Regatta Data Platform Documentation

Architecture & Decisions

None

None

Table of contents

1. Regatta Data Platform Documentation

This documentation describes the architecture, backend system and API of the Regatta Data Platform.

The platform is a fully functional data system for sailing regattas, including data ingestion, processing pipelines and a public API layer.

The goal of this documentation is to provide a clear and structured overview of the system for developers, collaborators and AI-assisted development tools.

1.1 Current Status

The platform currently includes:

- Data ingestion pipelines (web scraping and PDF parsing)
- ETL processing and canonical data model
- Fully implemented FastAPI backend
- Structured logging and execution tracking
- API endpoints for exploring regattas, boats and related entities

A frontend application is currently under development.

1.2 Download Documentation

You can download the complete documentation or access the Markdown source.

Download PDF{ .md-button .md-button--primary }

Download ZIP{ .md-button }

1.3 Documentation Structure

1.3.1 Project

High-level information about the project and its evolution.

- Project overview
- Project roadmap

1.3.2 Architecture

Technical documentation describing how the system is structured.

- System architecture
- Data model

1.3.3 API

Documentation of the FastAPI backend and available endpoints.

- API overview
- Endpoint structure
- Data navigation model

1.3.4 Architectural Decisions

Architecture Decision Records (ADRs) documenting key design choices.

1.3.5 AI Development Context

Guidelines for AI-assisted development and how AI tools interact with the project.

1.4 Purpose of this Documentation

This documentation exists to:

- document the architecture and backend system of the platform
- provide governance and technical clarity
- support AI-assisted development workflows
- ensure long-term maintainability of the system
- provide a clear interface for interacting with the data via the API

1.5 Vision

The long-term goal of the platform is to become a comprehensive data exploration tool for sailing regattas, combining structured data, analytics and user-facing applications.

2. Ai context

2.1 AI Development Context

This section contains documentation designed to support **AI-assisted development as an integrated part of the project workflow**.

The goal is to provide structured context that enables AI tools to understand the architecture, design principles and constraints of the system, allowing them to generate **consistent and architecture-aligned code**.

AI is used within the project as a development tool to assist in:

- code generation and refactoring
- architectural reasoning and validation
- documentation creation and maintenance
- iterative development workflows

These documents ensure that AI-generated outputs remain consistent with the system architecture, data model and established design decisions.

2.1.1 Role of AI in the Project

AI is treated as a **development accelerator**, not as an autonomous system.

All AI-generated outputs are:

- reviewed by a human developer
- aligned with architectural principles
- integrated through controlled iteration

This approach ensures that AI enhances productivity without compromising system integrity.

2.1.2 Contents

- **AI Development Context** – guidelines and architectural constraints for AI-assisted development workflows.

2.2 AI Development Context

This document provides guidance for AI-assisted development tool working within this repository.

Its purpose is to help AI systems understand the structure of the project and the architectural constraints that must be respected when generating or modifying code.

AI tools are used in this project as a **development accelerator**, but all generated code remains subject to human review.

2.3 Project Summary

The Regatta Data Platform is a structured data system designed to collect, normalise and expose information about sailing boats.

The system ingests regatta result pages and extracts structured information about boats and their associated entities.

The primary focus of the dataset is **boats and their relationships**, not race results.

Core entities include:

- boats
- owners
- clubs
- boat classes
- regattas and regatta editions

Regattas act primarily as **data sources for discovering boats and metadata**, rather than as the central dataset.

2.4 Technology Stack

Current core technologies:

- Python (data processing and backend)
- PostgreSQL (canonical database)
- FastAPI (API layer)
- Playwright (web scraping)
- Docker (containerisation)

Additional components include:

- CLI-based pipeline execution
- structured logging system
- centralised configuration management

The development environment is containerised using Docker to ensure reproducibility across systems.

2.5 Repository Structure

The repository is organised around a layered architecture.

Main components include:

- **scraping/** Scrapers responsible for extracting structured data from external sources.
- **pipelines/** ETL pipelines that process and transform data into the canonical model.
- **app/** Core application layer, including:
- API (FastAPI routes)
- services (business logic)
- repositories (database access)
- **scripts/** CLI tools for executing pipelines and utilities.
- **data/** Local storage for intermediate and generated data.
- **logs/** Persistent logs for pipeline execution and debugging.

2.6 Pipeline Execution Model

Pipelines are executed through a CLI-based system, allowing individual pipelines to be run independently.

This enables:

- modular execution
- easier debugging and monitoring
- flexible development workflows

Each pipeline is responsible for a specific domain (boats, classes, regattas, scraping, etc.).

2.7 Architectural Principles

The system follows several architectural principles that must be respected by generated code.

Database-first architecture

PostgreSQL acts as the **central system of record**.

All ingestion and processing workflows operate around the database rather than file-based storage.

