CSC 431

AGN-DB

System Architecture Specification (SAS)

Ethan Lichtblau	Student
Zain Khalid	Student
Sebastian Gonzales-Portillo	Student
Carlos Rodriguez	Student

Version History

Version	Date	Author(s)	Change Comments
1.0	04/10/25	Zain, Ethan, Carlos, Sebastian	Started on System Analysis: 1.1, 1.2, 1.3
1.5	04/11/25	Zain, Ethan, Carlos, Sebastian	finished system analysis: 1.4
2.0	4/13	Zain, Ethan, Carlos, Sebastian	finished Function design: 2.0, Structural design: 3.0
3.0	4/17	Zain, Ethan, Carlos, Sebastian	reviewed SAS document: improved 1.4, 2.1, 2.2, 3.1, 3.2, 3.3,

Table of Contents

Table of Figures

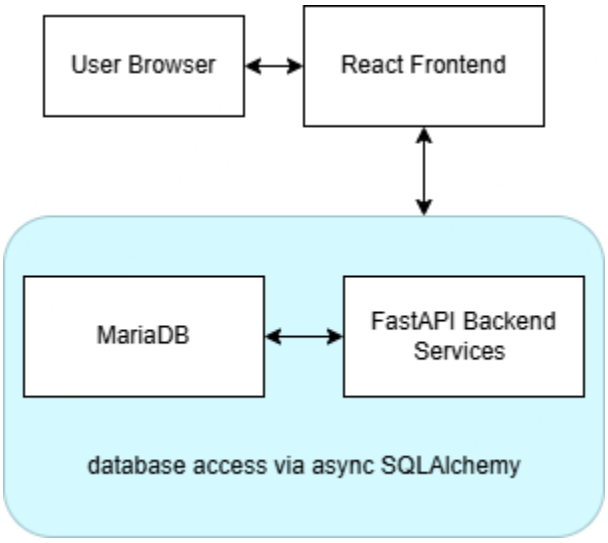
1. System Analysis

1.1 System Overview

AGN-DB is a sophisticated web application designed for astrophysics research, specifically focused on Active Galactic Nuclei (AGN) data management and analysis. The system provides a comprehensive platform for exploring, analyzing, and exporting astronomical data.

The application follows a modern microservices architecture with three main components: a React-based frontend single-page application, a FastAPI-based REST API backend, and a MariaDB database optimized for astronomical data storage. This architecture allows for scalability, maintainability, and separation of concerns, making it ideal for scientific research applications that require both performance and flexibility.

1.2 System Diagram



In this system, a user's browser loads a single-page application built with React. The React front-end sends HTTP/JSON requests to the FastAPI back-end services, which host all business logic. FastAPI uses async SQLAlchemy to query and update the MariaDB database, ensuring non-blocking, high-throughput data access. The shaded region denotes the server tier where FastAPI and MariaDB run together behind the scenes, while the unshaded boxes represent components executing in the user's browser.

1.3 Actor Identification

The following actors interact with the AGN-DB system:

- **Astrophysics Researchers:** Primary users who search for, analyze, and export AGN data for research purposes
- **Data Administrators:** Manage the system's dataset, handling data imports and quality control
- **System Administrators:** Maintain the technical infrastructure, handle deployments, and monitor system health
- **Automated System Services:** Scheduled tasks, data validation services, and external API integrations that interact with the system programmatically

1.4 Design Rationale

1.4.1 Architectural Style

- The AGN-DB application implements Clean Architecture, a layered architectural style that organizes the system into four concentric layers with dependencies pointing inward:
    - Entities (**Domain Layer**): Core business models and rules representing astronomical concepts, independent of application concerns.
    - Use Cases (**Application Layer**): Application-specific business logic that orchestrates domain entities to fulfill specific scientific operations.
    - Controllers/Presenters **(Interface Adapters Layer)**: Translates between use cases and external frameworks, containing controllers, presenters, repositories, and data schemas.
    - Frameworks & Drivers **(Infrastructure Layer)**: External technical infrastructure including FastAPI, SQLAlchemy ORM, MariaDB, and logging components.
- This architecture enforces a strict dependency rule where inner layers remain unaware of outer layers, and all dependencies point inward
- Clean Architecture was selected to:
    - Isolate scientific domain logic from technical implementations
    - Enable independent evolution of interfaces and infrastructure
    - Support complex astronomical data operations with clear boundaries
    - Ensure maintainability and testability as the system grows

