Merging and Anonymizing User Identities in Software Development

MASTER THESIS

Alexander Sacharenko

Submitted on 5 November 2025

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

Supervisor:
Thomas Wolter, 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, 5 November 2025

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see https://creativecommons.org/licenses/by/4.0/

Erlangen, 5 November 2025

Abstract

Data extracted from software engineering tools, such as GitHub, GitLab, and Jira, can serve as an important source for metrics and studies. However, to conduct accurate studies, we need to unify the non-uniform identities of developers on those platforms. Additionally, identities need to be anonymized before releasing research results. While there are available solutions, none meet the criteria for dependency management and data anonymization required by the MECOIS project.

This thesis presents the design and implementation of an internal identity matching tool that removes the reliance on third-party tools. The prototype implements a three-phase matching algorithm that matches email addresses to discover related profiles, automatically merging high-confidence matches, and flagging uncertain cases for manual review. The system implements a PostgreSQL database schema that uses UUIDs for anonymization and integrates into the existing MECOIS data pipeline. The implementation is flexible, allowing the use of different similarity algorithms without changing the core logic.

The prototype has been evaluated against the defined functional and non-functional requirements and shown that the core identity matching functions work as required. The system provides MECOIS with independence from external tools while retaining full control of the identity management and anonymization processes.

Contents

1 Introduction **1**

2 Literature Review **3**

- 2.1 Software Development Data Analysis 3
  - 2.1.1 Uses of Development Data in Research 4
  - 2.1.2 Multi-Source Integration Challenges 4
- 2.2 Record Matching and Entity Resolution 5
  - 2.2.1 Record Matching Fundamentals 6
  - 2.2.2 String Similarity Algorithms 6
  - 2.2.3 Threshold-Based Classification 8
- 2.3 Privacy and Anonymization 9
  - 2.3.1 Privacy Requirements in Research Data 9
  - 2.3.2 Anonymization Techniques 10
  - 2.3.3 Anonymization in Software Development Data 11
- 2.4 Internal vs. External Tooling 12

3 Requirements **17**

- 3.1 Functional Requirements 17
- 3.2 Non-Functional Requirements 18

4 Architecture **21**

- 4.1 Current Architecture 21
- 4.2 Database Schema Concept 23
- 4.3 Code Architecture 24

4.3.1 Coordinator 25
4.3.2 Data Access 26
4.3.3 Matching Strategy 26
4.3.4 Identity Matching 27

5 Design and Implementation 29
5.1 Design Decisions 29
5.1.1 Applied Design Patterns 30
5.1.2 Algorithm Design Choices 32
5.2 Component Implementation 36
5.2.1 Coordinator Implementation 37
5.2.2 Orchestrator Implementation 38
5.2.3 Data Access Layer Implementation 42
5.2.4 Matching Strategy Implementation 43
5.3 Database Schema 46
5.4 Technology Stack 49
5.4.1 Core Technologies 49
5.4.2 Python Libraries 50

6 Evaluation 53
6.1 Functional Requirements 53
6.1.1 Identity Data Management 54
6.1.2 Identity Anonymization 55
6.1.3 Workflow and Configuration 55
6.2 Non-Functional Requirements 56
6.3 System Quality Attributes 57
6.3.1 Architecture and Extensibility 58
6.3.2 Operations and Data Integrity 59
6.4 Limitations 60

7 Conclusion 63

Appendices **65**
Appendix A Complete Database Schema 67

References **69**

List of Figures

Figure 4.1: Simplified MECOIS data pipeline architecture (Buchner, 2024) 22
Figure 4.2: SortingHat database schema (Moreno, et al., 2019) 23
Figure 4.3: Conceptual database schema for identity management 24
Figure 4.4: Component architecture and interaction flow for the IMS 25

Figure 5.1: Strategy pattern example for email similarity algorithms 30
Figure 5.2: Example of a connected components algorithm result in Phase 3. 35
Figure 5.3: Shortened version of database schema for identity matching and anonymization . 46

List of Tables

Table 2.1: Comparison of string similarity algorithms 8

Table 5.1: Overview of used design patterns..32
Table 5.2: Matching algorithm decision matrix...41

Table 6.1: Overview of the evaluation result for functional requirements 53
Table 6.2: Overview of evaluation result of non-functional requirements 57

Listings

Listing 5.1: DFS implementation for finding connected components 40
Listing 5.2: Domain grouping to limit comparison scope...41
Listing 5.3: Naive WHERE clause failing NULL comparison 42
Listing 5.4: NULL-aware deduplication with compound WHERE clause construction 42
Listing 5.5: Resulting NULL-safe WHERE clause 42
Listing 5.6: EmailSimilarityStrategy abstract base class 44
Listing 5.7: SequenceMatcherStrategy for subsequence-based similarity 44
Listing 5.8: LevenshteinStrategy for edit distance-based similarity..................................44
Listing 5.9: JaroWinklerStrategy with prefix weighting...45

Acronyms

ACID Atomicity, Consistency, Isolation, Durability

CRUD Create, Read, Update, Delete

DAO Data Access Object

DBMS Database Management System

DFS Depth-First Search

GDPR General Data Protection Regulation

IMS Identity Management System

JSON JavaScript Object Notation

RDD Resilient Distributed Dataset

SQL Structured Query Language

UUID Universally Unique Identifiers

1 Introduction

Many software development projects are moving toward leveraging data analysis in all of their process steps (quality assurance, process optimization, strategic decision-making) and are therefore looking at ways to build more of their own tools and avoid the reliance on third-party vendors and tools (Buse & Zimmermann, 2012). The MECOIS project is one such project and aims to enable users to perform data analysis on data from software development platforms, including GitHub, GitLab, Jira, and Git repository data.

However, there are two major challenges to performing such analyses. Firstly, a given developer may be known under different identities (email addresses, display names) across different platforms, resulting in identity fragmentation. As a result, it is necessary to identify and merge related profiles into a unified profile to allow for the performance of accurate cross-platform analysis. The second challenge is that data collected during research must be anonymized prior to publication to protect the privacy of the individual developers and to comply with applicable data protection laws and regulations. Therefore, it is necessary to remove personal identifiable information, such as name and email address, from the research data while still allowing for traceability for internal purposes. Furthermore, as previously mentioned, organizations want control of their data processing pipelines so they do not need to depend upon third-party providers who may present a potential security risk, licensing issues, or who may cease providing the service at some point in the future. This thesis will address all of these areas by developing an internal data analysis component, which will replace external tools.

Currently, MECOIS relies on SortingHat, a third-party identity management tool, to address both of these challenges. However, there are several limitations that make it unsuitable for use in the MECOIS project. The method for anonymizing data does not meet the requirements of MECOIS for publishing research data. Furthermore, relying on a third party to manage the identity management process for MECOIS results in additional maintenance expenses, reduces MECOIS' ability to customize its workflows to meet specific requirements, and places MECOIS in a position where it has less control over the identity management process. Given these constraints, the decision was made to develop an internal identity matching system.

Therefore, the purpose of this thesis is to develop a prototype identity management system (IMS) that addresses these requirements, contributing the following functionality:

- A database schema that supports importing identities from multiple sources.

- The merging of identities based on similarity calculations, while tracking uncertain matches that require human review.
- The anonymization of personal identifiable information.
- The integration with the existing technology stack of MECOIS facilitates seamless use in the existing data pipeline workflow of MECOIS.

This thesis will serve as an engineering thesis[1]. Therefore, the primary focus will be on demonstrating the application of engineering principles and the use of technical expertise to fill the gap between theoretical knowledge and real-world problems. The thesis will demonstrate this through the design and implementation of a fully functional prototype that addresses concrete problems associated with the MECOIS project.

The thesis is organized as follows. Chapter 2 provides a review of the relevant literature associated with the analysis of software development data, techniques used to match records, privacy concerns associated with the collection and use of data, and the considerations associated with building internal versus external tools. Chapter 3 outlines the functional and non-functional requirements that guided the design of the system. Chapter 4 describes the system architecture, outlining both the current MECOIS pipeline and the new components added to provide identity management. Chapter 5 describes design decisions and implementation specifics for each of the components. Chapter 6 assesses the prototype developed in this thesis against the requirements outlined in chapter 3 and identifies its limitations. Finally, chapter 7 provides conclusions regarding the completed work and identifies opportunities for future enhancement.

[1] https://oss.cs.fau.de/theses/structure-content/engineering-thesis/

2 Literature Review

This chapter provides a theoretical basis for the identity matching implemented in this thesis. Software development data from sources such as GitHub repositories, issue tracking systems, and code review systems provide researchers with valuable data and insights. However, to effectively analyze these types of data, the researcher needs to resolve the identity resolution problem, which is when a developer uses one or multiple identifiers on various platforms, making it difficult to link their activity across all of those platforms to the correct individual. The prototype was created based on several key concepts, which included the characteristics of software development data analysis and what is required in order to perform accurate identity matching (section 2.1), techniques used for matching records in order to identify related profiles (section 2.2), how to protect individual privacy while conducting research using anonymization (section 2.3), and the respective advantages of internal and external solutions (section 2.4).

2.1 Software Development Data Analysis

Version control systems, as well as issue tracking systems, generate large amounts of data as they record all code changes made by developers along with their names and time stamps for when they were made. The same type of data is also generated through code review platforms, which document how developers collaborate on coding projects and what types of feedback they provide to each other. All of these types of data enable organizations and researchers to mine software repositories to study how software is developed, evaluate how productive software development teams are, identify who in an organization has worked together most frequently, and help them create better quality software (Kagdi, Collard, & Maletic, 2007). While there are many ways to utilize the wealth of data that is generated during the software development process, one major obstacle to utilizing this data effectively is that developers do not use consistent identifiers for themselves across different platforms and tools. The following sections will outline research usage of development data (section 2.1.1) and difficulties that involve dealing with multiple data sources (section 2.1.2).

2.1.1 Uses of Development Data in Research

The mining of software repositories is an area of study that analyzes development data to gain insight into how software was developed and maintained. The buildup of commit history enables the researcher to note the productivity patterns of particular developers, for example, the time of day when they are most active, or the speed at which bugs are remedied (Hassan, 2008). Collaboration analysis can determine the structural relationships of members of a team, helping to decipher the communication patterns within the team, which individuals work with whom, and with what components they interact (Bird, Nagappan, Murphy, Gall, & Devanbu, 2011). Quality prediction models can examine previous project data and determine the code areas that are most likely to have defects, helping teams direct testing, and review efforts to component areas with the most faults (D'Ambros, Lanza, & Robbes, 2012).

All of these studies rely heavily on accurate identification of the person committing the activity. When a programmer commits code to a repository using "john.smith@company.com" on GitHub, opens issues as "jsmith" on Jira, and then reviews code as "smithj@gitlab.company.com", they will appear in raw data as three individual identities. Without some form of identity matching, the collaborations between programmers will appear fragmented, the productivity of each programmer will be split across multiple identities, and longitudinal measurements of the contributions of individual programmers will be impossible. Therefore, the accuracy of any findings made from the data collected during these studies relies on the ability to link the correct activity to the correct individual.

In addition to being useful for research purposes, the data collected during the development of software can be used by organizations for many practical reasons. For instance, organizations use the development metrics to evaluate the performance of their teams, to identify knowledge silos, and to allocate resources more efficiently (Menzies & Zimmermann, 2013). Additionally, development metrics, such as velocity, code review turnaround time, and bug resolution rate, can be tracked and monitored through development dashboards. However, when the activities of a single developer are distributed over several unlinked identities, the reliability of the development metrics will be compromised. A manager cannot be confident that all of a specific developer's work is correctly attributed to them if the activities were completed under different usernames or email addresses depending on the platform that was used.

2.1.2 Multi-Source Integration Challenges

The process of modern software development is characterized by many different platforms and different identifier systems. The author email addresses and names, configured locally by developers, are stored in Git commit metadata as strings. In contrast, GitHub's usernames serve as primary identifiers for users, while allowing multiple email addresses to be associated with a user account. Each platform maintains its own user database, including GitLab, Jira, Slack, Asana, and other collaborative tools. Thus, the identity fragmentation issue is due to several factors. For example, developers may change email addresses as they move

between jobs or modify their personal privacy settings. Consequently, a developer may use "personal@gmail.com" to contribute to open-source repositories and "name@company.com" for company projects. Similarly, it is common for there to be variations of a developer's name: "John Smith", "J. Smith", "Smith, John", or "JSmith" for example, could all refer to the same individual. Furthermore, identity fragmentation across software repositories, which can happen through such variations, is one of the significant challenges in software repository mining (Goeminne & Mens, 2013). Therefore, automated solutions are necessary that will allow the detection of similarities regardless of the variability in the text. Additionally, typos and inconsistencies arise when developers manually configure author information in Git. In some cases, developers will purposefully use a variety of personas depending upon the organization or project being developed.

Cross-platform analysis is complicated because of the existence of these fragmented identities. Consider the case of a developer who participates in multiple projects within an organization but uses different identifiers for each project. For example, if he/she commits to Project X using "a.developer@company.com" on GitHub and creates issues for Project Y using the username "adeveloper" on Jira, then in order to correctly analyze the developer's workload across both projects, analysis must identify "a.developer@company.com" and "adeveloper" as being the same person. Otherwise, studies examining the total amount of work done by the developer for all projects will identify him/her as two different people and thus cause contribution metrics and collaboration patterns to become fragmented.

In addition to the complications caused by the existence of fragmented identities, bot accounts, and system identities also add additional complications to analysis. Automatic processes like continuous integration systems, deployment scripts, and repository maintenance bots generate commits and activity that should not be attributed to the human developers. For example, e-mail addresses like "noreply@github.com", "jenkins@build.server", or "dependabot@users.noreply.github.com" must be removed from the analysis to avoid causing artificial inflation in the number of hours worked by humans. Tools like SortingHat help to address these complications by providing a blacklist of well-known system accounts and using similarity-based matching for human identities (Moreno, et al., 2019).

Therefore, IMSs were developed in response to these complications. The first IMSs were based on manual curation, where researchers or administrators manually determined which identities should be merged. However, manual curation does not scale to large numbers of datasets or ongoing data collection. Therefore, automated identity matching based on similarity calculations was developed to provide a scalable method for resolving identities in large datasets (Goeminne & Mens, 2013). However, automated methods require careful design to minimize incorrect merges of distinct individuals that share similar identities.

2.2 Record Matching and Entity Resolution

In order to address the many issues related to the integration of multiple sources, IMSs use record matching and entity resolution technologies. The objective of this section is to outline

the methods that support automated identity matching: methods for comparison of records (section 2.2.1), string similarity metrics that measure the degree to which the identifiers being compared are similar (section 2.2.2), and the method of using thresholds to categorize records as a “match” or another category (section 2.2.3).

2.2.1 Record Matching Fundamentals

The matching of identities relies upon comparing the properties of each record to identify duplication. When the exact identifier is available (for example, the e-mail address), then it is clear that a duplicate exists. For instance, two records that contain identical e-mail addresses are representing the same individual. However, when developers utilize the same identifier but with variations between systems, a method called fuzzy matching is required to recognize the similarity between the identifiers regardless of textual differences.

There are two primary methods for performing fuzzy matching: deterministic rule-based approaches and probabilistic approaches. Rule-based methods are deterministic because they rely on explicit rules for matching, such as “merge the records if the local parts of the e-mail addresses differ by no more than 2 characters” or “merge the records if both the first and last names of the individuals represented in the records match exactly and the local parts of the domain of the e-mail addresses match exactly”. The advantages of deterministic methods include that they provide consistent, explainable results; however, these methods also require the domain knowledge to develop effective rules (Christen, 2012). Probabilistic methods evaluate the likelihood that two records represent the same individual based on patterns of agreement across multiple attributes. Methods, such as the Fellegi-Sunter model, can be used for this purpose by weighing the relative importance of each field comparison (Fellegi & Sunter, 1969). While probabilistic methods are generally more flexible than deterministic methods, they require training data to perform effectively and produce probability scores that may be difficult for domain experts to understand.

In general, e-mail addresses serve as particularly strong identifiers for developers during the process of identity matching. E-mail addresses are unique, consistent, and follow a structured format consisting of a local part and a domain part. Most developers at organizations will be utilizing an e-mail address provided by their organization and therefore, most of the time, the domain portion of the e-mail address will be consistent. The local portion of the e-mail address may include variations such as numbers or punctuation added to the developer’s name or other elements that create a predictable pattern of similarity and difference. Therefore, the structured nature of e-mail addresses makes them ideal for using string comparison algorithms for matching similar identifiers.

2.2.2 String Similarity Algorithms

String similarity algorithms compute the degree of similarity between two text strings. They output a number that is usually scaled between 0 (entirely different) and 1 (identical).

