Final Report - Corporativo

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

1. General project information

Information	Details
Partner Company	Startup de Pagamento LTDA
Project name	Dumbo's Project
Development Team	Gustavo Ferreira
Advising Professor	Guilherme Cestari
Start date	21/04/2025
Estimated End Date	26/06/2025

2. Project Objectives

The main objective of this project is to develop an optimized solution to correlate customer action event data from the API and Database of payments with their respective users **using graphs**. This will enable a more integrated and automated view of customer interactions, reducing data fragmentation and improving operational efficiency.

Additionally, an interactive dashboard will be developed for analyzing this data, serving two main audiences:

- **Support Team:** Will have a tool that speeds up problem diagnosis, avoiding manual queries across multiple databases and reducing SLA breaches.
- **Area Developers:** Will be able to publish data on product usage by their clients following the defined data standard, resulting in a unified database of event and customer data.

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

3. Project Scope and Backlog

The project aims to structure a graph-oriented database in Neo4j to store and correlate user interaction events with their respective actions already existing in the database or obtained via API. The main objective is to improve service efficiency and generate strategic insights by providing an integrated view and facilitating complex queries.

3.1. Epics and Features

Below are the epics that were developed during the module, which were discussed and planned in collaboration with the partner company and the advising professor.

- **Epic 1: Definition of Database Architecture and Initial Modeling**
  - **Feature 1.1:** Comparative analysis between PostgreSQL and Neo4J based on project needs and data characteristics.
  - **Feature 1.2:** Creation of an Entity-Relationship (ER) diagram that clearly represents the entities and their relationships.
- **Epic 2: Technical Documentation and Environment Setup**
  - **Feature 2.1:** Detailed description of the attributes and entities present in the ER model, along with their respective relationships.
  - **Feature 2.2:** Development of a Dockerfile to provision and run Neo4J in an isolated environment, facilitating deployment.
- **Epic 3: Initial Development of the Data Structure**
  - **Feature 3.1:** Preparation of the first version of the SQL script for table creation and initial data insertion.
  - **Feature 3.2:** Analysis and implementation of indexes on frequently used fields to optimize query performance.
- **Epic 4: Testing and Performance Validation**
  - **Feature 4.1:** Generation of realistic mock data and execution of simulated tests to validate database queries and performance.

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

- **Feature 4.2:** Ensuring that simple queries (e.g., retrieving customer data) return results in up to 1 second, and complex ones (e.g., retrieving payment events) in up to 5 seconds.

- **Epic 5: Query Optimization and Final Documentation**
  - **Feature 5.1:** Development of specific queries to address the system's main questions.
  - **Feature 5.2:** Creation of final documentation with best practices for the database structure and fields.

3.2. Out of scope

1. Development of the dashboard and frontend interface (to be addressed in another module).
2. Use of a relational database (PostgreSQL) as the primary data source.
3. Direct integration with other systems without using the intermediary API.

4. Module Roadmap and High-Level Schedule

As in Inteli's standard work model, it was agreed with the partner company that each sprint would last two weeks, ending with a presentation (sprint review) to validate the results achieved and align the next steps. Below is an overview of what was carried out during each sprint:

Sprint 1:

1. Use of Clean Architecture: Definition and implementation of clean architecture, ensuring that the code is modular, scalable, and easy to maintain.
2. Swagger: Configuration of Swagger for automatic API documentation and endpoint validation.

Sprint 2:

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

1. Repository with ORM Drizzle: Implementation of the Repository pattern using the Drizzle ORM for database abstraction and efficient data access.
2. Use Case with Business Rules: Development of the application layer, implementing use cases that define business rules.

Sprint 3:

1. Command Layer Validating and Building: Creation of the Command layer, which will be responsible for validating and building commands for executing operations in the system.

Sprint 4:

1. Endpoint Routes: Creation of endpoint routes to interact with the frontend and allow the system to process requests correctly.

Sprint 5:

1. Unit Tests: Implementation of unit tests to ensure that backend functionalities are well-tested and that the code continues to function as expected.

