

Cardano Analytics Platform

Project Execution Report

December 1st, 2025

Project Catalyst Fund13

Project ID: 1300034

Contributors

Marcio Moreno (mmoreno@mobr.ai)

Rafael Brandão (rafael@mobr.ai)

[Executive Summary](#)

[1. Introduction](#)

[2. Design and Development](#)

[2.1 The Ontology](#)

[2.2 Design](#)

[2.3 Development](#)

[3. Achievements](#)

[4. Implementation](#)

[4.1 Utility Modules](#)

[4.2 Telemetry and Configuration Modules](#)

[4.3 ETL Pipeline Modules](#)

[4.4 Database Modules](#)

[4.5 API Modules](#)

[4.6 RDF Modules](#)

[4.7 Core Modules](#)

[4.8 Services Modules](#)

[4.9 Test Modules](#)

[4.10 Use Cases](#)

[5. Final Remarks](#)

[6. Acknowledgments](#)

Executive Summary

This report details the successful completion of the CAP (Cardano Analytics Platform) project funded under Cardano Catalyst Fund 13, focused on developing an LLM-powered analytics prototype to ease access and analysis of Cardano on-chain data. Over the course of 11 months, the project achieved all outlined milestones, resulting in a fully functional, open-source prototype that proves the concept of a platform that enables natural language queries, insights, and dashboards. Key accomplishments include the creation of an ontology for the Cardano ecosystem, data extraction and ETL pipelines, an LLM query engine, and a user interface for data visualization. The platform has been publicly released on GitHub under the GPLv3 license, promoting transparency and community collaboration. Through testing and validation, the project has demonstrated its feasibility in easing blockchain analytics, reducing technical barriers, and allowing broader participation within the Cardano community.

1. Introduction

The Cardano Analytics Platform (CAP) is designed to simplify access to Cardano blockchain data through natural language queries powered by large language models (LLMs). On the one hand, LLMs offer powerful natural language understanding capabilities, including flexible natural language processing, contextual understanding, and the ability to generate human-like responses. However, they face challenges with factual consistency and domain-specific knowledge, often producing hallucinations or inconsistent outputs when dealing with specialized domain knowledge.

On the other hand, Knowledge Graphs (KGs) excel at representing structured domain knowledge and supporting precise querying. Their strengths include explicit relationship modeling, formal reasoning capabilities, and guaranteed factual consistency within their domain. However, KGs require specialized query languages and lack the flexibility to handle natural language inputs and outputs.

A Knowledge Graph-enhanced LLM combines these approaches to mitigate their respective limitations. By integrating LLM responses with a formal ontology and graph database, it is possible to ensure factual accuracy while maintaining natural language interaction capabilities. In the context of blockchain analytics, this integration enables CAP to leverage the LLM's ability to interpret user queries while ensuring that responses are based on accurate Cardano blockchain data and relationships defined in our ontology and KG.

The ontological foundation of the CAP system consists of two main components: the TBox (terminological box), which contains concept definitions and relationships from the Cardano network, and the ABox (assertional box), which contains actual blockchain data instances extracted from Cardano. This structure enables both formal reasoning about blockchain concepts and practical analytics over real-world data.

The project addresses the challenge of accessing and structuring on-chain data in the Cardano ecosystem by building an innovative analytics platform integrated with large language models (LLMs). This initiative aimed to support users of varying technical expertise to interact with blockchain data through natural language queries, thereby streamlining decision-making processes and enhancing overall ecosystem engagement. The development process followed a structured milestone-based approach, ensuring systematic progress from foundational research to prototype deployment. All specified outputs have been made fully open-source, allowing for ongoing community contributions and adaptations. This report outlines the work completed, the methodologies employed, and the tangible achievements realized throughout the project lifecycle.

2. Design and Development

2.1 The Ontology

CAP's ontology provides a conceptual framework for modeling the Cardano blockchain ecosystem. The ontology extends general blockchain concepts with Cardano-specific elements across several key domains. Figure 1 illustrates the ontology being curated with the use of the Protégé¹ tool.

