Predicting Inner Source Success: A Machine Learning Approach Utilising Open Source Proxy Data

Master's Thesis

for the degree of

Master of Science (M.Sc.)

Data Sience

at the Faculty of Sciences of
Friedrich-Alexander-Universität Erlangen-Nürnberg

submitted on 08 May 2026

by **Julian Maibach**

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

Friedrich-Alexander-Universität
Naturwissenschaftliche Fakultät

Contents

1 Introduction **1**

1.1 Motivation and Context 1

1.2 Predictive Modelling in Software Engineering 5

1.3 Research Objectives and Questions 6

1.4 Methodology 7

1.5 Thesis Structure 7

2 Problem Definition **9**

2.1 The Research Gap in Predictive Analytics 9

2.2 Core and Peripheral Contributor Dynamics 9

2.3 Measuring Software Development Success 10

2.4 The MECOIS Platform 12

3 Objective Definition **13**

4 Solution Design **16**

4.1 Open Source Proxy Strategy 16

4.2 Feature Engineering: The Independent Variables (X) 17

4.3 Target Variable: Defining Success (y) 18

4.4 The Machine Learning Architecture 19

4.5 System Architecture 21

5 Implementation **30**

5.1 The MECOIS Baseline Architecture 30

5.2 Development Environment and Technology Stack 30

5.3 Extending the MECOIS Data Pipeline 31

5.4 The Temporal Window Builder Algorithm 34

5.5 Realising the Machine Learning Pipelines 34

5.6 Database and UI Integration 36

5.7 Practical Challenges and Pragmatic Solutions 38

6 Demonstration **40**

7 Evaluation **44**
7.1 Objectives and Questions 44
7.2 Design and Data 45
7.3 Pipeline and Data Quality 46
7.4 Metric and Feature Evaluation 48
7.5 Machine Learning Model Evaluation 50
7.6 Dashboard and Usability 54
7.7 Threats to Validity 55

8 Conclusion **56**

A Appendix **60**
A.1 Solution Design 60
A.2 Demonstration 62
A.3 Evaluation 63

Bibliography **69**

Acronyms and Abbreviations **70**

Declaration

I hereby certify that I have written this thesis independently and that I have not used any sources or aids other than those indicated, that all passages of the work that have been taken over verbatim or in spirit from other sources have been marked as such, and that the work has not yet been submitted to any examination authority in the same or a similar form.

Erlangen, 08 May 2026 my signature

Abstract

Inner Source promises to improve collaboration and code reuse inside organisations, but many initiatives still struggle because managers lack predictive, data-driven support. Existing platforms, such as MECOIS, mainly offer descriptive views of past activity and do not help answer whether a repository is ready for Inner Source or how adoption is likely to develop. This thesis addresses that gap by designing, implementing, and evaluating a predictive module for the MECOIS analytics platform. The module uses development data from corporate Open Source projects on GitHub as a practical proxy for Inner Source environments. The thesis defines a collaboration-based success measure that captures the extent of integrated work contributed by developers outside the core team. Several Machine Learning models are applied to estimate the likelihood that a repository will reach a highly collaborative state and to identify the structural factors most strongly associated with success. The resulting predictions and explanations are embedded in an interactive MECOIS dashboard that allows managers to explore different 'what-if' scenarios. Although available Open Source data limits the evaluation and cannot capture all organisational and cultural influences, the work shows that predictive analytics for Inner Source is feasible and provides a foundation for more data-informed decisions about when and how to open internal repositories.

1 Introduction

1.1 Motivation and Context

Modern software organisations increasingly depend on open collaboration, yet many still operate with tools and structures designed for closed development. This shift can be understood in the context of how software development evolved from the 1950s onwards. At that time, software development was highly specialised and hardware-centric, with small teams typically working on isolated, ad-hoc projects to make early computers function, rather than following formal requirements specifications [1]. As software evolved into a commercially valuable, standalone product by the 1970s, companies increasingly treated their source code as proprietary intellectual property, shifting towards more closed development models [2]. In direct response to this isolation, the 1980s saw the birth of the Free Software movement, catalysed by the launch of the GNU Project in 1983 [3]. This ideological push to share code laid the groundwork for a shift in 1998, when the term Open Source (OS) was coined. The new label presented decentralised, global collaboration as an alternative to earlier, more isolated development models [2, 4].
Eric Raymond [4] uses the metaphor of 'cathedrals' and 'bazaars' to compare traditional software development with Open Source Software (OSS). Traditional development is characterised by a top-down hierarchy, strict access controls, and highly partitioned knowledge, in which individual developers work in isolation and release software only when it is polished. In contrast, OSS is compared to a collaborative, open bazaar where participants openly exchange agendas and approaches.
The transition from isolated, guarded development to the collaborative philosophy of OSS has led to an IT landscape in which companies must decide how to balance traditional software development with OSS approaches. Organisations may introduce OS practices to benefit from external contributors' insights and increase engagement with potential and long-term users, though this approach risks exposing sensitive company information. Alternatively, companies may choose traditional in-house software development to protect proprietary knowledge, but this limits opportunities for information exchange and code reuse. The legacy of such restrictive practices persists in many organisations, for example, in code ownership rules that prevent reuse across business units.
To improve collaboration and knowledge exchange, Inner Source offers a solution that integrates the advantages of both traditional and OSS development within an organisational framework. Inner Source is the adaptation of OS principles and practices within organisations with proprietary guidelines, as Cooper and Stol [5] note. The primary objective is to facilitate code sharing across teams, thereby accelerating feature development. However, cultural and infrastructural challenges may hinder the adoption of Inner Source [5].

In large organisations, independent software development departments are often described as 'silos', since they are grouped into units but remain isolated from other departments. This isolation leads to disadvantages such as increased effort due to re-

dundant project implementation. These downsides can be reduced by making code universally accessible within the organisation. Internal access to code repositories enables what Capraro et al. [6] describe as the 'patch-flow', which represents the movement of code contributions across organisational boundaries, including projects, units, or profit centres.

Chen et al. emphasise that improving internal code reuse requires organisations to make reusable assets and their ownership easily discoverable. Effective communication channels further support the practical implementation of Inner Source. Standardised documentation, well-structured repositories, and visibility of reusable code make it easier to migrate to an Inner Source approach. These approaches prevent duplication of effort and enable innovative solutions to build on existing work [7].

The success of software engineering projects is closely linked to developer satisfaction and motivation. In traditional environments, both repositories and developers may stagnate for various reasons. Stol et al. [8] note that developer expertise can reduce participation, thereby slowing progress on long-term projects as familiarity increases. Additionally, participation and engagement are positively correlated, while longer tenure is negatively associated with these metrics. Developer archetypes such as 'topic experts,' who seek to demonstrate proficiency to a broader audience, and 'code junkies/ninjas,' who are motivated by intellectual challenges and skill acquisition, can be identified.
Inner Source introduces a dynamic that grants developers greater autonomy, enabling each to contribute in ways that align with their preferences, as shown in [5]. The findings of [5] and [8] suggest that Inner Source can increase job satisfaction and support more sustained engagement.

Although the theoretical process of adopting Inner Source appears convincing, transitioning from a traditional 'cathedral' development environment to an Inner Source model involves complexities and potential pitfalls. Stol et al. underline that while general software engineering risks persist, Inner Source introduces additional organisational and cultural challenges [9].
Case studies compiled by Cooper and Stol [5] identify several failure modalities, or 'anti-patterns', that hinder the success of Inner Source initiatives and are introduced below. An ongoing issue is that some organisations treat Inner Source adoption as a rigid, fixed method and overlook the need to tailor it to the existing corporate culture. Even if the Inner Source initiative is well-designed, it can stall due to its proprietary code and the slow hierarchies common to large organisations. Resistance can thereby arise from key executives or middle managers who raise concerns about ownership, control, and profit from developed software. Given that funding targets the right areas, it is a vital catalyst for a long-lasting Inner Source initiative. Many organisations, however, underestimate the importance of continuously and adequately funding the initiative so that developers' intrinsic motivation is not the only driver. Due to its experimental nature, Inner Source relies more on active guidance and mentorship than on traditional corporate bureaucratic processes. What Cooper and Stol [5] further describe as 'executive air cover' is the explicit and sustained leadership support that ensures an effective and durable initiative; the absence of this can lead to a pre-

mature exit. It is also important to have a consistent definition of Inner Source. If managers mistakenly label non-collaborative, traditional projects as Inner Source, this can confuse employees and slow the company's cultural transformation. The principle of 'Community Before Code' emphasises the importance of vibrant, healthy communities in producing innovative code. Prioritising short-term feature delivery over community well-being increases the risk of project failure. Flat hierarchies and easy collaboration, however, help prevent project stagnation, even when key members leave.
The framework presented in [9] (see Figure 1) for adopting Inner Source, encloses nine key factors across the categories of software product, practices and tools, and organisation and community. These nine factors aim to limit the impact of existing anti-patterns and provide a baseline of what an Inner Source adoption process requires. They further note that split code ownership and modular coding practices help design software that is more attractive to potential new Inner Source users. Further, clear guidelines for best practices and a development environment that supports community participation in development and quality assurance are vital to facilitating Inner Source adoption. Lastly, they mention that social practices such as transparent, coordinated communication should not be overlooked.

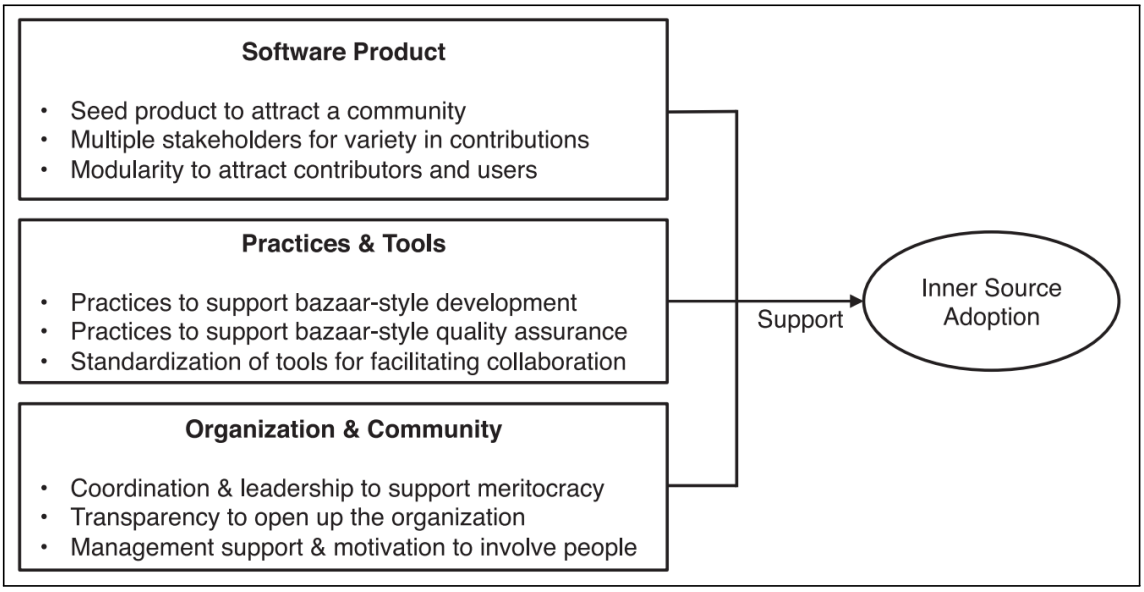

Figure 1: Key Factors for Adopting Inner Source: Factors that support Inner Source adoption across different aspects of software engineering [9, p. 18:9].

The presence of the aforementioned anti-patterns, common in large organisations, complicates the prediction of outcomes for Inner Source initiatives. Capraro [10] argues that simply counting developer contributions fails to capture the full dynamics of a project, as it overlooks important social factors. Consequently, accurately measuring success is challenging. The absence of advanced metrics leaves management without a clear understanding of project status, making early intervention in cases of community decline unlikely.

Within the context of OS, the onion model depicts a layered, hierarchical, and so-

cial structure that characterises a project's community. Drawing on empirical studies such as [11] and [12], this theory describes concentric circles that represent the overall influence of individuals within the project. Influence may derive from authority, responsibility, or the nature of one's contributions. In this circular hierarchy, layers farther from the centre play a less significant role, while those closer to the centre exert the greatest influence.

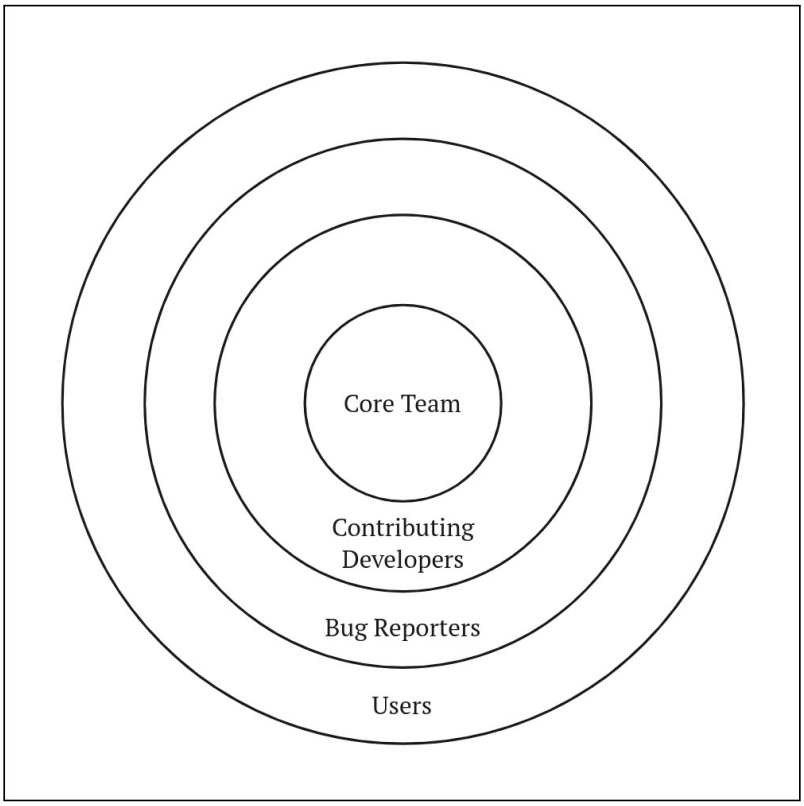

Figure 2: The onion model of an OS community: Application of the model in a sustainable software development community based on Aberdour [13, p. 59]

A central dynamic of the onion model is the progression of contributors toward the centre. This progression reflects a transfer of knowledge, as former users or sporadic contributors become increasingly invested in the project, learn from core contributors, and gain experience. As contributors advance through the onion's layers, they assume greater responsibility. For example, a project user may identify and fix bugs independently, learn project-specific details from feedback provided by experienced contributors, and, after several contributions, begin assisting others and taking on additional responsibilities.
In many OSS projects, only a few contributors form the inner circles, but are responsible for roughly 80% of total contributions. The majority of contributors are from the outer layers and produce the remaining 20% [13]. This distribution gives rise to the terms 'core' for the inner layers and 'non-core', which will be referred to as 'peripheral',

for the outer contributor layers. In this thesis, the onion model will be operationalised by classifying contributors into core and peripheral roles based on their cumulative contribution shares (see 4.1).

The gathered context indicates that adopting Inner Source successfully is difficult. Managers planning to introduce Inner Source often lack predictive, data-driven tools to assess the readiness of their teams and projects. Furthermore, existing metrics are frequently either naive or not tailored to Inner Source. In chapter 2, problems with this process are presented.

1.2 Predictive Modelling in Software Engineering

Researchers use Machine Learning (ML) to understand development trends, predict project evolution, and prevent potential failures in software development initiatives. Since several models will be discussed and used throughout this thesis, a short introduction on the fundamentals of well-established algorithmic approaches, particularly suited to analysing Version Control Systems (VCS) and collaboration data, will be provided initially.

Random Forest

Random Forest (RF) is an ensemble learning method that constructs multiple decision trees and combines their classifications. RF tends to be more robust to overfitting than single decision trees, because it employs several relatively simple trees, preventing any single tree from becoming too closely fitted to the training data [14]. Additionally, RF can handle complex nonlinear relationships and is suited for both classification and regression tasks. This versatility makes RF suitable for a wide range of predictive modelling applications, for example, predicting features derived from noisy and correlated repository data.

Logistic Regression

Hosmer Jr. et al. [15] describe Logistic Regression (LR) as a supervised ML algorithm that performs well in binary classification tasks. LR computes a linear combination of input variables and applies the sigmoid function to return a probability value between 0 and 1, making it suitable for binary classification. To address multicollinearity, regularisation techniques such as Lasso (L1) and Ridge (L2) are used. Lasso encourages sparsity by using the L1 norm, while Ridge, which employs the L2 norm, allows for coefficient shrinkage without setting feature weights to zero. Regularisation thus enhances interpretability and generalisation in binary classification problems by highlighting those coefficients that actually influence the classification decision [16].

Support Vector Machine

Support Vector Machine (SVM) is a learning algorithm capable of performing binary classification and regression tasks by identifying the optimal hyperplane that separates input data into classes. SVMs provide high accuracy predictions but are computationally intensive. The incorporation of soft margins, in addition to hard margins, allows the model to tolerate training errors and improves stability when predicting unseen data [17].

Autoregressive integrated moving average (ARIMA)

The Autoregressive Integrated Moving Average (ARIMA) model is a time-series forecasting algorithm that predicts future continuous values based on historical data. ARIMA integrates three components: The first part (autoregressive) relates past values to future values; The integrated part indicates the number of differences between each value and their previous value, required to achieve stationarity; and the Moving Average part incorporates a weighted sum of previous forecasting errors [18].

1.3 Research Objectives and Questions

The primary objective of this thesis is to design, construct, and evaluate a predictive artefact for users of the enterprise software development analytics framework MECOIS [19]. Further details about this platform are provided in section 2.4. The artefact will assess repository status and, using historical development data from GitHub, forecast the likelihood of successful future Inner Source adoption. The artefact is intended to help managers engage more proactively in project development through data-driven strategies rather than relying on reactive approaches. The thesis focuses primarily on the following two research questions:

RQ1: How can Inner Source success be defined and measured?

Training a ML model requires a clearly defined target variable. In this thesis, 'success' serves as the target and must be defined within the context of Inner Source. To achieve this, several approaches from the academic literature [6, 12, 20, 21, 22] will be analysed and compared to identify the most suitable method. The selected approach will provide a metric that serves as the ground-truth label for predictive ML models.

RQ2: To what extent can ML models predict the success of Inner Source initiatives based on historical collaboration data extracted from VCS?

The second research question examines the predictive quality of historical collaboration data. VCSs such as GitHub generate continuous data streams, including commit histories, pull request discussions, issue tracking logs, continuous integration results, and peer code review interactions.

This research aims to identify nonlinear patterns in these data sources to forecast project development accurately. As the artefact will function as a feature dashboard

within the MECOIS platform, it must operate using data available within the platform's scope. The artefact will be evaluated for practical utility by assessing its predictive accuracy and usability as a feature enhancement in a use-case scenario.

1.4 Methodology

To address the research questions and ensure both theoretical and practical correctness, this thesis adopts the Design Science Research (DSR) paradigm, which focuses on the production of innovative artefacts. In addition, it employs a data sourcing strategy that uses OSS as a proxy for Inner Source environments. The research follows the DSR methodology proposed by Peffers et al. [23], comprising six steps: problem identification, objective definition, design and development, demonstration, evaluation, and communication. The last step will be altered towards 'conclusion' to meet the thesis requirements. The artefact developed in this thesis is a feature enhancement to the MECOIS platform, specifically the Inner Source Prediction module. Subsequent chapters will detail the decision-making processes, limitations, and potential future extensions related to the artefact.

