Integrating and Evaluating a GraphQL Extraction Layer

BACHELOR THESIS

Benjamin Georg Koller

Submitted on 30 March 2026

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Thomas Wolter
Prof. Dr. Dirk Riehle, M.B.A.

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 30 March 2026

License

Erlangen, 30 March 2026

Abstract

Software productivity metrics are indicators for software quality, development efficiency and developer satisfaction, remaining crucial to project success. The MECOIS project provides a robust solution for analyzing development data. Central to this is a data foundation, leveraging automated pipelines to extract software-engineering metrics from development platforms such as GitHub. However, even though there is already an existing extraction layer, it relies exclusively on the REST API. Following a design science approach, this thesis introduces a complementary extraction approach based on the GraphQL API, on existing literature and established architectures. We analyze the accessible data, develop a solution that seamlessly integrates with the existing pipeline and demonstrate its applicability using realworld data from the AMOS program stored on Github. Furthermore, the subsequent evaluation provides a comparative analysis of the REST and GraphQL approaches, specifically focusing on data completeness, request efficiency and rate-limit consumption. We found that the GraphQL-based approach can reduce the number of Rate-Limit Cost drastically, in our demonstration by up to 97.8% for fork extractions and enables comprehensive access to metadata, such as custom project fields and timeline events, which are unavailable via the REST API.

Contents

Acronyms **xi**

1 Introduction **1**

2 Problem Identification **3**
2.1 The GitHub REST API . . . 3
2.2 Rate Limiting . . . 3
2.3 Over-fetching . . . 4
2.4 Under-fetching . . . 5
2.5 Inaccessible Data . . . 6

3 Objective Definition **7**

4 Solution Design **9**
4.1 Literature Review . . . 10
4.2 Field Coverage Analysis . . . 13
4.2.1 Approach . . . 13
4.2.2 Results . . . 13
4.3 Extraction Layer Design . . . 14
4.3.1 Architectural Overview . . . 14
4.3.2 Client Design . . . 15
4.3.3 Requestor Design . . . 15
4.3.4 Query Design . . . 15
4.3.5 Integration into the MECOIS Pipeline . . . 16

5 Implementation **17**
5.1 Development Environment and Tools . . . 17
5.2 Data Sources and Authentication . . . 17
5.3 Project Structure and Configuration . . . 18
5.4 Implementation of the Extraction Layer . . . 19
5.5 Storing Output . . . 20

6 Demonstration **21**
6.1 Data Sample 21
6.2 Configuration 21
6.3 Results 23
6.3.1 Functional Completeness 23
6.3.2 Over- / Underfetching 23
6.3.3 Efficiency 24

7 Evaluation **27**
7.1 Objective Criteria 27
7.2 Limitations 28

8 Conclusions **29**

Appendices **31**
A List of Repositories used in Demonstration 33
B Text Revision 34

References **35**

List of Figures

1.1 The Medallion Architecture in Data Lake from MECOIS. 1

2.1 Data under-fetching and the N+1 problem in REST-based architectures (taken from Quiña-Mera et al., 2023) 5

4.1 Overview of the solution design process 9

6.1 Benchmarking of execution times for scaling number of forks. . . . 24
6.2 Comparison of execution times. 25

List of Tables

2.1 Data categories with significant availability gaps in the GitHub REST API 6

4.1 Design principles derived from the literature review 13
4.2 Summary of the field coverage analysis 14

6.1 Field availability by API approach 23
6.2 Comparison of extraction metrics ($N = 0 \ldots 4$ forks) 23
6.3 Comparison of commit details extraction ($N = 611$) 24

1 Parent repository identifiers 33

Acronyms

API	Application Programming Interface
GraphQL	Graph Query Language
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
URL	Uniform Resource Locator
AMOS	Agile Methods and Open Source

1 Introduction

Software development projects generate a continuous stream of structured data. The MECOIS project[1] is a research platform whose purpose is to extract, transform and analyze this data from a variety of development tools, including GitHub, GitLab and Jira. By aggregating and analyzing various development metrics, MECOIS supports empirical software engineering research in areas such as productivity measurement, transfer pricing and impact assessment.

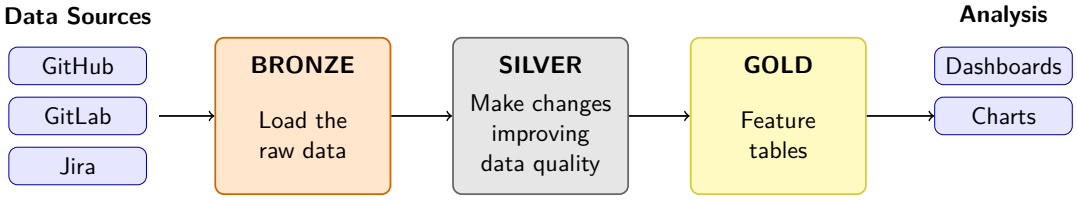


Figure 1.1: The Medallion Architecture in Data Lake from MECOIS.

Central to MECOIS is a pipeline architecture in which extractors fetch raw data from external APIs and pass it to downstream processing steps. To systematically structure the data, MECOIS implements the established Medallion architecture within its data lake (see Figure 1.1). Extractors load raw, unprocessed metrics from external data sources directly into the Bronze table layer. Subsequent transformations process this raw data to make changes improving data quality, forming the intermediate Silver table layer. Finally, the refined data is aggregated into feature tables within the Gold layer, providing a reliable foundation analysis.

This thesis addresses the step, where extractors load raw data into the Bronze table layer. This extraction layer of the architecture currently relies exclusively on the REST API. Its inherent limitations manifest in several ways: rate-limit exhaustion, over-fetching of unused data, N+1 request problems when assembling nested relationships and the absence of certain data fields from the REST schema.

[1] MECOIS - https://www.mecois.com/

GitHub's GraphQL API, introduced in 2016 (Daigle, 2016), offers a fundamentally different approach, specifically designed to address the limitations of traditional REST-based endpoints (GitHub, Inc., 2026a). Unlike REST, which relies on multiple endpoints returning fixed data structures, GraphQL uses a single endpoint with custom queries that allows clients to request exactly the data they need (V. Chaudhary et al., 2025). As Quiña-Mera et al. (2023) describe, this shifts the control over data shaping from the server to the client. Deeply nested resource graphs can be traversed in a single round trip, avoiding the penalty of sequential HTTP connections (Ambasht, 2023). These properties directly address the limitations of the REST-based extraction.

This thesis contributes to the field by integrating GitHub's GraphQL API into the MECOIS extraction layer. Building upon an analysis of current literature and the existing REST-based pipeline, we develop a robust GraphQL-based integration. Furthermore, we conduct a comparative demonstration of both interfaces, evaluating them based on data completeness, request efficiency and rate-limit consumption.

