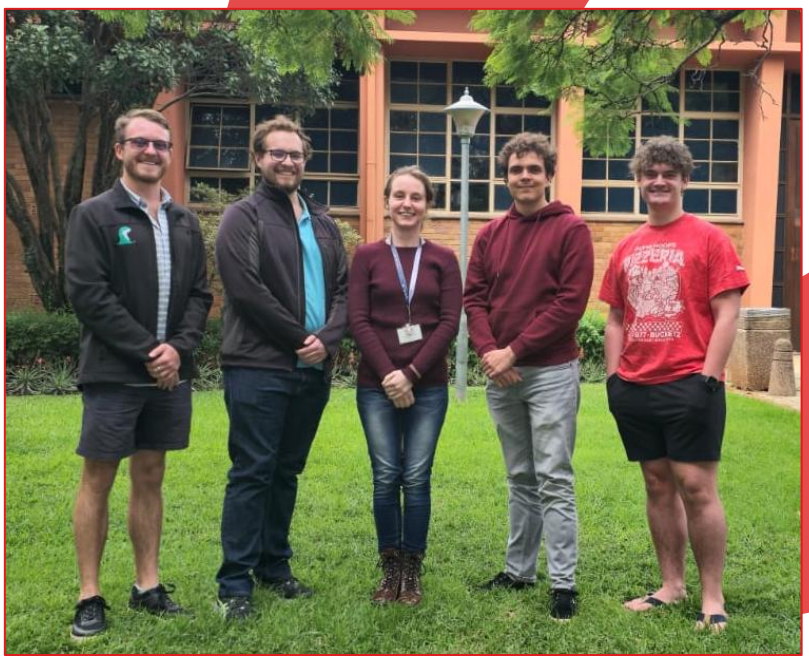
SuperLap Racing Line Optimization System

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

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 the AI-generated optimal racing line overlaid on the track so that I can compare it to my existing racing strategy.
3. 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.
4. 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.

5. 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.
6. 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.
7. 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

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

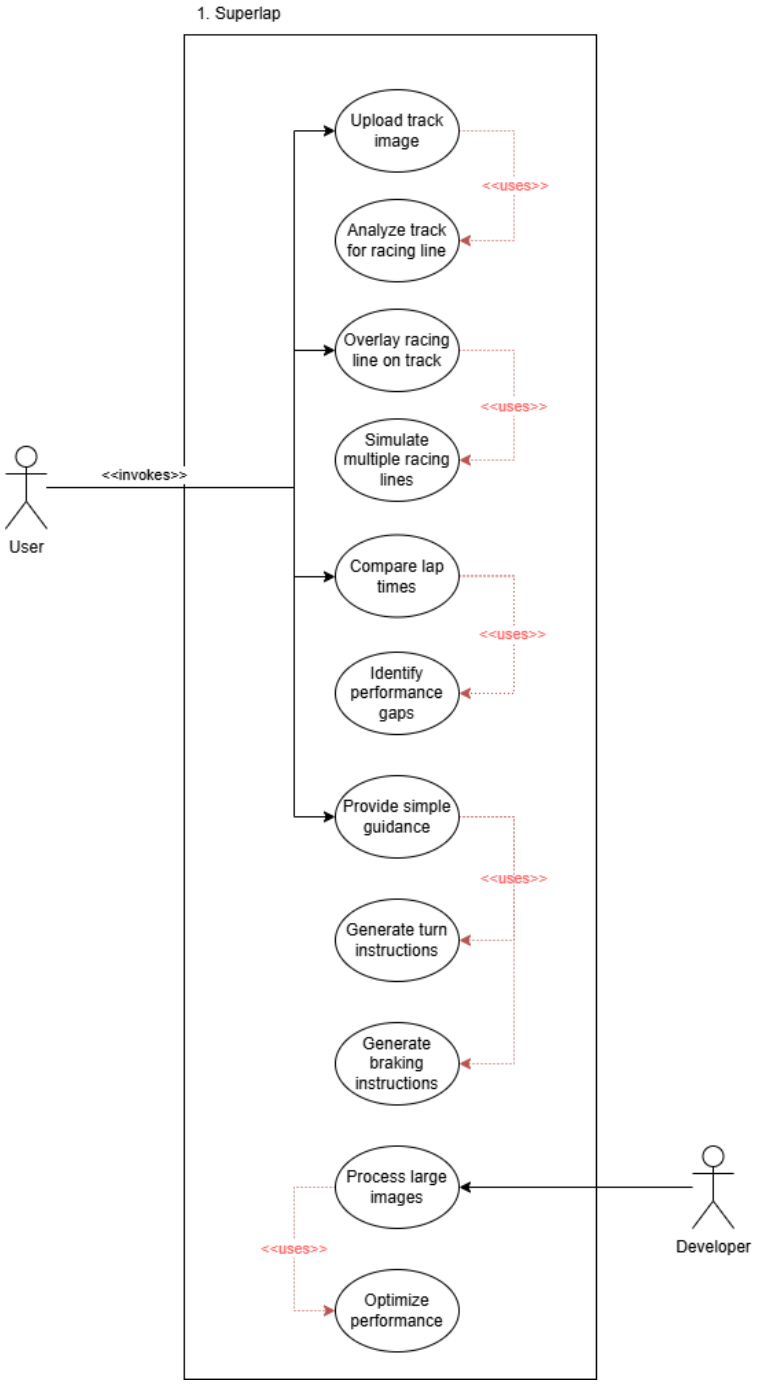
1. As a backend developer, I want the system to efficiently process large track images to reduce wait time for the user.
2. As a power user, I want to configure AI training parameters (e.g: epsilon decay, learning rate) for custom experiments.
3. As a team, we want to store training sessions and model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