Three families of string similarity algorithms are commonly used for identity matching applications:

Edit Distance Algorithms determine the smallest number of steps required to convert one string to another. The Levenshtein Distance measures the number of single-character insertions, deletions, and substitutions (Levenshtein, 1965) required to convert one string to another. An example would be converting "johnsmith" to "jonsmith", this can be done by deleting the 'h' from "johnsmith". This yields a Levenshtein Distance of 1. The resulting distance is then normalized by the maximum length of the input strings to yield a similarity score. Edit Distance performs well when there are typos or other minor changes to an identifier; however, it treats each position in the string as equal, which does not represent how humans view similarity in identifiers.

Sequence Matching Algorithms calculate the longest identical substring found between the two strings being compared. The Ratcliff-Obershelp algorithm is used in difflib.SequenceMatcher to determine the longest common continuous substring of the two input strings, and then it continues by recursively comparing each string to the left and right sides of that common subsequence (Ratcliff & Metzener, 1988). The use of a recursive approach allows for the calculation of similarity using the Ratcliff-Obershelp algorithm to handle transposed characters and out-of-sequence elements more effectively than edit distance. Since in many cases the order of characters in email local parts is important, sequence matching offers a robust method of comparing similar variations in email local parts, such as "john.smith" or "johnsmith" vs. "smith.john".

Prefix-Weighted Methods determine similarity by comparing individual characters, with greater emphasis placed on identical characters near the start of the string. The Jaro-Winkler algorithm determines similarity between two strings based upon the number of identical characters within a certain distance window, with greater emphasis placed on the identical characters near the start of the string (Winkler, 1990). The use of a greater emphasis on identical characters near the start of the string is especially valuable when comparing email addresses where common naming conventions include placing identifying information at the beginning of the string, such as "company.john.smith" vs. "company.jane.doe" where the "company" prefix clearly indicates that they belong to the same organization. Jaro-Winkler produces a value between 0 and 1, with values closer to 1 indicating greater similarity.

Algorithm	Type	Strengths	Weaknesses	Best Use Case
Levenshtein	Edit Distance	Handles typos well	Treats all positions equally	Minor character variations
Ratcliff-Obershelp	Sequence Matching	Handles transpositions	Computationally intensive	Out-of-order elements
Jaro-Winkler	Prefix-Weighted	Good for prefixes	Less effective for suffix differences	Email addresses with common prefixes

Table 2.1: Comparison of string similarity algorithms

The selection of an appropriate string similarity algorithm will depend on what type of variations are to be matched, as seen in Table 2.1. Edit Distance will favor strings that have fewer character differences due to edit operations. Sequence Matching will favor strings that have longer contiguous identical subsequences. Jaro-Winkler will favor strings that have matching prefixes. While all three methods perform reasonably well in comparison to each other when applied to email local parts, empirical testing with a representative set of data from the target domain will typically reveal which algorithm will produce the most accurate results for that particular dataset's variation pattern.

2.2.3 Threshold-Based Classification

Algorithms producing similarities will generate results that are always continuous, while IMSs must provide a discrete decision regarding whether they want to automatically merge the profile, manually review it, or treat it separately as an individual entity. The thresholds are used to create the transition from the similarity score to the discrete decision.

Single-threshold-based methods divide all the profiles into two classes: profiles that exceed the threshold are merged and profiles that do not exceed the threshold remain separate. When determining the optimal threshold for such methods, we must find the right balance between precision (not creating false positives) and recall (missing merges of profiles of two individuals that are supposed to be combined). If you set a very high threshold like 0.95, there will be very few false positives, but many legitimate matches will be missed due to their moderate variations. On the other hand, setting a low threshold like 0.60 means that more true matches will be detected but also increases the risk of merging two profiles of distinct individuals because their identifiers happened to be similar (Elmagarmid, Ipeirotis, & Verykios, 2007).

Multi-threshold-based methods propose intermediate classification levels. Profiles that have been matched at a high confidence level are immediately merged. Profiles that fall within a range defined by a lower and upper threshold are sent to a human reviewer for further evaluation. Profiles that were identified as being below the lower threshold will be evaluated as

separate entities. The multi-threshold method, also referred to as "clerical review" in the literature dealing with record linking (Fellegi & Sunter, 1969), recognizes that automated decisions should only be made when there is sufficient confidence to support them (Christen, 2012). Cases where the decision is uncertain and requires a specific type of domain knowledge or contextual information should be sent to a human reviewer that can evaluate the situation and make an appropriate decision.

In identity matching the trade-offs between precision and recall are uneven. Incorrectly identifying a match (false positive) between two profiles of distinct individuals will lead to analysis errors because data from both individuals will be merged together and can be difficult to detect and fix after the fact. Failure to identify a match (false negative) between two profiles of individuals that should be identified as a single profile will result in fragmented metric data, which could be addressed through repeated attempts to match the data or through manual review of suspicious cases. Due to the unevenness of the trade-offs between false positives and false negatives, organizations often choose conservative threshold settings that prefer precision over recall and accept some fragmentation to prevent incorrect merges.

The process of calibrating thresholds is dependent upon several factors, including the algorithm that was used to measure the similarity, the properties of the dataset, and the organization's acceptable risk of different types of errors. Different algorithms can produce different distributions of scores; therefore, what is considered an optimal threshold for use with one algorithm may not be an optimal threshold for use with another. Therefore, empirical testing of the thresholds using a validation dataset that includes correct matches is the best way to determine an optimal threshold setting (Christen & Goiser, 2007).

2.3 Privacy and Anonymization

Identity merging allows researchers to merge developer profiles in order to allow them to analyze their activity accurately; however, by doing so, it introduces privacy concerns. The merged identity will provide an encompassing view of a person's behavior on multiple platforms, such as their work habits, how they contribute to open-source code, and what collaboration networks they participate in. Data from research may be published externally after all personally identifiable information is removed from the data to ensure that the developer has some level of privacy and to meet legal obligations to protect privacy. This section examines privacy regulations that apply to researchers data (section 2.3.1), examines methods of anonymizing data to protect an individual's identity (section 2.3.2), and discusses the issues related to anonymizing software development data (section 2.3.3).

2.3.1 Privacy Requirements in Research Data

The European Union's General Data Protection Regulation (GDPR) imposes strict requirements on the collection, processing, and sharing of personal information (European Parliament and Council of the European Union, 2016). Personal data is defined as infor-

mation that can identify a particular person, e.g., names, e-mail addresses, and unique identifiers. If personal data are processed, the organization concerned shall establish legitimate grounds for processing, shall restrict its data processing to determined purposes, and shall provide for appropriate data security. If an investigation consists of the analysis of personal data, further considerations arise concerning the rights of the data subjects, consent, and restrictions on data sharing.

Sharing research data facilitates scientific validation and reproducibility. Published research should include enough data that other researchers may measure or verify the results of the research and build upon previous work. However, the sharing among researchers of datasets containing personal identifiers conflicts with the obligation to protect anonymity. Thus, data subjects have the right to restrict the processing of their personal data and to object to its use for research purposes. Scientists cannot merely publish raw development data that contain e-mail addresses and names even if the analysis may be used to derive valuable scientific information.

Anonymization solves this issue by altering datasets to remove personal identifiers while still allowing for analysis. According to Recital 26 of the GDPR, properly anonymized data are outside the scope of the data protection regulations since there is no means by which the person can be identified (European Parliament and Council of the European Union, 2016). This allows research data to be shared without individual consent being required or restrictions being placed upon downstream uses of it. Anonymization must, however, be at such a point that re-identification is impossible for the recipient, even if they have combined the data with other information available (Article 29 Data Protection Working Party, 2014).

2.3.2 Anonymization Techniques

There are several approaches for removing or masking personal identifiers in data sets. The selection of which one is appropriate depends on the characteristics of the data, the requirements of the analyses, and the threat model of the risk of re-identification afforded by the data.

Pseudonymization substitutes direct identifiers such as names and email addresses for pseudo-identifiers such as random identification codes or hashed values as described in article 4 of the GDPR (European Parliament and Council of the European Union, 2016). For example, a substitution of an actual identifier such as “john.smith@company.com”, is a universally unique identifier (UUID) such as “a3f52c9d-4b7e-4f1a-9d3c-2e8b5a7c9f1d”, which removes the direct relationship to the personal identity of the individual. Pseudonymization is reversible if the mapping of the original identifiers and the pseudonyms are retained; hence, it is of value in circumstances where internal traceability is important. Research organizations can internally analyze the pseudonymized data and share the pseudonyms with the outside world, preventing the external party from re-identifying individuals automatically unless it has access to the mapping of identifiers.

Generalization has the effect of losing some degree of precision of the data to increase the breadth of categories. Some possible examples are that instead of specific ages, the ages might be broken down into age groups, such as 25-34, or there might be geographic areas, namely regions, of countries instead of individual locations. The procedure of generalization achieves k-anonymity, that is, that each record becomes indistinguishable from at least k-1 other records (Sweeney, 2012). This allows protection against re-identification achieved by combination with outside datasets but limits the precision and usefulness of the analytical aspect of the data. For statistical purposes, generalization might mean expanding times to weeks or just grouping developers by organization instead of individual persons.

Perturbation on the other hand, allows for random noise to be added to the data values or for switching the data values between individual records. Differential privacy is one form of enforcing this, whereby the result of analysis of the data will not admit to the inclusion of any specific individual data in the dataset (Dwork & Roth, 2014). Carefully calculated noise can be introduced into the data values, providing a mathematical degree of assurance for the privacy guarantee of the resultant data. Perturbation also necessitates an overall loss of data precision, as the amount of noise required for a strong privacy guarantee is so significant that it degrades the quality of individual analysis.

Data Suppression will strip out and remove attributes of a sensitive nature. For example, if an analysis does not require emails and names, those attributes can simply be omitted from the datasets being shared. This provides very good privacy protection but limits the use of the data. (Christen, 2012). Complete deletion would eliminate any analysis based on developer identities, thereby limiting its suitability to only those analyses that provide only aggregate statistics and that do not allow tracking of specific developers.

The choice of technique rests in the consideration of the balance between privacy protection and analytical usefulness. The better the privacy protection, the farther one needs to go into eliminating or sacrificing the analytical capability of data or data accuracy.

2.3.3 Anonymization in Software Development Data

Anonymizing software development data is a challenging task because the granularity of the activity logged by development platforms allows for the creation of detailed behavioral profiles by combining many different types of data, including the developer's coding style, their timing patterns for committing, what they write in comments regarding issues, and how they participate in reviews of other people's code. These patterns have been shown to allow researchers to identify individual developers based solely on their behavioral patterns, especially for those who are highly active and have very distinct styles (Caliskan-Islam, Voss, Yamaguchi, & Greenstadt, 2015). However, the threat model for development data is not similar to that of more sensitive domains such as healthcare and financial services. Much of the activity related to development occurs in public repositories where everyone can see what was done and when. As a result, developers do not mind being identified as having contributed to an open-source project. The most significant privacy issue relates to the use of a third party to create a complete cross-platform profile of an individual's activities and

therefore identify patterns in those activities that were not intended to be associated with each other.

In the case of organizational analysis of development data, pseudonymization is an attractive trade-off. Organizational analysts require the ability to follow up with internal traceability to confirm findings, investigate anomalies, and establish confidence in data quality. Internal traceability also requires the ability to trace patterns back to the original data so that automated identity matching processes can be verified to have properly grouped the profiles. Therefore, pseudonyms must be maintained along with the mapping of the pseudonyms to the actual identities, at least in internal systems. Data sharing outside of these internal systems will remove the mapping as well as protect the user's privacy and allow for analysis of development patterns.

Prior tools for identity matching in development data took a variety of approaches to anonymize the data. For example, SortingHat, developed as part of the GrimoireLab toolset, focused on identity merging as well as affiliation tracking (Moreno, et al., 2019) and creates hash values of personal data (like email addresses) for profile identifiers. However, SortingHat does not include salting to protect the hash values from rainbow table attacks or reverse lookups, which could potentially limit the effectiveness of the anonymization process. Therefore, organizations using SortingHat for research requiring high levels of privacy protection may need to add additional layers of anonymization before making their data available for research purposes.

UUID-based pseudonymization has multiple benefits for the anonymization of development data. UUIDs are random strings that act as pseudonyms for each identity and therefore provide no insight into the identity itself (Leach, Mealling, & Salz, 2005). Additionally, due to the size of the UUID space (2^128 possible values), brute force guessing or collisions are essentially impossible to perform. Furthermore, unlike hashed email addresses, UUIDs are completely resistant to rainbow table attacks or dictionary attacks. Therefore, organizations can maintain internal maps for traceability while releasing only UUIDs in their externally published datasets. Researchers can use these persistent pseudonyms to track developer activity across projects in research datasets without exposing actual identities of the developers, thus providing both analytic value and protecting privacy.

2.4 Internal vs. External Tooling

Companies working with data analysis systems are faced with a decision on how to create them (whether to develop internally or use an external tool). The two involve several trade-offs, including development time or effort compared to customization ability, the responsibility for system maintenance, and the potential for long-term control over the system. This section examines factors that influence these options and when it would be better to develop internally versus using an outside tool.

Control and Customization

Internal solutions allow organizations to have complete control over how a solution works, is implemented, and will be evolved. An organization has total flexibility to design a custom workflow, data structure, and integration pattern to meet its specific needs. There are no restrictions on an organization's ability to modify a solution's architecture and function to meet its unique needs. When there are significant differences between an organization's requirements and what an external solution provides, or when an organization's needs and requirements change and the external solution is unable to adapt to meet those changes, internal solutions become crucial (Basili & Boehm, 2001).

While external tools typically provide standardized functionalities that may not be a perfect fit for an organization's needs, most do provide some level of configurability to tailor the tool to meet an organization's needs. However, the underlying architectural structure of the tool cannot be modified by the end-user. Therefore, an organization must design and use its existing workflows to accommodate the assumptions made by the tool or develop workarounds to improve its overall usability. Additionally, an organization must rely on the vendors' development roadmaps and priorities for new feature enhancements, which may not align with the organization's business objectives or timeframes (Northrop, et al., 2006). In addition, if the vendor is actively developing the tool and there is a large number of users, it is likely that the limitations of relying on vendor-provided enhancements will be tolerable. However, if the tool is highly specialized and has limited users, the lack of responsiveness by the vendor could result in a high degree of frustration for the organization.

Dependency Management and Technical Debt

The use of external tools produces various risks due to creating a reliance on other tools, which may stop being used at some point, causing a crisis for the company seeking an alternative. The direction of development of the external tool may not align with the organization's requirements and will demand a difficult decision regarding migration. The organization may need to redo its integration with the tool if later versions are revised. Vulnerabilities in the security of the tool are dependent on the vendor fixing them, while the organization has no option to resolve this vulnerability themselves and therefore is reliant on how quickly the vendor responds (Fairbanks & Garlan, 2010). Each dependency created for using an external tool adds to the overall complexity of the systems and also takes away from the ability of the organization to have control over its own critical infrastructure.

Using internal resources to build solutions removes the reliance on external tools; however, internal solutions provide a new obligation. The organization is responsible for all aspects of the solution, including maintaining it, bug fixes, security updates, and adding new features. Maintaining and developing internal solutions requires sustained engineering efforts, and for organizations to continue to maintain as well as modify the solution, they need to retain knowledge within the organization. Employee turnover can lead to knowledge gaps, making it difficult for employees to understand how to modify the solution in the future. Just like using external tools, internal solutions produce technical debt, but the organization is entirely responsible for managing it (Cunningham, 1992). The trade-off lies

in what organizations would rather manage: internal technical debt or external dependencies outside of their control.

Integration Requirements

When integrating new software deeply into existing hardware and other business systems, organizations may prefer to develop it internally. This may occur when the new software needs to work closely with proprietary software, internal databases, or workflow processes specific to the company, thereby creating an integration cost for the external tool that is greater than the cost of developing the application internally. The development of code required to integrate the external tool with the organization's current infrastructure will increase both the complexity and the ongoing maintenance requirements associated with using the external tool (Bass, Clements, & Kazman, 2003).

However, if the external tool uses industry-standard interfaces and provides flexibility in how it can be integrated, then the burden on the organization to create a custom integration code may be reduced. Tools that allow developers to extend their functionality using third-party-developed plugins, APIs, or configuration files are more likely to easily accommodate different business environments without requiring large amounts of custom code. Nevertheless, while these tools may have been designed to be more adaptable, their use will still require an understanding of their architecture and the ability to operate within its constraints. Therefore, organizations will need to assess whether the time and resources needed to integrate the external tool into their existing systems are less than the time and resources needed to build an internal system that was designed to be an extension of their current infrastructure.

Development and Maintenance Costs

In general, the initial cost advantage will favor an externally developed tool. In order to obtain a tool that has been previously developed, there is no up-front development effort required. Instead, you receive the tool's functionality immediately at the expense of licensing fees or operational overhead associated with a self-hosted open-source tool. Organizations are able to take advantage of ongoing development and improvement by using a well-maintained external tool with an active community while making no direct investment in the organization's own development resources (Kogut & Metiu, 2001). Therefore, external tools tend to be attractive when an organization requires standard functionality that aligns with the capabilities of the tool being used.

