Estimating Software Contribution Time from Git and GitHub Metadata

MASTER THESIS

Mathieu Stenzel

Submitted on 30 January 2026

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

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

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, 30 January 2026

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, 30 January 2026

Abstract

The problem of estimating the costs of a software development project has been a persistent challenge in the field. This is particularly relevant in projects developed by multiple contributors within companies (Inner Source), where costs must be allocated across multiple cost centres for management and for tax and transfer-pricing documentation. Simple proxies, such as the number of commits or lines of code alone, are insufficient to capture the actual effort involved. This thesis integrates a previously evaluated algorithm that uses Git repository metadata to estimate the time invested per commit into the MECOIS research project and extends it with the following: disassembly of squashed commits; filtering of no-effort commits and statistical outliers; a regression from changed lines of code to contribution time; support for co-authors; and time zone normalization. The extensions are evaluated against the base version using distributional statistics and an internal plausibility check, which identifies and caps implausibly large contribution times assigned to commits. The improved version reduces extreme estimates substantially: the median contribution time and the standard derivation per commit drop significantly for both of the two tested public repositories. The plausibility-cap events are reduced by more than 24 %. These improvements yield more robust and reliable allocations for the time invested per commit, strengthening the evidential basis for cost calculation and transfer pricing.

Contents

List of Figures

List of Tables

Acronyms

BAlgo	Base Algorithm
CT	Contribution Time
ETL	Extract-Transform-Load
FAU	Friedrich-Alexander University of Erlangen-Nuremberg
GDPR	General Data Protection Regulation
IAlgo	Improved Algorithm
IS	Inner Source
LOC	Lines of Code
MAPE	Mean Absolute Percentage Error
MECOIS	Metrics for Engineering Communities in Inner Source
MSR	Mining Software Repositories
PII	Personally Identifiable Information
PR	Pull Request
SHA	Secure Hash Algorithm

1 Introduction

Inner Source (IS) is the adoption of open source practices within organizational boundaries, along with the benefits that come with it (Dinkelacker et al., 2002). Recent research has shown that the use of IS in companies can increase software reusability, improve software quality, and even lead to greater job satisfaction (Edison et al., 2020; Pitkevics et al., 2025; Stol et al., 2024). As organizations expand their IS initiatives, questions about planning, managing, and evaluating these efforts become increasingly important. For an effective use of IS, companies seek reliable ways to calculate project costs, quantify contributions, and assess the value created through internal collaborative development (Buchner & Riehle, 2023). One approach discussed in this context is to estimate the Contribution Time (CT), which represents the time invested into the actual development of a software project. These time-based measures can then be used as basis for internal cost calculation and management reporting (Buchner & Riehle, 2022; Hirsch & Riehle, 2022).

However, this challenge is not unique to IS. Software engineering research has long emphasized that effort estimation is difficult and strongly dependent on context, available inputs, and assumptions (Jørgensen, 2014). Traditional metrics such as Lines of Code (LOC), function points, or high-level project plans and status reports can be useful in specific settings, but they often fail to represent the complex nature of modern software development work (Forsgren et al., 2021; Petersen, 2011; Symons, 2002). As development processes have shifted toward distributed collaboration and continuous integration, these limitations become even more visible because development work is fragmented and spread across many small activities rather than captured in a single planning artifact (Zhao et al., 2017).

The adoption of continuous integration is associated with changes in development practices and shorter feedback cycles, which supports more fine-grained analyses of development activity closer to where work is performed (Zhao et al., 2017). In GitHub-based workflows, a change is typically proposed via a Pull Request (PR) and reviewed before merging, enabling fine-grained analyses of development activity closer to where work is performed. In this context, the code repository

is a valuable resource: Git repositories record commits and metadata that can be used to approximate metrics such as development patterns. However, the data must be examined and considered with caution (Kalliamvakou et al., 2014; Zhao et al., 2017). Repository-based effort estimation has shown that activity traces can support approximations of CT when assumptions are made explicit and models are evaluated empirically (Amor et al., 2006; Robles et al., 2022). Importantly, overly simplistic metrics such as the number of LOC or commits per developer are widely considered weak proxies for effort (Afroz et al., 2025). Depending on working style and task type, one commit may represent minutes or days of work, and commit granularity differs substantially across projects and teams. Although approaches such as commit-level CT analysis show promise as effort or cost estimation proxies, their practical applicability, accuracy, and limitations remain insufficiently explored. Furthermore, it is unclear how these methods perform in real-world environments with globally distributed teams, and vastly different contribution practices, ranging from frequent, "small" commits to fewer, "larger" ones (regarding the value of LOC) and a variable amount of no-effort activity (noise) in the repository data.

This thesis investigates these challenges through two research questions:

> RQ1: *What repository-near methods to determine the CT in the development of software already exist, and what design or implementation challenges arise in real-world scenarios?*
>
> RQ2: *How can the estimation of the CT in the development of software be improved?*

The objective of this thesis is to develop and evaluate an algorithm that approximates the CT per contributor throughout the development process of a software project as closely as possible. The solution is intended to operate on real-world Git/GitHub repositories, accommodate distributed teams, and remain robust across diverse working styles. We integrate the proposed algorithm into the research project titled *Metrics for Engineering Communities in Inner Source (MECOIS)* at Friedrich-Alexander University of Erlangen-Nuremberg (FAU).

The remainder of this thesis is structured as follows. Chapter 2 reviews existing approaches for estimating the CT in software development and discusses real-world challenges and limitations. Based on these findings, requirements are derived and prioritized in chapter 3. The architecture of the MECOIS project, along with the architecture of the algorithm itself, is described in chapter 4. Chapter 5 presents the design decisions and the implementation of the proposed approach, and describes the extension of the first approach with multiple features. The two solutions are then evaluated against each other using distributional statistics and an internal plausibility check in chapter 6. Finally, chapter 7 summarizes the contributions and outlines future research directions.

2 Literature Review

This thesis focuses on a practical yet unresolved question in IS: How can the CT in the development of software be estimated when collaboration occurs across organizational boundaries, such as cost centers, and requires fair allocation? In corporate contexts, this way of working is desirable due to code reuse, faster delivery, shared ownership, among the most evident reasons, but they also raise the question about fair cost allocation across different teams. Literature offers several approaches, ranging from classical cost estimation to repository mining and IS-specific measurement. However, it does not converge on one universally accepted solution. The objective of this chapter is to delineate the solution space and to motivate the approach taken in this thesis.

2.1 Search Procedure

The literature search followed a process based on database queries and backward/forward snowballing, mainly conducted in ACM Digital Library[1], IEEE Xplore[2], Springer Nature Link[3] and ScienceDirect[4] with Google Scholar.

Searches were executed in three topics with the following keywords in different combinations:

- Mining Software Repositories: `"mining software repositories"`, `"msr"`, `"github metadata"`, `"git metadata"`, `"version control metadata"`
- Cost / Effort Estimation in Software Development: `"developer effort"`, `"time worked"`, `"effort estimation"`, `"cost estimation"`, `"ex-post estimation"`, `"repository effort"`, `"contribution time"`
- Inner Source: `"inner source"`, `"innersource"`, `"cost"`, `"accounting"`, `"adoption"`

[1] https://dl.acm.org
[2] https://ieeexplore.ieee.org
[3] https://link.springer.com
[4] https://www.sciencedirect.com

Papers were included if they either propose or evaluate methods to estimate effort or CT from software artifacts, or discuss IS measurement where effort attribution and estimation matter. For the most relevant papers in each cluster, references and citing works were analyzed to identify foundational and follow-up studies.

To provide an overview of the literature found, we used the tool Litmaps[5] to create a graph containing the included literature (bubbles) and their cross-citations (lines). The bubble size is related to the number of total citations of the referenced work. This graph is shown in figure 2.1.

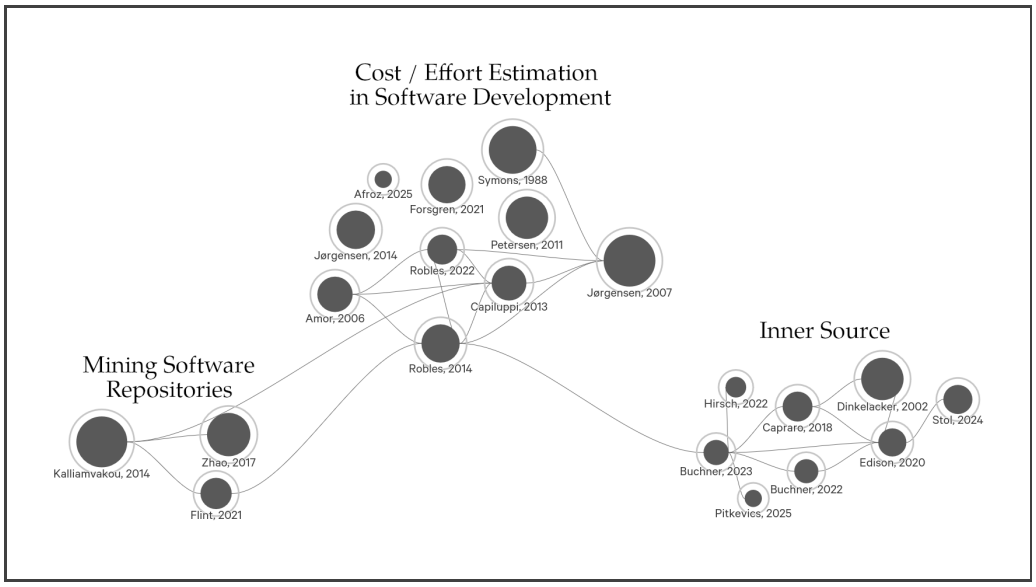

Figure 2.1: Literature graph

2.2 From Cost to Contribution Time Estimation

A logical starting point is classical software cost estimation research. A systematic review by Jorgensen and Shepperd (2007) demonstrates that the field of cost estimation is extensive, with numerous competing approaches, and that outcomes vary significantly based on context and data quality. This is an important aspect of the thesis because it highlights a general pattern: Although the target variable here is CT and not ”cost”, the estimate is only as useful as the assumptions behind it.

Many conventional estimation methodologies are primarily designed as *planning* instruments, i.e., they aim to forecast effort before or early during a project. For example, widely used parametric approaches such as COCOMO II explicitly

[5]https://www.litmaps.com/

target estimating effort and schedule in project planning and refine estimates as more information becomes available (Boehm CSSE, 2023).

In contrast, the objective of this thesis is aligned with *ex-post* (the use of recorded data) or continuous measurement. The idea is to estimate CT from trace data produced during the development process. This shift changes the technical problem: instead of selecting predictors such as requirements complexity or early size proxies, the challenge becomes identifying which repository signals can reasonably be interpreted as CT, and how to interpret the resulting values (Amor et al., 2006; Flint et al., 2021).

2.3 Mining Software Repositories

Mining Software Repositories (MSR) can provide a lot of metadata that can be used as measurement sources. However, the MSR literature is also clear that raw repository metadata should not be used as a direct measure of effort, without initial interpretation.

Kalliamvakou et al. (2014) describe why mining GitHub can lead researchers to draw the wrong conclusions if they do not take into account how repository events are created (e.g., forks, mirrors, incomplete histories, non-merged contributions, automation that creates non-human effort). For time-focused measurement this warning becomes even more critical: time-based approaches depend not only on "what" happened but also on "when" it happened and whether timestamps reflect real working sessions.

Flint et al. (2021) show that time-based Git analyses can be distorted by several recurring data-quality problems. They report *suspicious timestamps* (e.g., implausibly old dates), which often appear in repositories affected by migration tooling such as *git-svn*, and *out-of-order commits*, where a commit's timestamp is earlier than at least one of its parents, which can result from time zone inconsistencies and from workflows that rewrite history (e.g., rebasing, cherry-picking). As practical guidance, they recommend filtering, correcting or removing clearly invalid timestamps, handling out-of-order commits via filtering or reordering strategies, and preferring the author date over the committer date when the workflow includes history rewriting strategies.

Consequently, the analysis of these works leads to the conclusion that trace-based time estimation is feasible, on the condition that the method is explicit about its assumptions and robust against noisy or workflow-dependent signals.

2.4 Estimating Contribution Time from Repository Metadata

In the field of estimating CT, we identified relevant work for this thesis by examining literature focused on settings where recorded time tracking data is not available.

Robles et al. (2014) propose an ex-post effort estimation method, applied to the OpenStack project, that maps repository activity to effort in person-months. Concretely, they use two simple activity signals: the number of commits per developer and the number of active days (at least one commit) as proxies for development work. Their key design choice is to first identify likely full-time developers via an activity threshold, then assign them one person-month per month, while assigning non-full-time contributors only a fraction based on their activity relative to a full-time developer. To calibrate and validate the threshold, they combine repository mining with a developer survey. The paper also makes explicit the central limitation of trace-based effort estimation: repository data records "points in time" (commits), but not how long the work took, so the model depends on careful assumptions and calibration.

In their later publication, the authors generalize their effort estimation method to use it for multiple large open source projects. They refine the calibration logic by explicitly modeling how false positives and false negatives can compensate each other through the aggregation. They also concluded that the activity threshold varies by project and is influenced by development practices (e.g., stricter review and frequent squashing can make commits a coarser and more effort-heavy unit), so empirical testing or knowledge about contribution practices and project-specific adaptation is necessary for the model to remain credible (Robles et al., 2022).

Capiluppi and Izquierdo-Cortázar (2013) analyze the Linux kernel's Git history to characterize how development effort is distributed over time and what that implies for repository-based effort proxies. Using Git's locally recorded commit timestamps, they divide activity into three 8-hour time slots per day and study weekly and hourly patterns across the history of the project. In addition, they compare activity one week before and one week after major releases. Their results indicate that Linux development shows substantial activity outside typical office hours and that pre-release periods exhibit markedly different behavior than post-release periods. This leads to the conclusion, that too simple effort proxies can miss important characteristics for the estimation.

2.5 Estimating Contribution Time in Inner Source

Despite the growing field of IS research, actionable measurement tools for evaluation and guidance remain underdeveloped. Edison et al. (2020) identify a persistent lack of metrics and quantifiable evidence for assessing IS initiatives and explicitly frame the definition of such metrics as a key open research question. From a business perspective, Buchner and Riehle (2023) argue that economic assessment requires connecting organizational and financial context to engineering trace data. They describe commit data as a possible source for IS measurement, noting that no comprehensive, validated measurement toolset has emerged yet.

A concrete step toward quantifying collaboration is provided by Capraro et al. (2018), who propose the patch-flow method as a way to measure and quantify cross-boundary code contributions inside organizations. For economic assessment, measurement concepts must remain feasible in practice: Hirsch and Riehle (2022) derive management-accounting oriented models and underline that metrics should support controlling and quantification without obstructing collaboration.

In order to estimate CT at a fine granularity, Buchner and Riehle (2022) propose an approach to compute the ”time worked”, defined as the time elapsed while working on a single commit (which we call CT). This approach enables cost calculations based on development artifacts rather than coarse organizational aggregates.

This approach is relevant to the scope of this thesis, as it treats CT as a variable that can be inferred from commit sequences under explicit heuristics and assumptions. At the same time, it belongs to the broader category of repository-based estimation methods, where we previously concluded that repository metadata are a good starting point to obtain CT proxies (Amor et al., 2006; Robles et al., 2014, 2022).

2.6 Design and Implementation Challenges in Real-World Scenarios

The reviewed work indicates that repository-based effort estimation is sensitive to the quality and semantics of recorded repository events, project-specific workflows that shape commit granularity, and the extent to which the resulting measures remain interpretable to stakeholders.

Recurring challenges that must be addressed for real-world application include:

- **Time data quality and interpretation.** Timestamp anomalies, history rewriting, and migration artifacts can distort temporal analyses and therefore require explicit detection and handling strategies (Flint et al., 2021;

Kalliamvakou et al., 2014).

- **Workflow-dependent granularity.** Commit frequency and size depend on development and review practices; consequently, the same activity signal can correspond to different amounts of work across projects, which motivates calibration and project-specific adaptation (Capiluppi & Izquierdo-Cortázar, 2013; Robles et al., 2014, 2022).
- **Cross-boundary attribution.** IS scenarios require attributing CT across organizational and project boundaries. This presupposes operational definitions of cross-boundary contribution and rules for aggregating contributions in a way that remains consistent with the intended economic interpretation (Buchner & Riehle, 2022; Capraro et al., 2018).
- **Explainability and acceptance.** For governance and controlling, CT estimations must be transparent and understandable. IS research emphasize lightweight measurement approaches that do not obstruct collaboration and can be justified to stakeholders (Buchner & Riehle, 2022; Edison et al., 2020; Hirsch & Riehle, 2022).

Taken together, these challenges narrow the solution space: methods need to be explicit about their assumptions, robust against noisy repository data, and practical to apply without introducing additional reporting overhead (Flint et al., 2021; Robles et al., 2014). In this context, the algorithm by Buchner and Riehle (2022) provides a suitable baseline as it derives CT distributions from commit sequences using data that is already present in standard Git repositories, and it is explicitly motivated and evaluated for cost-related use cases in IS settings.

The next steps of this thesis consider Buchner and Riehle (2022) as the Base Algorithm (BAlgo) for the CT estimation, and then address the aforementioned challenges through requirements-driven adaptations to obtain the Improved Algorithm (IAlgo). The next chapter therefore derives the concrete requirements and integration constraints for this implementation work.

3 Requirements

After reviewing the relevant literature, we translate the identified challenges of repository-based CT estimation into concrete requirements for the software artifact developed in this work. This artifact integrates into the MECOIS project, which is described in chapter 4. The requirements include the handling of data noise, workflow diversity, limited interpretability of raw repository signals, and the need for stakeholder explainability.

To ensure a structured engineering process, the requirements for the CT estimation algorithm are categorized into functional and non-functional requirements. In this thesis, we adhere to the formal definitions provided by the 'ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary' (2017):

> **3.1704 functional requirement:** requirement that specifies a function that a system or system component shall perform.
>
> **3.2621 nonfunctional requirement:** software requirement that describes not what the software will do but how the software will do it.

Furthermore, we provide a list of technical constraints that must be taken into account during the integration of the algorithm into the MECOIS project.

To prioritize the requirements we apply the MoSCoW method (**M**ust, **S**hould, **C**ould, **W**on't have) by Clegg and Barker (1994). *Must-have* requirements are necessary to obtain a minimal version of the algorithm and a usable export interface for the subsequent downstream analytics (e.g. for transfer pricing). *Should-have* requirements represent improvements motivated by the limitations found in the literature and are expected to increase robustness and accuracy in realistic repository settings. *Could-have* requirements are desirable for future evolution of the system but are not essential for validating the thesis' contribution. *Won't-have* requirements are explicitly out of scope for this thesis.

3.1 Functional Requirements

The functional requirements specify *what* the algorithm must do to compute and expose CT estimates per contributor and repository. They cover three layers of functionality:

- **Repository-near data acquisition.** The algorithm depends on a trace of development activity that is fine-grained enough to support time-difference reasoning. Therefore, the system must extract commit metadata such as author identity, timestamps, and a change-size proxy (LOC).
- **CT estimation and export.** A baseline implementation of the four-case heuristic by Buchner and Riehle (2022) is required both as the starting point for the thesis' contribution and as the baseline against which improvements are evaluated. The computed results must be aggregated and written into a data lake table that serves as the export interface for downstream analytics.
- **Robustness features motivated by real-world workflows.** The literature indicates that raw Git and GitHub traces are affected by noise (e.g., automation), workflow diversity (e.g., squash-merged PRs), and global collaboration (e.g., working from different time zones). Consequently, the solution should include mechanisms that recover granularity lost through squash-merged PRs, ignore no-effort activity, reduce the impact of extreme outliers, adapt estimation parameters to repository-specific values, and support time zone independent collaboration.

Table 3.1 summarizes the functional requirements derived from these considerations. Requirements FR1–FR3 define the minimum viable end-to-end capability of the CT estimation algorithm: extracting commit traces, implementing the BAlgo, and producing an exportable data lake table. Requirements FR4–FR10 improve interpretability and precision under common development practices (squash-merged PRs, automated or no-effort activity, globally distributed teams, collaborative commits). The remaining items are explicitly excluded from the scope of this thesis but are listed to provide future research ideas for possible improvement.