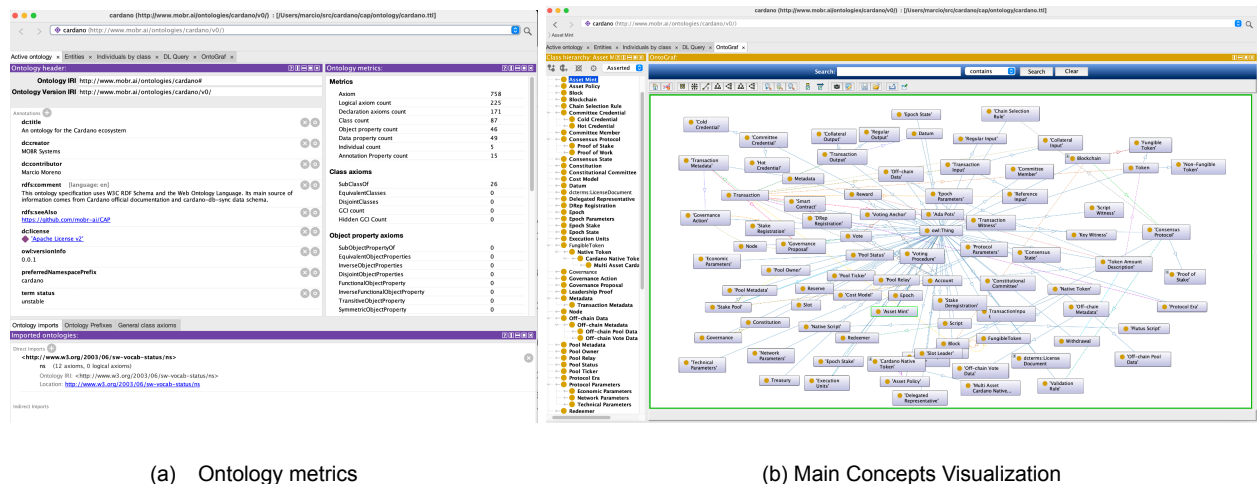


Figure 1. Using Protégé to curate CAP's Ontology

The ontology's foundation includes standard blockchain elements such as Blocks, Transactions, and Accounts, enhanced with Cardano's unique implementation details. For instance, the Transaction model incorporates specialized input and output types, including collateral and reference inputs that support Cardano's Extended Unspent Transaction Output (EUTXO) model.

The ontology models Cardano Native Tokens (CNTs) as an extension of the general Token concept, with properties for policy IDs and minting rules. This enables the representation of both fungible and non-fungible tokens, along with their associated metadata and policies.

The Ouroboros proof-of-stake protocol is modeled through classes representing Slots, Epochs, and Leadership Proofs. The ontology captures both the technical mechanics of block production and the economic incentives associated with stake pools and rewards.

A substantial portion of the ontology models Cardano's governance framework, including Delegated Representatives (DReps) and various types of governance actions, such as parameter changes and treasury withdrawals. The ontology also represents voting procedures, proposal lifecycles, and the relationships between different governance actors.

By providing coverage of the Cardano ecosystem, the ontology enables structured queries and advanced analysis, ensuring semantic clarity and formal reasoning capabilities.

¹ <https://protege.stanford.edu>

The complete ontology documentation is available on CAP's GitHub repository at <http://github.com/mobr-ai/cap> and online at <https://www.mobr.ai/cardano>.

2.2 Design

CAP was designed with a methodology and workflow that integrates ontological reasoning, knowledge graph querying, and natural language processing. A summary of the designed workflow is illustrated in Figure 2.

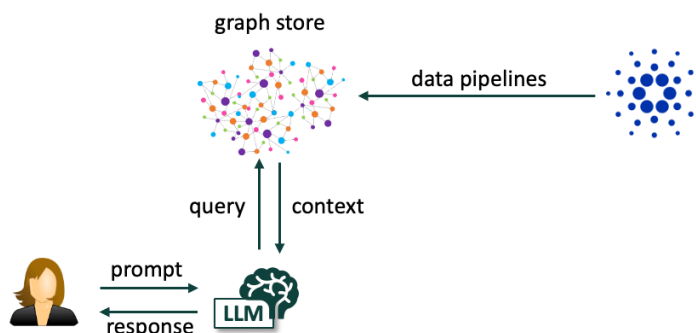


Figure 2. CAP's Workflow

Our methodology began with use case development, where we analyzed common blockchain analytics needs to identify representative queries and required data patterns. This analysis informed both the ontology design and the LLM training approach. CAP's ontology was developed iteratively to ensure it covers all necessary concepts for the identified use cases while maintaining alignment with established blockchain ontologies.

For knowledge graph population, CAP relies on data pipelines to extract and transform Cardano data into a structured knowledge graph while maintaining consistency with the ontology. The data is extracted mainly from a cardano-node instance and the PostgreSQL database managed by cardano-db-sync. The populated KG is the basis for supporting CAP's LLM model. Our roadmap foresees fine-tuning CAP's model on a curated dataset of Cardano network queries and responses, emphasizing accuracy in technical terminology and data interpretation.