Due to the confidential nature of the company data required to implement a predictive ML solution, this research utilises VCS data from OSS projects as an empirical proxy. In the absence of access to internal organisational repositories, this approach provides a practical alternative for working with organisation-like data. Although a small-scale data-collection survey within a corporate development team was initially considered, it was abandoned due to anticipated low response rates, as evidenced by the 2024 InnerSource Commons survey [24], which even reached a significantly larger audience. Training and testing ML models require substantial data to capture nonlinear dependencies in repository data, making the OSS proxy a viable option for developing a scalable and reproducible artefact. However, using OSS as a proxy requires simulating an Inner Source repository or restricting the analysis to contributors from a specific organisation. Organisational and repository-specific practices, such as the use of bots or multiple contributor identities, may complicate data attribution. Although some findings from the OSS proxy, such as repository workflows and structural patterns, may generalise to proprietary Inner Source settings, other aspects, such as contributor motivation, internal hierarchies, and resource allocation, may not be observable. Consequently, the results of this thesis must be interpreted with these limitations in mind. Because OSS projects differ from proprietary Inner Source settings in contributor motivation and governance, the generalisability of the results is limited (see 7.7).

1.5 Thesis Structure

The remainder of this thesis will be organised into several sections, beginning with the Problem Definition, which introduces the main problems related to the thesis objectives, the thesis scope, and the key challenges expected along the way. Afterwards, the Objective Definition will deliver the concrete research objectives and the anticipated

contributions. The Solution Design should then describe the realisation of the artefact and the algorithms used. The Implementation chapter will dive into the technical aspects of the artefact, depicting the implementation of the data pipeline and the ML models. The Demonstration section will present the possibilities and the usage of the artefact. In the Evaluation chapter, the artefact will be analysed with respect to both data-scientific and methodological aspects. Lastly, the Conclusion will summarise and review the thesis, concluding with final thoughts on the process.

2 Problem Definition

2.1 The Research Gap in Predictive Analytics

While the software industry is practised at looking backwards using descriptive analytics, it is much more difficult to look forward using predictive analytics. Noisy data and the complex nature of human and organisational dynamics are the main reasons for those difficulties.

A widely used descriptive framework for OS communities is the Community Health Analytics Open Source Software (CHAOSS) metric framework by the Linux Foundation [25], which has found application in GrimoireLab [26] and other analytics platforms. In the Inner Source space, MECOIS, which will be introduced in this thesis, serves a comparable descriptive function. However, both frameworks are strictly diagnostic and descriptive: they lack predictive tooling to forecast a project's future state. Despite the growing work effort on Inner Source adoption and collaboration metrics, there is currently no predictive artefact that estimates Inner Source success from VCS data and embeds these insights into an operational dashboard such as MECOIS. Existing studies either provide descriptive analyses of contribution patterns [12, 27, 28] or define success metrics without integrating them into a decision-support tool for managers [20, 21, 22]. This thesis addresses this gap by building a predictive module that uses VCS data to forecast Inner Source adoption outcomes at the repository level.

Predicting human behaviour through software artefacts is challenging. Sources of scientific uncertainty must be acknowledged when dealing with such a topic. Zimmermann et al. [29] found that transferring prediction models between projects is not effective in the majority of cases. This implies that models trained on one project's codebase do not typically generalise to other projects unless the projects share similar organisational structures and cultures. In practice, this means that a RF model trained on a large, well-documented Java repository can tend to perform worse on a niche, poorly documented Python microservice.
Furthermore, the definition of a repository's success changes over its lifetime. When a repository is created, for example, most contributions are likely core contributions, and commit velocity is often high. Once the repository has stabilised, however, stability and review capacity become more important than continuous growth in commit volume. This increases the complexity of predicting human behaviour in software development.

2.2 Core and Peripheral Contributor Dynamics

As discussed in section 1.1, the onion model differentiates between contributors that exhibit different collaboration patterns. While the core developers provide architectural stability, peripheral contributors introduce scalability and domain-specific adaptations. Measuring success in Inner Source, therefore, requires separating those two groups and their contributions. Further, the importance of non-code contributions has

to be addressed.
What Pham et al. [30] and later Pinto et al. [28] describe as a 'drive-by' commit is a contribution pattern in which developers submit a few contributions or only one single pull request with minimal effort and some impact. After that, those contributors do not return to the repository. This behaviour and the findings of Zhou and Mockus [31], which state that willingness, opportunity, and capacity of newcomers highly influence the probability of becoming a long-term contributor, lead to the conclusion that an implementation of the core and peripheral contributor dynamics in Inner Source requires a threshold for the minimum amount of contributions to appear as a peripheral contributor.
Another problem that addresses the distribution of roles between core and peripheral contributors is the identity problem, or aliasing. Bird et al. [32] highlight the necessity of identity resolution in their work with the Apacke HTTP server project. They determine that, to further process user data, all aliases used by users in the data source must be unified into single, unique identifiers to avoid duplicate entries in the contributor list. Consequently, this poses an additional requirement for the thesis data pipeline to handle aliasing.
To close the discussion on implementing the onion model, the transitions that individuals make between layers must be considered. In the work of Robles et al. [27], the dynamic nature of core teams is depicted. In most OS projects, the developer flow interacting with it is not static but evolves. When measuring and categorising a contributor as core or peripheral, the user's state at the time of the contribution must be considered, even if it later changes. For this thesis, it is therefore necessary to introduce a method that does not treat role classification as a global variable, but instead uses a temporal snapshot for classification, considering only current and past contributions.

2.3 Measuring Software Development Success

The problem of measuring software development success has been a topic of discussion among computer scientists for decades. A core argument in this debate is that software development is not a simple mechanical assembly-line process [22]. Rather, it is a complex socio-technical activity that depends on the creativity and collaboration of many individuals [9]. Defining 'success' solely in terms of quantitative measures can therefore lead to skewed or flawed data. An important guideline for the latter statement was proposed by Strathern [33], namely Goodhart's Law. It states, 'When a measure becomes a target, it ceases to be a good measure.' [33, p. 4], which affects the way success should be defined in this thesis. An artificial influence on data, for example, an intended practice from developers to split pull requests to raise the raw number per repository, would make the metric unsuitable for training as it was targeted and, according to Goodhart's Law, is no longer a good measure. Artificially manipulating metrics to improve a project's rating not only makes the metric less important but also has negative consequences. Splitting pull requests, for example, would result in a more extensive review and introduce a bottleneck that would have been unnecessary.

In [34], Fenton and Bieman discuss the reliability of metrics such as Lines of Code (LOC) and raw commit counts. They conclude that LOC is a flawed indicator of a successful repository because it does not account for efficient, elegant coding; instead, it can penalise good coding practices and promote redundant or bloated source code, again illustrating the risks highlighted by Goodhart's Law.
The aforementioned socio-technical complexity of software development processes is highlighted by Sommerville [35], who emphasises the importance of human factors such as communication, domain understanding, and team spirit. This invisible work is hard to capture solely from quantitative repository data, as processes like mentoring and information exchange can be highly verbal.
To address these problems in defining a suitable metric for 'success', the focus of developer output measurement has to shift toward the flow and dynamics of collaboration. Forsgren et al. [22] argue that software performance should be assessed using throughput (e.g., deployment frequency or lead time for changes) and stability (e.g., change failure rate). Preferring this approach over raw developer output laid the ground for the DevOps Research and Assessment (DORA) framework [36], a frequently cited and widely applied method that reflects a recent consensus in both research and practice.
While DORA focuses on the dynamics of software delivery, Inner Source is fundamentally concerned with the flow of collaboration across organisational boundaries. Capraro [10] introduced the concept of patch-flow and argues that Inner Source initiatives are successful only when code modifications traverse organisational boundaries sustainably.
For both OS and Inner Source, a clear categorisation of contribution types is essential. Pinto et al. [28] emphasise that these communities rely heavily on non-coding activities, such as bug reporting, documentation, and code review. This indicates that a purely commit-based measurement is inadequate, as it disregards contributions that are not solely code-based. Moreover, the point at which changes are integrated into the project is where a peripheral contribution effectively becomes part of the codebase: a pull request constitutes a realised contribution only if the merge process is successful. Gousios et al. [37] even identify the pull request evaluation process as a primary bottleneck in distributed development.
In this thesis, success measures are required to be (i) resistant to obvious gaming, (ii) not based solely on simple quantitative attributes, and (iii) sensitive to collaboration patterns. In addition, the flow and stability of deployed changes indicate the effectiveness of development processes and should therefore be considered when selecting appropriate metrics. Finally, contributions should be classified according to the onion model; non-code contributions must also be accounted for, and successful integration should be defined as contributions that are actually merged into and used within the project.

2.4 The MECOIS Platform

The MECOIS platform serves as the foundational software environment for this research. It is designed to measure and analyse software development data, with a core focus on data engineering for repository analytics. Using Python, Apache Spark, and a medallion-style data lake architecture with bronze, silver, and gold data stages in the backend, MECOIS provides an orchestrator-driven data pipeline that extracts raw development data from various sources (e.g., GitHub, GitLab) and exposes it to the MECOIS frontend dashboard (TypeScript and React with Vite) via Supabase. Furthermore, it offers a SortingHat API, a GraphQL-based web service that uses GrimoireLab SortingHat [38], and a GraphQL wrapper deployable with Docker Compose. These components enable the MECOIS platform to manage contributor identities by merging all aliases for the same individual and assigning each a Unique User Identity (UUID). Additionally, user profiles can be enriched with information such as a bot flag (e.g., indicating that the user is a GitHub automation bot) and organisational affiliations that are matched via SortingHat, which uses email domain names to assign organisational enrolments to unique users.

While MECOIS provides comprehensive descriptive analytics and data processing, its current capabilities are retrospective in nature. They are not optimised for processing the large volumes of raw data typically required to train an ML model. The platform also lacks the domain-specific logic needed to forecast future collaboration patterns. Specifically, the baseline platform does not natively classify contributors into core and peripheral roles as required by the onion model, nor does it offer predictive modelling to assess a repository's readiness for Inner Source adoption. For a manager deciding whether to open a repository for cross-silo collaboration, MECOIS currently provides historical context but lacks predictive decision support.
Apart from the identity management functionality inherited from SortingHat, the MECOIS pipeline does not currently enrich user records with additional organisational affiliation information, such as that from GitHub profiles. It also lacks backend methods for filtering repository data by time windows, which would enable additional data engineering tasks based on temporally constrained datasets. Finally, the platform does not yet provide pipeline functionality to execute ML tasks such as regression or classification, nor does it include a dedicated dashboard for evaluating Inner Source adoption scenarios.

This problem definition focuses on collaboration signals derived from VCS and related artefacts (commits, pull requests, reviews). It does not consider qualitative factors such as individual motivation, team culture, or detailed organisational structures beyond what can be inferred from contribution patterns. Furthermore, the thesis does not attempt to model all aspects of Inner Source governance (e.g., funding processes, executive sponsorship). Instead, it treats these as contextual factors discussed in the background literature.

3 Objective Definition

Research Process

The overarching goal of this thesis is to design, implement, and evaluate a predictive decision-support module within the MECOIS framework. To bridge the gap between descriptive tracking and predictive forecasting of Inner Source initiatives, the research process is structured into the following sequential steps:

1. **Data Extraction and Pipeline Integration:** Extend the existing data architecture for large repository data for GitHub commits and pull requests, avoiding the reimplementation of the core orchestrator logic.
2. **Identity Resolution:** Enrich raw GitHub identities using the SortingHat and GitHub profile information to accurately map contributors to organisations, identify bots, and resolve aliased accounts.
3. **Role Classification:** Operationalise the onion model by automatically classifying organisational contributors into 'core' and 'peripheral' roles based on their cumulative contribution share until the inspected contribution.
4. **Temporal Structuring:** Solve the 'cold start' problem regarding the predictive nature of the ML endeavour by splitting repository history into a baseline period (window A) and a target period (window B).
5. **ML Modelling:** Train composable ML pipelines - specifically K-means Clustering, RF, LR with Lasso and Ridge regularisation, SVMs and ARIMA - on the baseline features to predict future collaborative behaviour.
6. **Dashboard Embedding:** Update the MECOIS frontend via Supabase with the new predictive scenarios, clustering archetypes and feature impacts to enable a manager using the dashboard to assess a repository's readiness for Inner Source adoption.

A central objective of this research is to establish a predictable target for the ML models without relying on vanity metrics (e.g., LOC). Success is defined as growth in Inner Source adoption, measured by the Peripheral Contribution Rate (PCR) introduced in 4.3.

Functional Requirements

The system must ingest repository data and generate enriched silver tables that contain verified organisational identities and bot flags. Based on this information, it must dynamically calculate and classify contributors into core and peripheral roles, excluding unknown identities to maintain data integrity. Furthermore, the system must compute

gold-level metrics, such as review density, time-to-merge, and funnel width, segmented by contributor role. These features serve as the basis for training predictive models that must produce regression coefficients and feature importances. Finally, the system must upload the resulting predictive outputs to Supabase to support the frontend dashboard components. In addition, the artefact should provide K-means-based repository archetypes to give managers an interpretable entry point into the predictions. Further, the dashboard should contain the possibility to visualise time-series forecasts. To improve the application of the onion model, an advanced identity resolution could be introduced.

Non-Functional Requirements

For maintainability and easier future adaptation, the ML components should be implemented as pipelines that could be replaced or extended in future work. Predictive outputs should be transparent; therefore, the system should surface RF feature importances and LR coefficients so that users can understand the main drivers of the predictions. In addition, the Inner Source module should reuse MECOIS's existing pipeline orchestration, Spark execution environment, and frontend routing without requiring core architectural rewrites. The execution of optional ML modules, such as clustering and ARIMA, should not block the use of the baseline descriptive dashboard when data is too sparse to train these models reliably.

Research Questions and Methods Mapping

RQ1 (How can Inner Source success be defined and measured?) is addressed through a review and synthesis of existing success metrics and thresholding approaches [6, 12, 20, 21, 22] and by defining the PCR and a data-driven success threshold.
RQ2 (To what extent can ML models predict the success of Inner Source initiatives based on historical collaboration data extracted from VCS?) is managed by extracting VCS and collaboration metrics, training ML (K-means, RF, LR, SVMs, ARIMA), and evaluating them using regression metrics (Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), R-squared (R^2)), classification metrics (Area Under the Curve (AUC), F1, accuracy, precision/recall), and forecast selection criteria (Akaike Information Criterion (AIC), Bayesian Information Criterion (BIC)). A detailed overview of the evaluation techniques will be given in 4.5.

Data Sources, Scope and Limitations

The analysis is restricted to GitHub commits and pull requests, specifically focusing on main-branch integrations and metadata for Pull Request creators, authors, committers, and other participants. Because strict policies frequently protect proprietary corporate Inner Source data, this thesis utilises corporate-sponsored OSS repositories as a quantitative proxy for organisational collaboration (see section 4.1). The metrics

calculated in the data pipeline won't measure corporate realities, such as top-down management mandates, developer sprint allocations, or intrinsic employee motivation. Further, the onion model, in this thesis, won't guarantee a complete mapping of aliases. Lastly, the temporal split of repositories into two time windows won't be applied to all repositories, since not all meet the requirements for training ML models.

4 Solution Design

4.1 Open Source Proxy Strategy

The challenge of predicting Inner Source success in the scenario presented by the MECOIS dashboard is a cold-start problem: enterprise repositories that have not yet adopted Inner Source lack historical collaboration data, while data from successful initiatives is locked behind corporate confidentiality policies. To address this issue, the artefact developed in this thesis utilises public OSS repositories as a behavioural proxy.
To find repositories with high OS engagement, the companies that own such projects must first be identified. The Open Source Contributor Index (OSCI) [39] highlights organisations with the most active OS contributors. For selected organisations from the top of this list, OSSInsight [40] provides information about the repositories they own on GitHub. In the following, repositories with high activity over the last twelve months were selected from several organisations, if the company information for the pull request creators shows a substantial number of users affiliated with the owning organisations. The resulting repository pool contained up to 5 years of data from 2020 to the end of 2025, from 31 repositories and 10 organisations.

To tailor the predictive model to the real-life scenario, the adoption of Inner Source must be simulated across the selected OSS repositories. Since the majority of those projects have a steady stream of peripheral contributions, the data preparation before the ML training process must filter out repositories that cannot simulate Inner Source adoption. This simulation is realised by repeatedly splitting the repository into two time windows: window A (12 months) and window B (6 months). The lengths for window A and window B were chosen to ensure that each window contains sufficient activity to compute stable collaboration metrics and avoid very short observation periods.
The segmented windows should mimic the pre- and post-adoption time frames of a simulated Inner Source adoption process. Window A is a 12-month period during which the repository is dominated by a small, closed group of maintainers, thereby simulating a traditional, siloed corporate repository. Window B is a subsequent period in which broader community (peripheral) participation is present. To enrich the training data sets, several windows are created per repository. Multiple windows per repository are treated as independent samples during model training. Since they may overlap in time, temporal dependence poses a possible threat to validity. Then, the developer community is dynamically mapped to the onion model. The highest-volume contributors are classified as the core team, while the remaining affiliated contributors represent the peripheral developers.

To enable Inner Source simulation, each user must have a valid organisation allocation. Since public GitHub data is noisy, a closed corporate environment needs a pipeline to eliminate unaffiliated actors and short-lived drive-by contributors, and to correctly identify bots.

This is achieved through an 'Identity Resolution Pipeline' that leverages SortingHat and the GitHub API. The system cross-references commit emails, GitHub profiles, and organisational enrolment data. Contributions are only retained if all participants can be definitively linked to the target organisation. All unknown identities are excluded from the metric calculation and, therefore, from the training of the ML model, to prevent artificial inflation of the peripheral contributor pool.

The presented method of using OSS repositories as a proxy for Inner Source proposes several advantages. First, the fundamental mechanics of distributed collaboration (asynchronous communication, pull request submission and code review bottlenecks) are largely similar across both environments. The core team's workload for reviewing peripheral contributions does not differ, whether the repository is public (on an organisation's intranet) or private.
By strictly enforcing the specified temporal boundary between the two time windows A and B, the extracted features are less prone to bias, thereby improving the credibility of the model trained on this data.
Among the available options for data retrieval to train a predictive ML model, the proxy method provides a practical baseline. Given the difficulty of extracting large-scale, cross-company proprietary data, filtering OSS down to verified corporate contributors isolates the closest available socio-technical behaviour pattern for an empirical data science approach. Even though the OSS proxy is considered a good approach given the scenario, it is worth noting that the artefact derived from this data basis might not generalise to low-activity or non-corporate projects, since the data sample is restricted to active, corporate-sponsored OSS repositories.

4.2 Feature Engineering: The Independent Variables (X)

To predict future success, ML models must learn from the repository's structural and collaborative state before adoption. Therefore, all independent variables, called features, are extracted exclusively from the pre-adoption (window A) baseline timeframe.

Selected Features

- **Team Structure Metrics:** Metrics like the 'total core contributors' or the 'size of the core team' capture the baseline capacity of the repository to handle future code reviews.
- **Review Dynamics:** 'Average comments per core pull request' and 'average hours to merge core pull request' measure the existing friction in the integration process. As previously discussed, a core team that takes weeks to merge its own internal pull requests will likely fail to sustain a peripheral community.
- **Visibility and Ecosystem:** The number of stars of a repository (the GitHub 'stargazer count') and the number of forks form the 'fork-to-star ratio', which repres-

ents the relationship between discoverability and the architectural modularity of the repository.

Excluded Features

- **Target Leakage Variables:** Any metric involving peripheral contribution rates from the post-adoption time window B is excluded from the ML training process.
- **Raw Identity Artefacts:** Developer usernames, UUIDs, and email domains are excluded to prevent the models from memorising specific repository-related patterns and therefore overfitting, rather than learning generalisable structural patterns.