ID	Requirement	Description	Priority
FR1	Commit Extraction	Extraction of commit metadata (author, timestamp, LOC) from a Git repository.	Must
FR2	BAlgo Implementation	Implementation of the four-case heuristic by Buchner and Riehle (2022).	Must
FR3	Export Interface	Provision of a data lake table suitable for downstream analytics (e.g., transfer pricing).	Must
FR4	Squash Disassembly	Retrieval of commits from squash-merged PRs to increase resolution.	Should
FR5	Outlier Elimination	Identification and exclusion of unrealistic code contributions using LOC standard deviation.	Should
FR6	No-Effort Elimination	Identification and exclusion of no-effort activity such as merge commits.	Should
FR7	Dynamic Regression	Dynamic calculation of the regression line based on project-specific velocity.	Should
FR8	Co-Author Handling	Support for collaborative commits.	Should
FR9	Time Zone Safety	Adjustment of night-time windows based on the author's UTC offset.	Should
FR10	Bot-Activity Elimination	Identification and exclusion of non-human activity from the data set.	Could
FR11	Non-Linear Curve Fitting	Use of non-linear curve fitting for improved fit to observed development behavior.	Won't
FR12	Improved Case 2 Formula	Derivation of a more accurate formula for case 2 commits than the one defined by Buchner and Riehle (2022), to reduce outliers.	Won't
FR13	Novel Algorithm	Development of a novel algorithm not primarily based on another.	Won't

Table 3.1: Functional requirements

3.2 Non-Functional Requirements

The non-functional requirements describe *how* the CT estimation algorithm must behave in the final setting. An algorithm that processes potentially sensitive organizational development data and must remain usable on repositories of realistic size.

- **Privacy and compliance.** Especially in corporate IS scenarios Personally Identifiable Information (PII) must be anonymized early in the data lake to comply with General Data Protection Regulation (GDPR) and to prevent identity leakage into the export interface.
- **Configurability.** The accuracy of the algorithm is dependent on working styles and may need adaptation for specific repositories. Therefore, thresholds that assign commits to specific cases, and constants used in the functions to calculate the CT, must be adjustable without code changes.
- **Scalability.** Corporate IS repositories can contain a high number of commits, often exceeding tens of thousands, nevertheless the CT computation must remain feasible.
- **Improve accuracy.** The IAlgo must enable a measurable evaluation and demonstrate reduced error compared to the unoptimized BAlgo.
- **Extensibility.** The algorithm implementation should allow the integration of additional sources like GitLab or Jira without redesigning the computation core.

Table 3.2 lists these non-functional requirements and their priorities.

ID	Requirement	Description	Priority
NFR1	GDPR Compliance	Anonymization of PII (emails/usernames) using hashing before further processing.	Must
NFR2	Configurability	Configuration-driven control of pipeline and algorithm behavior via external configuration files.	Must
NFR3	Scalability	Processing of large repositories with around 40,000 commits in less than one minute.	Should
NFR4	Plausibility	Reduction of unrealistic outliers in the IAlgo compared to the BAlgo by Buchner and Riehle (2022).	Should
NFR5	Accuracy	Achievement of lower Mean Absolute Percentage Error (MAPE) compared to the BAlgo by Buchner and Riehle (2022).	Could
NFR6	Extensibility	Architectural support for adding new data sources (e.g., GitLab, Jira).	Could

Table 3.2: Non-functional requirements

3.3 Integration Constraints

In addition to the functional scope and quality attributes, the algorithm integration is bounded by concrete technical constraints imposed by the MECOIS system (which is described in chapter 4) and by the upstream data sources. These integration constraints are summarized in Table 3.3.

ID	Requirement	Description	Priority
IC1	Python Backend	Full compatibility with the existing Python 3.11 system backend.	Must
IC2	Spark Data Lake	Use of PySpark DataFrames for all medallion layer transitions.	Must
IC3	API Rate Limits	Compliance with GitHub API limits and utilization of authentication tokens.	Must

Table 3.3: Integration constraints

4 Architecture

4.1 MECOIS System Overview

The Professorship for Open-Source Software at FAU is currently developing the MECOIS project, which assists companies in measuring productivity, assessing impact, and determining transfer pricing in the context of software development. This thesis proposes an extension to the MECOIS project to incorporate a commit-based CT estimation approach, with the objective of enhancing the accuracy of transfer pricing calculations.

This section outlines the engineering infrastructure where the BAlgo will be integrated into. The system is designed as a modular Extract-Transform-Load (ETL) pipeline that extracts software development metadata from a data source, persists it into a multi-layered data lake, and aggregates it for analysis.

4.1.1 High-Level Architecture

The pipeline follows a "medallion" architecture pattern (Bronze → Silver → Gold), orchestrated by a central control unit which improves the data quality in every step. The precondition is an additional step to gather the data from the different sources which needs a modular setup with source pipelines.

The MECOIS pipeline architecture consists of three high-level components which are visualized in figure 4.1. The *SilverToGold* medallion pipeline (emphasized with a thick black box), will contain the CT estimation algorithm.

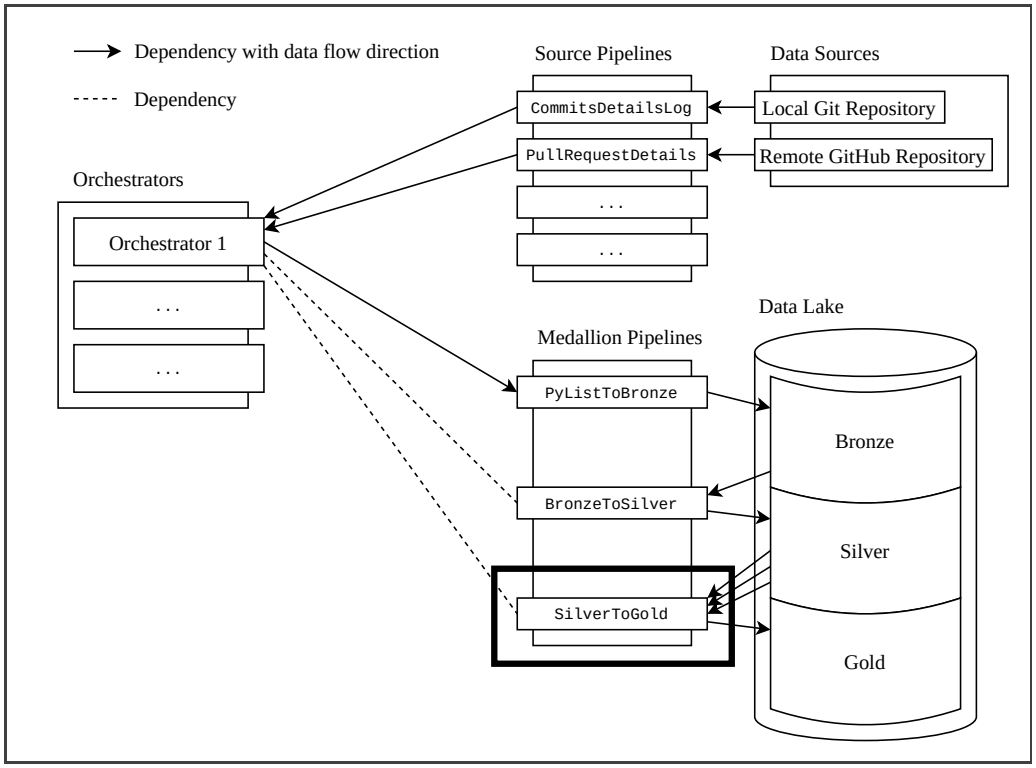

Figure 4.1: System overview: Data extraction, transformation, and layers

1. **Orchestrators:** Defines and executes a combination of pipelines.
2. **Source Pipelines:** Modular units responsible for extracting data from specific sources (e.g., for Commits, or PRs).
3. **Medallion Pipelines:** Pipelines which normalize, clean, aggregate and write data to the data lake. The *PyListToBronze* and *BronzeToSilver* pipelines are designed to execute operations on a single data source at a time. The *SilverToGold* pipeline has the capacity to process multiple data sources concurrently to execute use case-specific logic (e.g., compute the number of open pull requests and issues and perform a comparison).

4.1.2 Orchestration and Workflow

The entry point of the system is the `orchestrator` module. It enforces a strict separation of concerns: the orchestrator manages *flow*, while the individual pipelines manage *logic*.

The `execute_orchestrator()` function initializes the environment and iterates through defined data sources. It also handles the transition of data from the extraction phase directly into the three data transformation layers.

In more concrete:

1. Load pipeline configuration file.
2. Get data lake context.
3. For each data source needed later in the `SilverToGold` pipeline:
   (a) Call the individual modular source pipelines.
   (b) Call the `PyListToBronze`.
   (c) Call the `BronzeToSilver`.
4. Call the use-case-specific `SilverToGold`, that has access to all necessary Silver layer data for further analysis.

4.1.3 Modular Source Pipelines

Each data source (e.g., `CommitsDetailsLog`) is handled by a specialized pipeline class inheriting from a generic pipeline abstraction. This ensures consistency while allowing specific extraction logic.

Each Git/GitHub related modular source pipeline executes the following steps for every requested repository:

1. Add the data source's requestor execution to the pipeline
2. Add the `ProjectNameTransformer` execution to the pipeline
3. Add the `FileWriter` execution to the pipeline
4. Execute the pipeline

The requestor gathers the specified data using either the `git log` command, which provides metadata commit information within a locally cloned Git repository, or using the GitHub API, to get metadata about issues or PRs. The `ProjectNameTransformer` adds the owner and repository name to the dataset. This allows for the distinction of multiple repositories at a later stage. The role of the `FileWriter` is to write the provided data to the specified location in the data lake, or to provide it to another pipeline in case of the modular source pipelines. The sequence diagram in figure 4.2 illustrates the execution flow of the `CommitsDetailsLog` source pipeline. As outlined in section 4.1.1, the orchestrator also controls the medallion pipelines, which is not displayed in this diagram.

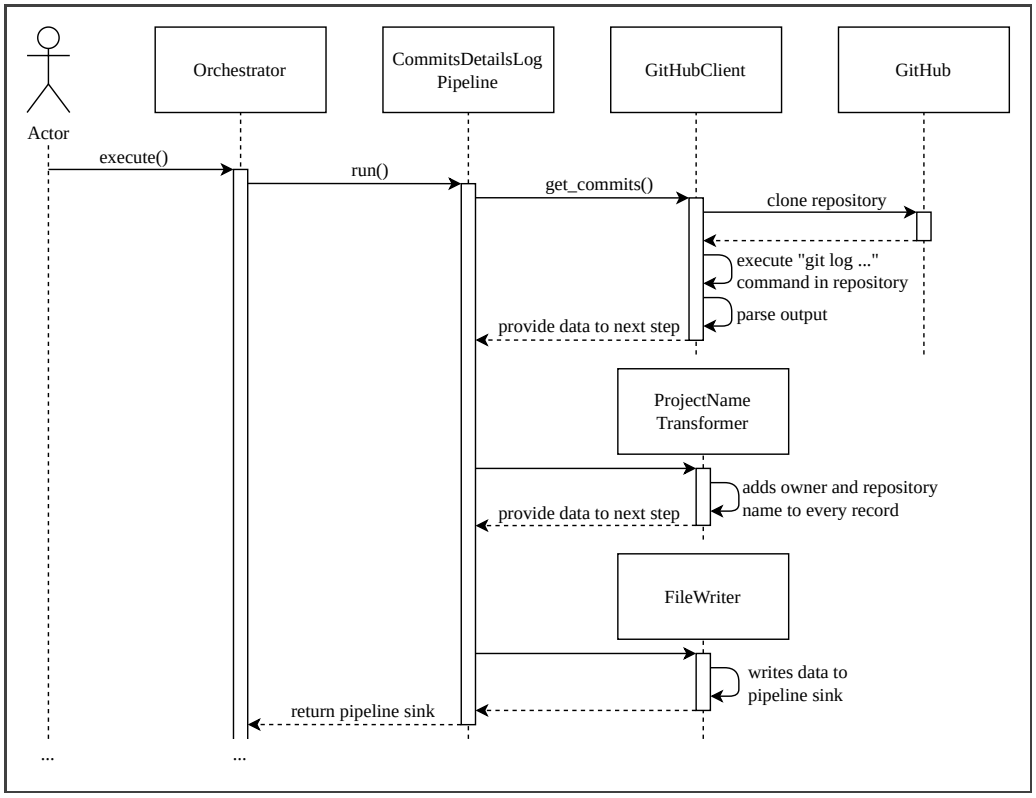

Figure 4.2: Execution flow of the `CommitsDetailsLog` pipeline

To provide a more concrete example, appendix A presents the output of the `git log` command with its parameters. This command is executed by the `CommitsDetailsLog` pipeline. The example repository contains two commits from the same user. Names and email addresses in the data have been modified to fictional values.

4.1.4 Data Lake Layering

The system persists data through three stages of refinement, defined in the pipelines `PyListToBronze`, `BronzeToSilver`, and `SilverToGold`.

1. **Bronze layer:** Raw ingestion. Data is saved as-is from the pipeline sink.
2. **Silver layer:** Validated schema. Columns are cast to correct data types (e.g., timestamps, integers), personal data is anonymized and null values are handled.
3. **Gold layer:** Aggregation. Specific logic required for each use case. Uses Silver layer to calculate meaningful metrics for the specific scenario.

`PyListToBronze` Pipeline

The `PyListToBronze` pipeline acts as the primary ingestion entry point for the data lake. It accepts the raw Python lists of dictionaries generated by the source pipelines (e.g., metadata extracted from `git log`) and converts them into a distributed format suitable for analytics (i.e., Spark DataFrames).

The primary objective of this pipeline is to persist the data in its rawest form without applying any logic or data transformations. This ensures that the Bronze layer serves as an immutable historical record. By maintaining the data "as-is", the system preserves the ability to re-process data from scratch if downstream transformation logic changes, without needing to re-query the original external APIs or repositories.

To continue with the example from above, appendix B contains the data of the Bronze layer table, after is has been parsed and loaded into the data lake. At this point, the keys `"diff_sha"`, `"co-authors"` and `"signed-by"` in this data can be ignored, as they are part of the subsequently added features described in chapter 5. However, what should be mentioned are the keys `"owner"` and `"repository"`, which are added to each data object by the pipeline step `ProjectNameTransformer` as mentioned above.

`BronzeToSilver` Pipeline

The `BronzeToSilver` pipeline is responsible for data cleaning, standardization, and schema enforcement. It reads the raw data from the Bronze layer and transforms it into a refined, trusted dataset. The operations performed in this stage include:

- **Column Pruning:** Dropping columns that are not required for subsequent analysis.
- **Standardize Timestamps:** Parsing and converting various string representations of dates and times into uniform, native Timestamp objects.
- **Data Privacy:** Anonymizing PII, to comply with the GDPR. Specifically, developer email addresses and usernames are hashed to an identity to ensure that no sensitive data propagates to the Gold layer.

Appendix C contains the data of the two commits from the example above, after the cleaning steps from the `BronzeToSilver` pipeline. Here, the key `"AuthorDate_tz"` can be ignored, as it is also part of the subsequently added features described in chapter 5.

`SilverToGold` Pipeline

The `SilverToGold` pipeline will contain the core logic and the implementation of the CT estimation algorithm. Unlike the previous pipelines which generally process one data source at a time, this pipeline can perform more complex aggregations and joins across multiple Silver layer tables (e.g., correlating Commits with PRs). The concrete implementation of this algorithm is described in chapter 5.

4.1.5 Pipeline Configurability

The pipeline is driven by a YAML configuration file, ensuring that runtime behavior can be altered without code changes.

```yaml
DataExtraction:
  write_to_storage: false   # Set 'true' for local debugging
  GitHub:
    - owner: "owner1"       # https://github.com/owner1/repo1
      repo_name: "repo1"
    - owner: "owner2"       # https://github.com/owner2/repo2
      repo_name: "repo2"

DataLake:
  table_path: path/to/folder/tables/
  process_file: path/to/file/data_lake_transformation.json
```

Figure 4.3: Configuration file snippet (pipeline)

4.2 Architecture for Algorithm Integration

This section specifies the architectural integration of the BAlgo and later the IAlgo into the existing MECOIS pipeline. In contrast to the modular source pipelines and the `PyListToBronze` / `BronzeToSilver` steps (which each operate on one data source at a time), the algorithm requires a *multi-source* view and is therefore implemented as a dedicated `SilverToGold` use-case pipeline. The purpose of this section is to define the integration boundaries and interfaces for the concrete design and implementation described in chapter 5.

4.2.1 Integration into the Medallion Pipeline

Architecturally, the algorithm is a calculation and aggregation unit that consumes one or more Silver layer tables and emits a single Gold layer table for export and subsequent downstream analytics (FR3).

Figure 4.4 summarizes the integration boundary and the data-flow at the architectural level.

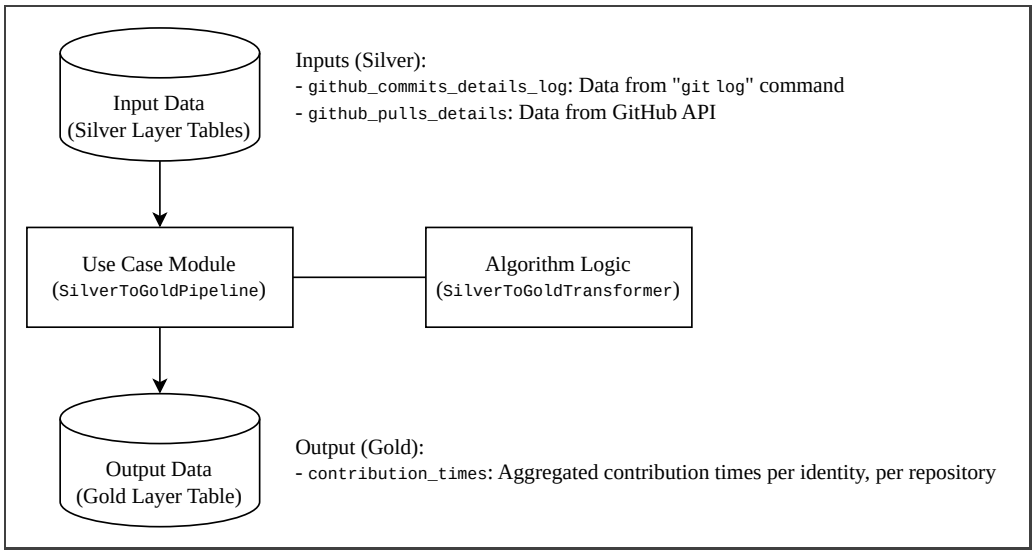

Figure 4.4: Architecture of the algorithm integration

4.2.2 Required Silver Layer Interfaces

The Silver layer tables in the data lake are stored as Spark DataFrames (IC2). The first step of the `SilverToGold` pipeline is to extract only the required data and provide it to the `SilverToGoldTransformer`. This keeps the interface stable even when additional inputs are introduced later (NFR6).

The IAlgo described in section 5.2 uses two Silver layer tables:

- **Commit data** (`github_commits_details_log`): per-commit metadata required for temporal classification, LOC value, and author attribution.
- **PR data** (`github_pulls_details`): closed PRs metadata required to replace squashed PRs by their contained commits to increase granularity.

Tables 4.1 and 4.2 define the input table schemas used by the algorithms. Table 4.3 define the output table schema produced by the algorithms. The presence of the mark `x` in column **BAlgo** indicates that the field is used by the implementation of the algorithm described by Buchner and Riehle (2022). The presence of the mark `x` in column **IAlgo** indicates that the field is used by the improved version. Additional fields in the Silver layer tables which are not used by the algorithm are not listed.