1.4.2 Design Patterns

- **Repository Pattern**: Abstracts data access logic to create a collection-like interface for domain objects, decoupling business logic from storage implementation.
- **Dependency Injection**: Applied via FastAPI's dependency system to promote loose coupling and enhance testability by providing dependencies at runtime.
- **CQRS-lite**: Separates read (query) and write (command) operations into distinct paths, allowing for independent optimization of each operation type.
- **Adapter Pattern**: Used in the schema layer to transform between external API representations and internal domain models.

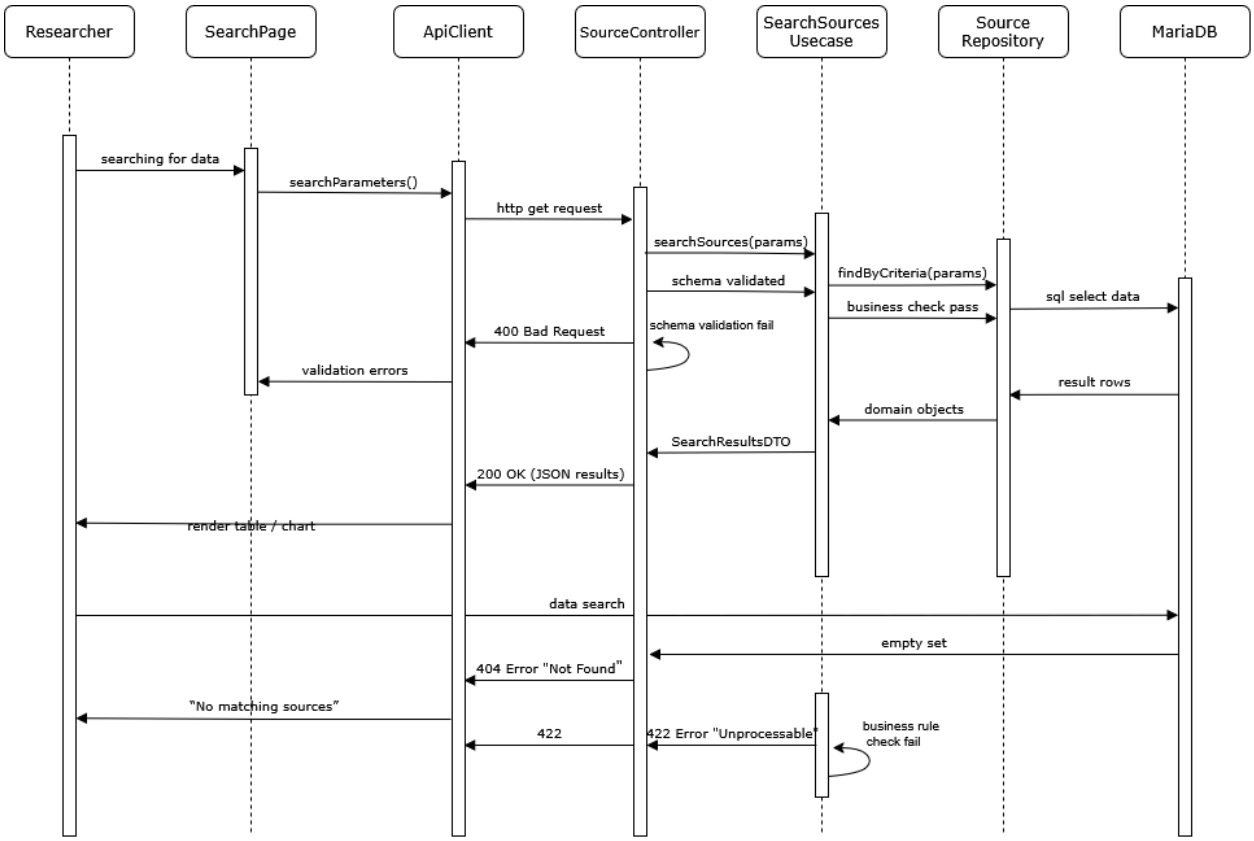
1.4.3 Framework

- **FastAPI**: A modern, high-performance web framework for building APIs with Python based on standard Python type hints. FastAPI was selected for:
    - Native async/await support for handling asynchronous operations efficiently
    - Automatic OpenAPI documentation generation for API endpoints
    - Built-in request validation and serialization using Pydantic
    - High performance compared to other Python web frameworks
    - Strong typing support that enhances developer experience and code reliability
- **SQLAlchemy** (with async support): An SQL toolkit and Object-Relational Mapping (ORM) library chosen for:
    - Flexibility to work with raw SQL when needed for complex queries