For a research approach we chose to follow the design science methodology. This choice was made, because creating a data extraction layer for the previously underexplored GraphQL API is an unsolved identified architectural problem in MECOIS, which design science is focused on solving. In detail, we follow the approach defined by Peffers et al. (2007) and structure the thesis as follows:

- In Chapter 2 we give an overview of related literature and establish the concrete problem this thesis addresses.
- In Chapter 3 we translate these shortcomings into a set of measurable objectives we expect the solution to meet.
- In Chapter 4 we present our approach to the design and development of our solution.
- In Chapter 5 we describe the technical tools and development steps taken to integrate the GraphQL extraction layer into MECOIS.
- In Chapter 6 we demonstrate the functionality of our solution by processing data from the AMOS programme[2].
- In Chapter 7 we evaluate our approach by assessing its efficiency and completeness against the predefined objectives and the existing REST implementation.
- In Chapter 8 we conclude the thesis with a reflection on the findings and an outlook on open research directions.

[2]AMOS Project - https://oss.cs.fau.de/teaching/the-amos-project/

2 Problem Identification

Technical design decisions and API limitations directly affect the efficiency of automated data collection at scale. This chapter details the shortcomings of the current REST-based pipeline and provides background context by evaluating related literature.

2.1 The GitHub REST API

The GitHub REST API follows the Representational State Transfer (REST) architectural style first introduced by Fielding and Taylor (2002). In a RESTful architecture, data is exposed as resources located at uniform URIs. Clients interact with these resources using standard HTTP methods (Doglio, 2018). This resource-centric approach means the server dictates the structure of the returned data (V. Chaudhary et al., 2025). The MECOIS pipeline utilizes a specialized client interface to interact with the GitHub REST API. This abstraction layer implements a pagination mechanism based on HTTP link headers, automatically following sequential references to retrieve complete datasets across multiple response pages.

For authenticated requests, the GitHub REST API permits up to 5,000 requests per hour (GitHub, Inc., 2026d). Secondary rate limits further restrict the number of concurrent requests and write operations. The existing extractors in MECOIS rely on this API for all GitHub data, including commits, issues, pull requests and pipeline runs.

2.2 Rate Limiting

The primary rate limit of 5,000 requests per hour appears sufficient for small repositories but becomes a binding constraint when extracting data from large-scale organisations. Such entities often encompass dozens of active repositories, many of which potentially contain thousands of individual commits and issues. A single extraction cycle across all resources within such an organisation can

quickly exhaust this budget, causing subsequent requests to fail until the rate-limit window resets.

Cost Model for the REST API

The GitHub REST API charges a flat cost of one point per HTTP request, irrespective of the size or complexity of the response (GitHub, Inc., 2026d). The default page size is 30 items, with a maximum of 100 (GitHub, Inc., 2026f). To retrieve N items from a paginated endpoint with a page size of P items per page, the number of requests and the rate-limit cost is:

$$C_{\text{REST}}(N, P) = \left\lceil \frac{N}{P} \right\rceil \tag{2.1}$$

For the common N+1 pattern, where a list of N parent resources is fetched first and then each resource requires one additional detail request, the total cost grows to:

$$C^{N+1}_{\text{REST}}(N, P) = \left\lceil \frac{N}{P} \right\rceil + N \tag{2.2}$$

Example. In the `amosproj` GitHub organisation, the repository `amos2025ws04-feature-board` contains 96 issues ($N \approx 100$). Fetching all (page size $P = 30$) and then enriching each issue with its custom project field values via a separate API call costs:

$$C^{N+1}_{\text{REST}}(100,\, 30) = \left\lceil \tfrac{100}{30} \right\rceil + 100 = 4 + 100 = 104 \text{ points}$$

2.3 Over-fetching

REST endpoints return fixed resource representations. The client cannot specify which fields it requires. When fetching commit data through the REST API, the response includes a comprehensive set of metadata for each entry. This includes deeply nested structures for authentication, tree references and various web links that provide an extensive but often excessive representation of the resource. For the analytical purposes of MECOIS, only a small fraction of these fields is actually required. The remaining data volume constitutes significant overhead during transmission and processing.

At scale, this over-fetching translates into unnecessary bandwidth consumption and increased JSON parsing times in PySpark. Quiña-Mera et al. (Quiña-Mera et al., 2023) identify over-fetching as one of the primary motivations for REST-to-GraphQL migration in data-intensive applications. Furthermore, Ambasht (2023)

emphasize that the ability to retrieve exactly the requested data using a single query is a defining advantage of GraphQL over traditional REST architectures, directly addressing this inefficiency.

2.4 Under-fetching

The inverse problem arises when a single REST endpoint does not return all data needed for a given use case, forcing the client to issue multiple sequential requests. This is commonly referred to as the N+1 problem: A particular endpoint does not provide enough information, so additional requests must be made to obtain the required information (Quiña-Mera et al., 2023). This is illustrated in Figure 2.1.

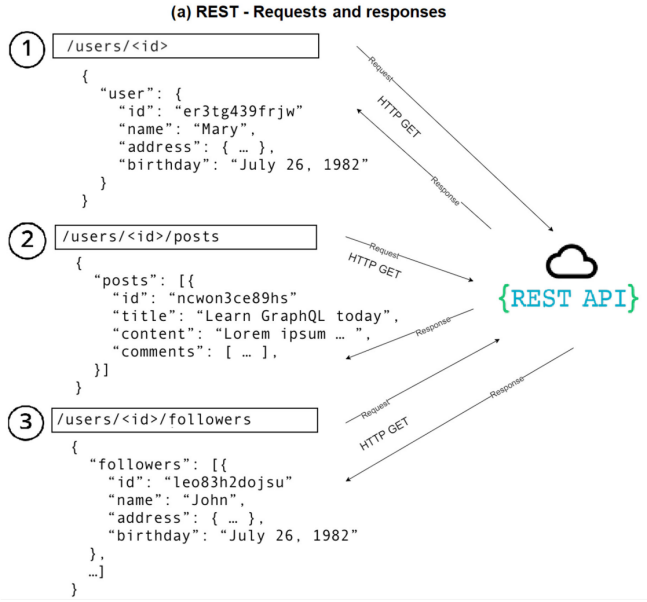


Figure 2.1: Data under-fetching and the N+1 problem in REST-based architectures (taken from Quiña-Mera et al., 2023)

.

A concrete example in MECOIS is the extraction of issue data enriched with project field values. The REST API provides no direct endpoint to retrieve custom project field values alongside issues. To obtain this data, a client must: (1) list all issues for a repository, (2) for each issue, query the project items associated with it and (3) for each item, query the field values. For a repository with 500 issues, this amounts to over 1,000 REST requests for a dataset that GraphQL can retrieve efficiently by allowing nested queries.

2.5 Inaccessible Data

Beyond performance bottlenecks, certain data required for the analytical objectives of MECOIS are entirely absent or only partially exposed through the GitHub REST API. Our analysis reveals that several categories of repository and project metadata cannot be retrieved in a structured format using standard REST endpoints, as summarised in Table 2.1.