The design includes both API endpoints and a user interface that supports natural language queries and visualization generation. The system workflow begins with query reception, as users input natural language queries (prompts) through the interface. This is followed by semantic processing, where the fine-tuned LLM analyzes the query to identify relevant blockchain concepts and relationships, generating formal queries aligned with CAP's ontology. Another LLM component processes query results, generating natural language explanations and visualization recommendations. Finally, the dashboard component controller generates appropriate visualizations based on data characteristics and query intent.

2.3 Development

Milestone 1 focused on ontology design and research. During this phase, a detailed ontology for the Cardano ecosystem was developed, encapsulating key entities such as blocks, transactions, accounts, stake pools, and governance actions. This ontology serves as the semantic foundation for the platform, drawing from existing glossaries and frameworks within the cardano domain. A technical report was produced, documenting the ontology's structure, relationships, and alignment with Cardano's data models. This report included diagrams and explanations, ensuring reproducibility and ease of understanding for future extensions.

Building on this foundation, Milestone 2 involved bootstrapping the knowledge base and initial data extraction. A functional knowledge graph was established using Virtuoso, a triplestore database, integrated with CAP's API for querying. Basic data extraction mechanisms were implemented to pull information from the Cardano network, converting raw blockchain data into RDF-triple representations. Demonstrations were conducted to showcase the knowledge graph's querying capabilities, verifying that extracted data could be accurately represented and retrieved in a structured format.

Milestone 3 centered on data extraction and ETL pipelines. Extraction components were refined to handle Cardano data, incorporating ontology alignment mechanisms to ensure semantic consistency. ETL pipelines were constructed for transforming and loading data, automating the process of ingesting blockchain information into the knowledge base. This phase included optimizations for handling high volumes of data, such as batch processing and error-handling protocols. Demonstrable extractions and transformations were performed, confirming the pipelines' efficiency in processing real-time and historical Cardano data.

Milestone 4 delivered an LLM query engine. This engine enables natural language querying by leveraging an LLM to translate user inputs into SPARQL queries against the knowledge graph. An API was created for processing these queries, supporting complex questions like identifying large ADA transactions or wallet distributions. Functional demonstrations illustrated the engine's accuracy, with utilities developed to convert SPARQL results into user-friendly key-value pairs. This integration marked a significant advancement in making cardano data accessible without requiring specialized query language knowledge. This milestone also encompassed UI and dashboard implementation. A user-friendly interface was designed for data visualization, incorporating tools for composing custom dashboards.

In the final milestone, the entire platform underwent testing, covering unit tests for extractors, loaders, and utilities, as well as integration tests for the natural language query pipeline. The produced prototype is available on GitHub with an open-source repository containing all code, documentation, and sample queries. This ensures that users can deploy and customize the platform independently, with built-in support for visualizations such as bar charts, pie charts, and line charts using Vega formatting.

Throughout the implementation, emphasis was placed on proving the feasibility of the project and that it is a viable solution for the stated problem. Utilities were developed to manage blockchain-specific data challenges, such as preserving large integers for token amounts and decoding hexadecimal strings for token names. Testing suites validated the platform's components, including ETL processes, SPARQL conversions, and visualization utilities, ensuring high reliability across various scenarios.

3. Achievements

The project successfully delivered a analytics prototype that integrates LLMs with Cardano's on-chain data, achieving seamless natural language interactions. Users can pose queries in everyday language, such as requesting the largest ADA transactions over a specified period or plotting trends in smart contract deployments, receiving intuitive responses with visualizations. The ontology and knowledge graph provide a semantic layer, enabling accurate data representation and retrieval. ETL pipelines handle data ingestion, supporting real-time updates and historical analysis. The query engine processes queries, incorporating custom utilities for data conversion, which enhance usability for non-technical users.

A notable achievement is the platform's open-source nature, with all code publicly available on GitHub under GPLv3. This includes detailed utilities for SPARQL result processing, Vega chart conversions, and testing frameworks that cover health checks, query validations, and data integrity. The platform has been tested against sample Cardano data, verifying its ability to handle metrics like transaction volumes, stake distributions, and governance activities. These features collectively reduce the technical barriers to blockchain analytics, making Cardano data more approachable and actionable.

The delivered prototype has as impact the capability of broadening access to on-chain data, by enabling developers, researchers, and enthusiasts to gain insights without deep technical knowledge. Analytics with customizable dashboards facilitate informed decision-making, potentially increasing transparency and engagement in ecosystem activities. By streamlining data analysis, the project contributes to greater community involvement and innovation within Cardano.