    - Support for async database operations through asyncmy driver
    - Comprehensive ORM capabilities for working with domain models
    - Database-agnostic design allowing for potential database migrations
    - Mature ecosystem with extensive documentation and community support
- **MariaDB**: A community-developed fork of MySQL used as the primary database system. MariaDB was selected for:
    - High performance and scalability to handle over 7 million astrophysical records
    - Legacy system, not within scope to change
- **React**: A JavaScript library for building user interfaces, selected for the frontend because of:
    - Component-based architecture that promotes reusability and maintainability
    - Industry standard, strong community support and extensive documentation
    - Compatibility with TypeScript for enhanced type safety in the frontend
- **Pydantic**: A data validation and settings management library using Python type annotations, selected for:
    - Seamless integration with FastAPI

    - Strong validation capabilities for API requests and responses
    - Clear error messages for invalid data
- **Loguru**: A library for Python logging chosen for:
    - Simplicity and powerful features for structured logging
    - Comprehensive exception capturing and contextualization

2. Functional Design

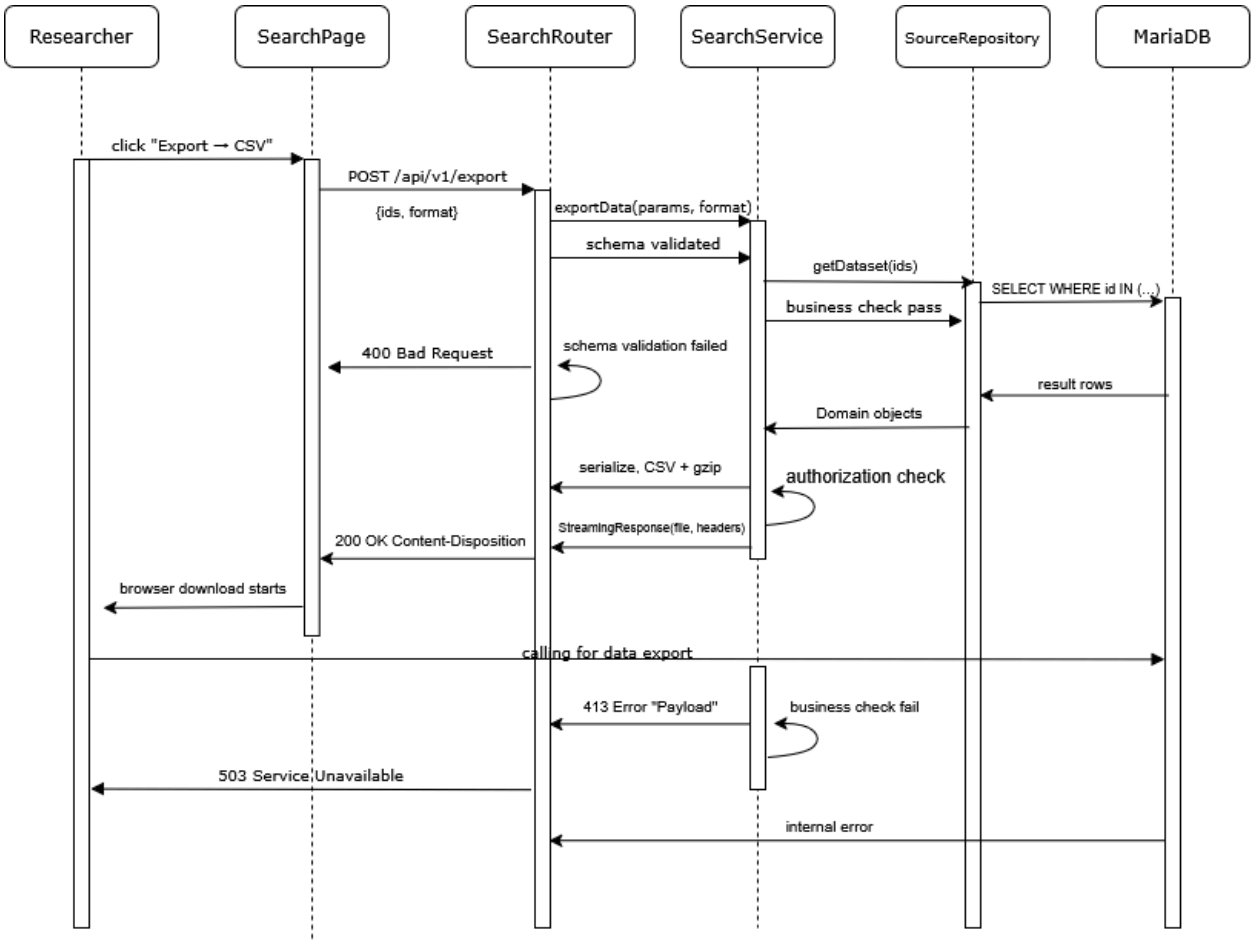
2.1 Sequence Diagram: Data Query Workflow