Raw vs canonical data separation

The database contains separate schemas for:

- raw ingestion data
- canonical relational entities

Raw data is stored unchanged and serves as the reproducible input for the normalisation pipeline.

Flexible ingestion layer

Scraped raw data is stored as JSONB to support heterogeneous source structures.

Normalisation pipelines transform this data into structured relational entities.

Entity-centric data model

The canonical dataset focuses on boats and their relationships.

Regattas act primarily as contextual sources of information.

Observability

The system uses a structured logging system across pipelines and scraping modules.

Logs provide:

- execution tracking
- warnings and error reporting
- debugging support

Logs are stored persistently and are used to monitor pipeline behaviour.

2.8 Architectural Decisions

Important architectural decisions are documented in the **ADR (Architecture Decision Record)** documents located in:

```
docs/decisions/
```

Examples include:

- database-first architecture
- separation of raw and canonical data
- JSONB storage for ingestion
- entity-centric data model

When generating code or suggesting architectural changes, these decisions must be **treated as constraints**.

If a proposed change conflicts with an ADR, the AI should highlight the conflict rather than silently ignoring the documented decision.

2.9 Expected AI Contribution

AI tools may assist with:

- generating modules
- implementing pipelines
- writing API endpoints
- suggesting refactors
- improving documentation

AI-generated code should:

- follow the existing architectural structure
- respect the database schema and ADRs
- avoid introducing unnecessary abstractions
- prioritise clarity and maintainability

Large changes should be broken into clearly explained steps.

2.10 Development Workflow

AI-generated code should be treated as **draft implementations** that require human validation.

Recommended workflow:

1. AI proposes implementation or module.
2. Human reviews architecture and logic.
3. Code is tested locally.
4. Adjustments are made before integration.

AI systems should prioritise **readable and explicit code** over overly complex abstractions.

2.11 Future Stack Evolution

The backend will remain Python-based.

A frontend layer is currently under development using a JavaScript framework (React or similar).

The API layer will expose structured access to the canonical dataset.

2.12 Guiding Principle

AI is treated as a **capability multiplier**, not as an autonomous decision-maker.

Architectural integrity and data correctness always take priority over development speed.

3. Api

3.1 Regatta Data API

This document describes the available endpoints of the Regatta Data API.

The API is designed around a hierarchical navigation model:

Regattas → Editions → Boats → Related entities (Classes, Clubs, Owners)

3.1.1 Search

Global search across entities.

GET /search?q={query}

Returns matching: - Boats - Regattas - Classes

3.1.2 Regattas

Access regatta information and related data.

GET /regattas
→ List all regattas

GET /regattas/{id}
→ Get detailed information about a specific regatta

GET /regattas/{id}/editions
→ List all editions of the regatta

GET /regattas/{id}/links
→ External sources (official pages, results)

3.1.3 Editions

Access information about a specific regatta edition.

GET /editions/{id}
→ Edition metadata (year, status, regatta)

GET /editions/{id}/boats
→ Boats participating in the edition

GET /editions/{id}/classes
→ Classes present in the edition

3.1.4 Boats

Access boat data and relationships.

GET /boats
→ List boats (supports future pagination)

GET /boats/{id}
→ Detailed boat information: - class - type - owners - clubs

GET /boats/{id}/owners
→ Boat owners

GET /boats/{id}/clubs
→ Boat clubs

GET /boats/{id}/editions
→ Participation history

3.1.5 Classes

Access boat class information.

GET /classes
→ List all classes

GET /classes/{id}
→ Class details

GET /classes/{id}/boats
→ Boats belonging to the class

GET /classes/{id}/types
→ Types within the class

3.1.6 Clubs

Access club information.

GET /clubs
→ List all clubs

GET /clubs/{id}
→ Club details

GET /clubs/{id}/boats
→ Boats associated with the club

GET /clubs/{id}/regattas
→ Regattas organized by the club

3.1.7 Design Notes

- The API follows an **entity-centric model focused on boats**

- Relationships are exposed through dedicated endpoints
- The API is designed for **exploration and analysis**, not just retrieval

4. Architecture

4.1 Data Model

4.1.1 Overview

The data model is designed to build a structured registry of boats and their associated entities.

While regatta results are used as the primary source of information, the goal of the database is not to store detailed race results. Instead, regatta appearances act as a **discovery mechanism** for identifying boats and extracting structured information about them.

The canonical dataset therefore focuses on **boats and their relationships**, including owners, clubs, and classes.

The model follows a layered structure:

Raw Data → ETL Pipelines → Canonical Entities

4.1.2 The canonical dataset is exposed through a structured API, enabling exploration and integration with external applications.

4.1.3 Raw Data Layer

Raw data is populated through ingestion pipelines and serves as the entry point for all downstream transformations.

Raw scraped data is stored in the `yacht_raw` schema.