4.3 Target Variable: Defining Success (y)

Applying the patch-flow method requires full access to an organisation's internal repositories and hierarchical structure and is therefore not suited to direct implementation within this thesis. Nevertheless, the underlying concept of Capraro et al's approach [6] can be adapted for this thesis. The success metric (see 2.3) must capture both the flow of external contributions and the efficiency with which they are integrated, while avoiding unreviewed or legacy contributions such as unmerged pull requests.
To train supervised models, the system requires a valid and operational definition of success. Relying on arbitrary thresholds or purely heuristic practices was therefore avoided.

Success is quantified using the PCR. Evaluated strictly within the target timeframe of window B, the metric calculates the proportion of integrated work that originates not from the core team but from the peripheral team and which, according to the case study by Mockus et al. [12], provides significant added value to the repository:

$$PCR = \frac{Peripheral\ Pull\ Requests\ +\ Peripheral\ Commits}{Total\ Pull\ Requests\ +\ Total\ Commits}$$

To translate this continuous ratio into a classification target where success is class 1, and failure is class 0, this artefact applies a data-driven discretisation approach. The threshold for success is dynamically set at the 75th percentile (Q3) of the PCR distribution across the training dataset. If a repository's window B PCR is in the top quartile among its peers, it is considered a success. The choice of the 75th percentile is pragmatic: it highlights the most collaborative quartile in this specific dataset, but it does not imply that repositories just below this threshold are failures. Rather, the success label should be interpreted as 'high collaboration relative to the studied repositories'. Varying the threshold and observing any effect on classification performance could assess robustness.

Those decisions were made based on the following academic software engineering methodologies:

In [20], Nam et al. establish that when using explicit qualitative labels, taking percentiles of a continuous software metric is a scientifically valid method to generate binary target variables. Alves et al. [21] highlight the importance of respecting the metric statistics when choosing thresholds. Decision boundaries must be derived directly from the statistical distribution of the analysed dataset to remain objective. Lastly, Forsgren et al. [22] found that, due to the accelerating dynamics of software delivery in the industry, benchmarking performance in a non-stationary rank relative to competing instances provides a comparison that adjusts over time and across environments.
A summary of these results shows that choosing the upper quartile (Q3) as the decision threshold for success classification ensures that ML models are optimised to identify repositories with a high level of cooperation and distinguish them from those with only an average level of cooperation. By deriving this threshold from the relative statistical distribution of competing initiatives, the classification adapts to contemporary events and the broader environment in which comparisons are made.

4.4 The Machine Learning Architecture

Instead of relying on a single algorithm that may not capture all the features, this thesis aims to provide an ensemble of models. Because managers using the dashboard are likely to have multifaceted questions, these models serve distinct, complementary use cases.

K-means Clustering (Unsupervised)

K-means Clustering serves as the entry point to the analysis within the prediction dashboard. Following the question *What type of repository am I looking at?*, the clustering groups features into distinct archetypes:

- **Fast Integrator:** Low core merge latency and strong review throughput.
- **Community Catalyst:** High peripheral participation with active review culture.
- **Governed Core:** Core-owned repository with slow, controlled change flow.
- **Visibility Magnet:** High external visibility and ecosystem attention.
- **Balanced Bridge:** Balanced core/peripheral collaboration across the repository.
- **Legacy Bottlenecks:** Large, highly visible project with slow collaboration flow.

Those archetypes cover all features available in the base data and provide a comparative context before exploring the more complex predictions. Assigning these archetypes to the clusters resulting from the K-means process is realised by specifying the key features for each archetype and scoring the clusters on them. The accuracy with which

the clusters represent an archetype is determined by the overall scoring of the key features and is expressed by a confidence value ranging from 0 to 1. The clustering will be evaluated using a silhouette score (squared Euclidean distance) ranging from -1 to 1 to assess the quality of the clustering results. The preferred cluster results are those that combine a positive, stable silhouette score with meaningful separation within the assigned cluster archetypes.

Random Forest (Supervised)

Following up with the question *How collaborative will this repository become?*, the RF model forecasts the future PCR value for window B, based on the baseline metrics from window A. The RF model robustly handles nonlinear relationships between the raw software metrics. It outputs a continuous PCR prediction and corresponding feature importance scores, showing the manager which structural features most strongly influenced the forecast and how changes to those features are expected to affect the predicted collaboration level. The continuous PCR forecast can be compared against the data-driven success threshold defined in 4.3 to conclude whether the repository is likely to reach the high-adoption group. The specific metrics that enable tuning and interpretation for the RF are primarily the RMSE, used to tune hyperparameters within the Spark-native `CrossValidator`, as well as the MAE and R^2 scores. A R^2 score below zero would indicate a non-existing explanatory utility for the resulting RF model; practical usefulness in the context of the MECOIS dashboard would be indicated by a value of 0.2 or higher.

Logistic Regression - Lasso/Ridge and Support Vector Machine (Supervised)

The next question *Which exact levers should be pulled to improve the success chances?* can be answered by the regularised LR. In general, software metrics are highly correlated (multicollinearity). If, for example, the net number of core contributors rises, the number of core pull requests will rise, while the time until a pull request is merged will decline, since more core contributors are available. These correlated metrics do not provide a clear picture of each metric's impact. Regularised LR helps with this issue. While Lasso (L1) shrinks the weights of less important baseline metrics to zero, thereby performing automatic feature selection, Ridge (L2) stabilises the remaining features by adjusting the metric weights based on their importance for the predicted outcome. This process produces a set of linear coefficients that indicate which metrics are most strongly associated with the predicted outcome.

Even though the SVM model delivers a highly accurate margin-based benchmark classifier, it provides no interpretable values, such as feature importance (RF) or coefficients (LR), and is therefore less suited to answering analytical questions in this context. Moreover, its use lies within the functionality of a comparative validation tool during the model evaluation to verify the decision boundaries established by the RF and the LR.

LR as well as SVMs will use several metrics to evaluate their performance. AUC will serve as the tuning parameter for the `CrossValidator`, and accuracy, F1, as well as weighted precision and recall, will serve as the basis for evaluating the models. An AUC higher than 0.70 and an F1 score not lower than 0.60 represent a valid classification model as long as there is no severe precision-recall balance. In the context of Inner Source success, precision describes the proportion of correctly classified successful repositories among all success predictions. In contrast, recall depicts the proportion of repositories categorised as successful among all genuine successful projects. An imbalance between those two values indicates an unstable classifier.

ARIMA - Time-Series Forecasting

ARIMA presents a useful insight regarding the repository's future. If the managerial question is *Will the core team have the capacity to handle peripheral work over the next 6 months?*, then ARIMA can forecast a trajectory of metrics from a training window into the future, providing a capacity planning visualisation for the dashboard. The windows for this model are not retrieved by the window builder algorithm, since ARIMA does not require the training window to meet specific thresholds. Instead, it uses the history as it is and forecasts based on that. In contradiction to the other supervised models, ARIMA cannot provide a classification of static success or comparable tasks. Evaluating the best time-series forecasting model can be accomplished by first choosing the one with the lowest AIC and then the one with the lowest BIC. Holdout MAE and RMSE on the predictive time window indicate the forecast's quality.

4.5 System Architecture

The Conceptual Flow

The new workflow that supports all the capabilities mentioned in this chapter consists of several steps. The initial process is the data retrieval. While GitHub commits are retrieved from the standard GitHub Representational State Transfer (REST) Application Programming Interface (API), pull requests require requests to the GitHub GraphQL API. Those results provide the bronze and the initial silver data following the medallion system of the data lake.
After fully requesting the specified data sources, the identities from those raw data entries must be processed in SortingHat to unify aliases and assign organisational enrolments to the individual contributors. After that, the silver dataframes are updated with the new UUID.
In the next step, the gold data, which contains insights on the silver data, is created using the metric orchestrator. This orchestrator applies onion model logic, identifies potential temporal splits (window A/B) for the ML process, and outputs the metric tables in the gold-standard format.
Subsequently, the gold data is used by the ML orchestrator to train the ML models (Clustering, RF, LR, SVM, and ARIMA). They train on pre-adoption data and export

the predictive models with their respective key values (probabilities, coefficients, and forecasts).
Lastly, the predictive outputs are pushed to Supabase via `upsert`. The MECOIS React frontend queries these tables to populate the interactive Inner Source Dashboard, allowing users to view baseline snapshots and manipulate features in a scenario evaluator.

Data Retrieval

To retrieve the commits, the `/repos/owner/repo/commits` URL has to be queried. Using the parameters `since` and `until`, the required five-year timeframe from 2020 to 2025 can be requested to avoid overloading the network. Following the pagination logic from GitHub REST and robust retry logic, the orchestrator ensures a complete and stable data retrieval process.
To acquire the pull request data, the GitHub REST API does not cover all requirements for the given scenario. Since every pull request combines the history of several commits, an implementation using the REST solution would require creating separate requests for each commit in each pull request. This would result in an unnecessarily high number of requests. Instead, GitHub's GraphQL solution allows a single request to return a pull request and all the commit data associated with it.
During the retrieval of those two data sources, a profile entry is created in the SortingHat database for each contributor mentioned in the resulting data streams. Therefore, the two data source pipelines provide all the information to support the SortingHat: commit data provides at most two, but at least one contributor, the committer and its author. In general, the author creates a commit in GitHub, while the committer publishes it to the target repository branch. Those two can be the same individual, but in frequent scenarios, the committer role is taken over by an automated bot. For pull requests, the procedure is more complex. While the pull request itself has an author and a responsible individual for merging, each unified commit also has authors and committers. As previously discussed, each person contributing to a pull request, which includes code and non-code contributions, must be identified and checked for enrolment status. Therefore, every commit and every other participant must also be considered within a pull request. Using GraphQL, a single request combines all the information required for correct post-processing.
After retrieving the raw bronze data from the corresponding GitHub endpoints and persisting it following the data lake structure presented in Figure 11 in the appendix, the data is converted to the silver status using the MECOIS-derived logic, which anonymises all personal information and creates profile entries in the SortingHat database. Before this happens, the personal data is used to enrich the dataframe with possible organisational enrolments from the GitHub profile information. Therefore, the `/orgs/org/members` endpoint is used to compare users in the local data lake with those whose organisation enrolment is included in their GitHub profile. This helps as a first step in getting additional user enrolments that the next step might miss. Unfortunately, the GitHub profile data and organisation member lists are far from complete.

Hence, this is no single solution but an addition to the main functionality presented in the next step.

Identity Management

Once the SortingHat database contains the first profile entries, the SortingHat worker runs automatic tasks. It finds aliases that belong to the same individual and merges them into a single profile with a single UUID. Additionally, the worker analyses the email domains individuals contributed to. If at least one alias domain of a profile matches the organisational domain specified in the user's settings, the profile is enrolled in the corresponding company. Multiple enrolments are possible as well.
In addition to the automatic tasks, each repository with contributors in SortingHat must be manually scanned for bots or similar profiles that handle automated tasks. Since such accounts often do not have corporate email domains, a commit or pull request involving a participating bot that is not labelled as such would otherwise be filtered out in subsequent pipeline steps. This has to be avoided, since bots used in GitHub repositories fulfil automatic tasks and do not interfere with the thesis application of the onion model. Therefore, the bot status allows their contributions to be actively overlooked by the later role separation between core and peripheral. Because bot usage is inconsistent across GitHub repositories, and each project that uses bots does so differently, this scan for bot accounts must be done manually. Missing a bot with significant participation in a repository results in the majority of contributions being filtered out in subsequent steps, making the repository data less valuable for ML training.
Once the worker and the manual bot labelling have been completed, the results of the identity management step must be returned to the silver data in the data lake. Thus, the existing identity handler pipeline was enhanced to support the transfer of enrolment and bot status information. The pipeline updates the identity information in the silver data, changing the UUIDs of aliases to those of the unified profile from SortingHat. Additionally, the enrolment and bot status of each participating individual of a commit or pull request are inserted into the silver data frames. Existing enrolments from the GitHub group API are thereby preserved and updated only when new information can be added.

Gold Transformation and Metrics Calculation

After handling the identity management, the next step is converting silver data to gold data. First, the silver data has to be filtered to implement the Inner Source proxy. This is realised by removing all unmerged pull requests, then filtering out all contributions that cannot be categorised as corporate internal contributions with certainty. In practice, this means that a commit must have an author and a committer, who is either enrolled in the organisation being processed or marked as a bot. For pull requests, this applies iteratively: first, the creator and the merger must be checked; then the commits within the PR are processed in the same manner as normal commits. Addi-

tionally, any users who only engage in the pull request conversation are considered.
Following the filtering, the contributions must be classified as core or peripheral. To accomplish this, the onion model must be applied to all contributors. Combining all contributions and cumulatively counting them for each user chronologically yields absolute and relative contribution rates for each user at the time of the examined contribution. This way, the importance of a contributor is not only considered over the full timeframe of up to five years, but also the development of one's participation rate is taken into account. If a contributor only participated heavily at the beginning, their validity as a core team member fades over time. Conversely, a contributor joining the repository newly as a peripheral member can move towards a core team role over time. With the newly acquired relative contribution rates, the users can now be classified using two rules. A user is categorised as a core member of the repository if:

1. The contributor falls within the 75% of the top cumulative contributors, or
2. The contributor has contributed over 5% of the total contributions to the repository.

These thresholds were selected empirically to balance two goals: capturing a sufficiently large core team for meaningful analysis while avoiding misclassifying occasional contributors as core members.
Having the role of every contributor at a given time, the commits and pull requests can be classified as purely core or peripheral. If all contributors to the commit or pull request have the core role at the time of the contribution or are flagged as bots, then the contribution can be classified as 'core'. If that is not the case, the contribution will instead be classified as 'peripheral'. This strict separation was found necessary, since even a single peripheral participant in an otherwise core process can indicate the application of Inner Source principles, such as knowledge transfer or community engagement.
Having categorised every contribution into core or peripheral, the gold metrics can be created. Starting with a time series data frame that holds the 'PCR' that was introduced in 4.3 at the time of every contribution. At each point in time, the rate therefore reflects only contributions that occurred before that point, disregarding future contributions. The next metric is the 'contributor funnel width', which reflects the cumulative new core and peripheral users at a given contribution time. This way, the flow of new users can be monitored, which is important to assess repository health. The 'review density' is the next metric created by the metrics orchestrator and represents the average number of comments per pull request and month, split between purely core and peripheral pull requests. This monitors the engagement from the core team towards peripheral contributors. The 'time to merge' metric reports the average and median hours it takes the responsible individuals to merge pull requests, split by type (core/peripheral) and month. This not only captures the efforts of the core team towards new contributions, but also the appeal of a repository for potential peripheral contributors. If the time to merge is very long for non-core members' pull requests, the initiative to invest time in this repository might be lower. Lastly, the 'fork-to-star

ratio' is a basic metric that compares a repository's visibility with the motivation to work on it in depth and is calculated from the two static GitHub variables 'forks' and 'stars'.

Temporal Window Builder (Windows A and B)

To support the predictive models, the repository history must be split into two time windows: a pre-adoption baseline and a subsequent outcome period. As discussed in section 4.1, the intuitive Inner Source scenario is that a repository starts as a largely core-driven, siloed project and later experiences a rise in peripheral contributions once it is opened to cross-silo collaboration. To approximate this scenario using OSS proxy data, a temporal-window builder algorithm is introduced. For each repository, contribution timestamps are converted into repository-relative months, with the month of the oldest contribution treated as month 1. The algorithm then searches for a split point that satisfies a set of duration and behavioural constraints. First, window A (baseline) must span `baseline_months` (12 months), and window B (target) must span `target_months` (6 months). Second, the average PCR in window A must not exceed a configurable maximum threshold (`baseline_pcr_ratio_max`), ensuring that the baseline resembles a mostly core-driven repository rather than an already open project. And third, the PCR in window B must exceed the baseline by at least the `min_peripheral_rise` percentage, ensuring that window B represents a period with a noticeable increase in peripheral participation, even though this does not guarantee an actual organisational adoption event. It could just be organic growth or external factors. Among all candidate split points that satisfy these constraints, the algorithm selects the ones that best match the parameterisation. Windows A are used to derive the baseline features (gold metrics) described earlier. In contrast, windows B are used to compute the target variables (e.g., continuous window B PCR for regression and the binary success label for classification). This design allows ML models to learn from comparable pre-adoption states across repositories that are otherwise very different.

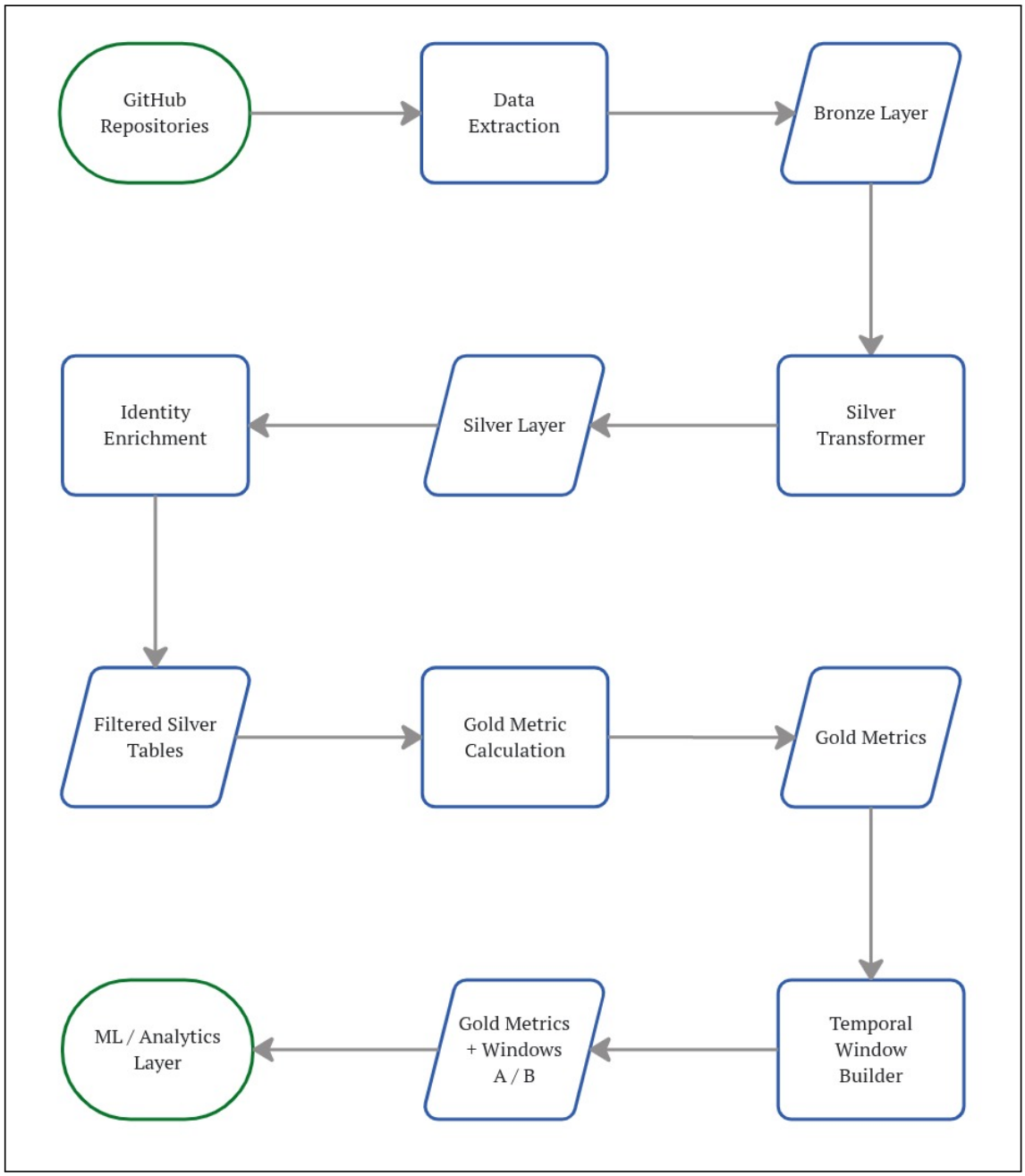

Figure 3: Data Processing Flow: From extracting bronze data, enriching with identities in silver data, creating metrics for the gold data tables, and finding windows for use in the ML step.