1. The researcher clicks Search (or presses Enter). This UI action is annotated as searching for data on the diagram.
2. The React page gathers the form fields into a parameter object and hands it to the browser-side API client
3. The ApiClient builds an HTTP request with the query parameters and sends it to the backend FastAPI route handled by SourceController
4. FastAPI and Pydantic perform schema / type validation on the incoming request body or query string.
    a. If validation fails, then SourceController immediately returns 400 Bad Requests.

    b. It then propagates back to ApiClient, then to SearchPage, which displays the validation error message.
5. When the schema is valid, the controller forwards the normalized parameters to the application-layer use-case.
6. Domain/business rules are verified (e.g., parameter ranges, user quotas).
    a. If a rule is violated it returns a domain error and SourceController converts it to 422 Unprocessable Entity.
    b. propagates to ApiClient then SearchPage, which shows the rule-error message
7. With all validations passed, the use-case delegates data access to the repository.
8. The repository composes a parametrized SQLAlchemy query that MariaDB executes.
9. The database returns either:
    a. A result set containing matching AGN records
    b. An empty set if nothing matches.
10. Rows are mapped to domain entities (or an empty list) and passed back to the use-case.
11. The use-case converts the domain data into a transport DTO ready for JSON serialization.
12. SourceController sends the data to the ApiClient with either:
    a. returns 200 OK with the DTO serialised as JSON.
    b. returns 404 Not Found (or 200 with an empty list, depending on API spec).
13. ApiClient sends the either:
    a. 200 OK: the client hands the JSON payload to the page, which renders a table or chart with the results.
    b. On 404: the page shows “No matching sources.”
14. The UI updates, delivering either the search results visualization or a helpful error/empty-state message to the researcher.