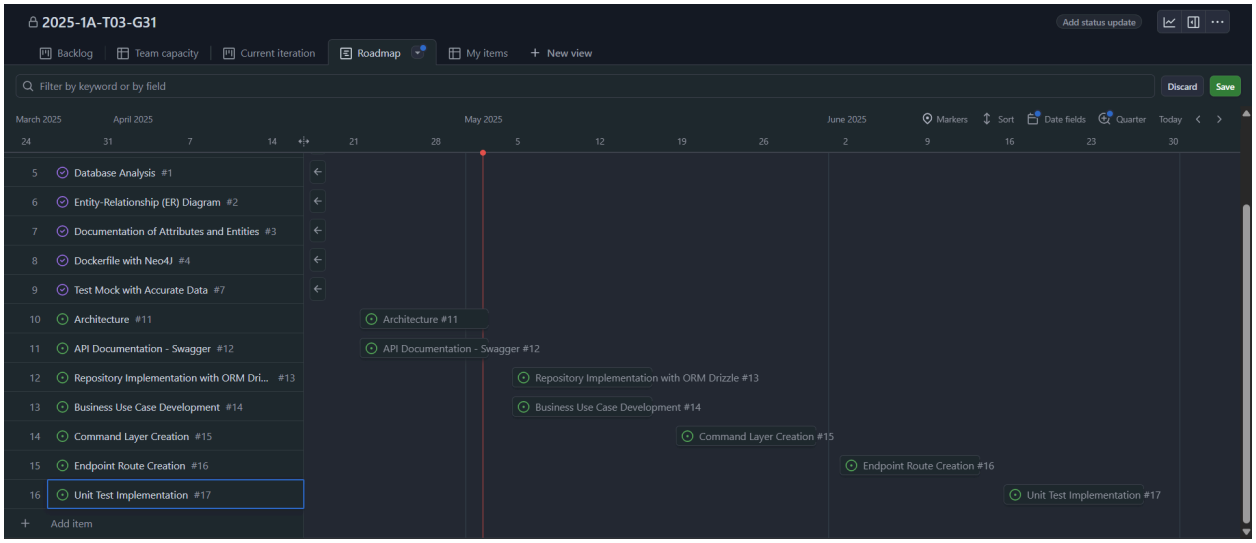

Figure 1: Project Kanban Board on GitHub

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

5. Project Constraints

Since this is an MVP (Minimum Viable Product) intended for validation before final implementation, the project must follow key constraints to ensure the solution's security, compliance, and efficiency, as well as data protection.

The main constraints to be considered are:

1. **Restricted Data Access**

a) External users or those without a direct link to the company will not have access to internal information, ensuring data security and confidentiality.

2. **Dependency on Existing Infrastructure**

a) The solution must integrate with tools already used by the partner, such as Azure and AWS, without requiring drastic changes to the current infrastructure.

3. **Data Processing Restrictions**

a) Data handling and analysis must comply with strict security and regulatory rules established by the LGPD (General Data Protection Law), preventing unauthorized access or exposure of sensitive information such as: company name, CNPJ or CPF, list of performed transactions, and recipient companies.

b) The implementation must not compromise the performance of existing systems or databases.

4. **Preservation of Current Customer Service Structures**

a) The new system must not completely replace current service practices without a carefully planned and validated transition.

6. Development

6.1.1. Clean Architecture

Clean Architecture, proposed by Robert C. Martin (Uncle Bob), aims to create modular systems that are independent of frameworks, user interfaces,

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

databases, and external devices. The focus is on a clear separation of responsibilities between layers, ensuring more scalable, maintainable, and testable code.

