1.1:** The system will process and analyse a racetrack image (≤10MB) in under 5 seconds.

- **NF1.2:** AI training simulations will run at ≥30 FPS for real-time feedback during optimization.
- **NF1.3:** Lap time predictions will be computed within 1 second after track processing.
- **NF1.4:** The system will support at least 50 concurrent users in cloud-based mode.

NF2: Security Requirements

- **NF2.1:** All user-uploaded track images and telemetry data will be encrypted in transit (HTTPS/TLS 1.2+).
- **NF2.2:** Sensitive user data (e.g: login credentials) will be stored using salted hashing (bcrypt/PBKDF2).
- **NF2.3:** The system will enforce role-based access control (RBAC) for admin vs. end-user privileges.
- **NF2.4:** AI models and training data will be protected against unauthorized modification.

NF3: Reliability & Availability

- **NF3.1:** The system will maintain 95% uptime under normal operating conditions.
- **NF3.2:** Critical failures (e.g: RL training crashes) will recover automatically within 10 minutes.
- **NF3.3:** Backup procedures will ensure no more than 1 hour of data loss in case of system failure.
- **NF3.4:** The offline mode will retain core functionality (track processing, pre-trained AI suggestions) without cloud dependency.

NF4: Usability Requirements

- **NF4.1:** The interface will be intuitive for non-technical users (e.g: drag-and-drop track uploads, one-click simulations).
- **NF4.2:** Visualizations (racing line overlays, metrics) will adhere to colorblind-friendly palettes.
- **NF4.3:** The system will provide tooltips/guided tutorials for first-time users.
- **NF4.4:** All critical actions (e.g: deleting data) will require user confirmation.

NF5: Scalability Requirements

- **NF5.1:** The system will scale horizontally to support up to 10,000 simulations/day via cloud resources.
- **NF5.2:** Modular architecture will allow integration of new physics models or RL algorithms without major refactoring.
- **NF5.3:** GPU-accelerated training will dynamically allocate resources based on workload.

NF6: Compatibility Requirements

- **NF6.1:** The system will support Windows, macOS, and Linux for desktop applications.
- **NF6.2:** Web-based access will be compatible with Chrome, Firefox, and Edge (latest versions).
- **NF6.3:** Track images will be accepted in JPEG, PNG, or SVG formats (≤10MB).

NF7: Maintainability Requirements

- **NF7.1:** Code will be documented with API specs, inline comments, and version control (Git).
- **NF7.2:** The system will log errors with timestamps, severity levels, and recovery suggestions.
- **NF7.3:** Third-party dependencies (e.g: PyTorch, OpenCV) will be pinned to stable versions.

NF8: Cost & Resource Constraints

- **NF8.1:** Cloud computing costs will not exceed R5000 (aligned with project budget).
- **NF8.2:** Offline mode will operate on consumer-grade hardware (e.g: NVIDIA GTX 1060+ for GPU acceleration).

Architectural Design and Pattern

[Needs to be created]

Architectural Constraints

Limited Real-World Telemetry Data

Obtaining authentic racing telemetry for supervised learning is challenging. Consequently, the system relies primarily on simulated or gaming data, which may not fully capture real-world nuances.

Model Reliability and Accuracy

AI outputs must be rigorously validated against established racing strategies to ensure accuracy and dependability, preventing flawed decision-making.

Image Processing Complexity

The system must accurately interpret 2D track images, correctly detecting circuit boundaries and optimal racing lines. Errors at this stage could compromise the entire prediction pipeline.

Computational Resource Demands

Reinforcement learning requires significant hardware resources, such as GPUs or cloud infrastructure, to train models effectively within reasonable timeframes. This may limit deployment on less powerful devices.

Focus on 2D Data for Initial Development

Due to time constraints, the system emphasizes 2D image data import and analysis rather than full 3D simulation. This prioritizes core functionality and simplifies early development.

Technology Choices

Programming Language for Core System Development / Backend

Options Considered:

- Python
- C#
- C++
- Java

Choosing the right languages was essential to meet the system's AI and real-time 3D simulation needs. Python excels in AI/ML with its rich ecosystem, while C# integrates seamlessly with Unity for visualization.

Technology	***Pros***	***Cons***
Python	Extensive ML/AI libraries (e.g: PyTorch, NumPy), easy-to-learn syntax, rapid development	Slower runtime performance
C#	Seamless integration with Unity, good tooling support	Slight learning curve, not optimal for AI/ML
C++	High execution speed, low-level memory control	Increased complexity, longer development time, risk of memory leaks
Java	Platform-independent, strong multithreading capabilities	Verbose syntax, limited traction in AI/ML research

Final Choice: Python and C#

Justification: Python was chosen for AI/ML due to its speed of development and strong scientific libraries. C# was selected for 3D visualization because of its native Unity support. This combination supports our modular design by matching tools to their strengths.

AI & Machine Learning Framework

Options Considered:

- Python
- PSO (Particle Swarm Optimization)
- C#

Selecting an AI/ML framework required balancing ease of development, training capability, and integration with the Unity-based system. The options explored each brought different strengths to these goals.

Technology	Pros	Cons
Python	Rich AI/ML libraries (e.g: TensorFlow, PyTorch), fast prototyping, widely used in research	Not natively compatible with Unity, slower runtime
PSO	Lightweight, easy to implement for rule-based behavior, useful for early-stage systems	Not a full ML framework, lacks training scalability
C#	Seamless Unity integration, easier maintenance in a single-language pipeline	Limited ML support, less mature ecosystem for training

Justification: We are currently using PSO for initial behaviour logic due to its simplicity and low overhead. However, the system will be upgraded to a trainable model in the future. C# was chosen as the implementation language for now due to its native compatibility with Unity, ensuring smooth integration with the rendering engine and simplifying the overall architecture. This decision supports modular development and aligns with the constraint of keeping visualization and logic tightly integrated during early stages, while allowing for future expansion using Python-based training modules externally if needed.

Image Processing Library

Options Considered:

- **OpenCV**
- **Scikit-image**
- **PIL/Pillow**

Technology	Pros	Cons
OpenCV	Real-time processing, comprehensive tools	Complex API for beginners

Scikit-image	High-level API, easy integration with SciPy	Limited real-time support
Pillow	Lightweight, easy to use	Not suitable for complex tasks like track detection

Final Choice: OpenCV

Justification: OpenCV supports binary image conversion, edge detection, and other critical preprocessing steps required for accurate track interpretation. It's also highly optimized for performance.

2D Data Visualization

Options Considered:

- OpenCV extensions
- Matplotlib
- Plotly
- Seaborn

Technology	***Pros***	***Cons***
OpenCV	Real-time display, direct image overlay support, fast rendering	Limited charting capabilities, lower-level API
Matplotlib	Widely used, customizable, good for static plots	Static, less interactive
Plotly	Interactive, web-ready graphs	Slightly more complex API
Seaborn	High-level statistical plots, attractive defaults	Built on Matplotlib, less low-level control

Final Choice: OpenCV

Justification: OpenCV was chosen because its extensions allow direct visualization of data on images, which none of the other tools support as effectively. It fits the system's needs for fast, integrated image rendering and is better suited for our computer vision–focused architecture.

3D Visualization / Frontend

Options Considered:

- **Unity**
- **Unreal Engine**
- **Gazebo**

Technology	***Pros***	***Cons***
Unity	Real-time rendering, strong physics support	Learning curve
Unreal Engine	High-fidelity graphics	Heavier, more complex
Gazebo	Robot simulation focused	Less suited for racing visualization

Final Choice: Unity

Justification: Unity provides a balance between ease of use and strong simulation capabilities. Its built-in physics engine supports the real-time feedback required to demonstrate AI performance. Compared to Unreal Engine, Unity is significantly easier to set up and run on a wider range of systems, making it more accessible for both development and deployment.

Frontend (Website)

Options Considered:

- React
- Angular
- HTML, CSS, and JavaScript

Technology	Pros	Cons
React	Component-based, reusable UI, large ecosystem	Overkill for a simple page, steeper learning curve
Angular	Full-featured framework, powerful tooling	Complex setup, heavy for small projects
Simple HTML/CSS/JS	Lightweight, easy to implement, no dependencies	Limited scalability and interactivity

Final Choice: HTML, CSS, and JavaScript

Justification: Since the website consists of only a single page with a download link for the system, using a full framework like React or Angular would have been unnecessary overhead. A simple static page was quicker to build, required no additional dependencies, and avoided the need to learn or configure complex frameworks for such a minimal requirement.