Table 2.1: Data categories with significant availability gaps in the GitHub REST API

Category	**Restricted or Absent Metadata in REST**
Project Management	Granular state transitions (Kanban movements) and user-defined custom fields for Projects.
Interaction History	Unified timeline event streams for issues and nested review feedback threads.
Repository Insights	Structural dependency manifests and forum-style repository discussions.
Advanced Metadata	Historical branch reference data and deeply nested resource relationships.
Security Context	Comprehensive vulnerability metadata, including per-rule severity and dismissal details.

In summary, the identified limitations of the current REST-based integration within MECOIS highlight a need for a more flexible and efficient extraction approach. These constraints serve as the primary motivation for exploring alternative API technologies that better align with the requirements of large-scale data analysis. In the following chapter, we translate these observations into a set of concrete objectives that guide the design and implementation of a GraphQL-based extraction layer.

3 Objective Definition

In Chapter 2, we identified that the GitHub REST API, as currently used in MECOIS, exhibits significant limitations for comprehensive data extraction. It imposes strict rate limits on large-scale runs, returns more data per response than the client requires, forces an excessive number of sequential requests to assemble related resources and provides no access to certain data types of analytical interest. Given these constraints, we aim to design and integrate a GraphQL extraction layer that addresses these shortcomings within the existing MECOIS data lake architecture. For this, we have defined six objectives that our solution should fulfil.

To begin with, the GitHub GraphQL API and the existing body of related work on REST-to-GraphQL migration provide a substantial basis that does not yet have a consolidated counterpart in the MECOIS context. We want our solution to use existing literature and API documentation as its foundation, synthesising insights from prior comparisons of REST and GraphQL (Ambasht, 2023; Quiña-Mera et al., 2023) to identify recurring limitations and proven design patterns that guide the query design.

Secondly, the boundary between the REST and GraphQL interfaces has not yet been formally documented for the MECOIS context. We want our solution to be supported by a systematic field coverage analysis that determines which data fields and resource types are available exclusively through GraphQL, ensuring that design decisions are based on an evidence-based foundation.

Thirdly, based on the field coverage analysis, we want to implement GraphQL extractors for the identified data types. These extractors must follow the existing MECOIS pipeline conventions, use the same `PipelineStep` interface and produce output in the expected JSON format. This covers both new data types with no REST equivalent and GraphQL variants of existing REST extractors.

Fourthly, we need to consider the rate-limit constraints of the GitHub GraphQL API. With a fixed budget of 5,000 points per hour, we want our queries to minimise point consumption per extracted record. The client should also handle rate-limit exhaustion automatically, pausing and resuming pipeline runs without

data loss.

Fifthly, the pipeline must remain operational under adverse conditions. GitHub API responses occasionally contain errors, null entries, or gateway failures. We want the extraction layer to handle these gracefully by implementing retry logic and defensive null-checks throughout the pagination loops.

Lastly, we want to evaluate how well the GraphQL extraction layer performs relative to the existing REST approach by collecting quantitative evidence on data completeness, request count, rate-limit consumption and elapsed time. The evaluation should be reproducible using the same target organisation and credentials for both approaches.

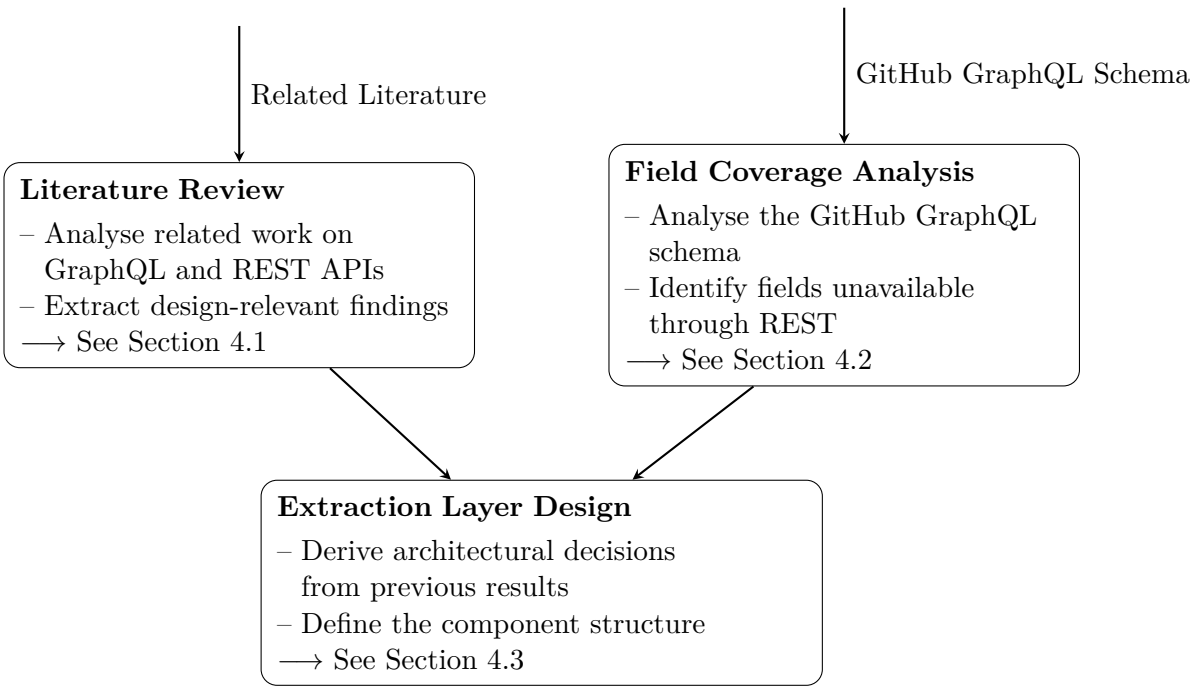
The objectives for the GraphQL extraction layer developed in this thesis can be summarised as follows:

1. The extraction layer should take the existing body of literature on GraphQL and REST APIs into account as a foundation for its design.
2. The solution should be supported by a systematic analysis of the fields available through GraphQL but not through the GitHub REST API.
3. The extraction layer should implement at least one new data source with no REST equivalent and at least one GraphQL variant of an existing REST extractor, both integrated into the MECOIS pipeline.
4. The solution should minimise rate-limit point consumption through deliberate query field selection and handle rate-limit exhaustion automatically.
5. The extraction layer should recover gracefully from transient API failures, null response values and network errors without interrupting a pipeline run.
6. The solution should be evaluated against the existing REST approach with respect to data completeness, request count, rate-limit consumption and elapsed extraction time.

4 Solution Design

In the previous chapter, we defined six objectives that guide the development of the GraphQL extraction layer. This chapter describes how the design of the extraction layer is derived from these objectives. We first review existing literature on GraphQL and REST APIs to identify recurring design-relevant findings. From this review, we extract the design principles that inform our solution. We then conduct a field coverage analysis of the GitHub GraphQL schema to determine which data types and fields are available exclusively through GraphQL. Lastly, we synthesise the results of both reviews into the architectural design of the extraction layer. Figure 4.1 gives an overview of the solution design process.

