Continuous Performance Benchmarking and Optimization of an ETL Data Pipeline

MASTER THESIS

Lennart Heimbs

Submitted on 04.05.2026

Friedrich-Alexander-Universität Erlangen-Nürnberg

Faculty of Engineering, Department Computer Science

Professorship for Open Source Software

Supervisor:
Julian Hirsch, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 04.05.2026

License

Erlangen, 04.05.2026

Abstract

Software engineering teams rely on development data for productivity measurement, transfer pricing, and impact assessment. Tools for this analysis typically extract the raw data from platforms like GitHub, GitLab, or Jira. The MECOIS project's ETL pipeline handles the analysis of the data and follows the medallion architecture with Bronze, Silver, and Gold layers. Over time, the pipeline has grown without performance instrumentation, leaving regressions and inefficiencies unattributable to specific changes. To close the gap, this thesis introduces a benchmarking framework and demonstrates it by optimizing an identified bottleneck. The framework uses three measurement tiers of varying granularity to capture pipeline performance at different scopes. Eight benchmarks built on these tiers record each measurement against a stored baseline, attributing performance changes to specific code revisions. Alongside the framework, the Bronze-to-Silver transition's identity resolution adopts GraphQL-based alias batching and Spark partition-level execution. This speeds up the transition by 2.93× and reduces external SortingHat requests by 99%. Across the full pipeline, this yields a 1.07× end-to-end speedup, with the framework's measurements identifying extraction, rather than identity resolution, as the dominant runtime cost. Beyond the demonstration, the framework runs in the project's CI to detect regressions before they reach production.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ACID	Atomicity, consistency, isolation, and durability
API	Application programming interface
CI	Continuous integration
CI/CD	Continuous integration/continuous deployment
CLI	Command-line interface
CV	Coefficient of variation
DORA	Deployment frequency, lead time for changes, change failure rate, and time to restore
ETL	Extract, transform, load
HTTP	Hypertext transfer protocol
JSON	JavaScript object notation
JSONL	JSON lines
JVM	Java virtual machine
JWT	JSON web token
LRU	Least recently used
MoSCoW	Must-Have, Should-Have, Could-Have, Won't-Have this time
NTP	Network time protocol
PR	Pull request
RDD	Resilient distributed dataset
REST	Representational state transfer
UDF	User-defined function
URL	Uniform resource locator
UUID	Universally unique identifier
WP	Work package

1 Introduction

Modern software engineering organizations increasingly rely on development data to make business decisions. Productivity assessment, transfer-pricing calculation across tax boundaries, and revenue-impact attribution all build on evidence extracted from digital traces of work (Hirsch & Riehle, 2022). Each of these applications faces a common reliability problem. Single metrics are unreliable in isolation, so any defensible analysis must combine multiple metrics across multiple sources (Forsgren et al., 2021). MECOIS is a project that delivers this engineering intelligence by extracting and analyzing engineering, organizational, and accounting data from sources like GitHub, GitLab, and Jira. This engineering thesis focuses on the pipeline's performance (Professorship for Open-Source Software, 2026).

The technical base of MECOIS is a Python-based extract, transform, load (ETL) pipeline used for data extraction and analysis, with a web frontend based on TypeScript and React for visualization and management of the pipeline. The pipeline is modeled after the medallion architecture described by Strengholt (2025). It uses three layers for extracted data, starting with Bronze for the raw data, followed by cleaned data in the Silver layer, and finally business-ready analytical data in the Gold layer. Using this architecture and its associated modular separation allows developers to rapidly add new sources, pipeline steps, and analytics. However, the quick iteration cycles and organic growth did not prioritize performance. When developers change the pipeline, the effects cannot be quantified without measurement. Therefore, regressions accumulate silently and improvements cannot be validated.

To address the gap, this thesis introduces a benchmarking framework with three distinct measurement tiers. It gives developers better tools to surface bottlenecks by measuring the performance at multiple levels, from individual components up to complete end-to-end runs of the pipeline. The benchmarks contact live service instances to capture measurements under real production conditions. Each result is then recorded against a stored baseline, so subsequent runs can be compared and both regressions and improvements quantified. Integrating the framework into the existing continuous integration (CI) workflow furthermore automates the detection of performance bottlenecks and highlights changes that introduce new bottlenecks or cause regressions in performance.

Beyond the framework, this thesis demonstrates its application by optimizing one of the candidate bottlenecks identified by a system analysis. Several candidates exist, including complete re-extraction of sources on every pipeline run and

resolution of developer identities across platforms. However, re-extraction requires pipeline-wide changes, whereas identity resolution can be confined to a single step. Identity resolution therefore serves as the demonstration case. It uses the external identity management service SortingHat to solve the identity fragmentation present when a given developer uses different identifiers across platforms, such as a GitHub username, a GitLab email, and a Jira login. SortingHat collects such identities and merges them into a single unified profile, which is then used for data analysis. The current implementation issues multiple application programming interface (API) requests for each identity record in a dataset, resulting in high numbers of requests. Using batching on top of the existing caching mechanism significantly reduces the incurred call volume, and with it the latency in this specific pipeline step. Finally, the optimization is evaluated using the framework, demonstrating its utility.

This thesis makes two contributions to the MECOIS project:

1. A performance benchmarking framework with three measurement tiers, integrated into the CI workflow for automated regression detection.
2. A batched GraphQL identity resolution implementation that demonstrates the framework by reducing call volume to the identity management service.

The content of this thesis is organized as follows: Section 2 covers the background and related work. Section 3 analyzes the MECOIS system, identifies candidate bottlenecks, and establishes the theoretical foundations. This is followed by Section 4, which states the requirements. Section 5 presents the architecture and design, Section 6 the implementation, and Section 7 the evaluation. Finally, Section 8 presents the conclusions and outlines directions for future work.

2 Background and Related Work

This chapter establishes the technical foundation that MECOIS builds on and surveys related systems and tools in the field.

2.1 Background

MECOIS extracts data from third-party collaboration platforms and refines it through a layered pipeline before any analytics take place. The next sections cover the pipeline pattern that organizes this refinement, the analytics domain in which MECOIS operates, and the API paradigms by which the source platforms expose their data.

2.1.1 Data Pipeline Architecture

A data lake architecture is a common basis for modern data analytics pipelines. The medallion architecture models the data flow by defining a layered structure. Each layer progressively refines the data as it moves through the pipeline. The design of the medallion architecture and the supporting storage and processing components are described in the following sections.

Medallion Architecture. Strengholt (2025, p. 50) describes the medallion architecture as organized into three progressively refined layers. These are named after the medallions often awarded in sporting competitions: Bronze, Silver, and Gold.

The Bronze layer is the lowest tier. It captures raw data exactly as received from source systems and stores it as-is. New data is always appended to existing data, making it the single source of truth for the pipeline.

The next layer is the Silver layer, in which the data gets refined through quality checks, standardization, deduplication, and other enhancements. This produces structured, query-friendly, and reliable data that is appropriate for analytical use.

Finally, the Gold layer delivers the business-ready output. Its data is optimized for specific reporting and decision-making needs by being highly curated and tuned for fast access to key metrics.

Data Lake Technologies. The medallion architecture requires a data lake architecture at its base. According to Strengholt (2025, p. 16), “data lakes emerged as a solution to the shortcomings of traditional data warehouses”, which were tightly integrated analytics solutions based on database systems. In contrast to a traditional database, a data lake uses a distributed architecture that manages vast amounts of data across many machines and drops the structural requirements

that database systems impose. It supports unstructured, semi-structured, and structured data at the same time and thus expands the usability of the data. However, a data lake on its own is passive storage. A distributed processing engine is needed to read, transform, and write data across its layers efficiently.

Apache Spark (Zaharia et al., 2010) is an open-source distributed processing engine that offers in-memory large-scale data processing with SQL query syntax and support for various programming languages like Python with PySpark (Zaharia et al., 2016). Spark follows the lazy query evaluation paradigm, as described by Armbrust et al. (2015, p. 1385). It defers acting on a query until a trigger is encountered that requires action, like saving the DataFrame to disk. This allows its query planner to optimize the full transformation chain before any data is read. To ease the development using it, Spark is designed to scale from a single machine to a full cluster without requiring code changes. This makes it well-suited as the processing layer for a medallion pipeline regardless of deployment size.

Atomicity, consistency, isolation, and durability (ACID) operations, present in most database systems, are a desired feature when working in a data lake. In database systems they guarantee that transactions are processed reliably and remain valid even in the event of errors, crashes, or power failures. To gain this functionality when working with data lakes, Armbrust et al. (2020) introduced the open-source storage layer Delta Lake. Delta Lake achieves the ACID guarantees by maintaining an ordered, immutable transaction log that serves as the single source of truth for all table operations. Beyond the basic transactional guarantees offered by Delta Lake, there are other additional features relevant to a medallion pipeline. Table versioning and time travel allow any previous state of a table to be queried or restored. Schema enforcement prevents writes that violate the defined table schema from occurring silently and corrupting data. Merge operations enable upsert semantics, which are essential for incrementally updating Silver and Gold layer tables.

2.1.2 Software Engineering Analytics

Modern software development leaves a rich trail of artifacts across platforms such as version control systems, issue trackers, and continuous integration/continuous deployment (CI/CD) pipelines. GitHub, Jira, and GitLab represent the top three code documentation and collaboration tools according to the 2025 Stack Overflow Developer Survey (Stack Overflow, 2025) and are primary targets for software engineering analytics. GitHub and GitLab offer metrics like commits, pull requests, issues, and CI/CD pipeline runs for analysis. Jira, as a collaboration platform, primarily offers project tickets and workflow transitions. None of these platforms, however, offer direct database access, so all data is retrieved through their respective APIs. This makes pagination handling and rate limit awareness inherent constraints of the extraction process.

While working across these platforms, a developer may go by a GitHub username, a GitLab email address, and a Jira login with no shared identifier between them. Without a way to link these, commits and issues from the same person appear as unrelated records. SortingHat (Dueñas et al., 2021) is an open-source identity management service that solves this issue. It maintains a database of profiles, each associating a set of platform-specific identifiers with one individual. Callers submit identifiers through a GraphQL API and SortingHat returns the matched or newly created profile.

2.1.3 API Design Paradigms

Software analytics projects that use data from public platforms like GitHub and GitLab depend on the APIs they expose. Typically access is granted via either representational state transfer (REST) or GraphQL API endpoints.

REST APIs identify resources by uniform resource locators (URLs) and use standard hypertext transfer protocol (HTTP) verbs like *GET*, *POST*, *PUT*, and *DELETE* to define the desired operation on a resource. REST is a stateless architectural style where each request carries all necessary context. The server determines what fields are included in the response and typically paginates large collections. REST APIs typically limit user access via rate limits per token or IP address, which restricts how fast a system can extract large amounts of data.

GraphQL is a query language for APIs developed by Facebook and publicly released as an open specification in 2015 (GraphQL Foundation, 2021). In contrast to REST, all operations are directed to a single endpoint and the client declares exactly which fields it needs. This avoids over- or under-fetching of data. It also uses a strongly typed schema that defines all available types and their relationships. This enables it to use the static type information to validate queries before executing them. A notable feature is query aliasing, which allows multiple independent operations to be batched into a single HTTP request by assigning each an alias (GraphQL Foundation, 2021, § 2.7). The server then resolves them and returns all results in one response.

2.2 Related Work

The work covered in this thesis addresses problems that are not unique to MECOIS. Pipeline performance measurement is relevant to any multi-source software analytics system. API-intensive identity resolution similarly arises whenever large amounts of multi-source data are analyzed. This section examines existing systems and tools that approach the same problems, and identifies the gaps that motivate the specific contributions of this thesis.

2.2.1 Software Engineering Analytics Platforms

Several open tools address the problem of collecting and analyzing developer activity data from software forges at scale. GrimoireLab (Dueñas et al., 2021) is the most architecturally comparable system to the MECOIS project. It is a modular Python toolset that retrieves developer activity from over 30 source types and processes it through a pipeline of four main components. The components are Perceval for data retrieval, GrimoireELK for enrichment, SortingHat for contributor identity management, and Kibiter for dashboard visualization. GrimoireLab uses a two-stage storage model, with immutable raw JavaScript object notation (JSON) documents as the first stage followed by enriched flat documents that are optimized for querying. This design parallels the Bronze and Silver layers in the medallion architecture of MECOIS. However, GrimoireLab targets research and open-source community analytics rather than enterprise software engineering analytics. It is not designed for performance-critical pipeline integration, as it exposes no benchmarking infrastructure and provides no pipeline performance measurement.

Apache DevLake (Apache Software Foundation, 2023a) takes a different architectural approach. Written in Go with a relational database backend (Apache Software Foundation, 2023b), it organizes ingested data into three layers. At the base are raw API responses, followed by tool-specific relational tables in the middle. A normalized domain layer that unifies GitHub pull requests and GitLab merge requests under a shared schema (Apache Software Foundation, 2023c) represents the top layer. Its plugin architecture enables independent extension with new data sources. In contrast to MECOIS, DevLake includes no identity resolution component. Cross-tool contributor linkage is instead handled through configuration-time mapping. Its primary analytical focus is deployment frequency, lead time for changes, change failure rate, and time to restore (DORA) metrics for engineering management, whereas MECOIS computes fine-grained engineering metrics at the commit and pull request level.

Augur (S. P. Goggins et al., 2021), developed at the University of Missouri as part of the CHAOSS project (S. Goggins et al., 2021), is designed for large-scale open-source health research. It uses a distributed worker model with Celery task queues backed by Redis to parallelize data collection across tens of thousands of repositories. Results are stored in PostgreSQL and CHAOSS-standardized health and sustainability metrics are exposed through a REST API. This distributed architecture contrasts with the layered pipeline of MECOIS, which is optimized for depth of analysis on a configured set of enterprise repositories rather than breadth across a large research corpus. As with DevLake, Augur includes no dedicated identity resolution component.

None of these platforms include infrastructure for measuring or benchmarking the performance of their own extraction pipeline, nor do they address the throughput

limitations of sequential API calls to identity resolution services. These are the two gaps addressed in this thesis.

2.2.2 Data Pipeline Benchmarking

Performance benchmarking for ETL systems has been approached from several directions. The TPC-DI benchmark (Poess et al., 2014) is the only data-integration standard from the Transaction Processing Performance Council, modeling an ETL load from heterogeneous file sources into a data warehouse and measuring throughput as successfully integrated data volume per unit time. It is designed for periodic certification-style comparison of commercial ETL tools against a fixed synthetic workload. A fixed workload, however, limits how it can be used for testing a custom pipeline. TPC-DI cannot be adapted to measure specific stages of a medallion architecture, and provides neither CI integration nor automated regression detection.

Simitsis et al. (2009) propose a measurement methodology for ETL workflows at two levels. The first level is called the micro level. It includes the evaluation of each individual activity within the pipeline. Each evaluation includes a taxonomy classification based on blocking behavior and functionality. At the macro level, the methodology describes measuring workflow topology patterns that model realistic pipeline structure. This two-level approach is the closest conceptual precedent for the three-tier measurement hierarchy developed in this thesis (Section 3.3). However, the work is a framework specification rather than a tool and provides no implementation. Further, it does not offer any CI integration considerations and the reference workloads are defined over the TPC-H synthetic schema.

HiBench (Huang et al., 2010) and SparkBench (Li et al., 2015) are general big data benchmark suites, targeting MapReduce workloads and Spark cluster performance respectively. While they cover benchmarking the throughput of an entire job, they have no concept of custom pipeline steps for business logic, medallion-layer transitions, or individual Delta Lake operations.

At the Python tooling level, *pytest-benchmark* (Ciuraru, 2014) provides a test fixture that executes arbitrary callables in calibrated repeated rounds and reports statistical timing results. Combined with CI integration tooling, it supports automated regression detection with configurable thresholds. However, *pytest-benchmark* is designed for single-process Python micro-benchmarks. It cannot instrument distributed Spark executor processes and does not model multi-minute pipeline runs spanning Java virtual machine (JVM) workers and Delta Lake I/O. It also carries no understanding of medallion pipeline semantics.

No existing tool simultaneously addresses custom ETL pipeline measurement, multi-tier granularity from individual service calls to fully orchestrated pipeline

runs, and CI-integrated regression detection for Spark workloads. This gap motivates the benchmarking framework developed in this thesis.

2.2.3 GraphQL Request Batching

The GraphQL aliasing mechanism introduced in Section 2.1.3 has a direct application in API-intensive pipelines. Consolidating multiple operations into a single HTTP request avoids repeated individual requests but results in larger payloads. Vogel et al. (2018) show this consolidation in practice when migrating REST services to GraphQL. There, multiple individual endpoint calls for related resources could be replaced with a single GraphQL query.

Field aliases, defined in GraphQL Foundation (2021, § 2.7), allow multiple independent operations to be included in a single GraphQL document under distinct response keys. As opposed to the consolidation described above, this allows combining multiple individual GraphQL requests into a single HTTP call. Furthermore, GraphQL Foundation (2021, § 6.3.1) guarantees that aliased mutations execute serially and predictably, so multiple mutations targeting the same object behave the same as multiple individual requests. When this mechanism is used, N independent mutations can be submitted in a single HTTP request, which reduces round-trips from N to one.

Alias-based batching within GraphQL documents forms the basis of the identity resolution optimization outlined in Section 3.2.5. While aliasing is well defined in the GraphQL specification, its use as a batching mechanism to optimize throughput in API-intensive pipelines has received little dedicated study in the literature surveyed here, and is the gap addressed in Section 5.

3 Problem Analysis and Theoretical Foundations

This chapter builds on the gaps that were identified in the previous chapter. It establishes concrete problems and theoretical foundations for the requirements in Section 4 and the design in Section 5. It starts with a description of the current design of the pipeline, followed by identifying the performance costs and outlining the two work packages addressed in this thesis. The remaining sections develop the measurement theory for the benchmarking framework and establish the theoretical foundation for identity resolution optimization.

3.1 Pipeline Architecture

The next subsections describe how MECOIS realizes the medallion pattern in practice. They begin with the overall architecture and then work through the layered data flow, the source extraction, and the gold-layer analytics.

3.1.1 Current Architecture

As described in Section 2.1.1, MECOIS implements the medallion architecture over a local Delta Lake, and uses PySpark as the transformation engine. This serves as a separation of concerns that confines transformation logic to discrete stages. Additionally, the immutable Bronze data allowing recomputation of values facilitates fast iterations on the pipeline.

The local development entry point for the system is the `execute.py` Python script. It dynamically loads a Python module called the orchestrator that composes and runs one or more pipelines. The components of the pipeline are organized into four functional layers. At the top, orchestrators are entry points that compose and sequence pipelines based on a configuration-driven source map. In the next layer, pipelines assemble the actual step sequences for each data source and medallion layer transition. Within each pipeline, individual pipeline steps implement three operations: extract, transform and write. Each of these steps is realized through a hierarchy of abstract base classes. These are constructed within a queue that chains the execution by passing data from one step through the next. Upon completion, each step signals the pipeline to invoke the next step in the chain, forwarding its output as input. This pattern follows a linked-handler structure similar to the Chain of Responsibility pattern (Gamma et al., 1994, p. 223) and allows for declarative pipeline definitions. As a consequence, all steps within a

single orchestrator run execute sequentially. Finally, infrastructure components for Spark session management and Delta Lake persistence provide the shared technical substrate used across all steps.

Figure 3.1 shows the three distinct pipeline steps. A requestor step is always first: it retrieves data from an external source, such as a REST API, a Delta Lake table, a file, or an in-memory object and forwards it into the chain. One or more transform steps follow, each applying a discrete operation such as schema normalization or identity resolution. A writer step closes the chain by persisting the transformed data typically to a Delta Lake table.

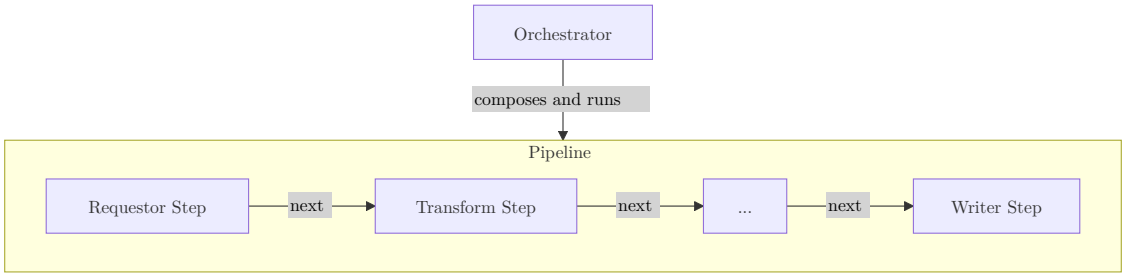

Figure 3.1: Pipeline execution structure with orchestrator, requestor, transform steps, and writer.

3.1.2 Data Flow Through Layers

The three medallion layers each hold data in a progressively more refined state, as illustrated in Figure 3.2.

At the Bronze layer, records arrive as raw API responses. They are persisted to the storage unmodified and append-only and are keyed by a SHA-based hash to prevent duplication across runs. This layer serves as an immutable audit trail and allows any values downstream to be recomputed from scratch.

The transition of data from one layer into the next is handled by transformer pipeline steps. Out of these steps, transitioning Bronze data to the Silver layer is important in the ETL pipeline because it creates the baseline data for future Gold transformations. Within this transformation, it enriches the raw Bronze data by applying column-level schema mappings first, followed by normalizing the types present in the data. Finally, it uses the external SortingHat service to resolve developer identities across platforms. During the identity resolution, it maps the same person's different usernames and email addresses used on GitHub, for example, to a single canonical identity. The resulting records are merged into Silver Delta tables using ACID upsert semantics. It produces deduplicated and schema-enforced datasets that are easily queryable.

Gold layer analytics modules read from the Silver tables and apply business logic. They aggregate development data into engineering metrics like cost estimates or

worktime measurements. The final results are written to Gold Delta tables, from which dashboards and reports are served.