2.2 Sequence Diagram: Data Export Workflow

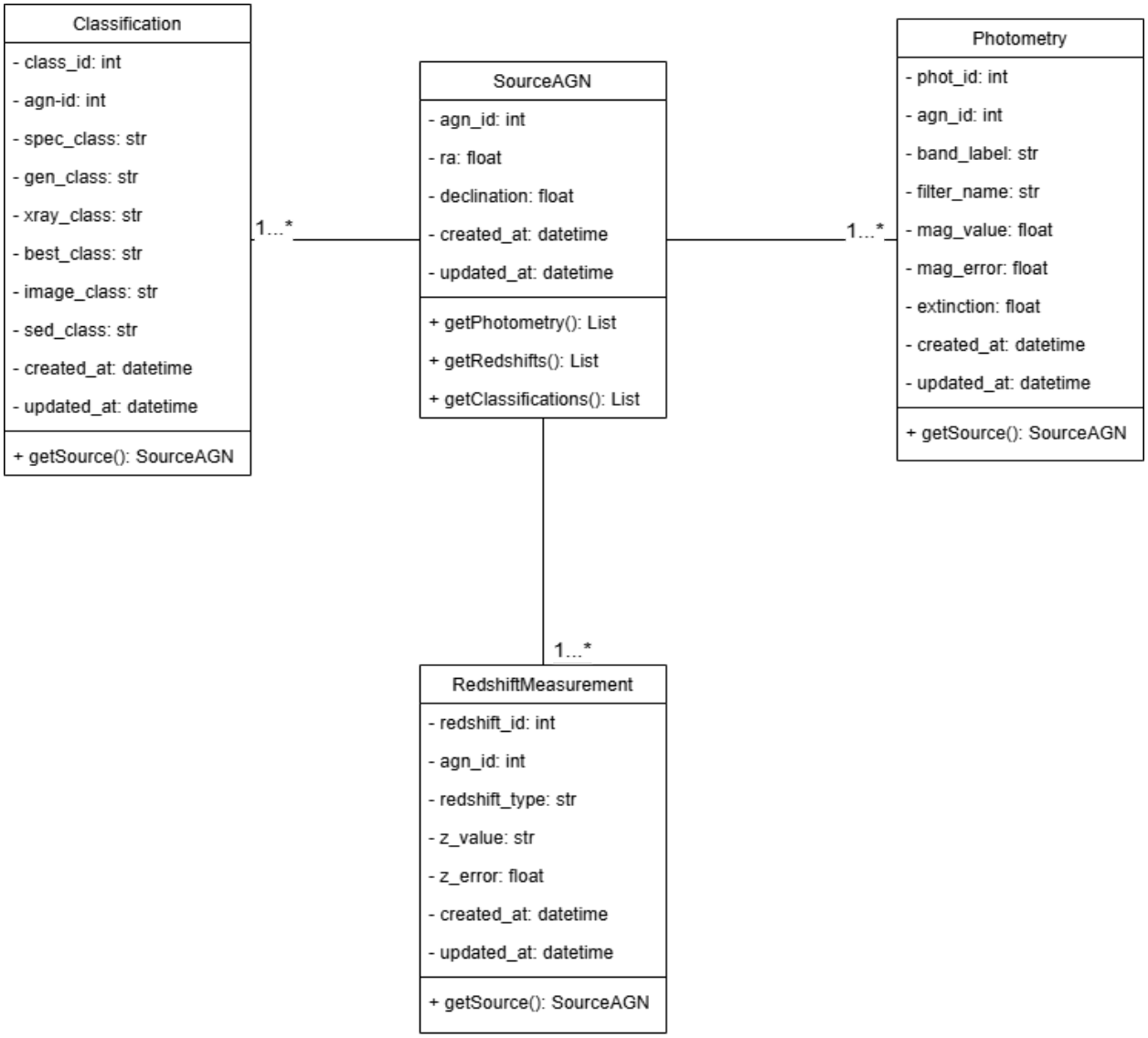


1. The researcher clicks Export to CSV on the results screen; this user-interface action is captured as clicking "Export to CSV" on the diagram.
2. The React SearchPage component bundles the selected record IDs and desired file format into a payload and posts it to the browser-side API client.
3. The API client issues a POST /api/v1/export request containing {ids, format} to the FastAPI route handled by SearchRouter.
4. FastAPI and Pydantic perform initial schema and authentication validation on the incoming request.
    a. If validation fails, SearchRouter immediately returns 400 Bad Request.
    b. The error propagates back through the API client to SearchPage, which displays the validation-error message to the researcher.
5. When the schema is valid, the router forwards the normalized parameters to the application-layer SearchService via the exportData(params, format) call.
6. SearchService enforces business-rule checks (for example, row-count limits or export-rate quotas).