Figure 4.1: Overview of the solution design process



4.1 Literature Review

The first objective (Chapter 3) requires the extraction layer to take the existing body of literature on GraphQL and REST APIs into account as a foundation for its design. In this section, we review the key findings from related work that are directly relevant to the design decisions of the extraction layer. For each finding, we describe its implication for our context.

Query-Level Field Selection

A defining characteristic of GraphQL, consistently highlighted across the literature, is its ability to let the client specify exactly which fields to retrieve in each request. Quiña-Mera et al. (2023) identify this client-driven field selection as one of the primary motivations for adopting GraphQL in data-intensive applications, as it directly addresses the over-fetching problem inherent in REST architectures. V. Chaudhary et al. (2025) confirm this advantage in their comparative analysis, noting that REST endpoints return fixed resource representations, whereas GraphQL allows the client to shape the response to its precise needs.

Design implication. Each GraphQL query in the extraction layer should request only the fields that MECOIS requires for downstream analysis. This selective field approach reduces payload size and avoids unnecessary data transfer.

Nested Resource Traversal

The literature consistently identifies the N+1 request problem as a major limitation of REST APIs. Quiña-Mera et al. (2023) describe this pattern, in which a single endpoint does not provide enough information, forcing the client to issue additional requests to obtain the required data. Ambasht (2023) emphasise that GraphQL resolves this by allowing deeply nested resource graphs to be traversed in a single round trip. In practical migration studies, Brito et al. (2019) report measurable reductions in the number of HTTP requests when replacing REST calls with equivalent GraphQL queries.

Design implication. The extraction layer should exploit the ability of GraphQL to fetch nested resources in a single query. Where the existing REST extractors require multiple sequential requests to assemble related data, the GraphQL queries should combine these into a single request with inline fragments or nested connection fields.

Cost Model for the GraphQL API

Unlike the REST API, which charges a flat cost of one point per HTTP request (see Section 2.2), the GitHub GraphQL API calculates the cost of a query from

its slicing arguments (first, last) (GitHub, Inc., 2026c). These arguments must be specified explicitly by the client, with a maximum value of 100 (GitHub, Inc., 2026c). The total budget remains 5,000 points per hour, but the cost per request is derived from the number of nodes the query may return, not the number it actually returns.

The calculation proceeds in two steps. First, the API counts the total number of potential node requests by multiplying the slicing arguments across nesting levels. For a query with a single paginated connection using a slicing argument S, the node count is $1+S$. For a query that nests a second connection with slicing argument S_{inner} inside the first, the node count grows to $1+S+S\cdot S_{\text{inner}}$, because the server must be prepared to resolve the inner connection for each of the S potential parent nodes. Second, the node count is divided by 100 and rounded up to obtain the point cost of a single request (GitHub, Inc., 2026c):

$$c_{\text{GQL}}(S) = \left\lceil \frac{1+S}{100} \right\rceil \tag{4.1}$$

$$c_{\text{GQL}}^{\text{nested}}(S, S_{\text{inner}}) = \left\lceil \frac{1+S+S\cdot S_{\text{inner}}}{100} \right\rceil \tag{4.2}$$

The total rate-limit cost of paginating through N items is then the number of page requests multiplied by the per-request cost:

$$C_{\text{GQL}}(N, S) = \left\lceil \frac{N}{S} \right\rceil \cdot\ c_{\text{GQL}}(S) \tag{4.3}$$

Example. Consider the same scenario from Section 2.2: extracting 96 issues ($N \approx 100$) together with their custom project field values from the `amos2025ws04-feature-board` repository. Using the same page size as the REST example ($S = 30$) and requesting up to $S_{\text{inner}} = 10$ field values per issue, the per-request cost is:

$$c_{\text{GQL}}^{\text{nested}}(30,\ 10) = \left\lceil \frac{1+30+30\times 10}{100} \right\rceil = \left\lceil \frac{331}{100} \right\rceil = 4 \text{ points}$$

Paginating through all 100 issues requires $\lceil 100/30 \rceil = 4$ requests, yielding a total cost of $4 \times 4 = 16$ points and only 4 HTTP round trips. In contrast, the REST N+1 pattern (Equation 2.2) consumes 104 points across 104 HTTP requests for the same data, demonstrating a significant advantage in both rate-limit budget and network overhead.

Design implication. The extraction layer must choose slicing arguments deliberately. Batch sizes should be large enough to minimise the number of HTTP requests but should not exceed what is necessary, as deeply nested queries with

large slicing arguments can accumulate significant per-request costs. Furthermore, the client must monitor runtime rate-limit consumption and pause automatically when the budget is close to exhaustion.

Cursor-Based Pagination

All paginated connections in the GitHub GraphQL API use an opaque cursor model, so the text is like a black box. Unlike REST APIs that often use numerical offsets, this model requires specifying the number of items to fetch via a `first` or `last` argument (between 1 and 100) and then traversing the dataset using cursors (GitHub, Inc., 2026e). Each response includes a `pageInfo` object with a `hasNextPage` flag and an `endCursor` value. To retrieve the next page, the client must pass the current `endCursor` to the `after` argument (GitHub, Inc., 2026e). This approach is consistent with the findings of Brito et al. (2019), who note that cursor-based pagination provides more predictable performance for deep traversal of large datasets, as the server can jump directly to the cursor position instead of counting through preceding offsets.

Design implication. The extraction layer must implement a generic cursor-based pagination loop that can be reused across all data types. Each requestor should iterate over pages by advancing the cursor until `hasNextPage` returns `false`.

Error Handling and Partial Responses

Unlike REST APIs, which signal errors exclusively through HTTP status codes, GraphQL APIs can return partial data alongside an `errors` array in the same response (Quiña-Mera et al., 2023). In the GitHub context, this means that a single response may contain valid data for most requested nodes but include error entries for nodes that are inaccessible or have been deleted. Additionally, gateway-level failures (HTTP 502, 504) occur during periods of high API load.

Design implication. The extraction layer must not treat the presence of an `errors` field as a fatal condition. Instead, it should log errors, extract whatever valid data the response contains and use retry mechanisms with exponential backoff for transient gateway failures.

Summary of Literature-Derived Design Principles

Table 4.1 summarises the design principles extracted from the literature review. Each principle addresses at least one of the objectives defined in Chapter 3.