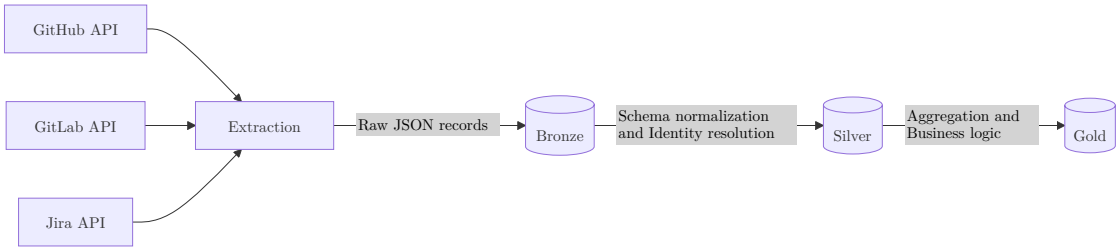

Figure 3.2: Data flow through the MECOIS medallion layers.

3.1.3 Development Data Extraction

The development data orchestrator is the primary pipeline in the MECOIS project for data extraction from available sources. Its purpose is to take developer activity data from all configured repositories across the sources and make it available in a clean form in the Silver layer. Because it can cover a wide range of sources, it deserves closer inspection of its performance characteristics and serves as the workload for the full-pipeline benchmark developed in this thesis.

Where data is extracted from is configurable and covers three platform categories. GitHub sources include commits, pull requests, events, issues, CI/CD jobs and pipeline runs, releases, and software bill of materials (SBOM) data. GitLab sources cover equivalent data: commit details, merge requests, events, issues, jobs, pipeline runs, and releases. Jira sources provide issue records and their change history.

For each source, an extraction run proceeds in three phases. The first phase calls the REST or GraphQL API of the given source-specific extraction pipeline. It extracts the data from all configured repositories and automatically handles the pagination of the data. All raw records are then collected into a Python list and passed to the next phase. In the second phase, the pipeline converts that list into a Spark DataFrame and applies a uniform Delta Lake schema. The resulting DataFrame is then merged into the Bronze table using SHA-based deduplication to prevent duplicate records across runs. The last step transforms the data into its final Silver layer form, by reading the Bronze table and applying a fixed sequence of transformation steps, illustrated in Figure 3.3. The first of these transformation steps flattens any nested struct columns from the raw API response into a flat column namespace. Next, irrelevant columns are pruned according to a per-source schema configuration file. Date and timestamp fields are then normalized from strings to timestamp types, and the timezone information is extracted into separate columns. Additionally, developer identities are resolved by calling the external SortingHat service, which maps contributor identities to a unique canonical identifier. Finally, a SHA-256 universally unique identifier (UUID) is

derived from the record's configured unique fields and appended as the primary key, after which the result is merged into the Silver Delta table.

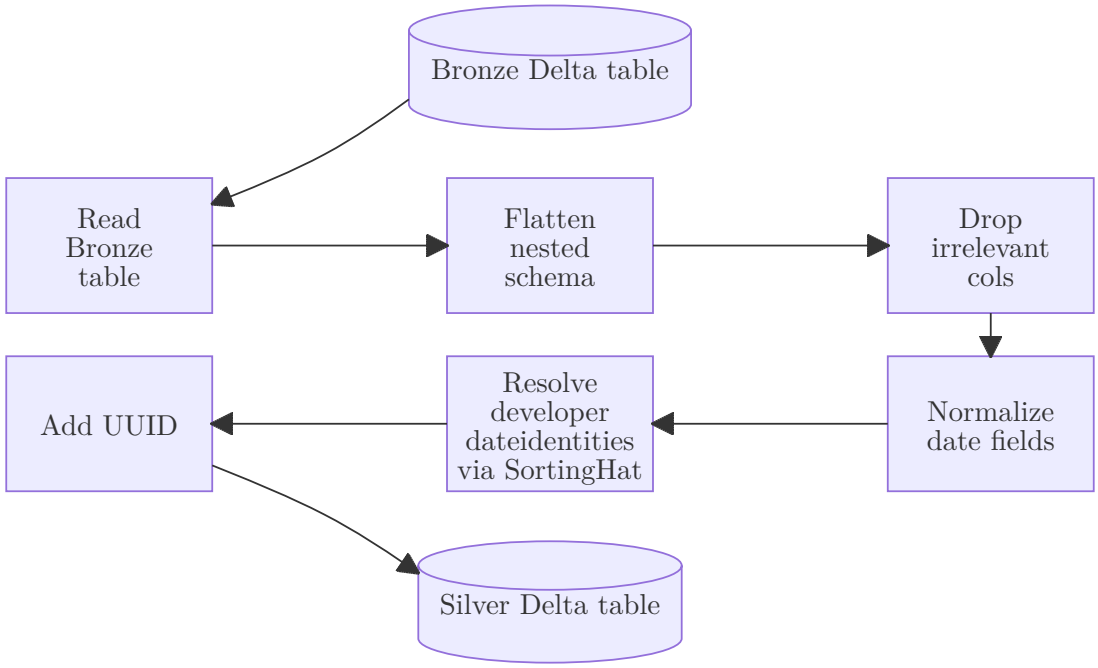

Figure 3.3: Bronze-to-Silver transformation steps applied per source in the development data orchestrator.

3.1.4 Gold Layer Analytics

The value MECOIS provides to its users comes from the extracted engineering analytics that use the Silver layer as their source data. Several metrics have been established in research and are implemented as Gold-layer analytics modules, each targeting a distinct metric.

Buchner & Riehle (2022) is implemented in the cost calculation module. It estimates the working time per commit and aggregates it per author and repository to produce a cost-share distribution. The worktime measurement module builds upon that work and adds non-coding overhead metrics to the commit-based worktime estimation. The inner source prediction module adapts the approach of Capraro et al. (2018) and Capraro (2020). It measures inner-source behavior by classifying contributors and computing engagement metrics over time. The code change summarization and marketplace data modules add supporting information based on the extracted engineering data.

Some modules depend entirely on Silver tables already populated. Others include their own extraction pipelines and execute the full Bronze-to-Silver chain for specific sources before computing Gold output.

3.2 Performance Characteristics

This section identifies where performance costs are incurred in the pipeline architecture above.

3.2.1 Absence of Measurement Tooling

Measurement is a basic concern for quantifying the performance of the pipeline. Yet the MECOIS pipeline currently records no runtime information. Performance is neither measured nor compared to a baseline, which means there is no reference point for comparisons across runs. Changes that degrade performance only surface as slowdowns when observed by the user, which makes attributing the slowdown to a specific change difficult.

3.2.2 Developer Identity Resolution

Apart from the missing instrumentation, the identity resolution step mentioned in Section 3.1.3 warrants a closer look. This is because it is the most structurally distinctive part of the Bronze-to-Silver transformation. The flattening and pruning of columns, time normalization and unique identifier generation are all based on PySpark mechanisms. Identity resolution stands out as the exception since it depends on the external identity management service SortingHat.

To understand the implications of the identity resolution mechanism, SortingHat's management of identities has to be covered first. SortingHat maintains a relational database of identities and unique individuals. Each identity record stores a name, email address, username, and source platform and gets assigned a distinct UUID. Multiple identity records belonging to the same real person are grouped under a single unique identity, carrying a stable UUID that is referred to as the main key and can be different from the UUID of an individual identity record. This grouping of new identity data into an existing identity is referred to as a merge. SortingHat's merging strategy is deliberately conservative. It only merges identities when there is high confidence they refer to the same individual. This is because erroneous merges cause more problems in practice than missed ones (Dueñas et al., 2021, p. 14). Where automatic merging is insufficient, manual curation can be performed via SortingHat's companion web interface, HatStall.

Within the pipeline, the Bronze-to-Silver transformer adds the identities present in a DataFrame to SortingHat. This operation is done for each row in the Bronze DataFrame and for every configured identity field within that row, as illustrated on the left side in Figure 3.4. SortingHat then checks each received identity and registers it to its database. For each new identity it returns a fresh UUID that serves as its canonical identifier. If the identity is already known to SortingHat, it returns the UUID of the matching identity record. This UUID may differ from

the canonical UUID of the individual when a merge has grouped the identity under another record. In this case the transformer queries SortingHat a second time to retrieve the canonical UUID. The canonical UUID then replaces the raw identity fields in the Silver record. As a result, all Silver records across all sources that refer to the same person share a single stable identifier. This makes it possible to correctly attribute contributions from that person throughout Gold layer analytics.

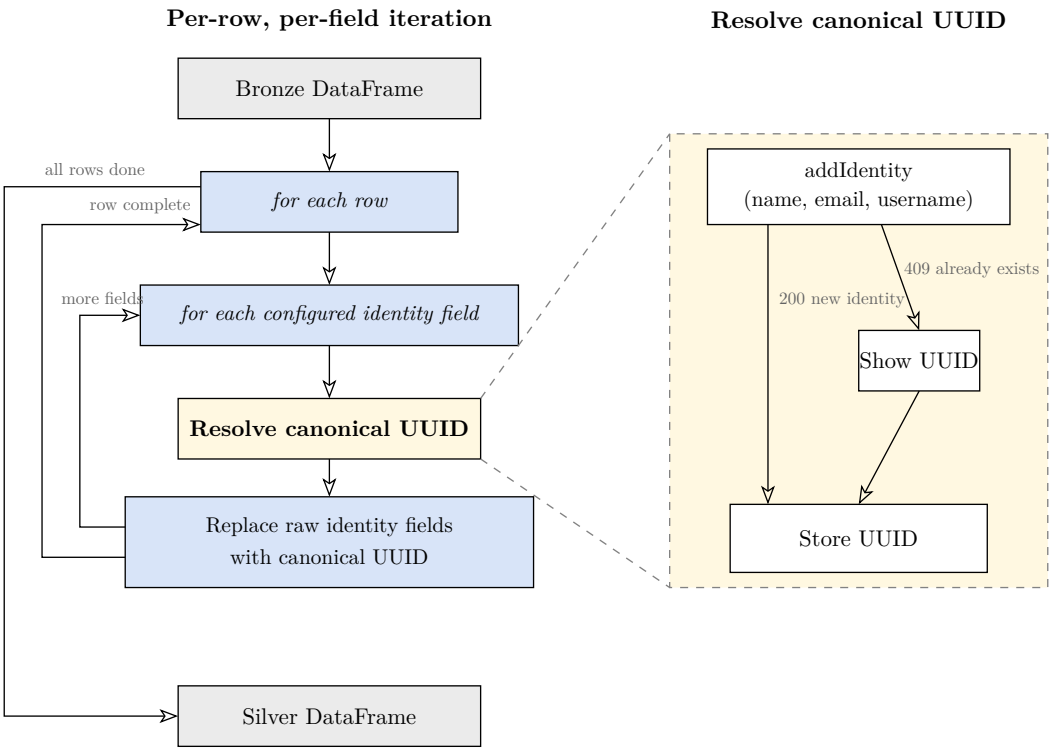

Figure 3.4: Per-row identity resolution in the Bronze-to-Silver transformer.

The two operations described above define the two code paths of the SortingHat requestor that are referenced throughout the rest of this thesis. Adding a new identity to SortingHat is referred to in this thesis as the add path, and resolving the canonical UUID of an existing identity is called the show path. The two paths are shown on the right side of Figure 3.4, with the show path only invoked when the identity already existed in SortingHat. To reduce the cost of this per-row resolution, the SortingHat requestor applies two caching layers. All individual resolution methods that carry identity data and return canonical UUIDs are decorated with an least recently used (LRU) cache. This is a caching mechanism integrated into Python, which maintains a fixed-size LRU eviction cache of the N most recent unique call results. In the MECOIS project, N is set to 1,024 for each identity resolution method. When an identity resolution is called a second time with the same arguments, the cache returns the result of the first invocation. To accompany this method-level cache, a class dictionary is checked and populated each time an

identity is added to SortingHat. This second caching layer contains every result for the lifetime of the Python process. However, the remaining three methods used for querying canonical UUIDs and identity profiles do not use this dictionary. Their only cache is the fixed-size LRU cache, which creates an asymmetry when the number of distinct identities exceeds the LRU cache's capacity. When it is exceeded, repeated calls to add a new identity still hit the dictionary without issuing requests. However, repeated calls to the methods without this dictionary cache fall through the LRU and produce HTTP requests on every invocation. At workloads that exceed the LRU cache capacity, adding an identity remains cached, while the other identity operations lose their cache coverage entirely.

After completing the identity resolution step, the DataFrame is repartitioned to a single Spark partition before it is passed on to the next step. This causes the remainder of the Bronze-to-Silver stage to run without the parallelism given when using multiple partitions. This is revisited in Section 5 as part of the identity resolution optimization.

3.2.3 Gold-Level SortingHat Usage

The sequential call pattern shown in Figure 3.4 is not limited to the Bronze-to-Silver stage. The code change summarization analytics module also calls SortingHat during its Gold-level transformation. It has to resolve an author name to a set of canonical UUIDs in order to filter its input DataFrame and retrieve author profiles for formatting. These calls are synchronous and sequential, following the same pattern identified in Section 3.2.2. Analytics pipelines that use SortingHat are therefore in scope for performance measurement.

3.2.4 API Extraction

Besides the identity resolution, API extraction from GitHub and GitLab is another performance cost area. During the data extraction, and in the development data orchestrator as an example, each source is processed sequentially and the extractor paginates through the full history of every configured repository. As a consequence, any bottleneck within the processing chain adds directly to the total extraction duration. Additionally, within a single extraction run, a project with multiple repositories and all sources enabled can produce several thousand API requests. This can cause slowdowns, because GitHub and GitLab enforce per-token rate limits for the use of their APIs. GitHub allows 5,000 requests per hour for its REST API and a separate budget of 5,000 points per hour for its GraphQL API, both scoped to the authenticated user (GitHub, 2024a; 2024b). GitLab allows 2,000 authenticated requests per minute by default (GitLab, 2024). MECOIS extraction is predominantly REST-based, with only the pull request extractor using GraphQL APIs. Since all extractors share one authentication token

the rate limit budget is consumed cumulatively across all configured sources and repositories within one run. When a limit is reached, the extractor must wait before continuing, which causes idle time that increases as more data volume is extracted.

Next to rate limits, fetching all records from scratch for each pipeline run is another source for overhead. There is no watermark and no incremental extraction, as the same data is re-fetched on every pipeline run regardless of whether it changed. Incremental extraction would reduce this overhead, but it is difficult to realize for git-based sources. Git history rewrites such as force-pushing can silently remove or alter previously seen commits and break the monotonic append assumption that a watermark-based approach depends on. Using source-level parallelism is another way to address the extraction cost, by running the per-source extraction loops concurrently instead of sequentially. Similarly, intelligent orchestrator triggering would reduce the amount of data extracted by skipping the extraction of sources that are not used by the downstream Gold-level analysis.

3.2.5 Scope of Work Packages

Of these performance characteristics, this study addresses measurement tooling and identity resolution as work packages. Establishing measurement tooling is a prerequisite for quantifying the effect any optimization has on the original implementation. With tooling in place, a suitable baseline can be recorded and used for evaluation of optimizations. Identity resolution is the most addressable candidate and provides a concrete use case for demonstrating the framework. Addressing API extraction performance requires changes throughout the extraction chain and is left as future work. Source-level parallelism and intelligent orchestrator triggering were also identified as optimization opportunities but require similar wide-ranging changes and are likewise out of scope. Section 4 formalizes these exclusions under the MoSCoW *Won't-Have* category. However, before these work packages can be designed, the theoretical foundations for measurement and optimization must be established.

3.3 Performance Benchmarking

Without measured baselines established before and after a change, it is not possible to determine whether the change had any effect on system performance. This section establishes the theoretical principles that such a framework must satisfy.

3.3.1 Measurement Validity

Lilja (2000, pp. 45–46) establishes that performance measurement is subject to two classes of error. Systematic error arises from conditions that consistently

shift results in one direction. This can be a Spark context that has not yet completed initialization or an operating system scheduler that assigns varying CPU time to the measured process. Random error arises from conditions that vary unpredictably between runs. This can be fluctuations in network latency, garbage collection pauses, disk I/O contention, or the upstream schema of an API suddenly changing. Both classes must be addressed for measurements to support valid conclusions, but they require different treatments.

To control systematic error, Lilja (2000, pp. 45–46) suggests resetting the environment to a known state before each run. Where initialization effects are relevant, an initial run should be excluded from measurement, which allows the system to reach steady state before results are recorded (Jain, 1991, § 9.3). Random error cannot be eliminated but its influence can be reduced by aggregating measurements over several runs and using a central-tendency statistic to reduce the impact of any single outlier on the result (Lilja, 2000, p. 27). Execution times specifically should be measured using the arithmetic mean, as its sum corresponds to a meaningful total execution time according to Lilja (2000, p. 30). Conversely, the harmonic mean is the appropriate summary for rates such as throughput (Lilja, 2000, p. 31). The standard deviation can be used to quantify the spread of individual measurements around the mean. Combined with the coefficient of variation (CV), which is defined as the ratio of standard deviation to mean, and which normalizes the spread to a dimensionless value, they enable the comparison across multiple measurements (Lilja, 2000, p. 39).

Another concern when benchmarking a system is the granularity of the measurements. The level at which a benchmark measures must match the level at which a bottleneck is suspected to lie. Using too coarse a granularity would aggregate the bottleneck with surrounding steps and potentially mask it entirely. A measurement taken at too fine a granularity may accurately capture one operation in isolation while missing interaction effects between steps that only become visible at a higher level. Since MECOIS is a sequential pipeline (see Section 3.1), a bottleneck in one stage can be masked or amplified by adjacent stages when measured at the wrong granularity. When the location of a bottleneck is not known in advance, measurements at multiple levels of granularity are required to locate and attribute it correctly.

Applied to ETL pipelines, Simitsis et al. (2009, pp. 2–5) propose a two-level measurement hierarchy with individual activity classification at the micro level and workflow topology at the macro level. What this does not account for, however, are the medallion layer transitions and individual steps within a medallion layer as distinct measurement units. These two tasks typically encompass more than a singular activity like calling a service but perform a more specific action than a full orchestrator run. This suggests widening the Simitsis et al. (2009) hierarchy into a three-level benchmarking design. How these measurement tiers are specifically

assigned to the MECOIS pipeline is developed as part of the framework design in Section 5.

3.3.2 Reproducibility

A measurement that cannot be reproduced provides no basis for comparison. In order to establish reproducibility, deterministic inputs, an isolated execution state, and a clearly defined reset of that execution state after each use need to exist. Jain (1991, § 9.3) suggests that the failure to reset state between benchmark runs is a common source of non-repeatable results. Additionally Lilja (2000, p. 45) observes that failing to restore the identical system state introduces systematic error. Deterministic inputs cause each benchmark run to operate on a fixed dataset as opposed to changing production data, so that all observed differences between runs represent the operation of the system and not variations in input. If a benchmark leaves behind modified data, like open connections or cached state, subsequent runs measure different conditions and produce results that cannot be meaningfully compared.

The setup and teardown pattern described below can be used to address the execution state requirements described above. Before each run, a setup step prepares the execution environment to a known state. After each run, a teardown step releases resources and removes any side effects produced during the run. When applying this pattern to benchmarks, it ensures that each measurement reflects only the operation under test and not the accumulated effects of prior executions.

In addition to establishing a method for reproducing results, establishing a method for comparing results across time is required. The baseline-and-compare approach establishes a known-good result set once and evaluates subsequent runs as deviations from that baseline (Nguyen et al., 2012, pp. 300–301). Instead of considering an individual measurement as having value independent of other measurements, this approach considers whether performance has changed since last measured relative to the previously established baseline. To effectively utilize this approach, however, a threshold needs to be established. This is a percentage change above which an observed deviation from the baseline is considered to be a degradation of performance instead of random fluctuations in measurement. Thresholds with a fixed percentage are simple to implement but they must be calibrated carefully against the natural variance of the measurement. A threshold that is too narrow will produce false positives on measurements with high natural variance. One that is too loose will fail to detect genuine regressions until they have grown large enough to exceed the noise floor. Practical considerations associated with these calibration trade-offs are discussed in Section 7.2.3.

Using a singular baseline as described above to observe performance over time presents two possible blind spots, based on the point in time the baseline is

recorded. A fixed baseline that was recorded once at the beginning of the comparisons anchors the regression detection to the system's initial behavior. However, should the system slowly improve over time, this initial fixed starting point becomes miscalibrated, causing a later regression to fall inside the baseline's threshold band. As a consequence, the fixed baseline misses the sudden regression of the system, as shown in Figure 3.5 (panel A). A rolling baseline on the other hand is able to capture this regression. This baseline is continuously updated with the latest available measurement and captures the evolution of the system between the previous and current measurement. Panel B of Figure 3.5 shows that it cannot detect a gradually increasing regression, which the fixed baseline is able to detect. The two blind spots cannot be addressed by a single baseline, since they occur at different times in the continuous benchmarking process. A combined strategy addressing this is discussed in Section 5.1.7.

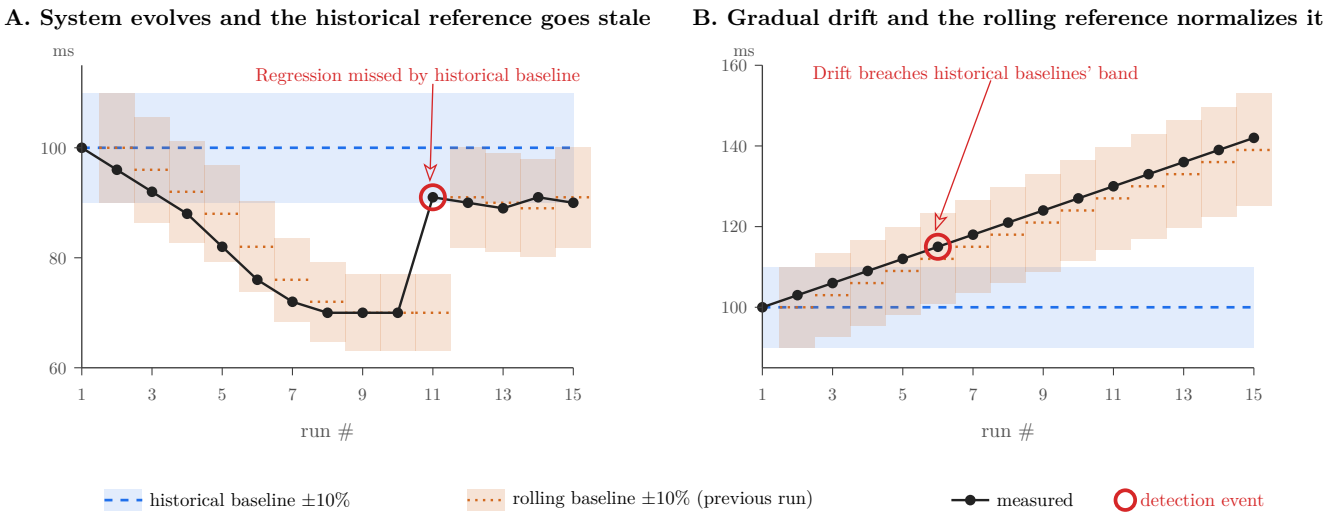

Figure 3.5: Complementary detection blind spots of historical and rolling baselines, shown for a regression after system evolution (panel A) and for gradual drift (panel B).