Gamification & Community Stories

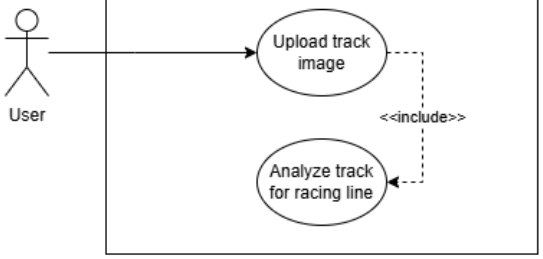
1. As a user, I want to share my best lap and AI-optimized strategy with others to compare and compete.

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.

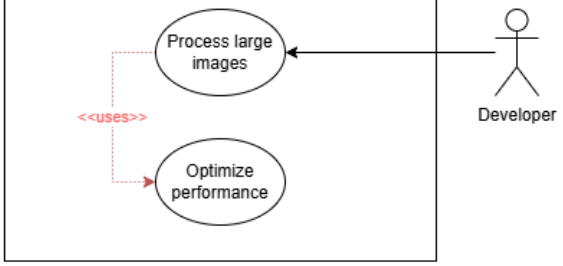
Use Case



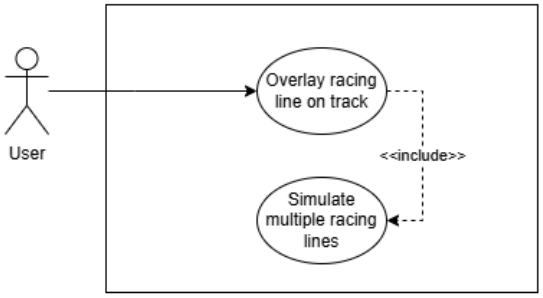
1.1 Image Upload



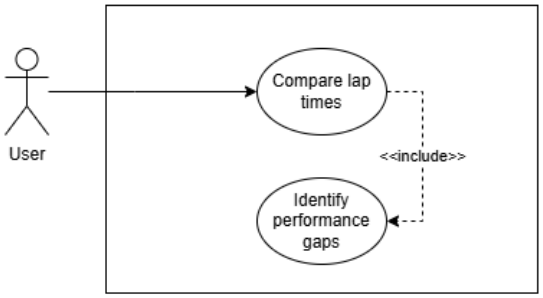
1.4 Image Processing



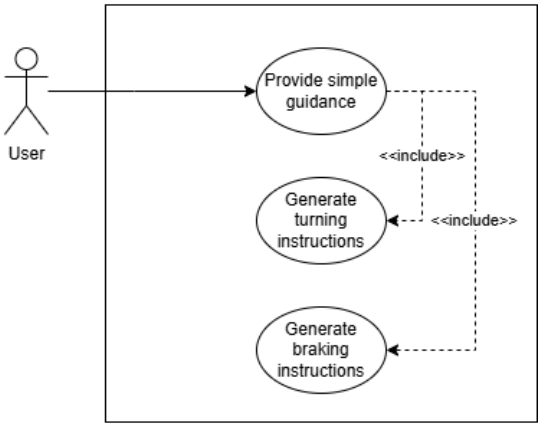
1.2 Racing Line Generation



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

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

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:

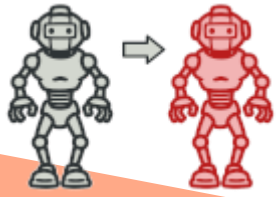
- Unit testing with PyTest and integration testing using Selenium.