The table:

```
yacht_raw.raw_regatta_results
```

stores the extracted information for each regatta page as a JSON object.

Each record typically includes:

- source metadata
- regatta name
- year
- scraped timestamp
- structured raw data (`jsonb`)

This layer preserves the original extracted values and allows all transformations to be reproducible and traceable.

Raw data is never modified or deleted.

4.1.4 Canonical Entity Model

The canonical schema (`yacht_db`) represents the structured dataset used by the system.

The model is centred around a set of core entities.

The canonical dataset is actively used by the API layer to provide structured access to boats and their relationships.

Boats

The `boats` table represents the canonical identity of a boat.

Each boat is defined by:

- boat name
- boat identifier (typically a sail number)
- associated boat class

Boat identity is resolved during the normalisation process using:

- cleaned sail numbers
- fuzzy name matching
- manual review where required

This allows the system to recognise the same boat appearing across multiple regattas even when naming conventions differ.

Owners

The `owners` table stores the canonical identities of boat owners.

A boat may have multiple owners.

The relationship is represented through the junction table:

```
boats_owner
```

which allows a many-to-many relationship between boats and owners.

Clubs

The `clubs` table represents sailing clubs associated with boats.

Club records may include additional metadata such as:

- location
- short name
- estimated number of members

Boats may be affiliated with multiple clubs, represented through the junction table:

```
boat_clubs
```

Boat Classes

The `boat_classes` table represents the general class of a boat, such as:

- Dragon
- Etchells
- J/70

Classes may include metadata such as:

- manufacturer
- rating rule
- crew size
- hull length

Boat Types

The `boat_type` table provides a more specific classification within a class, representing particular boat models or variants.

This allows the system to distinguish between boats that belong to the same class but have different design variants.

Regattas and Editions

Regattas are represented by two related tables:

```
regattas
regatta_editions
```

The `regattas` table represents the event itself, while `regatta_editions` represents a specific year of that event.

For example:

```
Cowes Week → Regatta
Cowes Week 2025 → Edition
```

This separation avoids duplication of event metadata across years.

Boat Participation

The table:

```
boat_editions
```

records the participation of boats in specific regatta editions.

This information is used primarily as contextual metadata and as a discovery signal for identifying boats and their relationships.

The system does not aim to store full race result analytics at this stage, focusing instead on entity discovery and relationship mapping.

Locations

The `locations` table provides geographic metadata used by both clubs and regattas.

Typical attributes include:

- city
- region
- country

This allows geographic grouping and analysis of clubs and events.

4.1.5 Entity Relationships

The model contains several many-to-many relationships:

Boat ↔ Owner Boat ↔ Club Boat ↔ Regatta Edition Edition ↔ Class

These relationships are implemented using junction tables to preserve flexibility and avoid duplication.

4.1.6 Normalisation Workflow

Raw values extracted from source pages are rarely consistent.

Normalisation currently follows a hybrid workflow:

1. Raw values are extracted from the JSON ingestion layer.
2. Lists of unique raw values are generated.
3. These values are reviewed and mapped to canonical entities.
4. Canonical entities are inserted into the database.

Mapping rules are currently stored in CSV lookup files but will eventually be migrated into database-managed mapping tables.

Normalisation is integrated into the ETL pipeline layer, enabling consistent transformation and insertion into the canonical database.

4.1.7 Data Usage

The canonical data model is designed to support:

- API-based exploration of boats and regattas
- relationship analysis between entities
- future frontend applications
- incremental enrichment of the dataset over time

The model prioritises flexibility and extensibility to support future analytical and product layers.

4.1.8 Design Principles

The data model follows several key principles:

Entity-centric design The system models boats and related entities rather than race results.

Traceability All canonical entities can be traced back to raw source data.

Flexibility Many-to-many relationships allow complex ownership and affiliation structures.

Incremental normalisation Data quality improves over time through iterative normalisation.

Separation of raw and canonical layers Raw scraped data is stored independently from the canonical relational model.

4.2 System Overview

4.2.1 Architectural Philosophy

The system is designed as a **database-centric data platform**.

PostgreSQL acts as the **single source of truth** for all structured data. All ingestion, normalisation and data access workflows are built around the database rather than file-based workflows.

The primary objective of the platform is to build a **structured registry of boats and related entities**, including owners, clubs, and classes.

Regatta results act primarily as a **data discovery source**, allowing the system to identify boats and extract associated information. The focus of the platform is therefore on **entity knowledge and relationships**, rather than on storing or analysing race results themselves.

This architecture ensures:

- traceability from raw data to canonical entities
- reproducibility of ingestion workflows
- clear separation between data storage and data consumption
- scalability as the dataset grows

The platform evolves from an initial script-based workflow toward a structured multi-layer architecture.