3.3.3 Real vs. Simulated Workloads

To assess the validity of a benchmark, not only the measurement methodology needs to be considered, but also what is being measured. Replacing live service dependencies with mock implementations is common to achieve small-scoped unit tests, but it needs to be considered carefully when creating a performance benchmark. If the target of the benchmark is the performance of the actual system, using a mock would measure the mock's performance, not that of the system. Network latency, for example, is highly variable in practice. Most requests complete within a short time, but a non-negligible fraction take substantially longer (Dean & Barroso, 2013, pp. 76–77). A mock that responds without delay will omit any variation. Thus using a benchmark that relies on such a mock cannot produce valid latency measurements for network-dependent code. For code that issues

API calls or interacts with external services, it is important to use real service endpoints that reflect the actual conditions in production instead of synthetic ones. Similarly, workload inputs must be representative in their structure and scale. A benchmark that operates on a small synthetic dataset might not expose behavior that is volume-dependent. If the production use of a system is likely to encounter large data sets, behavior like pagination handling and memory pressure should be exposed by the benchmark.

3.3.4 Continuous Performance Testing

Code changes introduce performance regressions in the same way that they can break the correctness of the code. A change that improves one part of a pipeline may degrade another. Without automated detection, regressions accumulate across multiple changes and only become visible through user-observable slowdowns that are difficult to attribute to a specific change. Integrating performance benchmarks into a CI pipeline allows regressions to be detected at the change that introduced them, before they reach production.

A CI-integrated approach inherits the requirements outlined in Section 3.3.2, but also needs to consider how often benchmarks should run. Executing the regression detection on every pull request (PR) would be the natural choice, as all changes to the project are required to come from a PR. However, this becomes impractical when benchmarks depend on external services like GitHub. Every time such a benchmark is executed, it consumes rate-limited API quota which could get exhausted quickly for large data sets. Furthermore, benchmarks can take longer than typical CI builds designed for fast feedback. Humble & Farley (2010, p. 120) and Beck & Andres (2004, p. 75) recommend fast feedback loops in CI stages that are no longer than 10 minutes. Benchmarks can take much longer than that (Laaber & Leitner, 2018, p. 122). Using nightly or weekly execution schedules avoids exhausting the quota and keeps PR feedback fast, while still attributing regressions to a narrow change window.

3.4 Identity Resolution Optimization

Section 3.2.5 identified sequential SortingHat API calls as the most addressable candidate in the Bronze-to-Silver transformer. Similarly, Section 3.2.3 documented that some Gold-level analytics pipelines use the same identity resolution mechanisms. To address this, three topics need to be inspected. The first one is investigating why these HTTP calls are problematic. Based on that, the question becomes why the cost is not eliminated by distributing these calls across Spark executors. The last topic is how batching reduces the number of requests without altering the data semantics.

3.4.1 Sequential Request Cost Model

Each time the pipeline makes an HTTP request to SortingHat, a fixed cost is incurred that is independent of its payload. First the client has to establish the TCP connection to the server and frame the HTTP request. Then on the server side, the dispatching and serialization of the response incur additional cost. While HTTP keep-alive can be used to reduce this overhead, it cannot eliminate it. Rather, it reduces the TCP handshake cost across multiple requests, but server-side processing and the network round-trip are still incurred on every request. Additionally, when a client issues requests sequentially, each one blocks until the previous request completes. This results in the total latency for N requests to the same service representing approximately N times the per-request latency. In contrast, issuing N requests in parallel results in total latency approaching the latency of the slowest single round-trip request rather than growing linearly with N.

When the number of requests N increases with the size of the dataset, the total latency grows linearly with the number of requests issued. Client- or server-side caching can reduce this by avoiding repeated lookups for previously resolved identities. However, for workloads dominated by identities that were previously unseen, the request count remains proportional to the record count. In both cases the linear relationship between issued requests and total latency is the primary throughput ceiling. The system has to reduce the number of requests in order to improve beyond it.

As documented in Section 3.2.2, within the Bronze-to-Silver transformation at least one request is issued per identity field per row in the DataFrame. Similarly, the code change summarization analytics module issues identity resolution calls during its Gold-level filtering and formatting stage (see Section 3.2.3). Both are limited by the same throughput ceiling, and any optimization needs to reduce the number of requests.

3.4.2 Distributed Execution and I/O Locality

While parallelism does not reduce the number of requests, it is still an option for reducing the total request latency, as shown in Section 3.4.1. Since Spark forms the basis of MECOIS' data processing, it is the natural target for parallelizing the SortingHat calls. Spark's execution model applies the parallelization at the partition level. It assigns independent partitions to executor tasks and processes them concurrently (Zaharia et al., 2010, pp. 2–4). The sequential request problem could be addressed by placing SortingHat calls inside a Spark transformation such as `map()` or `flatMap()`, and expecting the executor parallelism to distribute the calls across partitions and issue them concurrently. However, two properties of Spark's execution model prevent this.

First, the executor task that processes a given partition does not provide any parallelism within the task itself. It iterates through the partition on a single thread and processes the rows one after another. Therefore the network calls to SortingHat are still issued sequentially within the partition and the partition count only sets the number of independent request sequences.

Spark's method of serializing closures is another issue. To distribute a transformation to executors, Spark serializes the function closure and its captured state. Each executor then re-instantiates any stateful object captured in the closure (such as an HTTP session or a connection pool) once per partition it receives. Connection reuse across rows is therefore not possible when a network client is held inside a closure.

The solution addresses both problems independently. Serialization of the closure is resolved by instantiating the network client inside the transformation scope, instead of using an existing one from an outer context. The sequential request problem cannot be addressed from within Spark. It requires a different mechanism, which is described in Section 3.4.3.

3.4.3 Batch Request Amortization

Batching addresses the sequential request cost by grouping multiple logical operations into a single HTTP request. In GraphQL, field aliases (GraphQL Foundation, 2021, § 2.7) provide the language-level mechanism for this by allowing multiple independent operations to appear in a single document under distinct response keys. When batching is used, the fixed costs associated with an individual request are incurred only once instead of once per operation. As such, with a batch size of b, the fixed cost per operation drops from $O(1)$ to $O\left(\frac{1}{b}\right)$. For N total operations, the number of requests falls from $O(N)$ to $O\left(\lceil \frac{N}{b} \rceil\right)$.

A secondary benefit arises on the server side. A single request carrying b operations can allow the server to handle them in parallel and improve the throughput beyond what the round-trip reduction alone provides. However, the degree to which this mechanism is effective depends on the server implementation and is not guaranteed. When implemented, it adds to the client-side improvement.

In practice, selecting a suitable batch size poses a challenge. Choosing a batch size that is too small would provide little benefit. A large batch size is desirable, but if it is too large, it increases server-side processing time per request, potentially triggering request size limits or timeouts. This would in turn reduce fault isolation since if a single operation in the batch fails or the request times out, all b operations must be retried together. These competing constraints mean the optimal batch size is highly specific to the service and its limits. This justifies making it a configurable parameter rather than a hardcoded constant. As a default, 100 was

chosen to achieve a balance between keeping the GraphQL request size small while still resulting in a significant speedup.

Applied to the SortingHat integration described in Section 3.2.2, this batching pattern would replace the per-row API calls with aliased GraphQL mutation documents, each carrying up to b identity operations. The resulting request count of $O\left(\lceil \frac{N}{b} \rceil\right)$ directly addresses the linear scaling identified in Section 3.4.1.

The three-tier measurement structure and the batching pattern provide the theoretical foundations for the requirements stated in the next chapter and the architecture that follows.

4 Requirements

The analysis in Section 3 identified two issues in the current pipeline. First, the pipeline lacks any instrumentation for measuring and evaluating its performance. Second, the Bronze-to-Silver transformation resolves developer identities by issuing an individual HTTP request to SortingHat per record. When the identity already exists, an additional request is required to resolve it to its canonical identifier.

These findings motivate two work packages (WP). WP1 addresses the measurement gap by introducing a performance benchmarking framework. WP2 addresses identity resolution by replacing per-row API calls with a batched GraphQL alias implementation. WP1 is a logical prerequisite for WP2. The optimization can be implemented independently, but its effect on pipeline performance can only be quantified once the measurement infrastructure from WP1 is in place.

This chapter states the requirements for both work packages. Each requirement is derived from the problems identified in Section 3, the gaps in existing tools established in Section 2.2, or established engineering practice. They are grouped by their completion priority according to the Must-Have, Should-Have, Could-Have, Won't-Have this time (MoSCoW) framework (Kravchenko et al., 2022).

4.1 Must-Have Requirements

These requirements provide the minimum usable subset and are required for the contributions of this thesis.

WP1 RQ-1: *The benchmarking framework shall measure pipeline performance at three granularity tiers: individual external service operations (Tier 1), pipeline stage executions (Tier 2), and complete orchestrator runs (Tier 3).*

A two-level ETL measurement hierarchy was proposed by Simitsis et al. (2009) as a basis for benchmarking, with one level for individual activities and a coarser one for general workflow measurements. This three-tier structure extends this by introducing an intermediate tier. It includes individual pipeline steps like extraction from a specific source or medallion layer transitions as distinct measurement targets. For instance, a Tier 2 benchmark measurement can attribute overhead to a specific Bronze-to-Silver run without specifying a single service call as in Tier 1 or diluting the measurement across a full Tier 3 orchestrator run.

WP1 RQ-2: *The benchmarking framework shall persist benchmark results as a baseline associated with the codebase state at the time of capture.*

The system analysis identifies the absence of a performance baseline as the root cause of invisible regressions. Without a stored baseline each run represents an isolated data point without any reference for comparison. A baseline is the minimum artifact required to detect changes across separate runs.

WP1 RQ-3: *If a benchmark result deviates from the stored baseline by more than a configurable percentage threshold, the benchmarking framework shall classify the run as a regression.*

Without automated detection, regressions can accumulate silently across multiple changes. They only become visible when users observe slowdowns during pipeline execution. At this point it is difficult to pinpoint what change caused the slowdown. To account for natural variance during measurements, a configurable threshold shall be used. It has to be configurable, since a short benchmark that contacts an external system can exhibit larger variance than a longer benchmark.

WP1 RQ-4: *The benchmarking framework shall execute benchmarks against live service instances rather than mock implementations.*

While mock implementations respond synchronously and thus speed up the benchmarking process, they also bypass network calls entirely. However, the bottleneck identified in Section 3 is dominated by network round-trips to SortingHat during identity resolution and to external APIs during extraction. Consequently, a benchmark using mocked responses does not accurately represent the production conditions and therefore cannot adequately measure valid before/after performance comparisons.

WP2 RQ-5: *The optimized SortingHat integration shall collect identity fields from the Bronze DataFrame, group them into batches, and submit each batch as a single GraphQL request using field aliasing.*

The current implementation, documented in Section 3.2.2, issues one API call per added identity and optionally a second call to retrieve its canonical UUID, should the identity already exist. As a consequence, this produces one or two sequential HTTP requests per identity field per row in the DataFrame. Since the GraphQL specification (GraphQL Foundation, 2021, § 2.7) defines field aliasing as the mechanism for submitting multiple independent operations in a single request, this method is well suited to reduce the per-request overhead. This is accomplished by replacing the sequential calls with batched ones, applying field aliasing to the SortingHat requestor within the Bronze-to-Silver transformation.

WP2 RQ-6: *If provided the same input Bronze DataFrame, the optimized SortingHat integration shall produce a Silver DataFrame with the same canonical UUIDs, the same schema, and the same merge semantics as the sequential implementation.*

The accuracy of MECOIS' Gold-layer analytics depends on a stable canonical UUID for a given identity to connect contributions across various sources. An

optimization that alters the identity output would corrupt those analytics without producing any visible error. Consequently, the correctness of the implementation cannot be traded for performance.

WP2 RQ-7: *The optimized SortingHat integration shall expose the same call interface as the sequential implementation, requiring no changes to calling orchestrators or pipeline step definitions.*

The pipeline step hierarchy described in Section 3.1 is the architectural boundary of MECOIS. Changes that propagate above the Bronze-to-Silver transformer into orchestrator definitions are out of scope for this work. They would introduce regression risk across pipeline configurations that are not under test. The optimization must therefore be contained within the transformation layer.

WP2 RQ-8: *The optimized SortingHat integration shall reduce the HTTP request count for identity resolution by a factor of at least the configured batch size compared to the sequential implementation.*

Alias-based batching reduces the request count by a factor of the configured batch size b from $O(N)$ to $O\left(\lceil \frac{N}{b} \rceil\right)$. Additionally, for any batch size b, the ratio of individual to batched requests is approximately b, with a larger batch size yielding a proportionally greater reduction. This follows directly from the batching definition and holds regardless of workload size.

WP2 RQ-9: *The optimized SortingHat integration shall be verified by integration tests executed against a live SortingHat instance, covering batch boundary behavior, null and partial identity handling, and merged-identity resolution.*

RQ-6 requires the batched implementation to return the same canonical identities as the unbatched baseline. Unit tests with mocked responses cannot verify this, since the correctness of the batching depends on SortingHat's behavior. Therefore, integration tests against a live instance are required to establish that the correctness required by RQ-6 is achieved.

4.2 Should-Have Requirements

The following requirements are important and painful to leave out, but not vital to the viability of the solution. Omissions are defensible, but they should typically provide a workaround.

WP1 RQ-10: *The benchmarking framework shall provide a base class that encapsulates timing, result aggregation, and result persistence, requiring each benchmark to implement only its setup, execution, and teardown logic.*

Without a common interface, every benchmark repeats infrastructure boilerplate and the CI comparison and persistence logic cannot be generalized across benchmarks. The common interface also enables the CI workflow to invoke any

conforming benchmark without modification, by finding all classes that implement the base class.

WP1 RQ-11: *The benchmarking framework shall support the addition of new benchmarks without modification to the base class, the CI workflow definition, or the result storage schema.*

New measurement targets are identified as the pipeline evolves. The framework must accommodate them without requiring infrastructure changes each time. If each new benchmark requires changes, the marginal cost of extension eventually exceeds the cost of ad-hoc one-off measurement, defeating the purpose of a shared framework. This requirement is verified by implementation inspection in Section 6.

WP1 RQ-12: *The benchmarking framework shall execute benchmarks automatically on a schedule and report the comparison outcome against the stored baseline.*

No surveyed ETL and pipeline benchmarking tools from Section 2.2.2 provided automated regression detection in a CI workflow. Per-pull-request execution is impractical due to the involved extraction API calls consuming rate-limits and the overall long runtimes. A scheduled overnight run avoids both while still attributing regressions to the change window that introduced them instead of allowing them to accumulate undetected.

WP1 RQ-13: *The Tier 1 and Tier 2 benchmarks shall complete their combined execution within 20 minutes of wall-clock time on the default GitHub Actions runner.*

As noted in Section 3.3.4, 10 minutes is the upper limit for CI commit stage builds to give developers timely feedback. Since the benchmark job is not a commit stage build, and it includes full Spark Context initialization and live API calls for several benchmarks, 20 minutes was selected as double this ceiling. This limit is set as a Should-Have requirement because runtime cost and benchmark coverage pull in opposite directions. The 20 minutes may be exceeded if a benchmark required for valid measurements runs longer. While this does not determine how long a benchmark run may take, it aims to keep the combined Tier 1 and Tier 2 execution from consuming most of the weekly CI schedule.

WP2 RQ-14: *The optimized SortingHat integration shall reduce the wall-clock duration of the Bronze-to-Silver pipeline stage compared to the sequential implementation, with the magnitude of improvement established by the Tier 2 benchmark.*

Each sequential SortingHat call incurs a full network round-trip. Batching reduces this round-trip count by a factor of approximately the configured batch size, as required by RQ-8. Since network I/O is the dominant cost per SortingHat call, the stage-level duration is expected to decrease accordingly. The proportion of Bronze-to-Silver time attributable to identity resolution, and therefore the achievable stage-level speedup, will be quantified by the Tier 2 benchmark (RQ-1) as part of

the baseline measurement. No numeric target is stated here since the evaluation in Section 7 reports the measured improvement and assess whether it is sufficient to justify production adoption.

WP2 RQ-15: *The optimized SortingHat integration shall provide the operator with the ability to configure the batch size at runtime without code changes.*

The optimal batch size depends on various factors like the distribution of records across pipeline runs and the capacity of the SortingHat instance. A batch size that is too small fails to reduce per-request overhead. Conversely, one that is too large risks server-side timeouts and forces all b operations in a failed batch to be retried together, reducing fault isolation. Configuring the size at runtime allows tuning the pipeline without having to adjust the code.

4.3 Could-Have Requirements

The requirements listed below are preferred enhancements. Time permitting, they will be implemented, but the system functions correctly without them.

WP1 RQ-16: *The benchmarking framework should provide a visual representation of the benchmark health across multiple runs to give a health indication at a glance.*

Pairwise comparison against a single reference (RQ-3) provides an indication if there are issues relative to this one reference. However, it does not indicate if there is a directional trend across multiple runs. An accumulated history comparison graph reveals how gradually drifting values may be trending downward while staying under the per-run limit. This was designated as Could-Have since the required Must-Have regression detection works without it.

4.4 Won't-Have This Time

These items were considered but excluded from the scope of this thesis.

Several additional optimization opportunities were identified in Section 3.2.5 but have been intentionally left out of this thesis: incremental delta extraction, source-level parallelism across orchestrators, and intelligent orchestrator triggering that runs only the sources required by a target analytics module. While each can be independently beneficial, they are outside the scope of this thesis. They depend either on unanswered implementation questions such as git history modifications to support delta extraction, or on changes to more interfaces than the work in Section 3.2.5 covers. The most relevant of these are revisited under Future Work in Section 8.

4.5 Evaluation Criteria

Requirement	Priority	Verified in
RQ-1	Must	Usability and Extensibility 7.2.1
RQ-2	Must	Reproducibility Analysis 7.2.2
RQ-3	Must	Regression Detection Validation 7.2.3
RQ-4	Must	Reproducibility Analysis 7.2.2
RQ-5	Must	Batched Performance 7.3.2
RQ-6	Must	Batched Performance 7.3.2
RQ-7	Must	Batched Performance 7.3.2
RQ-8	Must	Batched Performance 7.3.2
RQ-9	Must	Evaluation Methodology 7.1
RQ-10	Should	Usability and Extensibility 7.2.1
RQ-11	Should	Usability and Extensibility 7.2.1
RQ-12	Should	CI Integration Assessment 7.2.4
RQ-13	Should	CI Integration Assessment 7.2.4
RQ-14	Should	Batched Performance 7.3.2
RQ-15	Should	Batched Performance 7.3.2
RQ-16	Could	Usability and Extensibility 7.2.1

Table 4.1: Traceability from requirements to MoSCoW priority and evaluation sections.

5 Architecture and Design

Section 3.2.5 defined two work packages for this thesis, which are supported by the theory developed in Section 3.3.1 and Section 3.4.1. The first is the benchmarking framework, which establishes the measurement infrastructure specified by WP1. The second is the batched identity resolution design, which replaces the individual HTTP requests in the Bronze-to-Silver transformer as required by WP2. In this chapter, the architecture and the design of both work packages are outlined, starting with the benchmarking framework and followed by the identity resolution optimization.

5.1 Benchmarking Framework

The following covers the design decisions for the framework and presents its components. The section then shows how the three-tier design outlined in Section 3.3.1 is applied to the MECOIS project. Next, it discusses the selection of the extraction repositories for GitHub and GitLab, followed by the design of the baseline management. The section concludes with the integration of the framework in the CI workflow.

5.1.1 Framework Architecture

In the design of the framework, an important first choice is how responsibilities should be spread across its components. Three options were considered: using an existing benchmarking library, organizing the framework into modules with specific responsibilities, or building several individual and self-contained benchmarks.

Pytest-benchmark is a widely used benchmarking library in the Python ecosystem. It provides timing, warmup runs, outlier filtering and result storage and uses a fixture-based execution model where setup and teardown are managed by *pytest* (Ciuraru, 2014). This is already used by MECOIS for its unit and integration tests. However, this model complicates integrating live Spark sessions and external API calls. It assumes cheap, side-effect-free test functions that are repeated multiple times per test. Live Spark sessions and external API calls require multi-second warmup and produce side effects across executions, going directly against these assumptions. Beyond this integration mismatch, it provides no mechanism to automatically carry baselines across ephemeral CI runners, as required by RQ-2.

Using multiple self-contained benchmarks for each targeted system would avoid shared infrastructure by not relying on a shared framework. On the other hand,

that couples every benchmark to a specific output format and persistence mechanism. This makes adding new output formats or changing how results are stored difficult, since it requires modifying each benchmark individually, which conflicts with RQ-11. Independent comparison is also impractical. Comparing two results would require re-executing the benchmark instead of loading previously saved results.

A custom framework with a modular design directly addresses these issues, with each module encapsulating a single concern that can be changed without affecting others. Instead of depending on the life-cycle handling from a harness like pytest, a modular framework is able to define its own. This way, shorter tests only involving Spark fit into the same life-cycle as longer-running API extraction benchmarks that produce various side effects. Using separate modules for the entry point of the framework, core components that are shared by all modules, benchmark execution, comparison, formatting of the output, and the benchmarks themselves creates a flexible design that simplifies development.

These modules are shown in Figure 5.1. A command-line interface (CLI) serves as the entry point to the framework. It serves as the orchestration layer by separating the intended action, like execution and comparison, from the actual implementation of that action. As a separated action, the benchmark execution module handles the discovery of the available benchmarks and their execution. It also collects and persists the benchmark results. Similarly, the comparison of benchmark results is handled by the comparison engine and decoupled from the CLI. By treating stored results as first-class artifacts, it is able to load and compare any two results, removing the need for dedicated baseline-handling inside the engine. The computed result of the comparison is then formatted by separate output formatters so that changes to the format do not require changes to the comparison logic. Shared components, such as a timer to measure the benchmark runtime and data models used for results, are grouped in a separate core layer.