However, long-term costs will shift the calculation. An internal solution will require continued investment in development resources; however, the organization will have full control over its priorities and timeline. Therefore, the organization can invest in the specific features it requires instead of buying a comprehensive tool that may not fully meet its needs. Also, maintenance costs for an internal solution are predictable and can be managed through decisions about resource allocation within the team. However, external tools can create unpredictable costs in the form of licensing fee changes, mandatory upgrades, or support services required by the vendor (Northrop, et al., 2006). Ultimately, the total cost of ownership will

depend largely upon how well the external tool meets the organization's requirements and how stable those requirements will be over time.

Domain-Specific Requirements

Organizations with specific domain needs (e.g., specific characteristics that are typically outside the norm) can often discover themselves in need of custom or internal solutions when their specific needs cannot be met through commercial/external products. A key example of such a case is the challenge of "Identity Matching" in software development data. There are many general-purpose "identity resolution" products available; however, these may not address the issues associated with software development (e.g., variations in email addresses across platforms, filtering out bots, etc.) and how these relate to software development data pipelines. An organization with an unmet specialized requirement would have to determine if it is more efficient to attempt to adapt an external product to meet their needs versus developing an internally "fit for purpose" solution (Basili & Boehm, 2001).

In addition to the factors relevant to organizations in general, several other factors specific to research and academia should also be considered. For example, as research organizations are seeking to maintain control over the algorithms used to resolve identities, publish transparent information about the processes used to make determinations about the identities, and provide the flexibility to test as well as experiment with different methods for resolving identities, commercially developed external products may not meet their needs. Specifically, externally developed products for production environments may prioritize operational stability and reliability over experimental flexibility. Publishing the results of one's research is enhanced when you are able to provide a detailed explanation of why your system was implemented as it was and defend those choices as needed. This requires an understanding of all aspects of your system (Howison & Herbsleb, 2011). The implementation of internally developed products provides both the transparency and control needed to achieve this; however, this comes at the expense of the amount of effort required to implement the product.

3 Requirements

This chapter outlines what the identity-matching component of MECOIS is supposed to do based on a combination of functional (section 3.1) and non-functional (section 3.2) requirements using the SOPHISTen GmbH requirements template (SOPHISTen, 2025). These requirements were used for designing and implementing the prototype in chapters 4 as well as 5 and will be used for evaluating the prototype in chapter 6.

3.1 Functional Requirements

Functional requirements are statements of services a system should offer, how it should respond to specific inputs, and how it should act in certain situations (Sommerville, 2015). The functional requirements for this prototype have been divided into three categories: identifying identity information from the sources of ingestion, matching identity information across platforms based on email similarity, and providing anonymous identity references to downstream users/analysts for further study.

FR-1: Identity data import

The system shall import identity records from supported sources such as Git, GitLab, or Jira into a database.

FR-2: Automatic Identity Merging

The system shall automatically merge raw user profiles into merged identities when similarity exceeds the auto-merge threshold.

FR-3: Candidate Identification

The system shall identify potential identity matches with moderate similarity and flag them as merge candidates for manual review.

FR-4: Handling blacklisted emails

During identity data import, the system shall exclude or flag identities whose email addresses are known dummies, bot, or noreply addresses.

FR-5: Identity Anonymization

The system shall anonymize merged identities using UUIDs.

FR-6: Internal UUID Mapping

The system shall maintain an internal mapping between merged identities and anonymized UUIDs to enable data analysis and result reproduction.

FR-7: Export Data Protection

Data exports shall exclude the UUID-to-identity mapping to ensure anonymization cannot be reversed externally.

FR-8: Decision Status Tracking

The system shall support tracking operator decisions on candidate groups with statuses including "pending", "approved", or "rejected".

FR-9: Batch Verification

Structured Query Language (SQL) checks shall be provided for the most recently completed batch to verify all eligible raw identities were linked and no raw identity maps to multiple merged groups.

FR-10: Database Schema Design

The system shall implement a relational database schema that supports identity import, merging, candidate management, blacklist filtering, and anonymization traceability.

FR-11: Pipeline Integration

The system shall integrate with the existing MECOIS data pipeline workflow.

FR-12: Configuration-Driven Identity Extraction

The system shall load identity field mappings from configuration files, allowing new data sources to be added without code changes.

3.2 Non-Functional Requirements

Non-functional requirements outline constraints that limit the functions/services provided by a system, which may include time constraints, constraints on the development process, and constraints due to external standards (Sommerville, 2015). Generally, these types of requirements apply to the overall system, rather than an individual feature within it. Non-functional requirements for this prototype focus on the quality characteristics of the system such as privacy, performance, maintainability, and transparency.

NFR-1: Scalability

The system shall be implemented to accommodate an increasing number of raw identities during data import without significant performance degradation.

NFR-2: Maintainability

The codebase shall be properly documented through docstrings and clear comments to support long-term maintainability.

NFR-3: Modularity

The system architecture shall be modular, separating the codebase into independent components to support the extensibility and testability of the prototype.

NFR-4: Separation of Responsibilities

The system shall separate responsibilities into distinct architectural layers.

NFR-5: Extensible Matching Methods

The system should support multiple similarity calculation algorithms, allowing alternative matching approaches to be implemented, and configured.

NFR-6: Configurable Anonymization

The system should support multiple anonymization approaches that can be selected based on data source and analysis requirements.

NFR-7: Operation Traceability

The system shall log manual decisions made by operators when merging or rejecting merge candidates, including operator identity, timestamp, and decision rationale.

NFR-8: Data Consistency

The system shall ensure database consistency through transaction management, ensuring that matching operations either complete fully or leave the database unchanged.

NFR-9: Database Technology

The system shall use PostgreSQL as the relational database management system (DBMS).

NFR-10: Privacy Protection

The anonymization method shall ensure that individual identities cannot be re-identified from anonymized datasets without access to internal mapping tables.

4 Architecture

The chapter will describe the system's architecture in depth. It starts with an overview of the current architecture to show how SortingHat fits into that (section 4.1). Next a simple version of the database schema will be presented along with an explanation of how it will meet the responsibilities of the current architecture that is being replaced (section 4.2). The last section of the chapter describes the target architecture along with all the parts and how they interact with one another (section 4.3). The information provided in this chapter lays the groundwork for the next chapter 5 which will describe the design and implementation of the prototype.

4.1 Current Architecture

The present structure is a series of processes to progressively prepare the processed information for analysis on the data. It follows the medallion pattern, which is a structure designed especially for use with data lakes. That pattern is composed of three layers:

Bronze: In this layer, the raw data is stored in its original format without undergoing any cleaning or transformation processes.

Silver: In this layer, the previously processed raw data will be cleaned up, filtered, validated, etc., to increase the quality and consistency of the data.

Gold: The highest-level layer is the gold layer. In this layer, the data received from the previous layer will be aggregated and structured to make it optimal for analytical use (Databricks, n.d.).

As shown in Figure 4.1, the pipeline construction begins with the processing of the previously selected source of data ("github_commits") through the process of data extraction. The newly processed data is then transferred into a PySpark DataFrame using the "TransformToBronze" pipeline stage and then placed into the Data Lake as raw data. Next, the raw data is cleaned up by the "TransformBronzeToSilver" pipeline stage using for instance, column flattening and column removal. The next two stages ("TransfromSilverToSilver" and "TransformToGold") are still in the development phase and have not yet been implemented (as of August 2025).

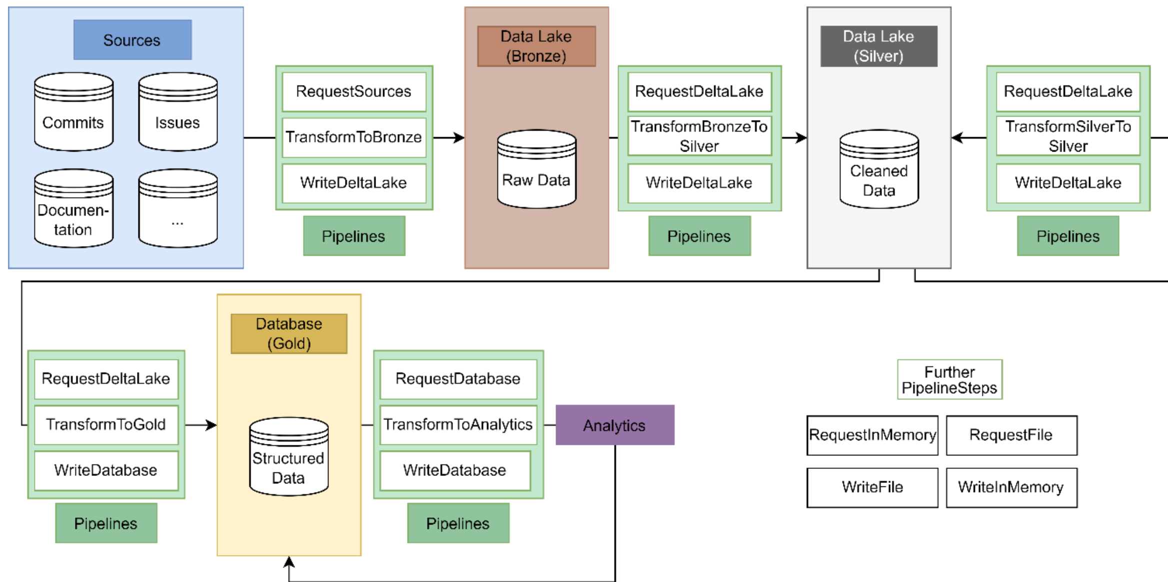


Figure 4.1: Simplified MECOIS data pipeline architecture (Buchner, 2024)

The identity management aspect happens in the "TransformBronzeToSilver" step and revolves around an external identity merger called SortingHat. In that pipeline step, all the implementation details related to the prototype for this thesis will be addressed. The following responsibilities are part of the external identity merger component (Moreno, et al., 2019):

1. Identity ingestion: Receives new identities from a given source.
2. Identity merging: Groups multiple identities belonging to one "real-world" person into a unified identity.
3. Tagging identities: Adds metadata such as organization affiliation, country, and gender.
4. Providing API access: Exposes a CLI and Python API for operations like querying, merging, and tagging.
5. Maintaining Traceability: Track the origin of identities for auditability.
6. Supporting manual corrections: Users can fix merging/tagging errors manually.
7. Storage of identities: Stores profiles and unified identities in a relational database.

SortingHat uses the database schema depicted in Figure 4.2 to store its identity data. This schema will be used as a starting point for the design process of the database schema of the prototype.

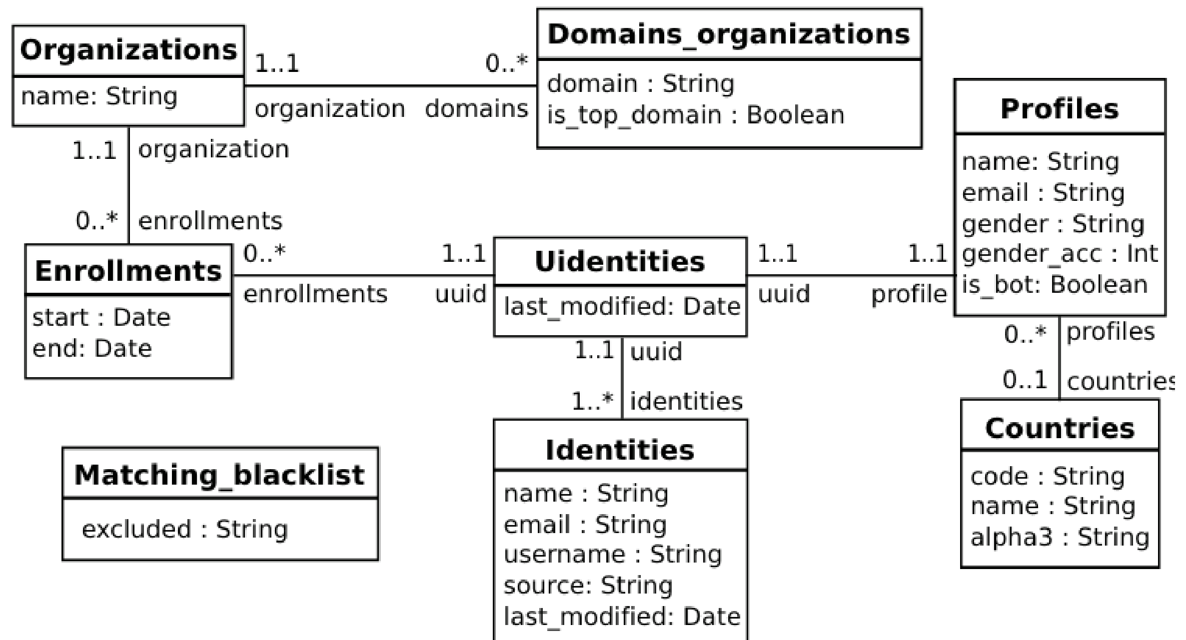


Figure 4.2: SortingHat database schema (Moreno, et al., 2019)

4.2 Database Schema Concept

The type of database is constrained by requirement FR-8 (see section 3.1), which specifies a relational DBMS. Accordingly, PostgreSQL is selected to satisfy this constraint. To support previous responsibilities and allow future additions to the schema, this first concept, as shown in Figure 4.3, was created. The diagram concept was created while keeping the original database schema for identity management in mind (see Figure 4.2). However, some structural changes were made, like changing merged identities to the central table of the schema and removing tables deemed unnecessary. This conceptual entity-relationship diagram covers the storage of newly extracted “raw_profiles” and defines their relationship to “merged_identities”. Anonymized identities should provide traceability by connecting them to the merged identity they associate with. Other key entities, such as “merge_candidates” have a connection to a table for their metadata (merge_candidate_groups) and to tables that include potential candidates for manual merges. Furthermore, to keep the option for tagging additional information to an identity, entities such as “organizations”, “enrollments” and “countries” have been added. Lastly, an entity called “matching_blacklist” is used to keep common or placeholder emails from being added to raw profiles.

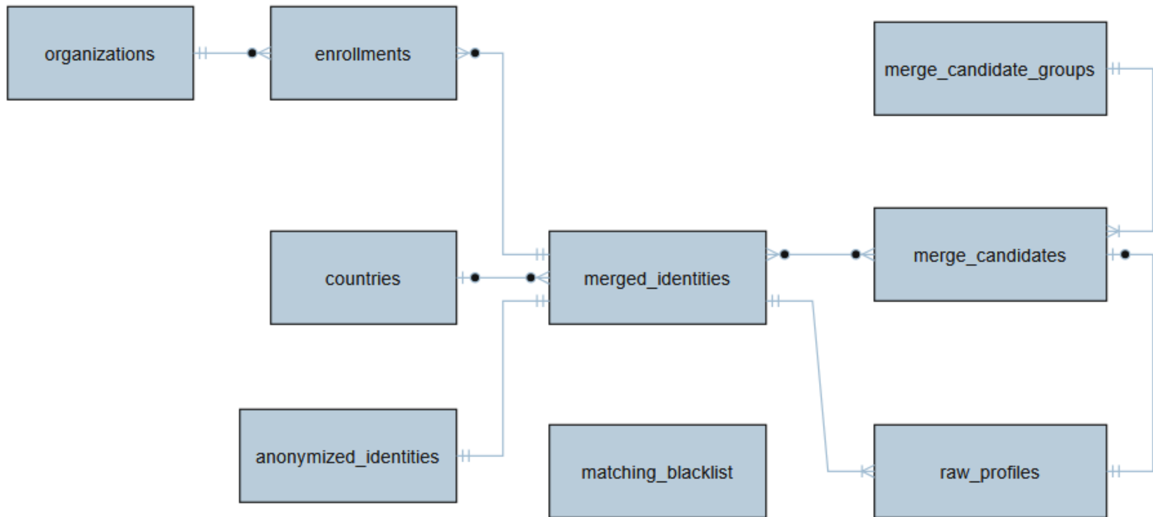


Figure 4.3: Conceptual database schema for identity management

4.3 Code Architecture

This system has four main components for the architecture of the prototype: Coordinator, Data Access, Matching Strategy, and Identity Matching Orchestrator. The Coordinator (see section 4.3.1) drives a reproducible run, and the Data Access Layer (see section 4.3.2) interacts with data from the persistent storage and thus allows access to the data that was stored there. The Matching Strategy (see section 4.3.3) determines how to calculate similarity, and the Orchestrator (see section 4.3.4) uses these strategies to create merged identities and review candidates as needed. Each of these components creates a separation of concerns that helps meet the design goals of modularity, extensibility, and defined responsibility areas. Figure 4.4 illustrates how the components will interact as part of the matching process. The Bronze-To-Silver transformation pipeline initiates the Coordinator, which then orchestrates the three-phase matching flow through the Orchestrator. The Orchestrator coordinates the interaction between the Data Access Layer (for read/write operations of identity records) and the similarity calculation component.