4.2.2 High-Level Architecture

The system consists of several layered components:

Sources → Ingestion → Raw Storage → ETL Pipelines → Canonical Data → API → Frontend (in progress)

---

1. Data Sources

The system collects information from heterogeneous external sources, typically regatta result websites.

These sources often contain valuable information about boats and their associated entities but present several challenges:

- inconsistent formatting
- varying naming conventions
- incomplete metadata

Scrapers extract structured information about **boats and related entities** from these sources. While the pages contain race results, the system primarily uses them as a **signal to identify boats and their attributes** such as class, club affiliation, and ownership.

---

2. Ingestion Layer

Scrapers retrieve regatta pages and extract structured information from them.

Each ingestion run collects:

- regatta metadata (as contextual information)
- boats appearing in the regatta
- associated attributes such as class, club, owner, or boat type

The extracted information for each regatta is stored together as a **JSON object**.

Ingestion pipelines are orchestrated through a CLI-based execution system, allowing individual pipelines to be triggered independently.

This enables modular execution, better control over pipeline runs, and improved observability.

Pipeline execution is monitored through a structured logging system, which records execution progress, warnings and errors for debugging and traceability.

---

3. Raw Data Storage

Raw scraped data is stored in PostgreSQL using **JSONB fields**.

Each record represents the raw extracted information for a single regatta and preserves the original values obtained from the source.

This layer acts as an **immutable record of the original source data**, enabling:

- reproducibility of transformations
- debugging of ingestion pipelines
- traceability between source data and canonical entities

Raw data is never modified or deleted.

---

4. Normalisation Pipeline

Raw data is processed through a Python normalisation pipeline.

The objective of this stage is to transform inconsistent raw values into a consistent canonical representation of entities.

Typical transformations include:

- mapping raw club names to canonical clubs
- mapping raw class names to canonical classes
- resolving boat identity across appearances
- validating or correcting inconsistent values

At the current stage, mapping rules are stored in **CSV-based lookup tables** used by the normalisation logic.

These mappings will eventually migrate into database-managed mapping tables.

Normalisation is designed to support a **human-in-the-loop workflow**, where edge cases and ambiguous values can be reviewed and corrected.

Normalisation logic is implemented as part of the ETL pipeline layer, enabling consistent transformations and integration with database operations.

---

5. Canonical Database

After normalisation, structured entities are written to canonical relational tables.

Core entities include:

- boats
- owners
- clubs
- boat classes
- regattas and regatta editions (stored primarily as contextual metadata)

These tables form a **structured registry of boats and their relationships**.

The goal of the canonical dataset is to provide reliable information about boats and associated entities rather than detailed race results.

Each canonical entity can be traced back to the raw data that generated it.

---

6. API Layer

The API layer is fully implemented and actively used to explore and interact with the dataset.

A FastAPI application provides structured access to the canonical dataset.

The API abstracts the database schema and exposes consistent endpoints for querying entities.

Typical responsibilities include:

- retrieving boats and their associated attributes
- exploring relationships between boats, clubs, owners, and classes
- filtering entities by attributes such as class or year
- exposing structured data for applications or analysis

The API layer decouples data storage from data consumption.

The API is designed to support frontend applications and external integrations.

---

7. Frontend Layer (In Progress)

A web-based frontend application is currently being developed on top of the API layer.

This interface is intended to support:

- exploring boats and their associated information
- understanding relationships between entities
- inspecting raw vs normalised values
- reviewing low-confidence mappings
- improving transparency and trust in the dataset

The UX will consume the API rather than interacting directly with the database.

4.2.3 Execution Environment

The system is designed to support both local execution and future containerised deployment in cloud environments.

Typical components include:

- PostgreSQL database container
- ingestion pipeline container
- API container

Containerisation ensures reproducibility and simplifies deployment to future environments such as a VPS or managed cloud infrastructure.

4.2.4 Pipeline Execution Model

Pipelines are executed through a CLI-based system, allowing selective execution of ingestion and transformation processes.

This approach provides:

- modular pipeline execution
- improved debugging and monitoring
- flexibility in development and production workflows

Each pipeline is responsible for a specific domain (boats, classes, regattas, etc.) and can be run independently.

4.2.5 Architectural Principles

The platform follows several key design principles.

Database-first architecture The database is the central system of record.

Raw data preservation Original extracted data is stored unchanged for traceability.

Human-in-the-loop normalisation Ambiguous data is resolved through structured human review.

API-based access All external data access occurs through a controlled API layer.

Layered architecture Each system layer has a clear and independent responsibility.

Entity-focused dataset The platform focuses on building a reliable registry of boats and related entities, using regatta results as a discovery source rather than as the primary analytical dataset.

5. Decisions

5.1 Architectural Decisions

This section contains the **Architecture Decision Records (ADRs)** for the project.

ADRs document important architectural decisions and the reasoning behind them.

They provide historical context for why certain technologies, patterns or structures were chosen.

These documents help ensure that future development respects the architectural principles of the system.