Field	Purpose	BAlgo	IAlgo
`owner`	Project scoping	x	x
`repository`	Project scoping	x	x
`AuthorDate`	Timestamp	x	x
`AuthorDate_tz`	Time zone offset		x
`author_name`	Contributor identity	x	x
`added`	LOC proxy	x	x
`removed`	LOC proxy	x	x
`branches`	Branch information		x
`diff_sha`	Change identity		x
`co-authors`	Co-contributor		x

Table 4.1: Silver layer table schema: `github_commits_details_log`

Field	Purpose	BAlgo	IAlgo
`owner`	Project scoping		x
`repository`	Project scoping		x
`state`	Lifecycle scoping		x
`merge_commit_sha`	PR-commit identifier		x
`commit_shas`	PR-contained commit identities		x
`commit_diff_shas`	PR-contained change identities		x

Table 4.2: Silver layer table schema: `github_pulls_details`

Field	Purpose	BAlgo	IAlgo
`project_name`	Project scoping	x	x
`owner`	Contributor identity	x	x
`contribution_time`	Aggregated CT	x	x

Table 4.3: Gold layer table schema: `contribution_times`

4.2.3 Control Interface and Configuration Separation

The algorithm integration distinguishes between:

- **Pipeline configuration** (global): defines which repositories are extracted and where the data lake is located (section 4.1.5).
- **Algorithm configuration** (use-case): defines assumptions and thresholds used by the CT computation.

From an architectural perspective, this separation prevents coupling algorithm calibration (e.g., case thresholds, working-day assumptions, night-time definitions) to the system-wide orchestration and improves reuse of the same extraction pipeline for other analyses.

4.2.4 Internal Transformer Architecture

Within the `SilverToGoldTransformer` step, the algorithm is structured as a stage pipeline with clearly defined responsibilities. This is not an implementation detail, but rather a boundary that improves maintainability. While the algorithm's runtime is not optimal, it is simple to understand.

Figure 4.5 provides the high-level stage architecture that mirrors the later implementation described in chapter 5.

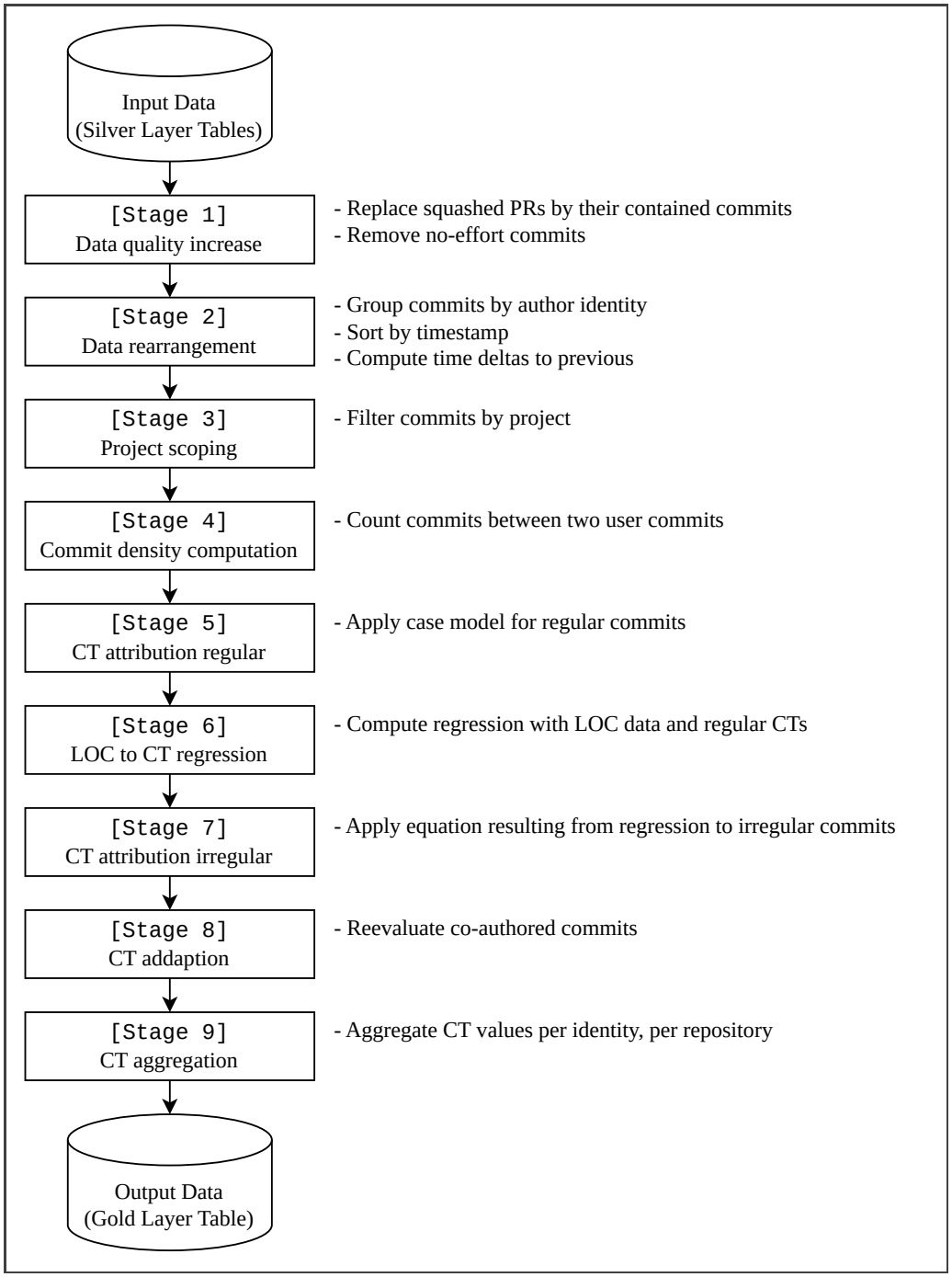

Figure 4.5: Stage architecture of the `SilverToGoldTransformer`

5 Design and Implementation

While section 4.1 outlined the high-level pipeline setup of the MECOIS project and section 4.2 described the architectural prerequisites for the integration of the new component, this chapter focuses on the design and subsequent implementation of the CT estimation algorithm within the MECOIS system architecture.

In order to address the discrepancy between classic coarse-grain methods and the objective of achieving precise CTs recorded between cost centers to facilitate effective transfer pricing, Buchner and Riehle (2022) proposed a novel, fine-grained method for determining the CT on an individual commit. As previously described in chapter 2, this method has significant limitations. In this chapter we will adopt the fundamental concepts of the algorithm proposed by Buchner and Riehle (2022) as BAlgo and extend it significantly to address the requirements defined in chapter 3, resulting in the IAlgo. The proposed approach is expected to result in enhanced data quality, and robustness. The following sections provide a detailed description of the integration process and of the subsequent development and implementation of the extensions necessary for improving the algorithm.

5.1 Integration of the Base Algorithm into the MECOIS Project

5.1.1 Base Algorithm Overview

The core design philosophy is to algorithmically reconstruct the time spent on the development of a specific software artifact by analyzing the metadata of the version control history. The base logic differentiates between ”regular” contributions, where a temporal relationship exists between commits, and ”irregular” contributions, where the CT must be estimated statistically (Buchner & Riehle, 2022).

The design categorizes every commit into one of four processing paths based on the time difference (Δt) from the previous commit by the same author:

1. **Continuous Work ($\Delta t < 360$ min):** The timestamps suggest a continuous block of work. The time difference is assigned directly as CT.
2. **Night-Interrupted Work ($360 \leq \Delta t \leq 720$ min):** The interval likely includes an overnight period. A "night share" factor is calculated to exclude non-working hours from the CT estimation.
3. **Day-Spanning Work ($720 < \Delta t \leq 2160$ min):** The interval spans more than a standard work cycle. The CT is assigned proportionally based on the typical commit frequency distribution of the organization.
4. **Irregular Work ($\Delta t > 2160$ min or no predecessor):** No immediate temporal context exists. The CT is derived via a static linear regression model correlating LOC to Δt.

The structure of the decision tree in figure 5.1 reflects the implementation of the four-case heuristic (upper bound, lower bound, middle bound).

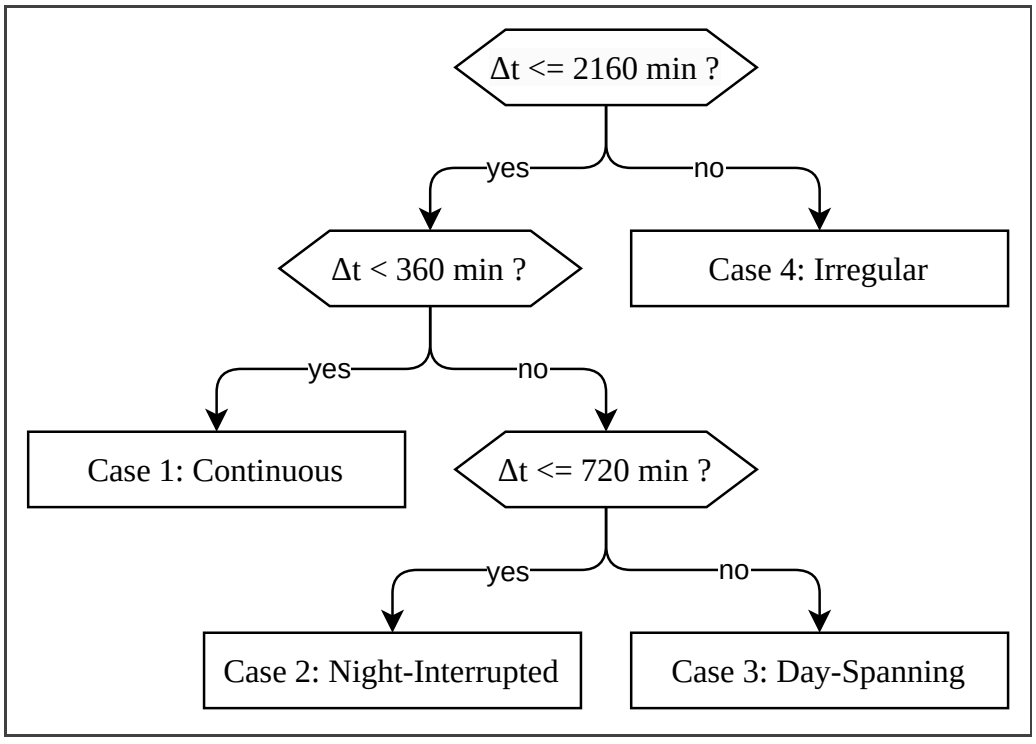

Figure 5.1: Four-case decision tree

5.1.2 Base Algorithm Implementation

The BAlgo (FR2) is implemented as a dedicated *use-case module* next to others within the `analytics` folder. In alignment with the MECOIS medallion architecture (chapter 3), the implementation is encapsulated in the `SilverToGold`

pipeline and its associated transformer `SilverToGoldTransformer`. With this modular approach, the existing pipeline structure can be used by multiple modules, without changing the functionality and the algorithm does not interfere with other use-cases.

Pipeline-Level Integration

The transformation logic is embedded into the MECOIS medallion architecture as a dedicated `SilverToGold` pipeline. At runtime, the pipeline requests the required Silver layer table(s), executes the CT computation, and persists the aggregated result as a Gold layer table in the data lake.

The initial integration of the BAlgo operates solely on commit metadata (FR1) and therefore requests only data from the `github_commits_details_log` pipeline, which is described in section 4.1.

```
Pseudocode: SilverToGold pipeline
Input: silver_table_paths
Output: -

// Initialize pipeline components
requestor  <- DeltaMultiRequestor(tables=silver_table_paths)
transform1 <- SilverToGoldTransformer()
transform2 <- JsonToDataLakeTransformer()
writer     <- OverWriter(table="contribution_times")

// Register pipeline steps
self.add(requestor.bulk_request)
self.add(transform1.create_gold_json)
self.add(transform2.transform_schema_to_delta_lake)
self.add(writer.write)

// Execute the pipeline
self.execute()
```

Figure 5.2: Pseudocode: `SilverToGold` pipeline

Input and Output Data Model

The `github_commits_details_log` pipeline provides the output data of the `git log` command executed in a locally cloned Git repository. The BAlgo by Buchner and Riehle (2022) only uses a small subset of attributes for the CT estimation: repository identifiers (`owner`, `repository`), an author identifier (`author_name`), a timestamp (`AuthorDate`), and a change size expressed as LOC (`added` + `removed`).

For the algorithmic processing, commits are grouped by author identifier into an in-memory Python dictionary, where each commit is reduced to the fields `commit_date` and `loc`. The data structure can be represented as follows: `data` : `author` $\mapsto$ [`commit`$_1, \ldots,$ `commit`$_n$].

After the algorithm completed and assigned a value to each commit, the transformer aggregates the per-commit CT estimates to an author-level sum per repository. The resulting list of records is then converted to the data lake data structure and written to the Gold layer table `contribution_times`, where it can be exported for further analysis or for displaying the data in the front end of the MECOIS project. Table 4.3 from the previous chapter shows the table schema of the export interface (FR3).

All temporal quantities produced and processed by this algorithm are represented in minutes. Minute-level granularity reduces discretization errors in boundary conditions (e.g., the computation of night shares) and ensures that aggregation over many short intervals remains numerically stable. A conversion to hours would only scale the final values for presentation. This is therefore treated as a user-interface concern and is not relevant for the algorithmic design documented in this chapter.

Algorithm Steps inside the Transformer

For each GitHub repository listed in the MECOIS pipeline configuration file, the `SilverToGoldTransformer` executes the BAlgo on the Silver layer commit data in a deterministic sequence of steps:

1. **Repository scoping:** Filter the data by the current repository.
2. **Identity grouping:** Group commits by author identifier.
3. **Commit classification:** Sort commits per identity and compute Δt (time to previous commit), followed by assignment to case 1-4 according to the thresholds defined in section 5.1.1.
4. **Concurrent activity proxy:** Compute the number of repository commits between the current and the previous user commit as a proxy for concurrent activity.
5. **CT assignment:** Apply the per-case calculations and assign the CT value to each commit.
6. **Aggregation:** Sum CTs per identity and emit one Gold layer record per `(repository, identity)` tuple.

Per-Case Calculations

The following paragraphs present the adapted formulas, originally by Buchner and Riehle (2022), that are used for each case to calculate the CT value per commit (ct).

Case 1: Continuous Work.

In this case, Δt is directly assigned as CT value.

Case 2: Night-Interrupted Work.

cav = Concurrent activity value

$C_{\Delta t}$ = Number of commits performed between current and previous user commit

C = Total number of commits in repository

$$cav = \frac{C_{\Delta t}}{C} \tag{5.1}$$

ns = Night share

Δt_{night} = Minutes of Δt that take part during night-time

N = Duration of one night

$$x_i = \begin{cases} 1, & \text{if } i \in \text{nighttime}, \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

$$ns = \frac{\Delta t_{night}}{N} = \frac{\sum_{i=timestamp_{prev}}^{timestamp_{cur}} x_i}{\sum_{i=timestamp_{NightBegin}}^{timestamp_{NightEnd}} 1} \tag{5.3}$$

W = Duration of one working day (e.g. 8h)

$\Delta t_{day} = \Delta t - \Delta t_{night}$

$$ct = ns * cav * W + (1 - ns) * \Delta t_{day} \tag{5.4}$$

Case 3: Day-Spanning Work.

$$ct = cav * \Delta t \tag{5.5}$$

Case 4: Irregular Work.

$$ct = 0.04267525 * loc + 258.58249058 \tag{5.6}$$

Calculation of commit density. For Cases 2 and 3, proportional attribution requires an estimate of how many other commits occurred between the current and the previous user commit. This is being realized by sorting all repository commit timestamps once and performing binary-search based range counting. For each identity commit, the number of commits between the two timestamps is computed as the difference of two insertion indices, resulting in $O(\log n)$ per query (see figure 5.3).

```
Pseudocode: Computation of commit density
Input: All timestamps T (sorted), boundaries (t_prev, t_curr)
Output: num_commits for each identity commit

for each commit c_i of an identity do
  start <- upper_bound(T, t_prev)
  end   <- lower_bound(T, t_curr)
  c_i.num_commits <- end - start
end for
```

Figure 5.3: Pseudocode: Computation of commit density

High-level pseudocode.

Figure 5.4 summarizes the steps of the `SilverToGoldTransformer` at a more implementation oriented level, compared to the original algorithmic description.

```
Pseudocode: High-level \ac{BAlgo}
Input: CommitsSilver
Output: GoldRecords

groups  <- groupByIdentity(CommitsSilver)
for each identity u in groups do
  sort groups[u] by timestamp
  computeTimeDiffAndAssignCase(groups[u])
end for
filterByRepo(groups)
computeCommitDensity(groups, commits.timestamps) // Case 2/3
computeNightShare(groups)                        // Case 2
computeWorkTime(groups)                          // Case 1-4
return aggregatePerIdentity(groups)
```

Figure 5.4: Pseudocode: High-level BAlgo description

5.2 Development of the Improved Algorithm

The full implementation of the BAlgo described in section 5.1, can be seen in appendix D. To improve the robustness and exactness of this algorithm, we developed the following enhancements, resulting in the IAlgo.

5.2.1 Disassembly of Squashed Commits

A significant challenge in analyzing Git history is the practice of "squashing" commits. This workflow compresses multiple atomic changes into a single history entry to maintain a clean main branch. However, this obscures the actual time distribution. A single squashed commit may represent days or months of development effort but appears as a single data entry. The BAlgo would categorize this as an "Irregular" commit and apply a generic regression, likely under or overestimating the CT, since it uses only the LOC and elapsed time as metrics.

To mitigate this, we found a method to *de-squash* commits that result from a GitHub PR that was merged into the main branch with the "squash and merge" option (FR4), as long as the information about the commits contained in the PR is still retrievable from GitHub via its API. The squashed commit can then be replaced by the more detailed data. As a consequence, the algorithm that assigns the time to each commit is now more precise.

The plan looks like this:

- For all *squash-merged PR* commits in main branch:
  - Import all commits contained in the PR to the data
  - Remove the *squash-merged PR* commit from the data

The merge commit of a PR in the main branch can be identified by utilizing the `merge_commit_sha` value of the PR. This value can be retrieved from the PR's metadata via the GitHub API endpoint `/pulls?state=closed`. However, in order to implement the previously described concept, it is necessary to differentiate between the three options for merging a PR into a branch in GitHub. Therefore, a comparison of the metadata of merged PRs is necessary:

- *merge*: A merge commit (`M`), which has two parents is added to the main branch. One parent is the previous commit on main branch (`B`) and the other parent is the last commit of the PR (`D`). All PR's commits keep their Secure Hash Algorithm (SHA) value. (see figure 5.5)

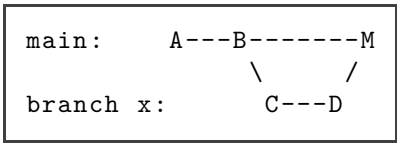

Figure 5.5: Visualization of the PR merge option *merge*

- *squash*: One new commit `S` (with new SHA value), that contains changes of all PR's commits (`C`, `D`) appears in main branch. (see figure 5.6)

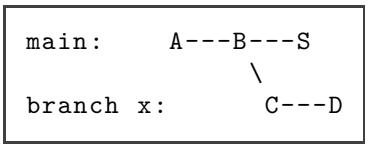

Figure 5.6: Visualization of the PR merge option *squash*

- *rebase*: All PR's commits (`C`, `D`) (with new SHA values -> `C'`, `D'`) appear in main branch. (see figure 5.7)

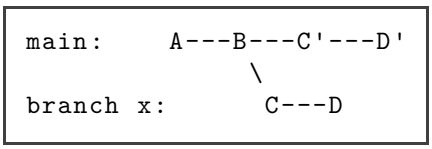

Figure 5.7: Visualization of the PR merge option *rebase*

After this analysis, we concluded that this metadata is only enough to differentiate *merge* from *squash* and *rebase*, but not to clearly identify *squash*. Therefore, we need to add another metric which will need an additional implementation step. For every commit we need to calculate a *diff SHA* value. Which identifies commits only by their changes and commit time. Unlike the existing SHA value, which additionally includes metadata, like the parent and the current state of the whole repository. With the new metric we are now able to differentiate the *squash* and *rebase* methods. However, there is one exception: A squashed PR which contains only one commit, will be identified as a rebased PR. For this particular instance, since there is no need to *de-squash* anything, we can disregard it.