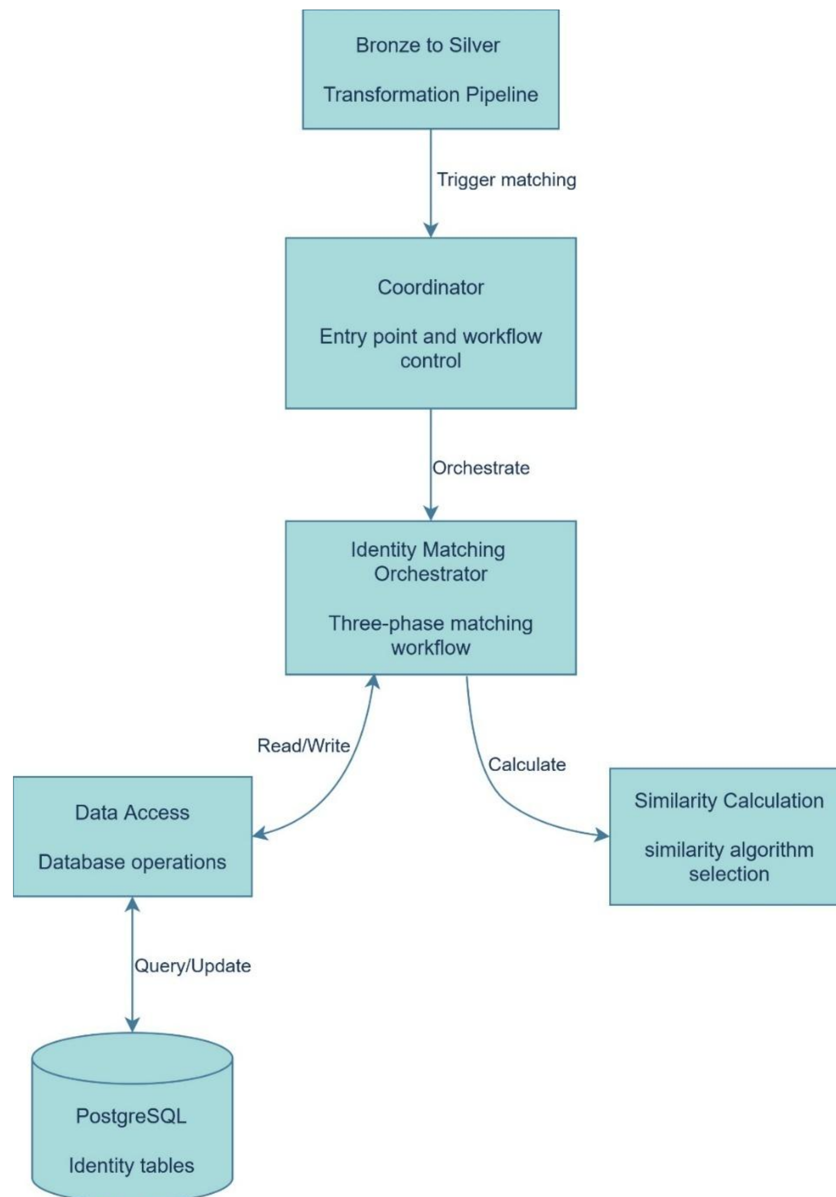


Figure 4.4: Component architecture and interaction flow for the IMS

4.3.1 Coordinator

The Coordinator is an application-level entry point for identity resolution in MECOIS and is called by the BronzeToSilverTransformer once raw profiles are stored in the database.

The Coordinator has three key functions in the architecture. Firstly, it provides the only access point to the identity matching subsystem in the data transformation pipeline. Secondly, it does the job of loading the identity block definitions from the data lake transformation config file that contains the information on how to extract identity information (name, email,

username) from each of the data sources. Thirdly, it controls the sequence of the identity management phases and post-processing steps; this comprises the decisions as to when to execute the profile extraction, matching, and consolidation functions.

It is the single entry point between the data transformation pipeline and the identity matching subsystem. The Coordinator relies on the identity mapping configuration (which is stored in data_lake_transformation.json), the Data Access Layer, and the Matching Orchestrator to fulfill its role. The Bronze-To-Silver transformation pipeline relies on the Coordinator as the entry point into identity resolution.

4.3.2 Data Access

Having an independent Data Access Layer ensures that business logic will not rely upon what data persistence methods are being utilized.

There are three key functions in which the Data Access Layer plays a role within the overall architecture. The first function of it is providing data abstraction by removing the details of the PostgreSQL implementation from the rest of the application. Secondly, the Data Access Layer performs management of transaction commits and the lifecycle of database connections. This ensures that data consistency is maintained during each operation. Thirdly, the Data Access Layer provides encapsulation of queries. In doing so, it confines all SQL queries to itself rather than allowing them to be distributed throughout the business logic.

Some of the main responsibilities of the Data Access Layer include performing CRUD (Create, Read, Update, Delete) operations on identity tables, retrieving profile information based upon their current processing state, and utilizing bulk inserts to improve performance when ingesting multiple large batches of raw profiles at once.

Separation of the Data Access Layer from the business logic also enables independent testing of the business logic and provides the ability to isolate changes to the database schema as well as provide maintainable boundaries between the workflow logic and data persistence. As such, the Orchestrator and Coordinator can focus solely on matching logic without regard to storage issues.

4.3.3 Matching Strategy

The option to select multiple similar calculation methods was an architectural decision for the overall system, as defined by NFR-5. An architectural decision such as this provides the opportunity to evaluate and improve different matching methods without having to change the core workflow of matching. The system uses email similarity as its primary matching factor, which is further explained in section 5.1.2.

The matching strategy has three primary functions within the architecture. First, it provides a layer of algorithms that allows for the separation of similarity calculations from the matching workflow, thereby providing isolation for the orchestration logic from the similarity calculation logic. Second, it provides flexibility because users can configure what algorithm

they want to utilize versus introducing changes to the source code. Third, it provides extensibility via the ability to add new similarity algorithms without affecting the current functionality of the system.

There are three types of algorithms used in the prototype, previously discussed in section 2.2.2. The SequenceMatcherStrategy utilizes Python's difflib for a character-by-character comparison of the two input strings and is the default method. The LevenshteinStrategy measures the minimum number of edits required to transform one string into another and uses the edit distance to calculate the similarity. The JaroWinklerStrategy utilizes a prefix-weighted similarity and provides higher scores when the strings have the same characters at the beginning of the string.

The separation between similarity calculation and matching logic means that improvements in one area do not require changes to the other.

4.3.4 Identity Matching

The Identity Matching Orchestrator implements the main matching algorithm that identifies and connects the same identity information that exists in various data systems. From an architectural perspective, the Orchestrator acts as the business logic layer that interacts with the Data Access Layer and the similarity calculations to implement the overall matching process.

Within the overall architecture, the Orchestrator has three key responsibilities. It executes the multi-step process used to perform the identity match as previously discussed. Secondarily, the Orchestrator provides decision-making logic to determine if two profiles should be automatically combined, manually reviewed, or treated as distinct profiles. Lastly, the Orchestrator provides process state management to track which profiles have been processed in each of the phases so that duplicate profile processing does not occur.

Three-Phase Architecture:

The Orchestrator uses a three-phase pipeline to match different types of matches and relationships. Phase 1 (Cross-Check) compares all new profiles to the existing merged identities to ensure that no duplicate merged identities are created. Phase 1 produces either profiles that have been auto-merged or candidate groups for manual review.

Phase 2 (Extended Cross-Check): This phase compares the remaining profiles to the new merged identities produced in Phase 1. The goal of this phase is to identify transitive relationships that may not be obvious when comparing the original two profiles. Phase 2 produces additional candidate groups.

Phase 3 (Internal Processing): This phase uses graph-based matching algorithms to process the remaining profiles and create groups of related profiles that did not have a previously merged identity. Phase 3 produces new merged identities or candidate groups.

Using a single-pass approach to match identities would likely result in both missing

transitive relationships and creating “split” identity groups, whereas the staged approach guarantees complete identity matching while also improving performance via progressive filtering. Once the three stages are completed, a final consolidation step is performed to eliminate redundant candidate groups to maintain a clean database after multiple pipeline runs.

5 Design and Implementation

This chapter outlines the design choices made to create the IMS and the development process that led to a working version. The technical process of creating each component is explained in detail within this chapter. While chapter 4 provided the overall architecture of the system and defined what responsibility was assigned to each component, this chapter explains the technical implementation of the components and how the design patterns they employed enabled them to perform their designated functions.

Firstly, the design patterns used to implement the entire system will be discussed (section 5.1), followed by the detailed implementation of all of the main components (section 5.2). Secondly, the design of the database schema for the system will be explained and the reasoning behind all the structural design decisions will be documented (section 5.3). Lastly, information regarding the technology stack and libraries that were utilized during the development of the prototype will be provided (section 5.4).

5.1 Design Decisions

Prior to developing the IMS, key design decisions were made in order to inform the direction of development. The primary focus was on how components of the system would communicate with each other, how algorithms used for determining an accurate match could be exchanged without modifying code, and how data storage or data retrieval could be abstracted from business logic.

The design decisions were organized into three categories: structural design patterns that define where responsibilities are placed within code, algorithmic design choices that determine the accuracy and performance of the matching algorithms, and dependency management design choices that affect testability. Some of these design patterns were applied uniformly across the entire system, while others were only partially applied because the requirements were evolving during its development.

This section addresses the structural design patterns that influenced the implementation (section 5.1.1) and the algorithmic design choices that influence the matching process (section 5.1.2).

5.1.1 Applied Design Patterns

Three design patterns shaped the structure of the system: the strategy pattern for selecting an appropriate algorithm, the Data Access Object (DAO) pattern for abstracting how to access the data, and dependency injection for managing dependencies between components. Each of these design patterns was used to solve a particular problem within the system's design.

Strategy Pattern for Similarity Calculation

The strategy pattern (see Figure 5.1) was used to define the similarity calculation layer to allow runtime choice of an algorithm when calculating similarities without altering the overall flow of the matching process. Strategy provides a way to define a family of algorithms, encapsulate each one, and make them interchangeable (Gamma, Helm, Johnson, & Vlissides, 1995). It does this by establishing an abstract base class, which serves as a contract for similarity calculations, allowing various strategies (SequenceMatcher, Levenshtein, and Jaro-Winkler) to be easily interchanged.

There are several reasons why this pattern was used. First, the best algorithm to use for a given dataset will depend upon the nature of the email data being processed (the email patterns are quite varied depending upon the organization). Second, the similarity thresholds that are appropriate for one strategy may not be suitable for other strategies. Third, future research could discover new strategies (for example, machine learning-based approaches) that would be useful to add to the system without having to refactor the core logic of the matching system.

The greatest advantage of this implementation method is its flexibility. A domain expert can now choose an appropriate strategy for computing similarities by configuring the system without requiring code modifications. This implementation incurs some performance overhead because of its polymorphic nature. However, the cost of this performance penalty is negligible when compared to the computational costs of the actual matching algorithms.

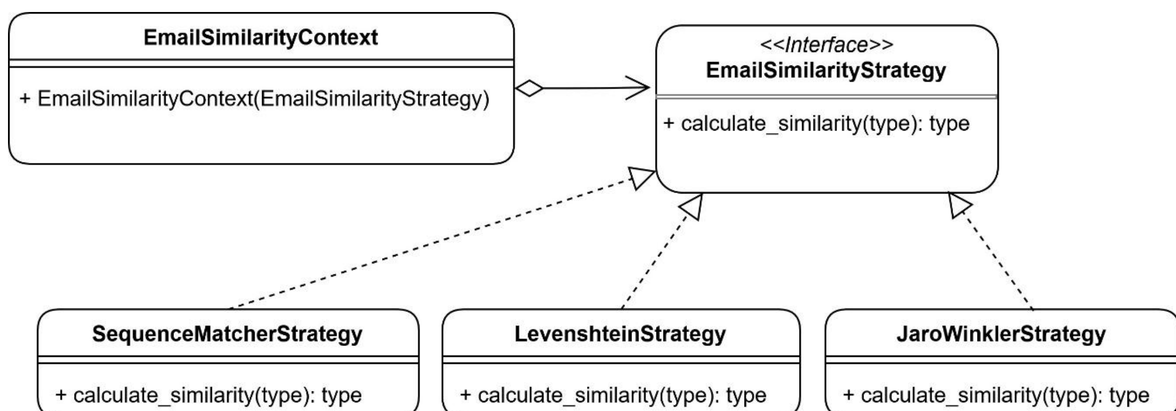


Figure 5.1: Strategy pattern example for email similarity algorithms

Data Access Object Pattern for Data Persistence

The implementation of the DAO pattern in the class IdentityDataAccess provides an abstraction between the database operations and the business logic (Alur, Malks, & Crupi, 2001). All the SQL statements are contained within this class, allowing the Orchestrator and the Coordinator to be database agnostic. This separation was necessary for both maintainability as well as testability of the code. In the absence of this separation, SQL would be scattered throughout the Orchestrator, tightly coupling the business logic to PostgreSQL and making its isolation tests difficult. The DAO pattern enables unit testing of the matching algorithms by allowing SQL statements to be mocked or replaced with substitute test functions.

The implementation of the DAO pattern packs all of the CRUD operations, transaction handling, and query-building functions in a single class. Methods such as get_unmerged_profiles_raw and insert_merged_identity encapsulate specific parts of database operations and hide SQL details, such as NULL handling in the WHERE clauses, as well as batch insert optimization functions from the business logic.

The primary advantage of the DAO pattern is the isolation of database access into a single, isolated layer, allowing for better testability and maintainability of the code. If at any later point, MECOIS was moved to a new database system or specific queries needed to be optimized, then the changes would need to be done to this single class. The disadvantage is that another level of abstraction is applied, which adds some level of indirection in the handling of database operations.

Dependency Injection for Component Composition

Dependency injection is a software design pattern that is used to pass dependencies to an object as opposed to having those dependencies created within the object itself (Fowler, 2004). Using this model allows for better testability (by substituting mock implementations for testing), as well as greater flexibility (because it decouples object creation from object usage). There were a few places throughout this prototype where dependency injection was implemented inconsistently. Some classes received their dependencies via their constructors, while others created their own dependencies internally.

In the case of the Orchestrator class, it is a good example of dependency injection, as it receives two of its dependencies (the Data Access Layer and the similarity strategy) via its constructor. This design makes it easy to swap out one implementation of a dependency for another when you need to run tests or when you need to move your app between development and production environments. The Coordinator has these dependencies instantiated and then injected into the Orchestrator during initialization.

Dependency injection, however, was used inconsistently throughout the system. The Coordinator receives connection parameters and creates the Data Access Layer instance internally instead of passing a pre-configured instance of the Data Access Layer that is externally configured. The same is true for the BronzeToSilverTransformer, which directly embeds database credentials when instantiating the Coordinator and instead should pass an already created (and thus configured) Coordinator to the BronzeToSilverTransformer.

Table 5.1 summarizes the three design patterns applied throughout the implementation and their respective contributions to the system's architecture.

Summary of Pattern Application:

Pattern	Component	Implementation Status	Primary Benefit
Strategy	Similarity Calculation	Fully implemented	Algorithm flexibility
DAO	Data Access Layer	Fully implemented	Database independence
Dependency Injection	Prototype-wide	Partially implemented	Testability (where applied)

Table 5.1: Overview of used design patterns

The following section examines the algorithmic design choices that complement these structural design patterns.

5.1.2 Algorithm Design Choices

There were several algorithmic choices that were necessary for the design of the structural design patterns. However, there were also algorithmic decisions that will affect how well the IMS will perform that are based on the matching workflow, threshold values for categorizing similarities between profiles, and methods for managing groups of related profiles.

Three-Phase Approach

The first major choice to be made in terms of implementing a matching workflow was to use a three-phase matching model rather than a one-time pass through the data. This choice came from initial testing experiences, which exposed a serious flaw in the design: profiles with transitive relationships were being split between two or more merged identities when those profiles belonged together.

A single-pass model would process each profile one at a time and compare it to the merged identities that existed up to that point in the processing sequence. Unfortunately, for certain types of profiles, if Profile A and Profile B were very similar to each other (both variations of "john.smith") but Profile B arrived in the system after Profile A had already been merged with another identity, the similarity based on their email addresses may not have been sufficient to match them against the representative email of the merged identity of Profile A. Thus, Profile B would be created as its own new identity because it did not meet the threshold for merging with Profile A, even though Profiles A and B were obviously intended to represent the same individual.