    a. If a rule is violated, it signals a domain error that SearchRouter converts to 413 Payload Too Large (or another appropriate code).
    b. That response travels back to the browser, and the page informs the researcher that the export request exceeds system limits.
7. If all validations pass, SearchService delegates data retrieval to SourceRepository by calling getDataset(ids).
8. The repository builds a parameterised SQLAlchemy query, which MariaDB executes (SELECT … WHERE id IN (…)).
9. The database returns either (a) a set of matching AGN records or (b) an empty set if nothing matches.
10. SourceRepository maps the rows to domain entities (or an empty list) and hands them to SearchService.
11. SearchService performs an authorization check to ensure the current user may export each record, then serialises the dataset to CSV, optionally compresses it with gzip, and wraps the file in a streaming response object.
12. SearchService returns that StreamingResponse to SearchRouter, which sends it to the browser with 200 OK and a Content-Disposition: attachment header.
13. The browser download dialog opens, and SearchPage shows a progress indicator or toast indicating that the export has started.
14. If any unexpected infrastructure failure occurs during steps 7–12 (for example, a database timeout), SearchRouter converts the exception to 503 Service Unavailable, and the page displays an “error ” message.
15. The UI updates—either triggering the file download, displaying a quota or validation error, or showing a server-error notice—so the researcher receives immediate feedback on the export attempt.

