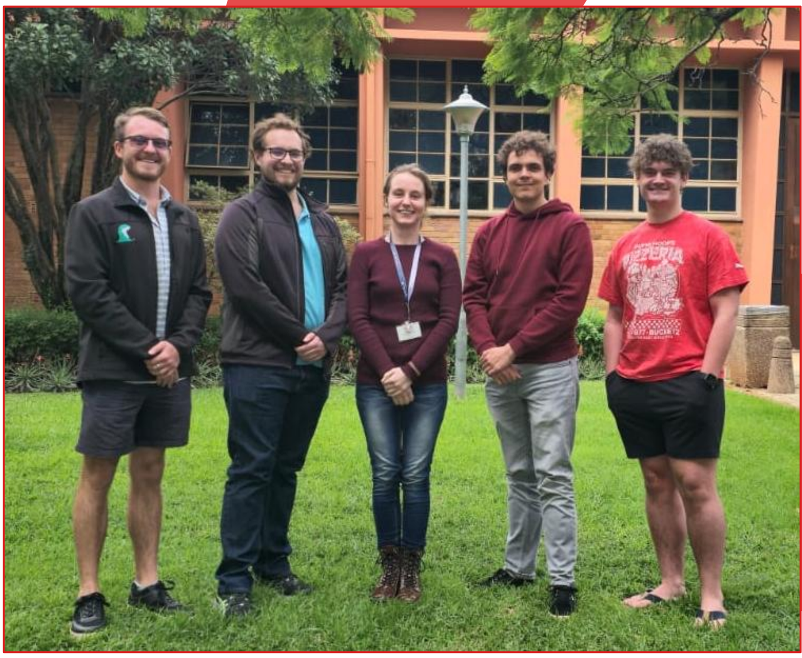
SuperLap Racing Line Optimization System

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

Architectural Design Strategy

This system adopts a **Design Based on Quality Requirements** strategy to guide architectural decisions. A diverse set of non-functional requirements – including real-time performance, security, scalability, and constraints related to maintainability, availability, and cost – necessitated a quality-driven approach.

These requirements informed key architectural choices such as:

- The adoption of an Event-Driven Architecture to support responsiveness and decoupling,
- The use of GPU offloading and model caching to meet real-time performance goals,
- Implementation of API gateways and role-based access control for security enforcement,
- A microservices-based structure combined with infrastructure-as-code for maintainability and scalable deployment.

By deriving architectural patterns from quality attributes, the system maintains alignment with both stakeholder expectations and technical constraints from the outset. This strategy ensures the architecture remains robust, adaptable, and performance-optimized under real-world conditions.

Architectural Strategies

NF1: Performance Requirements

Quality Requirement	Architectural Strategy
NF1.1: Process image ≤10MB in <5s	Image preprocessing optimization (OpenCV optimizations), GPU acceleration
NF1.2: AI training at ≥30 FPS	GPU offloading, model quantization, asynchronous training loops
NF1.3: Lap time prediction <1s	Precompiled inference graphs, model caching in memory
NF1.4: 50 concurrent users	Load balancing, horizontal scale-out, connection pooling

NF2: Security Requirements

Quality Requirement	Architectural Strategy
NF2.1: Encrypted in transit	TLS 1.2+ (HTTPS)
NF2.2: Secure data storage	Password hashing, environment-based secrets management
NF2.3: RBAC	Token-based access (JWT), access control middleware, claims-based auth
NF2.4: Model/data protection	Immutable infrastructure for models, audit logging, signed artifacts

NF3: Reliability & Availability

Quality Requirement	Architectural Strategy
NF3.1: 95% uptime	Multi-zone deployment, cloud managed services, health checks
NF3.2: Auto recovery	Self-healing, watchdog services
NF3.3: Backups	Scheduled backups, versioned S3 storage, RPO strategies
NF3.4: Offline mode	Local storage fallback, local-first design, PWA support

NF4: Usability Requirements

Quality Requirement	Architectural Strategy
NF4.1: Intuitive UI	Component-based UI frameworks, drag-drop file upload modules

NF4.2: Colorblind support	WCAG-compliant colour palettes, accessibility libraries
NF4.3: Tutorials/tooltips	Guided onboarding, contextual tooltips
NF4.4: Confirmation before actions	Modal confirmations, form validation with rollback support