This flaw was corrected using a three-phase model. Phase 1 processed simple matches against the existing identities. Phase 2 targeted the transitive case by allowing the remaining profiles to be compared against the representative emails of the identities created in Phase 1 as well as all of the emails that were included in the creation of those identities. Phase 3 then allowed the remaining unprocessed profiles to compare to each other in order to find groups that did not exist previously as merged identities. Although the additional phases increase the amount of work required to perform a match, they greatly improve its quality. For example, the transitive case would never be solved using a single-pass model, and many groupings that could have been found by simply grouping profiles based on their email addresses would have gone unnoticed. Because the match is performed in batch mode and not in real-time, the increased computing resource requirements for the three-phase model is acceptable.

Email as the Primary Matching Key

The primary match factor used by this system is email address. In section 2.2.1 we have already demonstrated that email provides a strong identity signal since it is stable, unique, and consistently present on multiple platforms, including GitHub, GitLab, and Jira. Names are variable (the format varies and can be changed) and therefore less consistent than emails. Likewise, usernames are platform-specific and do not support cross-platform linking of identities; however, email is both human-readable for validation purposes and supports cross-platform identity matching.

Similarity Threshold Selection

Two specific similarity threshold levels are used in the system to identify matches and group them into categories: an automatic merge threshold of 0.90 and a candidate threshold of 0.70 for matches that will be manually reviewed. Thresholds for both similarities were established based on iterative testing with example MECOIS sample data utilizing the SequenceMatcher strategy. Although calibrating the system with a validation set that contains known correct matchings provides a more objective method for establishing threshold values (Elmagarmid, Ipeirotis, & Verykios, 2007), the focus of the prototype scope was on establishing functional matching workflows. The selected thresholds worked acceptably well with the test data by avoiding obvious false positives and identifying obvious variation differences. Future development of the system will include further threshold calibration as described in section 6.3.

The auto-merge threshold of 0.90 represents near-exact matches of the local part of emails, such as “john.smith” and “johnsmith1”. Given the very few false positive matches at this level of similarity, the cost of automatically merging can be accepted. Setting this threshold above 0.90 (e.g., 0.95) would lead to missed valid matches due to slight differences in local parts, while setting this threshold below 0.90 (e.g., 0.80) would establish unnecessary risks of merging individuals who have coincidentally similar email patterns.

Matches that fall within the 0.70-0.89 similarity range have the potential to represent either the same individual or two distinct individuals with coincidentally similar naming patterns. This is why they are referred to as candidates and are flagged for human review rather than

allowing potentially incorrect automatic merging decisions. There is a buffer zone created between the two threshold levels where human decision-making is explicitly required.

Similarity profiles below 0.70 are considered unrelated and therefore are not grouped together. If a different similarity strategy (like Levenshtein or Jaro-Winkler) is used, the current threshold values used for mainly SequenceMatcher may need to be recalibrated. Each strategy produces a different distribution of scores for the same pair of email addresses.

Connected Components for Phase 3

With Phase 3, the algorithm compares the remaining profiles with no existing merged identities and applies a connected components algorithm instead of sequentially comparing pairs. The reason for this decision was to avoid a particular problem: profiles with transitive relationships being split into separate groups because of the processing sequence.

To illustrate this, consider the three profiles X, Y, and Z. Profiles X and Y have a similarity score of 0.95 and should be merged, whereas Y and Z have a similarity score of 0.75 (merge candidates). Using the sequential processing method, if X is processed first, it might be merged with Y because of the high similarity score. Next, when Z is added, it will only be compared to the representative profile from the X-Y merge. If Z's similarity to the representative is less than 0.70, then Z will become a separate identity. Although Z had an original similarity of 0.75 with Y.

In contrast, the connected components method constructs a graph of profiles where the profiles are the nodes and similarities above 0.70 represent edges between them. The connected components algorithm then takes a depth-first search (DFS) through the graph to identify all connected components (Cormen, Leiserson, Rivest, & Stein, 2009). Each identified component is treated as a single entity: if the greatest similarity value within the component is greater than or equal to 0.90, then the entire component is merged into a single identity. If the greatest similarity value falls between 0.70 and 0.89, then the entire component is treated as a candidate group. This is illustrated in Figure 5.2:

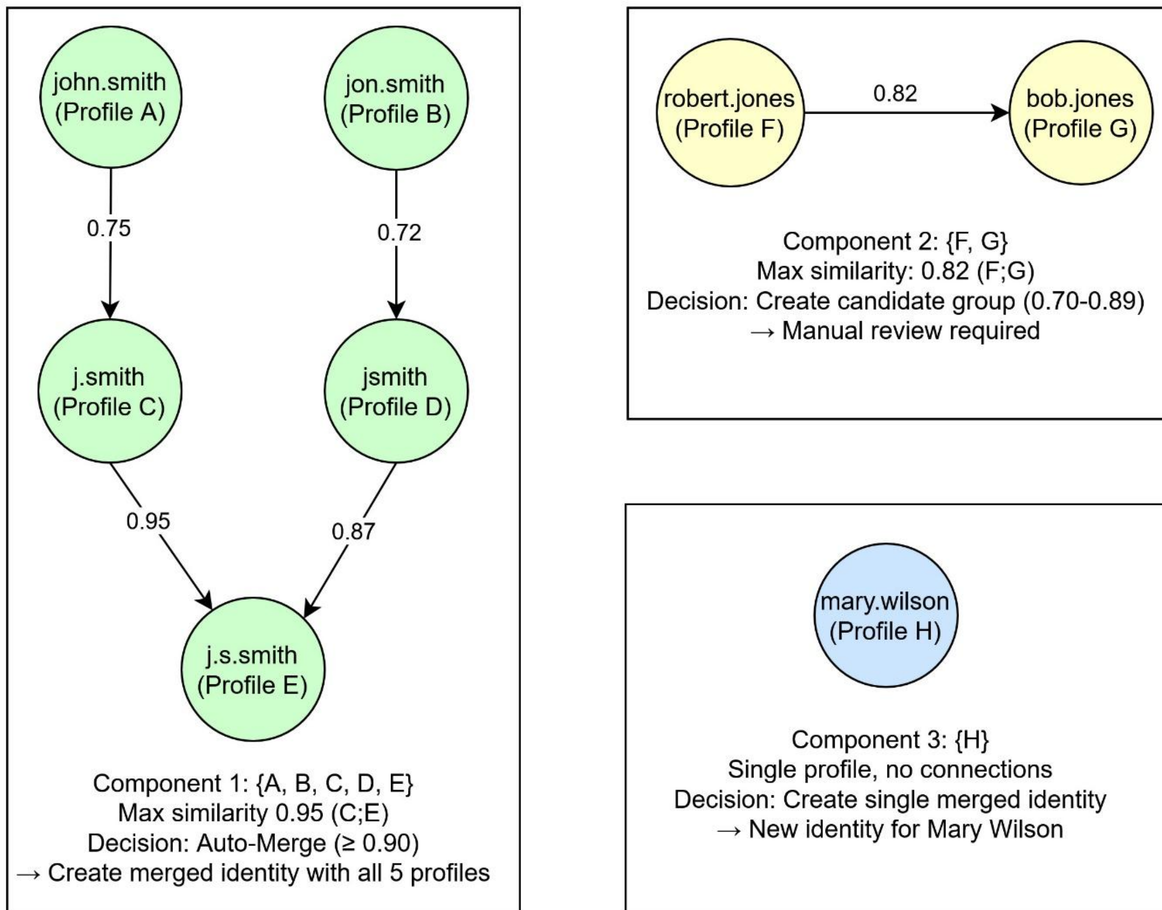


Figure 5.2: Example of a connected components algorithm result in Phase 3.

X, Y, and Z would then be treated as a connected set of profiles through their similarity relationships, and they would therefore be processed as a single decision, rather than being artificially separated due to the order in which they happened to be processed. A major trade-off here is the increased complexity of the computation process. Creating the graph requires $O(n^2)$ similarity comparisons within each domain group. Therefore, when there are ‘n’ number of profiles in a given domain, every profile must be compared to every other profile to build the graph, leading to $n \times (n-1)/2$ total pairwise comparisons. As such, for example, 10 profiles result in 45 total comparisons, whereas 100 profiles result in 4950 comparisons. While the increased number of comparisons may seem excessive at first, this overhead is deemed tolerable in exchange for the improved accuracy of the matches, and the next paragraph provides an additional reduction of ‘n’.

Domain Grouping Optimization

The computational overhead associated with making similarity comparisons can be lessened by grouping profiles by e-mail address domain prior to processing. The basis for this optimization is a fundamental understanding of the nature of cross-organization identity-based matching. As such, profiles from differing domains will never match, as there exist significant confounds introduced by cross-organization identity matching; specifically, when a developer changes organizations, their email domain also changes, and therefore the organizational context (development processes, team structures, tools utilized, and work culture) provides no clear means to isolate the environmental influences versus the individual influences on observed behaviors. Due to this fact, we have limited our identity-based matching to occur only within the same organizational bounds (i.e., email domains). Therefore, analysis based on the results of our matching process will always reflect the consistent organizational context in which that comparison was made. By limiting the comparison space through grouping profiles by domain, we significantly reduce the number of potential comparisons. For example, with a dataset of 10,000 profiles distributed across 100 domains, instead of performing comparisons between every possible pair (which totals 50,000,000 comparisons), we now only compare pairs within each of the 100 domain groups. In the case of an evenly distributed dataset, this would result in approximately 500,000 comparisons, a two-order-of-magnitude reduction in the comparison space, and thus, make the previously described $O(n^2)$ approach practical for use on large-scale real-world datasets.

Summary

Algorithm design choices were made to enhance both matching accuracy as well as the management of edge cases and not just to optimize processing speed. The three-phase approach, two threshold values, and connected components algorithm are designed to keep the system from experiencing the specific types of failures that were identified through the software development process (split groups, missing transitive relationships, and order-dependent results). Although increasing system complexity was the result of making these choices, they help ensure the matching system will produce reliable, quality results for research data analysis applications where accuracy is a significant concern.

5.2 Component Implementation

This chapter offers context for the implementation of the individual architectural components of the IMS described in section 4.3. The focus will be on how the components were made, the issues that arose during implementation, and how they were resolved. Each subsection will provide an overview of the major implementation decisions made and describe one of the following components: the Coordinator (section 5.2.1), the Orchestrator (section 5.2.2), the Data Access Layer (section 5.2.3), and the Matching Strategy (section 5.2.4).

5.2.1 Coordinator Implementation

The IdentityMatchingCoordinator acts as the entry point to the identity matching subsystem. While its architectural function was described in section 4.3.1 this chapter will provide information about how the Coordinator was actually implemented and why some of the implementation decisions were made.

Multiple Responsibilities Justification

The Coordinator is responsible for both the workflow orchestration and the identity data extraction. This violates the single responsibility principle, which states a class should only have one reason to be changed (Martin, 2003). Additionally, further decompositions were intentionally avoided due to a few factors.

Firstly, in the context of the MECOIS pipeline, identity extraction and matching workflow orchestration are tightly coupled. Therefore, the Coordinator must first extract identity information from source DataFrames and then store this identity information within the database prior to initiating the matching workflow. If we had to create separate classes to handle each concern, there would be a need for even more inter-class coordination, thereby increasing the number of issues vs. solutions.

Secondly, adding additional abstraction layers will increase unnecessary complexity for a prototype. Although a production system with a broader scope may be able to utilize additional abstraction layers for further decomposition, for the purposes of the thesis prototype, the increased complexity will far exceed any benefits. Due to its size (less than 200 lines of code), the Coordinator can still be maintained despite being responsible for multiple tasks.

Key Implementation Details

The Coordinator's implementation can be broken down into five main responsibilities:

Identity Mapping Configuration: The Coordinator uses a JSON (JavaScript Object Notation) config file to read how to map identity columns from all of the source's data. The mapping specifies which column in each of the source's data corresponds to identity fields (name, email, username, user_id). For example, if we are using GitHub then it will have "author_email" but if we are using GitLab it will use "user.email" for the same semantic concept. Therefore, this configuration-driven approach makes it so that new data sources do not require changing code.

Profile Extraction and Insertion: The Coordinator's insert_new_raw_profiles method extracts the identity information from the Spark DataFrames by utilizing the loaded configuration. It uses PySpark's Resilient Distributed Dataset (RDD) operation to flatten the identity blocks because one source record could contain multiple identities (i.e., author and committer). Then it filters all blacklisted emails. Finally, it inserts the new profiles into the database via the Data Access Layer.

Orchestrator Initialization: The Coordinator instantiates the IdentityMatchingOrchestrator and injects the appropriate dependencies into it. It creates the Data Access Layer during Coordinator initialization, whereas the similarity strategy is injected through the Coordinator's constructor. This shows proper dependency injection as described in section 5.1.1.

Matching Execution: The Coordinator will load the unprocessed profiles and the existing merged identities from the database and will then delegate the execution of the matching process to the IdentityMatchingOrchestrator. Once the matching process has been completed, the Coordinator will commit the transaction using a Data Access instance.

Three-Phase Matching Post-Processing: Once the three-phase match process has been completed, the Coordinator will trigger the post-processing step of consolidation. Consolidation identifies redundant candidate groups stored within the database and deletes them. Candidate groups are considered "redundant" if a larger candidate group completely encompasses all members of a smaller candidate group (i.e., when profiles A, B & C make up a candidate group and profiles A & B do as well, then the candidate group consisting of A & B would be redundant). Consolidation acts primarily as a secondary layer of defense against edge cases such as unexpected data quality issues and race conditions that may arise from concurrent processes during normal operation. During normal operation most redundancies can be prevented by the reuse logic in candidate group creation and the Phase 3 exclusion of profiles already in candidate groups.

Integration with Bronze-to-Silver Pipeline

The Coordinator is called by the BronzeToSilverTransformer when the source data has been processed into silver-level Spark DataFrames. The BronzeToSilverTransformer uses the Coordinator instance with the database parameters and a similarity strategy, invokes its methods to insert profiles as well as executes matching, and continues on to do UUID enrichment on the DataFrame. The UUID enrichment is done by the BronzeToSilverTransformer, which pre-fetches email-to-UUID mappings using the Coordinator's get_merged_identity_uuid_for_email method, creates an in-memory dictionary, and broadcasts it to the Spark workers to prevent database connection serialization issues.

A different approach is required for the UUID enrichment due to the distributed nature of Spark's architecture. Database connections cannot be serialized and shipped out to the worker nodes; therefore, the BronzeToSilverTransformer pre-fetched all of the email-to-UUID mappings in memory on the driver, built a dictionary, and broadcasted it to the worker nodes. Now each worker node will have access to an in-memory dictionary to look up UUIDs versus trying to make individual database connections.

5.2.2 Orchestrator Implementation

The IdentityMatchingOrchestrator contains the core matching logic. It was the most complex of the components to implement due to its responsibility for both coordinating the three-phase matching algorithm and keeping track of the state between phases.

Three-Phase Algorithm Implementation

The three-phase algorithm is implemented through a set of methods, each representing one phase of the algorithm. The benefit of separating the three phases into individual methods is that they are much easier to follow and maintain than a single large method, which would have represented all three phases.

Phase 1: Initial Cross-Check

The Phase 1 cross-check process is the initial comparison of unprocessed profiles versus all existing merged identities. To help the processing speed by reducing the number of potential matches, the implementation groups both the unprocessed profiles and the existing identities by email domain.

The implementation calculates the similarity for each pair of an unprocessed profile and an existing identity within each domain to create a mapping of similarities between the two. Based upon the similarity mapping, the profiles are assigned to one of three categories: If the similarity value is equal to or greater than 0.90, the unprocessed profile is automatically merged into the existing identity. If the similarity value is between 0.70 and 0.89, the unprocessed profile is placed in a candidate group for that existing identity.

Another key consideration in the implementation is the identification of processed profiles, so they do not get reprocessed in a subsequent phase. A list of processed profile IDs will be maintained throughout the phase to ensure this does not occur.

Phase 2: Extended Cross-Check

In Phase 2, the tool will identify transitive relationships based on remaining profiles being compared against all emails of newly formed identities. Once a merged identity has been created, the prototype retains a single representative email within the merged_identities.email field; however, as new profiles continue to be added to this identity, the raw_profiles table continues to collect other email addresses all pointing to the same merged identity. Therefore, in Phase 2, the tool compares all emails associated with merged identities with all remaining profiles, not simply the representative email.

The major challenge faced while implementing Phase 2 was how to retrieve all emails associated with each newly formed identity in an efficient manner. A naive approach to implementing this phase could have included a separate database query for each identity that had been merged. This approach would be inefficient, given the possibility of merging hundreds of identities. An alternative solution implemented in Phase 2 was to call the get_all_emails_for_merged_identity method to execute a bulk query to retrieve all relevant emails for all identities at once. All retrieved emails were then stored in memory organized by identity ID. This resulted in a significant reduction in the number of database round trips, from potentially hundreds to only one.