Each ADR includes:

- the context in which the decision was made
- the decision itself
- the consequences for the system

ADRs are immutable records. If a decision changes in the future, a new ADR should be created rather than modifying an existing one.

Current ADRs cover topics such as:

- database architecture
- separation of raw and canonical data
- ingestion storage strategy
- the entity-centric data model

5.2 ADR-001: Adopt a Database-First Architecture Using PostgreSQL

Status: Accepted **Date:** 2026-03-06

5.2.1 Context

The early exploration phase of the project relied on CSV files generated by scraping scripts.

This approach allowed quick experimentation and made it easy to demonstrate to stakeholders that structured data could be extracted from regatta result pages.

However, as the project evolved from experimentation toward a structured data platform, several limitations of a file-based workflow became clear:

- Difficult traceability between raw data and processed data
- Increasing complexity managing multiple CSV files
- Limited ability to enforce relational structure and constraints
- Poor scalability as the dataset grows
- Difficult integration with APIs and future web applications

The project required a more robust and scalable system of record capable of supporting ingestion pipelines, entity normalisation, and application-layer access.

5.2.2 Decision

The system adopts a **database-first architecture**, using **PostgreSQL as the central system of record**.

In this model:

- All raw ingestion data is stored in the database
- Normalised canonical entities are stored in relational tables
- CSV files are used only as temporary artefacts for data review or mapping workflows
- All application logic and APIs operate on top of the database

PostgreSQL was selected because:

- It is a mature and widely used relational database
- It supports complex relational modelling
- It provides excellent JSON support for raw ingestion data
- It integrates easily with modern Python frameworks such as FastAPI
- It supports future scalability and deployment options

5.2.3 Consequences

Adopting a database-first architecture provides several benefits:

- Clear separation between raw data and canonical data
- Improved traceability and reproducibility of transformations
- Strong relational structure for modelling entities and relationships
- Simplified integration with APIs and future applications
- Better scalability as the dataset grows

However, it also introduces some trade-offs:

- Increased initial complexity compared to simple CSV workflows
- Need for database schema design and migrations
- Additional infrastructure requirements for deployment

Overall, the database-centric architecture provides a solid foundation for building a scalable data platform and supports the long-term evolution of the project.

5.3 ADR-002: Separate Raw and Canonical Data Layers

Status: Accepted **Date:** 2026-03-06

5.3.1 Context

During the early stages of the project, scraped data was stored in CSV files before being processed and inserted into the database.

While this approach worked for initial experimentation, it created several problems as the project evolved into a structured data platform:

- Raw data existed outside the main system of record
- It was harder to reproduce transformations
- The ingestion workflow relied on multiple intermediate files
- Debugging and auditing transformations became more difficult

To simplify the architecture and maintain a consistent system of record, raw data ingestion was moved directly into the PostgreSQL database.

However, raw data and canonical data serve fundamentally different purposes and therefore should not share the same schema.

Raw data represents **unprocessed source information**, while canonical tables represent **cleaned and structured entities** used by the application.

5.3.2 Decision

The system separates raw and canonical data into different database schemas.

Two primary schemas are used:

```
yacht_raw
yacht_db
```

The `yacht_raw` schema stores raw ingestion data exactly as extracted by scrapers.

Example table:

```
yacht_raw.raw_regatta_results
```

Each record stores the raw extracted data for a regatta page as a JSONB object along with metadata such as source URL and scrape timestamp.

The `yacht_db` schema stores the canonical relational model used by the platform, including entities such as:

- boats
- owners
- clubs
- boat classes
- regattas and regatta editions

Raw data is treated as **immutable source input**, while canonical tables are generated through the normalisation pipeline.

The normalisation process always runs against the complete raw dataset, ensuring that improvements to the normalisation logic automatically propagate to all records.

5.3.3 Consequences

Separating raw and canonical layers provides several benefits:

- Clear distinction between ingestion data and trusted canonical entities
- Full traceability from canonical entities back to their raw source data
- Ability to reprocess the entire dataset when normalisation logic improves
- Simplified ingestion pipeline (scrapers write directly to the database)
- Reduced risk of accidental modification of source data

It also establishes a clean layered architecture:

```
Scrapers
    ↓
Raw ingestion (yacht_raw)
    ↓
Normalisation pipeline
    ↓
Canonical entities (yacht_db)
```

This design supports incremental improvements to data quality while preserving the original source data used to build the dataset.

The main trade-off is that the system stores both raw and processed data, which increases storage usage. However, the benefits in traceability and reproducibility outweigh this cost.

5.4 ADR-003: Store Raw Scraped Data Using JSONB

Status: Accepted **Date:** 2026-03-06

5.4.1 Context

The system collects information from multiple regatta result pages. These sources are heterogeneous and often inconsistent in the data they provide.