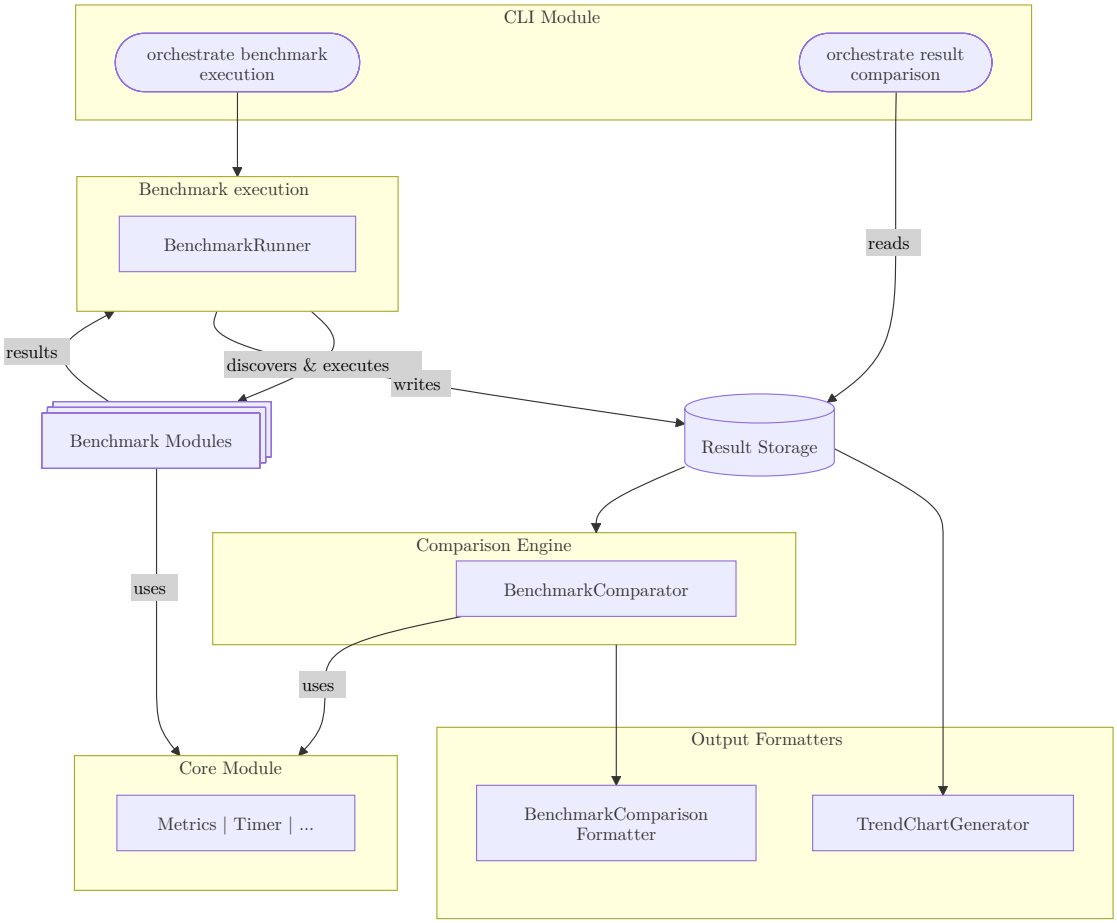

Figure 5.1: Module structure of the benchmarking framework with data flows between the modules.

The following sections go into detail on the specific design decisions required for the described modules.

5.1.2 Core Layer

The core layer implements shared abstraction concepts. They include a Timer concept and a data model concept for the benchmark results.

In order to benchmark the components of the pipeline, the framework has to measure their runtime. There are four functions in the Python standard library that can be used as alternatives for timing each measured section of code.

Function	Clock	I/O wait	Notes
time.time()	Wall-clock	Yes	Not monotonic: network time protocol (NTP) corrections can shift the clock backward, producing negative intervals

Function	Clock	I/O wait	Notes
time.process_time()	CPU	No	Excludes I/O wait and omits the dominant cost for network-bound benchmarks
time.monotonic()	Monotonic	Yes	Resolution not guaranteed and may be coarser on some platforms
time.perf_counter()	Monotonic	Yes	Highest available resolution and the default timer in Python's *timeit* module

Table 5.1: Python standard library timer candidates and their properties relative to the benchmarking requirements.

Based on the options presented in the table, `time.perf_counter` was selected because it is monotonic and provides the highest available resolution. It also includes I/O wait times and can be used in any measuring context. This makes it able to measure both very short-running benchmarks of specific services and long-running end-to-end benchmarks.

Next to the time measurement, the core module provides a two-layered design for storing the benchmark results as structured records. Individual measurements within a benchmark are stored with their value and an indicator for the direction in which the measured value improves. This indication is required for the subsequent comparison of individual metrics, since the comparison engine needs to decide whether a divergence of a value is either a regression or an improvement. All recorded measurements are then grouped into a parent record that additionally contains the metadata of the run in which the metrics were collected. These records are designed to use strict schemas. Without enforcing a schema for the metrics, comparisons across benchmark runs risk causing failures when one benchmark does not include a key that another one expects. However, some recorded data within a given benchmark might not be directly comparable, such as caching metrics that might change between implementations. These data points should be excluded from comparison. The schema storing the results is thus designed to include a single entry serving as a bucket for non-comparable data.

The metadata included in the result file is designed as two separate entries. While the first one contains metadata about the run itself, like the timestamp of the run and a summary of the results, the other stores more specific data about the environment the benchmark was executed in. This includes details about the executing system and the git branch and commit hash of the repository. Including this data allows for a direct assignment of a given test run to the code it tested, and satisfies RQ-2.

5.1.3 Comparison Engine

The comparison of the recorded metrics is handled by the comparison engine, which implements a pairwise regression classification on stored results. As shown in Figure 5.1, it is decoupled from the execution so it operates on any pair of saved results. If it were integrated into the execution stage, comparisons would have to be against the result obtained within the current run. Consequently, each time a historical comparison is needed, benchmarks would need to be re-executed. Keeping the comparison standalone lets it operate on any two saved result files and compare any pair of historical runs.

The comparison always operates on two benchmark result files. This pairwise comparison classifies each benchmark result against a reference baseline and produces pass, regression, or improvement decisions for that run, as shown in Figure 5.2. For each metric recorded in both runs, the comparison computes the percentage of the change. The classification then depends on the desired outcome for a given metric.

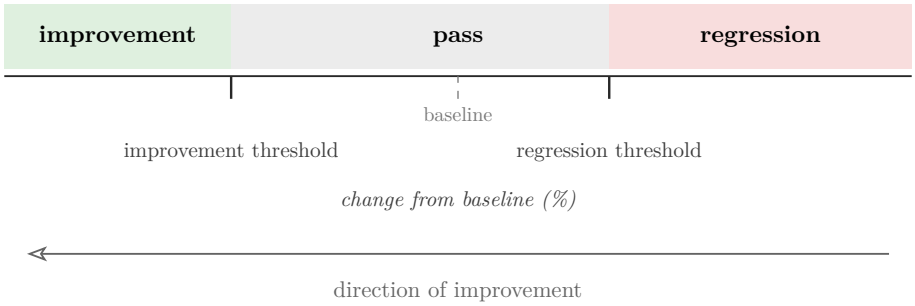

Figure 5.2: Classification of a metric's change from baseline into improvement, pass, or regression against asymmetric thresholds.

Should a change exceed a configured threshold in the direction of the improvement threshold, it is recorded as an improvement. When the threshold is surpassed in the wrong direction of the outcome, a regression occurs. Benchmarks present only in the current run are reported as new, and those missing from it are reported as missing. The configuration of the threshold is designed as an overridable hierarchy, starting with a global fallback threshold that is applied to all benchmark metrics. Should a benchmark need a more specific threshold, a per-benchmark override can be specified, and a metric-level threshold takes precedence over both. A short benchmark that contacts an external system can accumulate larger jitter than a long-running pipeline run. Furthermore, within a single benchmark, the duration varies based on external influences, whereas the API call count is dictated by the test data, so any change signals a regression. Figure 5.2 shows this for a sample metric configuration. A tight regression threshold catches regressions quickly and a looser improvement threshold avoids counting typical noise as a real gain.

Consequently, the used hierarchical system offers a flexible way of handling the varying requirements of the benchmarks, satisfying RQ-3.

Classified as a Could-Have requirement, RQ-16 requires an indication of the benchmark health past the pair-wise comparison of two results. To enable this temporal analysis, the results of individual comparisons need to be accumulated over time. This trend analysis accumulates result counts from each comparison into a history and generates a chart that shows the aggregate health over time. This visualizes possible gradual drift, which would be hard to identify based on individual comparisons only. It is designed as a separate phase of the comparison because it operates on accumulated comparison results instead of a pair of result files. However, an additional formatting option is required within the comparison since the run result needs to be added to the history used for the trend analysis.

5.1.4 Output Formatters

As shown in Figure 5.1, two output formatters handle the generated comparison and history data. The comparison formatter uses the output of the comparison engine, while the trend formatter handles the trend diagram generation from the historical data. Both are kept as separate modules from the engine to ensure a clean split of responsibilities. The comparison engine operates on two result files and the formatter operates on the data obtained from the comparison. To address a wide audience, the formatters are designed to produce comparison reports in multiple formats. Markdown was chosen as the format targeted for human inspection. This is because it is already widely used within the MECOIS project and supported by GitHub as the used code hosting platform. For programmatic consumers, JSON was chosen as the output format, since it is widely supported. Additionally, a static badge can be produced from the comparison that provides a fail/pass indication at a glance without having to read the report.

With the comparison triggered by the CLI, the formatters are designed to support a single invocation producing any combination of reports. This avoids repeating the comparison computation for each report. The comparison formatter implements the multiple formats described above and the trend formatter generates the visualization of the benchmark health required by RQ-16.

5.1.5 Three-Tier Benchmarking Design

To satisfy RQ-10, the framework provides an abstract base class that defines the benchmark execution. The variable behavior of each benchmark is delegated to concrete subclasses while the abstract base class only ensures the uniform structure is maintained. This uniform contract supports auto-discovery through allowing the benchmark runner to instantiate and execute any subclass without prior registration. This meets the requirement stated by RQ-11. The base class orchestrates

the benchmark's life-cycle in a fixed sequence, by running setup, timed execution, and teardown. Additionally, it guarantees that the teardown sequence is executed on failure so that live resources such as Spark sessions and API connections are always released.

The three-tier structure required by RQ-1 is realized as three sets of benchmark classes, each targeting a different level of the MECOIS pipeline. As developed in Section 3.3.1, medallion transitions and individual pipeline steps are intermediate measurement units that a two-tier structure cannot address without collapsing them into adjacent tiers. Figure 5.3 illustrates how each tier covers a different portion of the medallion stack. Additionally, as required by RQ-4, all benchmarks are designed to use live service endpoints. They expect a running SortingHat instance and the extraction benchmarks require GitHub and GitLab to be accessible.

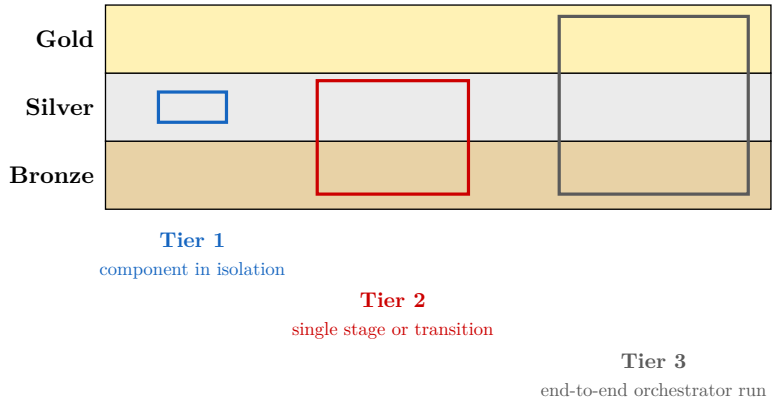

Figure 5.3: Scope of the three measurement tiers relative to the medallion layers.

Tier 1: Component isolation. Tier 1 benchmarks target individual service operations in isolation from the broader pipeline. Components that are part of a larger pipeline need to be tested in isolation to determine their individual contribution to the runtime. Measuring them only within a pipeline absorbs their individual cost into the total duration. Since the SortingHat requestor is the focus of WP2, it is the natural subject for component-level benchmarking. By isolating it at Tier 1, the framework can measure the effect of the batching optimization directly. Were it only executed inside a full Spark pipeline, the measurements would include JVM startup overhead and Delta Lake I/O and obscure the requestor-level cost. Tier 1 benchmarks therefore instantiate the requestor directly, bypassing Spark entirely.

The workload benchmarking the SortingHat requestor performs two passes over the same identity set. This design covers both production cases of encountering new identities in a dataset and repeated inputs in the same run.

Since the intended optimization of the SortingHat requestor is reducing the number of API calls made to the endpoint through batching, counting the number

of API calls helps with evaluating the optimization. This HTTP call count is tracked alongside the duration as the primary instrumentation metric. Excluded from the call count and resolution timing is the authentication with the SortingHat instance. Including this one-time request would combine it with the per-request latency and throughput that the benchmark is designed to isolate.

Tier 2: Stage-level measurement. Tier 2 benchmarks measure the execution of individual pipeline steps and the transitions between medallion layers. To satisfy RQ-4, pipeline steps that involve API extraction stay unchanged and issue live API calls to the intended external systems. Benchmarks that target the Bronze-to-Silver transformation equally issue live API requests to SortingHat, but use synthetic input data to ensure independent and repeatable benchmark work. The framework is designed to uniformly handle the creation and timing of the benchmarked steps and transitions as well as capturing the output across all stages. This ensures that identical measurement conditions are maintained and simplifies adding new stage benchmarks.

The synthetic test data for the Bronze-to-Silver transition mirrors the structure of production records. It uses nested fields that exercise schema flattening, identity fields that drive SortingHat resolution, date fields that trigger date transformation, and fields marked for dropping. This exercises all steps in the transformation logic. Since this benchmark includes the identity resolution, it serves as a complementary test to the Tier 1 SortingHat benchmark. The Tier 1 benchmark primarily measures the network round-trip and exercises the call count mechanism by using a higher number of identities than the LRU cache size of the SortingHat requestor. In contrast, by using a low cardinality for the synthetic Bronze-to-Silver identities, the cache can absorb repeated lookups and thus isolate partition-level execution gains.

PySpark's lazy evaluation model, however, creates a measurement problem. Spark DataFrame transformations are not executed until they are triggered by a specific action. So benchmarks that involve Spark operations have to explicitly materialize the respective DataFrames before the timer is stopped. If this step is omitted, the actual computation performed on the DataFrame would never be executed. Thus, the framework forces materialization of each step's output before recording elapsed time.

Tier 3: End-to-end orchestrator runs. Tier 3 benchmarks execute complete orchestrator pipelines from extraction through the full medallion stack. They are used to capture effects that occur in multiple stages and are invisible when stages are executed on their own. In addition, they can show whether a change at one stage produces a measurable effect on the full pipeline execution.

The setup step is central to the reproducibility of the Tier 3 benchmarks. Each run has to start from an empty state to make successive runs comparable regardless of what previous runs produced. Otherwise, leftover data from previous runs would

alter the work done by following runs and turn the measurement into a cumulative trace rather than a repeatable comparison.

The CI workflow runs all tiers on a weekly schedule. Per-pull-request execution is opt-in via a marker in the pull request body. This opt-in mechanism was chosen, so that routine pull requests do not consume API quota or runner time, while on-demand execution remains available when needed (see Section 3.3.4). The weekly schedule means regressions are attributed to a change window rather than a specific commit. This is a weaker attribution than per-commit detection, but it is the strongest attribution achievable without live API calls on every pull request.

5.1.6 Repository Selection

Extraction stage benchmarks and Tier 3 orchestrator runs require a real repository as their data source. Selecting an appropriate repository has direct implications for the reproducibility of a given benchmark. If the dataset changes between runs, timing differences reflect data volume growth as much as implementation changes, making regression detection unreliable.

Four different approaches were considered for choosing an appropriate repository for the extraction benchmarks. Active repositories under development change between benchmark runs. The continuous addition of new commits, pull requests, issues, and releases creates an increasing amount of new data. This results in each subsequent run extracting more data and producing longer duration values than would be expected by changes in implementation alone. Using the MECOIS repository itself was evaluated due to the direct control over it. It was rejected on the same grounds. Development continuously adds new data, causing the benchmark to measure dataset growth as much as pipeline performance. A fork of a suitable repository may provide a solution to the problem of continually growing data sets. However, GitHub and GitLab do not replicate all repository-level data when creating a new fork. GitLab explicitly excludes several key data types like issues and pull requests (GitLab, 2026). Likewise, GitHub forks have their own issues, pull requests, Actions, and tags that are not inherited from the upstream repository (GitHub, 2026). This means that a forked repository cannot exercise a large part of the available extraction targets. Small repositories with limited history were the remaining option. They provide insufficient data volume for a stable timing signal and cover only a subset of available API endpoints.

An owner can mark a repository as no longer maintained by archiving it, which makes the repository read-only. This mechanism, available with both GitHub and GitLab, satisfies both requirements mentioned above. The read-only attribute effectively freezes the repository's state. No new data can be added to it after the archival date, so the benchmark dataset is identical across every run. A repository with sufficient history before archival covers most available extraction targets such

as commits, pull requests, issues, releases, and workflow runs, providing a comprehensive extraction pipeline coverage. Therefore any timing differences between two benchmark runs can be directly attributed to implementation changes and not repository growth.

Two repositories were selected following this approach: *containrrr/watchtower* on GitHub and *gitterHQ/gitter-android-app* on GitLab. Both were identified by surveying the first several pages of each platform's archived repository list. Three criteria were applied for the selection:

- A history spanning several pages at the pipeline's default page size of 100 so that pagination is exercised across multiple API responses rather than exhausted in a single one.
- A meaningful number of contributors and activity across releases, issues, and pull requests, to exercise identity resolution realistically.
- The presence of platform-specific features such as environments, pipelines, and deployments to cover as many available extraction endpoints as possible.

Ownership-restricted endpoints on GitHub (security alerts and organization billing) are not accessible on third-party archived repositories and return no data, as noted in Section 6.1.5.2. Broad feature coverage implies non-trivial extraction volume, and the selected repositories are large enough to stress the CI execution budget, which is quantified and discussed in Section 7.2.4. Since all benchmarks targeting a given platform share a single authentication token, the cumulative API call count grows with both the extraction volume of the repository and the number of benchmarks that extract it independently. Multiple Tier 3 benchmarks each extract the same repository independently, amplifying the total consumption against the per-token rate limit documented in Section 3.2.4. The full per-platform extraction coverage breakdown is given in Appendix A.

5.1.7 Baseline Management

Section 3.3.2 identified that maintaining a single baseline is insufficient for continuous performance benchmarking. As a consequence, the framework is designed to maintain two baselines. A historical baseline compares the current run's results with a baseline that has been established at a specific previous point. It is used to detect cumulative degradation over a longer time span, for example a slow week-by-week degradation that is below the run-by-run threshold. This is shown in panel B of Figure 3.5. Additionally, a rolling baseline is used to catch changes that occur between consecutive runs, visualized in panel A of Figure 3.5. Combining both types provides complementary coverage with the historical baseline catching drift and the rolling baseline catching step changes. The disadvantage of this approach is the need to maintain two reference points instead of one. This overhead is negligible in practice, since the maintenance is covered in the CI workflow.

Bootstrapping and updating of the baselines are controlled and restricted. Both the historical and the rolling baseline are established on first run, so comparisons fall back to a self-comparison against the current result. Should a breaking change require a baseline reset, the historical baseline can be rewritten via a manual workflow trigger. The rolling baseline always makes the previous run the new reference, as it is updated after every scheduled run. Benchmark runs triggered by the opt-in pull request marker produce comparison reports but do not update the baseline. This prevents feature-branch results from overwriting the reference data used for the regression analysis.

To maintain the baselines within CI, they need to be stored and accessible to the benchmark framework. Three storage alternatives were considered for persisting baselines between workflow runs: a dedicated orphan branch, an external database, and GitHub Actions artifacts. An orphan branch is a branch with no shared history with the development branches, initially empty and holding only benchmark data. It keeps performance files out of the source history and has no external dependencies. A drawback is that it requires a commit step with repository write permissions on every run and grows without bound as results accumulate. An external database is well suited to store benchmark results. However, infrastructure has to be provisioned, credentials have to be managed, and the CI environment gains a network dependency. GitHub Actions artifacts let actions archive artifacts during the action run. They keep results self-contained within the CI platform since storage requires no repository write permissions. Additionally, their retention is configurable, no external services are required, and their expiry is automated by GitHub. The issue they pose is that standard artifacts were not designed for access between individual workflow runs, but rather for use within a single workflow run. A community-maintained action resolves this by exposing artifacts from previous workflow runs. This gives each run access to the baselines published by its predecessor.

The retention duration of the produced artifacts is configured independently for the different artifact types. Raw results and comparison outputs are retained for 30 days. This allows for a short-term review window after each run. Baselines are retained for 90 days, so that a detected regression can be investigated against any baseline from the preceding months. The weekly refresh schedule keeps the current baselines well within the retention window. The fixed 90-day retention period is preferable to unrestricted storage in orphan branches or the increased tooling of a database.

For trend visualization of the two baselines, Mermaid was used in preference to a Python plotting library. Plotting libraries such as matplotlib produce image files that have to be stored on disk and linked to from within the Markdown document. With the trend chart intended for use in the Markdown comparison report in the CI workflow, this method introduces extra friction. Mermaid diagrams can be

embedded directly as source code into a Markdown report and are rendered by GitHub's markdown engine, without needing to store a file for the diagram. The trade-off is limited visual customization. This is acceptable here because trend charts need to communicate aggregate benchmark health over successive runs, not detailed statistical distributions. An XY-diagram was chosen as the visualization method, because it supports displaying individual indicators for each of the two baselines natively. Each data point represents a health score for that run. This is computed as the total number of metrics across all benchmarks that passed or improved minus those that failed or regressed. The health scores are plotted against the run date.