For similarity calculation the tool calculates similarity against every email associated with each newly formed identity and takes the highest value as the representative measure of the similarity between the profile and the identity. This allows for identification of transitive

matches that would otherwise be missed because the profile email does not match the representative email.

Phase 3: Internal Processing with Connected Components

Phase 3 utilizes a graph-based approach to group remaining profiles. The phase builds an adjacency list of the similarity graph, as follows: Each node in the list represents a profile, and the edges represent connections from one node to another if their similarity is greater than or equal to 0.70.

The connected components are determined by utilizing DFS, a graph traversal method that continues to explore down as far as possible along each branch before backtracking (Cormen, Leiserson, Rivest, & Stein, 2009). Beginning with a previously unvisited profile, DFS will continue to traverse all similarity edges, identifying all other profiles that can be reached via the similarity graph. Once all profiles have been visited, they form one connected component. This process is repeated for each unvisited profile until all components have been identified. The DFS implementation utilized a stack-based approach (see Listing 5.1) instead of a recursive one to prevent the possibility of reaching a "stack overflow" with larger data sets:

```
1. def find_connected_component(start_id, graph, visited):
2.    component = []
3.    stack = [start_id]
4.    while stack:
5.       node_id = stack.pop()
6.       if node_id not in visited:
7.          visited.add(node_id)
8.          component.append(node_id)
9.
10.         # Add unvisited neighbors to stack
11.         for neighbor in graph[node_id]:
12.            if neighbor not in visited:
13.            stack.append(neighbor)
14.   return component
```

Listing 5.1: DFS implementation for finding connected components

Once a connected component has been identified, the implementation identifies the highest similarity value among all of the profiles in that component. If the highest similarity value is greater than or equal to 0.90, then the entire component of profiles will be merged into a single profile. If the highest similarity value is less than 0.90 but greater than or equal to 0.70, then all of the profiles in this component will be classified as a candidate group.

An overview of the three phases and the conditions of their actions is given below in Table 5.2.

Similarity Score	Phase 1 Action	Phase 2 Action	Phase 3 Action
>= 0.90	Auto-merge to existing identity	Auto-merge to newly merged identity	Auto-merge entire component
0.70-0.89	Create a candidate group with an existing identity	Create a candidate group with a newly merged identity	Create a candidate group
< 0.70	No action (continue)	No action (continue)	Create a single merged identity

Table 5.2: Matching algorithm decision matrix.

State Management Across Phases

A big challenge with implementing the application was how to track what profiles had been processed in the three phases. All three phases must know what profiles had already been processed by other phases so that they do not process the same profiles twice.

To implement this, the application used a set-based method to share the status of what profiles had been processed by each phase. After each phase processes its profiles, it passes on a list (or "set") of all the IDs of those profiles to subsequent phases. The subsequent phases then filter out those IDs from their inputs to ensure no duplicate work is being done. This is more efficient and easier to understand than using database queries for tracking and it allows for better visibility of what remains after each stage.

Performance Optimization: Domain Grouping

Domain grouping is implemented with a helper method called group_profiles_by_domain, which will organize all the profiles as shown in Listing 5.2:

```
1. def group_profiles_by_domain(self, profiles):
2.     domain_groups = defaultdict(list)
3.     for profile in profiles:
4.         domain_groups[profile["domain"]].append(profile)
5.     return domain_groups
```

Listing 5.2: Domain grouping to limit comparison scope

The method uses Python's default dict to automatically add an empty list for a new domain to avoid checking if the domain string exists or not. The returned dictionary has a domain

string mapped to a list of profiles; the matching algorithm can now process these two independent lists. This simple implementation provides the complexity reduction described in section 5.1.2, without increasing the code complexity significantly.

5.2.3 Data Access Layer Implementation

The IdentityDataAccess class encapsulates all database operations. As discussed in section 4.3.2 this section focuses on specific implementation challenges that emerged during development.

NULL-Aware Deduplication

A bug that was immediately identified during the testing phase involved duplicate profiles being written into the database even though a duplicate checking mechanism was present in the code. The issue came from the way SQL treats NULLs in comparisons.

The original code used standard equality checks in SQL WHERE clauses as shown in Listing 5.3:

```
1. WHERE name = %s AND email = %s AND username = %s
```

Listing 5.3: Naive WHERE clause failing NULL comparison

However, in SQL, NULL = NULL will evaluate to NULL (unknown) rather than TRUE. Therefore, profiles that have NULL values in any field would never be identified as duplicates.

The resolution to this issue was to create dynamic WHERE clauses in SQL (Listing 5.4) that check for NULL values:

```
1.  row_conditions = []
2.  for col, val in zip(identity_columns, row):
3.     if val is None:
4.        row_conditions.append(f"{col} IS NULL")
5.     else:
6.        row_conditions.append(f"{col} = %s")
7.  conditions.append(f"({' AND '.join(row_conditions)})")
```

Listing 5.4: NULL-aware deduplication with compound WHERE clause construction

This generates SQL as depicted in Listing 5.5:

```
1. WHERE name = 'John' AND email IS NULL AND username = 'jsmith'
```

Listing 5.5: Resulting NULL-safe WHERE clause

This method allows for proper identification of duplicates regardless if NULL values exist in some fields.

Batch Operations for Performance

Individually inserting each profile into the database using SQL would be an extremely long process when dealing with thousands of profiles. Using psycopg2.extras.execute_values allows us to perform bulk inserts into the database. A “bulk” insert allows us to send multiple rows in one database round trip instead of individually sending each row.

The function processes profiles in batches of 1,000 records at a time. For each batch, it queries to check which records already exist in the database and inserts only the new ones. Processing the profiles in chunks offers the following advantages: it breaks down the deduplication query into smaller pieces (making it less complex), allows for easier error checking, and reduces the size of the database transaction to a manageable amount. The use of a 1,000-record batch is a reasonable value that balances query complexity with insert efficiency.

Candidate Group Reuse Logic

To avoid duplicate candidate groups during pipeline runs, the implementation has reuse logic that adds new candidates to existing candidate groups instead of making a new one.

For internal candidate groups, the add_to_merge_candidates method finds any existing groups that overlap with the new profile IDs. If an overlapping group is found, the new profiles are appended to it. If there are multiple candidate groups that contain similar candidate IDs, the function selects the candidate group that has the highest number of overlapping IDs to limit the number of redundant groups.

For cross-check candidate groups, the add_cross_check_candidate_group method limits each merged identity to be associated with only one active (i.e., groups that have not been reviewed yet) group. Therefore, when additional candidate profiles are identified as matching an existing merged identity, those profiles will be added to that merged identity’s current candidate group and not create a new group. The solution accomplishes this by querying for the most recently created candidate group that matches the specific merged identity ID.

5.2.4 Matching Strategy Implementation

The similarity calculation strategies implement the abstract base class (see section 5.1.1) via the strategy pattern. Given two e-mail local parts, every strategy will calculate a similarity score that falls within the range from 0.0 (entirely dissimilar) to 1.0 (the same).

Abstract Base Class

The EmailSimilarityStrategy abstract base class (Listing 5.6) utilizes Python’s abc module to define the interface:

```
1. from abc import ABC, abstractmethod
2.
3. class EmailSimilarityStrategy(ABC):
4.     @abstractmethod
5.     def calculate_similarity(self, local_part1: str, local_part2: str) -> float:
```

Listing 5.6: EmailSimilarityStrategy abstract base class

All concrete strategies must implement this single method. The return value is always a float between 0.0 (completely different) and 1.0 (identical).

Every concrete strategy must provide an implementation of the single method shown above in Listing 5.6.

SequenceMatcher Strategy (Default)

The default strategy uses Python's built-in difflib.SequenceMatcher class. This implementation (see Listing 5.7) is simple because the heavy lifting is done by the standard library:

```
1. from difflib import SequenceMatcher
2.
3. class SequenceMatcherStrategy(EmailSimilarityStrategy):
4.     def calculate_similarity(self, local_part1: str, local_part2: str) -> float:
5.         return SequenceMatcher(None, local_part1, local_part2).ratio()
```

Listing 5.7: SequenceMatcherStrategy for subsequence-based similarity

As such, the SequenceMatcher is simple to use since it utilizes the Ratcliff/Obershelp algorithm to find the longest continuous matching subsequence and then calculates how similar they are based on that. It also works well with email addresses since it can account for differences in the ordering of characters.

Levenshtein Strategy

The Levenshtein strategy is an efficient method of computing the edit distance between two strings. The edit distance represents the minimum number of single-character operations (deletions, insertions, substitutions) necessary to transform one string into another. Since edit distance increases with dissimilarity, it must be normalized and inverted to produce a similarity score as shown in Listing 5.8:

```
1. class LevenshteinStrategy(EmailSimilarityStrategy):
2.     def calculate_similarity(self, local_part1: str, local_part2: str) -> float:
3.         distance = self._levenshtein_distance(local_part1, local_part2)
4.         max_len = max(len(local_part1), len(local_part2))
5.
6.         if max_len == 0:
7.             return 1.0
8.
9.         return 1.0 - (distance / max_len)
```

Listing 5.8: LevenshteinStrategy for edit distance-based similarity

Dynamic programming was used as the method to compute the edit distance efficiently. The length of email local parts is typically 5-20 characters long, which makes the algorithm run quickly and use minimal memory.

Jaro-Winkler Strategy

The Jaro-Winkler strategy provides a weighted factor on matching prefixes; this is useful when you have many email address patterns that contain organizational prefixes (i.e. “company.john” vs. “company.jane”).

In the implementation, the base Jaro similarity is computed, and then the Winkler modification is applied to common prefixes for up to four characters:

```
1. class JaroWinklerStrategy(EmailSimilarityStrategy):
2.     def __init__(self, prefix_weight=0.1):
3.         self.prefix_weight = prefix_weight
4.
5.     def calculate_similarity(self, local_part1: str, local_part2: str) -> float:
6.         jaro_sim = self._jaro_similarity(local_part1, local_part2)
7.
8.         # Find common prefix length (up to 4 chars)
9.         prefix_len = 0
10.        for i in range(min(len(local_part1), len(local_part2), 4)):
11.            if local_part1[i] == local_part2[i]:
12.                prefix_len += 1
13.            else:
14.                break
15.
16.        # Apply Winkler modification
17.        return jaro_sim + (prefix_len * self.prefix_weight * (1 - jaro_sim))
```

Listing 5.9: JaroWinklerStrategy with prefix weighting

The Jaro similarity calculation is more complex in comparison to the Levenshtein strategy. It includes the detection of matches within a distance window and also counts the number of transpositions. The complete implementation of Jaro similarity is part of the codebase; however, for the sake of brevity, it has been omitted from above (see Listing 5.9).

Strategy Selection

Selection of the strategy occurs via dependency injection at the time of instantiation. The Coordinator accepts an optional similarity_strategy parameter; therefore, the Coordinator defaults to Sequence-Matcher-Strategy if no strategy is provided. This provides flexibility to select different strategies by supplying the corresponding strategy instance at the time of Coordinator creation.

5.3 Database Schema

The three-phase matching process is supported through a database schema with a clearly defined separation of raw input data, processed identity information, and profiles awaiting manual review in the database. While section 4.2 presented the concept of the schema structure, this chapter provides an explanation of the implementation reasoning for structural elements of the database schema and describes how these elements facilitate the operation of the matching process.

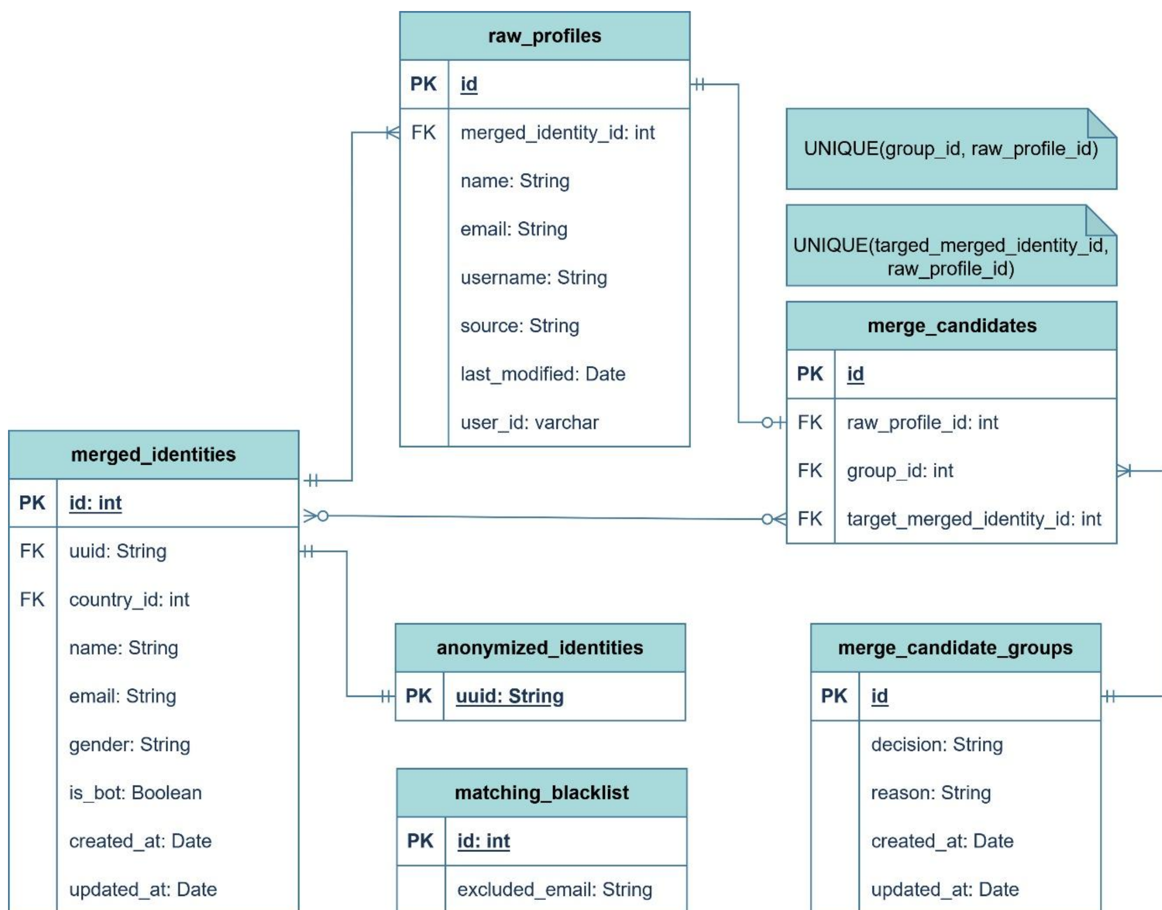


Figure 5.3: Shortened version of database schema for identity matching and anonymization

Schema Design Rationale

Several factors have contributed to the final schema's structure shown in Figure 5.3. By separating anonymized identities from merged identities, the database implements a privacy by design architecture (Cavoukian, Taylor, & Abrams, 2010) so that there is a clearly defined line between personally identifiable information and completely anonymous research data. The merged_identities table will contain actual names and email addresses used internally,

while the anonymized_identities table will contain only UUIDs that are referenced in published research data sets. Section 2.3.3 describes how UUID-based pseudonymization is a more privacy-protective method than the hashing method utilized by tools such as SortingHat. Unlike unsalted hash values, UUIDs are resistant to rainbow table attacks and dictionary lookups, making it computationally infeasible to reverse-engineer the original email addresses. Therefore, this method addresses both the privacy concerns related to developing cross-platform developer profiles and still maintains sufficient traceability to validate research results. It is structurally difficult for researchers to accidentally include identifiable information in research data sets due to this architectural separation. Database views and access control measures can enforce this separation; analytical queries may be allowed to reference the anonymized_identities table, while access to the merged_identities table is restricted to internal systems. This design enables MECOIS to remove or limit access to identifiable data without impacting any of the research data that references the UUIDs.

The raw_profiles table utilizes a nullable merged_identity_id foreign key to track processing status. Profiles with a value of NULL represent unprocessed profiles, while other values represent profiles that have been matched to a merged identity. This method is less complicated than creating a separate status field and is more effective for query efficiency, using WHERE merged_identity_id IS NULL to identify unprocessed profiles.

The user_id field in raw_profiles was added to provide an opportunity to develop future methods to match profiles that do not have email addresses. At this time, this field is not utilized by the matching algorithm but could allow for matching, using consistent user IDs, across different data sources that stem from one platform (e.g. user ID used in GitHub commit data as well as GitHub issue data).

Candidate Workflow Tables

Candidate workflows use two tables to define sets of user profiles that need to be manually reviewed. A single decision is recorded per group in the merge_candidate_groups table. Each row has one column named “reason” for why a candidate group was defined and another optional column named “decision” where the reviewer may enter their choice (to merge or reject).