Machine Learning Backend

Following up with the ML backend, different ML approaches proceed to generate their models. Serving as the headline predictor, RF handles nonlinear interactions to forecast the continuous future PCR. It answers how collaborative a repository will become based on its baseline state. Tuned via Spark cross-validation on parameters such as

tree depth and count, it outputs feature importance metrics to explain which baseline factors drove the prediction. LR treats success as a binary outcome, defined by a data-driven threshold (reaching the 75th percentile among all repositories in target-window PCR). Utilising L1 (Lasso) regularisation for sparse feature selection and L2 (Ridge) regularisation to stabilise correlated software metrics, it provides transparent directional coefficients that serve as interpretable managerial levers. The training parameters, such as the maximum number of iterations and regularisation parameters, are also tuned via cross-validation. The linear SVM provides an alternative, margin-based binary classification on the same success labels. Tuned via cross-validation, it offers a different inductive bias for separating successful adoption classes. ARIMA operates on regular monthly time series per repository and forecasts the operational trajectories of specific metrics. The parameters `p`, `d`, `q` are optimally ordered by minimising the AIC and the BIC [18].

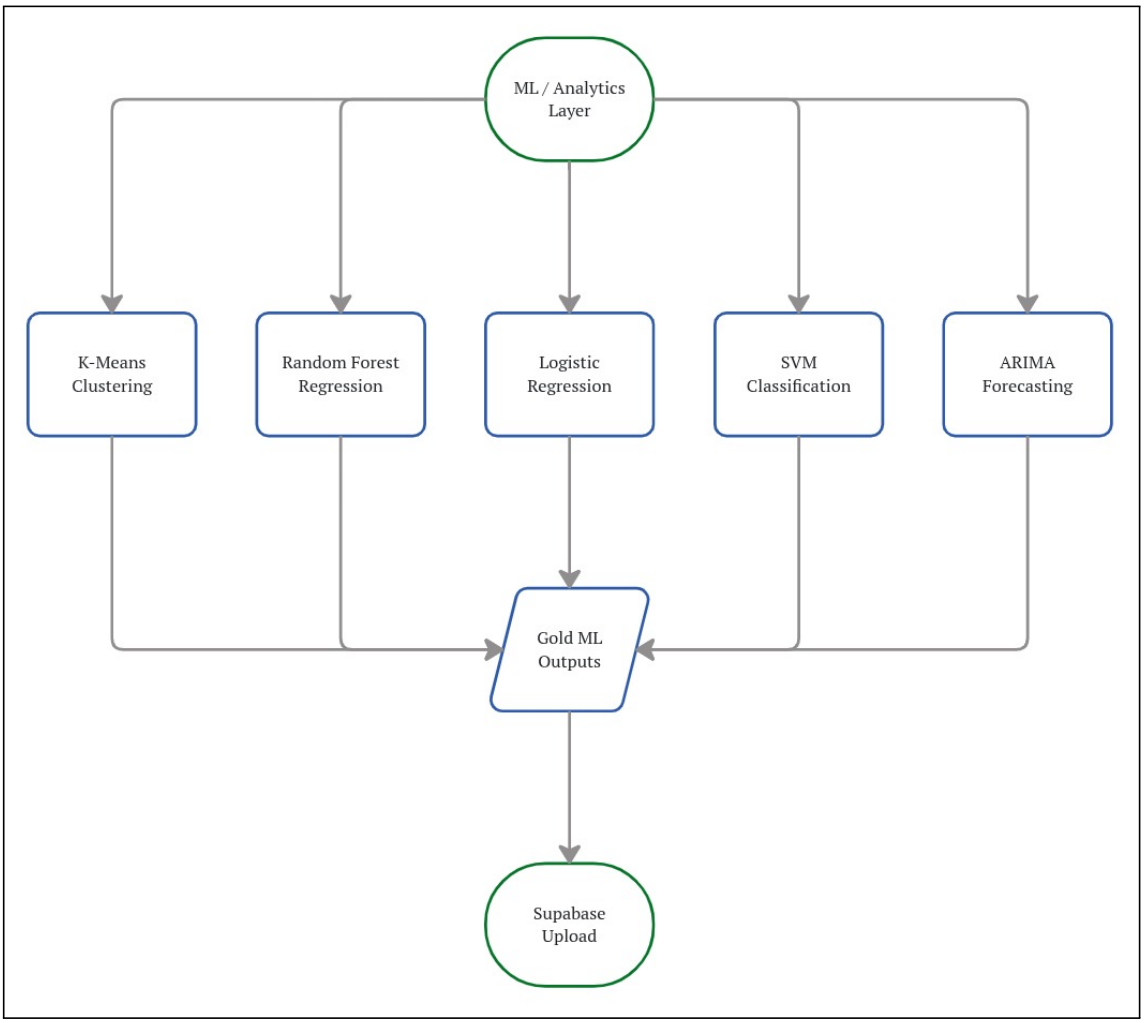

Figure 4: ML Flow: An overview of the ML backend in the system architecture

Supabase and Frontend

The Supabase orchestrator uploads the finalised silver data frames (commits and pull requests) and the gold metrics (ML data, such as predictions, clusters, and ARIMA forecasts). This separation supports a modular architecture. Because silver and gold data are independently deployable, the system allows uploading partial ML metrics, ensuring baseline capabilities remain functional even if specific predictive models are disabled or fail during execution. To maintain data integrity across repeated pipeline executions, the Supabase upload orchestrator enforces strict, table-specific upsert semantics. Silver tables utilise default UUIDs as conflict keys, while gold tables use composite keys, like project, repository and date. This strategy makes re-uploads possible by guaranteeing the deterministic replacement of stale rows and preventing accidental data duplication - a critical requirement for maintaining accurate time-series and model outputs.
The frontend is engineered as an interactive analytical client built on top of the Supabase APIs. The dashboard itself implements fallback-ready behaviour designed to handle the mixed availability of model outputs. While the repository selection and the baseline descriptive snapshots, which are queried from the silver schemas, are always available, the model-specific insights drawn from the gold schemas, including ML metrics such as RF predictions or ARIMA trajectories, are available only conditionally. This procedure ensures that the dashboard always delivers operational value and degrades its UI components if specific advanced predictive tables are absent from the database.
Instead of requesting recomputations from the backend server, scenario simulations are executed entirely on the client. The scenario evaluator dynamically combines persisted baseline model metrics (such as LR coefficients and RF feature importances) with user-defined feature overrides. By applying probability and rate transformations in the browser, the frontend delivers immediate ‘what-if’ feedback. This keeps system complexity manageable while still providing interpretable feedback that managers without a data science background can use.

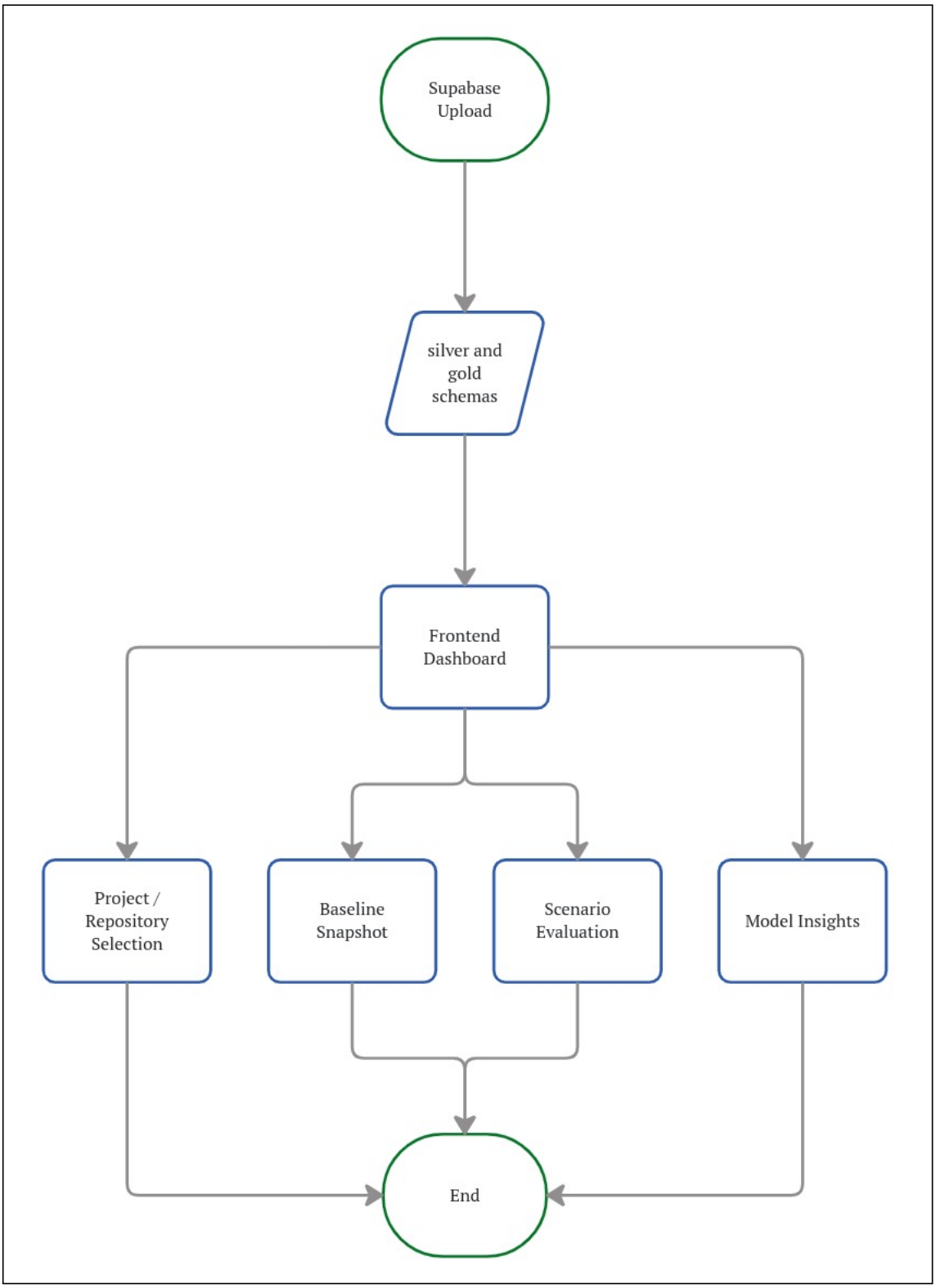

Figure 5: Supabase and Frontend Flow: The system architecture of the connection between backend and frontend, and the architecture of the dashboard

5 Implementation

5.1 The MECOIS Baseline Architecture

Before presenting the Inner Source prediction module, the baseline platform capabilities provided by the inherited pre-existing MECOIS architecture need to be established. To avoid redundant engineering and focus purely on predictive analytics, this thesis inherited several foundational systems. Even though the data pipeline had to be re-implemented to meet the throughput and specifications required for the Inner Source prediction task, the pipeline execution framework was used wherever possible. For the initial step, the abstract `Pipeline` and `PipelineStep` classes were reused, along with the orchestrator-driven execution model. This logic enables dynamic pipeline modification by adding steps where needed, without unnecessarily interfering with existing code.
The data lake topology within the MECOIS platform is realised through a medallion data lifecycle using `Delta Lake` and standard `pandas` and `numpy`. There are three states, each representing a different granularity of processed data. After data extraction, the raw data is stored in bronze data frames. To convert the data to the next state, the silver phase, several alterations are performed. By flattening the table schema, dropping unwanted columns and transforming dates, they transition to a more polished silver phase. After that, specific metrics can be calculated from silver data and written as gold data frames.
All the Extract, Transform, Load (ETL) tasks are conducted with the configured PySpark environment, provided by the MECOIS baseline. YAML configuration files provide centralised configuration for pipelines and other services.
Having silver and/or gold data available in the data lake, the Supabase backend can be used to upload data tables to the included PostgreSQL database. These data tables can subsequently be queried by the React frontend (written in TypeScript), following specified routing patterns. By serving those data tables to the frontend, MECOIS dashboards provide real-time access to all the data that has completed all these steps and can then be analysed in dashboards that render the data in charts and custom components using D3.js.
By leveraging these services, the implementation of this thesis was scoped to add Inner Source logic and ML pipelines.

5.2 Development Environment and Technology Stack

To the existing PySpark library that handles the distributed data processing, `Spark MLlib` was introduced for scalable cross-validation (CV) and model training in the ML process (RF, LR, SVM). To tune hyperparameters with CV for these ML algorithms, PySpark's `CrossValidator` with hyperparameter grids was introduced. The `Spark MLlib` provides classes for classification with LR and SVM, and for regression with RF. Several important evaluation statistics, such as AUC, accuracy, F1, and precision/recall, can be retrieved with this library as well. Following up on the powerful `Spark`

`MLlib`, `statsmodels` includes a specialised library for ARIMA, which is used to perform the forecast. The `scikit-learn` library provides functionality for performing K-means clustering and further pre-processing of the ML features (scaling and normalisation). The details on the realisation of the ML algorithms will be continued in 5.5.

5.3 Extending the MECOIS Data Pipeline

Following the inherited orchestrator pattern, the new data pipeline model integrates into the existing codebase. The newly implemented pipeline steps are based on the abstract `Pipeline` and `PipelineStep` classes and reuse existing `delta lake` functionality, including requests and writes to the data lake. In addition to the global `pipeline-config.yml`, the locally relevant `inner_source_config.yml` configuration file was introduced, which parameterises all orchestrators relevant only to the Inner Source prediction module. Starting with data extraction, the existing commit logic for querying GitHub commits on the default branch can be reused almost entirely. Key differences in the new Inner Source module version included the ability to parameterise the pipeline with the required time frame for extracting commits from the repository. This avoids unnecessary traffic and ensures a comparable data source. Implementing parallelised requesting, as well as smooth retry logic, counteracts the inconsistent behaviour of the GitHub REST API under high load. The original pipeline for retrieving GitHub pull requests also used the REST API in combination with the GraphQL API. The throughput and error rate of this logic were not in line with the requirements of the Inner Source module, as it was unstable in both respects. The main difficulty in retrieving pull requests is that, in addition to the pull request metadata, many nested data fields are required. Every pull request contains data on all the commits it unites and all participants who engaged with it in any way, whether by commenting, committing, or other forms of contribution. To follow the introduced approach of applying the onion model to the OSS data, each of the previously mentioned contributions that a pull request combines must be verifiable to determine whether it consists exclusively of core contributions. To ensure this, the pull request data is retrieved in a single request to the GitHub GraphQL API for each repository. Nested in this request, all pull requests from the repository that match the creation data filter parameters in the `inner_source_config.yml` are listed. Each pull request on this list includes all commits and participants who interacted with it. This new logic works more reliably than the inherited one, as it is parameterised by the local config for the page sizes of pull request, participant, and commit lists.
Having extracted the commit and pull request data, the raw JSON files are transformed to the bronze status and persisted in the data lake, following the original MECOIS implementation. When conducting the silver transformation in the next step, small changes were required. To make the best use of publicly available information about contributors, an extra step is introduced before anonymising their identities. By calling the `/orgs/org/members` endpoint of the GitHub REST API, the currently processed repository's owner can be queried as an organisation in GitHub, which returns

a list of all users that are enrolled in this specific organisation via their official GitHub profile. This provides additional enrolment information apart from the SortingHat enrolments, which will be described in the next step. By comparing the GitHub member IDs from the retrieved organisation list with those from the bronze data in the silver transformation pipeline, matching individuals can be locally marked as enrolled in the organisation. The remaining steps of the `bronze_to_silver_transfomer` remain unchanged from the inherited implementation.

With anonymised silver data in hand, identity resolution is possible. The original `identity_handler` orchestrator, which handles this task in MECOIS, was enhanced to meet the requirements for the Inner Source module. The original implementation allowed the overwriting of UUIDs that belonged to one of a contributor's aliases. The overwritten new value would then be the UUID from SortingHat, which unifies all aliases under a single ID. This was possible for both flat values and nested values organised in a dictionary. The latter is important, since pull requests can contain many individuals within a single entry, and all of those identities need to be updated. What the inherited architecture was missing, and what was therefore newly implemented, was the retrieval of organisational enrolment and bot status from the SortingHat. Therefore, each individual who would be processed under the existing logic was forwarded to the new logic, which accessed the SortingHat profile data for that contributor. Given that the individual was enrolled in an organisation or flagged as a bot in the SortingHat user interface, this information was retrieved and persisted in the silver dataframe. This was done for all participants of the contribution.

Moving on to the creation of repository metrics and, therefore, gold data frames, the next step was to apply the OSS proxy logic by filtering the contribution data frames to retain only contributions made entirely by organisation members. Therefore, every participation in a commit or a pull request is considered, from commit authorship, to committer, over to the creators of pull requests and the ones responsible for merging, but also all other participants within a pull request. Individuals classified as drive-by contributors are excluded because they do not regularly interact with the repository. The default threshold for contributions to count as a drive-by contributor is 3, meaning that committing more than 3 times to the same repository within the specified timeframe will prevent the contributor from being filtered out. As reasoned in section 4.5, strict filtering is applied: only contributions where all participants can be traced to the organisation (or labelled bots) are retained. The last prerequisite before the gold metrics can be calculated is applying the onion model. With the filtered contribution data tables, the next step is to aggregate the contribution amounts per unique contributor. Hereby, all participations are counted individually as contributions. If, for example, an individual creates a pull request containing a single commit, the commit, as well as the authoring and merging of the pull request, are counted as a single contribution. This allows for acknowledging every participation, even if it is only a small one in a complex pull request. With this in mind, time-sensitive classification can be conducted. The contributions are ordered chronologically, so that the classification can consider the current and all past entries. This enables the classification process to simulate the dynamic behaviour of contributing over time in a repository. Taking all

observed contribution events into account, the classification is then iteratively applied to the chronological contribution data. The classification itself thereby follows the two rules presented in 4.5. Spectating a contribution at a specific time, a contributor is considered a core member if either: the amount of running contributions (contributions until now) of said individual is in the top 75% of the entire cumulative running contributions of a repository; or the relative running contributions of a contributor are more than 5% of the overall running contributions. To avoid an unstable classification dynamic during the early events of a data set, the threshold `early_batch_days` is introduced. This sets the classification entry point in the time-series chronology, classifying contributions up to the threshold as a batch and classifying them per event afterwards, thereby avoiding frequent re-classifications of users in the early stages of the data set.
With the filtered and classified contribution data frames at hand, the gold metrics can be calculated for each repository, starting with the important `PCR` in a time-sensitive manner. Applying the formula introduced in 4.3 onto the cumulative contribution counts over time, a time-series data frame is created that holds the PCR at the time of the given contribution event, considering all running contributions on a daily level to reduce the calculation effort. The next calculated metric is the `contributor_funnel_width`. This measures the number of new developers who join each day and have not previously contributed to the repository. The end product is a dataset that groups new contributors to the repository by day. As discussed earlier, this metric is used to monitor the repository's attractiveness for new developers over time. Engagement between already existing participants is measured by the `review_density` metric. It focuses solely on pull requests and their review activities. This is implemented by averaging the number of comments per pull request over a month. Grouped by pull request classification and month, the result is a dataframe containing the average number of comments per core or per peripheral pull request and per month, along with the respective total number of pull requests. The average comments allow for comparison of review practices between core and peripheral members and therefore provide quantitative insight into community engagement. Another metric that analyses the repositories' interactions with different contributor types is `time_to_merge`. It is calculated by subtracting the time of pull request creation from the time of merging. The resulting time difference is then converted to hours, yielding the time it took to merge this specific pull request. Doing so for all pull requests, and again grouping the results by pull request classification (core/peripheral) and by month, provides insights into how well changes are introduced into the repository and whether pull requests are handled differently depending on their contributing members. The last metric analyses the repository's visibility and attractiveness to new contributors who might invest time in it. The `fork_to_star_ratio` is the ratio between the total forks and the total stars of the repository and gives high-level information about the general performance of the repository. Due to GitHub API limitations, this metric cannot be traced back to the time of individual contributions, as only the current fork and star counts at the time of the data retrieval are available. Therefore, this metric only indicates the repository's current state, rather than measuring the development of the numbers over time.