NF5: Scalability Requirements

Quality Requirement	***Architectural Strategy***
NF5.1: Scale to 10k/day	Horizontal scaling, job queueing, microservices
NF5.2: Modular updates	Plugin-based architecture, interface-based module boundaries
NF5.3: GPU scaling	Dynamic resource allocation (GPU scheduling), auto-scaling groups

NF6: Compatibility Requirements

Quality Requirement	***Architectural Strategy***
NF6.1: Desktop OS support	Cross-platform builds, OS-level abstraction layers
NF6.2: Web browser support	Progressive Enhancement, modern HTML5/JS frameworks
NF6.3: Multiple image formats	Unified image handler (OpenCV), file-type validation

NF7: Maintainability Requirements

Quality Requirement	***Architectural Strategy***
NF7.1: Documentation & version control	CI/CD with documentation checks, automated doc generators
NF7.2: Logging	Structured logging (e.g., JSON logs), ELK/EFK stacks
NF7.3: Stable dependencies	Dependency pinning, semantic versioning, lockfiles

NF8: Cost & Resource Constraints

Quality Requirement	***Architectural Strategy***
NF8.1: Cloud cost control	Budget-aware scaling policies, serverless functions for infrequent tasks
NF8.2: Low-spec hardware support	Model quantization, reduced-resolution processing, optional GPU fallback

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

NF1: Performance Requirements

Strategy	***Architectural Pattern***
Image preprocessing optimization	**Pipes and Filters** *(for sequential image processing steps)*
GPU offloading for AI training	**Compute-Intensive Component Offloading** *(not a classic GoF pattern, but used in distributed AI systems)*
Model caching in memory	**In-Memory Cache Pattern** *(e.g., Redis-based caching)*
Load balancing, horizontal scaling	**Microservices + Load Balancer** *(e.g., Kubernetes Services, NGINX)*

NF2: Security Requirements

Strategy	***Architectural Pattern***
HTTPS/TLS communication	**API Gateway** *(with TLS termination and auth validation)*
Secure storage (bcrypt, PBKDF2)	**Zero Trust Security Model** *(with secure storage & access layers)*
Role-based access control (RBAC)	**Access Control Pattern** *(Authorization Layer in API Gateway or Service Mesh)*
Protect model integrity	**Immutable Infrastructure Pattern, Service Mesh with mTLS** *(e.g., Istio)*

NF3: Reliability & Availability

Strategy	***Architectural Pattern***
Multi-zone deployment, cloud health checks	**Microservices + Service Discovery Pattern** *(with failover)*
Auto recovery on failure	**Self-Healing Architecture** *(Kubernetes Pod Restart Policies)*
Scheduled backups, versioned data	**Backup and Restore Pattern**
Offline fallback mode	**Client-Side Processing + Service Worker** *(PWA or Electron App support)*

NF4: Usability Requirements

Strategy	***Architectural Pattern***

Component-based UI (React/Unity)	**Model-View-Controller (MVC)**
Guided onboarding, modals, tooltips	**Presentation-Abstraction-Control (PAC)** *(sometimes used with rich UIs)*
User confirmation flows	**Command Pattern** *(paired with undo/redo logic)*

NF5: Scalability Requirements

Strategy	*Architectural Pattern*
Job queueing, horizontal scale	**Event-Driven Architecture (EDA) + Microservices**
Plugin-based architecture	**Plugin Architecture** *(or Component-Based Software Engineering)*
GPU scaling, Kubernetes	**Elastic Infrastructure Pattern** *(auto-scaling groups, GPU nodes)*

NF6: Compatibility Requirements

Strategy	*Architectural Pattern*
Cross-platform app support	**Cross-Platform Architecture Pattern** *(e.g., Electron)*
Responsive UI for browsers	**Progressive Web App (PWA) Pattern**
File-type abstraction	**Adapter Pattern** *(for converting file formats to internal representations)*

NF7: Maintainability Requirements

Strategy	*Architectural Pattern*
CI/CD with rollback	**Continuous Delivery Pattern** *(Blue-Green or Canary Deployments)*
Structured logging	**Observer Pattern** *(for event-based logging systems)*
Dependency pinning	**Immutable Infrastructure Pattern** *(also relates to CI/CD pipeline design)*

NF8: Cost & Resource Constraints

Strategy	*Architectural Pattern*
Budget-aware scaling	**Serverless Pattern** *(e.g., AWS Lambda for track preprocessing)*
Local fallback support	**Offline-First Pattern** *(especially for Electron/desktop apps)*

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.