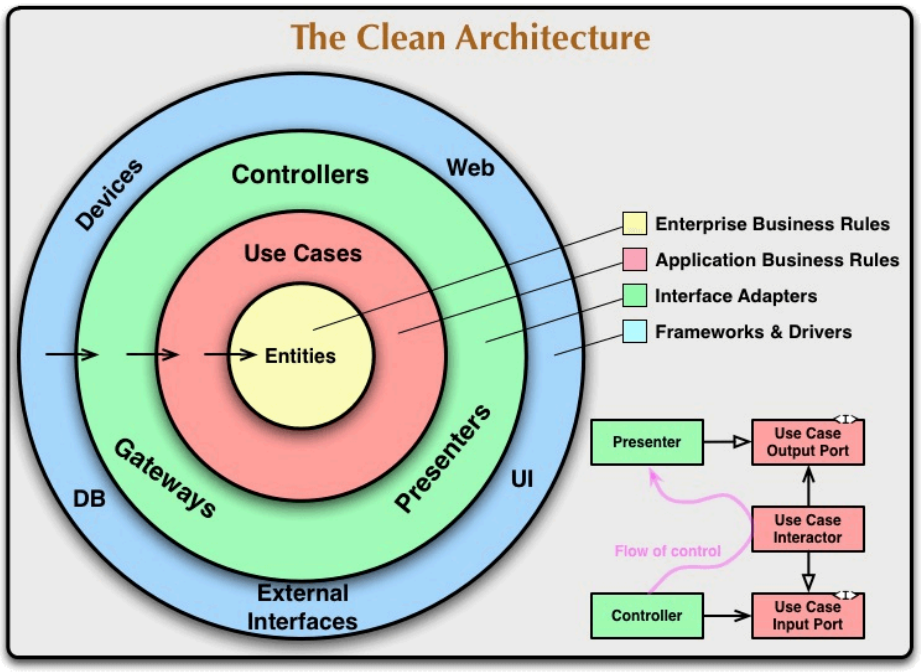

Figure 14: Diagram of Clean Architecture.

1. Enterprise Business Rules

Diagram Location: Center (yellow).
Responsibility: Contains business entities, which represent generic business rules that do not change over time or with technology.
Implementation in the Sprint:

- **Sprint 2:** Modeling of entities with the core rules of the application.

2. Application Business Rules (Use Cases)

Diagram Location: Second layer (red).

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

Responsibility: Defines the application's use cases, where specific business rules are implemented.

Implementation in the Sprints:

- **Sprint 2:** Creation of use cases with business rules using the defined models.
- **Sprint 3:** Creation of the command layer, responsible for validating and constructing operations (Command Pattern).

3. Interface Adapters

Diagram Location: Third layer (green).

Responsibility: Acts as the adaptation layer between the external and internal world, composed of:

- **Controllers:** Interpret requests and invoke the use cases.
- **Presenters:** Format the use case responses for the UI.
- **Gateways/Repositories:** Data access adapters.

Implementation in the Sprints:

- **Sprint 1:** Definition and initial structuring of the clean architecture.
- **Sprint 2:** Implementation of the Repository Pattern using the Drizzle ORM.
- **Sprint 4:** Implementation of endpoint routes, connecting controllers to the frontend.

4. Frameworks & Drivers (External Interfaces)

Diagram Location: Outermost layer (blue).

Responsibility: Contains external tools such as databases, web frameworks,

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

UI, etc. These should not influence the inner layers.

Implementation in the Sprints:

- **Sprint 1:** Integration with Swagger for automatic API documentation.
- **Sprint 4:** Connection with the UI layer through the exposed routes.

5. Flow of Control

The flow always moves from the outermost layer toward the center.

For example:

- A **Controller** receives an HTTP request;
- It forwards the request to the **Use Case Interactor**;
- Which accesses the **Gateway** to retrieve data;
- And then returns a formatted response via the **Presenter**.

6.1.2. Swagger Integration for API Documentation

During Sprint 1, Swagger was integrated to enable automatic generation of API documentation. The configuration ensures that all endpoints, request/response models, and relevant metadata are exposed through a user-friendly interface, facilitating development, testing, and integration with external systems. The Swagger UI is accessible via a dedicated route and updates dynamically as the application evolves. All configuration details and usage instructions are available in the GitHub repository.

6.1.3. Graph Architecture and Domain Modeling

During Sprint 2, the architectural foundation of the graph-based system was established, following a clean separation of concerns between Domain and Infrastructure layers. The Domain Layer encapsulates core business logic and abstractions, including entities (Node, Edge), shared result and

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

pagination patterns, and repository interfaces that define data access contracts. The Infrastructure Layer provides concrete implementations based on Neo4j, along with assemblers for translating between domain models and persistence representations. This architecture promotes maintainability, testability, and infrastructure agnosticism.

1. **Result Pattern Implementation**

A robust Result pattern was implemented to enforce type-safe success and failure handling throughout the domain. It enables expressive error propagation, consistent return types, and clear communication of operation outcomes. This pattern underpins several core use cases and ensures reliable domain behavior.

2. **Generic Pagination Support**

Pagination was introduced as a reusable utility to standardize list responses across the application. The design includes page size control, total count tracking, and navigation helpers. By embedding pagination into repository responses, the system guarantees a consistent interface for all paginated operations.

3. **Domain Entities: Node and Edge**

The domain model was expanded to define two central entities: Node and Edge. Nodes represent graph entities with typed classifications, unique identifiers, attributes, and timestamp metadata. Edges define directed relationships between nodes, enriched with weight, type, and optional custom attributes. These entities are designed to reflect core business semantics while enabling efficient graph traversal and querying.