Containerization

Options Considered:

- Docker
- Podman
- Vagrant

Technology	Pros	Cons
Docker	Industry standard, great tooling	Requires daemon, not rootless by default
Podman	Rootless containers, daemonless	Less ecosystem support

Technology	Pros	Cons
Vagrant	VM-based, good for OS-level testing	Slower and heavier than containers

Final Choice: Docker

Justification: Industry standard and it ensures consistency across development and deployment environments, simplifying CI/CD workflows and testing.

Database System

Options Considered:

- SQLite
- PostgreSQL
- MongoDB

Technology	Pros	Cons
SQLite	Lightweight, zero-configuration setup	Limited support for concurrent writes
PostgreSQL	Highly scalable, supports complex queries and transactions	More resource-intensive than SQLite
MongoDB	Schema-less, flexible data model, free and easy to use	Less suited for complex relational data

Final Choice: MongoDB

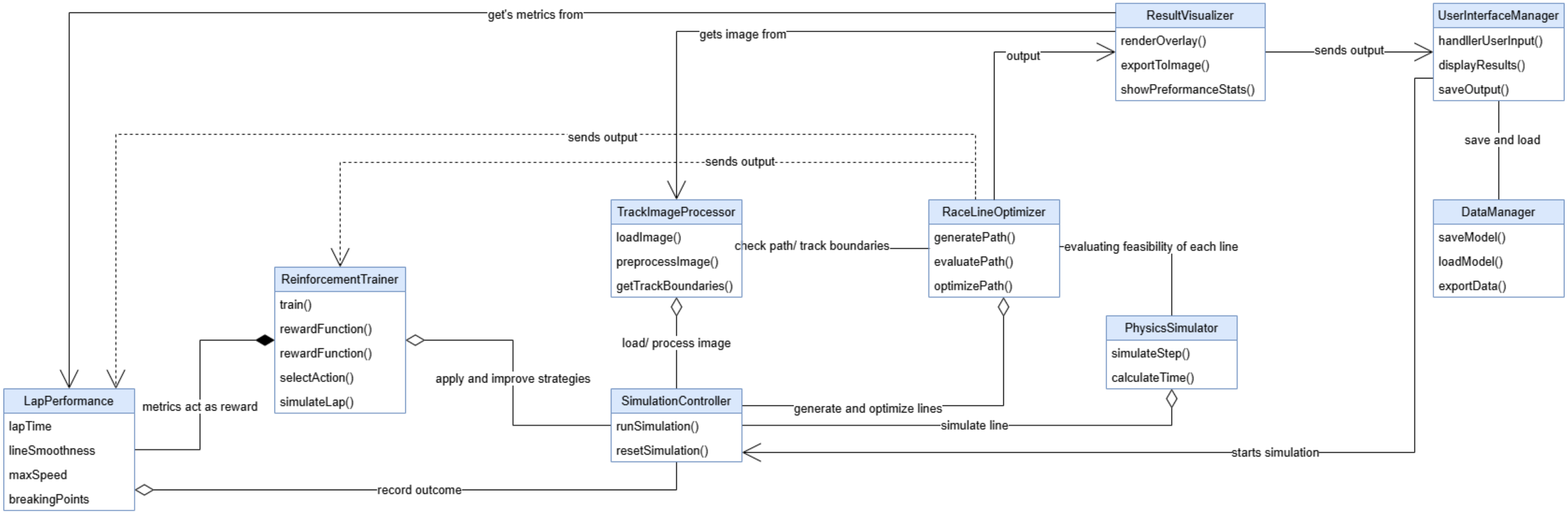
Justification: MongoDB was selected for its flexibility and ease of integration, especially given the schema-less nature of our data. Being free and straightforward to set up, it fits well with our system's need for fast access and simple maintenance without the overhead of rigid relational schemas.

Diagrams and Models

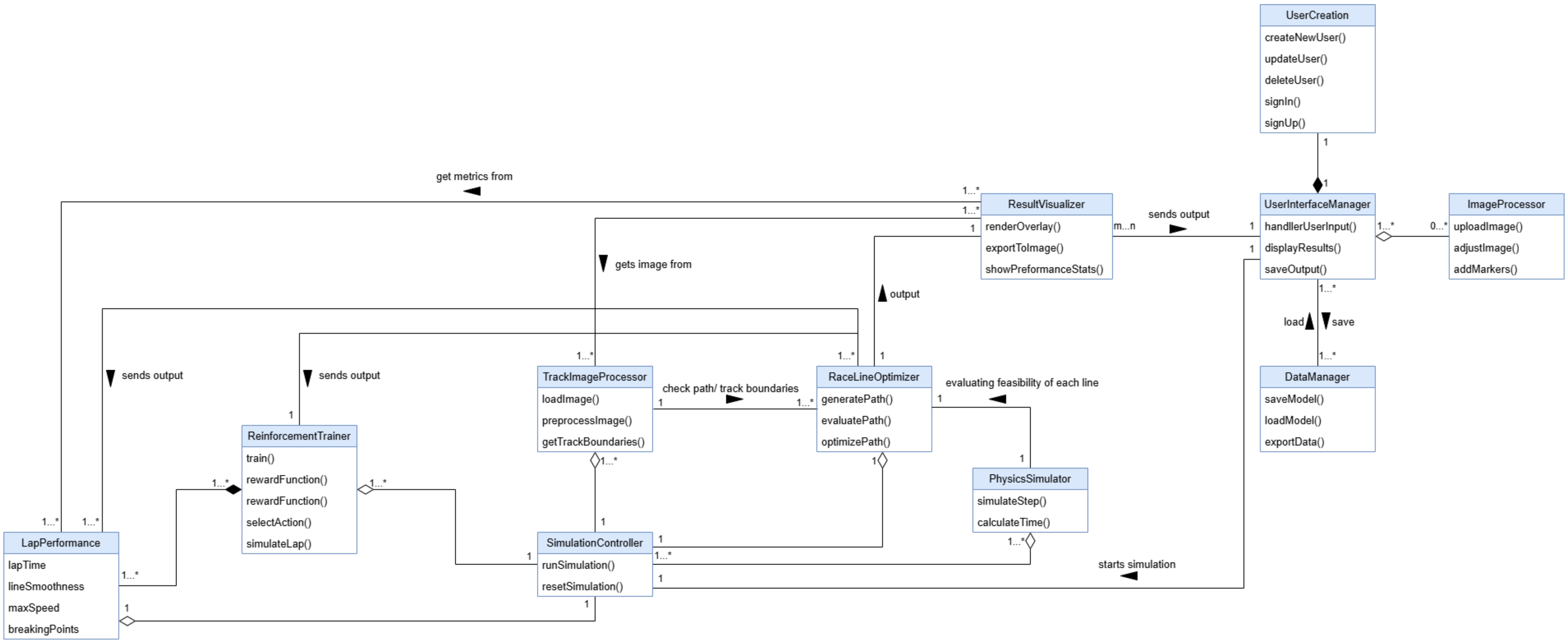
Architecture Diagram

[Needs to be created]