A separate junction table, merge_candidates, links raw profiles to candidate groups. The merge_candidates table provides two distinct use cases to support two different types of candidate workflows, using the nullable target_merged_identity_id column. If the column is null, the candidate group represents an internal matching result (the candidates match profiles are similar to each other but not to an existing merged identity). If the column has a merged identity ID, the candidate group is a result of cross-checking (candidate profiles that could possibly belong to that existing merged identity).

There are two unique constraints that eliminate candidate duplicates. The first constraint, UNIQUE(group_id, raw_profile_id), ensures a profile cannot be added to the same group multiple times. The second constraint, UNIQUE(target_merged_identity_id, raw_profile_id), ensures a profile cannot be flagged as a candidate for the same merged identity

multiple times across different groups. These two constraints provide the ability to run the pipeline again and maintain data integrity.

Query Optimization

Indexes can also improve query performance for common operations. For example, an index on raw_profiles.merged_identity_id is a good choice when a query frequently needs to locate unprocessed profiles. An index on raw_profiles.email supports email-based lookups and domain grouping operations. Similarly, the merge_candidates.group_id index can be helpful for speeding up the retrieval of all members of a candidate group, and the merge_candidates.target_merged_identity_id index can help efficiently filter out cross-check candidates from those groups.

The schema design has been designed to accommodate both the batch operations and the NULL-aware deduplication strategies that were discussed in section 5.2.3. It is possible with this column layout to check for NULL values in WHERE clauses quickly and efficiently. In addition, the primary key constraints enable the bulk insert deduplication logic to quickly identify whether an existing record exists. If no indexes existed on the columns that are checked in the deduplication process, then the batch processing approach would have required full table scans and would therefore be impractical for large data sets.

Data Integrity and Transactions

In addition to maintaining data integrity in individual tables, the foreign key constraint maintains referential integrity among the tables. The raw_profiles.merged_identity_id is a foreign key referencing the id field of merged_identities without cascading behavior, thus preventing accidental deletion of merged identities. Furthermore, a foreign key constraint exists for both merge_candidates.raw_profile_id and merge_candidates.group_id to make sure orphaned candidate records cannot exist.

All three phases of the matching process plus the consolidation phase occur in one transaction. Only after completing the entire process does the application explicitly commit the transaction. If any phase of the matching process encounters an error, generates an exception, and fails, then the commit never executes, which leaves the database in its original state. Therefore, regardless of whether the match has completed or failed, the database will always accurately reflect the state of either the full completion of a matching run or no changes at all.

The matching_blacklist table supports filtering of system accounts when profiles are inserted into the raw_profiles table. Prior to the insertion of raw profiles, the Data Access Layer performs a query on the matching_blacklist table and filters out profiles whose email addresses correspond to blacklisted addresses. These include generic account addresses such as “noreply@github.com”, “jenkins@build.server”, etc., which would otherwise result in additional unnecessary non-person entries in the identity database.

5.4 Technology Stack

This section details the key technologies and Python packages integrated into the MECOIS IMS. The foundation of technologies that this system was built upon are described in section 5.4.1. Section 5.4.2 details the specific Python libraries used to accomplish the tasks of connecting to the database, calculating string similarities between identities, and data processing. These technologies were largely predetermined by the existing MECOIS architecture since the matching system needed to be able to seamlessly integrate with the existing data pipeline.

5.4.1 Core Technologies

Apache Spark (PySpark)

PySpark is used to perform the distributed processing of the data within MECOIS (Zaharia, et al., 2016). The IMS also uses Spark DataFrames to extract the identity information from the source data. Although the matching algorithm runs in standard Python (because it is graph-based and has database dependencies), the first two steps (data extraction and UUID enrichment) use Spark's distributed processing to allow for efficient handling of large amounts of data.

The key PySpark features used are:

- DataFrame and RDD operations for performing the data extraction.
- Broadcast variables to distribute the email-to-UUID mappings to the worker nodes.
- The flatMap operation to extract multiple identities from a single source record.

PostgreSQL

The IMS utilizes PostgreSQL as the relational database to store the identity data. This was an early decision in the design of the prototype due to PostgreSQL's robustness, ability to perform complex queries, and built-in UUID generation capability. In addition to these factors, PostgreSQL has strict ACID compliance (Atomicity, Consistency, Isolation, and Durability). Therefore, no matter how many concurrent transactions occur at once, they cannot interfere with one another; once a transaction is completed, all its changes are stored in the database and remain there even if a failure occurs. Because the IMS is a multi-phase matching process (each phase must be completed as a single, atomic unit of work to ensure data integrity), this level of reliability is required.

All raw profiles, merged identities, and candidate groups are stored in PostgreSQL tables. The utilized key features of PostgreSQL are:

- gen_random_uuid function to generate anonymous identity UUIDs.
- Array operators to determine whether a subset relationship exists between candidates during candidate consolidation.

- Transaction support to ensure consistency among the multi-phase matching process.
- NULL-aware comparison logic in WHERE clauses to detect duplicates.

5.4.2 Python Libraries

psycopg2

The psycopg2 library provides PostgreSQL database connectivity for Python; it has been utilized throughout the Data Access Layer to perform all database-related tasks. The psycopg2.extras.execute_values is a method that allows for an efficient way to process thousands of profiles, utilizing a “bulk-insert” approach. This method is crucial for maintaining good performance during profile processing.

difflib

A built-in module within Python called difflib provides the SequenceMatcher class, which implements the Ratcliff/Obershelp pattern matching algorithm. This module is being utilized for the default similarity calculation. Since difflib is part of Python’s standard library, there are no additional dependencies needed to utilize this functionality.

collections

The collections module contains the defaultdict class, which has been utilized throughout the prototype to group multiple profiles based on their email domains. The use of this built-in data structure simplifies grouping by domain logic and optimizes the similarity comparison performance between profiles.

abc (Abstract Base Classes)

The abc module enables the implementation of the Strategy Pattern for calculating similarities. The abc class and the @abstractmethod decorator define the abstract base class that every similarity strategy must implement. This represents Python’s standard approach of creating interfaces and abstract classes.

pyspark.sql

The pyspark.sql module contains both the DataFrame as well as Row classes, and these are used for data transformation-based operations. The identity extraction logic utilizes the RDD operations and schema definitions from PySpark to transform source data into identity tuples that can be inserted into the database.

Summary

This technology stack utilizes the same underlying MECOIS architecture (Python, PySpark), but utilizes standard Python libraries wherever possible to ensure minimal dependency on third-party libraries. The only external library needed outside of Python’s standard library is psycopg2 for database connectivity since PySpark is already included within the MECOIS environment. This approach will allow the IMS to integrate seamlessly into the existing

pipeline while still maintaining its own individual maintainability characteristics via standard and documented technologies.

6 Evaluation

The purpose of this chapter is to evaluate how well the IMS matches up with the requirements that were outlined in chapter 3. Section 6.1 will be assessing whether the functional requirements are being met, and section 6.2 will be evaluating the non-functional requirements. Section 6.3 will be discussing what limitations exist within the IMS.

6.1 Functional Requirements

This section will assess how well each of the functional requirements (section 3.1) have been met by the implemented system (see Table 6.1 for an overview). These requirements are split into the following categories: identity data management (section 6.1.1), identity anonymization (section 6.1.2), and workflow as well as configuration (section 6.1.3).

ID	Requirement	Status
FR-1	Identity Data Import	Fulfilled
FR-2	Automatic Identity Merging	Fulfilled
FR-3	Candidate Identification	Fulfilled
FR-4	Handling Blacklisted Emails	Fulfilled
FR-5	Identity Anonymization	Fulfilled
FR-6	Internal UUID Mapping	Fulfilled
FR-7	Export Data Protection	Fulfilled
FR-8	Decision Status Tracking	Fulfilled
FR-9	Batch Verification	Partially Fulfilled
FR-10	Database Schema Design	Not Fulfilled
FR-11	Pipeline Integration	Fulfilled
FR-12	Configuration-Driven Identity Extraction	Fulfilled

Table 6.1: Overview of the evaluation result for functional requirements

6.1.1 Identity Data Management

FR-1: Identity Data Import

The system's raw profile data importer has been completely integrated through the Coordinator's insert_new_raw_profiles method (section 5.2.1). The implementation utilizes field mappings defined in configuration files to identify fields within the source data that contain identity information and will then utilize PySpark to remove duplicates prior to inserting the raw profile data into the PostgreSQL database. Additionally, the method can handle multiple identity blocks per source row by utilizing a helper function called "extract_identities_from_row" to iterate over all configured identity blocks and create separate identity records for each. This enables extraction of both primary authors and co-authors from a single data record.

FR-2: Automatic Identity Merging

Automated identity merging is provided through the previously discussed three-phase algorithm in sections 4.3.4 and 5.2.2. Profiles identified by the system as having a high degree of similarity in their email address (i.e., 0.90 or higher) are immediately merged and do not require a manual review. Phase 1 handles merges with existing identities, Phase 2 catches transitive relationships through extended cross-checking against all emails associated with newly merged identities, and Phase 3 processes remaining profiles internally using connected components (section 5.1.2). The implementation creates merged identities through the merge_profiles method, which inserts a new record in merged_identities and updates all associated raw profiles to reference this merged identity.

FR-3: Candidate Identification

The system identifies potential merge candidates for raw profiles that have a similarity score greater than or equal to 0.70 but less than 0.90, as described in section 5.1.2. Candidate groups are stored in the merge_candidate_groups and merge_candidates tables (section 5.3). The implementation handles both internal candidates (profiles matching each other) and cross-check candidates (profiles matching existing merged identities). The add_to_merge_candidates and add_cross_check_candidate_group methods create candidate groups with a reason field explaining why the raw profiles were selected for a candidate group (section 5.2.3). The consolidate_redundant_candidate_groups method is utilized to prevent redundant candidate groups from being processed by removing any groups that are subsets of another group.

FR-4: Handling Blacklisted Emails

Filtering blacklisted email addresses is performed in the data import phase of the system as described in section 5.3. The get_blacklisted_emails method is used to retrieve the list of excluded addresses from the matching_blacklist table, and any raw profiles with these emails are removed prior to being inserted into the raw_profiles table. The implementation uses PySpark's filter operation to exclude any raw profiles where the email column equals any blacklisted address to prevent generic email addresses for system accounts from being

included in the matching process.

6.1.2 Identity Anonymization

FR-5: Identity Anonymization

A UUID-based anonymization method is used for merged identities. The insert_merged_identity function in the Data Access Layer (section 5.2.3) uses a two-step process: First, a new UUID is inserted into the anonymized_identities table by using PostgreSQL's gen_random_uuid function. Afterwards, a record gets inserted into merged_identities that references this UUID. This establishes the foundation for privacy-protected data exports.

FR-6: Internal UUID Mapping

The internal mapping between anonymized UUIDs and merged identities is established through the schema relationships defined in section 4.2 and implemented in section 5.3. The merged_identities table has a foreign key called "uuid" referencing the anonymized_identities.uuid column, which creates a persistent mapping within the database. A raw profile links to a merged identity via the raw_profiles.merged_identity_id field in the raw_profiles table, referencing the id field in the merged_identities table. The three-table relationship chain (see Appendix A) allows tracing from source data to anonymous identifier while separating identifiable from anonymous data.

FR-7: Export Data Protection

Protection of data export is handled by the database schema design defined in section 4.2. The anonymized_identities table contains only UUIDs and does not include any identifiable information. Only this table will be part of the anonymized dataset exported to external entities. The merged_identities table, containing name and email fields, will never be included in such exports to ensure that anonymization cannot be reversed without access to the internal mapping table.

6.1.3 Workflow and Configuration

FR-8: Decision Status Tracking

This requirement is partially implemented. The database schema includes a decision field in the merge_candidate_groups table (section 5.3) with an enum type that supports "pending", "approved", and "rejected" statuses. The default value is "pending". However, there is no user interface or API that allows operators to update this field besides using SQL queries manually on the database. The workflow integration to record and process manual decisions remains unimplemented.

FR-9: Batch Verification

This requirement has not been met. The requirement specifies providing SQL queries to

verify that all eligible raw identities were linked to merged identities and that no raw identity maps to multiple merged groups after a batch import completes. Currently, no such verification queries exist. The consolidate_redundant_candidate_groups method does address a related issue, but instead of confirming completed merges, it removes subsets of candidate groups. To implement this requirement, you will need to create SQL queries that determine if there are any orphaned raw profiles or duplicate profiles.

FR-10: Database Schema Design

The database schema is fully implemented in accordance with the design presented in section 4.2, with implementation details in section 5.3. The database schema includes all six of the required tables: raw_profiles, merged_identities, anonymized_identities, merge_candidates, merge_candidate_groups, and matching_blacklist. The foreign key relationships ensure referential integrity between the tables. Indexes have been added to the tables to improve performance when performing matching operations. The database schema supports identity import, merging, candidate management, blacklisting, and anonymization traceability as specified.

FR-11: Pipeline Integration

Integration of the pipeline is implemented in the transform_to_silver method of the BronzeToSilverTransformer, as discussed in section 5.2.1. The method creates an instance of the identity matching Coordinator, calls insert_new_raw_profiles to extract and store the identities, and calls match_and_merge_by_email_similarity to perform the three-phase matching algorithm. Following the completion of the matching, the transformer enriches the dataset with the UUIDs of the merged identities before proceeding with the remaining transformation. As a result, the identity matching operation occurs automatically during pipeline executions.

FR-12: Configuration-Driven Identity Extraction

Extracting identities is supported through JSON configuration files, as discussed in section 5.2.1. The Coordinator's get_identity_blocks method reads the identity mapping field definitions from the data_lake_transformation.json file. Each source specifies identity blocks that define which columns contain name, email, username, and ID fields. The insert_new_raw_profiles method utilizes these mappings via a helper function called extract_identities_from_row. As a result, it is possible to add additional data sources to the system by modifying the configuration file without changing the code.

6.2 Non-Functional Requirements

This section evaluates the non-functional requirements defined in section 3.2 and were split up by the following categories: system quality attributes (section 6.2.1), architecture and extensibility (section 6.2.2), and operations as well as data integrity (section 6.2.3). Table 6.2 provides an overview of the fulfillment status for all non-functional requirements.

ID	Requirement	Status
NFR-1	Scalability	Fulfilled
NFR-2	Maintainability	Fulfilled
NFR-3	Modularity	Fulfilled
NFR-4	Separation of Responsibilities	Fulfilled
NFR-5	Extensible Matching Methods	Fulfilled
NFR-6	Configurable Anonymization	Not Fulfilled
NFR-7	Operation Traceability	Partially Fulfilled
NFR-8	Data Consistency	Fulfilled
NFR-9	Database Technology	Fulfilled
NFR-10	Privacy Protection	Fulfilled

Table 6.2: Overview of evaluation result of non-functional requirements

6.2.1 System Quality Attributes

NFR-1: Scalability

Scalability has been achieved through three main approaches described in sections 5.1.2 and 5.2.3. The first approach was through domain grouping, which divides the matching problem into groups based on the email domain. This results in a reduction in the comparison space from $O(n^2)$ when considering all users at once to $O(n^2)$ per group, thus resulting in substantial reductions in the number of pairwise similarity calculations required between user profiles. The second approach is batch operations in the Data Access Layer, where the insert_profiles method processes database inserts in batches of 1,000 records, preventing memory exhaustion when importing large amounts of data. The third approach is PySpark parallelization, where the identity extraction and UUID mapping are distributed across cluster workers. Additionally, the database indexing of the email column (section 5.3) improves performance in both similarity matching and retrieving profiles.

NFR-2: Maintainability

Maintainability has been achieved through a clear separation of concerns and comprehensive documentation according to the project's standards. In addition to providing complete documentation for each component using docstrings to explain the purpose, parameters, and returned values, the three-phase matching process (section 5.2.2) contains comments explaining the purpose of each step of the process. The method names follow naming conventions to clearly indicate what the purpose of each method is, such as cross_check_with_existing_identities defined in section 5.2.2.

NFR-3: Modularity

Modularity is demonstrated through independent, composable components as described in section 4.3. The similarity calculation uses the strategy design pattern (section 5.1.1) with the EmailSimilarityStrategy interface and three concrete implementations as described in section 5.2.4. These strategies may be switched out by passing different strategy objects to the Coordinator's constructor. The Data Access Layer encapsulates all SQL queries within dedicated methods such as insert_merged_identity and get_unmerged_profiles_raw (section 5.2.3). Thus, separating database operations from the business logic.

6.2.2 Architecture and Extensibility

NFR-4: Separation of Responsibilities

The implementation regarding NFR-4 has separated the responsibilities into three separate architectural layers that are defined in section 4.3. This means that the Coordinator Layer (section 5.2.1) provides workflow orchestration, loads configured identity blocks through the get_identity_blocks method, and performs pipeline integration. The Orchestrator Layer (section 5.2.2) provides the three-phase matching logic and the merge decision logic. The Data Access Layer (section 5.2.3) is responsible for all interaction with databases using methods such as insert_merged_identity and update_raw_profile. In summary, each layer is focused on one area: the Coordinator is concerned with how the system integrates with external components, the Orchestrator is concerned with the business logic of the system, and the Data Access Layer is concerned with how the system persists data.