For example, some regatta result pages may include:

- boat name
- sail number
- class
- club
- owner
- boat type

However, other sources may omit some of these attributes or present them in different combinations.

Examples of variation include:

- pages that include owner and class but no club
- pages that include club and class but no owner
- pages that include additional attributes not present in other sources

Because of this variability, the raw data extracted from each regatta cannot reliably fit into a fixed relational structure at the ingestion stage.

Initially, scraped data was exported to CSV files before being processed and inserted into the database. As the architecture evolved toward a database-first model, it became preferable to store the raw data directly inside PostgreSQL.

5.4.2 Decision

Raw scraped data is stored using a **JSONB column** in the raw ingestion table.

Example table:

```
yacht_raw.raw_regatta_results
```

Each row represents the raw data extracted from a single regatta page.

The extracted results are stored as a JSON object containing all records associated with that regatta.

Scraping pipelines typically follow this process:

```
scraper
    ↓
pandas dataframe
    ↓
df.to_dict(orient="records")
    ↓
JSONB storage in PostgreSQL
```

Using JSONB allows the system to store heterogeneous records without enforcing a rigid schema at the ingestion stage.

The responsibility for transforming these records into structured relational entities is delegated to the normalisation pipeline.

5.4.3 Consequences

Using JSONB for raw ingestion provides several advantages:

- Flexibility when handling inconsistent source data
- Ability to store heterogeneous record structures
- Simplified ingestion pipelines
- Reduced schema maintenance at the scraping stage
- Easier debugging of raw extracted data

This approach allows scrapers to focus only on extracting available information, while the normalisation layer handles the complexity of mapping fields into the canonical data model.

The main trade-off is that JSONB data cannot be queried as efficiently as structured relational tables. However, this is acceptable because raw data is primarily used as input for the normalisation pipeline rather than as a query layer.

This design preserves the raw source information while keeping the canonical relational model clean and consistent.

5.5 ADR-004: Adopt an Entity-Centric Data Model Focused on Boats

Status: Accepted **Date:** 2026-03-06

5.5.1 Context

The system collects information primarily from regatta result pages. These pages contain a mixture of data, including:

- boat names
- sail numbers
- owners
- club affiliations
- boat classes
- race results

However, the primary goal of the project has never been to build a database of race results. Instead, the objective is to construct a **structured dataset about boats and their associated entities**.

Regatta result pages are valuable because they often include detailed information about the participating boats, but they are treated primarily as **data sources** rather than the central subject of the system.

During early development it became clear that:

- The same boats appear repeatedly across multiple regattas.
- The most valuable information concerns the boats themselves and their relationships (owners, clubs, classes).
- Race positions provide little long-term value compared to the structural information about boats.

This led to the explicit decision to structure the system around **entities rather than events**.

5.5.2 Decision

The platform adopts an **entity-centric data model** in which the primary focus is on boats and their relationships.

The canonical dataset therefore centres around entities such as:

- boats
- owners
- clubs
- boat classes
- locations

Regattas and regatta editions are retained in the data model, but primarily as **contextual metadata and discovery signals** for identifying boats and extracting information about them.

Race results themselves are not treated as the primary analytical dataset.

Boat identity is resolved during the normalisation process using a combination of:

- sail number (boat identifier)
- boat name similarity

Two records are considered to represent the same boat only when both conditions are satisfied:

- the sail number matches after normalisation
- the boat name is sufficiently similar

This approach allows the system to merge appearances of the same boat across different regattas while avoiding incorrect merges when identifiers are ambiguous.

5.5.3 Consequences

Adopting an entity-centric model provides several advantages:

- The dataset focuses on durable entities rather than ephemeral race results.
- Boats appearing across multiple regattas can be unified into a single canonical record.
- Relationships between boats, clubs, and owners become easier to explore and analyse.
- The system produces a reusable knowledge base rather than a simple archive of race outcomes.

It also influences the structure of the canonical database:

- Many-to-many relationships are used for owners and clubs.
- Regattas act as contextual sources rather than the central dataset.
- Normalisation and identity resolution become critical components of the pipeline.

The main trade-off is that the system does not aim to preserve full analytical detail about race outcomes. Instead, it prioritises building a **clean and reliable registry of boats and related entities**.

5.6 ADR-005: Pipeline Execution via CLI

Status: Accepted
Date: 2026-03-30

5.6.1 Context

Initially, data pipelines were executed through a single Python entrypoint (`main.py`), which triggered all ingestion and transformation workflows sequentially.

This approach was sufficient during early development, when the number of pipelines was limited and execution requirements were simple.

However, as the system evolved, several limitations became evident:

- Lack of flexibility when running individual pipelines
- Difficulty isolating and debugging failures
- Inefficient execution when only a subset of pipelines needed to run
- Reduced control over pipeline orchestration

The growing number of domain-specific pipelines (boats, classes, clubs, regattas, schedule, scraping) required a more modular and controllable execution model.

