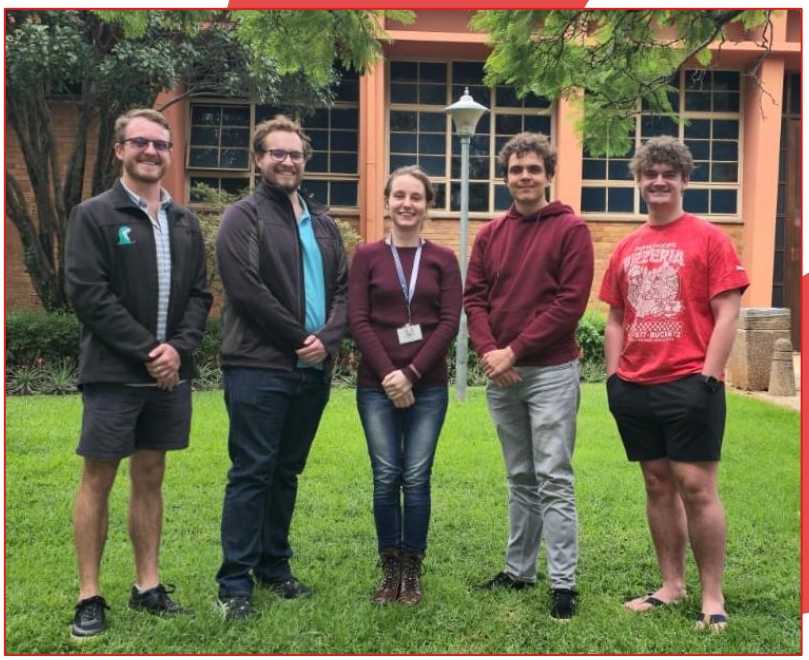
SuperLap Racing Line Optimization System

EPI-USE



Quintessential

Amber Ann Werner [u21457752]

Milan Kruger [u04948123]

Qwinton Knocklein [u21669849]

Sean van der Merwe [u22583387]

Simon van der Merwe [u04576617]

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:

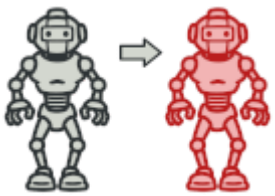
- Unit testing with PyTest and integration testing using Selenium.
- Automated rollback in case of deployment errors, triggered if failure rate exceeds 5% in canary deployments.

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



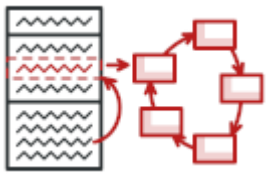
The Prototype pattern is employed to efficiently duplicate existing AI models, track configurations, or lap setups. This is particularly useful when users wish to reuse or slightly modify previously

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.