4. Implementation

The platform's codebase is organized into a modular structure under the "cap" package, encompassing utilities, ETL processes, database management, API endpoints, RDF handling, core functionalities, services, and testing. This architecture ensures maintainability, scalability, and separation of concerns, allowing for efficient development and integration of LLM-powered analytics with Cardano on-chain data. Below, each major module category is discussed, highlighting its purpose, key components, and contributions to the overall platform.

4.1 Utility Modules

The utility modules provide foundational tools for data processing and visualization, residing primarily in the "cap/util" directory. The "sparql_util.py" module is central to handling SPARQL query results from the triplestore, converting them into simplified key-value pairs optimized for LLM consumption. Complementing this, "vega_util.py" facilitates the transformation of query results into Vega-compatible formats for visualizations, supporting bar charts, pie charts, line charts, and tables. It intelligently identifies x-axis candidates (e.g., timestamps or categories) and series keys for multi-dimensional data, enabling dynamic chart generation based on user queries. Additionally, "epoch_util.py" offers helper functions for converting Cardano epoch numbers to dates, aiding in time-based analytics. These utilities collectively bridge the gap between raw data queries and intuitive outputs, enhancing the platform's accessibility for natural language interactions.

4.2 Telemetry and Configuration Modules

The "telemetry.py" module collects and logs performance metrics, such as query execution times and error rates, integrating with external monitoring tools for real-time insights. Paired with "config.py", which centralizes environment variables, database connections, and API keys, these modules ensure the platform's configurability and observability. They handle sensitive settings securely, supporting deployment in varied environments while facilitating debugging and optimization.

4.3 ETL Pipeline Modules

The ETL (Extract, Transform, Load) modules, under "cap/etl/cdb", form the backbone of data ingestion from Cardano sources into the knowledge base. Extractors ("extractors/" directory) include specialized classes for entities like accounts ("account.py"), epochs ("epoch.py"), blocks ("block.py"), transactions ("transaction.py"), multi-assets ("multi_asset.py"), scripts ("script.py"), datums ("datum.py"), stakes ("stake.py"), and governance actions ("governance.py"). Each extractor queries the PostgreSQL database using SQLAlchemy, batching data to manage large volumes efficiently. The "extractor_factory.py" dynamically creates these extractors based on entity types, promoting code reuse. Transformers ("transformers/" directory) mirror this structure, converting extracted relational data into RDF triples aligned with the Cardano ontology; for instance, "transaction.py" maps transaction fields to RDF predicates like "hasTx" or "hasOutput". The "transformer_factory.py" orchestrates this process. Finally, the

"loaders/loader.py" handles batch loading into the Virtuoso triplestore, with chunking for large datasets, progress metadata saving, and data integrity validation. The "service.py" oversees the entire ETL workflow, tracking progress with "ETLProgress" and "ETLStatus" enums. These modules enable robust, scalable data pipelines, ensuring the knowledge base remains up-to-date with on-chain information.

4.4 Database Modules

In "cap/database", the modules manage the relational database schema and migrations. "model.py" defines SQLAlchemy models for entities like users, dashboards, and metrics, incorporating fields for authentication and data visualization. "session.py" provides session management for database connections. The "alembic-migration/migrations/" subdirectory contains versioned migration scripts, such as "20251030_add_avatar_blob_add_avatar_fields_to_user.py" for adding user avatar support and "20251127_add_metrics_tables.py" for metrics tracking. The "env.py" configures Alembic for migration execution. This module ensures schema evolution, supporting features like user dashboards and performance logging.

4.5 API Modules

The API modules, situated in the "cap/api" directory, form the primary interface for client interactions with the platform, implementing RESTful endpoints using the FastAPI framework to deliver secure, high-performance access to core functionalities. These modules leverage FastAPI's asynchronous capabilities, dependency injection for handling authentication and database sessions, and automatic OpenAPI documentation generation via Swagger UI, which facilitates developer integration and testing. Security is paramount, with built-in support for JWT-based authentication, rate limiting to mitigate abuse, and CORS configurations for safe cross-origin requests. The API design emphasizes modularity, with each file focusing on a specific domain such as querying, user management, or administration, while shared utilities like logging and error handling ensure consistency across endpoints. Pydantic models are extensively used for request and response validation, preventing invalid data from propagating through the system and enhancing reliability for natural language analytics and data visualization tasks.