Table 4.1: Design principles derived from the literature review

Principle	Description	Objective
Selective fields	Request only the fields required by MECOIS.	1, 4
Nested traversal	Fetch related resources in a single query.	1, 4
Cost-aware batching	Choose slicing arguments to balance request count against point consumption.	4
Generic pagination	Implement a reusable cursor-based pagination mechanism.	1, 3
Graceful errors	Handle partial responses, null nodes and gateway failures without data loss.	5

4.2 Field Coverage Analysis

Our second objective requires the solution to be supported by a systematic analysis of the fields available through GraphQL but not through the GitHub REST API. To fulfill this requirement, we downloaded the complete GitHub GraphQL schema (GitHub, Inc., 2026b) and compared the types and fields it exposes against the endpoints documented in the GitHub REST API reference (GitHub, Inc., 2026a). Additionally, we used GraphQL Voyager[1] to obtain a visual overview of the schema's type relationships, which facilitated the identification of nested connections and exclusively GraphQL-accessible data types.

4.2.1 Approach

The analysis proceeded in three steps. First, we identified the data types currently extracted by the existing REST-based MECOIS pipeline (commits, issues, pull requests, pipeline runs, releases). Second, we examined the GraphQL schema for each of these types and recorded which fields have direct REST equivalents and which are exclusive to GraphQL. Third, we searched the schema for entirely new root types with no REST counterpart. The results are summarised in Table 4.2.

4.2.2 Results

Two categories stand out. First, Projects data is entirely absent from the REST API, meaning that custom field values, Kanban column movements and project views can only be accessed through GraphQL. Second, although both interfaces provide access to commit metadata, the current REST-based implementation in MECOIS relies on a local `git clone` to avoid the high operational overhead caused by REST under-fetching. The GraphQL API, in contrast, enables a streamlined, purely remote extraction process through nested connection tra-

[1] https://apis.guru/graphql-voyager/

Table 4.2: Summary of the field coverage analysis

Entity Type	REST availability	Notable GraphQL-only fields
Commits	Full	`statusCheckRollup`, per-branch traversal without local clone
Issues	Full	Inline timeline events, cross-reference details
Pull Requests	Full	Nested review threads with comments
Projects	None	Complete type: items, custom fields, views, status history
Discussions	None	Complete type: forum threads, answers
Code Scanning	Partial	Per-rule severity, dismissal metadata
Dependency Graph	None	Manifest files, dependency edges

versal. We illustrate this performance difference further in the demonstration (Chapter 6), where we compare both approaches using purely network-based requests.

These findings summarised in Table 4.2 directly inform which extractors the implementation must provide (Objective 3): at minimum one extractor for Projects data (no REST equivalent) and one for commits (GraphQL variant of an existing REST extractor).

4.3 Extraction Layer Design

Based on the design principles identified in Section 4.1 and the field coverage results from Section 4.2, we now describe the architectural design of the GraphQL extraction layer. The design establishes the component structure and interaction patterns that the implementation in Chapter 5 will follow.

4.3.1 Architectural Overview

The extraction layer is organised into three components that mirror the separation of concerns in the existing REST layer:

1. **Client:** A central GraphQL client component that encapsulates HTTP communication, bearer-token authentication and rate-limit handling. All queries pass through this single entry point.
2. **Queries:** Declarative GraphQL query strings, organised by root type (e.g.,

`Repository/`, `Organization/`). Each query requests exactly the fields required by the downstream MECOIS analytics and uses slicing arguments calibrated to minimise rate-limit consumption.

3. **Requestors:** Pipeline step classes that combine a query with the cursor-based pagination loop and null-node filtering logic. Each requestor extends the existing `PipelineStep` interface and passes its output through the standard `self.next()` mechanism.

This three-tier structure ensures that adding a new data type requires only a new query definition and a configuration of the generic requestor rather than substantial new Python code.

4.3.2 Client Design

The client is designed to handle two categories of failure transparently:

- **Rate-limit exhaustion:** After each request, the client should inspect the `X-RateLimit-Remaining` response header. When the remaining budget reaches one or fewer points, the client should read the `X-RateLimit-Reset` timestamp and sleep until that point in time before issuing further requests, rather than using arbitrary fixed delays.
- **Transient gateway failures:** HTTP 502 and 504 responses should be caught and retried with exponential back-off, capped at a configurable maximum number of attempts.

4.3.3 Requestor Design

A requestor is a reusable pipeline component that combines a GraphQL query with the common extraction logic: cursor-based pagination, rate-limit awareness and error handling. Given a query and a path to the paginated connection within the response, a requestor iterates through all pages, filters out null nodes and forwards the collected results to the next step in the MECOIS pipeline. This generic approach allows most data types to be extracted without writing specialised code. Only the query definition and the response path need to be configured.

4.3.4 Query Design

Each query is designed according to the principles of selective field retrieval and cost-aware batching. The queries request only the fields that are relevant for the MECOIS analytics layer and use inline fragments (`... on Type`) to handle polymorphic connections such as issue timeline events and project item content.

Query strings are stored as Python constants in dedicated modules, grouped by GraphQL root type, making them independently testable and auditable.

4.3.5 Integration into the MECOIS Pipeline

The GraphQL extraction layer is designed to coexist with the existing REST layer. Within the `sources/github/` directory, a dedicated `graphql/` sub-package separates the GraphQL components from the REST modules under `sources/github/rest/`. Both layers produce output in the same JSON format and use the same `PipelineStep` interface, so downstream transformations and storage steps require no modification. New GraphQL sources are registered in the central `pipeline-config.yml` alongside the existing REST sources.

This design ensures that the GraphQL extractors can be introduced incrementally, one data type at a time, without disrupting the existing pipeline. The implementation of this design is described in Chapter 5.

5 Implementation

This chapter describes the technical realization of the GraphQL extraction layer within the MECOIS project. We detail the development environment, the integration of data sources, the project's configuration patterns and the mechanisms for data storage. The implementation follows the architectural design established in Chapter 4.

5.1 Development Environment and Tools

The implementation was developed in Python 3.11 within a dedicated fork of the existing MECOIS project[1]. For iterative query design, we employed the Altair GraphQL Client[2] as a local desktop application. Furthermore, we utilized GraphQL Voyager[3] to visually explore and navigate the complex GitHub GraphQL schema.

5.2 Data Sources and Authentication

The component targets the GitHub GraphQL API. Unlike the REST API, which is spread across numerous endpoints, the GraphQL layer interacts with a single entry point: https://api.github.com/graphql.

The primary data source for the development and initial validation of the extractors are the repositories within the AMOS programme. These provide a rich set of development data, including repositories, issues, pull requests and historical activity, which serve as a good aquivalent for data which MECOIS project aims to extract.

Authentication is handled via a Personal Access Token. During the implementation, we prioritized security by ensuring that tokens are never embedded in the

[1]MECOIS Fork - https://github.com/Mecois/mecois-Stud-Koller
[2]Altair GraphQL Client - https://altairgraphql.dev/
[3]GraphQL Voyager - https://apis.guru/graphql-voyager/

source code and then are version controlled, in worst case to a public repository. Instead, the application reads the credentials from the local environment file. This setup allows the pipeline to run in various environments, such as local development machines or continuous integration runners, without modifying the codebase.