5.1.8 CI Workflow Structure

The CI workflow is organized into five jobs that are connected by artifacts. Figure 5.4 shows the job structure and the artifact flows that connect them.

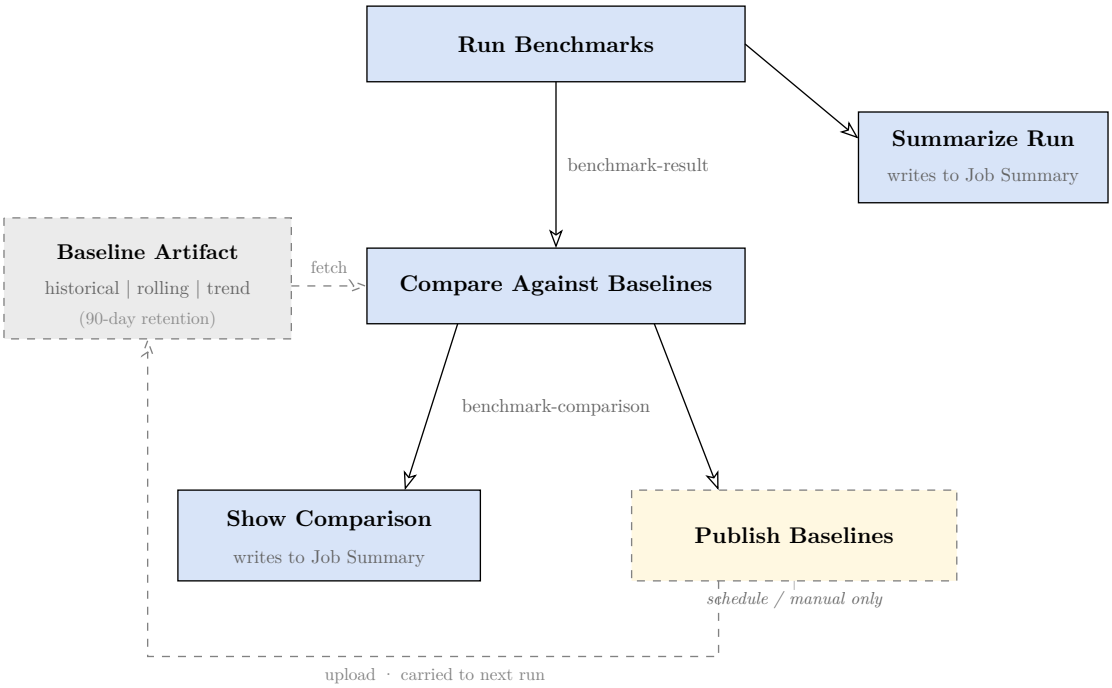

Figure 5.4: CI workflow job structure with within-run artifact flows (solid) and the cross-run baseline dependency (dashed).

There are three motivations for splitting the workflow into five individual jobs. Separating the run job from the analysis jobs makes it possible to record the results independently from any subsequent reporting steps. This means that a benchmark does not need to be repeated to look at the result artifact again. Separating the compare job from the job publishing the baselines gives the workflow better control over when the baselines get updated. The publish job is only executed in a scheduled workflow or when manually invoked. This protects the baseline from

being written by pull-request branches. Additionally, the benchmark summary is separated from the comparison and its accompanying show job, since the summary depends only on the run-job results. Generating it in parallel with the comparison provides fast feedback about the outcome of the executed benchmarks without depending on retrieval of the baselines that are required for the comparison.

The SortingHat service that the benchmarks rely on is executed within the run job. Running the service inside the runner rather than relying on an external instance makes the benchmark environment self-contained and reproducible across machines. The comparison job, coupled with the weekly scheduling, delivers the automated execution and reporting that is required by RQ-12.

The presentation of the generated reports within the CI workflow is another concern of the workflow design. Three approaches are available for displaying CI comparison results. Committing the report to the MECOIS GitHub repository would give it a permanent location outside of the GitHub Actions workflow. However, this requires a repository write on every benchmark run and directs users away from the workflow context. Alternatively, the report can be persisted as an artifact of the GitHub Actions workflow itself. This avoids writing directly to the repository, but would require downloading and extraction of the artifact in order to read the report. A solution native to GitHub Actions is the job summary mechanism (GitHub, Inc., 2022). By writing Markdown directly to the `$GITHUB_STEP_SUMMARY` environment variable, the workflow appends the report directly to the job's built-in summary page. This puts the created report directly in the context of the executing workflow job and the results are directly visible in the run view.

5.2 Batched Identity Resolution

The batched identity resolution design replaces individual HTTP requests in the SortingHat requestor. The following presents the design of the optimization based on the constraints outlined in Section 3.4.

Two constraints from Section 3.4.1 have to be addressed. First, the request count grows as $O(N)$ with both row count and identity fields per DataFrame row. Second, as noted in Section 3.4.2, closure serialization prevents HTTP session reuse across rows within a Spark partition. Additionally, SortingHat's specific identity resolution flow outlined in Section 3.2.2 for handling an already existing identity needs to be covered.

The batched design addresses the first two problems. Firstly, GraphQL field aliasing reduces the number of HTTP requests. Secondly, the partition-level execution enables session reuse and batch construction within each partition. The same mechanism also replaces the user-defined function (UDF)-based identity update service, which exhibits the same per-row request pattern.

5.2.1 Batching Mechanism

Three approaches were considered to reduce the request count made to SortingHat: asynchronous concurrent requests, a dedicated REST batch endpoint, and GraphQL field aliasing.

Issuing requests concurrently via asynchronous I/O reduces client-side runtime by overlapping round-trips, but does not reduce the total number of requests. The server still needs to handle the same amount of HTTP requests and operations. A dedicated batch endpoint would achieve both client- and server-side HTTP request reductions, but SortingHat does not provide one.

GraphQL field aliasing, established in Section 3.4 as the mechanism to reduce GraphQL requests, is the remaining option to reduce both costs. It does not require API changes to SortingHat, and it satisfies RQ-8 by reducing the number of requests issued.

To enable run-time change of the used batch size, an additional parameter is added to the existing SortingHat section in the config file for the pipeline execution. This section is already available within the SortingHat requestor because it holds the connection details used by the requestor to establish a connection to SortingHat. Adding the parameter for the batch size there satisfies RQ-15, since it requires no further code changes to adjust the batch size.

The same aliased batching mechanism applies to both the add path and the subsequent canonical UUID lookup. Batching in either path only pays off once enough operations are accumulated before a request is issued, which depends on how rows are traversed within a Spark partition. Thus, within a given partition, the Bronze-to-Silver stage first collects a list of identities. It then hands them to the SortingHat requestor, which in turn groups them into batches. The requestor sends each batch as a single aliased GraphQL request to the SortingHat instance. This satisfies RQ-5.

5.2.2 Execution Pattern

The execution pattern inside the Spark DataFrame decides whether the batching described above is possible at all. Processing rows individually would cause the batching mechanism to issue a single aliased operation per request, resulting in no improvement over the old implementation. Partition-level execution hands the entire partition to the processing function at once. This allows all rows to be collected into a work-list before any HTTP call is issued. Karau et al. (2015, p. 107) note that partition-level execution is the appropriate choice whenever an operation benefits from per-partition initialization or resource allocation, such as opening a network connection. The drawback is that the partition must fit in executor

memory during batch construction. This is deemed acceptable here because the partition was already held in memory by the preceding Spark operations.

Before the DataFrame can be processed, it is coalesced to a configurable maximum number of partitions. This serves two purposes. First, it adjusts the row count per partition so that batches can reach their configured batch size. Additionally, it caps the number of executor tasks running the requestor at the same time. This limits the number of concurrent HTTP sessions opened to make calls to SortingHat.

Alternatively, a repartition could have been used, distributing the rows evenly throughout the partitions by triggering a full reshuffle. This balances the load across executors but requires an additional network transfer step. Coalesce merges adjacent partitions without a shuffle, avoiding the additional network step at the expense of potentially uneven partition sizes. For this workload, uneven partition sizes affect executor load balance but not correctness. The avoided shuffle cost is the larger factor.

Within Spark, each executor task needs the pipeline configuration in which the SortingHat endpoint is defined and a SortingHat requestor instance that issues the batched identity resolution calls. The configuration is broadcast to all executors rather than serialized into each task closure. Without broadcast, Spark transmits a separate copy of it to every executor for every task dispatch (Zaharia et al., 2010, pp. 4–5). Broadcast distributes it once per executor and caches it there, reducing per-task overhead. The requestor is instantiated once per partition rather than shared via broadcast. As opposed to the configuration, its HTTP session stores mutable state that cannot safely be shared across concurrent partitions. Instantiating it within the partition function lets it reuse the HTTP session across all batches transmitted in that partition. The partition function that Spark dispatches to executors must itself be serializable to each Python worker process. This restricts what can be captured in its closure. How this is handled is described in Section 6.

As observed in Section 3.2.2, the SortingHat requestor repartitioned the final DataFrame into a single partition after identity resolution. The partition-level execution designed in this section removes that collapse of parallelism caused by the repartition as a side effect. Together, the round-trip reduction from Section 5.2.1 and the restored partition-level parallelism are expected to deliver the stage-level duration reduction required by RQ-14.

5.2.3 Correctness Guarantee

Once identity resolution requests are batched, each individual result in the response must be put back into its original row and field in the DataFrame. A simple positional alignment of the result and the DataFrame would fail whenever rows are skipped (because it does not contain identity data) or a batch request

fails to return a result. Instead, a work-list approach was chosen that uses key-based mappings.

When a batch request fails, the design retries each item in the failed batch individually rather than aborting the partition. A fail-fast policy was considered. It simplifies error handling but would terminate a pipeline run on any transient failure, even one that occurs after hours of successful processing. Instead, best-effort was chosen. Unresolved identities produce a null value in the Silver record and the pipeline continues. As a result, downstream Gold analytics must tolerate null identity values.

The batched implementation is designed to remain fully contained within the Bronze-to-Silver transformer layer. It also does not affect the schema transformation and merge operations within the transformer, thereby satisfying RQ-6. Additionally, the correctness of the batching mechanism is verified by integration tests for the SortingHat requestor, the Bronze-to-Silver transformer and the identity handler. They test against a live SortingHat instance. The tests cover the behavior of the batching mechanism at the batch boundary, ensure that missing and partially missing identities are handled as designed and verify that the merged-identity resolution works correctly. This satisfies RQ-9. Furthermore, individual identity resolution remains available alongside the batched path. No changes are required to orchestrator definitions or the pipeline step interface, satisfying RQ-7.

5.2.4 Caching Strategy

Similar to the original identity resolution implementations, the batched optimization is designed with caching in mind. As shown in Section 3.2.2, the sequential show path does not use a class-level cache, unlike the add path. The batched design of the add and show paths addresses this by sharing the same class-level dictionary cache between the two paths.

This caching strategy is correct within a single execution because the SortingHat requestor exposes only addition and query operations and no merge mutations. The identity-to-canonical mapping cannot change during a run. In support of this, the sequential LRU cache on the pre-batched show path already relied on this same assumption. Identity merges are performed externally between runs, after which a separate orchestrator responsible for identity handling re-resolves the affected UUIDs.

6 Implementation

The architectural decisions described in Section 5 translate into two independent bodies of code. The benchmarking framework lives under `tests/performance_tests/` and is responsible for evaluating system performance. The batched identity resolution work required coordinated changes across `services/identity_management/` and `infrastructure/data_lake/`. This chapter describes what was built and discusses the challenges encountered during implementation.

6.1 Benchmarking Framework

The core building blocks of the framework were implemented following the approach described in Section 5.1.1.

6.1.1 CLI

Serving as the main entry point of the benchmarking framework is the `run_benchmarks.py` script. It handles the command line arguments and manages the subsequent execution and analysis of the benchmark. As described in Section 5.1.1, it translates the two primary actions within the framework into two separate subcommands. The *run* command is responsible for triggering the orchestration of the benchmark execution. Similarly, the *compare* command handles the selection of the two result files for comparison and the desired report formats before triggering the comparison. An additional third subcommand *trend* is used to handle the generation of the trend diagram described in Section 5.1.3, as it operates on a different input file than the comparison. None of the subcommands handle the actual logic of their responsibilities themselves. Rather, they hand it off to the respective modules, whose implementation details are described in the following. With the execution of benchmarks being the first contact point when using the framework, the next section presents its implementation and how the benchmarks' life-cycle is managed.

6.1.2 Benchmark Life-cycle

The life-cycle of the benchmark execution is designed to cover the automatic detection of the benchmarks, their execution, and the persistence of the results. A dedicated `BenchmarkRunner` class handles the first two steps. It scans each benchmark tier's directory for Python source files and uses Python's *importlib* library to import the actual module. The module is then searched for any class

that implements the `Benchmark` base class described in Section 5.1.5. To allow for further modularity in the benchmark design, it additionally excludes all classes that are marked abstract. This allows for further abstractions of the benchmarking classes past the already used base class. To start the benchmark run, the runner class iterates sequentially over all detected benchmark classes, instantiating and executing them.

Next to handling the execution, the `BenchmarkRunner` class addresses a Spark-specific issue that occurs when running the benchmarks. Each Spark session is created as a singleton per Python process, which means every benchmark uses the same Spark session for the duration of the entire benchmarking run. Should an orchestrator decide to shut down a Spark session, the session is terminated for the entire Python process. Subsequent benchmarks that use Spark would then fail, as they cannot reliably start a new Spark instance. As a consequence, the benchmark runner has to manage the Spark session's life-cycle in addition to handling the life-cycle of the individual benchmarks. It achieves this by suppressing the explicit termination of the Spark session for the duration of the benchmark run and only stopping it once all benchmarks are completed.

The modular approach described in Section 5.1.1 is supported by the previously mentioned `Benchmark` base class that serves as the basis for all benchmark implementations. It provides a single fixed method that the benchmark runner executes. This ensures the correct handling of the setup, execution, and teardown abstract methods that each benchmark implements. Within the method, a try/except/finally-block is used to guarantee execution of the teardown block and the generation of a usable result, should the benchmark fail.

The life-cycle of the benchmark execution ends with the runner collecting the results and persisting them as a JSON file. This is facilitated by a separate `MetricsCollector` class. The class exclusively handles the collection of the results and enrichment of the result data with metadata and information about the execution environment. As designed in Section 5.1.2, the result file has to adhere to a strict schema to enable subsequent comparison and analysis. The implementation of the schema is described in the following section.

6.1.3 Benchmark Metrics

The schema of the result file is defined by two Python `@dataclass` classes, `Metric` and `BenchmarkResult`. At a granular level, the `Metric` class handles the storage of an individual benchmark measurement and holds three distinct values. The recorded value of the measurement is the primary data point. It is supported by an optional unit value that can be used to describe the value in reports, for example *23.4 calls/s* for a throughput measurement. Finally, a third value is used to indicate the

direction in which the recorded measurement would improve. This information is required for the comparison of the metrics, as described in Section 5.1.2.

Once all measurements of a given benchmark are collected, they are stored in the `BenchmarkResult` dataclass. This class serves as a container for the results, as well as metadata about the benchmark run, such as the timestamp of its execution and whether it completed successfully. To signal non-comparison data that is recorded by the benchmarks to the framework, a reserved *meta* key is added to the schema. It is excluded from comparison and passed through serialization unchanged. This allows benchmarks to record cache statistics and other non-comparable values in the persisted result file, alongside their primary metrics. The separation keeps regression evaluation focused on comparable metrics, while diagnostic information remains available for analysis and debugging. How the comparison engine performs the classification of benchmark results into regression, pass, and improvement is outlined in the next section.

6.1.4 Comparison and Regression Detection

The *compare* and *trend* subcommands hand off their actions to three classes. `BenchmarkComparator` holds the implementation of the comparison engine designed in Section 5.1.3. It reads two JSON result files and a threshold configuration from a YAML file and computes the classification. The result is then dispatched to the `BenchmarkComparisonFormatter` class, which writes each requested output format to disk in a single invocation, as designed in Section 5.1.4. Output formats offered by the comparison formatter include a Markdown summary of a given benchmark comparison and a detailed Markdown report with a breakdown of all recorded and compared metrics. Additionally, a static badge can be created as a pass/fail indicator that can be used in the generated Markdown report (Shields.io, 2026). Next to the reports targeting a human audience, programmatic consumers are also supported by producing a JSON report.

For the trend visualization, the `TrendChartGenerator` handles the creation of the source code of the Mermaid XY chart. It reads the JSON lines (JSONL) history file that can be written by the comparison formatter as an additional output format and parses each line into a `TrendEntry` record. The diagram is then written to an output file, similar to the comparison formatter's output.

6.1.5 Benchmark Implementations

Building on the framework and execution model described above, this section presents the concrete benchmark implementations that serve as the inputs for the comparison engine. The primary metric all benchmarks are expected to provide is the runtime of the benchmarked subject. For this measurement, the timing function established in Section 5.1.2 is implemented as a context manager that

wraps the measuring function. By wrapping the benchmark execution with the context manager, stopping of the timer is guaranteed, even if the measured code fails unexpectedly. A total of eight benchmarks have been implemented across the three tiers. Table 6.1 summarizes their scope and the primary metrics they record.

Benchmark	**Tier**	**Scope**	**Primary metrics**
SortinghatRequestor-Benchmark	1	SortingHat requestor in isolation, covering the add and show workloads	Duration per workload, JSON web token (JWT) initialization, cache statistics (meta)
BronzeToSilverBenchmark	2	Bronze-to-Silver transformation on a synthetic in-memory dataset	Wall-clock duration, records-per-second throughput
IdentityUpdateBenchmark	2	Batched canonical identifier resolution for existing identity records	Wall-clock duration, API call count
GithubAPIExtraction-Benchmark	2	Full Bronze extraction against archived GitHub repositories	Wall-clock duration, API call count, extracted record count
GitLabAPIExtraction-Benchmark	2	Full Bronze extraction against archived GitLab repositories	Wall-clock duration, API call count, extracted record count
DevelopmentData-Benchmark	3	Full pipeline from extraction through Silver	Total duration, API calls and request time, per-source counts, and overall throughput
CodeChangeSummarizationAnalytics-Benchmark	3	Silver-to-Gold code change summarization analytics	Wall-clock duration, API call count
CostCalculationAnalyticsBenchmark	3	Silver-to-Gold cost calculation analytics	Wall-clock duration, API call count

Table 6.1: Overview of the eight implemented benchmarks across the three measurement tiers.

6.1.5.1 Tier 1: Component-Level Benchmarks

`SortinghatRequestorBenchmark` measures the SortingHat requestor in isolation from Spark and the broader pipeline, as designed in Section 5.1.5. It runs the

add workload first, followed by the show workload. According to the design, the number of identities for this benchmark should be higher than the LRU cache's limit of 1,024. Thus, as a default this benchmark uses the same 2,000 identities for each add/show workload pass. The JWT initialization is timed as a separate metric that is excluded from the workload duration. Similarly, cache statistics are recorded as non-comparable diagnostic data and excluded from regression classification. They are added to the *meta* key implemented in Section 6.1.3. To ensure each run starts from a consistent and clean SortingHat state, the test identities used are created with a unique prefix in the setup step and removed during the teardown. The removal of the identities from SortingHat is realized in a new `SortinghatRequestorWithDelete` class that subclasses the `SortingHatRequestor` used in production and adds a new method to delete identities. This enables the use of the subclass throughout the tests while keeping the deletion of identities out of production code.

6.1.5.2 Tier 2: Single-Layer Benchmarks

Tier 2 benchmarks focus on individual stages of the processing pipeline, with four benchmarks defined that target specific pipeline steps. Each benchmark declares its target stage in a configuration object specifying the class to instantiate, the method to invoke, and the arguments to supply. The shared execution helper function `run_pipeline_steps_with_metrics` then handles instantiation, timing, and output capture. Due to PySpark's lazy evaluation model, the materialization described in Section 5.1.5 is achieved by calling `df.count()` on the DataFrame after each stage.

`BronzeToSilverBenchmark` measures the Bronze-to-Silver transformation for a single data source. The input is a synthetic in-memory dataset generated at a configurable record count, so no live extraction is required. It mirrors the structure of production Bronze records. As designed in Section 5.1.5, the identity cardinality is kept low by cycling a small set of unique authors and committers across all records. Concretely, the benchmark defaults to processing 10,000 synthetic records from 20 unique authors and committers. As a result, the benchmark measures throughput at scale rather than cold-resolution latency. The primary metrics recorded are wall-clock duration and records-per-second throughput. API call counts are not available for this benchmark because identity resolution occurs within Spark worker processes, which execute in a separate context.

`IdentityUpdateBenchmark` evaluates the identity update orchestrator. The orchestrator resolves existing identity records to their current canonical identifiers using the batched lookup path introduced in Section 5.2.

`GithubAPIExtractionBenchmark` and `GitLabAPIExtractionBenchmark` measure the full Bronze extraction step against the archived repositories selected in Section 5.1.6. Metrics collected include wall-clock duration, API call count, and extracted record count. The sources covered in both benchmarks are configurable.

At the time of writing, 18 sources are configured for the GitHub extraction and 12 sources for GitLab. The GitHub security alerts and organization billing requestors do not produce usable data, because their endpoints are not accessible on third-party archived repositories. They are nevertheless included in the configured sources for completeness.

6.1.5.3 Tier 3: Multi-Layer Benchmarks

The Tier 3 benchmarks execute complete orchestrator runs across multiple pipeline layers. Three benchmarks are defined and executed against the archived repositories established in Section 5.1.6. Before each run, a clean-slate state is created by deleting all prior Delta Lake data for the benchmark project and all associated SortingHat identities. Each execution therefore starts from a consistent and isolated state.

`DevelopmentDataBenchmark` executes the full pipeline from extraction through Silver. It records total wall-clock duration, total API call count and request time, items extracted per source, and overall throughput. The per-source item breakdown allows the evaluation to confirm extraction completeness across runs. GitLab sources are excluded from this benchmark because the GitLab extraction pipelines do not return their extracted data to the orchestrator, causing the downstream Bronze writer to fail on null input. The exclusion of the GitLab sources is performed in the benchmark's configuration, as the sources covered in it are configurable. As a default, 11 GitHub sources are configured for this benchmark.