5.6.2 Decision

The system adopts a **CLI-based pipeline execution model**, allowing each pipeline to be executed independently.

A command-line interface is introduced to:

- run individual pipelines
- run grouped pipelines when required
- provide a consistent execution interface

Each pipeline exposes a dedicated execution function, and the CLI acts as the orchestration layer.

5.6.3 Consequences

This decision provides several benefits:

- Modular execution of pipelines
- Improved debugging and error isolation
- Greater flexibility in development workflows
- Better alignment with future scheduling and orchestration systems

Trade-offs include:

- Increased complexity compared to a single entrypoint
- Need to maintain a consistent interface across pipelines

Overall, the CLI-based approach improves scalability and maintainability of the pipeline system as the project grows.

5.7 ADR-006: Centralised Configuration Management

Status: Accepted
Date: 2026-03-30

5.7.1 Context

Early versions of the system relied on hardcoded paths and configuration values embedded directly within the codebase.

Examples included:

- file paths for data storage
- logging directories
- configuration values for execution

As the project evolved, this approach introduced several limitations:

- Reduced portability across environments
- Difficulty adapting the system for Docker or cloud deployment
- Increased risk of inconsistencies between modules
- Harder maintenance when configuration changes were required

A more flexible and environment-agnostic configuration system was required.

5.7.2 Decision

The system adopts a **centralised configuration approach**, combining:

- environment variables (`.env`)
- a dedicated configuration module (`config.py`)

This configuration layer provides:

- centralised path management (data, logs)
- environment-based configuration
- separation between configuration and application logic

5.7.3 Consequences

Benefits include:

- Improved portability across environments (local, Docker, future cloud)
- Easier configuration changes without modifying code
- Reduced duplication of configuration logic
- Better alignment with containerised deployment

Trade-offs include:

- Dependency on environment variables
- Requirement for consistent usage across the codebase

This decision establishes a flexible foundation for deployment and future system evolution.

5.8 ADR-007: Structured Logging System

Status: Accepted
Date: 2026-03-30

5.8.1 Context

Initial pipeline and scraping processes relied on print statements to track execution progress and identify issues.

While sufficient for early experimentation, this approach presented several limitations:

- Lack of structured output
- Difficulty tracking execution across multiple pipelines
- No persistent record of execution history
- Limited visibility into warnings and errors

As the system grew in complexity, a more robust observability mechanism became necessary.

5.8.2 Decision

The system adopts a **structured logging approach** using Python's logging module.

Logging is integrated across pipelines and scraping modules, providing:

- multiple log levels (info, warning, error)
- consistent log formatting
- persistent log storage in files

Logs are stored in a dedicated directory and are used for debugging, monitoring and traceability.

5.8.3 Consequences

Benefits include:

- Improved observability of pipeline execution
- Easier debugging and error tracing
- Persistent execution history
- Better support for production environments

Trade-offs include:

- Increased complexity compared to simple print statements
- Need for consistent logging practices across modules

This decision enhances the reliability and maintainability of the system as it evolves.

6. Project

6.1 Project Overview

6.1.1 Purpose

The Regatta Data Platform is a structured data system designed to collect, normalise and expose regatta race data from heterogeneous sources.

Regatta results are often published across multiple websites with inconsistent formats and varying data quality. The goal of this project is to ingest these sources and transform them into a **clean canonical dataset** that can act as a **source of truth for regatta results**.

This dataset is intended to support analysis, discovery, and future applications built on top of reliable structured data.

The platform also exposes this data through a structured API, enabling exploration and integration with external applications.

6.1.2 Current Phase

The project is currently in an **early production-ready backend phase**, transitioning from prototype to application development.

The primary focus is on building a solid architectural foundation rather than releasing a finished product.

Current priorities include:

- maintaining and improving ingestion pipelines
- refining the canonical data model
- expanding API capabilities
- ensuring data quality and consistency
- preparing the system for frontend integration
- experimenting with AI-assisted development workflows

This stage prioritises **structural clarity and scalability** over feature completeness.

6.1.3 Current Capabilities

The platform currently provides:

- automated data ingestion from multiple web and PDF sources
- ETL pipelines for data transformation and normalisation
- a canonical relational database (PostgreSQL)
- a fully implemented FastAPI backend
- structured logging for pipeline execution and monitoring
- an API for navigating regattas, boats and related entities

6.1.4 Stakeholders

David Strategic oversight and product perspective. Provides governance guidance and reviews key architectural and project decisions.

Raul System architect and technical lead. Responsible for architecture design, data model definition, infrastructure decisions and AI-assisted development workflows.

Elena Supports data normalisation, validation and research of regatta data sources.