4. **Repository Pattern and Neo4j Integration**

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

The Repository pattern was applied to abstract data persistence, ensuring a clean API for data access regardless of the underlying technology. The infrastructure layer provides Neo4j-based implementations, including translation logic via assemblers to bridge between the domain and data models. This setup simplifies testing and future migration by isolating infrastructure concerns.

5. **Use Case Layer**

A set of use cases was developed to encapsulate business logic and orchestrate domain operations in a clean, testable manner. Each use case adheres to the application boundary principles, consuming domain entities and repositories while isolating business-specific flows from infrastructure concerns.

The following operations were implemented:

- **List Nodes by Filters**: Enables filtered and paginated retrieval of nodes based on type, attributes, and search parameters. Leverages domain repositories to abstract persistence logic.

- **List Edges by Filters**: Supports querying edges with flexible filters such as node relationships, types, and attributes. Includes pagination and consistent response structures.

- **Create User**: Handles user registration, including validation and uniqueness checks. Creates domain-level user entities and persists them through repository contracts.

- **Login User**: Authenticates users by validating credentials and issuing access tokens. Ensures secure login flow with clear error handling using the Result pattern.

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

6.1.4. Command Layer

During Sprint 3, a Request Assembler was introduced within the Command Layer to bridge the interface between the GraphQL input and the domain logic. This component is responsible for transforming and validating incoming client data, ensuring type safety and consistency across layers.

The assembler handles pagination parameters, normalizes optional inputs, and maps GraphQL input types to internal filter and command structures. By enforcing validation rules and applying default values, it guarantees reliable and predictable downstream processing. This layer contributes to maintainable API logic and decouples external input representation from internal processing models.

6.1.5. Query Layer

The Query Layer was expanded to implement dedicated orchestration logic for listing nodes and edges. These query handlers act as coordinators between domain repositories and presentation-level assemblers.

Each query encapsulates pagination logic, error handling using the Result pattern, and enforcement of relevant business rules. Responses are wrapped consistently to expose a clean, stable contract to the GraphQL layer.

This structure promotes reusability and testability, aligning with application boundary principles and ensuring that querying logic remains focused and decoupled from infrastructure dependencies.

6.1.6. Server Execution and Runtime Configuration

The server runtime was configured to expose the GraphQL API and serve auxiliary routes such as Swagger documentation. The entry point resides in

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

the cmd/server package, orchestrating service wiring, dependency injection, and environment configuration.

Upon initialization, the server loads environment variables, sets up dependency containers, binds the GraphQL handler to a defined route, and activates auxiliary monitoring or debugging endpoints.

Hot-reload capabilities and logging are integrated for improved development experience, while production-ready configurations ensure robust execution in deployment environments.

This setup ensures seamless startup, scalability, and integration readiness across different stages of the development lifecycle.