The "nl_query.py" module is dedicated to processing natural language queries, featuring a primary POST endpoint that accepts user inputs along with optional parameters like temperature for controlling LLM creativity or max tokens for response length. It orchestrates the query pipeline by invoking the "nl_service" to preprocess inputs, generate SPARQL via the Ollama client, execute queries against the triplestore, and apply post-processing for formats such as key-value pairs or Vega visualizations. Responses are streamed as server-sent events (SSE) to provide progressive updates, improving user experience for complex analyses. Error handling includes graceful fallbacks for ambiguous queries, with logging of query metadata for metrics tracking, making this module central to the platform's user-friendly analytics interface.

"sparql_query.py" enables direct interaction with the knowledge base through endpoints for executing raw SPARQL queries, primarily via POST requests that include the query string and optional graph identifiers. It performs syntax validation to catch errors early, routes execution to the triplestore client, and utilizes utilities from "sparql_util.py" to convert results into simplified structures, handling blockchain-specific conversions like lovelace to ADA. This module supports both SELECT and ASK query types, with optional caching integration via "redis_sparql_client.py" to accelerate repeated executions. It is particularly useful for advanced users or developers needing precise control over data retrieval, and includes query logging to monitor usage patterns.

The "dashboard.py" module manages customizable dashboards, offering CRUD (Create, Read, Update, Delete) endpoints for dashboard resources. Users can POST configurations to create new dashboards, specifying widgets linked to natural language or SPARQL queries, with support for visualizations via Vega utilities. GET endpoints retrieve dashboard states, while PATCH allows modifications like adding/removing widgets or updating layouts. Integration with the database stores persistent dashboard data, and sharing features enable collaborative access through unique URLs or permissions. This module enhances the platform's interactivity, allowing non-technical users to build intuitive interfaces for ongoing Cardano data monitoring.

"user.py" handles user profile management, with endpoints for user registration (POST), profile retrieval and updates (GET/PATCH), and deletion. It interfaces with database models to store and retrieve user information, including fields for avatars, preferences, and linked accounts. OAuth integrations, such as Google sign-in, are supported through callback endpoints that exchange codes for tokens. Validation ensures compliance with data privacy, and the module includes features for password reset and email verification, contributing to a seamless user onboarding experience.

"auth.py" specializes in authentication and authorization, implementing endpoints for login (POST with credentials returning JWT tokens), token refresh, and logout (invalidating sessions). It utilizes libraries like passlib for password hashing and PyJWT for token management, with dependencies injected for database access. Support for external providers like Google OAuth is included, with routes for authorization redirects and callbacks. Role-based access control (RBAC) is enforced, restricting administrative endpoints, ensuring that sensitive operations like ETL triggers remain protected.

Administrative functionalities are covered in several specialized modules. "etl_admin.py" offers endpoints for managing ETL processes, including POST to initiate jobs, GET for status monitoring, and DELETE for cancellation, integrating with the ETL service to track progress and handle errors. "cache_admin.py" provides tools for cache management, with endpoints to invalidate specific keys or flush entire caches via the Redis clients, useful for maintaining data freshness after updates. "demo_nlp.py" includes demonstration endpoints that showcase natural language queries with predefined examples, serving as an educational tool for new users or during onboarding. "waitlist.py" manages user waitlists for beta access or features, with POST for sign-ups and GET for status checks, including email notifications via the mailing service.

Complementing these, "models.py" defines Pydantic schemas for all API inputs and outputs, such as QueryRequest for natural language payloads or DashboardConfig for visualization setups. These models enforce type safety, automatic serialization, and validation rules (e.g., string lengths or numeric ranges), reducing runtime errors and improving API documentation. Overall, the API modules provide a secure gateway to the platform's capabilities, enabling the democratization of Cardano analytics as envisioned in the project.

4.6 RDF Modules

Under "cap/rdf", these modules interact with the semantic layer. "cdb_model.py" defines RDF classes and properties for Cardano entities, forming the ontology. "triplestore.py" encapsulates Virtuoso client operations, including graph creation, data insertion, and query execution. The "cache/" subdirectory enhances performance: "query_normalizer.py" and "sparql_normalizer.py" standardize queries; "placeholder_counters.py", "placeholder_restorer.py", "value_extractor.py", "pattern_registry.py", "semantic_matcher.py", and "query_file_parser.py" implement caching logic for repeated queries. This module enables semantic querying, crucial for LLM integration and data alignment.

4.7 Core Modules

The "cap/core" modules handle foundational logic. "google_oauth.py" manages OAuth flows; "auth_dependencies.py" and "security.py" enforce authentication and JWT handling; "main.py" bootstraps the FastAPI application, integrating all components. These ensure secure, cohesive operation across the platform.