- Automated rollback in case of deployment errors, triggered if failure rate exceeds 5% in canary deployments.

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



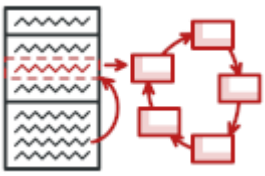
The Prototype pattern is employed to efficiently duplicate existing AI models, track configurations, or lap setups. This is particularly useful when users wish to reuse or slightly modify previously

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:

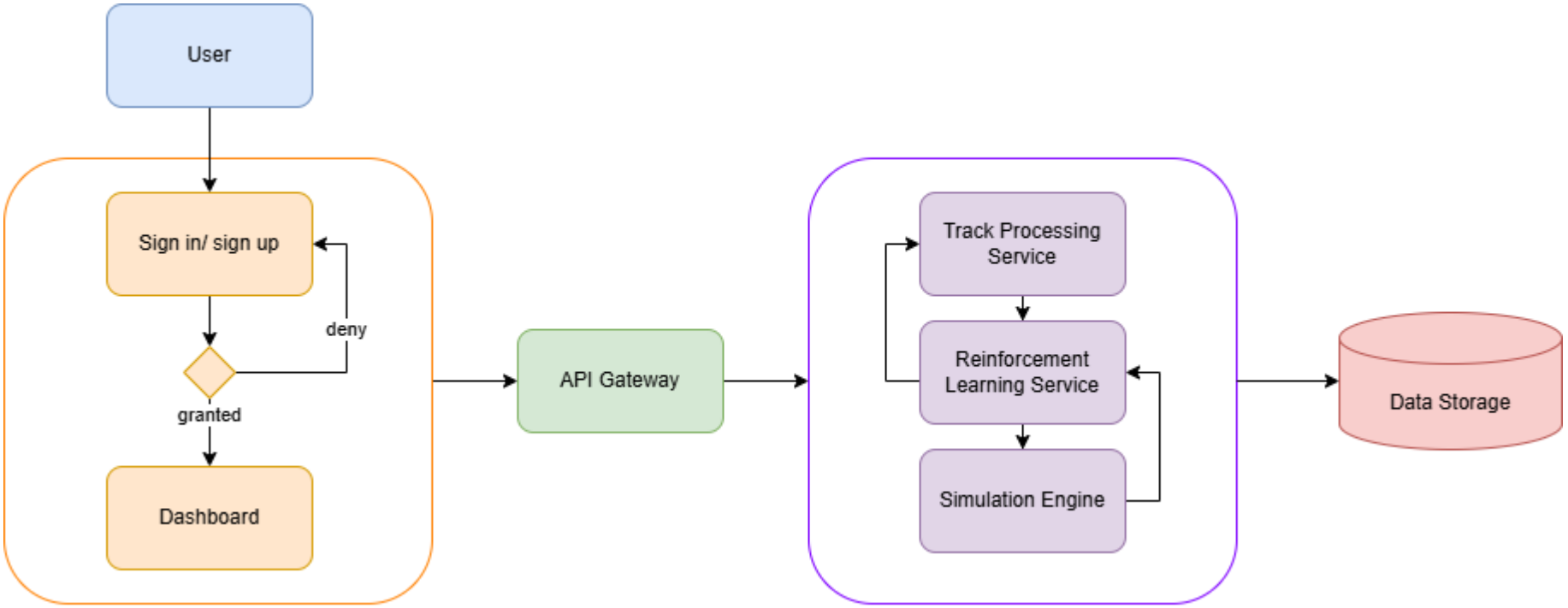
The system must be capable of accurately processing track images, particularly in identifying circuit boundaries and optimal racing paths. Misinterpretations at this stage could compromise the entire prediction pipeline.