5.4 The Temporal Window Builder Algorithm

The classified silver data and gold metrics must now be prepared for ML training to address the 'cold start' problem described in section 4.1. To obtain training and test data that reflect a pre-adoption baseline and a subsequent outcome period, the temporal window builder algorithm extracts two periods: window A (baseline) and window B (target). Implementing the window builder involves 4 steps. At first, the timestamps are converted to repository-relative months. The month from the oldest contribution is set as the first month. This procedure is then chronologically forwarded towards the repository data. Then, given the parameters `baseline_months` and `target_months`, which indicate the durations of windows A and B, respectively, several potential split points are evaluated. The months representing such a split point must ensure that both windows have at least the number of months specified by the parameterisation and that the selection criteria introduced in section 4.5 are satisfied. The splitting points that meet those requirements are then selected for the repository. In the third step, the input features for the windows A are calculated using the same calculations described for the gold metrics in the previous section. Finalising the window algorithm, the target variable is selected for the B windows. Depending on the application of the window builder algorithm, this variable can vary, though in most scenarios it is the PCR.

5.5 Realising the Machine Learning Pipelines

The implementation of the ML models and their pipelines follows the same architecture as the data extraction, using orchestrators, pipelines and transformers. In addition to the transformers, which handle the main training and testing processes, a helper function in `ml_utils.py` provides centralised logic that can be reused across models, such as data scaling, feature assembly, and evaluation metrics. The feature assembly consists of a vectorising logic that consolidates all features required for training the model into one single vector using the `VectorAssembler` function from the PySpark ML library. Here, any missing values are also detected and replaced. Another important step in preparing for the training of ML models is standardising the input features. Using PySpark's `StandardScaler` function, features with different scales (e.g., fork counts versus hours-to-merge) can still contribute equally to the model training. For classification tasks like LR, a success label is assigned if the window B (target window) PCR exceeds the 'success percentile' of the training distribution. This percentile is provided via a threshold parameter in the corresponding configuration files.

Moving on to the technical implementation of the ML models, all of them (except ARIMA) use their related PySpark library. The standard approach to splitting data into training and test sets is a 70/30 split.

LR is implemented in two variants, differing in the parameter `elasticNetParam`. This parameter is either set to 0 to use the L2 norm (Ridge regularisation; handling multicollinearity among metrics) or to 1 to use the L1 norm (Lasso regularisation; fea-

ture selection by shrinking the coefficients of less important features to 0). Finding the right regularisation parameters and maximum iterations (the hyperparameters) is achieved by applying the `CrossValidator` with a `ParamGridBuilder` that explores different hyperparameter combinations. The evaluation of those combinations is conducted by PySpark's `BinaryClassificationEvaluator`, which provides a parameter (in this case AUC) to the CV that should be maximised when choosing the hyperparameters. The configuration of an LR model could look like the YAML profile in listing 1 in the appendix.
The SVMs are realised by implementing a linear kernel SVM for binary classification (success/no success). The outputs from the corresponding PySpark library contain feature coefficients indicating the influence of each feature on the classification result. Further, the library produces a decision margin that measures the distance between the decision boundary (a hyperplane) and the nearest data points from both classes (the support vectors). The optimisation of the hyperparameters, as well as the evaluation, is conducted in the same manner as for the LR.
The implementation of the RF uses the regression variant from the PySpark library. The PCR in the target window (window B) of the training data set is the target on which the regressor is trained. Since the RF uses multiple decision trees, the number and depth of those trees represent the hyperparameters for the training. Those parameters are again tuned using K-fold CV. The parameter of the `RegressionEvaluator`, in this case the RMSE, specifies which configuration yielded the best results, the one with the lowest RMSE. Those hyperparameters are then used for training. The output of the RF model training provides a list of scores that indicate the importance of different features on the prediction. Those scores represent the average importance of each feature across all trees in the ensemble. The importance vector is normalised at the end.
The only unsupervised technique used in this thesis is K-means clustering. To find the optimal value for K (the number of clusters), the results of multiple cluster counts are compared. Therefore, a minimal and maximal value for K are defined in the `inner_source_config.yml`. The best K is determined by comparing silhouette scores (delivered from the `ClusteringEvaluator`) for each different-sized clustering. Having identified the optimal number of clusters for the underlying repositories, the clusters are assigned to an archetype (see 4.4) using a rule-based scoring system. Each cluster has a set of metrics that are averaged across all repositories in that cluster. The scoring system then compares each cluster's metrics with the median across all clusters to determine how that cluster performs relative to the others. This checks whether a cluster deviates explicitly in specific metrics. Each archetype requires the cluster to vary from the overall median for certain metrics. These specific metrics are the archetype's characteristics.
The last ML model implemented is the ARIMA time-series forecasting. It uses the ARIMA library from the `statsmodels` package. To fit a time-series model to historical data, the data must first be resampled to a monthly frequency. Since the statsmodels library does not operate directly on PySpark dataframes, the monthly data must be converted to a `pandas` Series before continuing. In the next step, the data is split into two sets: the training and holdout sets. The size of both data sets is again defined

by the parameters `baseline/target months`. While the train set is used to train the model to learn historical patterns, the holdout set is withheld from the model during training. It is used later to compare it with the model forecasts and evaluate the model fit. Unlike supervised ML models, ARIMA does not use cumulative metrics but instead uses raw monthly volumes. Cumulative metrics would lead to rapid convergence, and forecasting the actual monthly flow would not be possible. The ARIMA model further relies on three hyperparameters, p (autoregression), d (differencing) and q (moving average). To find the best configuration of these parameters, every possible combination is tested. The candidates are hereby specified via the configuration file. The best combination is determined by evaluating a model's performance across different AIC-BIC pairs. Applying both criteria, an optimal configuration is found that balances accuracy and simplicity. Having tuned the hyperparameters, the forecasting is conducted over the provided amount of `forecast_periods`. In the end, the forecast results are converted back into PySpark data frames.

5.6 Database and UI Integration

Having completed the implementation of the ML models, their functionality, along with the silver and gold data, must be made available to the end user in the Inner Source prediction dashboard. Since the frontend, built with React, uses Supabase as an integration layer to access data, the locally created data tables must first be uploaded to the Supabase PostgreSQL database. This is achieved by the `inner_source_dashboard_upload_processor`, which reuses the inherited `SupabaseUploadPipeline` and extends its functionality. Processing the local data tables in batches and converting them to JSON, the upload follows optimal throughput to handle the Supabase API limits. The pipeline supports routing data to bronze, silver, and gold PostgreSQL schemas, for which several new schemas were created to match the newly acquired data structure of the gold metrics and the ML model outputs. Each part of the uploaded data is linked to an `organisation_id`, confirming that the frontend dashboard retrieves only data relevant to the user's current organisation context. Re-running the pipeline to add changes to existing data tables in Supabase is also possible; therefore, several upsert conflict constraints are implemented to ensure data integrity across successive ML experiments.

After the Python backend generated the silver and gold data tables and the upload orchestration pushed them to Supabase, the frontend application can now fetch baseline data (ML matrices, coefficients and forecasts) via the Supabase client. To provide the possibility for the user to perform real-time scenario evaluations, which use the baseline repository data and altered parameters that the user can introduce via user interface slider modules, the dashboard first needs to retrieve the pre-trained model coefficients for LR and the feature importances for RF from the Supabase gold layer. When the user now operates a slider and introduces a delta for a feature metric, the frontend adapts the displayed ML predictions. If LR is selected, the baseline feature vector is used to recalculate the classification probabilities using `logit` and `logistic` functions. This is required, since the LR coefficients do not operate in a probability

space, but in a log-odds space. Thus, to apply the impact of the changed slider, the baseline probability p of the trained model is converted to log-odds space using the `logit` helper function from Equation (1).

$$logit(p) = \ln \frac{p}{1-p} \tag{1}$$

Now, the user input $\vec{\Delta}$ is multiplied with the coefficients $\vec{c}$ of the LR model in Equation (2). Both vectors contain values for the different features. Adding the weighted coefficients to the log-odds value of p from Equation (1), the new log-odds value x reflects the adjustment the user made in the frontend by changing the sliders.

$$x = logit(p) + \sum_i \Delta_i \cdot c_i \tag{2}$$

To return a human-readable probability in the range of 0 to 1, the newly adjusted log-odds value x has to be converted back to a probability space using the inverse of the `logit` function, the `logistic` function. This yields the new weighted probability p_w, which incorporates the changes from the user interface.

$$p_w = \frac{1}{1+e^{-x}}; \quad p_w \in (0,1) \tag{3}$$

If the RF model is selected, the adaptation of the prediction is implemented as in Equation (4). Because the feature importances of RF reflect the impact of each feature on the variance across all decision trees during training, adjusting those importances requires complex recalculations that are not feasible in the dashboard environment. Since feature importances are normalised (i.e., they sum to 1), the impact of a change in the user interface can be approximated using a first-order linear model. Therefore, it is assumed that the user's local rate of change is directly proportional to the feature's global importance, with the change in probability Δp approximated by the product of the manual feature weight change Δx and the feature importance weight w_f delivered from the RF, as depicted in Eqaution (4). The manual change the user applies via the slider inputs operates on a normalised version of the feature, returning a percentage change value. Since this is only an approximation, interval bounding is required to ensure that the resulting probability remains in the correct interval $(0,1)$.

$$\Delta p \approx w_f \cdot \Delta x; \quad w_f \in (0,1) \tag{4}$$

This architecture enables immediate updates to the user interface, visualising the altering adoption scores and forecasts as users adjust key parameters via slider input elements.
To display other data from the ML models, different data visualisation techniques from the MECOIS frontend implementation are used. Visualising time-series forecasting based on the ARIMA forecast is achieved using D3-based line charts. The forecasts

visualised here respond dynamically to the user's scenario changes. To familiarise the user with the feature impact estimates, horizontal bar charts visualise the relative impact of each scenario feature on the final prediction. Raw metrics, such as commit and pull request counts, the archetype derived from K-means clustering, and other repository details, are provided in separate spaces.

5.7 Practical Challenges and Pragmatic Solutions

Starting with the implementation of the data pipeline for the Inner Source module, the first major obstacle was the GitHub resource limit exhaustion when using the REST and GraphQL API. Since the original MECOIS pipeline was not intended for the broad data retrieval required for training the Inner Source module, the inherited pipeline quickly reached its limit when processing thousands of pull requests, each with nested commits and participants, leading to abrupt terminations of the API requests or timeouts on the connection to the API service. Introducing the pure GraphQL approach helped address many of the inconveniences, but still resulted in frequent resource limits and timeout errors. Those errors occurred because the queries required to retrieve all relevant contribution data from the pull request endpoint. Implementing an adaptive retry mechanism (comparable to, but enhanced over, the original MECOIS implementation) led to a more stable data extraction process. The new mechanism detects resource limits, automatically reduces the page sizes of the different parameters (pull requests, commits, participants), and retries the specific failed partition. This allows extraction to proceed for dense repositories without disrupting the entire pipeline.
Apart from the mentioned difficulties in acquiring organisational project data, the creation of the Inner Source metrics also introduced complications. Inner Source metrics are intended to measure activity within an organisation, but because the available projects are OS repositories, contributions sometimes originate from external developers. Additionally, many repositories use automated systems (bots) for committing, merging or interacting with contributions in other ways. The kinds of bots that find application on GitHub are diverse, leading not only to different names and versions of bots, but also to some that resemble normal contributors and others that are specifically designed for a repository. This made finding and flagging them as bots a manual and iterative task, since missing a bot can lead to filtering out all contributions it participated in due to the enrolment logic described in 5.3. To avoid external developers or automated systems skewing the Inner Source metrics, the strict enrolment filtering was introduced. Any contribution that could not be fully traced to members of the organisation, including bots, was dropped.
Having already mentioned the source of the training data, another issue resulted from the missing 'cold start' within the base data. Because MECOIS is designed to analyse mainly repositories that are not yet opened for Inner or Open Source, the training of the ML models had to be conducted on such repositories. As the available projects were mostly OS repositories, the cold start had to be simulated. Thereby, the differing start times and evolving speeds of the repositories had to be considered. Using fixed calendar dates for training introduces unwanted noise, as minimal activity phases are

treated the same way as active development phases. The temporal window builder was therefore implemented to use relative month indices. That way, every repository's timeline is indexed starting from its first contribution (month 1). The builder then searches for a rise in peripheral contribution activity to identify the most informative windows A (baselines) and windows B (targets) for the supervised learning methods. The resulting windows A and B form comparable pre-adoption and outcome states.
Training, and especially CV on a parameter grid for LR, posed severe performance and stability issues in PySpark. Using a hyperparameter grid with 8 combinations and 5-fold CV required 40 training cycles for Lasso and Ridge, respectively. Training 80 LR models on a weak CPU caused the model to not complete training due to the Java Virtual Machine running out of heap space and using most of the processing time to clean up allocated memory with the Garbage Collector. Only after applying targeted caching and unpersisting logic, limiting driver throughput, and avoiding automatic broadcast joins on small datasets was the available memory used effectively, and the training process could succeed.
Applying the ARIMA time-series forecasting on repository contribution data showed problems when dealing with discontinuous data sequences, which are common when working with contribution data. Gaps in the analysed monthly activity result in 'silent months' with no activity, which could lead to convergence failures when fitting the model to the training data. To avoid fitting ARIMA to dead repositories that exhibit long, contributionless trends, the training data had to be filtered.

6 Demonstration

The MECOIS Inner Source Scenario Dashboard was designed to assess whether strengthening Inner Source practices is likely to improve collaboration within a given repository. By translating complex ML models into an interactive interface, the dashboard allows engineering managers to adjust organisational levers and visualise how structural changes affect a repository's likelihood of adopting Inner Source. This chapter demonstrates the user interface of the MECOIS Inner Source Scenario Dashboard, outlines its main segments, and shows how the dashboard supports exploratory 'what-if' reasoning for Inner Source adoption decisions.

User Interface

The dashboard is structurally divided into three primary segments: Context and current reality, scenario levers, and predictive projections. While Figure 12 in the appendix provides an overview of the entire dashboard, the following section will concentrate on the specific parts of the user interface.

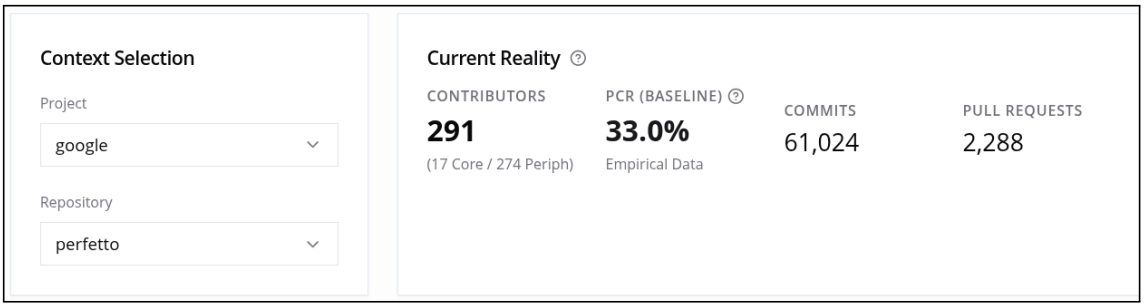

Figure 6: Dashboard - Top navigation bar and Current Reality card

The top part of the dashboard establishes the empirical baseline (see Figure 6). Users can select their target environment by choosing a specific project and repository (e.g., Project: `google`, Repository: `perfetto`). Once a repository is selected, the dashboard queries the database to display live baseline metrics: the total number of contributors, divided into core and peripheral groups (e.g., 291 total: 17 core / 274 peripheral), the empirical baseline PCR, the total number of commits, and the total number of pull requests.

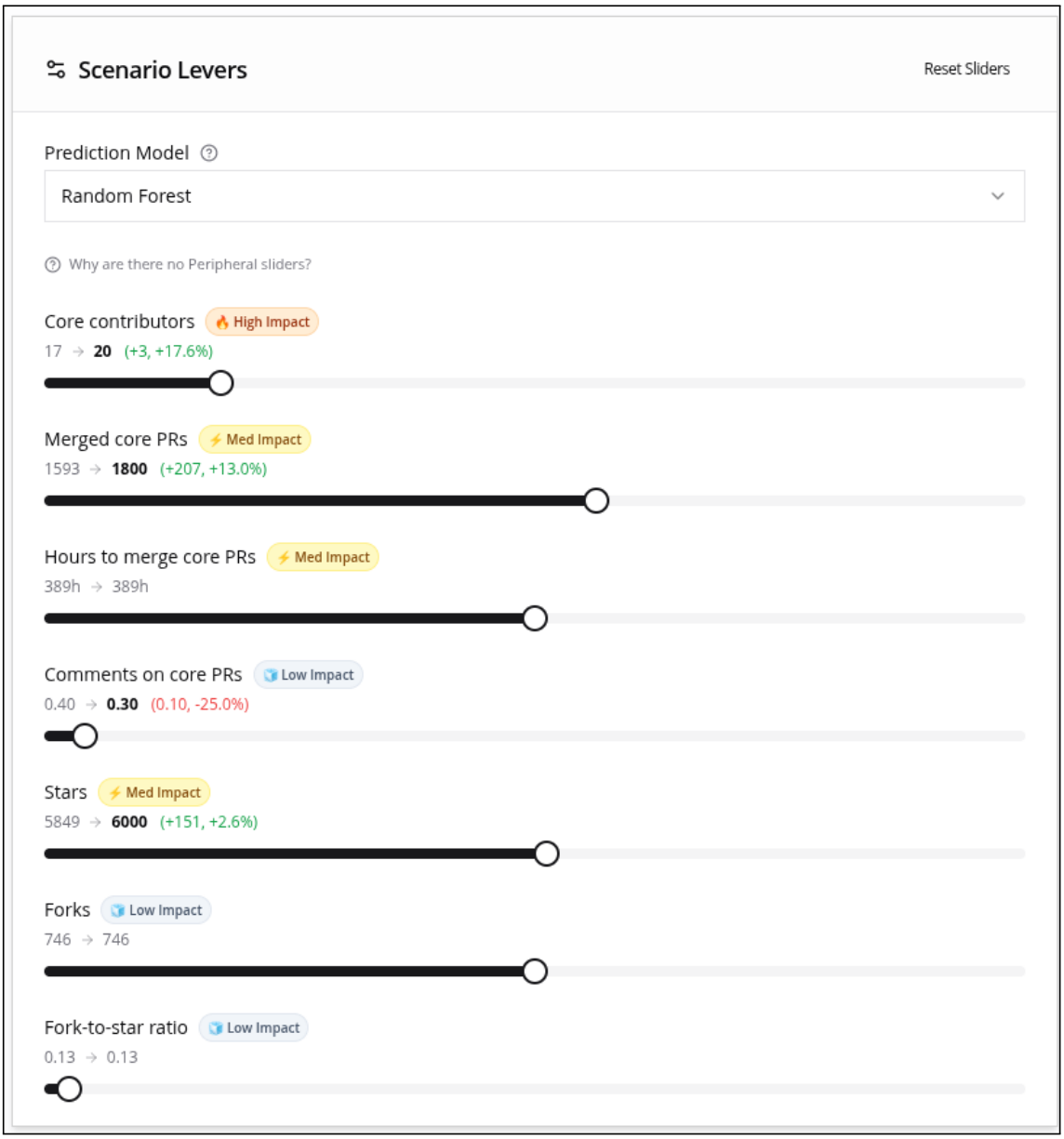

Figure 7: Dashboard - Scenario levers and prediction model selection: The column on the left of the dashboard can be used to change the prediction model (Lasso LR, Ridge LR and RF), or simulate a change in a feature parameter through one of the sliders.