3. Structural Design

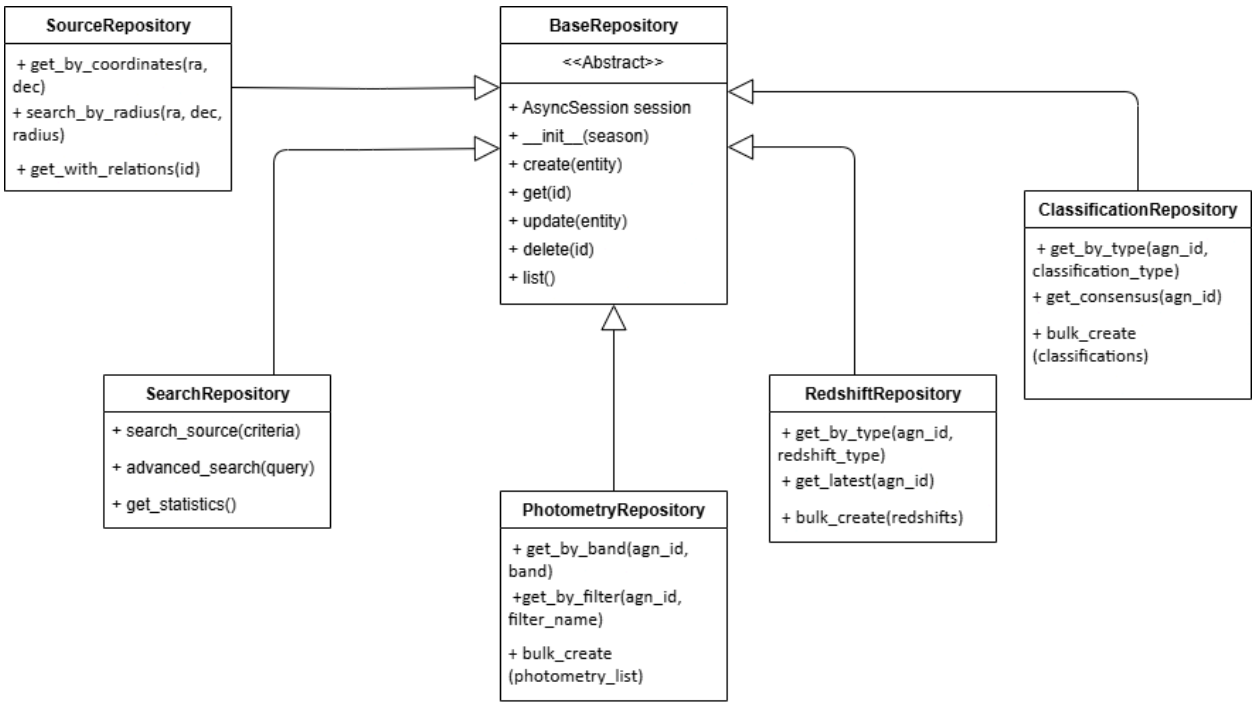
3.1 Core Entity Class Diagram



This class diagram represents the core entities of the system that will simulate the management of Active Galactc Nuclei (AGN) observations. The central entity, SourceAGN, represents an astronomical object and links to the associated Photometry, RedshiftMeasurement, and Classification records through one-to-many relationships. Each related entry captures specific scientific data- such as brightness, redshift, and classification types- along with timestamps to allow for better data tracking, helping with holding researchers accountable for changes, and

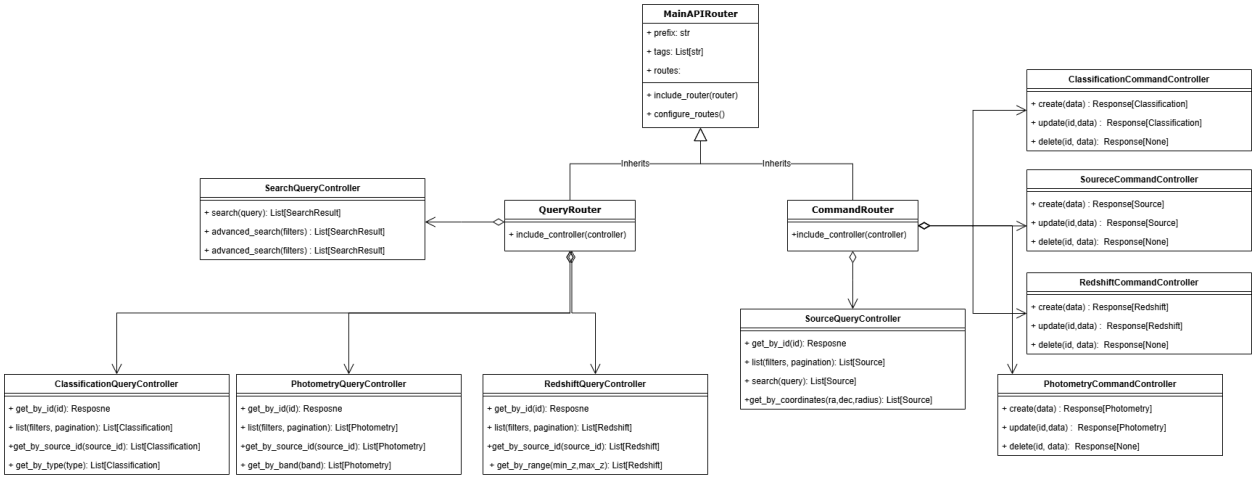
providing traceability. The design allows the code to be modular and extendable for future AGN research.

3.2 Repository Layer Class Diagram



This class diagram represents the Repository Layer of the application. It breaks down many of the operations into modular, reusable components. The BaseRepository defines common CRUD operations using an asynchronous SQLAlchemy session. All other repositories inherit methods from this class; specific repositories (e.g., SourceRepository, PhotometryRepository) extend BaseRepository by adding domain-specific query methods, such as searching by coordinates, filtering by band, retrieving consensus classifications, etc. The design of the Repository Layer promotes code reuse, allows for better maintainability in the data access layer, and allows for these parts to be scalable.

3.3 API Layer Class Diagram



This class diagram represents the API Layer of our system, which is responsible for routing and handling all HTTP interactions in the system. It follows a modular design using FastAPI routers to organize functionality into query and command operations. The MainAPIRouter serves as the root entry point and delegates responsibilities to specialized components such as QueryRouter and CommandRouter. Each router includes domain-specific controllers- e.g. SourceQueryController and PhotometryCommandController- which expose route handlers for reading and modifying domain data. Query controllers provide read and write operations such as filtered lists or retrieval by ID while command controllers encapsulate write operations like creating, updating, or deleting records. This design supports the system's CQRS-lite architecture, promoting modularity, allowing for components to be scalable, easier testing of API components, and separation of concerns.