4.8 Services Modules

The services modules, located in the cap/services directory, encapsulate high-level backend functionalities that integrate various components of the platform, ensuring seamless operation of natural language processing, caching, metrics tracking, and language detection. These modules act as intermediaries between the core application logic and external or specialized services, promoting modularity and ease of maintenance. Each module is designed with asynchronous capabilities where appropriate, leveraging libraries like asyncio for efficient handling of I/O-bound operations, and they collectively support the platform's goal of providing responsive, intelligent analytics through LLM integration.

The ollama_client.py module serves as the interface to the Ollama LLM service, facilitating the generation of SPARQL queries from natural language inputs. It includes classes and functions for establishing connections to Ollama servers, managing model selections (such as Llama or other open-source LLMs), and handling prompt engineering to optimize query translation accuracy. Key features include retry mechanisms for failed requests, token limit management to prevent overflow, and response parsing to extract structured query outputs. This module is crucial for the platform's core innovation, enabling users to interact with Cardano data in plain English by converting queries like "Show the top ADA holders" into executable SPARQL statements, while logging interactions for debugging and performance tuning.

The `nl_service.py` module orchestrates the natural language processing pipeline, coordinating between user inputs, LLM query generation, SPARQL execution, and result formatting. It defines a service class that encapsulates the end-to-end workflow: receiving queries via API, preprocessing them for context (e.g., entity recognition), invoking the Ollama client for translation, executing queries against the triplestore, and post-processing results with utilities like those in `sparql_util.py` for key-value conversion or `vega_util.py` for visualizations. Error handling is robust, with fallback mechanisms for ambiguous queries and integration with caching clients to reduce redundant computations. This module enhances user experience by ensuring low-latency responses and supporting advanced features like multi-turn conversations or query refinements.

Caching is managed through `redis_nl_client.py` and `redis_sparql_client.py`, which provide Redis-based caching layers for natural language queries and SPARQL results, respectively. The `redis_nl_client.py` caches LLM-generated SPARQL queries by hashing natural language inputs, allowing quick retrieval for repeated or similar questions and reducing LLM invocation overhead. It includes expiration policies, invalidation logic tied to data updates in the knowledge base, and serialization for complex query objects. Similarly, `redis_sparql_client.py` caches raw SPARQL execution results, normalizing queries to handle variations in formatting or parameters, and stores them with metadata like timestamps for freshness checks. These modules significantly improve platform performance, especially under high query loads, by minimizing database hits and LLM computations, while supporting distributed deployments through Redis clustering.

The `lang_detect_client.py` module handles language detection for incoming queries, integrating with libraries like `langdetect` or `fasttext` to identify the query's language and, if necessary, translate it to English for LLM processing. It includes fallback to default English assumptions and logging for multilingual analytics. This ensures the platform's accessibility to a global audience, automatically routing non-English queries through translation services before feeding them into the NL pipeline, thereby broadening its utility beyond English-speaking users.

4.9 Test Modules

The test modules, housed in the `"tests/"` directory, constitute a suite designed to validate the prototype's reliability, functionality, and performance across all components, ensuring high-quality deliverables aligned with the project's milestones. Utilizing frameworks like `pytest` for unit and integration testing, along with asynchronous support via `pytest-asyncio`, these modules cover a wide spectrum from individual ETL extractors to end-to-end natural language query pipelines. The suite emphasizes automation, with fixtures for database sessions and a triplestore client to simulate real-world environments, and includes assertions for data integrity, error handling, and expected outputs. Coverage is extensive, incorporating mock data for Cardano entities, health checks for services, and scenario-based validations to mimic user interactions. Logging is integrated to capture test results, facilitating debugging and continuous improvement. Overall, these tests confirm the platform's robustness, with `pytest`s covering individual test cases spanning unit, integration, and system levels, and real test cases contributing to the successful public release and open-source repository.

Pytest

"conftest.py" provides shared pytest fixtures, such as triplestore and async clients, enabling consistent setup across tests. This foundational module supports all others, ensuring efficient test execution and resource management. Collectively, these test modules underpin the project's trust and accountability, validating feasibility through empirical evidence and aligning with the open-source ethos by being publicly available for community scrutiny.

The "test_etl_extractors.py" module focuses on verifying the ETL extraction layer, testing the creation and functionality of extractors for various Cardano entities such as accounts, epochs, blocks, transactions, multi-assets, scripts, datums, stake addresses, stake pools, delegations, rewards, withdrawals, governance actions, DRep registrations, and treasuries. It employs pytest fixtures to create database sessions, asserting that extractors are properly instantiated via the ExtractorFactory, handle batch sizes correctly, retrieve total counts and last IDs accurately, and serialize data (e.g., epoch timestamps) without loss. Asynchronous tests validate batch extraction for large datasets, while error cases check for invalid extractor types, raising appropriate ValueErrors. This module ensures the data ingestion foundation is solid, preventing issues in downstream transformations and loads.