Class Diagram



Domain Model



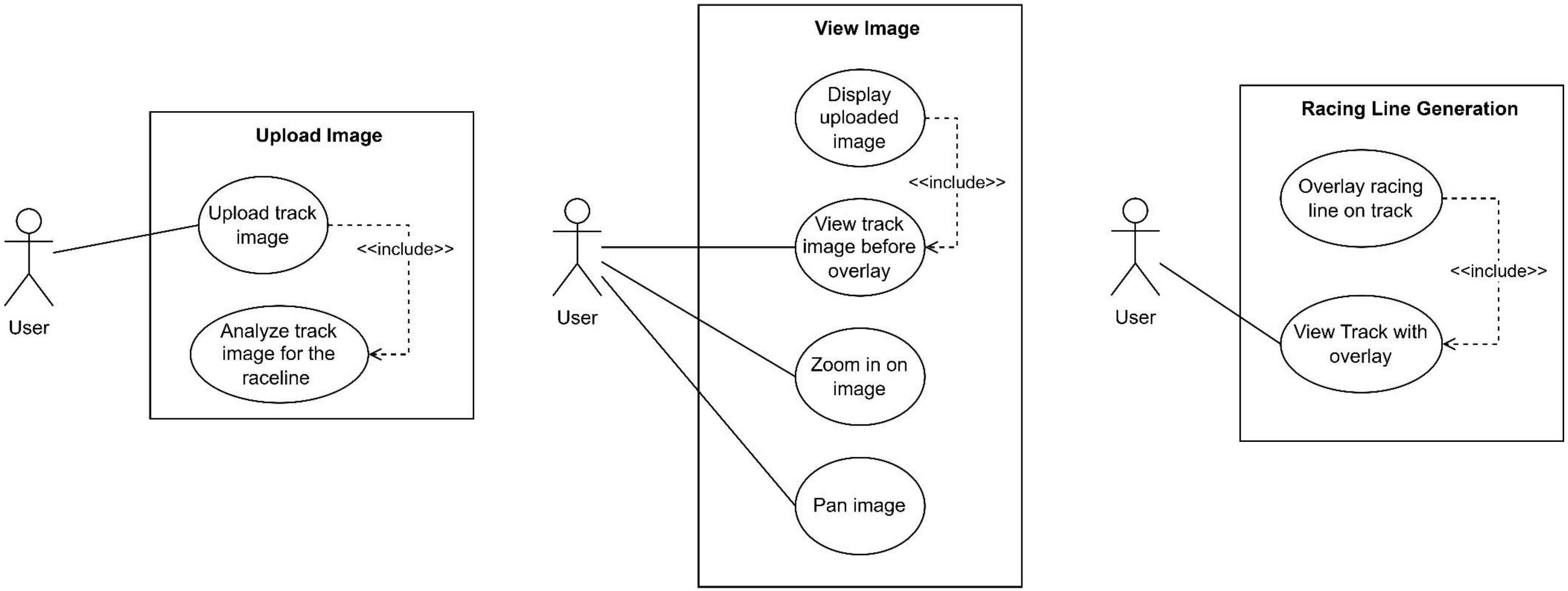
Deployment Model

[Needs to be created]

Live Deployment System

[Needs to be created]

Use Case Diagrams



Manuals

Installation Manual

[Needs to be created]

Technical Installation Manual

[Needs to be created]

User Manual

System Requirements

- Minimum hardware and software requirements to run the system.
    - Windows OS (for the current system)
    - Unity version 6.0 installed.
    - Docker Desktop should be installed

How to Access the System

- Step-by-step on how to open or install:
    - If web-based: how to navigate to the site.
    - If Unity: how to open the Unity project.
    - If Docker: how to run using Docker (docker-compose up, etc.).
- Include screenshots where applicable.

Navigation Overview

- **Include a labeled screenshot of the UI**
    - Briefly describe the main parts of the interface (e.g: “Upload Track”, “Start Processing”, “View Raceline”).
- Explain what each section of the interface does in simple terms.

Step-by-Step Workflow

Guide the user through a basic use case. This might look like:

1. **Upload Track Image**
    - Describe where and how to upload the track image from Unity or web.
    - Include screenshot.

2. **Processing**
   - Explain what happens once the track is uploaded.
   - Mention processing time or expected behavior.
   - (Optional: Screenshot of progress bar or waiting screen.)
3. **Viewing Results**
   - Show where the user will see the optimized raceline.
   - Describe different output formats if relevant (map overlay, stats, data export).
4. **Exporting/Using Output**
   - If the system allows exporting files or sending data to Unity, explain how.
   - Include screenshot of export/download option.

Troubleshooting

- You cannot log into the system?
  - Ensure that it is running first.
- You can't upload an image?
  - Ensure the system is still running and has not crashed.
- If Docker will not run try the following in the terminal (only run the next one, once the first is finished building):

```
docker-compose down --remove-orphans
docker-compose build --no-cache
docker-compose up
```

Contact or Support Info

Please send any issues you may experience to our email at:

ctprojectteam3@gmail.com

Specifications and Standards

Machine Learning Specification

[Needs to be created]

API Documentation

[Needs to be created]

Coding Standards

Naming Conventions

File Names: A mix of `PascalCase` and `camelCase` is used.

Folder Names: Generally, use `PascalCase`. However, some folders follow lowercase naming conventions for system compatibility – for example, the docs folder is lowercase to enable GitHub Pages hosting.

Class Names: All class names follow `PascalCase` for clarity and consistency.

Special Cases:

- API and `RacelineOptimizer` follow `PascalCase` as they are core modules.
- `image_processing` uses `snake_case` to align with external library conventions and improve readability in multi-word module names.

File and Folder Structure

The project is organized into modular folders to separate concerns and support scalable development. Below is the structure of the repository:

Repository Root

- `Backend/` – Contains core backend components including:
    - `API/`: Handles external communication (e.g: Unity and MongoDB).
    - `ImageProcessing/`: Processes images received from Unity, converting them into usable track data.
    - `RacelineOptimizer/`: Uses processed images to determine the optimal raceline.
- `docs/` – Stores documentation and static site files (used for GitHub Pages hosting). Subdirectories include:
- `css/, js/, images/, wordDocs/, and index.html.`
- `scripts/ – Contains setup scripts and developer utilities:`

    - `setup-act.sh`: Installs `nektos/act` to run GitHub Actions locally.
    - `ACT.md`: Documentation for using local workflows.
- `Unity/` – The front-end Unity project used for rendering and interaction.
- `Website/` – Web-related files for convenience and deployment purposes.
- `README.md` – Project overview and general instructions.

Docker and Testing

- Each service folder (except Unity) contains its own `Dockerfile`.
- A global `docker-compose.yml` file is located in the project root.
- `.dockerignore` files are placed in each relevant directory.
- Testing directories (e.g: `tests_integration/`, `e2e/`, `unit/`) are found within service folders for modular test execution.

Formatting Standards

- **Indentation**: Tabs are used for indentation across the project for consistency.
- **Line Length**: No strict limit has been enforced, but lines are generally kept concise for readability.
- **Braces**: Opening braces are placed on the same line as control statements (e.g: `if (...) {`), with the block content starting on the next line.
- **Spacing**: Standard spacing is followed, including spacing around operators and within brackets (e.g: `{ int = 0; }`).
- **Comments**:
    - Both single-line (`//`) and block (`/* */`) comments are used.
    - Single-line comments are used for short explanations, while block comments provide contextual or functional documentation.
- **Docstrings**: No specific docstring format is used in this project.

Coding Practices

- **Naming**: Functions and files are named to clearly reflect their purpose or output. Descriptive naming is prioritized over name length limitations.
- **Structure**: Code is kept modular and functions are designed to handle specific tasks where possible.
- **General Practices**: Standard coding practices are followed, including avoiding deeply nested logic, keeping code readable, and minimizing redundancy.

Version Control Guidelines

Commit Messages: All commit messages must be clear, descriptive, and explain what the commit does.

Branching Strategy:

The primary branches are:

- `main`: Stable production-ready code.
- `dev`: Integration branch for completed features.

Feature branches are categorized by function:

- `UI/`: Frontend and website-related work
- `Backend/`: Backend processing and API
- `CICD/`: Continuous Integration and Deployment scripts/tests

Branch naming follows a consistent format:

- Example: Backend-PSA-start, UI-Web-LandingPage

Commit Frequency: Developers are expected to make a minimum of 10 commits per week, ideally after every significant update on their feature branch.

Pull Requests:

- Pull requests must be submitted once a branch feature is complete.
- Each PR must be reviewed by **at least two team members** before being merged.

- Branches are merged progressively: `feature → category (e.g: UI) → dev → main`.
- Direct commits to main are not allowed.

CI/CD: The main branch runs the CI/CD pipelines to ensure stability.

Tools and Configurations

CI/CD: A basic CI/CD setup is implemented, currently running automated tests from the various tests folders.

Docker:

- Each backend component (API, ImageProcessor, RacelineOptimizer) has its own `Dockerfile`.
- A root-level `docker-compose.yml` is used to orchestrate the containers.

Scripts: Utility scripts are stored in the scripts/ directory for local tool setup and CI/CD helpers.

Linters/Formatters: Not strictly enforced, but individual team members may use personal formatting tools suited to their language. There is also currently linting present in our C# code.

Language/ Framework-Specific Conventions

Unity and RacelineOptimizer: Written in C#. Follows typical Unity/C# naming and structure conventions.

API: Implemented in Node.js using JavaScript/TypeScript.

Image Processor: Written in Python, using common Pythonic conventions (e.g: `snake_case`, modular scripts).

Website: Built with standard HTML, CSS, and JavaScript, organized within the docs/ folder for GitHub Pages compatibility.