The left column houses the 'Scenario Levers'. Here, the user can select their preferred prediction model (regularised LR or RF). Below the model selection, users are presented with interactive sliders for structural metrics, including 'Core contributors', 'Merged core PRs', 'Hours to merge core PRs', or 'Comments on core PRs'. To guide the user's attention, each lever features a dynamically calculated impact badge based on the selected ML model. For example, 'Core contributors' may be flagged as 'High Impact' lever, while 'Comments on core PRs' might register as 'Low Impact'. The feature

badges thereby depend on the feature importances (RF) and the feature coefficients (LR) and may therefore differ across the selectable ML models. In a real-life scenario, a manager might find that increasing the number of core contributors by 20% offers a significant increase in the projected PCR. At the same time, the other metrics remain unchanged, which often leads the model to predict that the repository will successfully adopt Inner Source, highlighting the high feature importance of this metric.
The interface features a toggleable tooltip asking, 'Why are there no Peripheral sliders?'. This explains the cold-start theory applied in this thesis to managers. Because the model predicts future dynamics of peripheral contributors, users can only manipulate the core parameters of the baseline repository.

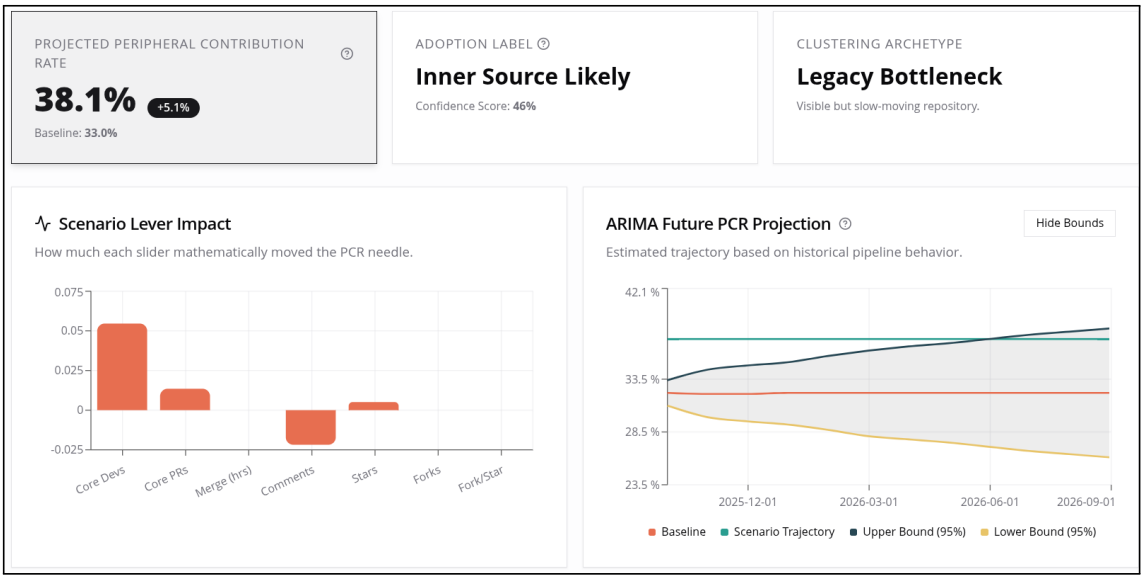

Figure 8: Dashboard - Prediction Visualisations: The right-hand column displays the results predicted by the selected model in real time, based on the user's changes in the scenario levers panel.

As the user adjusts the scenario levers on the left side of the dashboard, the right column updates in real-time to visualise the projected outcomes (see Figure 8). The dashboard thereby dynamically calculates the new, theoretical PCR based on changes from the scenario lever card. If the rate crosses the model's success threshold, defined as the top quartile in the cross-project PCR distribution, the repository earns an 'Inner Source likely' adoption label, backed by a confidence value representing a heuristic probability derived from the sum of the scenario forecast and the weighted delta value of the PCR change. Below the PCR and adoption cards, a bar chart breaks down how much each adjusted slider mathematically influenced the change in PCR, providing insights into the model's decision-making. Next to the bar chart, a time series chart plots the baseline PCR trajectory against the newly simulated scenario trajectory using the ARIMA forecasting data. The scenario trajectory is obtained by applying the predicted delta in PCR to the baseline ARIMA forecast over the prediction horizon. A shaded 95% confidence interval supports the baseline to convey forecast uncertainty

transparently.

The MECOIS dashboard is designed, on the one hand, for managers who need a quick, comprehensive overview of different repositories (e.g., a company portfolio) and who must be able to identify projects with the highest potential for adopting Inner Source and plan resource allocation accordingly. On the other hand, developers and maintainers can analyse repositories that are struggling to attract new contributors. If, for example, a repository shows a low PCR, its maintainer can use the dashboard to better understand which factors are preventing the project's progress towards Inner Source adoption and to support decision-making on how to improve the repository's structure and collaboration.

7 Evaluation

7.1 Objectives and Questions

The purpose of the evaluation is to assess whether the artefact designed in this thesis fulfils its planned role as a predictive decision-support module for Inner Source adoption within the MECOIS platform. In line with the research questions introduced in chapter 1.3, the evaluation follows four main objectives:

1. **Data and Pipeline Quality:** Verify that the extended MECOIS data pipeline correctly implements the OSS proxy, identity resolution, and onion model logic, and that the resulting bronze, silver, and gold data are complete and internally consistent for the selected repositories.
2. **Metric Behaviour and Feature Validity:** Examine whether the derived collaboration metrics (in particular the PCR, contributor funnel width, review density, time-to-merge, and fork-to-star ratio) behave in a manner that is consistent with the underlying Inner Source theory and are suitable as features and targets for predictive models.
3. **Predictive Performance of the Models:** Evaluate the extent to which the ML models (RF regression, LR, SVMs, K-means clustering, and ARIMA time-series forecasting) can accurately predict future collaboration outcomes and correctly classify Inner Source adoption scenarios as successful and non-successful.
4. **Practical Utility of the Dashboard:** Assess whether the integration of the predictive artefact into the MECOIS dashboard provides interpretable insights for managers, in particular through archetype labelling, feature importance visualisation, and real-time scenario evaluation.

These objectives can be summarised in the following evaluation questions (EQs), which complement the research questions from chapter 1.3:

EQ1: Does the data pipeline produce a reliable and organisation-consistent view of repository collaboration that is suitable for Inner Source analysis?

EQ2: Do the implemented metrics and features capture meaningful aspects of Inner Source collaboration and align with the conceptualisation of success developed in this thesis?

EQ3: To what extent do the ML models achieve predictive performance that is sufficient for decision support when forecasting PCR and classifying repositories into successful and non-successful Inner Source adoption groups?

EQ4: Does the MECOIS Inner Source dashboard present the predictive results in a form that is interpretable and practically useful for managers when exploring repository readiness?

7.2 Design and Data

To address the evaluation objectives and questions outlined above, this chapter follows an evaluation design that examines (i) data and pipeline quality, (ii) metric behaviour, (iii) predictive performance of the ML models, and (iv) the practical utility of the MECOIS Inner Source dashboard. The evaluation is structured along different components. The pipeline and data quality evaluation focuses on the correctness of the extended MECOIS pipeline by analysing the different stages of the medallion data lifecycle. This includes verifying that silver tables correctly apply identity anonymisation and organisational enrolment via SortingHat and the GitHub API, and that gold tables contain only contributions that satisfy the OSS proxy and onion model criteria. Summary statistics are computed for the number of raw identities, resolved profiles, labelled bots, and retained contributions per repository. Further, the behaviour of the derived gold metrics is examined on representative repositories from the evaluation dataset. Time-series plots and descriptive statistics for PCR, contributor funnel width, review density, time-to-merge, and fork-to-star ratio are examined to determine whether they reflect theoretically expected collaboration patterns and whether they provide discriminatory power between more and less collaborative repositories. To evaluate the predictive ML models, a test set and cross-validation are used, following the training procedures described in chapter 4 and chapter 5. RF regression models are evaluated using RMSE, MAE, and R^2 against a naive mean-prediction baseline. LR and SVM classifiers are evaluated using AUC, accuracy, F1-score, and (weighted) precision and recall. K-means clustering is assessed using internal validation measures such as silhouette scores and by qualitatively interpreting the resulting archetypes. ARIMA models are compared using AIC and BIC, and by inspecting forecast errors on a holdout period. Moving on to evaluating the front-end dashboard's usability, the integration of the predictive artefact into the MECOIS dashboard is qualitatively assessed by examining whether archetype labels, feature importance visualisations, and scenario-based probability and forecast updates are presented consistently and in an interpretable manner.

The evaluation is based on the dataset assembled and processed in chapter 4 and 5. The key characteristics of this dataset are as follows:

- **Source Projects:** The repositories originate from corporate-sponsored OSS projects identified via the OSCI and OSSInsight platforms. The final dataset comprises 31 repositories from 10 organisations, spanning up to 5 years from 2020 to the end of 2025, although some repositories cover only a shorter period.
- **Data Sources:** For each repository, commit histories and pull request data are collected from the GitHub REST and GraphQL APIs. Bronze tables store the raw JSON responses; silver tables contain flattened and anonymised records, enriched with SortingHat-derived organisational enrolments and bot labels; gold tables hold the derived collaboration metrics and ML features.
- **Identity and Proxy Filtering:** Identity resolution via SortingHat and GitHub organisation membership lists ensures that all contributions can be attributed

either to enrolled organisation members or to labelled bots. Contributions that cannot be fully traced to the organisation, as well as contributions from 'drive-by' contributors below a configurable activity threshold, are excluded to implement the Inner Source proxy.

- **Temporal Windows:** For each repository that meets the minimum activity and duration criteria, the temporal window builder algorithm identifies baseline periods (windows A) and subsequent target periods (windows B) using relative monthly indices. Respective windows A are used to derive input features. In contrast, windows B are used to compute target values such as PCR for regression and to derive binary success labels for classification.
- **Training and Test Partitions:** Within the selected repositories, data are partitioned into training and test sets. For the supervised models, a standard 70/30 split for the train/test sets is applied after window construction, with k-fold CV on the training set for hyperparameter tuning. For ARIMA, the time series are partitioned into a training window and a holdout window based on the `baseline_months` and `target_months` parameters, without the use of the window builder algorithm.

7.3 Pipeline and Data Quality

The first step in the evaluation is to assess whether the extended MECOIS pipeline produces data that are sufficiently complete, consistent, and aligned with the Inner Source proxy design to support the subsequent ML models (EQ1). Table 1 provides an overview of the identity resolution outcomes, presenting values from exemplary repositories, and Table 2 covers the Inner Source proxy filtering steps across the same repositories, with respect to the 31 repositories in the evaluation dataset.

Table 1: Identity resolution across the evaluation dataset.

Project	Repository	Raw IDs	Resolved Profiles	Bots	Enrolled Profiles
nvidia	TensorRT-LLM	682	682	4	215
intel	llvm	8528	5541	11	1250
ibm	mcp-context-forge	195	133	4	46
microsoft	vscode	4765	3788	22	399
google	perfetto	763	583	16	423
	Median	268	242	5	104
	Total	29156	21673	218	5845

Table 2: Proxy filtering across the evaluation dataset.

Repository	Total Commits	Kept Commits	Total PRs	Kept PRs	Kept overall %
TensorRT-LLM	4427	2440	6836	2658	45.26
llvm	249267	39671	17325	10539	18.83
mcp-context-forge	1781	1226	950	428	60.56
vscode	85989	80399	41285	31309	87.77
perfetto	61671	61024	2915	2288	98.03
Median	5807	4398	4342	2288	65.88
Total	667591	343306	213425	123982	53.04

Across all repositories, the identity resolution pipeline successfully mapped 79.02% of raw contributor identities to SortingHat profiles, with 5845 profiles receiving at least one organisational enrolment. A total of 218 accounts were manually labelled as bots. Contributions associated with unresolved identities or external contributors were removed during the Inner Source proxy filtering stage, leaving 73.17% of commits and 53.51% of pull requests for further analysis. This confirms that the majority of activity in the evaluation dataset can be attributed to verifiable organisation members or bots, as required by the proxy design. The temporal window builder algorithm (see section 5.4) was applied to all 31 repositories. Table 8 in the appendix reports to what extent the tested repositories satisfied the minimum duration and threshold constraints and therefore yielded valid baseline and target windows. Out of 31 repositories, the algorithm identified 14 windows (45.16%) that met both the minimum duration criteria (12 months for window A and 6 months for window B), as well as the `baseline_pcr_ratio_max` (25%) and `min_peripheral_rise` (5%) thresholds. As a concrete example, in the `adobe/react-spectrum` repository, the algorithm selects months 8-19 as window A and months 20-25 as window B. In this case, the average baseline PCR in window A is about 0.19 and rises to 0.37 in window B, satisfying both required thresholds. Repositories that showed a less noticeable increase in peripheral participation were used as fallback time windows. The remaining repositories that exhibited insufficient activity over time were excluded from the supervised ML training. This filtering aligns with the intended Inner Source scenario and ensures that models are trained preferably on repositories that exhibit meaningful collaboration dynamics. Spot checks and automated validation rules were used to verify that the medallion stages are consistent:

- For a sample of repositories, the number of commits and pull requests in bronze tables matched the counts returned by the GitHub APIs within the expected margin imposed by the time filters.
- For the same sample, the number of contributions in silver tables equalled the sum of contributions in gold tables after filtering and onion-model classification,

confirming that no additional loss of data occurred beyond the designed proxy and drive-by filters.

- Random checks of data integrity confirmed that all contribution records in the gold tables are linked to resolved contributor profiles and repositories.

No systematic anomalies were observed that would call into question the pipeline's correctness. Minor discrepancies (e.g., missing GitHub profile enrolments) were expected given the incompleteness of public profile data and were mitigated by the SortingHat-based identity resolution.

7.4 Metric and Feature Evaluation

Before assessing the predictive ML models, the behaviour of the derived collaboration metrics is examined to determine whether they capture meaningful Inner Source dynamics and provide useful predictive signals (EQ2).
Figure 9 illustrates the evolution of the PCR over time for two representative repositories: one that was labelled as successful (top-quartile window B PCR) and one that was not.

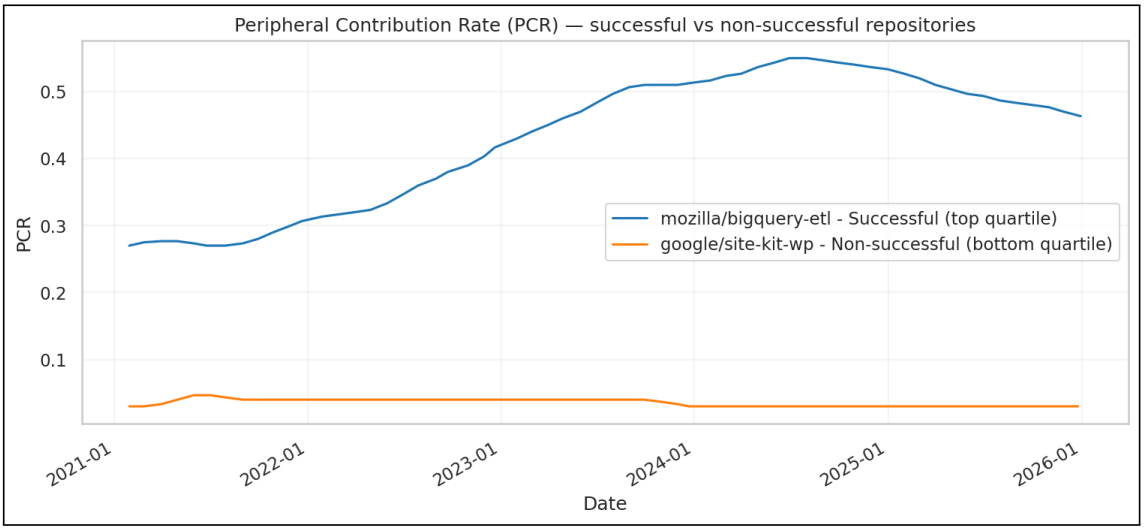

Figure 9: PCR for two classified repositories: Example for a successful repository (mozilla/bigquery-etl) from the top quartile and an unsuccessful repository (google/site-kit-wp) from the bottom quartile, classified according to the Inner Source success-threshold decision boundary introduced in section 4.3.

In successful repositories, PCR typically starts near zero and increases steadily as more peripheral contributors begin submitting and successfully merging pull requests. In contrast, unsuccessful repositories either show a flat trajectory with little peripheral

activity or only short-lived spikes that do not persist. For instance, in a successful repository like `mozilla/bigquery-etl` from Figure 9, window A PCR is around 0.28 and increases to approximately 0.51 in window B, whereas in a non-successful repository, `google/site-kit-wp`, the PCR remains below 0.09 in both windows. Even though the successful repository did not pass the `baseline_pcr_ratio_max` threshold of the window builder algorithm and is consequently only selected as a fallback window combination, it still shows a significant increase in window B PCR and is therefore classified as successful by the 75th-quartile approach. This behaviour supports the use of PCR as a target variable and as a core indicator of cross-silo collaboration in this thesis. The contributor funnel width metric tracks the daily or monthly influx of new contributors who have not previously participated in the repository. Figure 10 compares the funnel width distributions for the top- and bottom-quartile repositories by window B PCR. The top-quartile group exhibits higher and more sustained inflows of new peripheral contributors, leading to a more constant low core/peripheral ratio over time. In contrast, the bottom quartile shows sporadic growth and slower adaptation to a stable low core/peripheral ratio. This pattern aligns with the expectation that successful Inner Source initiatives attract and retain a broader contributor base. Review density and time-to-merge are used to characterise how the core team interacts with both core and peripheral contributions.

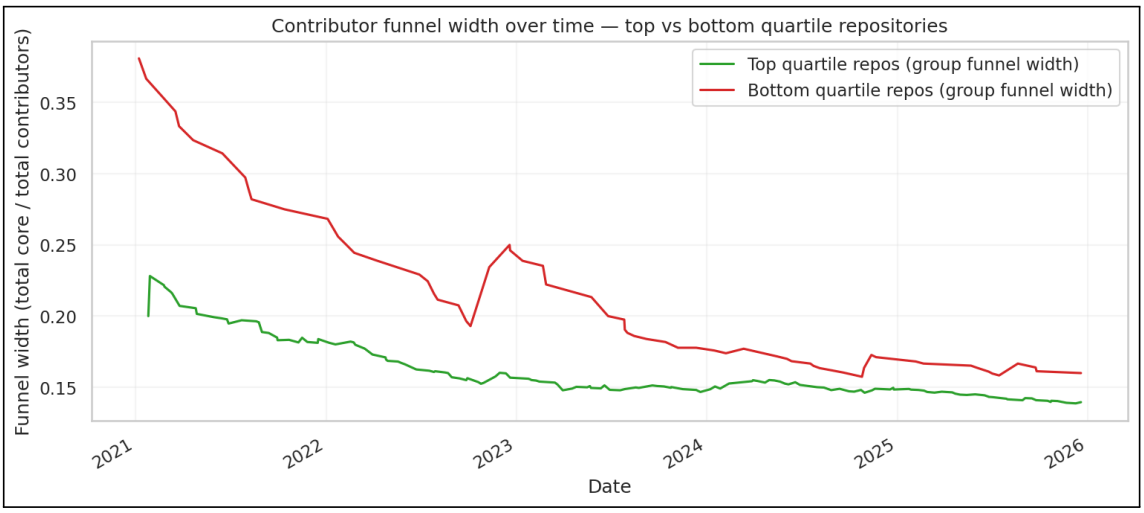

Figure 10: Funnel width for classified top and bottom quartile repositories: Time series depicting the average cumulative new core/peripheral contributors ratio based on the success classification using window B PCR.