To simplify the creation of Tier 3 analytics benchmarks, the common base class `BaseAnalyticsBenchmark` is added. It provides templated benchmark execution by implementing general setup, credential validation, API counting, and teardown steps. Concrete analytics benchmarks declare only the sequence of orchestrators to execute. To add API call counting in the analytics benchmarks, a single call counter wraps the entire orchestrator sequence, providing total API usage alongside per-orchestrator breakdowns. Concretely, `CodeChangeSummarizationAnalyticsBenchmark` and `CostCalculationAnalyticsBenchmark` benchmark the code change summarization and cost calculation modules introduced in Section 3.1.4. These benchmarks measure the analytics transformation, including the impact of any external service calls invoked during processing.

To offer continuous performance feedback to developers, the implemented benchmarks need to run frequently and automatically. The following section describes the integration of the framework into the CI workflow facilitating the feedback loop.

6.1.6 CI Integration

The structure of the workflow defined in Section 5.1.8 with its five jobs is realized in GitHub Actions as follows. The run job provisions a live SortingHat instance inside the runner via Docker Compose. It polls a health endpoint of the service until it responds to ensure the service has started and that measurements are not producing errors due to failed requests. Once execution completes, the job uploads the benchmark result as an artifact with a retention period of 30 days.

The compare job downloads the result artifact, and fetches the baselines as described in Section 5.1.7. Since the standard download action does not support resolving artifacts from previous workflow executions, the *dawidd6/action-download-artifact* (Dziurla, 2024) action is used to download the two established baselines. It then produces both baseline comparisons, generates the trend chart, and uploads all comparison outputs as a second artifact.

Subsequently, presentation of the results is handled by two separate jobs. The show job writes the comparison summaries, status badges, and the trend chart to the job summary as described in Section 5.1.8. In parallel, the summarize job generates a per-benchmark duration table and appends it to the same summary page. This provides a concise overview of execution performance independent of comparison outcomes.

Baseline publication is handled by a dedicated publish job that runs only on scheduled and manually triggered workflows. The job preserves the existing historical baseline unless a manual reset is requested. It then updates the rolling baseline to the current result, and caps the accumulated trend data to a configurable window of 12 weeks by default.

6.2 SortingHat Batching

With the framework implementation established, this section describes the batching mechanisms applied to SortingHat operations to improve throughput while preserving correctness and fault isolation. The focus is on how batching is implemented and how failures are handled at runtime.

6.2.1 Batched Identity Resolution

To facilitate the batched processing of identities designed in Section 5.2, three new methods are added to the `SortingHatRequestor` class: `add_identities_batch`, `show_batched`, and `show_profile_batched`. Additionally, as defined in Section 5.2.3, the original sequential implementations have been kept and are used by the batched implementations as fallback methods, should the batching mechanism fail. The new batched methods differ from the original ones in their signature by

accepting and returning lists of identities instead of individual ones. Within each method, the identity lists are split into several chunks whose size is determined by the configured batch size. Each chunk is then translated into a single GraphQL document with one aliased mutation or query operation per identity. The default batch size is set to 100 as a compromise to keep the request size manageable while still gaining a significant increase in processed identities.

Listing 6.1 shows a sample GraphQL request and response for the add path with a batch size of three. It demonstrates the indexed translation of list items into prefixed aliases, and how the response is parsed back into an ordered list by re-mapping each alias to its source position. This batching mechanism is identical for both the add and the show paths, and is implemented in the `_execute_batch_operations` method. They only differ in the actual operation that is stated in the GraphQL document, such as `addIdentity` shown in the listing or `individuals` for lookups in the show path.

Request

```graphql
mutation {
  op_0: addIdentity(email: "alice@example.com") { uuid }
  op_1: addIdentity(email: "bob@example.com")   { uuid }
  op_2: addIdentity(email: "carol@example.com") { uuid }
}
```

Response

```json
{
  "data": {
    "op_0": { "uuid": "5b9f7a..." },
    "op_1": { "uuid": "8a4e21..." },
    "op_2": null
  },
  "errors": [
    { "message": "Could not add identity", "path": ["op_2"] }
  ]
}
```

Listing 6.1: Simplified examples of the aliased GraphQL request and response.

The new identity lists that are passed as arguments are not hashable in Python and are therefore incompatible with the LRU cache used by the sequential methods. Consequently, both batched methods rely only on the class-level dictionary cache per the strategy defined in Section 5.2.4. The cache is consulted per identity

before the request batch is built and is populated with the results returned by the service. For the two paths, the cache keys differ. The add path encodes the details of the identity in the cache key, whereas the show path uses the complete generated query for the lookup. On cache hits, the batched implementation skips the inclusion of the given item in the GraphQL document and writes it directly into the result list, so cached operations never enter a batch. Similarly, duplicate entries within the input list are excluded from the batch by mapping each unique key to a single index. Table 6.2 summarizes the resulting cache coverage across the different identity resolution operations. For each of the two paths, it compares the original sequential implementation with the new batched version. It shows that, while the fixed-size LRU caches have been dropped, the batched identity show path gained an unbounded class-level dict. Additionally, with the old implementations retained as fallbacks, the table shows that both add path implementations share the same dictionary keys. This makes them interoperable in their cache use. Should a batched request fail, the fallbacks can still access the full cache.

Operation	**Path**	**Cache**	**Key**	**Capacity**
Identity addition	Sequential	LRU + class-level dict	*(source, name, email, username)*	1,024 / unbounded
Identity addition	Batched	Class-level dict	*(source, name, email, username)*	unbounded
Identity lookup	Sequential	LRU	*uuid*	1,024
Identity lookup	Batched	Class-level dict	*(filter, query)*	unbounded

Table 6.2: Cache configuration per identity-resolution path.

6.2.2 Integration into the Pipeline

With the addition of the batched methods to the SortingHat requestor, adoption of these methods in the Bronze-to-Silver transformer required changes to the existing execution flow. Figure 6.1 contrasts the batched data flow with the original sequential implementation and highlights how identity resolution is incorporated into the transformation stage.

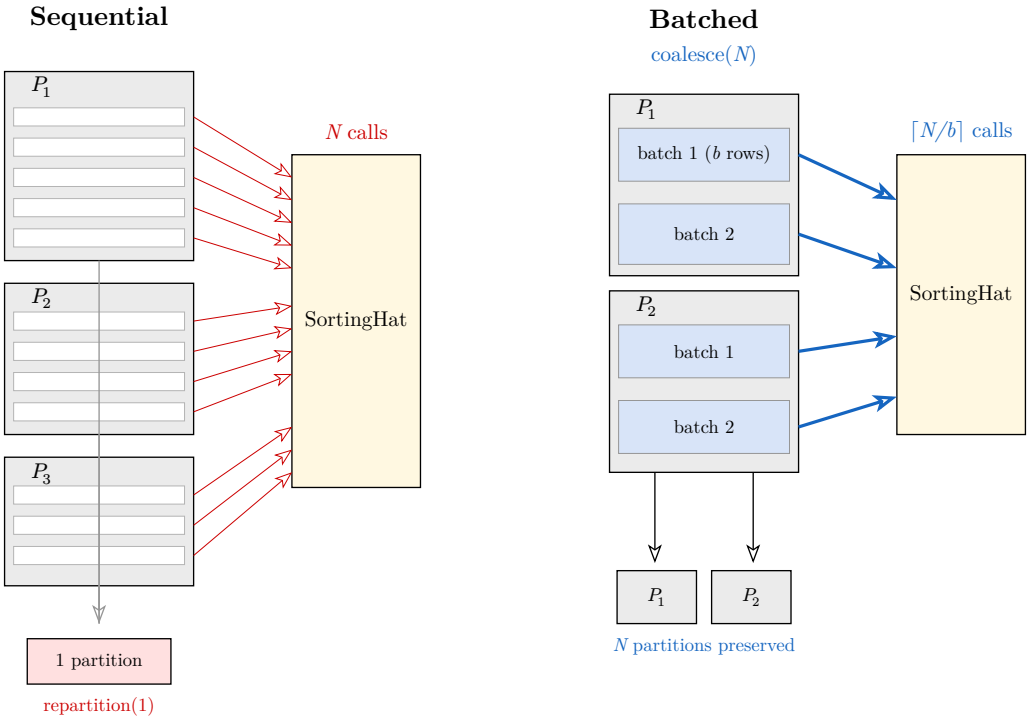

Figure 6.1: Sequential versus batched identity resolution in the Bronze-To-Silver transformer.

The goal was to introduce batching while preserving correctness, fault isolation, and compatibility with the surrounding pipeline logic. To achieve these, four implementation constraints had to be resolved, namely the partition function's serialization, the configuration distribution, concurrency control, and result mapping.

Function serialization. As noted in Section 5.2.2, the partition function must cleanly serialize to the worker process inside Spark. However, Spark distributes partition functions to worker processes using Python's pickle serializer (Apache Software Foundation, 2026; Karau et al., 2015, p. 31). This rules out functions that close over `self` or other non-serializable values, since they cannot be pickled. The partition-handling functions are therefore defined at module level with no enclosing state.

Configuration broadcast. With the partition function implemented at the module level, and the SortingHat requestor created once per partition as designed in Section 5.2.2, the configuration data is the final component needed for the batched identity resolution within Spark. The configuration data includes the complete pipeline config (carrying the SortingHat connection details), the definitions of the identity columns within each source DataFrame, and the batch size. Spark's broadcast mechanism handles this efficiently by transmitting each value once per executor and caching it locally. This avoids the repeated serialization into the task closure.

Concurrency control. Once all data is present, Spark performs the batched identity resolution within its partitions. Consequently, each partition opens an HTTP session and issues batched requests concurrently. Without setting explicit limits for the number of opened sessions, a large number of concurrent requests can overload the SortingHat service. As designed in Section 5.2, the `coalesce` operation limits the number of partitions processed before `mapPartitions` is applied, reducing the level of concurrency. This is shown on the right side of Figure 6.1. The three partitions on the left side with differing numbers of rows are coalesced into two partitions of two batches each. The maximum number of partitions, and thus parallelism, is configurable through `Sortinghat.max_parallelism` in the pipeline config. Higher values increase throughput but place more load on the service, while lower values serialize more work to reduce pressure on the backend.

Result mapping. Within the batched identity resolution, accurate mapping of DataFrame rows to the batched result is required, as outlined in Section 5.2.3. The implementation builds an explicit ordered list of entries before any HTTP call, with one entry per identity field per row. Within the list, each entry records the row position, field name, identity data, and a flag indicating whether the entry contains resolvable data. Entries without resolvable data are included in the list to preserve position alignment but excluded from the batch request. After the batch is sent to the SortingHat instance, the results are matched back to the positions on the work-list by referencing the entry index rather than the response offset. In cases where there is an array-typed identity column, the values in that column are flattened into individual entries, indexed by column and element position. After resolution, the identity values are again grouped together based on the column position. Rows with partial or null identity data are handled explicitly and mapped back to their original positions, preventing positional misalignment and keeping downstream transformations aligned at the row level.

6.2.3 Identity Update Batching

Next to the batched identity resolution in the Bronze-to-Silver transformer, the identity update path within the `IdentityHandler` requires similar adjustments to make use of the added batched methods. It reuses the same per-partition pattern as described in Section 6.2.2 with module-level functions for the partitioning, broadcast of the configuration data, controlled concurrency, and a dedicated requestor per partition. While the original sequential identity update iterated over the DataFrame per row and per column, the new implementation uses the `mapPartitions` Spark function to deduplicate the canonical UUIDs into a set of distinct entries before issuing a batched lookup call. The difference between the implementation for the identity update and the identity resolution is that the resolution requires the work-list approach to maintain correctness. In contrast, the update path input is already a single UUID per field. Consequently, deduplication

within the set of distinct entries is sufficient. After the batched lookup, the results are distributed back to the DataFrame rows through UUID equality lookups. Once `mapPartitions` completes, Spark returns an resilient distributed dataset (RDD), losing the DataFrame schema. To preserve the schema of the DataFrame throughout the identity update, the new method `_build_identity_schema` rebuilds the schema before the DataFrame is recreated from the RDD.

The migration also corrected a bug present in the sequential update path. The original UDF raised a `TypeError` when encountering null arrays in dict identity columns, whereas the `mapPartitions` implementation handles null arrays explicitly and preserves them unchanged.

6.2.4 Testing and Validation

The functional correctness validation designed in Section 5.2.3 is verified by three integration test suites for the SortingHat requestor, the Bronze-to-Silver transformer, and the identity handler. A total of 96 integration tests run against a live SortingHat instance (completing in 2 minutes 51 seconds). The test suites cover batch boundary behavior, partial and null identity data, multiple identity columns per row, variable-length array columns, and special characters in name fields. Together, they verify the three designed guarantees:

- equivalence of the batched and sequential implementation in Section 6.2.1,
- work-list order preservation in Section 6.2.2, and
- partition-level consistency when using `mapPartitions` in Section 6.2.2.

The tests also verify the null-array regression fix from Section 6.2.3. All tests skip automatically when SortingHat is not reachable, allowing the suite to run in environments without a live instance. The details of the individual tests are presented in Appendix D.

A dedicated test validates the correctness of merged identities. It manually merges two identities via SortingHat and passes the secondary identity through the full Bronze-to-Silver transformer. It then asserts that the output matches the post-merge canonical identifier. This test was added in response to a correctness issue discovered during development and serves as a regression guard for the canonical identifier resolution step.

7 Evaluation

This chapter evaluates the benchmarking framework (WP1) and the batched identity resolution (WP2) against the requirements stated in Section 4. Within the evaluation, the benchmarking framework also serves as the measurement instrument for the batched identity resolution. All performance comparisons in Section 7.3 were collected using the framework itself.

7.1 Evaluation Methodology

Four separate methods were used to evaluate the benchmarking framework. First, it was inspected against each requirement defined in Section 4, followed by measuring and evaluating the variance of the repeated benchmark runs. After that, the regression detection was validated with a controlled degradation test and finally the CI integration was assessed against the defined requirements.

WP2 was evaluated using the framework from WP1 by comparing the performance before the optimization against the batched optimization. This was performed across all three benchmark tiers. The isolated requestor measurements in Tier 1, the Bronze-to-Silver stage benchmark, and the identity update benchmark were specifically designed for this comparison. Additionally, an end-to-end pipeline measurement was used to judge the effect on an entire pipeline run.

Each local benchmark execution was performed five times on the same machine to measure run-to-run variance. Five runs are sufficient for a 90% confidence interval at $\pm$ 5% accuracy. This is demonstrated by applying the formula from Lilja (2000, pp. 52–53) for determining the number of measurements needed for the benchmark with the highest recorded CV as a worst-case scenario. Applied to the `IdentityUpdateBenchmark` with its CV of 6.4% (see Appendix C), the formula yields $n = 4.4$. Rounding it up to five confirms the number as a valid choice. The five measurements per benchmark were taken both before and after the identity resolution optimization was implemented.

One limiting factor for the measurements was identified. The API call count inside Spark worker processes cannot be recorded with the current tooling, as noted in Section 6.1.5.2. As a result, the Tier 2 and Tier 3 benchmarks do not include the requests made to the SortingHat service in the Bronze-to-Silver transformer. Only the Tier 1 benchmark is used to verify the call-count reductions. All local benchmarks were executed on a 12-core CPU, 23 GB RAM system running Python 3.11.12 on Linux 6.19.

Tier 2 and Tier 3 benchmarks that involve live API extraction run against two archived repositories: *containrrr/watchtower* on GitHub and *gitterHQ/gitter-android-app* on GitLab. As established in Section 5.1.6, archiving freezes the repository state, so successive runs fetch identical item counts. Thus, timing differences between runs on the archived repositories reflect implementation changes instead of a changing dataset.

To measure the combined Tier 1 and Tier 2 execution time on the default GitHub Actions runner, as required by RQ-13, the CI workflow was triggered via manual dispatch. RQ-9 is verified by the integration test suite described in Section 6.2.4. The remaining requirements were then validated by priority class. Must-Have requirements were assessed by direct measurement against the stated quantitative targets and by inspection or integration tests against the stated qualitative criteria. Should-Have requirements that missed their target were defended by a combination of measurement and qualitative argument. Could-Have requirements were validated by inspection of the shipped artifact.

7.2 Benchmarking Framework Effectiveness

Before the framework is used to measure the WP2 optimization, it must itself meet the requirements set out for WP1. The content below covers usability, reproducibility, regression detection, and the CI integration in turn.

7.2.1 Usability and Extensibility

RQ-1 requires benchmarks at three granularity tiers. Eight benchmarks were implemented, as shown in Table 6.1, and each completed the five-run measurement successfully.

RQ-10 requires a base class that encapsulates the benchmark life-cycle. As described in Section 6.1.2, the base class enforces the setup/run/teardown sequence and executes teardown inside a `finally` block, ensuring cleanup runs even when a benchmark fails. Should a benchmark fail, a failure result is generated instead of an exception. This ensures that the complete benchmarking run is not aborted prematurely.

RQ-11 requires that adding a benchmark requires no changes to the base class, the CI workflow, or the result storage schema. As described in Section 6.1.2, the framework scans tier directories for `Benchmark` subclasses at startup. Thus, a new benchmark only needs to implement this base class in the appropriate tier directory.

A visual representation of the benchmark health over time is required in RQ-16. The *trend* subcommand described in Section 6.1.4 reads the JSONL history artifact and creates a Mermaid XY chart. It displays one line for the historic

benchmark comparison and one for the rolling one. Each shows the aggregate health score across a configurable number of benchmark runs. By inspecting the output of the subcommand against the JSONL history file from the comparison job, the requirement was verified.

7.2.2 Reproducibility Analysis

RQ-4 requires benchmarks to run against live service instances. Each benchmark that exercises identity resolution contacted a live SortingHat instance that was provisioned via Docker Compose. The connection to SortingHat is confirmed in the Tier 1 benchmark by the JWT token generation time that is recorded separately. The Tier 2 extraction benchmarks running against live GitHub and GitLab APIs verify live service contact by counting the API calls for each step in the generated result file. Similarly, Tier 3 benchmarks also record the API calls to the external services. Thus, no mocks were used in any timing path.

Persistent baseline storage is required in RQ-2. This was verified in the manually triggered CI workflow runs in the GitHub repository. The CI run produced three artifact types: the raw result file with a 30-day retention window, the comparison outputs, and a baselines artifact. This artifact contains the historical baseline, the rolling baseline, and the trend data with a 90-day retention window. The historical baseline was established on the first CI run and the rolling baseline was updated on the subsequent run, both confirmed by the artifact contents.

To judge whether the implemented framework and benchmarks are stable enough to support before/after comparisons, the five-run measurements before and after the WP2 optimization were evaluated. Table 7.1 shows the mean, standard deviation, and CV across five runs for the benchmarks used in Section 7.3. The full table with the data for all eight benchmarks is listed in Table C.6 in the appendix. With the standard deviation and CV defined in Section 3.3.1 quantifying the spread and normalizing it to a dimensionless ratio, the executed benchmarks can be compared on a common scale. Since API counts are deterministic, every run before and after the optimization returns the same count, causing the standard deviation to be zero. The durations varied more, but the CV stayed within single-digit percentages across all runs. The highest observed value was 6.4% on the batched `IdentityUpdateBenchmark`, corresponding to a standard deviation of 0.43 s on a 6.82 s mean. Across all measured benchmarks the CV stayed below 6.5%, confirming that the framework produces measurements stable enough for reliable before/after comparison.

Benchmark	Tier	Before/After WP2 optimization	Metric	Mean ± SD	CV (%)
SortinghatRequestorBenchmark	1	before	add duration	35.85 ± 1.54 s	4.3
SortinghatRequestorBenchmark	1	after	add duration	9.17 ± 0.14 s	1.5
SortinghatRequestorBenchmark	1	before	show duration	50.35 ± 1.16 s	2.3
SortinghatRequestorBenchmark	1	after	show duration	3.89 ± 0.11 s	2.8
BronzeToSilverBenchmark	2	before	wall-clock duration	14.48 ± 0.44 s	3.0
BronzeToSilverBenchmark	2	after	wall-clock duration	4.94 ± 0.05 s	0.9
IdentityUpdateBenchmark	2	before	wall-clock duration	7.76 ± 0.15 s	1.9
IdentityUpdateBenchmark	2	after	wall-clock duration	6.82 ± 0.43 s	6.4

Table 7.1: Mean, standard deviation, and CV across five runs before and after WP2 optimization.

7.2.3 Regression Detection Validation

RQ-3 requires that deviations from the baseline exceeding the configured threshold are classified as regressions. To confirm this, two tests were conducted. One was designed as a true-positive test with a deliberately degraded configuration, the other as a false-positive test with an identical repeated run.

True-positive test. To force a deliberate degradation of a given benchmark, the optimized variant of the `SortinghatRequestorBenchmark` was chosen. Its `batch_size` configuration parameter controls how many identities are combined

into one GraphQL alias request. At the configured value of `batch_size=100`, the benchmark issues $\lceil \frac{N}{100} \rceil$ requests for N identities. Setting the batch size to one forces a single request per identity, recovering the $O(N)$ call behavior of the individual implementation. Thus, with a baseline that uses the chosen value, every metric that is sensitive to the recorded call count would deviate from the baseline by an amount that far exceeds any reasonable threshold. As shown in Table 7.2, all metrics were detected as regressions.

Metric	**Baseline**	**Degraded**	**Change**	**Result**
add_total_api_calls	20	2,000	+9,900%	Regression
show_total_api_calls	20	2,000	+9,900%	Regression
add_duration_seconds	9.22 s	30.50 s	+230.7%	Regression
show_duration_seconds	3.90 s	19.27 s	+394.1%	Regression
duration_seconds	13.12 s	49.77 s	+279.2%	Regression
add_operations_per_second	433.7 ops/s	131.1 ops/s	− 69.8%	Regression
show_operations_per_second	1,025.7 ops/s	207.6 ops/s	− 79.8%	Regression

Table 7.2: True-positive regression test with `batch_size=100` baseline and `batch_size=1` degraded run.

False-positive test. A false-positive test checks that repeated runs do not trigger a regression. To accomplish this, the same benchmark was run twice with identical configuration. The first run served as the baseline and the second run was compared against it. Table 7.3 shows that no metric was classified as a regression.

Metric	**Change**	**Threshold**	**Result**
add_total_api_calls	0.00%	5%	Pass
show_total_api_calls	0.00%	5%	Pass
add_duration_seconds	+5.07%	20%	Pass
add_operations_per_second	− 4.83%	20%	Pass
show_duration_seconds	+0.36%	20%	Pass
show_operations_per_second	− 0.36%	20%	Pass
duration_seconds	+3.67%	20%	Pass

Table 7.3: False-positive test across two consecutive runs with identical configuration.