We found the following approach to distinguish merged PRs by their merge method:

- *merge*: If all PR's commit SHA values are found in main branch (else is either squash or rebase)

- *squash*: If **none** of the PR's commit *diff SHA* values are available in main branch
- *rebase* (or squash containing one commit): If **all** of the PR's commit *diff SHA* values are available in main branch

Implementation. First of all we need to make the data needed for the above developed concept available. From the existing `github_commits_details_log` pipeline currently only SHA values and the branch information of every commit is available. The metrics still needed are the information about PRs and the *diff SHA* values of the commits, in order to identify rebased commits.

Rebased commits have a different SHA value than their equivalent origin, because the hash also depends from the branch information. Since an identifier, which is only dependent on the actual change in the code, is not part of the available metadata, we need to calculate it and adapt the modular source pipeline.

After the commit data from the `git log` command is collected, for every commit SHA value, the pipeline is executing the following steps:

1. Execute the following command:

   ```
   git -C <path-to-repo> diff-tree --no-commit-id <commit-sha>
   ```

   This command lists only the changes, that the commit represents in the repository, without adding the commit SHA to the output.
2. Calculate a hash value with the output of the previous command.
3. Save hash value in the key `"diff_sha"` of the commit.

Now we need to add a complete new pipeline which gets the PR information and the containing commit list, from the repository by using the GitHub API. The new modular source pipeline `github_pull_details` uses two API endpoints in order to get this information.

1. Endpoint to get all PR information:

   ```
   /repos/<owner>/<repo>/pulls?state=all
   ```
2. Endpoint to get the contained commits:

   ```
   /repos/<owner>/<repo>/pulls/<pr-id>/commits
   ```
3. For every PR save a list of SHA values and diff SHA values of the contained commits.

Now that we have all required data, we can implement the developed concept in the method `get_desquashed_commit_data()`. The method first filters the data to the default branch (i.e., main) to prevent repeated assignment of CT

values. It then constructs two sets: `main_sha_set` (commit identities) and `main_diff_sha_set` (change identity). PRs are reduced to closed PRs with a `merge_commit_sha`.

For each PR whose merge commit is present on `main`:

1. if all PR commit SHA values are already on `main`, the PR is a normal merge

   -> nothing to do

2. if none of the PR commit SHA values are on `main` and all PR diff hashes are present, the PR is treated as rebase

   -> nothing to do

3. otherwise, the PR is classified as squash

   -> the PR merge commit is removed from the data and the commits contained in the PR are added to the data

Figure 5.8 the explained approach as pseudocode.

```
1  Pseudocode: De-squashing of squashed PRs
2  Input: Commits (main), PRs (closed)
3  Output: De-squashed commit list
4
5  main_sha_set      <- {c.sha | c in Commits}
6  main_diff_sha_set <- {c.diff_sha | c in Commits}
7  for each pr in PRs where pr.merge_commit_sha in main_sha_set
8    if pr.commit_shas subset of main_sha_set then
9      // merge
10     continue
11   else if pr.commit_shas disjoint with main_sha_set and pr.
       commit_diff_shas subset of main_diff_sha_set then
12     // rebase (or single-commit squash)
13     continue
14   end if
15   // remove PR commit
16   main_sha_set <- (main_sha_set \ {pr.merge_commit_sha})
17   // add PR contained commits
18   main_sha_set <- main_sha_set union pr.commit_shas
19 end for
20 return {c in Commits | c.sha in main_sha_set}
```

Figure 5.8: Pseudocode: De-squashing of squashed PRs

5.2.2 Elimination of Outliers

Since for irregular commits, the BAlgo applies a regression, to assign a CT based on LOC, we need to make sure, that the underlying data is representative for a normal commit behavior (FR5). The version control history is often polluted by activities that can result in significant changes to files that are not directly proportional to the time spent working. These activities can include automatic code generation, bulk code changes (e.g. refactoring variable names or relocating functions) and dependency updates. Including these data points in the regression would severely skew the *LOC/Time* correlation.

To detect such outliers, the standard deviation of the *LOC/Time* ratio across the dataset is calculated. Commits that fall outside a defined confidence interval are identified. In our case, we conducted a small evaluation of multiple repositories, ranging from ones with less than 100 commits to ones with more than 40,000 commits. As a result, we determined that it is appropriate to identify commits with a LOC value that is more than four standard deviations away from the mean. In order to ensure that the model reflects human effort rather than automated output, these outliers are excluded from the data set used to generate the regression coefficients. The design of the regression is described in section 5.2.4.

Implementation. Outlier elimination is applied directly before fitting the regression coefficients. In the method `calculate_loc_to_time_regression()`, the transformer extracts (LOC, ct) pairs from all regular commits (Cases 1-3) and computes the mean LOC value μ_{LOC} and the standard deviation σ_{LOC}. A commit is retained if $|LOC - \mu_{LOC}| \leq 4\sigma_{LOC}$. The filtered dataset is then used to perform the linear regression (section 5.2.4). This ensures that automated bulk changes, refactorings, or dependency updates do not disproportionately influence the calculated slope and intercept.

```
1 Pseudocode: Outlier filtering before linear regression
2 Input: Pairs = {(LOC_i, ct_i)}
3 Output: FilteredPairs
4
5 mu    <- mean({LOC_i | i in Pairs})
6 sigma <-  std({LOC_i | i in Pairs})
7 return {(LOC_i, ct_i) | |LOC_i - mu| <= 4 * sigma}
```

Figure 5.9: Pseudocode: Outlier filtering before linear regression

5.2.3 Elimination of No-effort Commits

Not all commits necessarily represent significant economic value. Contributions that do not represent any actual code changes, such as simple merge commits, can artificially inflate commit counts. Worse, if these commits are processed as standard "Continuous Work" (Case 1) commits, the CT assigned to them may not be proportional to their actual economic utility. We should therefore remove such commits from the data set (FR6).

We can filter these commits by analyzing the *diff*, which is also required for the de-squash feature described in section 5.2.1. Consequently, commits without code changes, are eliminated from the dataset, ensuring that only commits that represent substantive modifications to the codebase are processed for estimating the CT.

Implementation. No-effort commits can by identified, if the `diff_sha` is either 0 or if it appears multiple times in the dataset. In these cases the commits should be removed of the dataset to prevent overestimating the CTs. Since this data field is already used in the feature in section 5.2.1, we use the existing for-loop to identify and remove the unwanted commit data. Figure 5.10 visualizes the algorithm.

```
1 Pseudocode: Elimination of no-effort commits
2 Input: NormalizedCommits
3 Output: CleanCommits
4
5 for each commit c do
6   drop c if c.diff_sha = 0
7   drop c if c.diff_sha already seen
8 end for
9 return CleanCommits
```

Figure 5.10: Pseudocode: Elimination of no-effort commits

5.2.4 Lines of Code to Contribution Time Regression

For commits classified as "Irregular" (Case 4), Buchner and Riehle (2022) demonstrate that while LOC grows exponentially, CT tends to scale linearly. Therefore, estimating CT requires a regression model derived from the "Regular" commits (Cases 1–3). To produce an algorithm that adapts to different working styles across repositories, we implemented a dynamic regression, which is executed once for each analyzed repository (FR7).

The design utilizes a median regression line to minimize the impact of remaining anomalies. The formula used for estimation is:

$$ct_{linear} = \alpha \cdot LOC + \beta \tag{5.7}$$

Where α represents the time-per-line coefficient and β represents the base overhead per commit. Unlike the fixed coefficients suggested by Buchner and Riehle (2022) ($\alpha = 0.04267525$ and $\beta = 258.58249058$), this implementation recalculates α and β dynamically based on the specific project data, allowing the model to adapt to different programming languages and development styles.

Implementation. The regression parameters are computed once per repository in `calculate_loc_to_time_regression()`. The method is being executed after the CT assignment for the regular commits, since they provide internally consistent target values. After LOC outlier filtering (section 5.2.2), a linear regression model is fitted using `sklearn.linear_model.LinearRegression`. The calculated coefficients α and β are then inserted into the linear equation 5.7 to calculate the CT for the irregular commits based on the LOC value.

Figure 5.11 shows the regression plot for a repository[1] with over 40,000 commits[2].

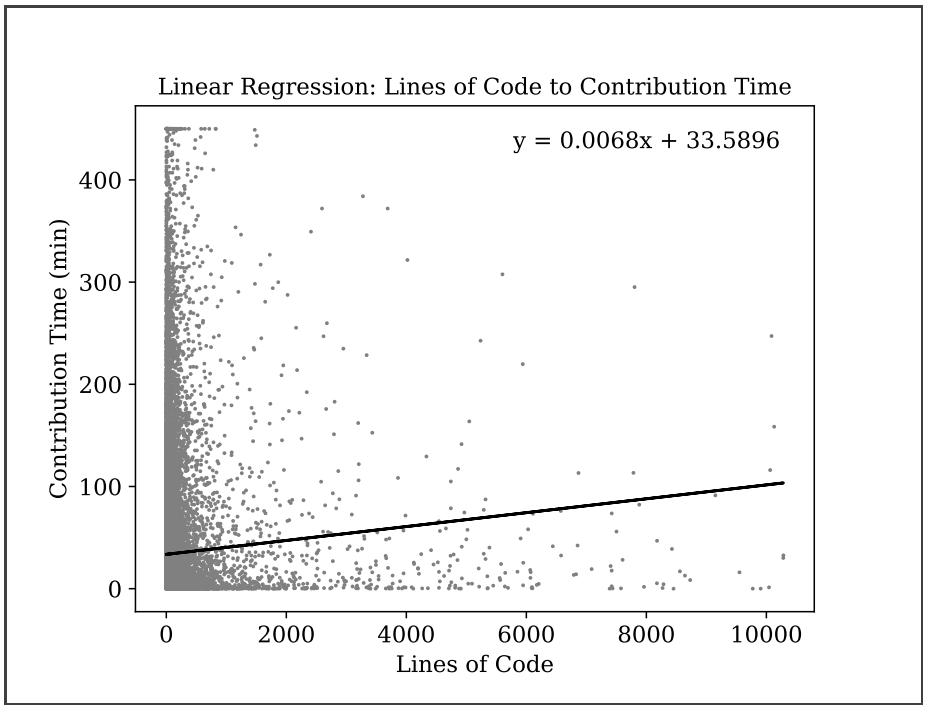

Figure 5.11: Linear regression: Lines of code to contribution time

[1]https://github.com/porsche-design-system/porsche-design-system
[2]The data used was last updated: 24.01.2026 15:00 CET

5.2.5 Handling of Co-Authors

Modern distributed version control systems support the `Co-authored-by` trailer, allowing multiple developers to be credited for a single commit (e.g., during pair programming). The original model assigns CT solely to the primary committer, which can lead to misallocation of costs when developers from different organizational units collaborate on a single artifact. Therefore, we implement a model that assigns CT to all co-authors of a commit (FR8).

Implementing this concept necessitates the extension of the pipeline logic to parse commit descriptions by the key `Co-authored-by` and subsequently enrich the metadata with co-author information, if available.

Now we need to define the CT distribution to such collaborative commits. In our opinion the co-authors should get at most the CT value assigned, that the main author gets. With this approach the co-authors invest at most the same amount of time into the development, than the main author.

Figure 5.12 provides an example for this approach, where a commit has one main author (`Main-A`) and two co-authors (`Co-A1` and `Co-A2`). The initial CT assignment (`Initial ct`) is calculated with the BAlgo (see section 5.1.2). In this example `Main-A` gets assigned 2 hours, `Co-A1` gets 1 hour and `Co-A2` gets 3 hours. Now we need to adjust the co-authors CTs, so that they do not exceed the main author's CT. Therefore, `Co-A1` keeps his assigned time of 1 hour, since it is less than `Main-A`'s time of 2 hours. `Co-A2` however needs to have his CT adjusted from 3 hours to 2 hours (see `Final ct`).

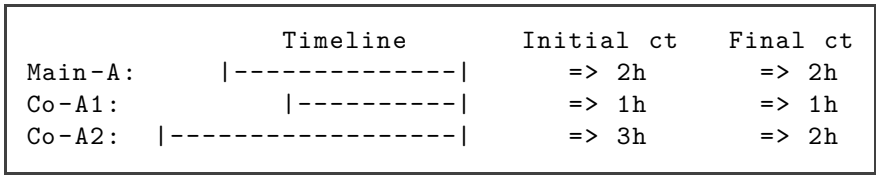

Figure 5.12: Co-authors time assignment visualization

An alternative approach could involve treating co-authors and the main author equally. Subsequently, an equal CT value would be allocated to each party of a collaborative commit. This value could simply be the calculated CT of the main author, if the number of calculations is to be minimized, or the values of all parties could be calculated and the minimum or the mean could be assigned to everyone. As previously stated, we prefer the first method, illustrated in figure 5.12, and describe its implementation in the subsequent paragraphs.

Implementation. Co-author handling is implemented in two phases. First, `get_commit_info_per_identity()` recognizes co-authored commits and addi-

tionally creates a duplicate entry for every extracted co-author and adds it to the respective commits list. The duplicates get an additional key `co-author-duplicate_from`, which stores the primary author's identity. This allows the subsequent time-difference computation to treat co-authors independently, reflecting that a co-author's local commit history may differ from the main author's.

Second, `reevaluate_coauthored_commits()` reconciles duplicates after CTs have been assigned. For every duplicate, the method retrieves the corresponding primary author commit with the same SHA value and compares their computed CTs. The lower value is then used as a conservative estimate to avoid over-attribution in collaborative commits. Additionally, we added an option to scale the assigned CTs of the co-authors by a configurable factor `coauthor_share`. Assigning the value `1` would make no additional change. The pseudocode in figure 5.13 reflects the above description of the concept.

```
Pseudocode: Co-author handling
Input: Commits (with "occasionally" co-authors)
Output: Adjusted CTs per identity

In method "get_commit_info_per_identity()":
for each commit c do
  add c to author(c)
  for each co in c.co_authors do
    add copy of c to co with marker duplicate_from=author(c)
  end for
end for

In method "reevaluate_coauthored_commits()":
for each commit c where duplicate_from exists do
  m <- find commit with same sha in duplicate_from identity
  c.ct <- min(c.ct, m.ct) * coauthor_share
end for
```

Figure 5.13: Pseudocode: Co-author handling

5.2.6 Handling of Time Zones

The calculation of the "night share" (used to reduce costs for overnight commits) relies on defining a specific non-working period. Buchner and Riehle (2022) set the night in their example from 10 p.m. to 7 a.m. However, the application of a static time window is inadequate in the context of globally distributed IS scenarios. A commit made at 2 p.m. in the UTC time zone is during the workday for a developer in London but is effectively "night" for a developer in Tokyo.

To address this, we calculate the "night" interval dynamically according to the

commit's time zone (FR9). By extracting the time zone offset from the commit timestamp, the algorithm adjusts the start and end times of the working day relative to the specific author's locale. This ensures that the exclusion of non-working hours (see equation 5.3) is applied correctly regardless of the developer's geographic location.

Implementation. To obtain the time zone offset, the timestamp parsing needs to be adapted. Now each commit record has the key `AuthorDate_tz` which contains the time zone offset information. The implementation of the transformer method `calc_nightshare_daycoverage()` converts the handled timestamp into a time zone aware datetime using the stored offset. The values `nightshare` and `daycoverage` are then being calculated as described in section 5.1.2.

5.2.7 Algorithm Configurability

In addition to the global pipeline configuration, the CT computation is controlled by a dedicated configuration file (NFR2). This separation is intentional: it allows tuning algorithmic assumptions (e.g., length of a working day or night interval) without affecting extraction or data lake behavior.

The transformer loads the configuration at initialization time and fails fast on missing or malformed configuration. The excerpt in figure 5.14 shows the relevant parameters:

```
working_day_minutes: 480      # 8h working day (minutes)

night_hours:                  # night window (local time)
  start: "22:00"
  end: "07:00"

time_thresholds:              # case boundaries in (minutes)
  regular_max: 2160           # 36h
  subgroup_1_max: 360         # 6h
  subgroup_2_max: 720         # 12h

coauthor_share: 1.0           # fraction assigned to co-authors
```

Figure 5.14: Configuration file snippet (`SilverToGoldTransformer`)

Table 5.1 summarizes how these values influence the computation.

Parameter	Effect on computation
`working_day_minutes`	Reference duration used for proportional assignment in Case 2 and for capping in regular cases.
`night_hours.start/end`	Defines a time zone aware night interval; (section 5.2.6).
`time_thresholds.*`	Case boundaries for classifying commits into cases 1-4 (section 5.1.2).
`coauthor_share`	Scaling factor for co-author attribution after reconciling duplicates (section 5.2.5).

Table 5.1: Semantics of the configuration parameters

This parametrization supports a controlled calibration process when applying the algorithm to new repositories, programming languages, or organizational contexts. In particular, the time thresholds and night-window definitions can be adapted to corporate policies or specific development patterns without code changes.

6 Evaluation

Having verified the functionality of the pipeline and the error-free execution of the BAlgo and the IAlgo with sample data, we proceed to assess the plausibility of the results from the IAlgo (NFR4). To make the analysis reproducible, we selected two publicly available GitHub repositories. All used data for the evaluation was last updated last on 24.01.2026 at 15:00 CET.

First, we use the repository "Porsche Design System" (repository 1)[1], which provides a large main-branch history. This makes it well-suited for statistical evaluation at scale. The development of the selected repository was previously IS, but has been changed to open source development. However, at the time of evaluation, the historical branch references required to reconstruct additional commit-level information for squash-merged PRs (see Section 5.2.1) were no longer available for this repository, because the corresponding PR branches had been deleted.

Therefore, we evaluate a second public repository "GitHub Actions Runner" (repository 2)[2], where sufficient PR branches and referenced commits are still accessible.

In the following sections, the plausibility of the IAlgo is evaluated, and a comparison and analysis of statistical data from both repositories and both algorithms is conducted. Subsequently, the fulfillment of the requirements defined in chapter 3 will be evaluated, and finally, the constraints and limitations of the algorithms will be discussed. The raw outputs of the algorithms for each repository, showing the CT assignments per identity (export interface), can be found in appendix F.

6.1 Plausibility Evaluation

Both, the BAlgo (appendix D) and the IAlgo (appendix E) were adapted to record every case where the computed CT exceeded a hard upper bound and had to be capped. Each commit is processed sequentially, and on every initially set

[1] https://github.com/porsche-design-system/porsche-design-system
[2] https://github.com/actions/runner

CT value ct, a sanity check is applied: ct is compared to a maximum admissible value derived from the elapsed time between the current commit and the previous commit by the same author and an additional threshold per day. If ct is higher than this maximum, it is replaced by the cap value and the event is logged.

Equation 6.1 illustrates the calculation of the cap value, while W is the duration of a working day in minutes (e.g., 480), Δt_i is the amount of elapsed minutes since the previous commit of the same user (e.g., 120), T is the minimum amount of elapsed time (this threshold should be set to the same share, that the upper bound of case 1 of the four-case heuristic has to a defined working day, in order to filter out the commits, where Δt_i is assigned as CT by design), and P is a percentage of CT over-assignment per elapsed working day, that is tolerated (e.g., 20 %).

$$\text{cap}_i = W * (1 + P) * \max\left(T, \frac{\Delta t_i}{1440}\right) \tag{6.1}$$

With the example values above, the equation would result in $\text{cap}_i = 576$. So if the CT assigned previously by the algorithm exceeds this value it would be capped and logged. In other words, this configuration does not allow CT assignments to commits, that assign more than 120 % of a regular working day per elapsed day.

The evaluation was executed on the BAlgo and on the IAlgo with the variable assignments: $W = 480$, $P = 0.25$ and $T = 0.75$ (we calculated T with: 6h (=> limit case 1) / 8h (=> working day) = 0.75).

For each cap event, the evaluation script stores the following attributes: the commit identifier (short SHA), the assigned group (`Regular` or `Irregular`) and subgroup (if applicable), the time difference to the previous commit by the same author Δt in minutes, the originally calculated CT ct in minutes, the limit value cap in minutes, and ct/cap as percentage. The logged events are then sorted by over in descending order and printed as a table. This produces a compact view of the most extreme instances where the estimation procedure would otherwise produce values beyond the chosen plausibility bound and therefore required intervention by the sanity check.