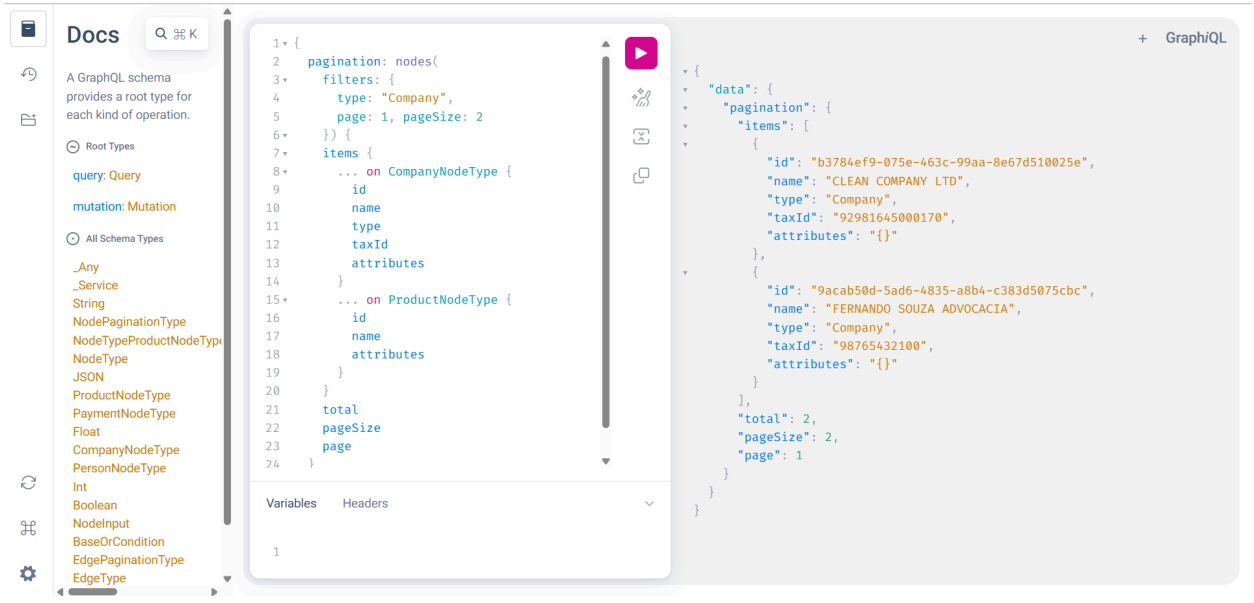

Figure 15: Server running with API response.

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

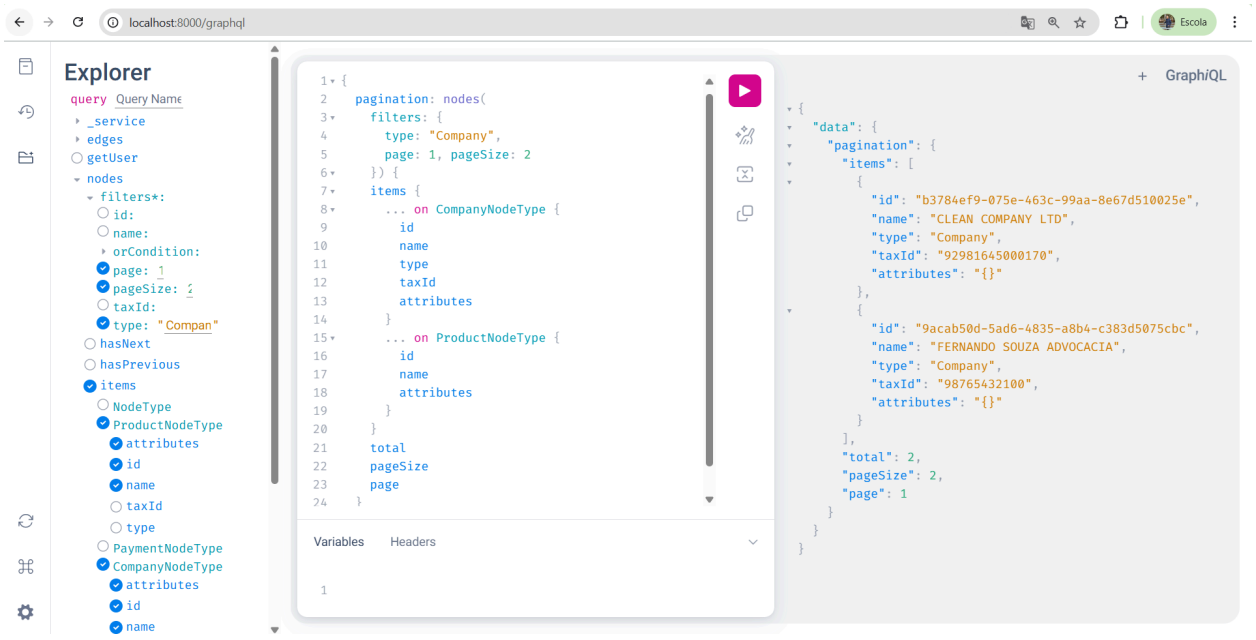

Figure 16: Playground explore with all params allowed.