The `add_duration_seconds` metric varied by 5.07% between the two identical runs. This exceeds the 5% fallback threshold set for all benchmarks but falls within the 20% per-metric threshold configured for `SortinghatRequestorBenchmark`. The

workload's short duration of 9 s can amplify small variations in the requests to the live SortingHat service into a larger relative change. Without per-metric threshold overrides this run would have been a false positive. This test confirms that a global threshold value is not enough to handle benchmarks with higher natural variance and that per-metric threshold overrides are needed. The two tests confirm that RQ-3 is fulfilled. A metric that exceeds its threshold is detected as a regression, and a metric that does not change does not trigger false regression classifications.

7.2.4 CI Integration Assessment

RQ-12 requires the automatic execution on a schedule and reporting of the comparison. As described in Section 6.1.6, the CI workflow runs on a weekly schedule. It runs all benchmarks from the three tiers and adds the resulting comparison together with a summary of the run's results to the job summary page in the Actions run view. The baseline creation is gated by the scheduled run and the manual CI trigger. Therefore, optional pull-request executions are able to compare only to the most recent baselines. This gating of baseline updates prevents individual developers from overwriting the original baselines created as reference points. Consequently, all decisions regarding what artifacts are in the baseline are made solely by the maintainers. Three artifact types were confirmed to have been generated during the CI execution: the unaltered raw result file, the comparative output files, and the baselines artifact.

RQ-13 requires Tier 1 and Tier 2 benchmarks to complete within 20 minutes on an `ubuntu-latest` runner. Table 7.4 shows the measured execution time for all eight benchmarks on a standard GitHub Actions runner with a 2-core CPU and 7.75 GB RAM. The Tier 1 and Tier 2 total was 2,799.8 seconds (46.7 minutes), which exceeds the 20-minute limit.

Benchmark	Tier	Duration	Extraction API calls
SortinghatRequestorBenchmark	1	23.4 s	—
IdentityUpdateBenchmark	2	12.9 s	—
BronzeToSilverBenchmark	2	13.1 s	—
GitLabAPIExtractionBenchmark	2	1,164.4 s	6,538
GithubAPIExtractionBenchmark	2	1,586.0 s	4,841
Tier 1 + Tier 2 total	—	**2,799.8 s**	—
DevelopmentDataBenchmark	3	1,290.5 s	4,800
CostCalculationAnalyticsBenchmark	3	781.7 s	700
CodeChangeSummarizationAnalyticsBenchmark	3	failed	—
All tiers total	—	**4,872.1 s**	—

Table 7.4: Benchmark execution times and API call counts on a GitHub Actions runner with a clean rate limit budget.

The non-extraction benchmarks (`SortinghatRequestorBenchmark`, `BronzeToSilverBenchmark`, and `IdentityUpdateBenchmark`) completed in 49.4 seconds combined, whereas the extraction benchmarks accounted for the remaining 2,750.4 seconds of the Tier 1 and Tier 2 total. Looking at the GitLab extraction benchmark alone shows it already nearly exhausted the budget afforded by RQ-13.

The 20-minute requirement was set to accommodate network latencies and startup times of Spark and the JVM. With the chosen repositories for GitHub and GitLab generating more than enough extraction volume to exhaust the budget, RQ-13 could not be fulfilled.

Beyond the time limit, Table 7.4 notably shows the `CodeChangeSummarizationAnalyticsBenchmark` as failing. An analysis of the run logs shows that it fails with a *403 Forbidden* error due to rate-limit exhaustion. Similar to the time limit exhaustion, this is traceable to the extraction repository selection. Four benchmarks issue GitHub REST API calls against the same repository using a single authentication token. In the observed CI runs (Appendix B), the first three benchmarks extracting GitHub data consumed over 10,000 REST calls before `CodeChangeSummarizationAnalyticsBenchmark` started. This is over twice the 5,000 requests per hour rate limit described in Section 3.2.4. Thus, `CodeChangeSummarizationAnalyticsBenchmark`, which is the last benchmark executed, consistently fails because the rate limit is already exhausted. When its extraction fails, the downstream pipeline stages also fail because of missing Delta Lake tables or unresolved DataFrame columns, since the expected Silver-layer data was never written. The benchmarks share a single authentication token without quota coordination, so the failure is of a structural nature.

The repository selection criteria defined in Section 5.1.6 favored larger repositories, whose size clashes with the rate limits set by GitHub and GitLab. Three mitigations were considered but not implemented in this thesis.

The first is a purpose-built repository owned by the project on both platforms. Commits, contributors, issues, and other artifacts are specifically sized to cross the first page boundary on each endpoint without producing excessive volume. The number of API calls per extraction would drop accordingly. Measurements would still be valid because extraction would still use a real repository from a real platform. The drawback is that the content of the repository has to be crafted carefully to reflect real repository data and maintained across every extraction endpoint. This causes significant engineering effort on its own.

Another option is using extraction limits for each source at the framework level. By capping the number of API calls per extraction, the total amount of extraction can be carefully controlled to stay within the rate-limit budget. This approach does require modifications to the existing extractors to strictly observe extraction limits. A configuration-level approach was attempted but did not work reliably.

Lastly, making each benchmark explicitly aware of the available request budget is another way to avoid running into rate limits. Benchmarks can query GitHub for the rate-limit information and wait until sufficient budget is available. This would prevent the specific cascading failures described above at the cost of extending the total runtime of a benchmarking job to potentially several hours.

The rate limit exhaustion also exposed that the extraction layer does not handle rate limiting robustly. When a request fails due to an exhausted rate limit, the extractor treats it as a generic request failure and returns empty data rather than waiting for the limit to reset and retrying. The pipeline continues with the empty result, producing cascading failures downstream. Missing Silver-layer tables and unresolved DataFrame columns are consequences of silently lost extraction data, not of the rate limit itself. This failure is incidental. No benchmark was designed to trigger rate limiting, yet the combination of repository size and benchmark count created the conditions for it. The finding suggests that a dedicated benchmark targeting the rate limit boundary with a carefully chosen repository would have diagnostic value for validating extraction robustness, not only performance.

7.3 SortingHat Optimization Results

This section covers the before/after comparison for WP2 using the benchmarking framework from WP1. Four benchmarks were implemented in Section 6.1.5 to cover different scopes. The `SortinghatRequestorBenchmark` (Tier 1) contacts SortingHat directly without any Spark involvement and measures API call counts and the duration directly. `BronzeToSilverBenchmark` (Tier 2) measures the impact on the full Bronze-to-Silver stage and `DevelopmentDataBenchmark` (Tier 3) measures

the end-to-end pipeline effect. `IdentityUpdateBenchmark` (Tier 2) exercises the batched canonical identifier resolution path of the `IdentityHandler`.

7.3.1 Sequential Baseline

The sequential baseline was measured across five runs using the configuration without the batched identity resolution. Following Section 3.3.1, the execution times use the arithmetic mean, whereas the throughput rates apply the harmonic mean and are calculated as the total work divided by the mean duration. Table 7.5 reports the results alongside the batched measurements.

The sequential `SortinghatRequestorBenchmark` made 2,000 API calls for the add workload and 4,000 for the show workload. Each workload makes two passes over a synthetic identity set, designed in Section 5.1.5 to exercise the caches on the second pass. This causes the asymmetry between the show and the add calls, as analyzed in Section 3.2.2.

7.3.2 Batched Performance

All measurements were taken across five runs at `batch_size=100` using the batched configuration.

Benchmark	**Metric**	**Sequential**	**Batched**	**Change**
SortinghatRequestorBenchmark	add API calls	2,000	20	− 99.0%
SortinghatRequestorBenchmark	show API calls	4,000	20	− 99.5%
SortinghatRequestorBenchmark	add duration	35.85 s	9.17 s ± 0.14	− 74.4% (3.9×)
SortinghatRequestorBenchmark	show duration	50.35 s	3.89 s ± 0.11	− 92.3% (12.9×)
SortinghatRequestorBenchmark	total duration	86.20 s	13.06 s ± 0.24	− 84.8% (6.6×)
BronzeToSilverBenchmark	duration	14.48 s	4.94 s ± 0.05	− 65.9% (2.93×)

Benchmark	Metric	Sequential	Batched	Change
BronzeToSilverBenchmark	throughput	691 rec/s	2,024 rec/s ± 19	+193%
IdentityUpdateBenchmark	duration	7.76 s	6.82 s ± 0.43	− 12.1% (1.14×)
IdentityUpdateBenchmark	throughput	129 id/s	147 id/s ± 8.9	+14.1%

Table 7.5: Sequential versus batched results across five runs per configuration.

HTTP request reduction (RQ-8). At a configured batch size of 100 with 2,000 identities, the batched implementation issued 20 API calls for the add workload and 20 for the show workload. The sequential implementation issued 2,000 calls for add and 4,000 for show. The reductions are of factor 100 (99%) and 200 (99.5%) respectively. With RQ-8 requiring a reduction by a factor of at least the batch size chosen above, the measured reductions meet or exceed that. This realizes the field-aliased GraphQL batching mechanism specified by RQ-5.

To verify that the call count scales as $\lceil \frac{N}{b} \rceil$, the regression validation run with the batch size of one was used to compare with the sequential baseline. Both produced 2,000 calls. For the add path this matches the sequential baseline's count. In the case of the show path the count halves because of the dictionary cache added by the batched implementation in Section 5.2.4. This confirms the scaling relationship and that batch size reduces call count proportionally. The same contrast exercises the runtime-configurable `batch_size` parameter required by RQ-15. No code changes were required between the $b = 1$ regression run (Table 7.2) and the $b = 100$ measurements above.

The larger speedup of the show workload (12.9×) versus the add workload (3.9×) is caused by the reworked caching described above. Both batched workloads make 20 calls to the SortingHat service, so the difference comes from the asymmetry of the sequential paths issuing 2,000 calls for the add path and 4,000 for the show path. Had the batched show path's dictionary cache been omitted, the optimized path would issue 40 requests and the speedup would drop by roughly half to 6–7×. Thus, the measured 12.9× speedup reflects the batching reduction and the added dictionary cache.

Bronze-to-Silver speedup (RQ-14). The `BronzeToSilverBenchmark` is the primary benchmark concerned with the Bronze-to-Silver improvement. It processes the synthetic workload designed in Section 5.1.5 with a low identity cardinality across the dataset. With the duration reduced from 14.48 s to 4.94 s, the

batched implementation speeds up the transformation by 2.93×. Similarly, the throughput increased from 691 to 2,024 records per second. Since the LRU cache in the sequential implementation is able to handle the 20 individual identities of the benchmark, the performance gains are not directly attributable to the aliased batching mechanism. The larger contributions to the speedup are more likely to come from the removal of the `repartition(1)` pattern that restored the partition-level parallelism and the elimination of the per-row overhead through `mapPartitions`. However, their individual impacts on the speedup were not separately measured.

IdentityUpdate. The `IdentityUpdateBenchmark` measures the identity update path, which resolves existing Silver-layer records to their canonical identifiers after SortingHat merge operations. Duration fell from 7.76 s to 6.82 s (1.14×). The small improvement is explained by the LRU cache in the sequential UDF path, which already deduplicates repeated lookups within a run. The main benefit of the `mapPartitions` migration is the correction of the null-array `TypeError` described in Section 6.2.3, with performance improvement as a secondary outcome.

Inspection of the WP2 implementation shows that no changes to orchestrator or pipeline-step interfaces were required. This satisfies RQ-7. Integration tests against a live SortingHat instance confirm that the batched path produces identical canonical UUIDs, schema, and merge semantics as the sequential baseline. The correctness design is specified in Section 5.2.3 and verified by the test suite in Section 6.2.4, satisfying RQ-6.

7.3.3 End-to-End Pipeline Impact

The `DevelopmentDataBenchmark` executes the full pipeline from extraction to Silver against 11 GitHub sources from the *containrrr/watchtower* repository (see Section 6.1.5.3). Table 7.6 reports the results across five runs before and after the optimization.

Metric	**Sequential**	**Batched**	**Change**
Total duration	2,181.5 s ± 40.7	2,047.6 s ± 32.3	− 6.1% (1.07×)
Throughput	7.56 items/s	8.05 items/s	+6.5%
Total API calls	5,707	5,633	− 1.3%
GET calls (extraction)	5,663	5,599	− 1.1%

Table 7.6: End-to-end pipeline results across five runs per configuration.

It shows that the optimized identity resolution improves the pipeline duration by 6.1% for the development data extraction. API calls are recorded for this benchmark, but they do not include calls from the Bronze-to-Silver transformation's

identity resolution, which runs inside a Spark worker process. The recorded call volume does show that the pipeline is dominated by extraction *GET* requests. Thus, resolving the identity resolution bottleneck only produces a small end-to-end improvement.

The `CodeChangeSummarizationAnalyticsBenchmark` and `CostCalculationAnalyticsBenchmark` showed no meaningful change (− 1.4% and − 0.1% respectively, see Appendix C). Since the `CostCalculationAnalyticsBenchmark` does not call SortingHat, this is expected. While `CodeChangeSummarizationAnalyticsBenchmark` does (see Section 3.2.3), its SortingHat call volume is a small fraction of the total benchmark time. Given this and extraction dominating the Tier 3 runtime (as established by the `DevelopmentDataBenchmark`), the SortingHat improvement is negligible for this benchmark.

7.4 Requirements Validation

The requirements in Section 4 state 9 Must-Have, 6 Should-Have, 1 Could-Have, and 3 Won't-Have criteria. All in-scope requirements were verified against both work packages, and the evaluation found only the Should-Have requirement RQ-13 unfulfilled. The three Won't-Haves (incremental delta extraction, source-level parallelism across orchestrators, and intelligent orchestrator triggering) are out of scope for this thesis and are therefore not evaluated. They are revisited as future work in Section 8.

Table 7.7 summarizes the verification status of each requirement, sorted by MoSCoW priority and then by RQ number. Each requirement is traced to the section or artifact that provides the evidence.

Req.	**Priority**	**WP**	**Evidence**	**Status**
RQ-1	Must	WP1	Eight benchmarks across three tiers implemented and executed without failure	Fulfilled
RQ-2	Must	WP1	Historical and rolling baselines published as GitHub Actions artifacts after each scheduled run	Fulfilled
RQ-3	Must	WP1	7/7 regressions detected in true-positive test and 0/7 false positives in false-positive test (Table 7.2, Table 7.3)	Fulfilled
RQ-4	Must	WP1	JWT timing in result files confirms live SortingHat contact in all Tier 1 runs and API call counts confirm live GitHub and GitLab contact in extraction benchmarks	Fulfilled

Req.	Priority	WP	Evidence	Status
RQ-5	Must	WP2	SortingHat requestor issues batched GraphQL requests through aliasing of a configurable batch size (see Section 6.2.1)	Fulfilled
RQ-6	Must	WP2	Integration tests against live SortingHat verify correct UUID resolution, schema preservation, batch boundary handling, null/partial data, and merged-identity resolution	Fulfilled
RQ-7	Must	WP2	No changes to calling orchestrators or pipeline step definitions. Verified by inspection	Fulfilled
RQ-8	Must	WP2	Target: reduction by factor $\geq b$ ($b = 100$ in evaluation). Result: 100× add (2,000 → 20), 200× show (4,000 → 20, including dictionary cache on second pass) (Table 7.5)	Fulfilled
RQ-9	Must	WP2	96 integration tests passed against a live SortingHat instance (see Section 6.2.4)	Fulfilled
RQ-10	Should	WP1	Setup/run/teardown life-cycle enforced. The uniform contract enables CI invocation without per-benchmark changes. Verified by inspection	Fulfilled
RQ-11	Should	WP1	Verified by inspection; described in Section 6	Fulfilled
RQ-12	Should	WP1	Weekly CI schedule with PR opt-in, comparison summaries published to job summary and baseline publication gated to scheduled runs	Fulfilled
RQ-13	Should	WP1	Target: Tier 1+2 ≤ 20 min on ubuntu-latest. Result: 46.7 min measured (Table 7.4)	Not met
RQ-14	Should	WP2	Target: Bronze-to-Silver duration reduction (no numeric target). Result: 2.93× reduction (14.48 s → 4.94 s); see Section 7.3.2 for IdentityUpdate and end-to-end measurements as supporting context	Fulfilled
RQ-15	Should	WP2	Configurable *batch_size* parameter exposed in the SortingHat pipeline config (see Section 5.2); runtime configurability confirmed by the call-count scaling in Table 7.2	Fulfilled

Req.	Priority	WP	Evidence	Status
RQ-16	Could	WP1	Mermaid trend chart generated from accumulated comparison history; verified by inspection of *trend* subcommand output	Fulfilled

Table 7.7: Requirement verification summary.

RQ-13 is the only requirement not met. As a Should-Have requirement it was expected to be attainable, so the miss is informative rather than contradictory to the thesis contributions. The non-extraction benchmarks completed in 49.4 seconds combined, well within the budget. The GitLab extraction benchmark alone ran for 19.4 minutes, and the combined extraction benchmarks pushed the total to 46.7 minutes. The 20-minute target was established to accommodate JVM startup, Spark initialization, and live API latency. It did not account for the extraction volume of the archived repositories. The repository selection criteria for reproducibility (see Section 5.1.6) favored large archived repositories, which outweighed the runner-time target. The root cause and three considered mitigations are discussed in Section 7.2.4.

7.5 Threats to Validity

Internal validity. The principal threat to the internal validity is the load on the test machine and the latency of the network between individual runs, since measuring the duration of a benchmark is sensitive to both. Section 7.2.2 uses central tendency and dispersion measures to show that the recorded benchmarks are stable. Beyond that, running the SortingHat service via Docker Compose and ensuring it is running before starting the benchmarks removes its startup as a source of timing differences. The primary source for internal variation is the live-service HTTP latency to SortingHat and GitHub/GitLab with a proportionally larger effect on short benchmarks. This is shown in Section 7.3.2.

External validity. As opposed to the internal validity, the external validity is concerned with two aspects. Firstly, are the results valid beyond the MECOIS project, and secondly, do they generalize beyond the workload shape used for the evaluation. Answering these questions requires separating the two classes of claim the evaluation produces, since each has different generalization properties.

Along the workload axis, RQ-8′s call-count reduction is algorithmic and holds for any workload with N distinct identities. It reflects the $O\left(\lceil \frac{N}{b} \rceil\right)$ cost of batched resolution against the $O(N)$ cost of individual requests. The duration speedups from RQ-14 on the other hand are dependent on the workload and track network round-trips, LRU cache hit rates, and partition-level execution. At the cardinality used by the `BronzeToSilverBenchmark`, the sequential LRU cache already absorbs

most repeated lookups. This reduces the measured 2.93× relative to a workload that overflows the cache. The Tier 1 `SortinghatRequestorBenchmark` runs at 2,000 identities, exceeds the cache, and reports 3.9× and 12.9× duration reductions (Table 7.5). Higher-cardinality deployments should therefore see speedups at least as large as those measured at Tier 2.

Along the deployment axis, MECOIS is the only system under test. The archived repositories provide representative API surface coverage but do not reflect the growth patterns of active repositories. Identity field densities and SortingHat instance capacities are those of the MECOIS deployment, so pipelines configured differently may observe different RQ-14 magnitudes, though RQ-8′s algorithmic claim still holds by construction.

Construct validity. Construct validity asks whether each measurement reflects what its requirement says it should measure. In the case of RQ-8, HTTP call count is exactly the quantity that is named by the requirement, so the construct matches directly. The duration of a benchmark is an appropriate construct for RQ-14 because network round-trips are the dominant cost of both the sequential and batched identity resolution paths. A single measurement tier, however, understates the construct. While the Tier 1 benchmark uses 2,000 identities with the cache overflowing, the low number of identities in the `BronzeToSilverBenchmark` (20 unique identities per 10,000 records) means the sequential LRU cache absorbs most repeated lookups. This does constrain the measurable speedup compared to a workload without caching. Consequently, the Tier 1 benchmark measures the network round-trip reduction and Tier 2 the partition-level execution improvements. Hence, the two tiers capture both mechanisms the requirement covers. CI execution time is the construct specified in RQ-13 and is measured directly from the run timestamps.

Reliability. Reliability looks at whether repeated execution of the same evaluation gives equivalent results. The reproducibility of the benchmark environment is given since its code, configuration, and CI workflow are version-controlled, so the benchmarks can be rerun from any commit. The CI workflow provisions SortingHat via Docker Compose and polls the health endpoint before benchmarks start. This ensures a repeatable state at every benchmark run. Similarly, the archived repositories provide identical datasets on every run. The exception to the reproducibility is the use of live GitHub and GitLab endpoints. Their response times depend on the server-side load at the time of the run. The low CV values observed for extraction benchmarks (2.1– 4.0%) suggest this source of variance was small during the evaluation window. Larger variance cannot be ruled out under different server load conditions. A second caveat concerns the call-count limitation in Tier 2 and Tier 3 benchmarks noted in Section 7.1. The 99% call reduction established by the Tier 1 benchmark cannot be independently confirmed at either Tier 2 or Tier 3 without additional instrumentation.

8 Conclusions

This thesis introduced a performance benchmarking framework for the MECOIS pipeline and demonstrated it by optimizing the Bronze-to-Silver identity resolution. Beyond this demonstration, the framework continues to detect regressions automatically in CI. Its measurements also revealed that extraction, not identity resolution, dominates pipeline runtime. The following sections summarize the contributions, present the key findings, discuss limitations, and outline directions for future work.

8.1 Summary of Contributions

The developed performance benchmarking framework successfully addresses the measurement gap. It provides a flexible environment for the development of benchmarks and their execution. By running it on a weekly schedule within the project's CI, newly introduced performance regressions can be identified and addressed in a timely manner. Benchmark results are compared to the previous run and a historical baseline, to surface immediate issues and slow degradations over time. The three-tier system used to differentiate benchmark types serves well for locating a potential regression. If one occurs inside an individual system, such as identity resolution, a Tier 1 benchmark shows the regression. A regression that only appears in a specific pipeline step, like the Bronze-to-Silver medallion layer transition, might be missed by a Tier 1 benchmark. However, a Tier 2 benchmark designed for the specific step catches it. Lastly, Tier 3 serves to highlight the end-to-end pipeline performance. It is used for complete extraction and analytical runs and surfaces significant regressions that impact the entire pipeline. In total, eight benchmarks have been implemented across the three tiers (Table 8.1). Adding new benchmarks requires only a new Python module with no changes to the existing infrastructure.