Testing Policy

Testing Scope & Levels

Level	*Focus*	*Tools/Methods*
Unit Testing	Individual functions (e.g: track image processing, RL reward function).	Pytest (Python), JUnit (Java).
Integration Testing	Interaction between services (e.g: track processor → RL engine).	Postman, Jest (API tests), Selenium (UI flows).
System Testing	End-to-end workflows (e.g: upload image → simulate → visualize).	Cypress, Robot Framework.
Performance Testing	Scalability (e.g: 50 concurrent users), RL training speed.	Locust (load testing), NVIDIA Nsight (GPU profiling).
Security Testing	Data encryption, auth vulnerabilities.	OWASP ZAP, SonarQube.
User Acceptance (UAT)	Real-world usability (by target users).	Beta releases, A/B testing.

Testing Types & Frequency

Test Type	*Description*	*Frequency*
Automated Regression	Validate existing features after updates.	On every Git commit (CI/CD).
Manual Exploratory	Unscripted UX/edge-case testing.	Before major releases.
Physics Validation	Compare AI racing lines against known heuristics (e.g: apex accuracy).	Per RL model update.

Entry & Exit Criteria

Entry Criteria (Tests Start When):

- Requirements are documented.

- Code is merged to the test branch.
- Test environment mirrors production (GPU-enabled).

Exit Criteria (Tests Pass When):

- **Unit/Integration:** ≥90% code coverage (measured via Coveralls).
- **Performance:** <2s response time for track processing; RL training FPS ≥30.
- **Security:** Zero critical OWASP vulnerabilities.
- **UAT:** ≥80% positive feedback from beta testers.

Defect Management

- **Severity Levels:**
  - **Critical** (Crash/data loss): Fixed within 24h.
  - **Major** (Feature failure): Fixed in next sprint.
  - **Minor** (UI glitch): Backlogged for prioritization.
- **Tracking:** Jira/Linear with labels (bug, reproducible, blocker).

Environments

Environment	*Purpose*	*Access*
Development	Feature development.	Engineers only.
Staging	Pre-production (mirrors prod).	QA/Product Team.
Production	Live user-facing system.	Automated deployments only.

Test Data Management

- **Realistic Datasets:**
  - 10+ sample tracks (F1, MotoGP circuits).
  - Synthetic data from racing sims (Assetto Corsa).
- **Anonymization:** User-uploaded tracks scrubbed of metadata.

Compliance & Reporting

- **Audits:** Monthly test coverage/review meetings.
- **Reports:** Dashboards for:
    - Test pass/fail rates.
    - Performance trends (e.g: lap time prediction accuracy).

Policy Exceptions

- **Emergency Fixes:** Hotfixes may bypass some tests but require:
    - Post-deployment regression testing.
    - Retrospective review.

Contribution of Teammates

Project Manager

Amber Werner

[list contributions]

Backend Developers

Qwinton Knocklein

[list contributions]

Sean van der Merwe

[list contributions]

Front End Developers

Simon van der Merwe

[list contributions]

Milan Kruger

[list contributions]

Appendix: Old Versions of SRS

Version 1 [26/05/2025]

Introduction

There is a growing need for accessible, data-driven training tools in motorsports, especially among students, amateur riders, and enthusiasts who lack access to expensive telemetry systems or real-world testing environments. SuperLap Racing Line Optimization System addresses this need by providing an AI-powered platform that helps superbike riders identify the fastest possible racing line on a racetrack.

The project aims to develop a Reinforcement Learning and Computer Vision-based system that analyses a top-down image of a racetrack, simulates thousands of optimal pathing scenarios, and overlays the ideal racing line on the map. Designed with usability and precision in mind, SuperLap focuses on delivering accurate, performance-enhancing insights in a visually intuitive format, supporting smarter race training without the traditional barriers of cost or access.

User Characteristics

Amateur & Hobbyist Racers

Characteristics:

- **Skill Level:** Novice to intermediate riders.
- **Goals:** Improve lap times, learn optimal racing lines, and understand track dynamics.
- **Technical Proficiency:** Basic (comfortable with apps but not deep technical knowledge).
- **Usage:**
    - Uploads track images from local circuits.
    - Uses AI-generated racing lines as training aids.
    - Compares different lines for self-improvement.
- **Motivation:** Cost-effective alternative to professional coaching/telemetry.

Example: A track-day rider at Kyalami Circuit who wants to shave seconds off their lap time.

Motorsport Coaches & Instructors

Characteristics:

- **Skill Level:** Advanced (former/current racers).
- **Goals:** Teach students optimal racing strategies using AI insights.
- **Technical Proficiency:** Moderate (understands racing physics but not AI/ML).
- **Usage:**
    - Validates AI suggestions against their experience.
    - Generates visual training materials for students.

  - Compares different rider lines for debriefs.
- **Motivation:** Enhances coaching efficiency with data-backed insights.

Example: A riding instructor at a racing school who uses SuperLap to show students braking points.

Sim Racing Enthusiasts

Characteristics:

- **Skill Level:** Varies (casual to competitive sim racers).
- **Goals:** Optimize virtual racing performance in games like *Assetto Corsa* or *Gran Turismo*.
- **Technical Proficiency:** High (comfortable with mods/data analysis).
- **Usage:**
  - Imports game track maps for AI analysis.
  - Compares SuperLap's line against in-game telemetry.
  - Shares optimized lines with sim racing communities.
- **Motivation:** Gain a competitive edge in online races.

Example: An iRacing league player who wants the perfect Monza line.

Professional Racing Teams (Small/Privateer)

Characteristics:

- **Skill Level:** Expert (professional riders/engineers).
- **Goals:** Fine-tune bike setup and validate strategies.
- **Technical Proficiency:** High (understands AI, telemetry, and vehicle dynamics).
- **Usage:**
  - Cross-checks AI predictions with real-world data.

- Tests "what-if" scenarios (e.g: wet vs. dry lines).
- Integrates with existing telemetry tools (if API available).

- **Motivation:** Affordable alternative to high-end motorsport analytics.

Example: A privateer Moto3 team optimizing cornering lines on a budget.

Engineering & Motorsport Students

Characteristics:

- **Skill Level:** Academic (learning racing dynamics/AI).
- **Goals:** Study racing line theory, RL applications, and vehicle physics.
- **Technical Proficiency:** Medium (some coding/math knowledge).
- **Usage:**
  - Experiments with different AI models (e.g: DQN vs. PPO).
  - Validates academic theories against SuperLap's simulations.
- **Motivation:** Research and project-based learning.

Example: A mechanical engineering student analysing Suzuka's "S-curves" for a thesis.

User Stories

Core User Stories (Functionality & User Experience)

1. As a rider, I want to upload a top-down image of my racetrack so that the system can analyse it for racing line optimization.
2. As a user, I want to see and customize the image that I have uploaded.
3. As a user, I want to see the AI-generated optimal racing line overlaid on the track so that I can compare it to my existing racing strategy.
4. As a motorsport enthusiast, I want the system to simulate multiple racing lines using reinforcement learning so I can see which line performs the best under different conditions.

5. As a rider, I want to compare my recorded lap times with the AI's optimal lap time so I can identify areas for improvement.
6. As a user, I want the app to visually simulate the lap with a bike animation in Unity so I can better understand the racing line's logic.
7. As a beginner racer, I want simple guidance such as "brake here" or "turn in here" based on the AI's racing line, so I can apply it in real life.
8. As a user, I want to toggle between 2D and 3D views of the track to better analyse racing lines.

Visualization & Comparison Stories

1. As a racer, I want to switch between different racing line strategies (self-set vs. AI-optimized) so I can decide which one is best suited for my skill level.
2. As a user, I want the ability to scrub through a lap simulation to analyse key moments like braking zones and apex points.
3. As a coach, I want to export performance data and AI-generated lines for further analysis outside the app.

Interface & User Experience Stories

3. As a casual user, I want a guided tutorial on how to interpret AI racing lines and use the app effectively.
4. As a user, I want to switch between light and dark modes for better visibility depending on the time of day.