Computational Resource Demands:

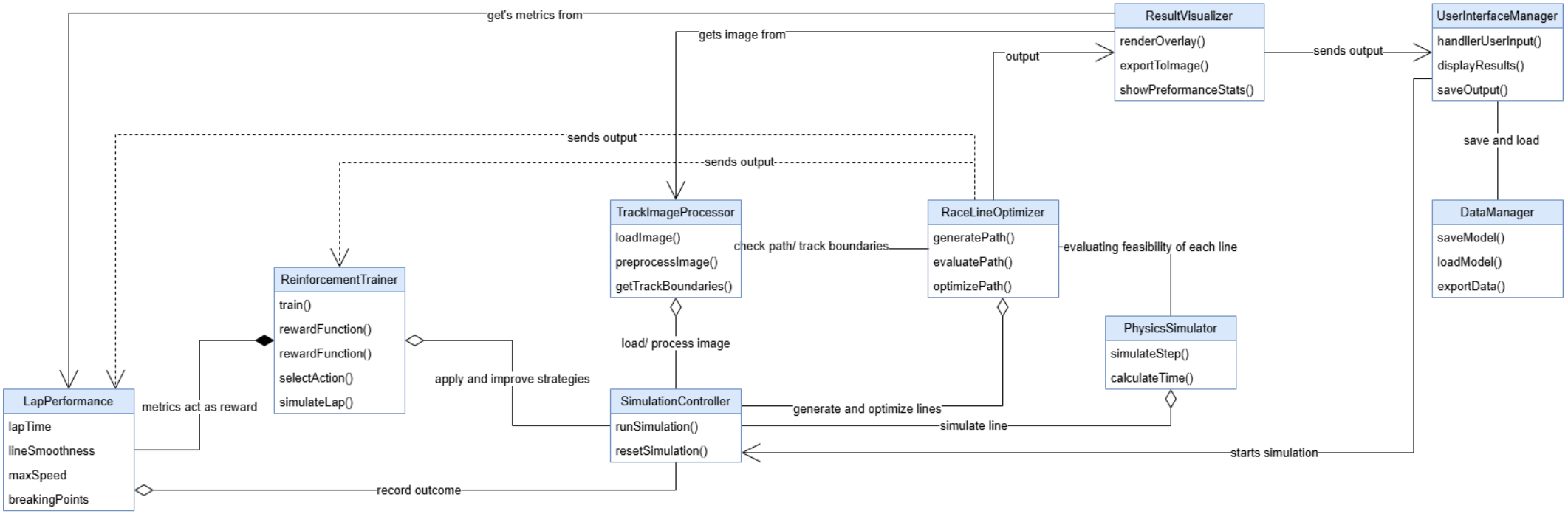
Reinforcement learning processes are computationally intensive and require adequate hardware resources, such as GPUs or cloud-based solutions, to train effectively within a reasonable timeframe.