5.3 Project Structure and Configuration

The GraphQL extraction layer is integrated into the existing MECOIS module structure, following the project's established conventions for source-specific extensions.

Project Structure

All GraphQL-related components reside within the `sources/github/graphql/` directory, which is structured as follows:

- `client/`: encapsulates the HTTP communication logic and rate-limit tracking.
- `requestors/`: contains the classes responsible for orchestrating the extraction of specific data types.
- `queries/`: stores the GraphQL query definitions as reusable string constants.

Configuration

The extraction process is controlled through the central `pipeline-config.yml` file. This configuration provides a declarative way to register new GraphQL-based data sources. Each entry in the configuration specifies the type of extractor to use and the target parameters (e.g., repository names or organization handles). This high-level configuration separates the extraction logic from the specific data targets, allowing users to extend the pipeline's scope without changing Python code.

The following code snippet Listing 5.1 shows a typical query definition as stored in the `queries/` directory. It illustrates how the extraction layer requests specific fields to reach parity with existing REST data while minimizing payload size:

```
"""
GraphQL Query for Repository General Info parity.
"""

REPOSITORY_INFO_QUERY = """
query GetRepositoryInfoParity($owner: String!, $name: String!) {
  repository(owner: $owner, name: $name) {
    name
    createdAt
    updatedAt
    diskUsage
    defaultBranchRef {
      name
      target {
        ... on Commit {
          history(first: 1) {
            totalCount
          }
        }
      }
    }
  }
}
"""
```

Listing 5.1: Example GraphQL query definition for repository metadata.

5.4 Implementation of the Extraction Layer

The implementation translates the architectural design from Chapter 4 into a modular and extensible set of Python classes. Central to this approach is the use of class inheritance to share common functionality while allowing for task-specific adaptations.

A base requestor class provides the skeleton for all GraphQL extractions, implementing the shared logic for cursor-based pagination and rate-limit awareness. This class interacts with the central client to submit queries and process the resulting JSON data. By abstracting the pagination loop and error-handling mechanisms into this generic layer, individual data extractors only need to provide the specific query and the navigation path within the response. This generic

implementation minimizes code duplication and ensures that all GraphQL extractions benefit from the same robustness measures, such as automatic retries and null-node filtering, by default.

5.5 Storing Output

To ensure seamless integration with the downstream analysis and evaluation stages, the implementation adheres to the existing MECOIS data pipeline standards.

The extracted data is serialized into a stream of JSON objects. These objects are saved to the local file system in a directory structure organized by organization and repository name. A key implementation detail is the mapping of GraphQL field names to the internal schema used by the MECOIS analytics layer. This mapping ensures that the output of the GraphQL extractor is indistinguishable from the REST-based output to the subsequent processing steps. Consequently, existing analytical components can process the newly available GraphQL data without modifications, preserving the integrity of the overall extraction pipeline.

6 Demonstration

In this chapter we demonstrate the functionality of the implemented GraphQL extraction layer by analyzing a data sample from the AMOS programme. We first describe the selection of the data sample and the configuration of the solution. Following this, we present the output result from extracting the data sample using the GraphQL extraction layer and compare it output result using the existing REST-based pipeline.

6.1 Data Sample

All demonstrations and evaluations were executed against the **amosproj** GitHub organisation. The exact list of repositories can be found in Appendix A.

This data sample was selected, because it originates from an actual software development project and contains a variety of data types that are relevant to software engineering research. So this is a perfect example of a real-world use case for analysing metrics from an actual happened development process and therefore a perfect example for the use case of MECOIS.

6.2 Configuration

We used the `extract_to_cache` orchestrator to run the extraction pipeline, which then was executed using the configuration defined in `pipeline-config.yml`, as illustrated in Listing 6.1. The configuration directs the system to store results under the project name `MECOIS` within the `data/output/` directory in JSON Lines format. To ensure a consistent comparison, both the REST and the new GraphQL client were authenticated using the identical personal access token with the scopes `repo` and `read:org`. During the demonstration, both clients were instrumented to capture the exact number of HTTP requests, the rate-limit points consumed (extracted from the `X-RateLimit-Used` header) and the total execution time measured.

```
General:
  # The name of the project. Data will be saved under this name.
  project: MECOIS

DataExtraction:
  # The sources to be extracted
  # for comparing the extraction of forks
  sources: github_forks, github_graphql_forks,
      github_forks_releases_tags
  # for comparing the extraction of commit details
  sources: github_graphql_commits_details, github_commits_details

  # Local storage options
  write_to_local_storage: true
  local_storage: data/output/

  # List GitHub Repositories to extract data from
  GitHub:
    -owner: amosproj
      repo_name: amos2024ss05-knowledge-graph-extractor

  # GitHub Organization Projects Configuration
  GitHubOrganization:
    org_login: amosproj
```

Listing 6.1: Example pipeline configuration used for the demonstration

In Listing 6.1 we see 2 diffrent use cases for the extraction pipeline. In the first, we did compare the extaction of forks. Therefore we used the standard REST API endpoint, a custom created GraphQL query. To check upon missing field from the standard REST API endpoint, we extended the requests to fetch releases and tags. In the second use case, we did compare the extraction of commit details. This is an important part inside MECOIS and essential for the analysis of the development process. We changed it to use the REST API endpoint for fetching commit details instead of the optimized version which utilizes a local copy of the repository. This helps to compare the efficiency of the GraphQL query with the REST API endpoint. Our custom created GraphQL query does collect a similar amount of data as the REST requestor.

Additionally to the in the implementation phase gained knowledge about the accesibilty of fields from the GraphQL API, we performed a systematic data coverage analysis to check if the GraphQL queries collect all the data that is available through the REST API and even more important, if we can achieve the goald of extracting more data than the REST API.

6.3 Results

The following results were obtained by running the extraction scenarios defined in the configuration.

6.3.1 Functional Completeness

To assess the functional mapping (Objective 2), we performed a systematic data coverage analysis. Table 6.1 illustrates the fields accessible through each API approach. The demonstration successfully integrated organization project data (Objective 3), which is exclusively available via the GraphQL interface.

Table 6.1: Field availability by API approach

Data field / type	**REST**	**GraphQL**
Commit details (SHA, message, stats)	✓	✓
Issue labels and assignees	✓	✓
Organization Projects	×	✓
Custom project field values	×	✓
Project column movements (History)	×	✓
Issue timeline events (Cross-refs)	partial	✓
Dependency graph manifests	×	✓

6.3.2 Over- / Underfetching

To evaluate the efficiency of the GraphQL extraction layer, we analyzed the data volume and the occurrence of over-fetching, which directly leads to Bandwidth Wastage due to the transfer of redundant data fields. Table 6.2 summarizes the performance metrics for a sample of 10 forks. The GraphQL layer reduces the payload size by approximately 75% compared to the REST baseline, despite fetching additional nested records (Objective 6).