Tier	**Benchmarks**
Tier 1 (component-level)	SortinghatRequestorBenchmark
Tier 2 (single-layer)	BronzeToSilverBenchmark, IdentityUpdateBenchmark, GithubAPIExtractionBenchmark, GitLabAPIExtractionBenchmark

Tier	Benchmarks
Tier 3 (multi-layer)	DevelopmentDataBenchmark, CostCalculationAnalyticsBenchmark, CodeChangeSummarizationAnalyticsBenchmark

Table 8.1: Implemented benchmarks grouped by tier.

Of the candidates identified in Section 3.2.5, the identity resolution mechanism was selected as the most immediate improvement opportunity. Located within the Bronze-to-Silver transition and using the external SortingHat system, the mechanism inefficiently used API requests to communicate with SortingHat. It issued at least one HTTP request per developer identity per row in a DataFrame. The thesis replaced this with batched GraphQL alias requests that group identities into configurable-size batches that are resolved with a single API call. Changing the Spark partitioning scheme also contributed to addressing the identity resolution bottleneck. Previously, it collapsed all partitions into a single one and thus reduced the available parallelism. The thesis implementation reestablished Spark parallelism by mapping the identity resolution results onto multiple partitions which Spark can execute in parallel.

8.2 Key Findings

Using the developed framework to benchmark the optimization of the identity resolution showed a reduction of the API call count by 99.0% and a 6.6× speedup at the Tier 1 component level. Similarly, the optimizations made to the Spark partitioning, combined with the batching mechanism, improved the Bronze-to-Silver transition by 2.93×. End-to-end performance, however, was not significantly improved by the optimizations. Benchmarks measuring the full extraction pipeline up to the Silver medallion layer and select analytics pipelines showed only marginal improvements. The largest is 6.1% for the development data extraction benchmark. This highlights the extraction process as the dominant pipeline cost, whereas identity resolution plays only a small part overall.

Of the in-scope requirements, only the Should-Have RQ-13 was not met. All others were achieved. RQ-13 allotted a 20-minute runtime budget in the CI workflow for the implemented Tier 1 and Tier 2 benchmarks. The actual runtime came to 46.7 minutes, exceeding the budget. The GitHub and GitLab extraction benchmarks dominated this total at 26.4 and 19.4 minutes, respectively (see Appendix B).

8.3 Limitations

Three limitations qualify the reported results.

Repository selection imposes the largest constraint. Large archived repositories were chosen for full source and pagination coverage. They drove most of the Tier 2 and Tier 3 runtimes and exhausted the RQ-13 time budget. Absolute timings would change for different repositories, though the algorithmic call-count reduction (RQ-8) holds by construction.

The second limitation is the synthetic data used in the Tier 1 and Tier 2 identity benchmarks. Tier 1 deliberately overflows the LRU cache to expose network round-trip cost. Tier 2 stays below it to expose partition-level execution gains. Production workloads at intermediate cardinalities may yield different speedup ratios.

A final limitation is run-to-run variance. Five runs per benchmark were used to reduce it. The coefficient of variation stayed below 6.5% on the local evaluation machine but cannot be guaranteed for production CI execution.

8.4 Future Work

Several open opportunities follow from the analysis. The most direct target is the extraction cost to the pipeline. As with the optimization of the identity resolution, the sequential pagination in the extraction steps could be improved by combining GraphQL with REST requests to distribute requests across separate rate-limit budgets. Similarly, adding asynchronous concurrent requests would reduce the time spent making network requests. On a higher level, by limiting what gets extracted at the orchestrator level and selecting only the required data for a given analytics job, the total amount of data could be reduced significantly. In the same direction, avoiding the re-extraction of already present data can further reduce the extraction volume. However, this requires careful calibration of the collected development data, due to the mutability of git commits from force-pushes. Regarding the performance benchmarking framework, extending the coverage provided by its benchmarks, like covering more Gold-layer analytics, remains open and is directly supported by the framework design. Finally, the RQ-13 miss identified three mitigation options to address the long Tier 2 and Tier 3 extraction times. A purpose-built and smaller repository specifically crafted for use in the benchmarks, per-source item limits applied during the benchmark extraction, and scheduling the extractions in a rate-limit-aware fashion can independently shorten the benchmark duration.

Appendices

A Benchmark Repository Extraction Coverage

The following tables list every standalone extraction requestor in the pipeline source modules for each platform, together with whether it is included in the benchmark configuration. Notes record the reason for any exclusion or for an included requestor returning no data on the benchmark target. Enrichment steps that operate on already-fetched data (default branch requestors) and transformer modules are not extraction requestors and are omitted.

Data type	**Included**	**Note**
Commits	Yes	
Commit details	Yes	
Commit details (log format)	Yes	
Commit statuses	Yes	
Pull requests	Yes	
Pull request details	Yes	
Issues	Yes	
Issue events	Yes	
Repository events	Yes	
Workflow runs	Yes	
Workflow jobs	Yes	
Environments	Yes	
Deployments	Yes	
Releases	Yes	
Software bill of materials	Yes	
Code information	Yes	
Security alerts	Yes	Returns no data without repository ownership
Organization billing	Yes	Returns no data without organization admin access

Table A.2: GitHub extraction coverage for *containrrr/watchtower*. All standalone extraction requestors in `sources/github/` are listed.

Data type	**Included**	**Note**
Commit details	Yes	
Commit statuses	Yes	
Deployments	Yes	
Environments	Yes	
Repository events	Yes	
Issues	Yes	
Issue events	Yes	
Issue statistics	Yes	
Pipeline jobs	Yes	
Merge requests	Yes	
Pipelines	Yes	
Releases	Yes	
Repository statistics	No	Requires membership access to the project

Table A.3: GitLab extraction coverage for *gitterHQ/gitter-android-app*. All standalone extraction requestors in `sources/gitlab_sources/` are listed.

B CI Benchmark Run Data

Three manually triggered CI runs are referenced in Section 7.2.4. All runs used the same benchmark configuration (all eight benchmarks enabled, no extraction item limits) on a GitHub Actions `ubuntu-latest` runner (2-core CPU, 7.75 GB RAM). Runs 1 and 3 started with a fresh rate limit budget. Run 2 started with the rate limit only partially recovered.

Benchmark	**Tier**	**Run 1**	**Run 2**	**Run 3**
SortinghatRequestor-Benchmark	1	23.4 s	22.0 s	22.3 s
IdentityUpdateBench-mark	2	12.9 s	12.8 s	11.9 s
BronzeToSilverBench-mark	2	13.1 s	13.1 s	12.0 s
GitLabAPIExtraction-Benchmark	2	1,164.4 s	failed (502)	1,159.2 s
GithubAPIExtraction-Benchmark	2	1,586.0 s	1,397.5 s	1,484.5 s
DevelopmentDataBench-mark	3	1,290.5 s	failed (403)	1,183.6 s
CostCalculationAnalytics-Benchmark	3	781.7 s	failed (cascading)	757.3 s
CodeChangeSummariza-tionAnalyticsBenchmark	3	failed (403)	failed (cascading)	failed (403)
Total duration		**4,872.1 s**	**1,445.3 s**	**4,630.7 s**
Successful / Failed		7 / 1	4 / 4	7 / 1

Table B.4: Benchmark results across three CI runs on 2026-04-11, 2026-04-12, and 2026-04-13.

Benchmark	**Run 1**	**Run 2**	**Run 3**
GitLabAPIExtractionBenchmark	6,538	—	6,538
GithubAPIExtractionBenchmark	4,841	4,703	4,715
DevelopmentDataBenchmark	4,800	—	4,740
CostCalculationAnalyticsBenchmark	700	—	700
CodeChangeSummarizationAnalyticsBenchmark	—	—	—

Benchmark	**Run 1**	**Run 2**	**Run 3**
Cumulative GitHub REST calls	**10,341**	**4,703**	**10,155**

Table B.5: API call counts per benchmark across the three CI runs.

C Benchmark Variance

Table C.6 reports the mean, standard deviation, and CV across five runs before and after the WP2 optimization for every metric collected by the eight benchmarks on the 12-core, 23 GB RAM Linux evaluation machine. The subset cited inline is shown in Table 7.1 in Section 7.2.2.

Benchmark	**Tier**	**Before/After WP2 optimization**	**Metric**	**Mean ± SD**	**CV (%)**
SortinghatRequestorBenchmark	1	before	add API calls	2,000 ± 0	0.0
SortinghatRequestorBenchmark	1	after	add API calls	20 ± 0	0.0
SortinghatRequestorBenchmark	1	before	show API calls	4,000 ± 0	0.0
SortinghatRequestorBenchmark	1	after	show API calls	20 ± 0	0.0
SortinghatRequestorBenchmark	1	before	add duration	35.85 ± 1.54 s	4.3
SortinghatRequestorBenchmark	1	after	add duration	9.17 ± 0.14 s	1.5
SortinghatRequestorBenchmark	1	before	show duration	50.35 ± 1.16 s	2.3
SortinghatRequestorBenchmark	1	after	show duration	3.89 ± 0.11 s	2.8

Benchmark	**Tier**	**Before/After WP2 optimization**	**Metric**	**Mean ± SD**	**CV (%)**
SortinghatRequestorBenchmark	1	before	total duration	86.20 ± 1.03 s	1.2
SortinghatRequestorBenchmark	1	after	total duration	13.06 ± 0.24 s	1.9
BronzeToSilverBenchmark	2	before	wall-clock duration	14.48 ± 0.44 s	3.0
BronzeToSilverBenchmark	2	after	wall-clock duration	4.94 ± 0.05 s	0.9
IdentityUpdateBenchmark	2	before	wall-clock duration	7.76 ± 0.15 s	1.9
IdentityUpdateBenchmark	2	after	wall-clock duration	6.82 ± 0.43 s	6.4
GithubAPIExtractionBenchmark	2	before	wall-clock duration	2,944.77 ± 116.62 s	4.0
GithubAPIExtractionBenchmark	2	after	wall-clock duration	2,761.58 ± 106.91 s	3.9
GitLabAPIExtractionBenchmark	2	before	wall-clock duration	1,959.81 ± 60.84 s	3.1
GitLabAPIExtractionBenchmark	2	after	wall-clock duration	1,844.70 ± 38.28 s	2.1
DevelopmentDataBenchmark	3	before	wall-clock duration	2,181.49 ± 40.67 s	1.9

Benchmark	Tier	Before/After WP2 optimization	Metric	Mean ± SD	CV (%)
DevelopmentDataBenchmark	3	after	wall-clock duration	2,047.65 ± 32.28 s	1.6
CostCalculationAnalyticsBenchmark	3	before	wall-clock duration	871.31 ± 9.77 s	1.1
CostCalculationAnalyticsBenchmark	3	after	wall-clock duration	870.49 ± 9.47 s	1.1
CodeChangeSummarizationAnalyticsBenchmark	3	before	wall-clock duration	2,106.81 ± 26.61 s	1.3
CodeChangeSummarizationAnalyticsBenchmark	3	after	wall-clock duration	2,076.41 ± 87.16 s	4.2

Table C.6: Full per-metric variance across five runs before and after the WP2 optimization on a 12-core, 23 GB RAM Linux machine.

D SortingHat Batching Validation

The three integration test suites referenced in Section 6.2.4 total 96 tests across the SortingHat requestor, the Bronze-to-Silver transformer, and the identity handler. The following lists every test class, with individual test functions promoted to their own row when they encode a specifically called-out regression guarantee. The *Source* column names the design section that promised the property or the requirement identifier the test retires.

Test class / function	**Validated property**	**Source**
TestAddIdentities-BatchEquivalence	Batched add output equals sequential add output	Batched Identity Resolution
TestAddIdentitiesBatch-Caching	Class-level dict cache survives repeated batch calls	Caching strategy
TestAddIdentities-BatchErrorScenarios	Duplicate, pre-existing, and mixed-success items resolve correctly	Correctness guarantee
TestAddIdentities-BatchEdgeCases	Large batches, batch size of one, default source handling	Batched Identity Resolution
TestShowBatchEquivalence	Batched show output equals sequential show output	Batched Identity Resolution
TestShowBatchCaching	Class-level dict cache persists across repeated lookups	Caching strategy
TestShowBatchErrorScenarios	Nonexistent and mixed-validity UUIDs return null entries	Correctness guarantee
TestShowBatchEdge-Cases	Large batches, duplicate UUIDs, batch size of one	Batched Identity Resolution
TestOn409UUIDIsMain-UUID	409 response carries the canonical UUID before and after merge	RQ-9

Table D.7: Requestor suite validation tests.

Test class / function	Validated property	Source
TestAddIdentities	Single, partial, multi-column, and multi-row identity resolution	Correctness guarantee
TestAddDictIdentities	Array-typed identity columns flatten and regroup correctly	Integration into the Pipeline
TestReplaceIdentities	Identity columns are swapped for resolved id columns	Correctness guarantee
TestFullIdentityPipeline	End-to-end identity flow through the transformer	Integration into the Pipeline
TestImplementationCorrectness	Partition consistency, large batches with duplicates, combined add and dict	Integration into the Pipeline
test_exact_batch_boundary_rows	Result mapping at the exact batch boundary	Correctness guarantee
test_two_batch_boundary_with_409_in_second_batch	Boundary handling combined with 409 resolution	RQ-9
test_row_order_preserved_after_identity_resolution	Work-list mapping preserves row order	Integration into the Pipeline
test_multiple_partitions_with_shared_identity_across_partitions	Partition-level deduplication is consistent across partitions	Integration into the Pipeline
test_merged_identity_resolves_to_canonical_mk	Post-merge identities resolve to canonical UUID in Silver	RQ-9

Table D.8: Bronze-to-Silver transformer suite validation tests.

Test class / function	Validated property	Source
TestUpdateIdentitySingleField	Single-field update across unchanged, merged, and missing UUIDs	Identity Update Batching
TestUpdateIdentityMultipleFields	Multi-field updates and unchanged non-identity columns	Identity Update Batching

Test class / function	**Validated property**	**Source**
TestUpdateDictIdentities	Array-typed update with mixed-changed and null entries	Identity Update Batching
test_null_array_preserved	Null arrays no longer raise a *TypeError*	Identity Update Batching
test_partitioned_dataframe_consistency	*mapPartitions* produces consistent output	Identity Update Batching
test_self_next_called_with_result	Pipeline-step contract preserved	RQ-7

Table D.9: Identity handler suite validation tests.

References

Apache Software Foundation. (2023a,). *Apache DevLake*. https://devlake.apache.org/

Apache Software Foundation. (2023c,). *Apache DevLake — Architecture Overview*. https://devlake.apache.org/docs/Overview/Architecture

Apache Software Foundation. (2023b,). *Apache DevLake — Source Repository*. https://github.com/apache/incubator-devlake

Apache Software Foundation. (2026,). *RDD Programming Guide*. https://spark.apache.org/docs/latest/rdd-programming-guide.html

Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., Hovell, H. van, Ionescu, A., Łuszczak, A., undefinedwitakowski, M., Szafrański, M., Li, X., Ueshin, T., Mokhtar, M., Boncz, P., Ghodsi, A., Paranjpye, S., Senster, P., … Zaharia, M. (2020). Delta lake: high-performance ACID table storage over cloud object stores. *Proc. VLDB Endow.*, *13*(12), 3411–3424. https://doi.org/10.14778/3415478.3415560

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., & Zaharia, M. (2015). Spark SQL: Relational Data Processing in Spark. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 1383–1394. https://doi.org/10.1145/2723372.2742797

Beck, K., & Andres, C. (2004). *Extreme Programming Explained: Embrace Change*. Addison-Wesley. https://books.google.de/books?id=32RGBAAAQBAJ

Buchner, S., & Riehle, D. (2022). Calculating the Costs of Inner Source Collaboration by Computing the Time Worked. *Proceedings of the 55th Hawaii International Conference on System Sciences, HICSS '22*, 7466–7475. https://doi.org/10.24251/HICSS.2022.896

Capraro, M. (2020). *Measuring Inner Source Collaboration* [Doctoral dissertation]. https://open.fau.de/items/91be414a-8140-4226-831d-74c433e41d80

Capraro, M., Dorner, M., & Riehle, D. (2018). The Patch-Flow Method for Measuring Inner Source Collaboration. *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*, 515–525. https://doi.org/10.1145/3196398.3196417

Ciuraru, I. C. (2014,). *pytest-benchmark*. https://pytest-benchmark.readthedocs.io/

Dean, J., & Barroso, L. A. (2013). The tail at scale. *Commun. ACM*, *56*(2), 74–80. https://doi.org/10.1145/2408776.2408794

Dueñas, S., Cosentino, V., Gonzalez-Barahona, J. M., Castillo San Felix, A. del, Izquierdo-Cortazar, D., Cañas-Díaz, L., & Pérez García-Plaza, A. (2021). GrimoireLab: A toolset for software development analytics. *Peerj Computer Science*, *7*, e601. https://doi.org/10.7717/peerj-cs.601

Dziurla, D. (2024,). *action-download-artifact: GitHub Action to download an artifact associated with given workflow and commit or other criteria.* https://github.com/dawidd6/action-download-artifact

Forsgren, N., Storey, M.-A., Maddila, C., Zimmermann, T., Houck, B., & Butler, J. (2021). The SPACE of Developer Productivity: There's more to it than you think. *ACM Queue*, *19*(1), 20–48. https://www.microsoft.com/en-us/research/publication/the-space-of-developer-productivity-theres-more-to-it-than-you-think/

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns.* Addison Wesley.

GitHub. (2024b,). *Rate limits and node limits for the GraphQL API.* https://docs.github.com/en/graphql/overview/rate-limits-and-node-limits-for-the-graphql-api

GitHub. (2024a,). *Rate limits for the REST API.* https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api

GitHub. (2026,). *About forks.* https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/about-forks

GitHub, Inc. (2022,). *Workflow commands for GitHub Actions: Adding a job summary.* https://docs.github.com/en/actions/reference/workflows-and-actions/workflow-commands#adding-a-job-summary

GitLab. (2024,). *Rate limits.* https://docs.gitlab.com/user/gitlab_com/#rate-limits-on-gitlabcom

GitLab. (2026,). *Forks.* https://docs.gitlab.com/user/project/repository/forking_workflow/

Goggins, S. P. et al. (2021). Making Open Source Project Health Transparent. *Computer*, *54*(8), 104–111. https://doi.org/10.1109/MC.2021.3084015

Goggins, S. et al. (2021). Open Source Community Health: Analytical Metrics and Their Corresponding Narratives. *2021 IEEE/ACM 4th International Workshop on Software Health in Projects, Ecosystems and Communities (Soheal)*, 25–33. https://doi.org/10.1109/SoHeal52568.2021.00010

GraphQL Foundation. (2021). *GraphQL Specification, October 2021 Edition* [Technical report]. https://spec.graphql.org/October2021/

Hirsch, J., & Riehle, D. (2022,). Management Accounting Concepts for Inner Source Software Engineering. *Proceedings of the 13th International Conference on Software Business (ICSOB 2022)*. https://mecois.com/wp-content/uploads/2024/03/hirsch-riehle_2022.pdf

Huang, S., Huang, J., Dai, J., Xie, T., & Huang, B. (2010). The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW)*, 41–51. https://doi.org/10.1109/ICDEW.2010.5452747

Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* (1st ed.). Addison-Wesley Professional.

Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons.

Karau, H., Kowinski, A., Hamstra, M., & Zaharia, M. (2015). *Learning spark*. O'Reilly Media.

Kravchenko, T., Bogdanova, T., & Shevgunov, T. (2022). Ranking Requirements Using MoSCoW Methodology in Practice. In R. Silhavy (Ed.), *Cybernetics Perspectives in Systems: Cybernetics Perspectives in Systems*.

Laaber, C., & Leitner, P. (2018). An evaluation of open-source software microbenchmark suites for continuous performance assessment. *Proceedings of the 15th International Conference on Mining Software Repositories*, 119–130. https://doi.org/10.1145/3196398.3196407

Li, M., Tan, J., Wang, Y., Zhang, L., & Salapura, V. (2015,). SparkBench: A Spark Benchmarking Suite Characterizing Large-Scale In-Memory Data Analytics. *Proceedings of the 12th ACM International Conference on Computing Frontiers (CF)*. https://doi.org/10.1145/2742854.2747283

Lilja, D. J. (2000). *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press.

Nguyen, T. H. D., Adams, B., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2012). Automated Detection of Performance Regressions Using Statistical Process Control Techniques. *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE '12)*, 299–310. https://doi.org/10.1145/2188286.2188344

Poess, M., Rabl, T., Jacobsen, H.-A., & Caufield, B. (2014). TPC-DI: The First Industry Benchmark for Data Integration. *Proceedings of the VLDB Endowment*, *7*(13), 1367–1378. https://doi.org/10.14778/2733004.2733009

Professorship for Open-Source Software. (2026,). *Engineering Thesis: Structure and Content*. https://oss.cs.fau.de/theses/structure-content/

Shields.io. (2026,). *Shields.io*. https://shields.io/

Simitsis, A., Vassiliadis, P., Dayal, U., Karagiannis, A., & Tziovara, V. (2009). Benchmarking ETL Workflows. *Performance Evaluation and Benchmarking (TPCTC 2009)*, *5895*, 199–220. https://doi.org/10.1007/978-3-642-10424-4_15

Stack Overflow. (2025,). *2025 Stack Overflow Developer Survey: Technology — Code, documentation, and collaboration tools*. https://survey.stackoverflow.co/2025/technology#1-code-documentation-and-collaboration-tools

Strengholt, P. (2025). *Building medallion architectures*. O'Reilly Media.

Vogel, M., Weber, S., & Zirpins, C. (2018). Experiences on Migrating RESTful Web Services to GraphQL. *Service-Oriented Computing – ICSOC 2017 Workshops*, *10797*, 283–295. https://doi.org/10.1007/978-3-319-91764-1_23

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010, June). Spark: Cluster Computing with Working Sets. *2nd USENIX Workshop on Hot Topics in Cloud Computing (Hotcloud 10)*. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., & Stoica, I. (2016). Apache Spark: a unified engine for big data processing. *Commun. ACM*, *59*(11), 56–65. https://doi.org/10.1145/2934664