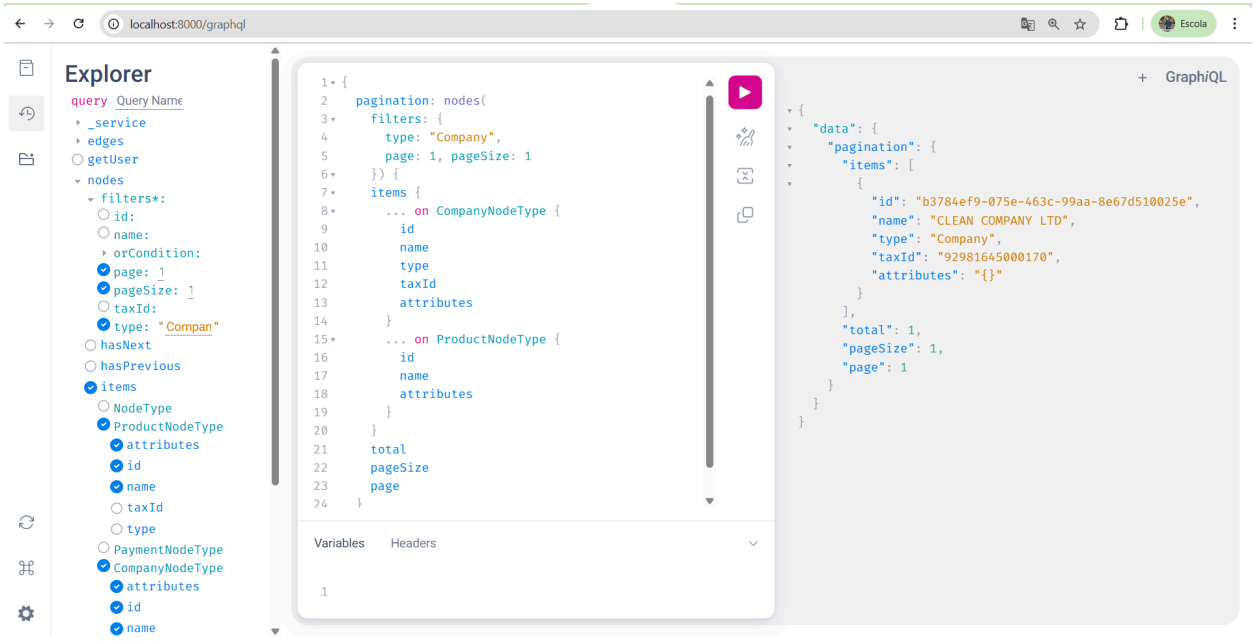

Figure 17: Example of pagination usage.

6.1.7. Unit Test Implementation

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

During Sprint 5, a comprehensive suite of unit tests was developed to validate the core behavior of domain use cases, entity logic, and infrastructure interactions in a deterministic and isolated manner. The testing strategy was aligned with the architectural principles defined in earlier sprints, ensuring full coverage across the Domain, Use Case, and Assembler layers.

The testing framework adopted was Pytest, chosen for its simplicity, extensibility, and compatibility with modern Python testing practices. Tests were structured around the Arrange–Act–Assert paradigm, promoting clarity and reliability in each scenario. Fixtures and test doubles were introduced via conftest.py to simulate external dependencies and control execution environments for repository interactions and result propagation.

Key testing areas included:

Domain Entities: Verification of attribute constraints, value object behaviors, and state transitions for Node and Edge entities.

Use Cases: Independent validation of business rules such as filtered listing, user creation, and authentication, with explicit success and failure assertions using the Result pattern.

Assemblers: Assurance of data normalization, default fallback behavior, and GraphQL-to-domain model mapping under various input conditions.

To guarantee consistency in paginated responses, pagination utilities were also tested with different page sizes, empty states, and boundary conditions. Repository interfaces were covered using mocks that simulate Neo4j responses without requiring actual database access, maintaining test isolation and speed.

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

- **Continuous Integration and Test Automation**

The unit test suite was fully integrated into the CI pipeline, reinforcing a test-driven development workflow and preventing regressions. Upon every commit or pull request, GitHub Actions triggers a dedicated job that:

1. **Installs Python dependencies using a cached virtual environment.**
2. **Runs static code analysis via Flake8 and type checking with Mypy.**
3. **Executes all Pytest test cases, tracking coverage with pytest-cov.**
4. **Fails the pipeline if any assertion errors, unhandled exceptions, or code regressions are detected.**

Test coverage reports are published as pipeline artifacts. This ensures that quality gates are enforced at the code review stage and that functional correctness is continuously validated across development iterations.

This structured approach to unit testing and automation contributes directly to system reliability, maintainability, and onboarding efficiency, enabling confident refactors and safe extension of the codebase in future sprints.

7. Conclusion

The design and implementation of a robust GraphQL API—structured around clean architectural principles, use case orchestration, and infrastructure abstraction—enabled efficient interaction with the graph data model. The API acts as a unified and scalable interface for querying and manipulating nodes and relationships, while enforcing validation, pagination, and secure access control. Its modular construction, paired with a fully automated testing and deployment pipeline, reinforces maintainability and adaptability, ensuring that future extensions or integrations can be delivered with confidence and consistency.

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br

This combination of data architecture and API design not only supports analytical objectives, but also lays a solid foundation for future innovation, enabling smarter systems that evolve in response to user behavior and business needs.

Inteli - Instituto de Tecnologia e Liderança
Avenida Professor Almeida Prado, 520, Butantã, São Paulo, SP.
CEP: 05508-070
www.inteli.edu.br