Table 9 in the appendix summarises the median review density and time-to-merge values for core and peripheral pull requests across the evaluation dataset. In the majority of repositories, peripheral pull requests receive about 64% more review comments than core pull requests. Given that a more in-depth review process is in place, the median time-to-merge for peripheral contributions is also 66% higher than for core contributions. This suggests that, where Inner Source practices are emerging, core teams are actively engaging with peripheral contributors rather than systematically

deprioritising their work. The fork-to-star ratio provides a coarse indication of how many users are willing to invest effort in working with the codebase relative to its overall visibility. Across the dataset, successful repositories (top-quartile window B PCR) tend to have slightly higher fork-to-star ratios than unsuccessful repositories (median values of 0.4159 vs 0.2041), although the effect size is modest. This is consistent with the interpretation of the ratio as a weak but complementary signal of architectural modularity and contribution attractiveness. Altogether, the observed patterns in PCR, contributor funnel width, review density, time-to-merge, and fork-to-star ratio are consistent with theoretical expectations for Inner Source collaboration and support their use as features and targets in the subsequent ML models.

7.5 Machine Learning Model Evaluation

In this step, the predictive performance of the ML model is evaluated to answer EQ3.

Random Forest Regression

The RF regressor estimates the future PCR in window B from baseline features in window A. As described in section 4.5, model selection was performed via CV over tree depth and tree count, using RMSE as the optimisation criterion. Table 10 in the appendix summarises the CV results on the training set. The best configuration in terms of RMSE was obtained with a maximum depth of 10 and 200 trees, yielding an RMSE of approximately 0.0582 on the CV folds. For the final cross-project model, performance was evaluated on an 11-repository test set (17 repositories in the training set). Table 3 reports the resulting RMSE, MAE, and R^2 values.

Table 3: Random Forest regression performance on the held-out test set.

Model	RMSE	MAE	R^2
Random Forest (cross-project)	0.0595	0.0372	0.8802
Naive mean predictor	–	–	0.0

The RMSE of 0.0595 is small in absolute terms, and the R^2 value of 0.8802 indicates that, on this small test set, the model outperforms a naive predictor that always returns the mean window B PCR by far. A closer inspection of the predictions (see Table 4) shows that the RF regressor's predicted values are accurate across several repository windows, however, deviating in a few others.

Table 4: Exemplary extract of all RF predictions compared to true window B PCR.

Project	Repository	Prediction	True PCR	Window A End	Window B End
adobe	react-spectrum	0.3199	0.3025	21	27
adobe	spectrum-web-components	0.5562	0.7053	56	62
cloudflare	cloudflare-docs	0.3519	0.3403	37	43
google	android-cuttlefish	0.2794	0.4122	35	41
google	perfetto	0.313	0.2906	42	48
intel	llvm	0.45	0.473	65	71
jetbrains	kotlin	0.4393	0.6346	34	40
microsoft	vscode	0.3779	0.3766	41	47
mozilla	bigquery-etl	0.798	0.8525	28	34
nvidia	TensorRT-LLM	0.4961	0.47	17	23

For example, for `microsoft/vscode` the model predicts a window B PCR of 0.3779 compared to the true value of 0.3766, whereas for `jetbrains/kotlin` it predicts 0.4393 versus an actual 0.6346, illustrating both close and less accurate cases within the small test set. Given that the cross-project model is trained on only 17 repositories and evaluated on 11, and that window B PCR is influenced by many unobserved organisational factors (e.g., internal funding decisions, management priorities, and team restructuring), it is surprising that the RF regressor not only achieves the R^2 threshold of 0.2 defined in section 4.5, but exhibits a performance not far from a perfect (R^2 of 1) fit. Considering the resulting R^2 score and the MAE of 0.0372, the model's evaluation indicates strong adaptation to the training datasets (overfitting), which explains these good metrics. Given the data constraints, the resulting model should be interpreted more as an exploratory tool rather than a precise forecaster. Still, the RF's metrics suggest that the model can provide useful information about the relative importance of baseline metrics. Table 5 lists the normalised feature importances for the cross-project model.

Table 5: Normalised feature importances from the cross-project RF model.

Feature	Importance
avg_comments_core_prs	0.0873
total_core_contributors	0.3096
fork_count	0.0773
merged_core_prs	0.1036
stargazer_count	0.1977
fork_to_star_ratio	0.0946
avg_hours_to_merge_core	0.1299

These importances suggest that core review practices (time-to-merge), team structure (especially number of core contributors), and repository visibility (stars) are among the most influential baseline indicators of future peripheral collaboration. Even though the overall predictive performance of the RF is questionable, these patterns provide hints about which aspects of a repository the managers should focus on when preparing for Inner Source adoption.

Logistic Regression and SVM

Regression models with L1 (Lasso) and L2 (Ridge) regularisation are used to classify repositories into successful and non-successful Inner Source adoption groups based on the binary label derived from window B PCR (section 4.3). Hyperparameter tuning was performed over `reg_param` and `max_iter` using AUC as the optimisation metric. The results (see Table 11) suggest an optimal configuration in hyperparameters in choosing a small regularisation parameter (≈ 0.01) for both L1 and L2 regularisation. This yields an AUC of approximately 0.74, regardless of the selected maximum iterations. A higher regularisation penalty quickly leads to a decrease in the target metric, indicating that oversimplified models perform worse.

Table 6: Performance of L1- and L2-regularised LR models on test samples.

Model	Accuracy	AUC	F1	Precision	Recall
Lasso	0.7532	0.7905	0.6472	0.5674	0.7532
Ridge	0.7576	0.8058	0.6648	0.736	0.7576

Table 6 reports the performance of the LR models using Lasso and Ridge regularisation on a data set of 720 samples (489 training set / 231 test set). Both models achieve AUC values around 0.8 and F1-scores around 0.65 on the test set, which meets the acceptance thresholds defined in section 4.5 (AUC > 0.7 and F1 ≥ 0.6). This indicates a moderate ability to discriminate between successful and non-successful repositories based on the baseline features. Table 12 in the appendix lists the standardised coefficients learned by the best L1 and L2 models. The Lasso model retains a relatively small subset of metrics with non-zero weights, highlighting `total_core_contributors`, `avg_hours_to_merge_core`, `merged_core_prs`, `fork_count`, and `fork_to_star_ratio` as the most informative baselines. The Ridge model distributes weight more broadly but shows a similar pattern. These coefficients are consistent with the intuition that a sufficiently sized core team, active review practices, and a balanced fork-to-star relationship support Inner Source adoption. In contrast, slow integration of core changes and a high raw fork count can signal coordination or maintainability issues.

To validate the robustness of the LR findings, the results of the Linear SVM with class-weight balancing are evaluated. Table 7 summarises the performance of the best SVM

configuration (`reg_param` = 0.01) on the tested samples (same as for LR: 720 samples: 489 training / 231 test).

Table 7: Performance of the Linear SVM classifier on test samples.

Model	Accuracy	AUC	F1	Precision	Recall
LinearSVC	0.5801	0.8289	0.5967	0.8346	0.5801

With an AUC of 0.8289, the SVM slightly outperforms the LR models in overall discriminative ability, exceeding the acceptance threshold defined in section 4.5. While the raw accuracy and F1-score (0.5967) are lower than the L1 and L2 models, driven by a conservative recall (0.5801) but an exceptionally high precision (0.8346), the strong AUC confirms that the baseline features contain robust signals for predicting Inner Source adoption. The coefficients learned by the SVM closely mirror the feature coefficients identified by the Lasso and Ridge models. The SVM assigns the highest absolute weights to the same metrics (see Table 13). This structural alignment across the two fundamentally different algorithms supports the earlier conclusion: successful Inner Source adoption is supported by adequately sized core teams and rapid code integration, whereas high repository fragmentation (forks) and bottlenecked core maintainers tend to hinder success.

Clustering and ARIMA

To complement the supervised models, unsupervised clustering was applied to time series of PCRs to identify typical patterns of Inner Source adoption across repositories. Using the feature representation described in section 4.4, a K-means configuration with $k = 2$ was selected. Assigning all 31 repositories to two distinct archetypes yielded a silhouette score of 0.59, indicating reasonably well-separated groups. More clusters, therefore, did not produce robust separation. Cluster 0 is characterised by comparatively high and sustained peripheral activity and is labelled the 'community catalyst' archetype. Repositories in this cluster exhibit higher recent PCRs (0.37 on average) and a higher success rate of 0.50 for adopting Inner Source. Example repositories include well-known, visible projects such as `intel/llvm`, `jetbrains/kotlin` or `microsoft/vscode`. These projects tend to combine a strong core team presence with frequent peripheral pull requests, and they often provide good contribution processes and documentation. This pattern aligns with the earlier finding that a sufficiently sized, active core team, together with engaged review practices, supports Inner Source adoption. Cluster 1 is labelled the 'legacy bottleneck' archetype. It contains the majority of the repositories (25 out of 31) and is characterised by a lower recent PCR (0.28 on average) and a lower success rate (0.20). Exemplary repositories are `adobe/react-spectrum`, `google/perfetto` and `sap/spartacus`. While these repositories are often highly visible and technically central within their respective organisation, they show slower peripheral uptake. This finding, along with the baseline

feature analysis, suggests that visibility alone is insufficient for Inner Source success. Given the small number of repositories, the two clusters should be viewed purely as descriptive groupings rather than as stable archetypes.
In addition to the clustering, ARIMA models were fitted to the PCR time series from the repositories. For each selected repository, the ARIMA order (p, d, q) was chosen via standard information criteria (AIC/BIC). Models were trained on most of the time series and evaluated on a small holdout window before producing 12-period forecasts. Across the forecasted repositories, the in-sample performance is good, with MAE and RMSE values of around 0.01. The holdout performance remains competitive across most projects. Overall, only some repositories show fluctuations in their PCR forecasts, with MAE values exceeding 0.05, suggesting that their peripheral contribution dynamics are more irregular or too complex to be captured by an autoregressive model. For example, in the stable `google/perfetto` repository (see Figure 8), the 6-month holdout MAE is approximately 0.0025, while in the more volatile `sap/spartacus` repository, it reaches about 0.0733, reflecting much noisier peripheral contribution dynamics. Those less predictable projects are often part of the 'legacy bottleneck' cluster, which reinforces the clustering interpretation.

7.6 Dashboard and Usability

This evaluation step examines whether the Inner Source dashboard fulfils its intended role as a decision-support interface for managers (EQ4). Instead of a formal user study, a heuristic-based walkthrough was conducted to compare the implemented dashboard with the conceptual design in chapter 4. The interface connects technical model outputs and managerial levers: LR coefficients, RF feature importances, and ARIMA forecasts are presented through sliders, impact badges, archetype labels, and adoption scores rather than as raw numerical tables. By focusing the controls on core-side levers (e.g., core contributors, review density, time-to-merge) and omitting direct manipulation of peripheral activity, the dashboard also makes the 'cold-start' assumption clear, supporting the theoretical stance that peripheral participation should emerge from improvements in the core team. Impact badges are computed relative to the magnitudes of the coefficients or importances of the active model (e.g., Lasso LR vs RF), and scenario outcomes are visually distinguished with ARIMA-derived baseline forecasts, allowing managers to compare 'doing nothing' with specific interventions (slider adjustments) in a single view. At the same time, the dashboard embodies several pragmatic trade-offs. To maintain responsiveness, the scenario evaluator approximates RF effects linearly on the client side instead of re-running the full model in the backend for every slider change, so scenario impacts should be interpreted as indicative rather than exact. The interface further assumes a relatively technical audience, surfacing terms such as PCR, ARIMA, and confidence intervals with only lightweight explanations, which may require additional technical knowledge for managerial users. Minor synchronisation delay can occur because the ML models are trained on windowed baseline data, whereas the dashboard's descriptive statistics may reflect more recent repository activity. Finally, the scenario evaluator conceptually treats peripheral col-

laboration as emerging from a low baseline, even though many repositories already exhibit non-zero PCR; in such cases, the sliders project how existing collaboration might scale rather than creating it from scratch. Despite these limitations, the dashboard demonstrates that non-trivial ML models can be exposed in a form that supports interactive 'what-if' reasoning, thereby contributing to the artefact's overall value as a decision-support tool for Inner Source adoption.

7.7 Threats to Validity

Evaluating the thesis, several threats to the validity of the findings have to be considered. Starting with construct validity, the first issue to note is that the thesis uses PCR within a specific time window as the main indicator of Inner Source adoption. This may not fully capture qualitative aspects such as knowledge sharing, documentation quality, or long-term cultural change, so success might be only partially reflected. Since the different time windows for one repository can overlap, temporal dependence in the derived features cannot be ruled out. Further, core team structure, review latency, forks, and stars are treated as indicators of collaboration readiness and repository visibility, even though these metrics can be imperfect and are not always representative. Regarding the interpretation of the cluster archetypes, it should be noted that the cluster labels are a qualitative interpretation. They help discuss patterns but may also oversimplify underlying organisational technical contexts. Evaluating the metrics obtained from the ML approaches, it should also be noted that meeting the predefined thresholds in section 4.5 does not automatically yield an excellent model. Since the sample sizes are small and confidence intervals are not exhaustively reported, the measured effects may be of less importance than the point estimates suggest. Continuing with the internal validity, there are still many unobserved factors, such as organisational policies or release schedules, that are likely to influence Inner Source adoption but are not included in the models. This limits the ability to assign causality to the observed relationships. Regarding the data on which the evaluation is based, it is worth mentioning that only a subset of prominent OS repositories from a few organisations is analysed. Their characteristics, such as size, popularity, and company culture, may confuse the relationship between baseline metrics and Inner Source outcomes. Next, although regularisation and evaluation are used, the small sample size of 31 repositories increases the risk that some findings are specific to this dataset, as the models may overfit to their training data. In terms of external validity, the studied repositories originate from large companies, which limits the general applicability of the results to smaller organisations or domains with different contribution patterns. Since public GitHub repositories are used as proxies for Inner Source-like collaboration, it is important to note that real Inner Source programs may differ in terms of contributor motivations, governance, and access controls. Transferring the findings directly to closed-source contexts is therefore not recommended.
Taken together, these threats imply that the artefact is better suited for supporting qualitative reasoning about Inner Source readiness than for making precise quantitative predictions.

8 Conclusion

The overarching goal of this thesis was to address a practical and methodological gap in the adoption of Inner Source. While many organisations collect descriptive analytics about their software development processes, managers often lack predictive, data-driven tools to assess whether a repository is ready for Inner Source and how collaboration is likely to evolve. Building on the MECOIS platform and the literature on OS, Inner Source, and software metrics, this work designed, implemented, and evaluated a predictive decision-support artefact that estimates Inner Source success from version control data. The artefact simulates Inner Source behaviour by proxying OSS repository data, applying the onion model, introducing a patch-flow-inspired PCR, and using a temporal window builder to construct pre- and post-adoption time windows as inputs for a ML ensemble integrated into the MECOIS platform.

Answers to the Research Questions

RQ1: How can Inner Source success be defined and measured?

The thesis answers this first research question by defining a success metric based on collaboration rather than raw productivity. First, the onion model was applied to the OS proxy data to distinguish core and peripheral contributors dynamically over time. Contributors were classified as core if they fell within the top 75% of cumulative contributions or contributed 5% or more of total contributions, with temporal snapshots ensuring that role transitions were captured realistically. This enabled a separation of architectural stability (core) from scalability and innovation (peripheral). Second, building on Capraro's patch-flow concept, the PCR was introduced as the primary success metric. PCR measures the fraction of merged commits and pull requests that originate from peripheral contributors within a given window, thereby capturing both the flow and integration of cross-silo contributions. By focusing on merged pull requests, the metric avoids counting unreviewed or rejected contributions and aligns with the view that a successful contribution arises when external changes become part of the codebase. Third, to obtain a binary success label suitable for classification, the thesis adopted a percentile-based thresholding strategy. Taken together, these steps provide a concept of Inner Source success that (i) focuses on collaboration across organisational boundaries, (ii) is resistant to simple metric tuning, and (iii) can be applied as both a continuous target (PCR) and a binary label for predictive modelling.

RQ2: To what extent can ML models predict the success of Inner Source initiatives based on historical collaboration data extracted from VCS?

The second research question investigated how far ML models can go in predicting Inner Source success under realistic data constraints. The evaluation results show that, given the available dataset, predictive performance is moderate and practically useful. On the small test set, the RF regressor achieves a low MAE and a high R^2 compared to a naive mean baseline. However, given that these results are based on only 11 test repositories, they should be interpreted as an upper bound under favourable conditions

rather than as robust evidence of generalisable performance. The LR classification models with L1- and L2-regularisation achieve good metrics in AUC and F1-score, thereby meeting the acceptance thresholds defined in section 4.5. Within the limits of the available data, these results indicate a moderate but practically useful ability to discriminate between successful and unsuccessful repositories based on baseline features. Despite the SVM performing slightly better, LR was preferred in the dashboard because it exposes coefficients that can be translated more directly into managerial levers. Beyond the supervised models, K-means clustering produces repository groups that can be mapped to interpretable descriptive groupings (archetypes), and ARIMA models provide stable short-term trajectories for PCR across many repositories, and are mainly valuable for visualising trends. Despite these data limitations, the models achieve moderate predictive accuracy and still capture collaboration patterns that appear useful for decision support.

This thesis makes methodological, technical, and practical contributions. It proposes the PCR as a success metric for Inner Source, based on the concepts of the onion model and the patch-flow. PCR focuses on the proportion of integrated work originating from peripheral contributors and thereby captures a core aspect of Inner Source collaboration. It introduces a percentile-based success-labelling strategy that derives binary success labels from the observed distribution of PCR. Further, the thesis presents a temporal window builder algorithm that simulates pre- and post-adoption phases on OSS repositories. Lastly, it shows how identity resolution, drive-by filtering, and strict enrolment filtering can be combined to approximate Inner Source collaboration from public GitHub data using SortingHat and GitHub organisation membership information. On the technical side, it extends the MECOIS platform with a GitHub data pipeline using REST and GraphQL APIs, a medallion-style data lake, and enhanced identity management that includes organisational enrolments and bot labelling. It implements a reusable ML backend that integrates an ensemble of algorithms into MECOIS, with hyperparameter tuning and persisting model output in Supabase. It develops a client-side scenario evaluator that leverages existing ML models, providing immediate what-if feedback on the dashboard without re-running the backend models. On a practical level, the thesis delivers a prototype Inner Source prediction dashboard that extends MECOIS's descriptive capabilities with archetype labelling, success estimates, collaboration forecasts, and interactive scenario analysis. It identifies which baseline characteristics are most influential for sustaining peripheral contributions in the studied repositories. It also documents practical challenges in applying Inner Source analytics to OSS data, including rate limits, aliasing, bot detection, and the scarcity of long-running Inner Source-like collaboration patterns.

The evaluation suggests several practical recommendations. When considering Inner Source adoption, the results suggest that success is less about raw volume of commits and more about sustained, cross-silo collaboration. Metrics such as PCR, contributor funnel width, review density, and time-to-merge provide deeper insights into the repository's health compared to simple contribution counts alone. In particular, the feature importance analysis indicates that investing in responsive, transparent review processes for peripheral contributors, maintaining a sufficiently sized and active core

team, and ensuring that repositories are visible and approachable can all contribute to higher levels of peripheral participation. The implementation demonstrates that, even though such predictive capabilities can be layered on top of existing projects like ME-COIS without disrupting baseline functionality, the quality of upstream data engineering is at least as important as the choice of ML algorithms. Integrating domain-specific logic, such as the onion model classification and drive-by filtering, is essential to aligning predictive outcomes with Inner Source concepts. Finally, the models' moderate but context-dependent predictive performance reinforces the view that such tools should be positioned as decision support rather than decision automation. Predictions and archetypes can help managers prioritise repositories for Inner Source adoption and identify potential bottlenecks, but qualitative reviews and domain knowledge should complement them.