Backend & Performance Stories

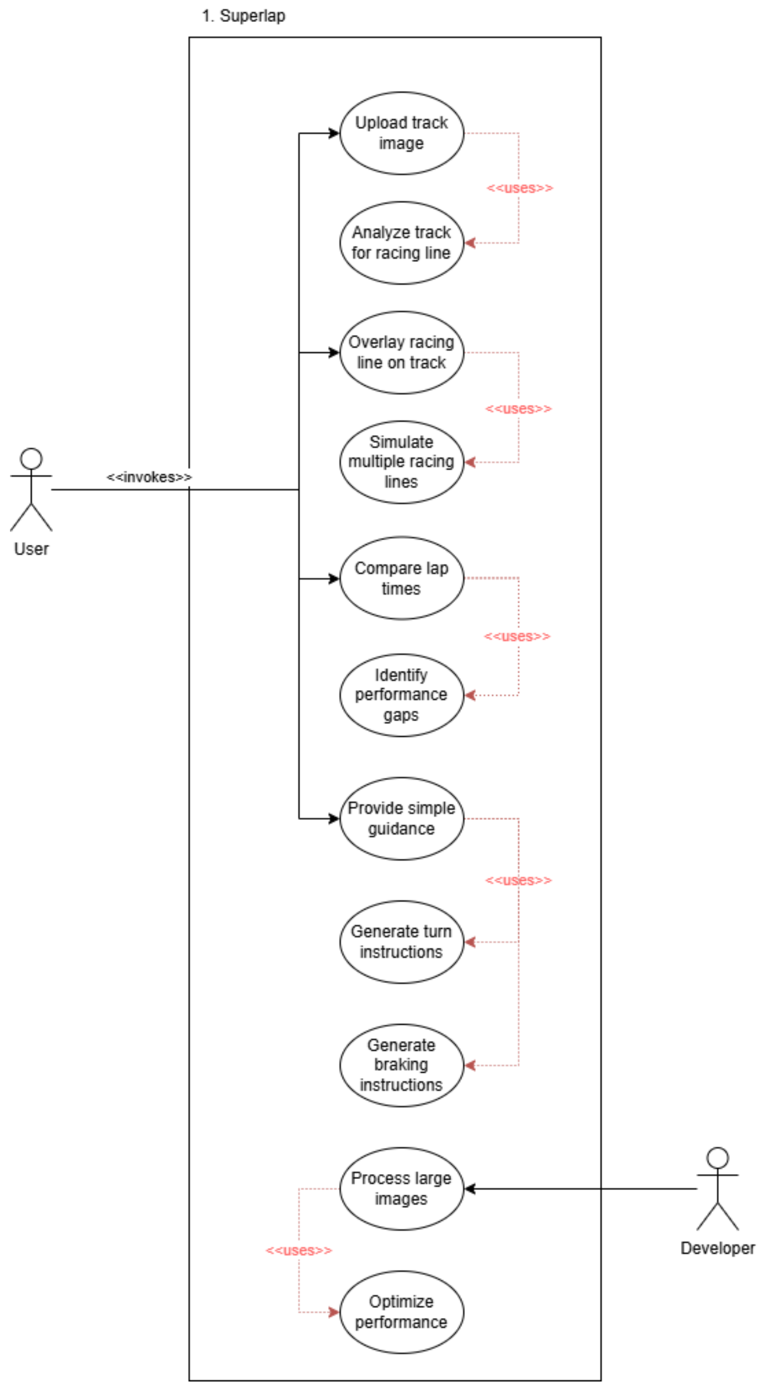
4. As a backend developer, I want the system to efficiently process large track images to reduce wait time for the user.
5. As a power user, I want to configure AI training parameters (e.g: epsilon decay, learning rate) for custom experiments.
6. As a team, we want to store training sessions and model states securely in a database so that progress isn't lost between runs.

Gamification & Community Stories

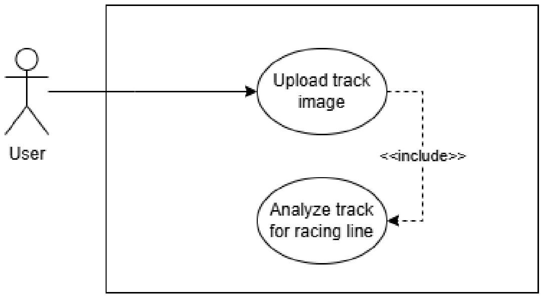
4. As a user, I want to share my best lap and AI-optimized strategy with others to compare and compete.

5. As a community member, I want to vote on or comment on AI racing lines that others have shared to collaborate and learn.
6. As a racer, I want leaderboards showing AI lap times vs. user lap times to motivate improvement.

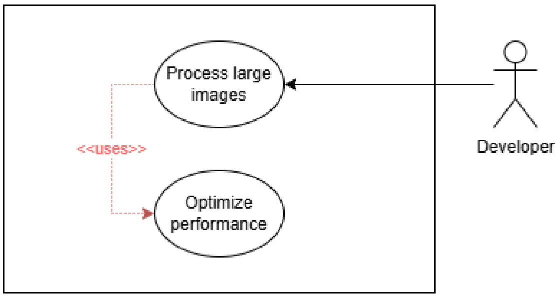
Use Case



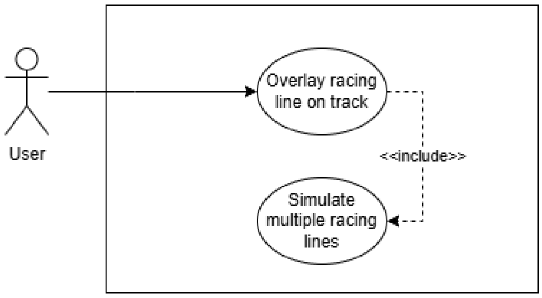
1.1 Image Upload



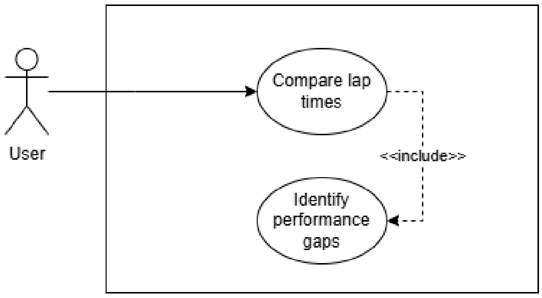
1.4 Image Processing



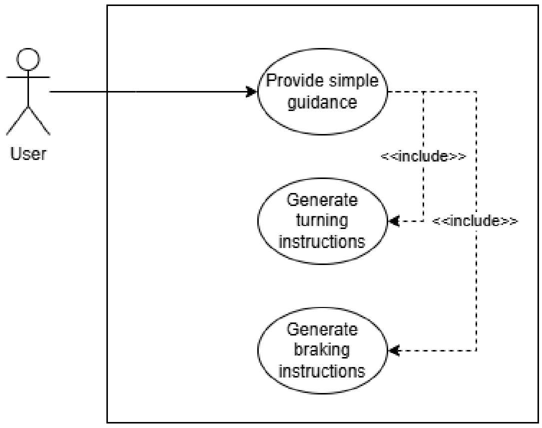
1.2 Racing Line Generation



1.3 Performance Analysis



1.4 User Guidance



Service Contracts

Track Image Processing

Aspect	Description
Service Name	Track Image Processing
Description	Allows users to upload a top-down image of a racetrack. The system processes and standardizes it for analysis.
Inputs	Image file (JPG/PNG), optional track name or location
Outputs	Normalized track layout data (internal format), confirmation message
Interaction	Frontend sends image via HTTP POST; backend responds with processed track data or error

Racing Line Optimization

Aspect	Description
Service Name	Racing Line Optimization
Description	Calculates optimal racing line based on uploaded track image and racing parameters
Inputs	Track layout data, user skill level (optional), simulation settings
Outputs	Optimal line data (coordinates + speed/brake points), estimated lap time
Interaction	Backend returns optimized racing line as data or overlaid image

AI Training Service

Aspect	Description
Service Name	AI Training Service
Description	Trains reinforcement learning models to simulate different racing strategies on the track
Inputs	Track layout, AI parameters (e.g: learning rate, episodes), training goals

Outputs	Trained model, performance logs, fastest simulated lap time
Interaction	Invoked from backend or developer interface; may take time (async)

Visualization Service

Aspect	Description
Service Name	Visualization of Results
Description	Visually simulates laps using 2D/3D track views and overlays AI data on the track
Inputs	Racing line data (AI and/or user), view preferences (2D/3D), playback controls
Outputs	Unity-powered animation/render, scrub controls, brake/acceleration cues
Interaction	Real-time interaction on frontend with data fetched from backend

User Account Management (optional)

Aspect	Description
Service Name	User Account Management
Description	Handles user registration, login, and preferences storage
Inputs	Email, password, user profile info
Outputs	Auth tokens, session info, user data
Interaction	API-based login/signup endpoints, token-based authentication for access to services

Lap Time Comparison

Aspect	Description
Service Name	Lap Time Comparison
Description	Compares user-recorded lap times against AI's optimal laps
Inputs	User lap times (manually entered or uploaded), AI lap data
Outputs	Comparison report, performance delta, suggestions for improvement

Interaction	Web interface comparison, downloadable report or visual overlay

Requirements

Technology Requirements

Technology	Purpose	Justification
Git & GitHub	Version control & collaboration	Enables efficient branching, tracking, and CI/CD workflows via GitHub Actions.
Python	AI/ML Model development	Widely adopted in ML with extensive libraries (e.g: PyTorch, NumPy).
PyTorch/TensorFlow	Reinforcement Learning (RL) framework	Industry-standard for RL; supports GPU acceleration for faster training.
OpenCV	Image processing & track detection	Effective for binary conversion and track boundary detection.
Matplotlib/Plotly	2Ddata visualization	Ideal for overlaying racing lines on images for analysis.
Unity	3D Visualization	Provides immersive simulations with real-time physics rendering.
React	Web interface	Modern, responsive frontend that integrates well with visualization libraries.
Express	Backend API service	Lightweight Node web framework for seamless model serving and data routing.
Docker	Containerization	Ensures reproducibility across environments (e.g: cloud, local).