Since no ground-truth time tracking per commit exists for the evaluated repository, the analysis does not claim absolute accuracy. Instead, the cap mechanism is used as an observable plausibility constraint: a cap event occurs whenever the raw estimator would exceed the predefined upper bound from equation 6.1 under the chosen parameters. Consequently, the number and magnitude of cap events provide a comparative indicator of how often the estimation approaches produce extreme outputs that violate this constraint. The evaluation therefore interprets cap-event reductions as evidence of improved stability compared to the baseline under identical conditions.

6.1.1 Results

Tables 6.1 and 6.2 list all cap events registered with the two algorithms using repository 1. The BAlgo triggered 66 cap events, whereas the IAlgo triggered 50 cap events.

Cap events (total=66), showing 10 (sorted descending by *ct*/*cap*)

SHA	Group	Δt (min)	*ct* (min)	*cap* (min)	*ct*/*cap*
1f7ce26	Irregular	2974.2	6502.6	1239.2	524.7 %
9eae731	Irregular	2750.9	2761.3	1146.2	240.9 %
5e5edf8	Irregular	7023.7	5212.8	2926.5	178.1 %
7b4e773	Irregular	6710.9	4804.1	2796.2	171.8 %
c1e5fce	Irregular	4038.0	2878.5	1682.5	171.1 %
792ef42	Regular (2)	713.4	714.0	450.0	158.7 %
dfccdcb	Regular (2)	692.6	693.0	450.0	154.0 %
de85ec6	Regular (2)	679.9	680.0	450.0	151.1 %
45d9831	Regular (2)	654.9	655.0	450.0	145.6 %
c0a28b3	Irregular	8740.9	4794.3	3642.0	131.6 %

Table 6.1: Cap events: BAlgo

Cap events (total=50), showing 10 (sorted descending by *ct*/*cap*)

SHA	Group	Δt (min)	*ct* (min)	*cap* (min)	*ct*/*cap*
8b06bcd	Regular (2)	701.3	702.0	450.0	156.0 %
dfccdcb	Regular (2)	692.6	693.0	450.0	154.0 %
45d9831	Regular (2)	654.9	655.0	450.0	145.6 %
d8dad16	Regular (2)	591.8	592.0	450.0	131.6 %
6f0ebe8	Regular (2)	552.0	553.0	450.0	122.9 %
21e4514	Regular (2)	549.6	550.0	450.0	122.2 %
c35b306	Regular (2)	544.0	545.0	450.0	121.1 %
44afdc8	Regular (2)	545.0	545.0	450.0	121.1 %
e2d26d2	Regular (2)	665.8	544.7	450.0	121.1 %
bcf2e27	Regular (2)	535.0	535.0	450.0	118.9 %

Table 6.2: Cap events: IAlgo

6.1.2 Interpretation

A row in these tables represents a commit for which the estimation logic produced a CT value that was capped to the upper bound described by equation 6.1. In

other words, *ct* was larger than the maximum plausible CT for the corresponding elapsed period (the "sanity bound"). In those cases, the final CT value is forced to the cap, and the magnitude of the correction is represented by *ct*/*cap*.

Two observations are important:

- **Frequency of cap events.** Capping is a rare event by design and should ideally occur only in exceptional situations (e.g., unusual commits or corner cases), not as a regular part of the estimation. In this dataset, the IAlgo required fewer (50) caps than the BAlgo (66). This represents a reduction of cap event of more than 24 %, which indicates that the IAlgo more often stays within the plausible range implied by elapsed time, i.e., it produces fewer extreme overshoots that require correction.
- **Severity and nature of overshoots.** The BAlgo shows several extremely large overshoots, predominantly within the Irregular group, leading to *ct*/*cap* values exceeding 150 % of the maximum plausible time. In contrast, the IAlgo's capped commits are confined to Regular subgroup 2 and show comparatively smaller overshoots. This suggests that the IAlgo avoids extreme inflation in the cases where the BAlgo can generate very large estimates.

Taken together, this supports the conclusion that the IAlgo behaves more plausibly under the same elapsed-time constraint. Specifically, it produces fewer and less severe violations of the sanity bound and therefore requires less corrective intervention by the capping mechanism. In the context of this thesis, where no external ground-truth time tracking is available, this is an important indicator of improved estimation robustness: fewer extreme outliers imply that the algorithm's output distribution is less dominated by artifacts and is more consistent with the observable time structure of the commit history.

6.2 Contribution Time Distribution Summary

To complement the cap-event analysis (which focuses on extreme overshoots), we additionally report descriptive statistics of the resulting per-commit CT distribution. This provides a broader view of how the estimator behaves across typical commits and whether changes affect only capped outliers or also the overall distribution.

6.2.1 Results and Interpretation on Repository 1

Table 6.3 and the boxplot in figure 6.1 summarizes the per-commit CT distribution for repository 1.

Measure	BAlgo	IAlgo
Number of evaluated commits	40656	34510
Number of removed commits	0	6146
Number of added commits	0	0
Sum of assigned times	1554621.19 min	1186408.44 min
Mean	38.24 min	34.38 min
Median	10.78 min	11.55 min
Standard deviation	114.09 min	61.41 min
90th percentile (P90)	98.59 min	94.25 min
95th percentile (P95)	163.61 min	157.27 min
99th percentile (P99)	331.05 min	305.03 min
Maximum assigned time	15216.46 min	2447.39 min

Table 6.3: Contribution time summary (repository 1): BAlgo vs. IAlgo

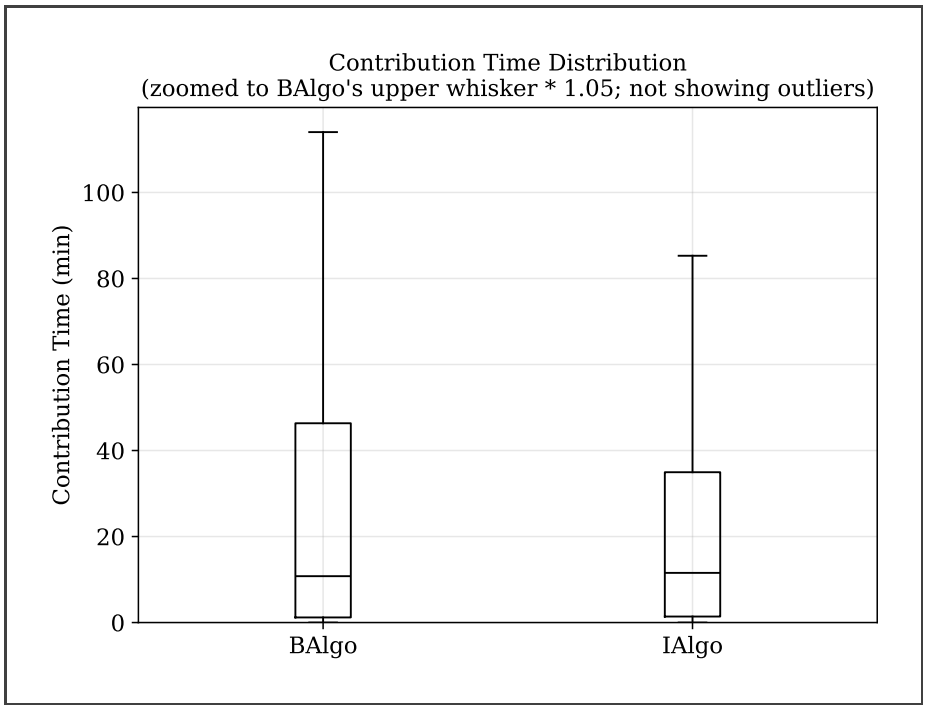

Figure 6.1: Contribution time boxplot (repository 1): BAlgo vs. IAlgo

The statistics in table 6.3 and the boxplot in figure 6.1 indicate that the IAlgo substantially reduces extreme CT assignments compared to the BAlgo. While the median remains in the same range (10.78 vs. 11.55 minutes), the upper tail shifts downward: P90 drops from 98.59 to 94.25 minutes, P95 from 163.61 to 157.27 minutes, and P99 from 331.05 to 305.03 minutes. Most notably, the maximum assigned CT decreases from 15216.46 to 2447.39 minutes and

the standard deviation drops from 114.09 to 61.41 minutes, indicating far fewer extreme outliers and a more stable overall output distribution. The total assigned CT is also lower in the IAlgo, which can be explained, by the elimination of no-effort commits (i.e., commits without any change or duplicated commits).

6.2.2 Results and Interpretation on Repository 2

To specifically assess the impact of the *squash-merged PRs* reassembly feature (Section 5.2.1), we repeat the distribution analysis on repository 2. Unlike the first case study, this repository still exposes sufficient PR branches and referenced commits to reconstruct the individual commits hidden behind squash merges. The results are presented in the same manner than above in table 6.4 and figure 6.2.

Measure	BAlgo	IAlgo
Number of evaluated commits	1513	1635
Number of removed commits	0	35
Number of added commits	0	157
Sum of assigned times	92933.12 min	62984.73 min
Mean	61.42 min	38.52 min
Median	60.30 min	38.17 min
Standard deviation	227.33 min	55.96 min
90th percentile (P90)	79.41 min	47.06 min
95th percentile (P95)	121.44 min	100.93 min
99th percentile (P99)	363.31 min	353.85 min
Maximum assigned time	8538.29 min	450.00 min

Table 6.4: Contribution time summary (repository 2): BAlgo vs. IAlgo

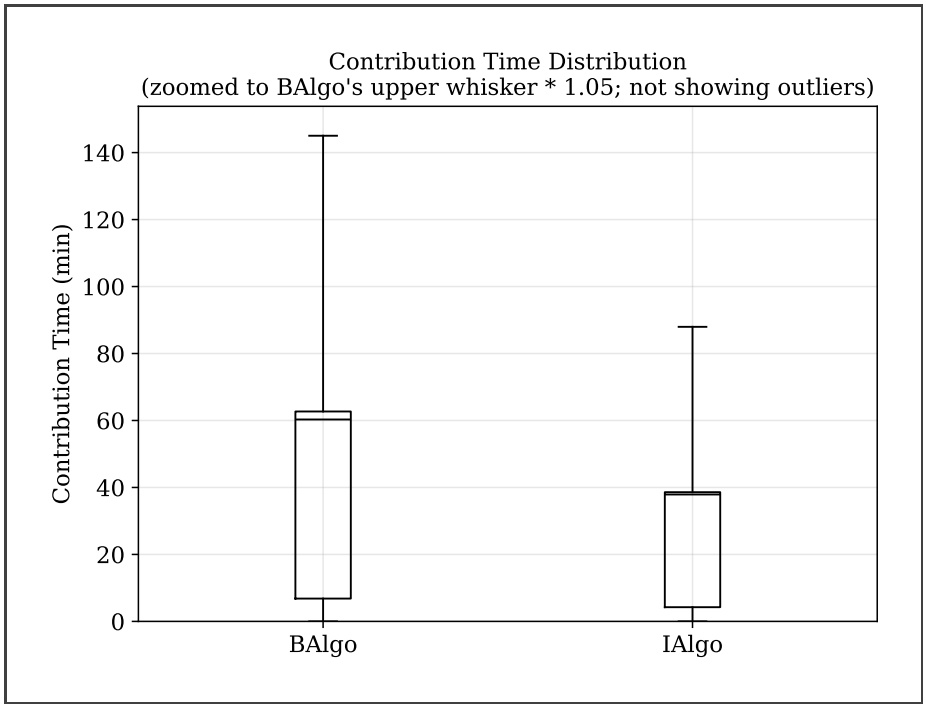

Figure 6.2: Contribution time boxplot (repository 2): BAlgo vs. IAlgo

The results in table 6.4 and the boxplot in figure 6.2, mirror the trend observed with repository 1: the IAlgo reduces dispersion and produces a tighter central tendency. Notably, the median and standard deviation are markedly lower (dropping from `60.30` to `38.17` and from `227.33` to `55.96`), indicating that the IAlgo not only suppresses extreme outliers but also stabilizes the typical per-commit estimates. The *number of removed commits*, contains the number of removed no-effort commits together with the number of removed PR merge commits, which were replaced by a number of contained commits, reflected in the *number of added commits*, together with the added commits due to co-authors.

Comparing the statistical results from repository 1 and repository 2, suggests that the use of the algorithm feature to reconstruct squash-merged PRs in the IAlgo can have a huge impact on the outcome of the CT estimation.

6.3 Validation of Requirements

To validate that the artifact meets the requirements defined in chapter 3, we apply a requirements-based evaluation. In the following tables we are assigning a status code C (defined in equation 6.2) and an evidence to each requirement.

$$C = \begin{cases} 0, & \text{if requirement is not validated} \\ 1, & \text{if requirement is not met} \\ 2, & \text{if requirement is partly met} \\ 3, & \text{if requirement is met} \end{cases} \tag{6.2}$$

ID	Requirement	C	Evidence
FR1	Commit Extraction	3	The MECOIS pipeline provides cleaned commit metadata with the field required by the algorithm.
FR2	BAlgo Implementation	3	The algorithm as described by Buchner and Riehle (2022) was implemented and integrated into the MECOIS project.
FR3	Export Interface	3	The produced table has the columns `project_name`, `owner`, and `working_time`, and contains the per user and project aggregated CT.
FR4	Squash Disassembly	3	Implemented via PR metadata and (diff) SHA logic to replace squash-merged PR commits with the contained commits.
FR5	Outlier Elimination	3	LOC values beyond 4σ are filtered before regression fitting.
FR6	No-Effort Elimination	3	No-change and duplicate commits are dropped to eliminate repeated/no-effort activity.
FR7	Dynamic Regression	3	Irregular-commit time is computed using a repository-fitted linear regression based on regular commits, instead of fixed global constants.
FR8	Co-Author Handling	3	Co-author duplicates are created per identity and adapted with a conservative share rule.
FR9	Time Zone Safety	3	Night-share calculation uses per-commit time zone offsets (`AuthorDate_tz`) and converts timestamps before computing night/day coverage.
FR10	Bot-Activity Elimination	1	Not implemented (was ”Could”).
FR11	Non-Linear Curve Fitting	1	Not implemented (was ”Won’t”)
FR12	Improved Case 2 Formula	1	Not implemented (was ”Won’t”)
FR13	Novel Algorithm	1	Not implemented (was ”Won’t”)

Table 6.5: Validation of functional requirements

ID	Requirement	C	Evidence
NFR1	GDPR Compliance	3	Silver tables contain pseudonymized identity fields (hashed identifiers); the algorithm uses these and does not re-introduce PII into Gold.
NFR2	Configurability	3	CT thresholds and parameters are loaded from `worktime-config.yml` (night window, thresholds, caps, shares).
NFR3	Scalability	2	The pure IAlgo (`SilverToGold` pipeline) was executed with repository 1 (more than 40,000 commits) in less than a minute (36.264 seconds). However, the full pipeline including data gathering and the medaillion process runs longer than one minute.
NFR4	Plausibility	3	Using cap-event logging as a robustness proxy, the IAlgo triggers fewer caps than the baseline on the evaluation repo, indicating fewer extreme overshoots.
NFR5	Accuracy	0	MAPE requires external ground truth (real time worked). This thesis uses sanity checks / cap events as an observable proxy instead.
NFR6	Extensibility	3	The MECOIS pipeline architecture and the algorithm implementation are modular and can easily be extended to additional sources like GitLab or Jira to improve the accuracy of the CT estimation.

Table 6.6: Validation of non-functional requirements

The integration constraints are met by design, since the algorithm was successfully integrated into the MECOIS architecture.

Figure 6.3 summarizes the multidimensional evaluation of the requirements. The values on the y-axis corresponds the values for C that were used above. Additionally to the IAlgo, the graph also contains an evaluation of the BAlgo with the same requirements. It can be concluded, that the IAlgo fulfills more requirements.

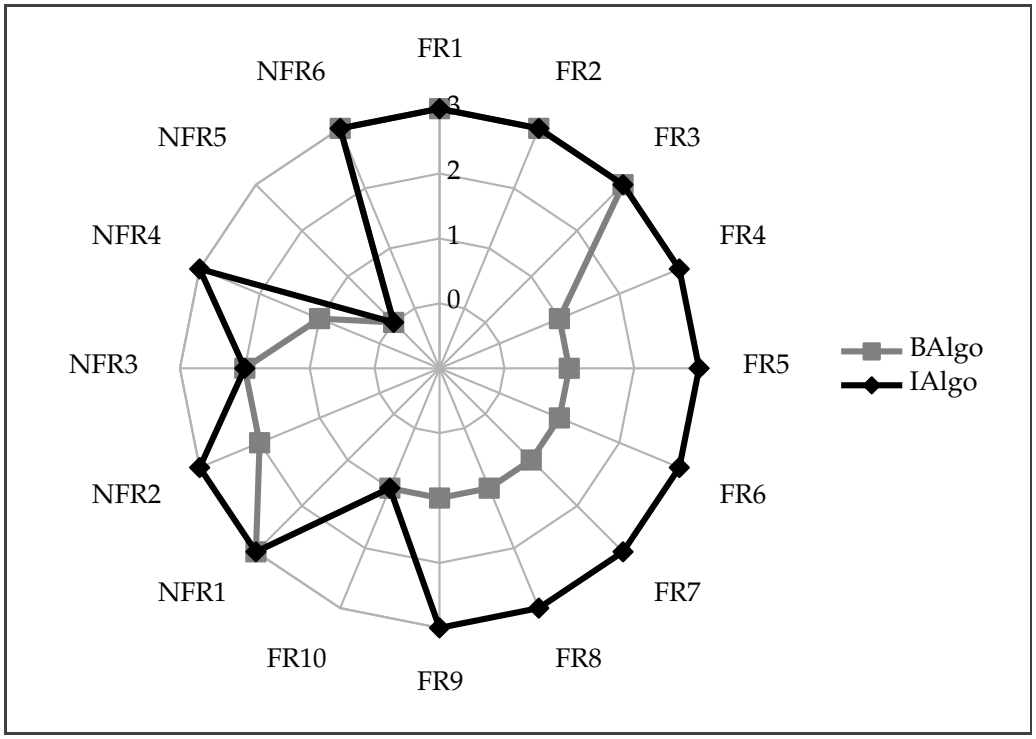

Figure 6.3: Requirements radar chart: BAlgo vs. IAlgo

6.4 Constraints and Limitations

This section summarizes the boundary conditions under which the proposed CT estimation approach is intended to be used. The goal is to make the interpretation of the produced estimates transparent: the artifact is designed to deliver consistent and explainable CT proxies derived from repository traces, while deliberately staying lightweight and reproducible.

6.4.1 Scope and Interpretation

The method infers CT from signals that are reliably available in Git-based repositories (e.g., commit timestamps and change-size proxies such as LOC). This makes the approach broadly applicable and easy to reproduce, but it also means that activities not reflected in commits (e.g., meetings, planning, discussions, debugging without code changes, or time spent in external tools) are not directly captured. The resulting values should therefore be interpreted as *repository-observable development activity* and as a consistent proxy for contribution intensity.

Another point is that ground truth working hours are typically not available for public repositories, which makes direct accuracy validation infeasible in a repro-

ducible setting. Consequently, the evaluation focuses on plausibility and robustness: the IAlgo aims to reduce obvious overestimation effects and generate stable, explainable results under realistic repository artifacts. This supports the validity of the method for analytical use cases, while keeping the claims appropriately scoped.

6.4.2 Data Availability and Quality

The identity attribution relies on Git commit metadata (author name/email and platform-side mappings). In practice, this metadata is not always perfectly consistent (e.g., multiple aliases, `noreply` addresses in UI-generated commits), which can fragment a single contributor into multiple identities. A reliable identity resolution would require additional sources like organizational directories to identify same users with different usernames or email addresses in the Git metadata.