6.1.5 Core Technology

Current stack:

- Python (data processing and backend)
- PostgreSQL (canonical database)
- FastAPI (API layer)
- Playwright (web scraping)
- Docker (deployment and portability)

Development follows a layered model:

Data Sources → Ingestion → ETL Pipelines → Canonical Database → API → Frontend (in progress)

6.1.6 Governance Model

The project uses a clear separation of responsibilities between systems:

- **GitHub** → codebase and technical implementation
- **Zensical** → governance, architecture documentation and decisions
- **Email / lightweight coordination** → operational updates and communication

This structure maintains a clear traceable history of architectural decisions while keeping development workflows lightweight.

6.2 Project Roadmap

This roadmap outlines the expected evolution of the Regatta Data Platform.

It is intended as a **strategic direction rather than a fixed commitment**, and phases may evolve as the project progresses.

The goal of the roadmap is to ensure the system evolves in a **structured and scalable way**, prioritising architectural stability before expanding functionality.

6.3 Phase 0 – Architecture Stabilisation (Completed)

Objective

Transform CSV-based workflows into a structured, database-driven system with operational pipelines and API access.

This phase focuses on establishing a **robust technical foundation** before expanding the platform.

Environment

- Private GitHub repository
- Local development environment
- Docker used locally
- No external infrastructure cost

Key Workstreams

Database Foundation

- Design PostgreSQL schema
- Define raw ingestion tables
- Define canonical relational tables
- Implement ingestion timestamping
- Establish traceability between raw and canonical data

Pipeline Architecture

- Refactor ingestion scripts to write directly to the database
- Create structured ingestion pipelines
- Introduce batch identifiers for traceability

API Layer (Initial)

- Establish FastAPI project structure
- Implement initial read-only endpoints
- Support basic filtering (year, confidence, source)

Data Normalisation

This workstream runs in parallel with the architecture build.

Activities include:

- defining normalisation rules
- mapping raw values to canonical entities
- assigning confidence levels
- documenting edge cases
- researching missing metadata

Completion Criteria

- 2025 dataset stored in PostgreSQL
- Raw → canonical traceability operational
- API capable of querying structured data

Achieved Outcomes

- Fully operational ETL pipelines
- Canonical PostgreSQL database in use
- FastAPI backend implemented
- Structured logging system in place
- CLI-based pipeline execution

6.4 Phase 1 – Private Pilot (Current phase)

Objective

Create a functional internal prototype usable by Raul and David.

Environment

- Low-cost VPS (e.g. Hetzner)
- Docker deployment
- Private access only

Key Goals

Infrastructure

- Deploy Docker stack to VPS
- Configure SSH access and firewall
- Implement automated database backups

Dataset Completion

- Complete ingestion of the 2025 dataset
- Perform data quality review
- Operationalise confidence scoring

Semi-Automated Discovery

- Maintain regatta calendar
- Introduce URL-based ingestion workflow
- Allow predefined ingestion workflows to be executed by junior contributors

API & Product Layer

- Improve endpoint design and performance
- Introduce pagination and filtering
- Enhance API documentation

Frontend Development

- Build initial React-based interface
- Enable data exploration (regattas, boats)
- Connect frontend to API endpoints

Additional Focus Areas

- integration of frontend application
- improving API usability for exploration
- refining data quality and consistency

Completion Criteria

- 2025 dataset trusted
- 2026 ingestion requires minimal manual effort
- Junior ingestion workflow stable
- System usable without developer intervention

6.5 Phase 2 – Controlled Release

Objective

Prepare the system for broader controlled access.

Potential Infrastructure Evolution

- Evaluate migration to managed cloud infrastructure (e.g. AWS)
- Introduce monitoring and operational tooling

Key Areas of Work

Security & Access Control

- User authentication
- Role-based access control
- Audit logging

Operational Stability

- Error monitoring
- Performance optimisation
- Backup and recovery validation

6.6 Phase 3 – Scalable Production

This phase is not an immediate focus but represents the potential long-term evolution of the platform.

Possible areas of expansion include:

- multi-user access
- service-level reliability
- legal and compliance considerations
- expanded dataset coverage

6.7 AI-Assisted Development (Cross-Phase Initiative)

AI tools are integrated throughout the project as a **development capability layer**.

Current and future exploration areas include:

- AI-assisted code generation
- architecture-aware refactoring
- automated test scaffolding
- anomaly detection in datasets
- rule suggestion for normalisation workflows
- conversational interaction with the dataset

AI is treated as:

- a productivity multiplier
- a structured development tool
- a capability to be explored incrementally

All AI-generated outputs remain **human-reviewed and architecturally supervised**.

6.8 Ongoing Maintenance (From Pilot Phase)

Once the pilot phase is active, recurring operational tasks will include:

- monitoring ingestion failures
- reviewing low-confidence mappings
- adjusting schema when required
- dependency updates
- verifying backups
- reviewing system performance as the dataset grows
- monitoring API performance and usage
- maintaining frontend-backend integration