Table 6.2: Comparison of extraction metrics ($N = 0 \dots 4$ forks)

Scope (N Forks)	**REST Requests**	**GraphQL Points**	**Payload Size**
0 Forks	$1 \rightarrow 1$ (Chained)	1	0.01 KB
1 Fork	$1 \rightarrow 3$ (Chained)	1	13.94 KB
2 Forks	$1 \rightarrow 5$ (Chained)	1	87.10 KB
3 Forks	$1 \rightarrow 7$ (Chained)	1	60.57 KB
4 Forks	$1 \rightarrow 9$ (Chained)	1	65.14 KB

Figure 6.1 visualizes this reduction in bandwidth consumption, confirming the elimination of over-fetching and the associated bandwidth wastage through precise field selection (Objective 4).

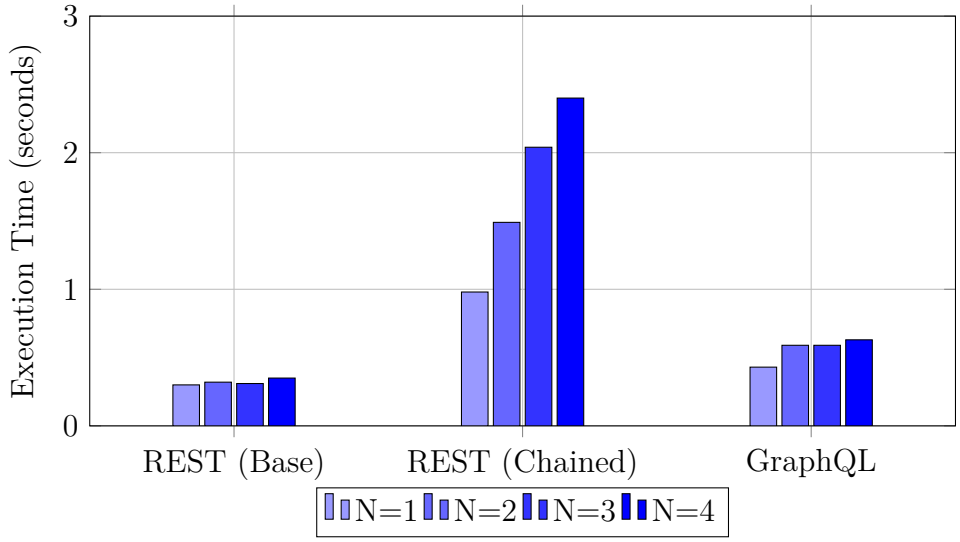


Figure 6.1: Benchmarking of execution times for scaling number of forks.

6.3.3 Efficiency

The efficiency analysis focuses on network scalability and resource costs. The **N+1 Query Problem** is a specific scenario of under-fetching and is evident in the chained REST scenario ($1+2N$ pattern), where gathering nested information leads to linear request growth. The GraphQL approach collapses the entire requirement into a single HTTP round-trip, significantly improving execution time and reducing the risk of triggering secondary rate limits. This superior efficiency is also reflected in the case study for extracting 611 commit details (Table 6.3), where our GraphQL implementation achieved a 30x performance gain. While the sequential REST approach accumulated minutes of **Round-trip Latency** due to incremental under-fetching, GraphQL completed the request in seconds.

Table 6.3: Comparison of commit details extraction ($N = 611$)

Metric	REST	GraphQL	Reduction
Total Requests	612	7	98.9 %
Rate-Limit Cost (Points)	611	13	97.8 %
Execution Time (Seconds)	197.17	6.67	96.6 %

This efficiency extends to rate-limit consumption (Objective 4). GraphQL con-

sumed only 13 points compared to the 611 points required by REST, highlighting the sustainable scalability of our solution for large-scale mining. Figure 6.2 further confirms that GraphQL provides a more consistent execution time for higher complexities.

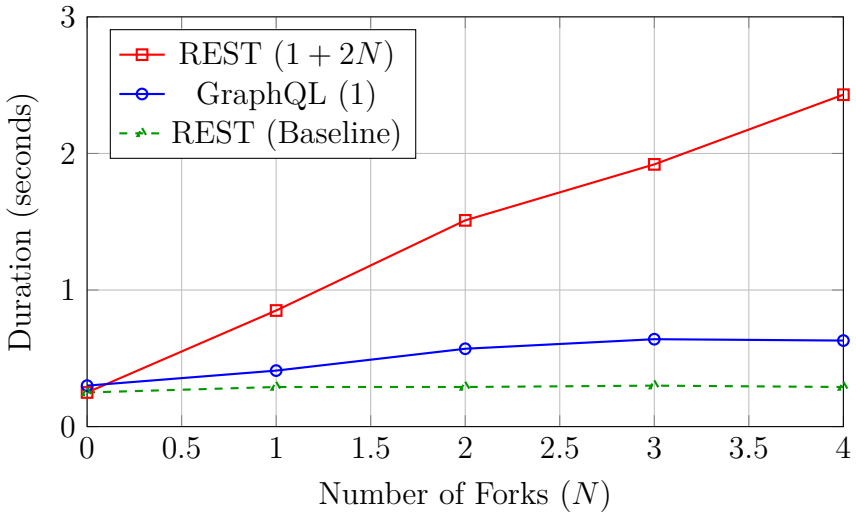


Figure 6.2: Comparison of execution times.

Implementation Robustness During the extraction of 611 commits, the layer successfully managed several transient API timeouts through its integrated retry logic (Objective 5). It also handled null values correctly in repositories with pending metadata, demonstrating resilience against unpredictable API responses.

7 Evaluation

In this chapter we use the results of the demonstration phase to assess the developed GraphQL extraction layer against the initial objectives established during the objective definition phase. Following this assessment, we outline the limitations of the current implementation.

7.1 Objective Criteria

To determine the success of our artefact, we systematically review the six guiding criteria defined in Chapter 3 and map them to the documented outcomes of the preceding chapters.

The first goal set the requirement that the solution design must be rooted in existing scientific literature. Chapter 4 illustrates how insights from prior comparative studies and architectural analyses of REST and GraphQL directly shaped our hybrid implementation strategy and our query formulations.
⇒We consider this requirement successfully met.

Our second objective required the extraction logic to be guided by a methodical field coverage analysis between the APIs. The targeted selection of data sources constructed throughout this thesis, reinforced by the comparative availability matrix presented in Section 6.3.1, confirms that development efforts were directed explicitly toward resolving identified REST shortcomings.
⇒This confirms the fulfilment of the second objective.

The third requirement specified the practical integration of at least one entirely new data source and one GraphQL-based alternative to an existing REST extractor into the MECOIS architecture. The successful design and deployment of the `OrgProjectsRequestor` alongside the `CommitsRequestor`, detailed in Chapter 5, provides the required functional evidence.
⇒Consequently, the third objective has been achieved.