SQLite/PostgreSQL	Database for storing track/line data	Lightweight (SQLite) or scalable (PostgreSQL) for performance logs.
PyBullet/MuJoCo	Physics engine (optional)	Simulates bike dynamics and tire friction for more accurate RL training.

Functional Requirements

R1: Track Image Processing

R1.1: Image Conversion

- The system will convert top-down racetrack images into binary maps for AI analysis.
- The system will load data from saved csv files for comparison.

R1.2: Boundary Detection

- The system will accurately detect and distinguish track boundaries from off-track areas.
- The system will store this information for future use.

R2: Racing Line Optimization

R2.1: Reinforcement Learning

- The system will apply Reinforcement Learning (RL) to simulate and refine racing lines.
- The system will use data saved as .csv files to train the AI.

R2.2: Path Evaluation

- The system will iterate through multiple paths to determine the fastest racing line.

R3: AI Training and Simulation

R3.1: Training Data Input

- The system will train AI agents using simulated or game-based datasets.

R3.2: Physics Modelling

- The system will incorporate physics-based models to ensure realistic performance.

R4: Result Visualization

R4.1: Line Overlay

- The system will overlay the optimized racing line on the track image.

- The system will allow for adjustments to the overlay.

R4.2: Performance Metrics

- The system will display key performance indicators such as estimated lap time and braking zones.

R5: Infrastructure Integration

R5.1: Computation Support

- The system will support sufficient computational resources (e.g: GPU) for RL training.

R5.2: Cloud Compatibility

- The system will optionally integrate with cloud services to allow for scalability and extended computation.

R6: Adaptive AI Strategies

R6.1: Dynamic Track Conditions

- The system will adjust racing lines based on simulated track conditions (e.g: wet/dry surfaces).

R7: Enhanced Visualization & User Interaction

R7.1: Interactive 3D Simulation (Optional)

- The system will provide a 3D interactive visualization of the track and optimized racing line.

R7.2: Dynamic Line Adjustment

- The system will allow users to manually adjust the racing line and re-simulate performance with sliders and input areas.

R7.3: Heatmap of Speed/Acceleration Zones

- The system will generate a speed/acceleration ‘heatmap’ overlay for performance analysis.
- The system will allow users to provide feedback on AI-generated lines for iterative improvement.

Architectural Requirements

Quality Requirements

NF1: Performance Requirements

- **NF1.1:** The system will process and analyse a racetrack image (≤10MB) in under 5 seconds.
- **NF1.2:** AI training simulations will run at ≥30 FPS for real-time feedback during optimization.
- **NF1.3:** Lap time predictions will be computed within 1 second after track processing.
- **NF1.4:** The system will support at least 50 concurrent users in cloud-based mode.

NF2: Security Requirements

- **NF2.1:** All user-uploaded track images and telemetry data will be encrypted in transit (HTTPS/TLS 1.2+).
- **NF2.2:** Sensitive user data (e.g: login credentials) will be stored using salted hashing (bcrypt/PBKDF2).
- **NF2.3:** The system will enforce role-based access control (RBAC) for admin vs. end-user privileges.
- **NF2.4:** AI models and training data will be protected against unauthorized modification.

NF3: Reliability & Availability

- **NF3.1:** The system will maintain 95% uptime under normal operating conditions.
- **NF3.2:** Critical failures (e.g: RL training crashes) will recover automatically within 10 minutes.
- **NF3.3:** Backup procedures will ensure no more than 1 hour of data loss in case of system failure.
- **NF3.4:** The offline mode will retain core functionality (track processing, pre-trained AI suggestions) without cloud dependency.

NF4: Usability Requirements

- **NF4.1:** The interface will be intuitive for non-technical users (e.g: drag-and-drop track uploads, one-click simulations).
- **NF4.2:** Visualizations (racing line overlays, metrics) will adhere to colorblind-friendly palettes.
- **NF4.3:** The system will provide tooltips/guided tutorials for first-time users.
- **NF4.4:** All critical actions (e.g: deleting data) will require user confirmation.

NF5: Scalability Requirements

- **NF5.1:** The system will scale horizontally to support up to 10,000 simulations/day via cloud resources.
- **NF5.2:** Modular architecture will allow integration of new physics models or RL algorithms without major refactoring.
- **NF5.3:** GPU-accelerated training will dynamically allocate resources based on workload.

NF6: Compatibility Requirements

- **NF6.1:** The system will support Windows, macOS, and Linux for desktop applications.
- **NF6.2:** Web-based access will be compatible with Chrome, Firefox, and Edge (latest versions).
- **NF6.3:** Track images will be accepted in JPEG, PNG, or SVG formats (≤10MB).

NF7: Maintainability Requirements

- **NF7.1:** Code will be documented with API specs, inline comments, and version control (Git).
- **NF7.2:** The system will log errors with timestamps, severity levels, and recovery suggestions.
- **NF7.3:** Third-party dependencies (e.g: PyTorch, OpenCV) will be pinned to stable versions.

NF8: Cost & Resource Constraints

- **NF8.1:** Cloud computing costs will not exceed R5000 (aligned with project budget).
- **NF8.2:** Offline mode will operate on consumer-grade hardware (e.g: NVIDIA GTX 1060+ for GPU acceleration).

Architectural Pattern

Architectural Overview

The system will adopt a **microservices-based architecture** and **event driven architecture** to ensure modularity, scalability, and maintainability. Each major functionality – such as image preprocessing, reinforcement learning (RL) training, visualization, and user management – will be encapsulated within its own loosely coupled service. These services will communicate through event-driven mechanisms using technologies such as Kafka or RabbitMQ, enabling asynchronous processing and reactive behaviour across the platform.

This architectural approach is particularly suited to our application's workflow, where user actions (e.g: uploading a track or sharing a lap) trigger a cascade of processing stages. By decoupling components and promoting asynchronous event handling, the system remains scalable and resilient to failure in individual services.

Architectural Patterns

Event-Driven Architecture (EDA)

The system will heavily rely on Event-Driven Architecture to coordinate asynchronous tasks. When users upload new track images, an event will trigger the preprocessing pipeline. Similarly, once RL model training completes, another event will initiate the visualization service to generate optimal racing lines.

Examples of events include:

- `TrackUploaded` → triggers `TriggerPreprocessing`
- `ModelTrainingCompleted` → triggers `GenerateOptimalLine`
- `UserSharesLap` → triggers `UpdateLeaderboard`

This architecture allows components to remain decoupled and scale independently, improving performance and fault tolerance.

Model-View-Controller (MVC)

For user interaction and visualization, especially within Unity and potential web-based frontends, the system will follow the Model-View-Controller (MVC) design pattern:

- **Model**: Represents application data such as track metadata, AI model outputs, and simulation results (stored in MongoDB and PostgreSQL).
- **View**: Consists of Unity-based 3D visualizations and optional web dashboards built using React and Three.js.
- **Controller**: Handles user input, routes it to backend services, and updates the view with the appropriate state changes.

This separation of concerns simplifies UI development and makes the interface more responsive and maintainable.

Core Components and Interactions

The core system components and their interactions are described as follows:

- Track Processing Service
    - Input: Top-down track images (JPEG/PNG).
    - Output: Binary maps and detected boundaries, stored in Redis for fast retrieval.
    - Technology: Python, OpenCV.
- Reinforcement Learning (RL) Training Service
    - Input: Binary maps and physics parameters (e.g: tire grip, bike specs).
    - Output: Optimized racing lines with version control, stored in PostgreSQL.
    - Technology: PyTorch/TensorFlow, Python.
- Simulation Engine
    - Models realistic physics using a simulation library such as PyBullet or a custom engine.
- API Gateway
    - Offers REST and GraphQL endpoints for frontend access and internal coordination.
- Frontend

  - Web-based interface using React and Three.js, with optional desktop client via Electron.
  - Visual rendering through Unity.