Diagrams and Models

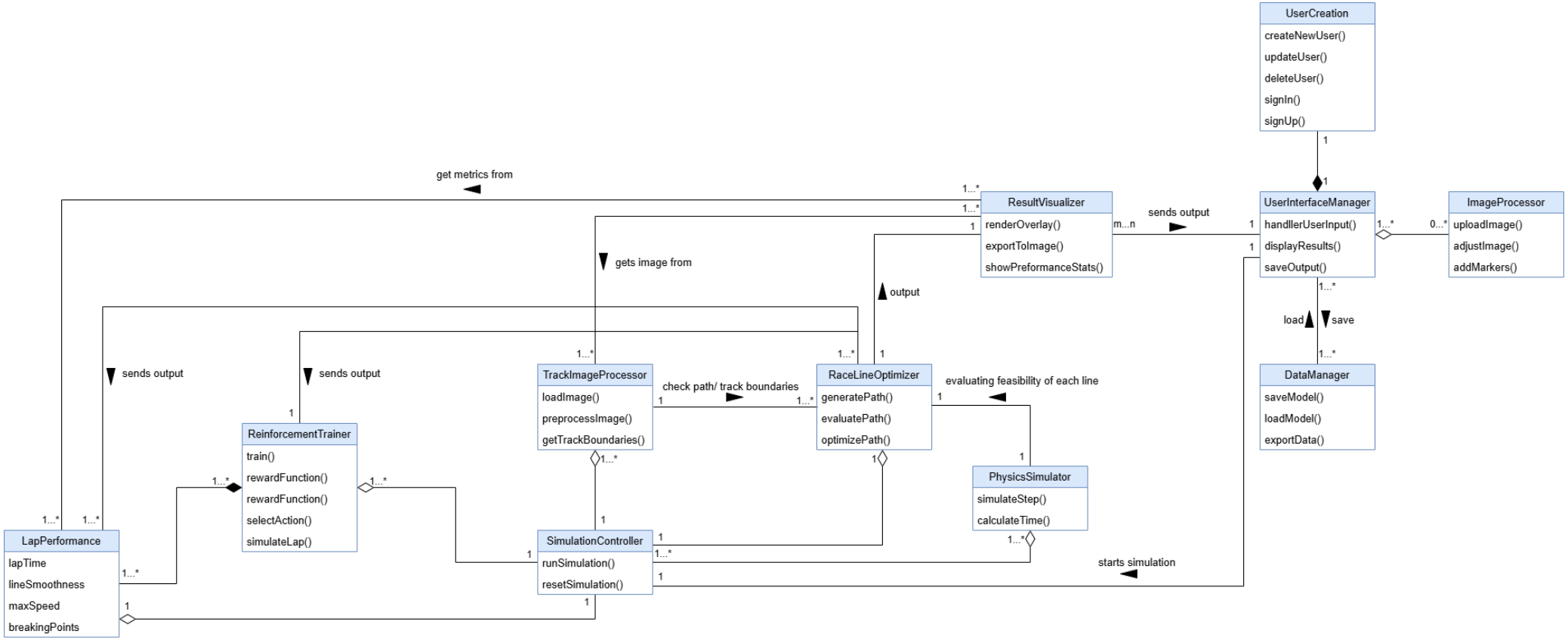
Architecture Diagram



Class Diagram



Domain Model



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.
Hardware Compatibility	GPU/CPU performance benchmarks.	Quarterly.

Entry & Exit Criteria

Entry Criteria (Tests Start When):

- Requirements are documented (e.g: FR/NFRs).
- 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]

Current Document.