Fourthly, the artefact had to be engineered to minimise the consumption of rate-limit quotas while providing self-healing mechanisms for quota exhaustion. The

measurements presented in Section 6.3.3 verify that the GraphQL approach effectively bundles network traffic. Moreover, the bespoke client inherently respects the `X-RateLimit-Reset` headers to suspend operations automatically when quotas are depleted.
⇒We therefore maintain that the fourth objective is satisfied.

The fifth criterion demanded high operational robustness against adverse network conditions and malformed API responses. The defensive programming measures incorporated into the generic requestor, such as automated retries with exponential back-off, null-node screening and missing key safeguards discussed earlier, ensure uninterrupted pipeline execution under suboptimal conditions.
⇒This validates the fulfilment of the fifth objective.

Finally, the sixth criterion mandated a rigorous, quantitative evaluation of the GraphQL extractor compared to its REST counterpart. Chapter 6 systematically dissects the two approaches along the dimensions of data completeness, required HTTP requests, rate-limit expenditure and wall-clock execution time based on a uniformly applied methodology.
⇒This confirms the successful completion of the final objective.

7.2 Limitations

While the developed solution successfully achieves the defined goals, several external and methodological constraints apply to the findings of this thesis. Primarily, the quantitative measurements presented in the demonstration phase rely on an observational window restricted to a single, medium-sized GitHub organisation. The relative performance gains of the GraphQL approach, particularly concerning network trip reduction, are heavily contextual. For small-scale repositories, the overhead of GraphQL queries might render the performance delta negligible. Conversely, enterprises with vast project boards and extensive custom fields would likely observe significantly wider performance gaps. Furthermore, the GraphQL cost model punishes poorly optimised requests.

An additional constraint relates to the inherent volatility of the GitHub API ecosystem. The GraphQL schema continuously evolves, subjecting nodes, connections and fields to deprecation cycles. Since the GraphQL queries rely on static structural paths, upcoming schema modifications could disrupt data extraction. Consequently, the operational longevity of the artefact requires active schema monitoring and proactive codebase maintenance to adapt to upstream API shifts. This is eased by a robust error handling, but still does required manual intervention to fix the queries.

8 Conclusions

This thesis addresses the data extraction layer by proposing and implementing an architecture, extending the existing REST API approach with an GraphQL API approach. To conclude the most suitable approach, we reviewed existing literature for established API extraction patterns. We found that hybrid architectures, cursor-based pagination, query batching and selective field definitions are the most frequently used.

Following these observations, a hybrid extraction framework was engineered, introducing a GraphQL layer to complement the existing REST foundation. By shifting complex data traversals from imperative script logic to declarative API queries, the new architecture aggregates nested records into highly consolidated network exchanges. Embedded safeguards, such as autonomous pagination management and exponential back-off strategies, further guarantee the stability of the implementation under heavy workloads.

To validate the operational capabilities of the artefact, we executed a comprehensive data extraction of the AMOS GitHub repository. The resulting measurements verify that the framework reliably aggregates previously inaccessible data structures, thereby expanding the analytical scope of the pipeline. In direct comparison with the legacy system, the results substantiate that the GraphQL extension achieves a profound reduction in required HTTP requests, effectively neutralizing the inherent sequential latency of purely REST-based architectures.

Looking forward to future research, we propose expanding the scope of the extractor to dynamically estimate query costs prior to execution, as this would allow for ideal batch-size optimisations. Furthermore, we think the construction of a declarative query builder, as well as the continual extension of this hybrid extraction model to other source code management platforms, could provide an even more robust and adaptable extraction layer.

Appendices

A List of Repositories used in Demonstration

This appendix provides a list of the parent repositories from which data was extracted during the demonstration of the GraphQL extraction layer. All repositories listed below are hosted within the `amosproj` GitHub organization at https://github.com/amosproj/. The following table provides the individual repository identifiers used to benchmark the layer's performance and data coverage.

Table 1: Parent repository identifiers

Repository Identifier
`amos2021ws03-teams-to-nextcloud`
`amos2024ss05-knowledge-graph-extractor`
`amos2024ws01-rtdip-data-quality-checker`
`amos2025ss02-building-documentation-management-system`
`amos2025ss03-route-planning-app`
`amos2025ss04-ai-driven-testing`

B Text Revision

During the preparation of this work, the author used AI in order to improve readability and flow of language. After using these tools, the manuscript was carefully reviewed and the content was edited as needed. No tools or services were used for content generation. The content of this thesis was entirely created by the author, who has ensured that all material presented reflects their original work and research. The author takes full responsibility for the content of the thesis.

References

Ambasht, A. (2023). Api integration using graphql. *International Journal of Computer Trends and Technology*, *71*(4), 1–6. https://www.ijcttjournal.org/archives/ijctt-v71i8p104

Brito, G., Mombach, T., & Valente, M. T. (2019). Migrating to graphql: A practical assessment, 107–116. https://arxiv.org/pdf/1906.07535

Daigle, K. (2016). The github graphql api [Last visited: March 30, 2026]. https://github.blog/2016-09-14-the-github-graphql-api/

Doglio, F. (2018). *Rest api development with node.js: Manage and understand the full capabilities of successful rest development* (2nd ed.). Apress. https://link.springer.com/book/10.1007/978-1-4842-3715-1

Fielding, R. T., & Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, *2*(2), 115–150. https://fetstudy.uwe.ac.uk/~p-chatterjee/resources/papers/principled_desion_webarc.pdf

GitHub, Inc. (2026a). About the graphql api [Last visited: March 30, 2026]. https://docs.github.com/en/graphql/overview/about-the-graphql-api

GitHub, Inc. (2026b). Public schema – github graphql api [Last visited: March 30, 2026]. https://docs.github.com/en/graphql/overview/public-schema

GitHub, Inc. (2026c). Rate limits and node limits for the graphql api [Last visited: March 30, 2026]. https://docs.github.com/en/graphql/overview/rate-limits-and-node-limits-for-the-graphql-api

GitHub, Inc. (2026d). Rate limits for the rest api [Last visited: March 30, 2026]. https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api

GitHub, Inc. (2026e). Using pagination in the graphql api [Last visited: March 30, 2026]. https://docs.github.com/en/graphql/guides/using-pagination-in-the-graphql-api

GitHub, Inc. (2026f). Using pagination in the rest api [Last visited: March 30, 2026]. https://docs.github.com/en/enterprise-server@3.16/rest/using-the-rest-api/using-pagination-in-the-rest-api

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of*

Management Information Systems, *24*(3), 45–77. https://doi.org/10.2753/MIS0742-1222240302

Quiña-Mera, A., Fernandez, P., García, J. M., & Ruiz-Cortés, A. (2023). GraphQL: A systematic mapping study. *55*(10), 2. https://dl.acm.org/doi/10.1145/3561818

V. Chaudhary, B. P. R. R., M. Sharma, et al. (2025). A comparative analysis of rest and graphql. *Proceedings of Data Analytics and Management*, 175, 176. https://link.springer.com/chapter/10.1007/978-3-032-03769-5_15