Repositories differ widely in their practices (merge commits vs. squash merges, rebasing/force pushes, monorepo conventions, automation). The IAlgo explicitly addresses some common patterns—most notably by reconstructing granular activity for squash-merged PRs—yet it does not normalize every possible workflow. Importantly, this reconstruction depends on the availability of PR metadata and the PR-contained commits. If these references are not available anymore (for instance because PR branches were deleted), the reassembly cannot be applied retrospectively and the algorithm falls back to the observable main-branch history. As a result, estimates are most comparable within a repository (or across repositories with similar practices), while cross-repository comparisons should be interpreted carefully.

Many projects contain automated commits (dependency updates, formatting, releases). The algorithm filters commits with no changes, but comprehensive bot detection (FR10) was left out. In automation-heavy repositories, remaining bot activity may still influence the estimated CT distribution.

7 Conclusions

This thesis investigated how the CT in software development can be approximated from repository-near data in the context of IS. The motivation is that organizations increasingly rely on internal, cross-departmental collaboration, while established accounting practices often remain too coarse-grained to attribute CT in a transparent manner (e.g., for management reporting and transfer pricing). The objective was therefore to design, implement, and evaluate an approach that derives CT estimates from Git/GitHub traces, remains applicable under real-world working styles, and integrates into the MECOIS pipeline.

The research questions were the following:

> RQ1: *What repository-near methods to determine the CT in the development of software already exist, and what design or implementation challenges arise in real-world scenarios?*
>
> RQ2: *How can the estimation of the CT in the development of software be improved?*

7.1 Summary of Findings

Regarding **RQ1**, the literature review in chapter 2 indicates that existing approaches can be divided into two broad directions.

Traditional effort estimation (e.g., algorithmic models and expert judgment) is primarily designed for predictive, top-down planning and budgeting. These methods can be useful when inputs such as requirements documents or function-point style abstractions are available. However, they are not well suited for the retrospective granular attribution problem that arises in IS, where contributions are distributed, self-directed, and often span organizational boundaries.

Repository-based approaches address this gap by using development traces that are recorded close to where work is performed. In particular, commit-level heuristics such as the method by Buchner and Riehle (2022) use commit timestamps and repository signals to approximate CT.

Three classes of challenges stand out. First, repositories contain systematic noise and workflow artifacts (e.g., merges, automation, and squash-merged PRs) that distort estimates unless the history is reconstructed and cleaned. Second, globally distributed work makes time-based assumptions (such as night-time subtraction) sensitive to time zone context. Third, collaboration patterns such as co-authored commits violate the single-author assumption that many repository heuristics implicitly make. Taken together, these issues explain why naive activity proxies (like only the number of commits or only the LOC contributed) are insufficient to estimate the CT of the development of a software.

With respect to **RQ2**, this thesis proposed and implemented an improved estimation approach that extends the baseline commit-level heuristic and integrates it into the MECOIS architecture (chapters 4 and 5). The improvements follow directly from the requirements derived in chapter 3. Concretely, the solution increases trace granularity through squash disassembly (FR4), reduces distortion through no-effort and outlier elimination (FR5–FR6), improves irregular-commit handling through repository-specific regression (FR7), adds explicit handling of co-authored commits (FR8), and makes night-subtraction robust for distributed development by using time zone aware offsets (FR9).

The evaluation in chapter 6 supports that these changes improve robustness under realistic conditions. Using the same plausibility guardrail configuration, the IAlgo triggers over 24 % fewer cap events than the baseline, which indicates fewer extreme overshoots that require corrective intervention. Moreover, the remaining capped cases are more concentrated, closer to the mean and less impactful than those produced by the baseline. In addition to this empirical robustness proxy, the requirements-based evaluation shows that the improved implementation fulfills a larger share of the functional and non-functional requirements than the base implementation.

7.2 Contributions

The contributions of this thesis are threefold: it advances commit-level CT estimation as a practical method for repository-near analysis in IS, it integrates the implementation of the algorithm by Buchner and Riehle (2022) as BAlgo into the MECOIS project and it adds features to create a more robust algorithm, the IAlgo.

First, this work contributes a refined estimation procedure that is explicitly designed for realistic repository workflows. Instead of treating Git history as a clean activity log, the approach treats it as an noisy trace that requires reconstruction and filtering before it can be used for CT estimation. This is reflected in the implemented support for squash-merged PRs (FR4) and the elimination of no-effort

activity and statistical outliers (FR5–FR6). As a result, the produced CT signals are less dominated by artifacts and more closely aligned with the observable time structure of the repository.

Second, the thesis contributes context-aware attribution features that address typical IS characteristics. By incorporating co-author handling (FR8), the approach reflects collaboration patterns that are common in IS settings (e.g., pair programming). By using time zone safe night subtraction (FR9), it reduces systematic bias that would otherwise arise when distributed contributors are evaluated under a single local-time assumption. These extensions are essential for interpreting CT estimates across organizational boundaries, where differences in working style and location are common.

Third, the thesis contributes an end-to-end integration into the MECOIS pipeline (FR1–FR3) and a structured evaluation strategy. The integration produces an exportable Gold layer table suitable for analytics analytics like transfer pricing.

7.3 Future Work

Several directions remain to strengthen both scientific validity and industrial applicability. A primary next step is validation against ground truth CT per commit data. This thesis deliberately avoids claims of absolute accuracy because no ground truth data exists for the evaluated repository. Future studies could therefore combine the presented algorithm with time reporting data in order to quantify error measures such as MAPE (NFR5) and to better characterize when the estimator is reliable.

A second direction concerns attribution quality. Identity resolution across username aliases and multiple platforms, as well as systematic bot and automation detection (beyond the current no-effort filtering), would reduce noise and improve contributor-level aggregation. This would be particularly relevant for organizations using automation and service accounts within repositories.

Third, the approach could be extended by incorporating additional signals beyond commits. Repository traces capture only work that results in committed changes; meetings, planning, discussions, and other non-code activities remain invisible. Combining Git-based traces with complementary sources (e.g., code review activity, issue / feature trackers, or CI events) could improve interpretability and the CT estimation itself.

Finally, scalability and pipeline hardening remain relevant engineering topics. The current implementation choices could be improved to avoid driver-side collection of large datasets and to better support very large histories, strengthening the scalability requirement (NFR3).

Appendices

A Example Commits Data: Raw

```
$ git log --raw --numstat --pretty=fuller --decorate=full --all --reverse --
    topo-order --parents -M -C -c --remotes=origin
commit ab8922bda46ed182f3c2f299880c33288614fb92
Author:     TestUser <test.user@test-email.com>
AuthorDate: Mon Sep 29 11:13:18 2025 +0200
Commit:     TestUser <test.user@test-email.com>
CommitDate: Mon Sep 29 11:13:18 2025 +0200

    initial commit with empty file

:000000 100644 0000000 e69de29 A        initialfile
0       0       initialfile

commit 83131b5260e724bd9789cf8507613433568efec4
    ab8922bda46ed182f3c2f299880c33288614fb92
Author:     TestUser <test.user@test-email.com>
AuthorDate: Mon Sep 29 11:20:24 2025 +0200
Commit:     TestUser <test.user@test-email.com>
CommitDate: Mon Sep 29 11:20:24 2025 +0200

    second commit

:100644 100644 e69de29 d95f3ad M        initialfile
1       0       initialfile
```

B Example Commits Data: Bronze


```
{
  "AuthorDate": "Mon Sep 29 11:13:18 2025 +0200",
  "CommitDate": "Mon Sep 29 11:13:18 2025 +0200",
  "added": 0,
  "author_mail": "test.user@test-email.com",
  "author_name": "TestUser",
  "branches": [
    "branch2",
    "branch3",
    "main",
    "coauthorbranch",
    "branch1",
    "new-branch",
    "branch4"
  ],
  "co-authors": [],
  "committer_mail": "test.user@test-email.com",
  "committer_name": "TestUser",
  "diff_sha": "0",
  "files": [
    {
      "action": "A",
      "added": "0",
      "file": "initialfile",
      "indexes": [
        "0000000",
        "e69de29"
      ],
      "modes": [
        "000000",
        "100644"
      ],
      "removed": "0"
    }
  ],
  "merge": false,
  "message": "initial commit with empty file",
  "num_files": 1,
  "owner": "owner1",
  "parents": [],
  "refs": [],
  "removed": 0,
  "repository": "repo1",
  "sha": "ab8922bda46ed182f3c2f299880c33288614fb92",
  "signed-by": []
}
```

```
{
  "AuthorDate": "Mon Sep 29 11:20:24 2025 +0200",
  "CommitDate": "Mon Sep 29 11:20:24 2025 +0200",
  "added": 1,
  "author_mail": "test.user@test-email.com",
  "author_name": "TestUser",
  "branches": [
    "branch2",
    "branch3",
    "main",
    "coauthorbranch",
    "branch1",
    "new-branch",
    "branch4"
  ],
  "co-authors": [],
  "committer_mail": "test.user@test-email.com",
  "committer_name": "TestUser",
  "diff_sha": "33b6f34e74ed8e90fe3f929c94d816445dc21af8e9675f36ceac4f97a5e07",
  "merge": false,
  "message": "second commit",
  "num_files": 1,
  "owner": "owner1",
  "parents": [
    "ab8922bda46ed182f3c2f299880c33288614fb92"
  ],
  "refs": [],
  "removed": 0,
  "repository": "repo1",
  "sha": "83131b5260e724bd9789cf8507613433568efec4",
  "signed-by": []
}
```

C Example Commits Data: Silver


```
{
  "added": 0,
  "author_mail": "531c49b17716a57ca53cfd26918da701c3cf33fb",
  "author_name": "531c49b17716a57ca53cfd26918da701c3cf33fb",
  "branches": [
    "main",
    "branch1",
    "branch4",
    "branch3",
    "coauthorbranch",
    "new-branch",
    "branch2"
  ],
  "co-authors": null,
  "committer_mail": "531c49b17716a57ca53cfd26918da701c3cf33fb",
  "committer_name": "531c49b17716a57ca53cfd26918da701c3cf33fb",
  "diff_sha": "0",
  "files": [
    {
      "action": "A",
      "added": "0",
      "file": "initialfile",
      "indexes": [
        "0000000",
        "e69de29"
      ],
      "modes": [
        "000000",
        "100644"
      ],
      "removed": "0"
    }
  ],
  "merge": false,
  "message": "initial commit with empty file",
  "num_files": 1,
  "owner": "owner1",
  "parents": [],
  "refs": [],
  "removed": 0,
  "repository": "repo1",
  "sha": "ab8922bda46ed182f3c2f299880c33288614fb92",
  "signed-by": null,
  "AuthorDate": 1759129998000,
  "AuthorDate_tz": "+0200",
  "CommitDate": 1759129998000,
  "CommitDate_tz": "+0200",
  "was_processed_1": "531c49b17716a57ca53cfd26918da701c3cf33fb",
  "was_processed_2": "531c49b17716a57ca53cfd26918da701c3cf33fb",
  "uuid": "4d3971908a1dd305843c5c494fe91aa0c7ec70afb08e9f1fb745b007bd3eb4b2"
}
```

```
{
  "added": 1,
  "author_mail": "531c49b17716a57ca53cfd26918da701c3cf33fb",
  "author_name": "531c49b17716a57ca53cfd26918da701c3cf33fb",
  "branches": [
    "main",
    "branch1",
    "branch4",
    "branch3",
    "coauthorbranch",
    "new-branch",
    "branch2"
  ],
  "co-authors": null,
  "committer_mail": "531c49b17716a57ca53cfd26918da701c3cf33fb",
  "committer_name": "531c49b17716a57ca53cfd26918da701c3cf33fb",
  "diff_sha": "33b6f34e74ed8e90fe3f929c94d816445dc21af8e9675f36ceac4f97a5e07",
  "files": [
    {
      "action": "M",
      "added": "1",
      "file": "initialfile",
      "indexes": [
        "e69de29",
        "d95f3ad"
      ],
      "modes": [
        "100644",
        "100644"
      ],
      "removed": "0"
    }
  ],
  "merge": false,
  "message": "second commit",
  "num_files": 1,
  "owner": "owner1",
  "parents": [
    "ab8922bda46ed182f3c2f299880c33288614fb92"
  ],
  "refs": [],
  "removed": 0,
  "repository": "repo1",
  "sha": "83131b5260e724bd9789cf8507613433568efec4",
  "signed-by": null,
  "AuthorDate": 1759130424000,
  "AuthorDate_tz": "+0200",
  "CommitDate": 1759130424000,
  "CommitDate_tz": "+0200",
  "was_processed_1": "531c49b17716a57ca53cfd26918da701c3cf33fb",
  "was_processed_2": "531c49b17716a57ca53cfd26918da701c3cf33fb",
  "uuid": "716eaedfa45daab0821490e39386bbf189d2d4db0c78e89ac23e8477c40d8c5b"
}
```

D Base Algorithm Implementation

```
"""Module for transforming silver level data to gold level data."""

import bisect
import datetime
import json
import logging

import yaml
from pyspark.sql import DataFrame as SparkDataFrame

from analytics.cost_calculation.data_export import save_cap_events,
    save_worktime_distribution_summary
from pipeline.controller.pipeline import Pipeline
from pipeline.workflow.transformer.transform_silver_to_gold_pipeline_step import
    TransformSilverToGoldPipelineStep
from services.identity_management.sortinghat_requestor import SortingHatRequestor

logger = logging.getLogger(__name__)


class SilverToGoldTransformer(TransformSilverToGoldPipelineStep):
    """Transformer for converting silver level data to gold level data."""

    def __init__(self, cfg: dict, pipeline: Pipeline = None) -> None:
        """Initialize the transformer with the given context, config, source, and
    pipeline."""
        super().__init__(pipeline=pipeline)
        self.cfg = cfg
        self.sortinghat_requestor = SortingHatRequestor(cfg)
        self.wt_cfg = self._load_worktime_config()
        self.cap_events = []

    def _load_worktime_config(self) -> dict:
        """Load working time configuration from YAML file."""
        config_path = "./analytics/cost_calculation/worktime-config.yml"
        try:
            with open(config_path, encoding="utf-8") as file:
                return yaml.safe_load(file)
        except FileNotFoundError:
            logger.exception("Working time configuration file not found at '%s'",
    config_path)
            raise
        except yaml.YAMLError:
            logger.exception("Error parsing working time configuration")
            raise

    def create_gold_json(self, df: dict[SparkDataFrame]) -> None:
        """Transform the given dataframe from silver level to gold level.

        Args:
            df: The dataframe representing the silver table

        Returns:
            None (calls self.next with the transformed data)

        """
        logger.info("Starting transformation to gold level...")
        try:
            df_commits = df["github_commits_details_log"]
```

```
56          except KeyError:
57              logger.exception("Required dataframes 'github_commits_details_log' or
    'github_pulls_details' not found in input dictionary.")
58              self.next(None)
59              return
60
61          raw_commits = df_commits.toJSON().collect()
62
63          json_commits = [json.loads(commit) for commit in raw_commits]
64
65          repos = self.cfg["DataExtraction"]["GitHub"]
66
67          logger.info("Grouping commit data by identity...")
68          data = self.get_commit_info_per_identity(json_commits)
69          logger.info("Calculating time differences between commits...")
70          data = self.add_hours_to_previous_commit(data)
71          costs_data = []
72          for repo in repos:
73              repo_name = f"{repo['owner']}/{repo['repo_name']}"
74              logger.info("Processing repository '%s'...", repo_name)
75              refined_data = self.filter_data_by_repo(data, repo["owner"], repo["
    repo_name"])
76              if not refined_data:
77                  logger.warning("No commits found for repository '%s'. Skipping
    ...", repo_name)
78                  continue
79              logger.info("Calculating number of commits between two user commits
    ...")
80              refined_data = self.add_num_of_commits_to_previous_commit(refined_data
    )
81              logger.info("Calculating working times for regular commits...")
82              refined_data = self.calculate_costs_regular(refined_data)
83              logger.info("Calculating working times for irregular commits...")
84              refined_data = self.calculate_costs_irregular(refined_data)
85
86              if self.wt_cfg.get("save_worktime_distribution_summary", False):
87                  save_worktime_distribution_summary(
88                      [float(commit["working_time"]) for user_commits in
    refined_data.values() for commit in user_commits],
89                      self.wt_cfg["save_worktime_distribution_summary_path_1"],
90                  )
91
92              # remove "_parsed_date" field before logging / returning
93              for user_commits in refined_data.values():
94                  for commit in user_commits:
95                      if "_parsed_date" in commit:
96                          del commit["_parsed_date"]
97              logger.info("Refined data for repo '%s':\n%s", repo_name, json.dumps(
    refined_data, indent=2))
98
99              logger.info("Aggregating working times per author...")
100             for author, commits_info in refined_data.items():
101                 author_data = {
102                     "project_name": repo_name,
103                     "owner": author,
104                     "working_time": round(sum(commit["working_time"] for commit in
     commits_info), 2),
105                 }
106                 costs_data.append(author_data)
107
108         if self.wt_cfg.get("save_cap_events", False):
109             logger.info("Saving cap events...")
110             save_cap_events(self.cap_events, self.wt_cfg["save_cap_events_path"])
```

```
111
112         self.next(costs_data)
113
114     def filter_data_by_repo(self, data: dict[list], owner: str, repo_name: str) ->
     dict[list]:
115         """Filter list of JSON objects by repository owner and name."""
116         filtered_data = {}
117         for user, user_commits in data.items():
118             filtered_user_commits = [commit for commit in user_commits if commit["
    owner"] == owner and commit["repo"] == repo_name]
119             if filtered_user_commits:
120                 filtered_data[user] = filtered_user_commits
121         return filtered_data
122
123     def get_commit_info_per_identity(self, commits: list[dict]) -> dict[list]:
124         """Get the number of commits per identity for a specific repository."""
125         commits_per_identity = {}
126         for commit in commits:
127             if "main" not in commit["branches"]:
128                 continue
129             try:
130                 author = commit["author_name"]
131             except KeyError:
132                 logger.warning("Dropping commit '%s' as it does not have the field
     'author_name'.", commit["sha"])
133                 continue
134             new_entry = {
135                 "sha": commit["sha"],
136                 "commit_date": commit["AuthorDate"],
137                 "_parsed_date": datetime.datetime.fromisoformat(commit["AuthorDate
    "]),
138                 "tzinfo": commit["AuthorDate_tz"],
139                 "repo": commit["repository"],
140                 "owner": commit["owner"],
141                 # TODO: differentiate between adding / deleting / modifying
142                 "loc": commit["added"] + commit["removed"],
143             }
144             commits_per_identity[author] = [*commits_per_identity.get(author, []),
     new_entry]
145             coauthors = commit.get("co-authors", [])
146             if coauthors:
147                 for coauthor in coauthors:
148                     commits_per_identity[coauthor] = [
149                         *commits_per_identity.get(coauthor, []),
150                         {**new_entry, "co-author-duplicate_from": author},
151                     ]
152         return commits_per_identity
153
154     def add_hours_to_previous_commit(self, data: dict) -> dict:
155         """Add the number of hours to the previous commit of the same author id
    ."""
156         processed_commits = 0
157         time_thresholds = self.wt_cfg["time_thresholds"]
158         regular_max = time_thresholds["regular_max"]
159         subgroup_1_max = time_thresholds["subgroup_1_max"]
160         subgroup_2_max = time_thresholds["subgroup_2_max"]
161
162         for user_commits in data.values():
163
164             # Sort user_commits by parsed timestamps for better performance
165             user_commits.sort(key=lambda x: x["_parsed_date"])
166
167             # Initialize the first commit
```

```
168             user_commits[0]["time_difference"] = -1.0
169             user_commits[0]["group"] = "Irregular (no previous commit)"
170             processed_commits += 1
171 
172             # Calculate time differences for subsequent commits
173             for i in range(1, len(user_commits)):
174                 prev_timestamp = user_commits[i - 1]["_parsed_date"]
175                 curr_timestamp = user_commits[i]["_parsed_date"]
176                 mins_diff = float((curr_timestamp - prev_timestamp).total_seconds
    () / 60)
177                 user_commits[i]["time_difference"] = mins_diff
178                 processed_commits += 1
179                 if processed_commits % 1000 == 0:
180                     logger.info("Processed commits: %d", processed_commits)
181                 # Assign group and sub-group based on time difference
182                 if mins_diff <= regular_max:
183                     user_commits[i]["group"] = f"Regular (up to {regular_max / 60}
     hours)"
184                     if mins_diff < subgroup_1_max:
185                         user_commits[i]["sub-group"] = f"1 (less than {
    subgroup_1_max / 60} hours)"
186                     elif mins_diff <= subgroup_2_max:
187                         user_commits[i]["sub-group"] = f"2 (at least {
    subgroup_1_max / 60} and up to {subgroup_2_max / 60} hours)"
188                     else:
189                         user_commits[i]["sub-group"] = f"3 (more than {
    subgroup_2_max / 60} and up to {regular_max / 60} hours)"
190                 else:
191                     user_commits[i]["group"] = f"Irregular (more than {regular_max
     / 60} hours)"
192         return data
193 
194     def add_num_of_commits_to_previous_commit(self, data: dict) -> dict:
195         """Add the number of commits that happened timewise between two commits of
     the same author id."""
196         # Create a single sorted list of ALL commits with their timestamps
197         all_commits_with_timestamps = []
198         all_commits_with_timestamps.extend(commit["_parsed_date"] for user_commits
     in data.values() for commit in user_commits)
199         all_commits_with_timestamps.sort()
200 
201         # Use binary search for each author's commits
202         processed_commits = 0
203         for user_commits in data.values():
204             for i in range(1, len(user_commits)):
205                 processed_commits += 1
206                 if processed_commits % 1000 == 0:
207                     logger.info("Processed commits: %d", processed_commits)
208                 prev_timestamp = user_commits[i - 1]["_parsed_date"]
209                 curr_timestamp = user_commits[i]["_parsed_date"]
210 
211                 # Use binary search to find commits in the time window
212                 start_idx = bisect.bisect_right(all_commits_with_timestamps,
    prev_timestamp)
213                 end_idx = bisect.bisect_left(all_commits_with_timestamps,
    curr_timestamp)
214                 user_commits[i]["num_commits"] = end_idx - start_idx
215 
216         return data
217 
218     def calculate_costs_regular(self, data: dict) -> dict:
219         """Calculate costs based on the refined data."""
220         # values for "Regular" "2"
```

```
221         working_day = self.wt_cfg["working_day_minutes"]
222         total_commits = sum(len(user_commits) for user_commits in data.values())
223         night_start = datetime.datetime.strptime(self.wt_cfg["night_hours"]["start
    "], "%H:%M").time()
224         night_end = datetime.datetime.strptime(self.wt_cfg["night_hours"]["end"],
    "%H:%M").time()
225         start_min = night_start.hour * 60 + night_start.minute
226         end_min = night_end.hour * 60 + night_end.minute
227         night_mins = end_min - start_min if start_min <= end_min else (24 * 60 -
    start_min) + end_min
228 
229         processed_commits = 0
230         for user_commits in data.values():
231             for commit in user_commits:
232                 processed_commits += 1
233                 if processed_commits % 1000 == 0:
234                     logger.info("Processed commits: %d", processed_commits)
235 
236                 wt = 0.0
237                 time_difference = commit["time_difference"]
238 
239                 if commit["group"].startswith("Regular"):
240                     match commit["sub-group"][0]:
241                         case "1":
242                             wt = float(time_difference)
243                         case "2":
244                             nightshare, daycoverage = self.
    calc_nightshare_daycoverage(commit, start_min, end_min, night_mins)
245                             commit["night_share"] = nightshare
246                             wt = nightshare * ((commit["num_commits"] /
    total_commits) * working_day) + (1 - nightshare) * daycoverage
247                         case "3":
248                             wt = float((commit["num_commits"] / total_commits) *
    time_difference)
249 
250                 commit["working_time"] = wt
251                 commit["working_time"] = self.cap_working_time(commit)
252 
253         return data
254 
255     def calc_nightshare_daycoverage(self, commit: dict, start_min: int, end_min:
    int, night_mins: int) -> tuple:
256         """Calculate nightshare and daycoverage for a given commit using the
    commit's timezone."""
257         # correct the timezone of the commit with the stored tzinfo
258         commit_dt = commit["_parsed_date"].astimezone(datetime.datetime.strptime(
    commit["tzinfo"], "%z").tzinfo)
259 
260         # compute the interval covered by the commit (in the commit's timezone)
261         start_time = commit_dt - datetime.timedelta(minutes=commit["
    time_difference"])
262         end_time = commit_dt
263 
264         nightcoverage = 0
265         daycoverage = 0
266 
267         # iterate minute-by-minute between start_time and end_time (both are
    timezone-aware)
268         current = start_time
269         while current <= end_time:
270             current_min = current.hour * 60 + current.minute
271             in_night = start_min <= current_min < end_min if start_min <= end_min
    else current_min >= start_min or current_min < end_min
```

```
272             if in_night:
273                 nightcoverage += 1
274             else:
275                 daycoverage += 1
276             current += datetime.timedelta(minutes=1)
277 
278         nightshare = (nightcoverage / night_mins) if night_mins > 0 else 0
279         return nightshare, daycoverage
280 
281     def calculate_costs_irregular(self, data: dict) -> dict:
282         """Calculate costs for irregular commits."""
283         slope, intercept = (0.04267525, 60)
284 
285         for user_commits in data.values():
286             for commit in user_commits:
287                 if commit["group"].startswith("Irregular"):
288                     commit["working_time"] = slope * commit["loc"] + intercept
289                     commit["working_time"] = self.cap_working_time(commit)
290 
291         return data
292 
293     def cap_working_time(self, commit: dict) -> float:
294         """Cap working time at configured percentage of working day per elapsed
    day since last commit."""
295         # get commit data
296         calculated_wt = commit["working_time"]
297         time_difference = commit["time_difference"]
298         if time_difference < 0:
299             return calculated_wt
300         # get config data
301         working_day = self.wt_cfg["working_day_minutes"]
302         max_worktime_multiplier = self.wt_cfg["max_worktime_multiplier"]
303         minimum_elapsed_days = self.wt_cfg["minimum_elapsed_days"]
304         # calculate maximum allowed working time
305         elapsed_days = max(minimum_elapsed_days, time_difference / (24 * 60))
306         max_wt = working_day * (1 + max_worktime_multiplier) * elapsed_days
307         if calculated_wt > max_wt:
308             # log the capping event
309             logger.info("Capping working time of %s from %s to %s.", commit["sha
    "], calculated_wt, max_wt)
310             self.cap_events.append(
311                 {
312                     "sha": commit["sha"],
313                     "date": commit["commit_date"],
314                     "group": commit["group"],
315                     "sub_group": commit.get("sub-group", ""),
316                     "time_diff_min": float(time_difference),
317                     "elapsed_days": float(elapsed_days),
318                     "calculated_wt_min": float(calculated_wt),
319                     "new_wt_min": float(max_wt),
320                     "over_by_min": float(calculated_wt - max_wt),
321                 },
322             )
323             return max_wt
324         return calculated_wt
```