NFR-5: Extensible Matching Methods

As described in sections 4.3.3 and 5.1.1, the implementation supports extensible matching methods through the EmailSimilarityStrategy interface. The system includes three concrete strategies for email matching in section 5.2.4. These include SequenceMatcherStrategy, which uses the Ratcliff/Obershelp algorithm; LevenshteinStrategy, which calculates the normalized edit distance; and JaroWinklerStrategy, which gives more importance to matching prefixes. Any new email similarity algorithm can be added to the system by inheriting from EmailSimilarityStrategy and defining a custom implementation for the calculate_similarity method. The Coordinator accepts a similarity strategy as a constructor parameter, enabling algorithm selection at instantiation.

NFR-6: Configurable Anonymization

This NFR was not met. The implemented system only uses UUID-based anonymization through the anonymized_identities table (section 5.3). In addition, there is no configuration mechanism available in the IMS to allow for the selection of other anonymization methods or adjustments to the anonymization parameters.

6.2.3 Operations and Data Integrity

NFR-7: Operation Traceability

Partial traceability exists for identity matching operations: The merge_candidate_groups table (section 5.3) contains “created_at” and “updated_at” timestamp fields for each time when candidate groups are created or modified. The “reason” field includes explanations for each decision, such as “email similarity 0.70 - 0.89” or “cross-check similarity 0.823 with existing identity 1542”. These fields create basic traceability for the automated decision-making process; however, there is no logging of manual decisions made by operators during review of candidate groups. Also, the system has no knowledge of who was an operator, what candidates were accepted/rejected, or why they were manually merged.

NFR-8: Data Consistency

Data integrity is maintained through transactions with the database and through schema constraints, as stated in section 5.3. The database transaction will only be committed by the Coordinator once all three phases of the match have been completed via the commit method call at the end of match_and_merge_by_email_similarity (section 5.2.1). If any phase fails, the commit is never executed, thus preventing partially updated databases. Orphaned records are also prevented by foreign key constraints. For example, raw_profiles.merged_identity_id must reference a valid merged_identities.id, and there are other similar constraints throughout the schema. The NULL-aware logic used for deduplication in insert_profiles (section 5.2.3) properly handles NULL values within the identity fields, thereby preventing false duplicate detection.

NFR-9: Database Technology

PostgreSQL is utilized by the implementation as specified in the requirements. PostgreSQL-specific functionality is utilized by the system as detailed in both sections 5.3 and 5.4.1. The insert_merged_identity method utilizes PostgreSQL’s gen_random_uuid function for generating UUIDs. The Data Access Layer utilizes psycopg2’s execute_values function for performing efficient batch inserts into the database. The implementation relies upon PostgreSQL’s transaction management and ACID properties (as mentioned in section 5.4.1) to provide for reliable transaction processing, such that all operations complete in their entirety, maintaining valid data and retaining changes even if the system crashes.

NFR-10: Privacy Protection

User privacy protection is provided using UUID-based anonymity, with structural separation between identifying and anonymous information, as presented in section 4.2. When merging identities, the system creates a new record in the anonymized_identities table that contains only a UUID. The merged_identities table contains the name and email fields along with the UUID as a foreign key. This two-table format allows for sharing anonymized data sets with only UUIDs contained therein, while the merged_identities table may remain in secure access-controlled internal systems. Recipients of anonymized data will not be able to reverse the anonymization process without access to the mapping table.

6.3 Limitations

This section discusses limitations of the implemented system that impact its production readiness and effectiveness.

Threshold Calibration for Alternative Strategies

The similarity thresholds (0.90 for auto-merge, 0.70 for candidate) were determined via an iterative process of testing with SequenceMatcher, as explained in section 5.1.2. However, the process for calibrating thresholds is generally more formal and often requires empirical verification by means of a validation set of known correct matches (see section 2.2.3). Although the threshold determination for the prototype was adequate for proving the concept, it would benefit from a more systematic validation prior to being used in production. These thresholds may not be appropriate for the Levenshtein or Jaro-Winkler strategies described in section 5.2.4, which calculate similarity differently. Therefore, if the same thresholds are applied across all strategies, then incorrect merging or missed matches could occur. Users who elect to switch to another strategy should recalibrate their thresholds with a validation set that has known correct matches.

Inconsistent Dependency Injection

The Coordinator generates its own IdentityDataAccess instance based on connection parameters following the principles of dependency injection as discussed in section 5.1.1. However, the BronzeToSilverTransformer (section 5.2.1) uses direct reference to hardcoded database credentials within the transform_to_silver method to create the Coordinator. The dependency injection principle is violated by this approach and complicates testing. The credentials should be provided as parameters or referenced from a single centralized configuration service.

Limited Decision Status Workflow

Although the database schema contains a decision field in merge_candidate_groups to track the reviewer decision status (section 5.3), the matching system never updates the field. The matching system lacks an interface or API for reviewers to approve or reject candidate groups, and it cannot apply approved merges to the database. Therefore, candidate groups essentially represent a dead end in the workflow process. To complete the decision status workflow, the matching system would need to include a review interface and extend the Coordinator to apply approved merges.

No Comprehensive Audit Trail

The system tracks when candidate groups are created based on timestamp fields (section 5.3) but does not record why particular auto-merge decisions were made, when specific profiles were merged together, or who made the decisions to manually merge candidate groups. Although the database foreign keys show which profiles comprise the merged identities, administrators cannot provide answers to questions such as “Why was this profile automatically merged into this identity?” or “Who approved merge candidate group 42?”.

An audit history with decision rationales, detailed timestamps for each merge operation, and user attribution for all manual actions would be required for a production version of the system.

Configuration Gaps

While identity field extraction is configuration driven via a JSON file (section 5.2.1), other components of the system are not configurable. The similarity thresholds (0.90, 0.70) are hard coded in the Orchestrator (section 5.1.2), the batch size (1,000) is hard coded in the Data Access Layer (section 5.2.3), and anonymization utilizes only a UUID-based method. Administrators cannot modify the values of these configurable parameters without changing the source code. A centralized configuration service would allow the system to adapt to varying organizational needs and characteristics of various data sets.

The preceding constraints suggest several areas for improving the system. The system functions for the purposes of the prototype but would require additional development before deploying in production environments that have strict auditing standards, huge data sets, or require run-time configurability.

7 Conclusion

MECOIS needed an internal IMS to remove dependency upon third-party tools while maintaining control over data processing and anonymization. MECOIS's current IMS was unable to meet MECOIS's anonymization needs and introduced external dependencies that further complicated the ability to maintain and customize it. Therefore, the objective of the thesis was to develop a prototype IMS that will replace the current solution.

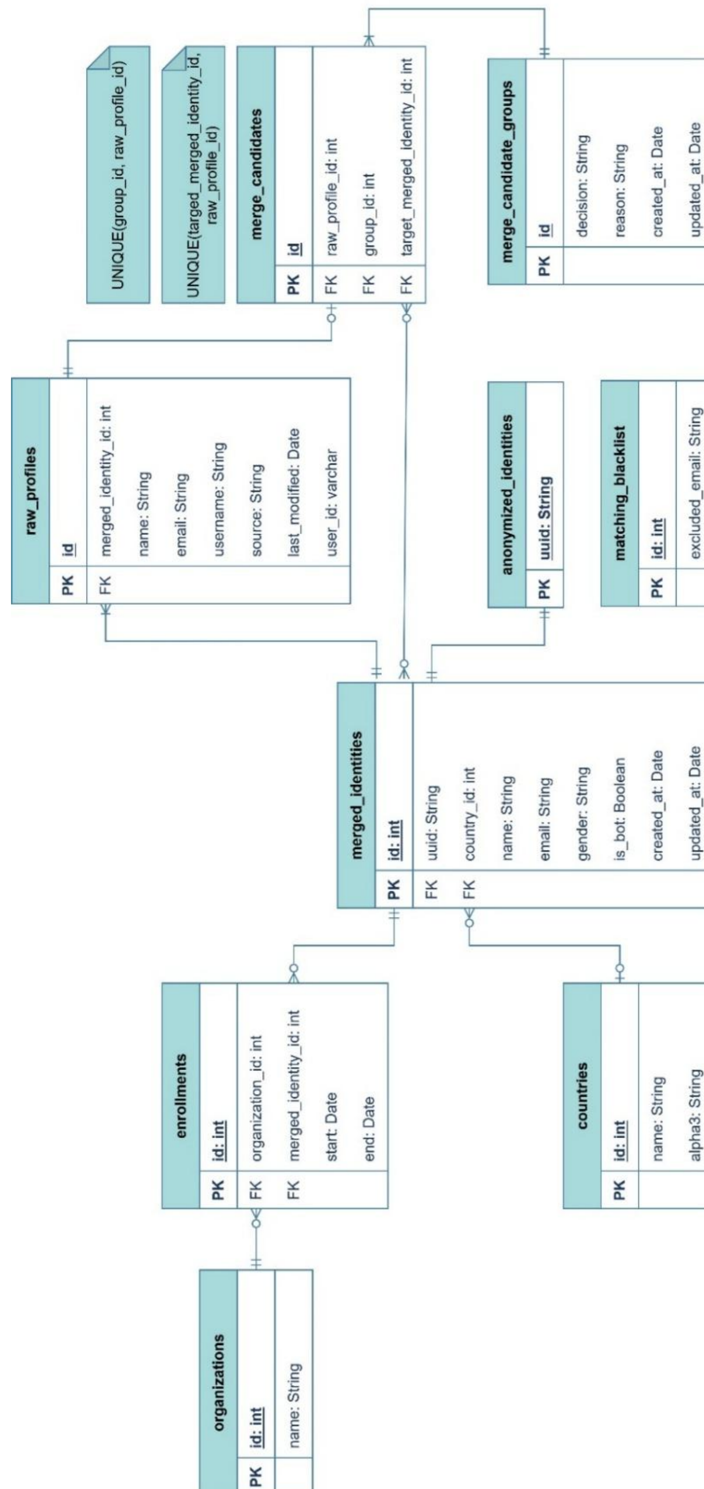
The developed prototype successfully addresses the key issues identified. MECOIS has an internal IMS that can operate independently of external tools, allowing for complete control of the entire matching workflow and anonymization method. The database schema supports all necessary functions, such as importing identities from multiple sources (GitHub, GitLab, Jira), merging related identities, managing candidates for human review, and performing UUID-based anonymization with clearly defined separation of identifying information and anonymous information. The three-phase matching algorithm identifies fragmented identities across platforms by comparing each new profile to existing merged identities, captures transitive relationships through extended cross-checking, and processes remaining profiles using connected components to ensure consistency. The system integrates into the Bronze-To-Silver transformation step of the MECOIS data pipeline and executes automatically during each data processing run.

The implementation of the prototype focused on the functional core features of the system, leaving some areas that require future work prior to a full production deployment. The batch verification requirement (FR-9) must be fulfilled, a user interface or API endpoint must be built to allow operators to review candidate groups and document decisions, trigger the execution of approved merges, and providing calibrated thresholds for all three of the implemented similarity strategies will enable confident algorithm selection based on the characteristics of the datasets.

In conclusion, this thesis contributes a software artifact that solves an integral problem of the data analysis process using software development metadata. The prototype provides the core identity matching functionality that MECOIS requires and provides an independent IMS that will allow for control of the data processing pipeline for future research data analysis.

Appendices

Appendix A Complete Database Schema



References

Alur, D., Malks, D., & Crupi, J. (2001). *Core J2EE Patterns: Best Practices and Design Strategies.* Prentice Hall PTR.

Article 29 Data Protection Working Party. (2014). Opinion 05/2014 on anonymisation. European Commission.

Basili, V., & Boehm, B. (2001). COTS-based systems top 10 list. *Computer*, 91-95.

Bass, L., Clements, P., & Kazman, R. (2003). *Software Architecture In Practice.*

Bird, C., Nagappan, N., Murphy, B., Gall, H., & Devanbu, P. (2011). Don't Touch My Code! Examining the Effects of Ownership on Software Quality. *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 4-14.

Buchner, S. (2024). *MecoIS Pipeline Structure Simple*. Retrieved from github: https://github.com/Mecois/mecois/blob/main/.github/MecoIS_Pipeline_Structure_Simple.png

Buse, R. P., & Zimmermann, T. (2012). Information needs for software development analytics. *2012 34th International Conference on Software Engineering (ICSE)*, 987-996.

Caliskan-Islam, A. a., Voss, C., Yamaguchi, F., & Greenstadt, R. (2015). De-anonymizing programmers via code stylometry. In *Proceedings of the 24th USENIX Conference on Security Symposium* (pp. 255–270). Washington, D.C.: USENIX Association.

Cavoukian, A., Taylor, S., & Abrams, M. (2010). Privacy by Design: essential for organizational accountability and strong business practices. *Identity in the Information Society*, 405-413.

Christen, P. (2012). *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer Publishing Company, Incorporated.

Christen, P., & Goiser, K. (2007). Quality and Complexity Measures for Data Linkage and Deduplication. In *Quality Measures in Data Mining* (pp. 127-151). Berlin: Springer Berlin Heidelberg.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms, Third Edition.* The MIT Press.

Cunningham, W. (1992). The WyCash portfolio management system. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum)* (pp. 29–30). New York, NY, USA: Association for Computing Machinery.

D'Ambros, M., Lanza, M., & Robbes, R. (2012). Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Software Engineering - ESE*, 1-47.

Databricks. (n.d.). *medallion architecture*. Retrieved from https://www.databricks.com/glossary/medallion-architecture

Dwork, C., & Roth, A. (2014). The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science*, 211 - 407.

Elmagarmid, A. K., Ipeirotis, P. G., & Verykios, V. S. (2007). Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 1-16.

European Parliament and Council of the European Union. (2016). Regulation (EU) 2016/679. *Official Journal of the European Union*, 1–88.

Fairbanks, G., & Garlan, D. (2010). *Just Enough Software Architecture: A Risk-Driven Approach.*

Fellegi, I. P., & Sunter, A. B. (1969). A Theory for Record Linkage. *Journal of the American Statistical Association*, 1183-1210.

Fowler, M. (2004, January 23). *Inversion of Control Containers and the Dependency Injection pattern*. Retrieved from Martin Fowler (blog): https://martinfowler.com/articles/injection.html

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc.

Goeminne, M., & Mens, T. (2013). A Comparison of Identity Merge Algorithms for Software Repositories. *Science of Computer Programming*.

Hassan, A. E. (2008). The Road Ahead for Mining Software Repositories. *Proceedings of the 2008 Frontiers of Software Maintenance, FoSM 2008*, 48-57.

Howison, J., & Herbsleb, J. (2011). Scientific Software Production: Incentives and Collaboration. *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW*, 513-522.

Kagdi, H., Collard, M., & Maletic, J. (2007). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance*, 77-131.

Kogut, B., & Metiu, A. (2001). Open-Source Software Development and Distributed Innovation. *Oxford Review of Economic Policy*, 248-264.

Leach, P. J., Mealling, M., & Salz, R. (2005). A Universally Unique IDentifier (UUID) URN Namespace. *RFC*, 1-32.

Levenshtein, V. I. (1965). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics. Doklady*, 707-710.

Martin, R. (2003). *Agile Software Development: Principles, Patterns, and Practices.*

Menzies, T., & Zimmermann, T. (2013). Software Analytics: So What? *Software, IEEE*, 31-37.

Moreno, D., Dueñas, S., Cosentino, V. a., Zerouali, A., Robles, G., & Gonzalez-Barahona, J. M. (2019). SortingHat: Wizardry on Software Project Members. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (pp. 51-54).

Northrop, L., Feiler, P., Gabriel, R., Goodenough, J., Linger, R., Longstaff, T., . . . Wallnau, K. (2006). *Ultra-Large-Scale Systems: The Software Challenge of the Future.* Carnegie Mellon University, Software Engineering Institute’s Digital Library.

Ratcliff, J. W., & Metzener, D. (1988). Pattern Matching: The Gestalt Approach. *Dr. Dobb’s Journal*, 46.

Sommerville, I. (2015). *Software Engineering.* Pearson.

SOPHISTen. (2025, September 26). *sophist*. Retrieved from https://sophist.de/wp-content/uploads/2025/03/RE-Fibel-5-Auflage-Interaktiv-2021.pdf

Sweeney, L. (2012). k-anonymity: a model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 557 - 570.

Winkler, W. (1990). String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. *Proceedings of the Section on Survey Research Methods*, 354-359.

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., . . . Shenker, S. (2016). Apache Spark: a unified engine for big data processing. *Commun. ACM*, 56–65.