"test_etl_transformers.py" complements the extractors by testing the transformation of extracted relational data into RDF triples. It covers transformer classes mirroring entity types, such as transforming account data into RDF with predicates like "hasStakeAmount" or "delegatesTo". Using mock extracted data, tests assert correct ontology alignment, handling of nested structures (e.g., transaction outputs), and generation of Turtle-formatted RDF. The TransformerFactory is validated for dynamic creation, with edge cases for empty datasets or malformed inputs, ensuring semantic consistency in the knowledge base.

The "test_etl_loader.py" module rigorously evaluates the loading process into the Virtuoso triplestore. It includes tests for initialization, and progress metadata saving using ETLProgress and ETLStatus models. Asynchronous fixtures manage graph cleanup before and after tests to isolate environments. Key validations cover empty data handling (no graph creation), data integrity checks (comparing expected vs. actual triple counts), graph statistics retrieval (subjects, predicates, objects, triples), and clearing non-existent or populated graphs. Error scenarios, like saving progress with ETLStatus.ERROR, are tested to confirm no persistence occurs. This module guarantees reliable data population and metadata management, critical for milestone deliverables like knowledge base bootstrapping.

"test_etl_service.py" assesses the overarching ETL workflow orchestration, testing the ETL service's ability to coordinate extraction, transformation, and loading across entities. It verifies progress tracking, status updates (e.g., RUNNING to COMPLETED), and error recovery, using mock dependencies to simulate full pipelines. Asynchronous tests measure performance under load, ensuring scalability for real-time Cardano data ingestion.

"test_etl_pipeline.py" provides end-to-end pipeline testing, integrating extractors, transformers, and loaders in sequence. It simulates complete ETL runs on sample Cardano data, asserting that data flows correctly from PostgreSQL to RDF to triplestore, with validations on triple counts, ontology compliance, and metadata accuracy. This module includes timeout handling and resource cleanup, confirming the platform's data pipeline feasibility as per the project's capability assessments.

"test_api.py" covers API endpoints, using httpx's AsyncClient for asynchronous requests. It tests health checks, NL query processing (including streaming responses), SPARQL execution, dashboard CRUD operations, user authentication (login, OAuth), metrics retrieval, and admin functions like ETL initiation and cache flushing. Assertions check status codes, response bodies, and error handling, ensuring secure and performant API interactions.

SPARQL Queries

The "sparql_tests.py" and "sparql_generation_tests.py" modules target SPARQL query handling and generation. "sparql_tests.py" executes predefined SPARQL queries against a test graph populated with sample data, verifying results for scenarios like top ADA outputs, average fees, wallet counts, and governance votes, using assertions on binding counts, ordering, and aggregations. "sparql_generation_tests.py" focuses on LLM-generated SPARQL from natural language, testing accuracy in translating queries like "largest ADA transactions" into valid SPARQL, with validations for syntax, variable detection (e.g., ADA variables), and result formatting utilities.

Natural Language Queries

"oc_tests.py" addresses specialized validations on how the LLM model answers general questions.

"nl_query_tests.py" is a standalone script for testing the natural language query pipeline as the full integration test for CAP, not using pytest but runnable manually. It includes a test harness for health checks and sample queries (e.g., current epoch, latest blocks, top stake pools), streaming responses via SSE, and summarizing results. This module aids in verifying all the modules of the system, including SPARQL generation and query engine management to contextualize LLM model processing and responses, and user-facing features during development.

4.10 Use Cases

As proof of the platform's conceptual effectiveness in addressing real-world analytical needs within the Cardano ecosystem, a set of ten representative use cases was defined early in the project, drawing from common queries that stakeholders such as developers, researchers, ADA holders, and governance participants might pose. These use cases were documented within the technical report produced during the first milestone, which outlined their relevance to the ontology and knowledge graph. Each use case was initially implemented by crafting

corresponding SPARQL queries that leverage the developed ontology, ensuring semantic accuracy and alignment with Cardano's data structures. These queries were then integrated into the LLM query engine, allowing natural language inputs to be translated into executable SPARQL, executed against the triplestore, and formatted for user-friendly outputs, including visualizations where applicable. The implementation process involved iterative refinement during the LLM query engine development milestone, incorporating utilities for ADA conversions, token name decoding, and Vega-compatible charting to handle blockchain-specific data nuances. was conducted in the final milestone, using sample Cardano data to simulate real-time scenarios, with results verified for correctness in terms of data retrieval, aggregation, and presentation.