Data Flow Overview:

```
User Upload → Track Processing → RL Training → Simulation → Visualization
```

Data Management

The system will employ a hybrid data storage strategy:

- Track images and metadata will be stored in AWS S3 (or equivalent blob storage) for cost-efficient scalability.
- Simulation results and racing lines will be stored in MongoDB (for structured queries).
- Training datasets ingested from games or simulators will use Parquet file format for optimized columnar storage and analytics.

Scalability and Performance

RL training will be horizontally scalable using Kubernetes, allowing auto-scaling across GPU-enabled nodes.

During periods of peak usage, image preprocessing workloads will be offloaded to AWS Lambda for efficient resource utilization.

The frontend will leverage CDN caching to serve static assets rapidly and reliably.

Fault Tolerance and Recovery

RL training processes will checkpoint progress every 15 minutes, ensuring minimal data loss in the event of failure.

A replica standby PostgreSQL instance will provide automatic database failover.

User uploads will automatically retry up to 3 times before surfacing an error to the user, increasing resilience to transient issues.

Security Architecture

The system will adopt a zero-trust security model, incorporating the following mechanisms:

- Authentication & Authorization: All API requests will be validated using JWT tokens.
- Network Isolation: Training workloads will run in isolated VPCs for enhanced security.
- Data Encryption:
    - At rest: AES-256 encryption for data in S3 and PostgreSQL.
    - In transit: All communication between services and users will be secured using HTTPS and mTLS.

Deployment and DevOps

The system's infrastructure will be managed using Infrastructure-as-Code (IaC) tools such as Terraform and Ansible. A robust CI/CD pipeline will be implemented using GitHub Actions or Jenkins, enabling:

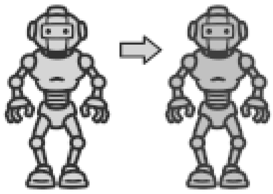
- Unit testing with PyTest and integration testing using Selenium.
- Automated rollback in case of deployment errors, triggered if failure rate exceeds 5% in canary deployments.

Design Patterns

Façade

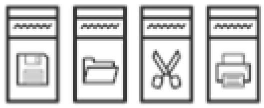
The Façade pattern is used to provide a simplified interface to the complex subsystems within the application. This design allows clients (e.g., frontend components or external APIs) to interact with the system through a unified entry point, hiding the complexity of underlying operations such as track processing, AI training, and data visualization. It promotes loose coupling between components and enhances maintainability by centralizing control logic.

Prototype



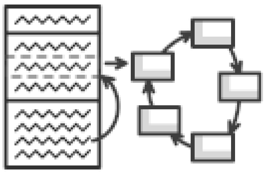
The Prototype pattern is employed to efficiently duplicate existing AI models, track configurations, or lap setups. This is particularly useful when users wish to reuse or slightly modify previously trained models or configurations without reprocessing them from scratch. Deep cloning ensures that replicated objects maintain their own state, avoiding unintended side effects caused by shared references.

Command



The Command pattern encapsulates user actions (such as uploading a track, modifying lap data, or initiating a simulation) as standalone command objects. This abstraction enables queuing, logging, and the ability to implement undo/redo functionalities. By decoupling the invoker from the execution logic, the system gains flexibility in handling user interactions in both the UI and backend workflows.

State



The State pattern allows the system to alter its behaviour dynamically based on its current state. For example, the UI and backend processing logic behave differently depending on whether a track is being uploaded, a model is in training, or results are ready for visualization. This pattern ensures that transitions between states (e.g., Idle → Processing → Completed) are handled cleanly and predictably, improving the system's reliability and user experience.

Constraints

Access to Real-World Telemetry:

Obtaining authentic racing telemetry for supervised learning models may pose a challenge. As a result, alternative sources such as data from racing simulators or games may need to be utilized.

Model Reliability and Accuracy:

The outputs generated by the AI must be carefully compared against established racing techniques and strategies to ensure they are both accurate and dependable.

Complexity in Image Analysis:

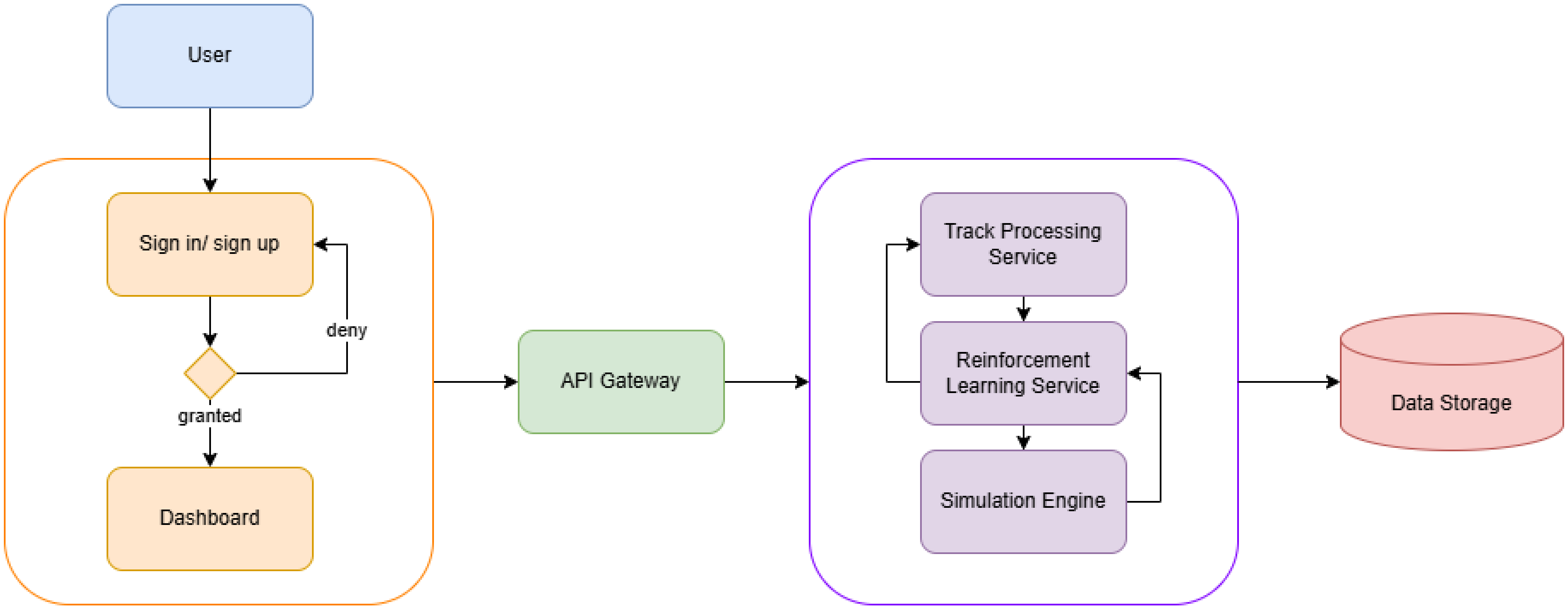
The system must be capable of accurately processing track images, particularly in identifying circuit boundaries and optimal racing paths. Misinterpretations at this stage could compromise the entire prediction pipeline.