E Improved Algorithm Implementation

```
"""Module for transforming silver level data to gold level data."""

import bisect
import datetime
import json
import logging

import numpy as np
import yaml
from pyspark.sql import DataFrame as SparkDataFrame
from sklearn.linear_model import LinearRegression

from analytics.cost_calculation.data_export import save_cap_events,
    save_regression_plot, save_worktime_distribution_summary
from pipeline.controller.pipeline import Pipeline
from pipeline.workflow.transformer.transform_silver_to_gold_pipeline_step import
    TransformSilverToGoldPipelineStep
from services.identity_management.sortinghat_requestor import SortingHatRequestor

logger = logging.getLogger(__name__)


class SilverToGoldTransformer(TransformSilverToGoldPipelineStep):
    """Transformer for converting silver level data to gold level data."""

    def __init__(self, cfg: dict, pipeline: Pipeline = None) -> None:
        """Initialize the transformer with the given context, config, source, and
    pipeline."""
        super().__init__(pipeline=pipeline)
        self.cfg = cfg
        self.sortinghat_requestor = SortingHatRequestor(cfg)
        self.wt_cfg = self._load_worktime_config()
        self.cap_events = []
        self.cnt_removed_commits = 0
        self.cnt_added_commits = 0

    def _load_worktime_config(self) -> dict:
        """Load working time configuration from YAML file."""
        config_path = "./analytics/cost_calculation/worktime-config.yml"
        try:
            with open(config_path, encoding="utf-8") as file:
                return yaml.safe_load(file)
        except FileNotFoundError:
            logger.exception("Working time configuration file not found at '%s'",
    config_path)
            raise
        except yaml.YAMLError:
            logger.exception("Error parsing working time configuration")
            raise

    def create_gold_json(self, df: dict[SparkDataFrame]) -> None:
        """Transform the given dataframe from silver level to gold level.

        Args:
            df: The dataframe representing the silver table

        Returns:
            None (calls self.next with the transformed data)

```

```
56          """
57          logger.info("Starting transformation to gold level...")
58          try:
59              df_pulls = df["github_pulls_details"]
60              df_commits = df["github_commits_details_log"]
61          except KeyError:
62              logger.exception("Required dataframes 'github_commits_details_log' or
    'github_pulls_details' not found in input dictionary.")
63              self.next(None)
64              return
65
66          raw_pulls = df_pulls.toJSON().collect()
67          raw_commits = df_commits.toJSON().collect()
68
69          json_pulls = [json.loads(pull) for pull in raw_pulls]
70          json_commits = [json.loads(commit) for commit in raw_commits]
71
72          repos = self.cfg["DataExtraction"]["GitHub"]
73
74          logger.info("De-squash squashed pull requests...")
75          data = self.get_desquashed_commit_data(json_commits, json_pulls)
76          logger.info("Filter out duplicate and empty commits...")
77          data = self.get_filtered_commit_data(data)
78          logger.info("Grouping commit data by identity...")
79          data = self.get_commit_info_per_identity(data)
80          logger.info("Calculating time differences between commits...")
81          data = self.add_hours_to_previous_commit(data)
82          costs_data = []
83          for repo in repos:
84              repo_name = f"{repo['owner']}/{repo['repo_name']}"
85              logger.info("Processing repository '%s'...", repo_name)
86              refined_data = self.filter_data_by_repo(data, repo["owner"], repo["
    repo_name"])
87              if not refined_data:
88                  logger.warning("No commits found for repository '%s'. Skipping
    ...", repo_name)
89                  continue
90              logger.info("Calculating number of commits between two user commits
    ...")
91              refined_data = self.add_num_of_commits_to_previous_commit(refined_data
    )
92              logger.info("Calculating working times for regular commits...")
93              refined_data = self.calculate_costs_regular(refined_data)
94              logger.info("Calculating working times for irregular commits...")
95              refined_data = self.calculate_costs_irregular(refined_data)
96              logger.info("Reevaluate co-authored commits...")
97              refined_data = self.reevaluate_coauthored_commits(refined_data)
98
99              if self.wt_cfg.get("save_worktime_distribution_summary", False):
100                 save_worktime_distribution_summary(
101                     [float(commit["working_time"]) for user_commits in
    refined_data.values() for commit in user_commits],
102                     self.wt_cfg["save_worktime_distribution_summary_path_2"],
103                     self.cnt_removed_commits,
104                     self.cnt_added_commits,
105                 )
106
107             # remove "_parsed_date" field before logging / returning
108             for user_commits in refined_data.values():
109                 for commit in user_commits:
110                     if "_parsed_date" in commit:
111                         del commit["_parsed_date"]
112             logger.info("Refined data for repo '%s':\n%s", repo_name, json.dumps(
```

```
    refined_data, indent=2))

            logger.info("Aggregating working times per author...")
            for author, commits_info in refined_data.items():
                author_data = {
                    "project_name": repo_name,
                    "owner": author,
                    "working_time": round(sum(commit["working_time"] for commit in commits_info), 2),
                }
                costs_data.append(author_data)

        if self.wt_cfg.get("save_cap_events", False):
            logger.info("Saving cap events...")
            save_cap_events(self.cap_events, self.wt_cfg["save_cap_events_path"])

        self.next(costs_data)

    def filter_data_by_repo(self, data: dict[list], owner: str, repo_name: str) -> dict[list]:
        """Filter list of JSON objects by repository owner and name."""
        filtered_data = {}
        for user, user_commits in data.items():
            filtered_user_commits = [commit for commit in user_commits if commit["owner"] == owner and commit["repo"] == repo_name]
            if filtered_user_commits:
                filtered_data[user] = filtered_user_commits
        return filtered_data

    def get_desquashed_commit_data(self, commits: list[dict], pulls: list[dict]) -> list[dict]:
        """Remove commits that are part of a squashed pull request."""
        # save all commit SHAs in a set for quick lookup
        commits_sha_set = {c["sha"] for c in commits}
        # filter commits by "main" branch (TODO: dynamically use default branch of repo)
        commits_on_main = [c for c in commits if "main" in c.get("branches", [])]
        # save only the commit SHAs and diff SHAs in sets
        main_sha_set = set()
        main_diff_sha_set = set()
        for c in commits_on_main:
            diff_sha = c["diff_sha"]
            main_diff_sha_set.add(diff_sha)
            main_sha_set.add(c["sha"])

        # filter pull requests that are closed and have a merge commit SHA
        relevant_pulls = [pr for pr in pulls if pr.get("state") == "closed" and pr.get("merge_commit_sha")]
        merge_sha_to_pr = {pr["merge_commit_sha"]: pr for pr in relevant_pulls}  # map merge commit SHA to PR for quick lookup

        merge_shas_on_main = main_sha_set.intersection(merge_sha_to_pr.keys())  # only merge SHAs that are actually on main

        # find all squashed PRs
        logger.info("There are %s PRs merged into main branch to analyze for squashing.", len(merge_shas_on_main))
        for merge_sha in merge_shas_on_main:
            pr = merge_sha_to_pr[merge_sha]

            commit_diff_shas = pr["commit_diff_shas"]
            if "-1" in commit_diff_shas:
                logger.info("PR %s contains commits with diff_sha of '-1'.
```

```
    Skipping entire PR...", merge_sha)
166                 continue
167
168             # Convert PR lists to sets once for fast ops
169             pr_commit_set = set(pr["commit_shas"])
170             pr_diff_set = set(pr["commit_diff_shas"])
171
172             if pr_commit_set <= main_sha_set:  # TODO: lookup
173                 # All PR commits are already on main -> normal merge commit
174                 logger.info("'%s' is a merged PR. -> Nothing to do.", merge_sha)
175                 continue
176
177             if pr_commit_set.isdisjoint(main_sha_set):
178                 # None of the PR commits are on main
179                 if pr_diff_set and pr_diff_set <= main_diff_sha_set:
180                     # All diffs are present -> rebased (or a squashed PR that had
    only one commit)
181                     logger.info("'%s' is a rebased PR (or a single-commit squash).
     -> Nothing to do.", merge_sha)
182                     continue
183
184                 # Squashed PR: remove squash merge commit, add original PR commits
185                 logger.info("'%s' is a squashed PR. -> Dropping squash commit and
    adding PR commits to main branch.", merge_sha)
186                 main_sha_set.discard(merge_sha)
187                 # Find the commit and count co-authors
188                 merge_commit = next((c for c in commits if c["sha"] == merge_sha),
     None)
189                 coauthors_count = len(merge_commit.get("co-authors", [])) if
    merge_commit else 0
190                 self.cnt_removed_commits += 1 + coauthors_count
191
192                 actual_added = pr_commit_set.intersection(commits_sha_set)
193                 main_sha_set.update(actual_added)
194                 # Count co-authors for each added commit
195                 for sha in actual_added:
196                     commit = next((c for c in commits if c["sha"] == sha), None)
197                     coauthors_count = len(commit.get("co-authors", [])) if commit
    else 0
198                     self.cnt_added_commits += 1 + coauthors_count
199                 continue
200
201             # Some, but not all of PRs "commit_shas" are in main branch
202             # TODO: we need to check if this can safely be ignored
203             logger.info("'%s' is an inconsistent PR. It will be ignored (for now)
    .", merge_sha)
204
205         # filter commit list to those in the final list of commit SHAs
206         return [c for c in commits if c["sha"] in main_sha_set]
207
208     def get_filtered_commit_data(self, commits: list[dict]) -> list[dict]:
209         """Filter out duplicate commits and those without changes."""
210         seen_diff_shas = set()
211         unique_commits = []
212         for c in commits:
213             diff_sha = c.get("diff_sha", "0")
214             if diff_sha == "0":
215                 logger.info("Dropping commit '%s' as it has no changes.", c["sha
    "])
216                 self.cnt_removed_commits += 1
217                 continue
218             if diff_sha in seen_diff_shas:  # skip duplicate
219                 logger.info("Dropping commit '%s' as its diff_sha is a duplicate
```

```
      ", c["sha"])
220                 self.cnt_removed_commits += 1
221                 continue
222             seen_diff_shas.add(diff_sha)
223             unique_commits.append(c)
224         return unique_commits
225 
226     def get_commit_info_per_identity(self, commits: list[dict]) -> dict[list]:
227         """Get the number of commits per identity for a specific repository."""
228         commits_per_identity = {}
229         for commit in commits:
230             try:
231                 author = commit["author_name"]
232             except KeyError:
233                 logger.warning("Dropping commit '%s' as it does not have the field
       'author_name'.", commit["sha"])
234                 continue
235             new_entry = {
236                 "sha": commit["sha"],
237                 "commit_date": commit["AuthorDate"],
238                 "_parsed_date": datetime.datetime.fromisoformat(commit["AuthorDate
      "]),
239                 "tzinfo": commit["AuthorDate_tz"],
240                 "repo": commit["repository"],
241                 "owner": commit["owner"],
242                 # TODO: differentiate between adding / deleting / modifying
243                 "loc": commit["added"] + commit["removed"],
244             }
245             commits_per_identity[author] = [*commits_per_identity.get(author, []),
       new_entry]
246             coauthors = commit.get("co-authors", [])
247             if coauthors:
248                 for coauthor in coauthors:
249                     commits_per_identity[coauthor] = [
250                         *commits_per_identity.get(coauthor, []),
251                         {**new_entry, "co-author-duplicate_from": author},
252                     ]
253         return commits_per_identity
254 
255     def add_hours_to_previous_commit(self, data: dict) -> dict:
256         """Add the number of hours to the previous commit of the same author id
      ."""
257         processed_commits = 0
258         time_thresholds = self.wt_cfg["time_thresholds"]
259         regular_max = time_thresholds["regular_max"]
260         subgroup_1_max = time_thresholds["subgroup_1_max"]
261         subgroup_2_max = time_thresholds["subgroup_2_max"]
262 
263         for user_commits in data.values():
264 
265             # Sort user_commits by parsed timestamps for better performance
266             user_commits.sort(key=lambda x: x["_parsed_date"])
267 
268             # Initialize the first commit
269             user_commits[0]["time_difference"] = -1.0
270             user_commits[0]["group"] = "Irregular (no previous commit)"
271             processed_commits += 1
272 
273             # Calculate time differences for subsequent commits
274             for i in range(1, len(user_commits)):
275                 prev_timestamp = user_commits[i - 1]["_parsed_date"]
276                 curr_timestamp = user_commits[i]["_parsed_date"]
277                 mins_diff = float((curr_timestamp - prev_timestamp).total_seconds
```

```
() / 60)
                user_commits[i]["time_difference"] = mins_diff
                processed_commits += 1
                if processed_commits % 1000 == 0:
                    logger.info("Processed commits: %d", processed_commits)
                # Assign group and sub-group based on time difference
                if mins_diff <= regular_max:
                    user_commits[i]["group"] = f"Regular (up to {regular_max / 60}
 hours)"
                    if mins_diff < subgroup_1_max:
                        user_commits[i]["sub-group"] = f"1 (less than {
subgroup_1_max / 60} hours)"
                    elif mins_diff <= subgroup_2_max:
                        user_commits[i]["sub-group"] = f"2 (at least {
subgroup_1_max / 60} and up to {subgroup_2_max / 60} hours)"
                    else:
                        user_commits[i]["sub-group"] = f"3 (more than {
subgroup_2_max / 60} and up to {regular_max / 60} hours)"
                else:
                    user_commits[i]["group"] = f"Irregular (more than {regular_max
 / 60} hours)"
        return data

    def add_num_of_commits_to_previous_commit(self, data: dict) -> dict:
        """Add the number of commits that happened timewise between two commits of
 the same author id."""
        # Create a single sorted list of ALL commits with their timestamps
        all_commits_with_timestamps = []
        all_commits_with_timestamps.extend(commit["_parsed_date"] for user_commits
 in data.values() for commit in user_commits)
        all_commits_with_timestamps.sort()

        # Use binary search for each author's commits
        processed_commits = 0
        for user_commits in data.values():
            for i in range(1, len(user_commits)):
                processed_commits += 1
                if processed_commits % 1000 == 0:
                    logger.info("Processed commits: %d", processed_commits)
                prev_timestamp = user_commits[i - 1]["_parsed_date"]
                curr_timestamp = user_commits[i]["_parsed_date"]

                # Use binary search to find commits in the time window
                start_idx = bisect.bisect_right(all_commits_with_timestamps,
prev_timestamp)
                end_idx = bisect.bisect_left(all_commits_with_timestamps,
curr_timestamp)
                user_commits[i]["num_commits"] = end_idx - start_idx

        return data

    def calculate_costs_regular(self, data: dict) -> dict:
        """Calculate costs based on the refined data."""
        # values for "Regular" "2"
        working_day = self.wt_cfg["working_day_minutes"]
        total_commits = sum(len(user_commits) for user_commits in data.values())
        night_start = datetime.datetime.strptime(self.wt_cfg["night_hours"]["start
"], "%H:%M").time()
        night_end = datetime.datetime.strptime(self.wt_cfg["night_hours"]["end"],
"%H:%M").time()
        start_min = night_start.hour * 60 + night_start.minute
        end_min = night_end.hour * 60 + night_end.minute
        night_mins = end_min - start_min if start_min <= end_min else (24 * 60 -
```

```
    start_min) + end_min
329
330         processed_commits = 0
331         for user_commits in data.values():
332             for commit in user_commits:
333                 processed_commits += 1
334                 if processed_commits % 1000 == 0:
335                     logger.info("Processed commits: %d", processed_commits)
336
337                 wt = 0.0
338                 time_difference = commit["time_difference"]
339
340                 if commit["group"].startswith("Regular"):
341                     match commit["sub-group"][0]:
342                         case "1":
343                             wt = float(time_difference)
344                         case "2":
345                             nightshare, daycoverage = self.
    calc_nightshare_daycoverage(commit, start_min, end_min, night_mins)
346                             commit["night_share"] = nightshare
347                             wt = nightshare * ((commit["num_commits"] /
    total_commits) * working_day) + (1 - nightshare) * daycoverage
348                         case "3":
349                             wt = float((commit["num_commits"] / total_commits) *
    time_difference)
350
351                 commit["working_time"] = wt
352                 commit["working_time"] = self.cap_working_time(commit)
353
354         return data
355
356     def calc_nightshare_daycoverage(self, commit: dict, start_min: int, end_min:
    int, night_mins: int) -> tuple:
357         """Calculate nightshare and daycoverage for a given commit using the
    commit's timezone."""
358         # correct the timezone of the commit with the stored tzinfo
359         commit_dt = commit["_parsed_date"].astimezone(datetime.datetime.strptime(
    commit["tzinfo"], "%z").tzinfo)
360
361         # compute the interval covered by the commit (in the commit's timezone)
362         start_time = commit_dt - datetime.timedelta(minutes=commit["
    time_difference"])
363         end_time = commit_dt
364
365         nightcoverage = 0
366         daycoverage = 0
367
368         # iterate minute-by-minute between start_time and end_time (both are
    timezone-aware)
369         current = start_time
370         while current <= end_time:
371             current_min = current.hour * 60 + current.minute
372             in_night = start_min <= current_min < end_min if start_min <= end_min
    else current_min >= start_min or current_min < end_min
373             if in_night:
374                 nightcoverage += 1
375             else:
376                 daycoverage += 1
377             current += datetime.timedelta(minutes=1)
378
379         nightshare = (nightcoverage / night_mins) if night_mins > 0 else 0
380         return nightshare, daycoverage
381
```

```
382     def perform_linear_regression(self, x_values: list[float], y_values: list[
        float]) -> tuple:
383         """Perform linear regression and return slope and intercept."""
384         x = np.array(x_values).reshape(-1, 1)
385         y = np.array(y_values)
386         model = LinearRegression()
387         model.fit(x, y)
388         slope = model.coef_[0]
389         intercept = model.intercept_
390         if self.wt_cfg.get("save_regression_plot", False):
391             save_regression_plot(x_values, y_values, slope, intercept, self.wt_cfg
        ["save_regression_plot_path"])
392         return slope, intercept
393 
394     def calculate_loc_to_time_regression(self, data: dict) -> tuple:
395         """Calculate the LOC to time regression parameters."""
396         # Prepare data for regression analysis
397         loc_values = []
398         time_values = []
399         for user_commits in data.values():
400             for commit in user_commits:
401                 if not commit["group"].startswith("Irregular"):
402                     loc_values.append(commit["loc"])
403                     time_values.append(commit["working_time"])
404 
405         # remove loc that are more than 4 standard deviations away from the mean
406         loc_array = np.array(loc_values)
407         mean_loc = np.mean(loc_array)
408         std_loc = np.std(loc_array)
409         filtered_loc_values = []
410         filtered_time_values = []
411         for loc, time in zip(loc_values, time_values, strict=True):
412             if abs(loc - mean_loc) <= 4 * std_loc:
413                 filtered_loc_values.append(loc)
414                 filtered_time_values.append(time)
415             else:
416                 logger.info("Removing outlier LOC value: %d", loc)
417 
418         # Perform linear regression to find slope and intercept
419         slope, intercept = self.perform_linear_regression(filtered_loc_values,
        filtered_time_values)
420         logger.info("Calculated LOC to time regression: slope = %.4f, intercept =
        %.4f", slope, intercept)
421         return slope, intercept
422 
423     def calculate_costs_irregular(self, data: dict) -> dict:
424         """Calculate costs for irregular commits."""
425         slope, intercept = self.calculate_loc_to_time_regression(data)
426 
427         for user_commits in data.values():
428             for commit in user_commits:
429                 if commit["group"].startswith("Irregular"):
430                     commit["working_time"] = slope * commit["loc"] + intercept
431                     commit["working_time"] = self.cap_working_time(commit)
432 
433         return data
434 
435     def reevaluate_coauthored_commits(self, data: dict) -> dict:
436         """Reevaluate working times for co-authored commits.
437 
438         For all co-authored commits duplicates, compare calculated working times
        with the original ones.
439         Assign the lower one to the one owned by the co-author.
```

```
440         """
441         coauthor_share = self.wt_cfg["coauthor_share"]
442         for user, user_commits in data.items():
443             for commit in user_commits:
444                 if "co-author-duplicate_from" in commit:
445                     main_a = commit["co-author-duplicate_from"]
446                     main_a_commit_sha = commit["sha"]
447                     main_a_commits = data.get(main_a, [])
448                     main_a_commit = next((c for c in main_a_commits if c["sha"] ==
     main_a_commit_sha), None)
449                     if main_a_commit:
450                         co_a_wt = commit["working_time"]
451                         main_a_wt = main_a_commit["working_time"]
452                         adjusted_wt = min(co_a_wt, main_a_wt) * coauthor_share
453                         if co_a_wt != adjusted_wt:
454                             logger.info(
455                                 "Adjusted working time for co-authored commit '%s'
     with main-author '%s' for co-author '%s' from %s to %s.",
456                                 main_a_commit_sha,
457                                 main_a,
458                                 user,
459                                 co_a_wt,
460                                 adjusted_wt,
461                             )
462                             commit["working_time"] = adjusted_wt
463                     else:
464                         logger.warning(
465                             "Could not find main author '%s' commit '%s' for co-
    authored commit '%s'.",
466                             main_a,
467                             main_a_commit_sha,
468                             commit["sha"],
469                         )
470         return data
471
472     def cap_working_time(self, commit: dict) -> float:
473         """Cap working time at configured percentage of working day per elapsed
    day since last commit."""
474         # get commit data
475         calculated_wt = commit["working_time"]
476         time_difference = commit["time_difference"]
477         if time_difference < 0:
478             return calculated_wt
479         # get config data
480         working_day = self.wt_cfg["working_day_minutes"]
481         max_worktime_multiplier = self.wt_cfg["max_worktime_multiplier"]
482         minimum_elapsed_days = self.wt_cfg["minimum_elapsed_days"]
483         # calculate maximum allowed working time
484         elapsed_days = max(minimum_elapsed_days, time_difference / (24 * 60))
485         max_wt = working_day * (1 + max_worktime_multiplier) * elapsed_days
486         if calculated_wt > max_wt:
487             # log the capping event
488             logger.info("Capping working time of %s from %s to %s.", commit["sha
    "], calculated_wt, max_wt)
489             self.cap_events.append(
490                 {
491                     "sha": commit["sha"],
492                     "date": commit["commit_date"],
493                     "group": commit["group"],
494                     "sub_group": commit.get("sub-group", ""),
495                     "time_diff_min": float(time_difference),
496                     "elapsed_days": float(elapsed_days),
497                     "calculated_wt_min": float(calculated_wt),
```

```
                    "new_wt_min": float(max_wt),
                    "over_by_min": float(calculated_wt - max_wt),
                },
            )
            return max_wt
        return calculated_wt
```