The first use case, "What were the 10 largest ADA transactions in the last 24 hours?", translates to a query that filters transactions by timestamp, aggregates output values in lovelace (converted to ADA), and orders them descendingly, limited to the top ten. This query demonstrates the platform's real-time capabilities, utilizing time-bound filters and value aggregations from the ontology's transaction and output classes. Similarly, "What were the average transaction fees last week?" aggregates fee values over a seven-day period, employing average calculations on transaction metadata, which highlights the CAP's efficiency in handling temporal data transformations.

For distribution-focused queries, "How many accounts hold over 200,000 ADA?" counts wallets exceeding the threshold by summing stake and balance amounts, filtered accordingly, showcasing the ontology's account and holding representations. "What are the average staking rewards for accounts holding more than 1 million ADA?" extends this by averaging reward values for qualifying accounts, integrating reward and delegation predicates to reflect staking dynamics.

Visualization-oriented use cases, such as "Plot a bar chart showing monthly smart contract deployments in the last year," generate grouped counts by month, with results formatted via Vega utilities for bar chart rendering, illustrating the platform's support for graphical insights through monthly aggregations on contract embedding timestamps. "List the number of NFTs minted in the last month" counts distinct NFTs minted within the timeframe, leveraging asset and minting classes to capture non-fungible token activities.

Transfer and creation trends were addressed in "Which tokens were the most frequently transferred in the last week," which groups and counts token transfers by currency, ordered by frequency, demonstrating multi-asset handling in the knowledge graph. "How many new accounts were created daily over the last month" aggregates new account appearances by date, using first-appearance predicates to track ecosystem growth.

Stake distribution was demonstrated in "What percentage of delegated ADA is to the top 10 stake pools," calculating sums for top pools relative to total staked ADA, with subqueries for ranking and percentage computation, emphasizing delegation ontology elements. Finally, "How many accounts voted on the latest governance proposal" counts distinct voters on the most recent proposal, identified by timestamp ordering, underscoring the platform's coverage of Cardano's governance features.

These implementations were evaluated through automated execution on the platform, resulting in a metrics summary that confirms 100% success across all ten queries. The total execution mean time was 399.24 seconds, with an average of 39.92 seconds per query, a minimum of 17.71 seconds, and a maximum of 151.22 seconds, the latter occurring for the largest transactions query due to its data-intensive aggregation. Individual breakdowns showed efficient performance for simpler counts, such as the governance votes query at 17.71 seconds, while more complex aggregations like token transfers at 47.24 seconds reflected the depth of processing involved. This evaluation, conducted as part of the final testing phase, not only validated the platform's feasibility but also informed optimizations in caching and query normalization, ensuring scalability for production use. The successful execution of these use cases directly supports the project's impact goals, providing empirical evidence of simplified, real-time analytics that empower broader community participation.

5. Final Remarks

The Cardano Analytics Platform represents a significant advancement in facilitating access to blockchain data analysis. By combining a domain ontology with natural language processing capabilities, it bridges the gap between technical blockchain data and meaningful insights accessible to a wide range of stakeholders.

The platform's architecture introduces several key innovations for the Cardano community. The use of semantic technologies enables rich context-aware queries that understand the relationships across different aspects of the Cardano ecosystem. The LLM-powered query system enhances accessibility, allowing users without technical expertise in programming languages or blockchain internals to perform complex analytics. In addition, the modular design supports continuous improvements to both the ontological model and analytical capabilities as the Cardano ecosystem evolves.

By focusing on user personas and specific use cases discussed in the technical report delivered in Milestone 1, the platform ensures practical utility while maintaining the flexibility to adapt to emerging needs in blockchain analytics. Upon completing the milestones outlined in the project proposal, the team envisions productizing the platform and integrating off-chain analytics to enrich analysis.

This project has fulfilled its objectives, delivering an LLM-powered analytics prototype that can transform how users interact with Cardano on-chain data. Through development across all milestones, the prototype stands as an open-source tool that enhances accessibility and usability. The achievements underscore the project's success in addressing key challenges in blockchain analytics, paving the way for future enhancements and community-driven expansions. This report marks the end of the Fund 13 phase, with the source code now available for widespread adoption, contribution and further development to achieve productization.

6. Acknowledgments

This work was supported by a grant from the Fund13 of Project Catalyst. We would like to express our gratitude to the Cardano community and the Project Catalyst Team.