Computational Resource Demands:

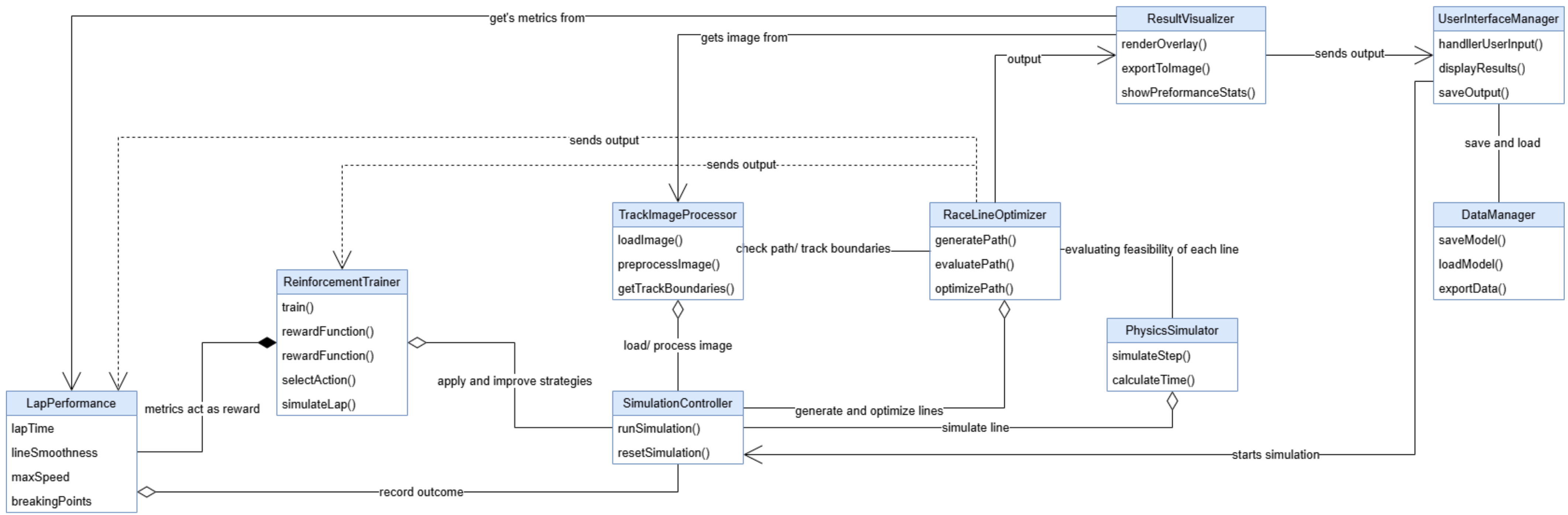
Reinforcement learning processes are computationally intensive and require adequate hardware resources, such as GPUs or cloud-based solutions, to train effectively within a reasonable timeframe.

Diagrams and Models

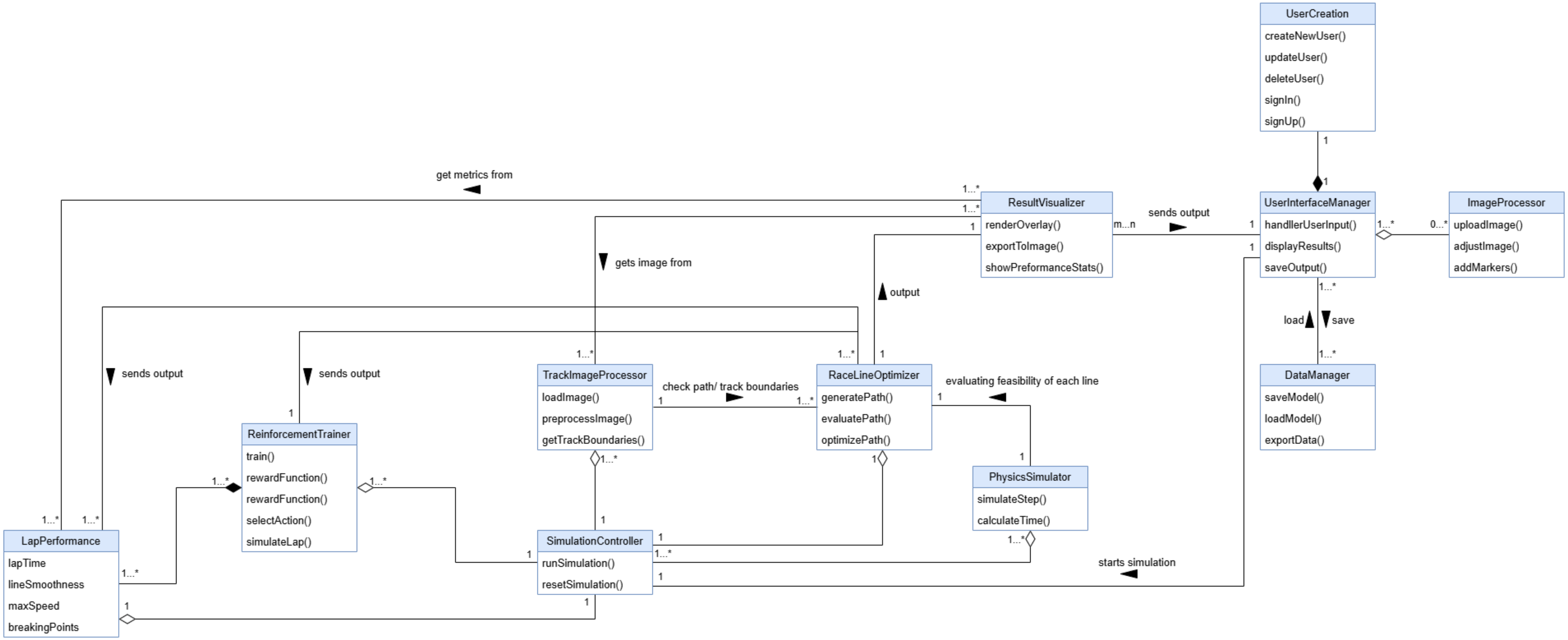
Architecture Diagram



Class Diagram



Domain Model



Quintessential
ctprojectteam3@gmail.com

Deployment Diagram

[Needs to be created]

Manuals

Installation Manual

[Needs to be created]

Technical Installation Manual

[Needs to be created]

User Manual

[Needs to be created]

Specifications and Standards

Machine Learning Specification

[Needs to be created]

API Documentation

[Needs to be created]

Coding Standards

[Needs to be created]

Testing Policy

Testing Scope & Levels

Level	Focus	Tools/Methods
Unit Testing	Individual functions (e.g: track image processing, RL reward function).	Pytest (Python), JUnit (Java).
Integration Testing	Interaction between services (e.g: track processor → RL engine).	Postman, Jest (API tests), Selenium (UI flows).
System Testing	End-to-end workflows (e.g: upload image → simulate → visualize).	Cypress, Robot Framework.
Performance Testing	Scalability (e.g: 50 concurrent users), RL training speed.	Locust (load testing), NVIDIA Nsight (GPU profiling).
Security Testing	Data encryption, auth vulnerabilities.	OWASP ZAP, SonarQube.
User Acceptance (UAT)	Real-world usability (by target users).	Beta releases, A/B testing.

Testing Types & Frequency

Test Type	Description	Frequency
Automated Regression	Validate existing features after updates.	On every Git commit (CI/CD).
Manual Exploratory	Unscripted UX/edge-case testing.	Before major releases.
Physics Validation	Compare AI racing lines against known heuristics (e.g: apex accuracy).	Per RL model update.

Entry & Exit Criteria

Entry Criteria (Tests Start When):

- Requirements are documented.
- Code is merged to the test branch.
- Test environment mirrors production (GPU-enabled).

Exit Criteria (Tests Pass When):

- **Unit/Integration:** ≥90% code coverage (measured via Coveralls).
- **Performance:** <2s response time for track processing; RL training FPS ≥30.
- **Security:** Zero critical OWASP vulnerabilities.
- **UAT:** ≥80% positive feedback from beta testers.

Defect Management

- **Severity Levels:**
  - **Critical** (Crash/data loss): Fixed within 24h.
  - **Major** (Feature failure): Fixed in next sprint.
  - **Minor** (UI glitch): Backlogged for prioritization.
- **Tracking:** Jira/Linear with labels (bug, reproducible, blocker).

Environments

Environment	Purpose	Access
Development	Feature development.	Engineers only.
Staging	Pre-production (mirrors prod).	QA/Product Team.
Production	Live user-facing system.	Automated deployments only.

Test Data Management

- **Realistic Datasets:**

- 10+ sample tracks (F1, MotoGP circuits).
- Synthetic data from racing sims (Assetto Corsa).

- **Anonymization:** User-uploaded tracks scrubbed of metadata.

Compliance & Reporting

- **Audits:** Monthly test coverage/review meetings.
- **Reports:** Dashboards for:
  - Test pass/fail rates.
  - Performance trends (e.g: lap time prediction accuracy).

Policy Exceptions

- **Emergency Fixes:** Hotfixes may bypass some tests but require:
  - Post-deployment regression testing.
  - Retrospective review.

Contribution of Teammates

Project Manager

Amber Werner

[list contributions]

Backend Developers

Qwinton Knocklein

[list contributions]

Sean van der Merwe

[list contributions]

Front End Developers

Simon van der Merwe

[list contributions]

Milan Kruger

[list contributions]

Version 2 [26/05/2025]

Current Document.