F Export Interface Outputs

```
Used Algorithm: BAlgo
Repository: https://github.com/porsche-design-system/porsche-design-system
Data last updated: 24.01.2026 15:00 CET
Output: First 15 records sorted by "contribution_time" descending

+--------------------+--------------------+-----------------+
|               owner|        project_name|contribution_time|
+--------------------+--------------------+-----------------+
|0ce7f3beacbc6a752...|porsche-design-sy...|        271384.28|
|51a6ffd5e52e7b604...|porsche-design-sy...|        154283.42|
|3e9a4fd531e4310df...|porsche-design-sy...|        133722.68|
|e32eae38d28275733...|porsche-design-sy...|        125365.16|
|353331c587113614f...|porsche-design-sy...|        123514.59|
|ac543d94d97e11059...|porsche-design-sy...|        103333.35|
|26dfd64e8c77cdb28...|porsche-design-sy...|        100989.66|
|47d241638fb7d6905...|porsche-design-sy...|         97591.46|
|44dcff791c41752cc...|porsche-design-sy...|         66028.35|
|7279c4a8728277d9c...|porsche-design-sy...|         55105.12|
|2cd88bc58b3e4fd92...|porsche-design-sy...|         30757.83|
|d945f005a919e093e...|porsche-design-sy...|         30677.21|
|22792341b2faabb56...|porsche-design-sy...|         29776.11|
|061df89b81ca8e58e...|porsche-design-sy...|         28317.11|
|b85297ec93491f73b...|porsche-design-sy...|         17846.00|
+--------------------+--------------------+-----------------+
```

```
Used Algorithm: IAlgo
Repository: https://github.com/porsche-design-system/porsche-design-system
Data last updated: 24.01.2026 15:00 CET
Output: First 15 records sorted by "contribution_time" descending

+--------------------+--------------------+-----------------+
|               owner|        project_name|contribution_time|
+--------------------+--------------------+-----------------+
|0ce7f3beacbc6a752...|porsche-design-sy...|        225785.53|
|51a6ffd5e52e7b604...|porsche-design-sy...|        140101.35|
|e32eae38d28275733...|porsche-design-sy...|        116483.87|
|3e9a4fd531e4310df...|porsche-design-sy...|        111998.06|
|353331c587113614f...|porsche-design-sy...|        100331.13|
|26dfd64e8c77cdb28...|porsche-design-sy...|         91072.46|
|ac543d94d97e11059...|porsche-design-sy...|         80819.22|
|47d241638fb7d6905...|porsche-design-sy...|         73139.60|
|44dcff791c41752cc...|porsche-design-sy...|         60786.42|
|7279c4a8728277d9c...|porsche-design-sy...|         39564.70|
|22792341b2faabb56...|porsche-design-sy...|         24977.80|
|b85297ec93491f73b...|porsche-design-sy...|         16193.60|
|7fb7c2fddf3807702...|porsche-design-sy...|         14652.02|
|6e72de0533e59b903...|porsche-design-sy...|         12978.42|
|bfb8f33d0011034af...|porsche-design-sy...|         10921.62|
+--------------------+--------------------+-----------------+
```

```
Used Algorithm: BAlgo
Repository: https://github.com/actions/runner
Data last updated: 24.01.2026 15:00 CET
Output: First 15 records sorted by "contribution_time" descending

+---------------+--------------+-----------------+
|          owner|  project_name|contribution_time|
+---------------+--------------+-----------------+
|443a0dabb130...|actions/runner|          23063.3|
|001daab0e8e6...|actions/runner|          8741.29|
|0bdb2fe20443...|actions/runner|          6245.34|
|c3d922226e8a...|actions/runner|          5491.09|
|ad57a7bbb75d...|actions/runner|          4770.13|
|60e1ab0ae006...|actions/runner|          2778.80|
|c5655c4037c1...|actions/runner|          2211.17|
|e916ed60f0fc...|actions/runner|          2137.20|
|77c6cf6bdf56...|actions/runner|          1962.52|
|5d5ef79f8e0c...|actions/runner|          1919.49|
|b45410e916dc...|actions/runner|          1684.33|
|eb558b93d700...|actions/runner|          1435.90|
|3fc548c25ad9...|actions/runner|          1328.69|
|dcdd5aa9e924...|actions/runner|          1316.89|
|2daf50092332...|actions/runner|          1281.66|
+---------------+--------------+-----------------+
```

```
Used Algorithm: IAlgo
Repository: https://github.com/actions/runner
Data last updated: 24.01.2026 15:00 CET
Output: First 15 records sorted by "contribution_time" descending

+---------------+--------------+-----------------+
|          owner|  project_name|contribution_time|
+---------------+--------------+-----------------+
|443a0dabb130...|actions/runner|         16563.29|
|0bdb2fe20443...|actions/runner|          3871.04|
|ad57a7bbb75d...|actions/runner|          3281.42|
|8d39a881a928...|actions/runner|          3083.19|
|c3d922226e8a...|actions/runner|          3070.76|
|1e492ece5b42...|actions/runner|          2317.71|
|60e1ab0ae006...|actions/runner|          2025.33|
|77c6cf6bdf56...|actions/runner|          1925.20|
|e916ed60f0fc...|actions/runner|          1666.91|
|c5655c4037c1...|actions/runner|          1561.14|
|5d5ef79f8e0c...|actions/runner|          1256.26|
|b45410e916dc...|actions/runner|          1097.22|
|0f2f7b880d7b...|actions/runner|          1007.04|
|2daf50092332...|actions/runner|           914.94|
|eb558b93d700...|actions/runner|           913.57|
+---------------+--------------+-----------------+
```

References

Afroz, S., Feng, Z., Kimura, K., Trinkenreich, B., Steinmacher, I., & Sarma, A. (2025). Developer productivity with genai. *arXiv preprint arXiv:2510.24265*. https://doi.org/10.48550/arXiv.2510.24265

Amor, J. J., Robles, G., & Gonzalez-Barahona, J. M. (2006). Effort estimation by characterizing developer activity. *Proceedings of the 2006 international workshop on Economics driven software engineering research*, 3–6. https://doi.org/10.1145/1139113.1139116

Boehm CSSE. (2023). Cocomo ii. Retrieved January 9, 2026, from https://boehmcsse.org/tools/cocomo-ii/

Buchner, S., & Riehle, D. (2022). Calculating the costs of inner source collaboration by computing the time worked. *55th Hawaii International Conference on System Sciences*, 7466–7475. https://doi.org/10.24251/HICSS.2022.896

Buchner, S., & Riehle, D. (2023). The business impact of inner source and how to quantify it. *ACM Computing Surveys*, *56*(2), 1–27. https://doi.org/10.1145/3611648

Capiluppi, A., & Izquierdo-Cortázar, D. (2013). Effort estimation of floss projects: A study of the linux kernel. *Empirical Software Engineering*, *18*(1), 60–88. https://doi.org/10.1007/s10664-011-9191-7

Capraro, M., Dorner, M., & Riehle, D. (2018). The patch-flow method for measuring inner source collaboration. *Proceedings of the 15th International Conference on Mining Software Repositories*, 515–525. https://doi.org/10.1145/3196398.3196417

Clegg, D., & Barker, R. (1994). *Case method fast-track: A rad approach*. Addison-Wesley Longman Publishing Co., Inc.

Dinkelacker, J., Garg, P. K., Miller, R., & Nelson, D. (2002). Progressive open source. *Proceedings of the 24th international conference on software engineering*, 177–184. https://doi.org/10.1145/581339.581363

Edison, H., Carroll, N., Morgan, L., & Conboy, K. (2020). Inner source software development: Current thinking and an agenda for future research. *Journal of Systems and Software*, *163*, 110520. https://doi.org/10.1016/j.jss.2020.110520

Flint, S. W., Chauhan, J., & Dyer, R. (2021). Escaping the time pit: Pitfalls and guidelines for using time-based git data. *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 85–96. https://doi.org/10.1109/MSR52588.2021.00022

Forsgren, N., Storey, M.-A., Maddila, C., Zimmermann, T., Houck, B., & Butler, J. (2021). The space of developer productivity: There's more to it than you think. *Queue*, *19*(1), 20–48. https://doi.org/10.1145/3454122.3454124

Hirsch, J., & Riehle, D. (2022). Management accounting concepts for inner source software engineering. *International Conference on Software Business*, 101–116. https://doi.org/10.1007/978-3-031-20706-8_7

Iso/iec/ieee international standard - systems and software engineering–vocabulary. (2017). *ISO/IEC/IEEE 24765:2017(E)*, 1–541. https://doi.org/10.1109/IEEESTD.2017.8016712

Jørgensen, M. (2014). What we do and don't know about software development effort estimation. *IEEE software*, *31*(2), 37–40. https://doi.org/10.1109/MS.2014.49

Jorgensen, M., & Shepperd, M. (2007). A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering*, *33*(1), 33–53. https://doi.org/10.1109/TSE.2007.256943

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., & Damian, D. (2014). The promises and perils of mining github. *Proceedings of the 11th working conference on mining software repositories*, 92–101. https://doi.org/10.1145/2597073.2597074

Petersen, K. (2011). Measuring and predicting software productivity: A systematic map and review. *Information and Software Technology*, *53*(4), 317–343. https://doi.org/10.1016/j.infsof.2010.12.001

Pitkevics, G., Dubey, L., Choa, C. H., Koleva, Y., Yilmaz, M., Clarke, P. M., & McCarren, A. (2025). Inner source, outer source, low code, and no code: Pros, cons and contexts-results from an mlr. *European Conference on Software Process Improvement*, 124–142. https://doi.org/10.1007/978-3-032-04288-0_8

Robles, G., Capiluppi, A., Gonzalez-Barahona, J. M., Lundell, B., & Gamalielsson, J. (2022). Development effort estimation in free/open source software from activity in version control systems. *Empirical Software Engineering*, *27*(6), 135. https://doi.org/10.1007/s10664-022-10166-x

Robles, G., González-Barahona, J. M., Cervigón, C., Capiluppi, A., & Izquierdo-Cortázar, D. (2014). Estimating development effort in free/open source software projects by mining software repositories: A case study of openstack. *Proceedings of the 11th Working Conference on Mining Software Repositories*, 222–231. https://doi.org/10.1145/2597073.2597107

Stol, K.-J., Schaarschmidt, M., & Morgan, L. (2024). Does adopting inner source increase job satisfaction? a social capital perspective using a mixed-methods

approach. *The Journal of Strategic Information Systems*, *33*(1), 101819. https://doi.org/10.1016/J.JSIS.2024.101819

Symons, C. R. (2002). Function point analysis: Difficulties and improvements. *IEEE transactions on software engineering*, *14*(1), 2–11. https://doi.org/10.1109/32.4618

Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., & Vasilescu, B. (2017). The impact of continuous integration on other software development practices: A large-scale empirical study. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 60–71. https://doi.org/10.1109/ASE.2017.8115619