Several limitations constrain the conclusions which can be drawn from this work. The evaluation is based on a relatively small dataset of 31 corporate-sponsored OSS repositories from 10 organisations, with valid baseline and target windows found for only 14 of them. This limited sample size, combined with noise in the target variable, limits the full potential of the ML models and introduces overfitting. Further, the Inner Source proxy is necessarily approximate. OSS projects differ from proprietary Inner Source environments in contributor motivation, visibility, and governance. As a result, the models may miss key explanatory variables, and their generalisability to proprietary Inner Source settings is limited. The feature set, while richer than simple commit counts, is still relatively coarse. It does not incorporate network-structural properties (e.g., contributor collaboration graphs and code ownership networks) or specific process metrics (e.g., review queues and test coverage). PCR itself depends on threshold choices for core/peripheral classification and drive-by contributors, which, although motivated by the literature and empirically tuned, remain pragmatic. The limited training data hinders the ability of the RF and classification models to learn robust patterns. In addition, ARIMA models are sensitive to discontinuous time series and perform best for short-term horizons, limiting their usefulness for long-range planning. Because the dataset focuses on highly active, corporate-sponsored OSS repositories, it is unclear how well the findings transfer to low-activity projects, purely community-driven OSS, or proprietary Inner Source repositories. The lack of direct access to internal Inner Source initiatives also prevents validation of the artefact against real-world adoption outcomes in enterprises.

The limitations outlined above point to several directions for future research and development. First, the most direct extension is to apply the artefact to genuine Inner Source repositories within one or more organisations, once data access is possible. This would allow both the OSS proxy and the PCR-based success definition to be validated and refined against real adoption processes, and would provide improved data for training predictive models. Second, the dataset could be expanded in breadth and depth. Including more repositories and longer time spans would increase statistical power and enable experimentation with more refined models, such as gradient-boosted trees or temporal graph neural networks that use collaboration networks. At the metric level, incorporating structural and network-based features (e.g., central-

ity measures, code ownership dispersion) could capture interaction patterns that are invisible to aggregate metrics. Third, a concrete next step would be to combine PCR with at least one qualitative indicator, such as survey-based assessments of community health, and test whether this improves predictive performance. This would support multi-dimensional success labels and more specific analyses of trade-offs between throughput, stability, and collaboration. Finally, from a tooling and usability perspective, the MECOIS Inner Source dashboard could be evaluated in formal user studies with engineering managers and technical leads. Such an evaluation could assess how the predictions, archetypes, and scenario sliders influence real decision-making and inform improvements to the user interface, explanation mechanisms, and recommended actions (e.g., suggesting concrete steps to reduce time-to-merge peripheral pull requests).

This thesis has shown that it is possible, but challenging, to apply predictive analytics to Inner Source adoption using only version-control data and an OSS-based proxy. While limited data and noise constrain the predictive models developed here, they nonetheless achieve moderate predictive performance on the available dataset. The work establishes a methodological and technical foundation for combining Inner Source theory, software metrics, and ML within an operational analytics platform. As organisations continue to combine open and inner collaboration, similar artefacts may help managers base Inner Source decisions more on observed collaboration patterns and less on intuition alone.

A Appendix

A.1 Solution Design

```
data/data-lake/tables/
├── {project}/
│   ├── github_commits/
│   │   ├── bronze/ (raw JSON files)
│   │   └── silver/ (normalised, Delta format)
│   ├── github_pull_requests/
│   │   ├── bronze/
│   │   └── silver/
│   └── {metric_tables}/
│       └── gold/ (computed metrics)
├── _clustering/ (cross-project models)
├── _logistic_regression/
├── _random_forest/
├── _svm/
└── _arima/
```

Figure 11: Data Lake Structure: The exemplary structure of the data lake, containing extracted data from multiple repositories and calculated metrics in all granularities (bronze, silver and gold). Additionally, the ML models' output is saved in the corresponding algorithm folder.

```yaml
DataExtraction:
  Commits:
    per_page: 100
    since: '2020-01-01'
    until: '2025-12-31'

  PullRequests:
    since: '2020-01-01'
    until: '2025-12-31'
    pr_page_size: 40
    participants_page_size: 20
    commits_page_size: 25

InnerSourceMetrics:
  minimum_contributions_threshold: 3
  early_classification_batch_days: 45

TemporalSplit:
  baseline_months: 12
  target_months: 6
  success_percentile: 0.75

```

```
Clustering:
  projects: adobe, cloudflare, google, ibm, intel, jetbrains,
   microsoft, mozilla, nvidia, sap
  k_min: 2
  k_max: 6

RandomForest:
  projects: adobe, cloudflare, google, ibm, intel, jetbrains,
   microsoft, mozilla, nvidia, sap
  target_column: target_window_pcr
  num_trees: 100
  max_depth: 5
  test_split: 0.3
  tuning_enabled: true
  tune_num_trees: 50, 100, 200
  tune_max_depth: 3, 5, 10
  cv_num_folds: 5
  cv_parallelism: 4

LogisticRegression:
  projects: adobe, cloudflare, google, ibm, intel, jetbrains,
   microsoft, mozilla, nvidia, sap
  target_column: target_window_pcr
  threshold: 0.75
  reg_param: 0.1
  max_iter: 100
  test_split: 0.3
  tuning_enabled: true
  tune_max_iters: 100, 200
  tune_reg_params: 0.01, 0.1, 1.0, 10.0
  cv_num_folds: 5
  cv_parallelism: 4

SVM:
  projects: adobe, cloudflare, google, ibm, intel, jetbrains,
   microsoft, mozilla, nvidia, sap
  test_split: 0.3
  tuning_enabled: true
  tune_reg_params: 0.01, 0.05, 0.1, 10
  tune_max_iters: 100, 200
  cv_num_folds: 5
  cv_parallelism: 4

ARIMA:
  projects: adobe, cloudflare, google, ibm, intel, jetbrains,
   microsoft, mozilla, nvidia, sap
  target_column: peripheral_contribution_rate
  forecast_periods: 12
  auto_select_order: true
  p_candidates: 0, 1, 2, 3
  d_candidates: 0, 1
  q_candidates: 0, 1, 2, 3
```

Listing 1: Exemplary configuration of the entire Inner Source prediction module.

A.2 Demonstration

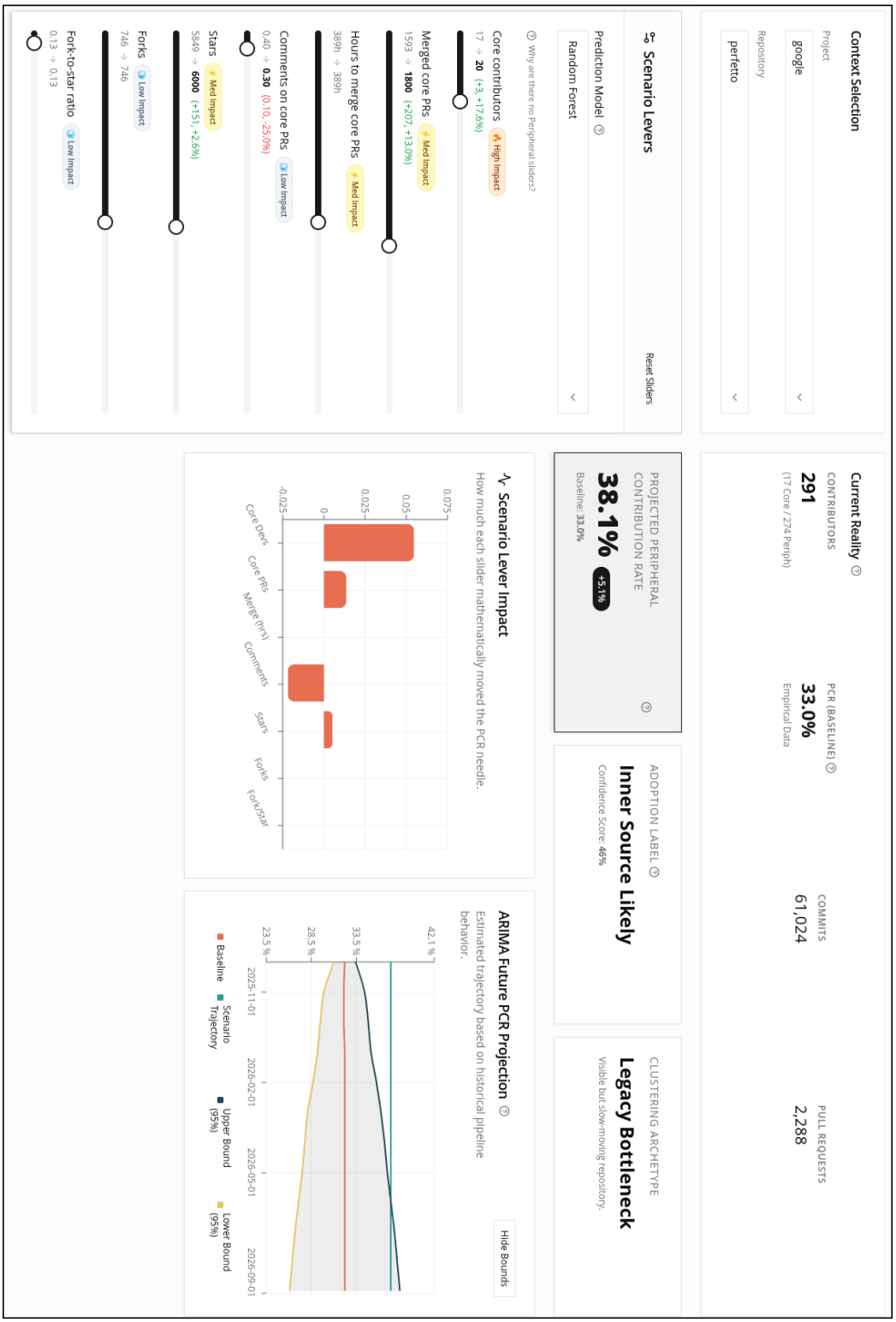

Figure 12: Dashboard - Overview: An overview of the dashboard, including all information, settings and visualisation cards.

A.3 Evaluation

Table 8: Coverage of valid window A/B splits across repositories.

Category	Count	Percentage
Repositories with valid windows	14	45.16
Repositories failing duration constraints	3	9.68
Repositories failing maximum baseline PCR rate	5	16.13
Repositories failing peripheral-rise constraints	9	29.03

Table 9: Median review density (comments per pull request and time-to-merge grouped by pull request type.

Pull Request Type	Review Density	Time-to-Merge (hours)
Core PRs	1.0700	17.97
Peripheral PRs	1.7500	23.95

Table 10: Random Forest cross-validation results on the training set.

maxDepth	numTrees	RMSE
3	50	0.1047
5	50	0.078
10	50	0.0594
3	100	0.1045
5	100	0.0786
10	100	0.0587
3	200	0.1046
5	200	0.078
10	200	0.0582

Table 11: Logistic Regression cross-validation results for Lasso (L1) and Ridge (L2) regularisation on the training set.

Model	Regularisation Parameter	Maximum Iterations	AUC
Lasso	0.01	100	0.7671
Lasso	0.01	200	0.7671
Lasso	0.1	100	0.504
Lasso	0.1	200	0.504
Lasso	1.0	100	0.5
Lasso	1.0	200	0.5
Lasso	10.0	100	0.5
Lasso	10.0	200	0.5
Ridge	0.01	100	0.7667
Ridge	0.01	200	0.7667
Ridge	0.1	100	0.7514
Ridge	0.1	200	0.7514
Ridge	1.0	100	0.7407
Ridge	1.0	200	0.7407
Ridge	10.0	100	0.7308
Ridge	10.0	200	0.7308

Table 12: Standardised Logistic Regression coefficients for L1 and L2 regularisation.

Model	Feature	Coefficient
Lasso	total_core_contributors	0.0998
Lasso	avg_comments_core_prs	0.0
Lasso	avg_hours_to_merge_core	-0.8222
Lasso	merged_core_prs	-0.3591
Lasso	fork_count	-1.1782
Lasso	stargazer_count	0.0
Lasso	fork_to_star_ratio	0.0279
Ridge	total_core_contributors	0.1999
Ridge	avg_comments_core_prs	0.0456
Ridge	avg_hours_to_merge_core	-0.8152
Ridge	merged_core_prs	-0.5986
Ridge	fork_count	-1.2442
Ridge	stargazer_count	0.0035
Ridge	fork_to_star_ratio	0.1502

Table 13: Linear Support Vector Machine coefficients.

Feature	Coefficient
total_core_contributors	0.5142
avg_comments_core_prs	0.0238
avg_hours_to_merge_core	-0.8073
merged_core_prs	-0.7199
fork_count	-2.1082
stargazer_count	-0.1748
fork_to_star_ratio	-0.0382

References

[1] N. Wirth, "A brief history of software engineering," *IEEE Annals of the History of Computing*, vol. 30, no. 3, pp. 32–39, 2008.

[2] S. Weber, *The Success of Open Source*. Cambridge, MA and London, England: Harvard University Press, 2004. [Online]. Available: https://doi.org/10.4159/9780674044999

[3] C. M. Kelty, *Two Bits*. Durham, NC, USA: Duke University Press, 2008. [Online]. Available: https://doi.org/10.1515/9780822389002

[4] E. Raymond, "The cathedral and the bazaar," *Know Techn Pol*, vol. 12, no. 3, pp. 23–49, Sep. 1999. [Online]. Available: https://doi.org/10.1007/s12130-999-1026-0

[5] D. Cooper and K.-J. Stol, *Adopting InnerSource*. Sebastopol, CA, USA: O'Reilly Media, 2018. [Online]. Available: https://www.oreilly.com/library/view/adopting-innersource/9781492041863/

[6] M. Capraro, M. Dorner, and D. Riehle, "The patch-flow method for measuring inner source collaboration," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 515–525. [Online]. Available: https://doi.org/10.1145/3196398.3196417

[7] X. Chen, M. Usman, and D. Badampudi, "Using innersource for improving internal reuse: An industrial case study," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 348–357. [Online]. Available: https://doi.org/10.1145/3593434.3593466

[8] K.-J. Stol, M. Schaarschmidt, and S. Goldblit, "Gamification in software engineering: the mediating role of developer engagement and job satisfaction," *Empir Software Eng*, vol. 27, no. 2, p. 35, Dec. 2021. [Online]. Available: https://doi.org/10.1007/s10664-021-10062-w

[9] K.-J. Stol, P. Avgeriou, M. A. Babar, Y. Lucas, and B. Fitzgerald, "Key factors for adopting inner source," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 2, Apr. 2014. [Online]. Available: https://doi.org/10.1145/2533685

[10] M. Capraro, "Measuring Inner Source Collaboration," Ph.D. dissertation, Friedrich-Alexander-Univ. Erlangen-Nürnberg, 2020.

[11] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, "Evolution patterns of open-source software systems and communities," in *Proceedings of the International Workshop on Principles of Software Evolution*, ser. IWPSE '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 76–85. [Online]. Available: https://doi.org/10.1145/512035.512055

[12] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, p. 309–346, Jul. 2002. [Online]. Available: https://doi.org/10.1145/567793.567795

[13] M. Aberdour, "Achieving quality in open-source software," *IEEE Software*, vol. 24, no. 1, pp. 58–64, Jan.-Feb. 2007. [Online]. Available: https://doi.org/10.1109/MS.2007.2

[14] T. K. Ho, "Random decision forests," in *Proceedings of 3rd International Conference on Document Analysis and Recognition*, ser. ICDAR '95, Montreal, QC, Canada, 1995, pp. 278–282 vol.1. [Online]. Available: http://doi.org/10.1109/ICDAR.1995.598994

[15] D. W. Hosmer Jr., S. Lemeshow, and R. X. Sturdivant, *Applied Logistic Regression*, 3rd ed. Hoboken, NJ, USA: Wiley, 2013. [Online]. Available: https://learning.oreilly.com/library/view/applied-logistic-regression/9781118548356/

[16] H. M. Nayem, S. Aziz, and B. M. G. Kibria, "Evaluating estimator performance under multicollinearity: A trade-off between mse and accuracy in logistic, lasso, elastic net, and ridge regression with varying penalty parameters," *Stats*, vol. 8, no. 2, May 2025. [Online]. Available: https://www.mdpi.com/2571-905X/8/2/45

[17] C. Cortes and V. Vapnik, "Support-vector networks," *Mach Learn*, vol. 20, no. 3, pp. 273–297, Sep. 1995. [Online]. Available: https://doi.org/10.1007/BF00994018

[18] A. Shadab, S. Said, and S. Ahmad, "Box–Jenkins multiplicative ARIMA modeling for prediction of solar radiation: a case study," *Int J Energ Water Res*, vol. 3, no. 4, pp. 305–318, Dec. 2019. [Online]. Available: https://doi.org/10.1007/s42108-019-00037-5

[19] MECOIS. Accessed: May. 05, 2026. [Online]. Available: https://www.mecois.com/

[20] J. Nam and S. Kim, "Clami: Defect prediction on unlabeled datasets (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15, Lincoln, NE, USA, 2015, pp. 452–463. [Online]. Available: http://doi.org/10.1109/ASE.2015.56

[21] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10, Timisoara, Romania, 2010, pp. 1–10. [Online]. Available: https://ieeexplore.ieee.org/document/5609747

[22] N. Forsgren, J. Humble, and G. Kim, *Accelerate: The Science of Lean Software and DevOps Building and Scaling High Performing Technology*

Organizations, 1st ed. IT Revolution Press, 2018. [Online]. Available: https://learning.oreilly.com/library/view/accelerate/9781457191435/?ar=

[23] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, Dec. 2007. [Online]. Available: http://www.jstor.org/stable/40398896

[24] State of InnerSource Survey 2024. Accessed: May. 05, 2026. [Online]. Available: https://innersourcecommons.org/learn/research/state-of-innersource-survey-2024/

[25] Home - CHAOSS — chaoss.community. Accessed: May. 05, 2026. [Online]. Available: https://chaoss.community/

[26] GrimoireLab - Software Development and Community Analytics platform — chaoss.github.io. Accessed: May. 05, 2026. [Online]. Available: https://chaoss.github.io/grimoirelab/

[27] G. Robles, J. M. Gonzalez-Barahona, and I. Herraiz, "Evolution of the core team of developers in libre software projects," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09, Vancouver, BC, Canada, 2009, pp. 167–170. [Online]. Available: http://doi.org/10.1109/MSR.2009.5069497

[28] G. Pinto, I. Steinmacher, and M. A. Gerosa, "More Common Than You Think: An In-depth Study of Casual Contributors," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER '16, Osaka, Japan, 2016, pp. 112–123. [Online]. Available: https://doi.org/10.1109/SANER.2016.68

[29] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 91–100. [Online]. Available: https://doi.org/10.1145/1595696.1595713

[30] R. Pham, L. Singer, O. Liskin, F. F. Filho, and K. Schneider, "Creating a shared understanding of testing culture on a social coding site," in *2013 35th International Conference on Software Engineering*, San Francisco, CA, USA, 2013, pp. 112–121. [Online]. Available: http://doi.org/10.1109/ICSE.2013.6606557

[31] M. Zhou and A. Mockus, "What make long term contributors: Willingness and opportunity in OSS community," in *2012 34th International Conference on Software Engineering*, Zurich, Switzerland, 2012, pp. 518–528. [Online]. Available: http://doi.org/10.1109/ICSE.2012.6227164

[32] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 137–143. [Online]. Available: https://doi.org/10.1145/1137983.1138016

[33] M. Strathern, "'Improving ratings': audit in the British University system," *European Review*, vol. 5, no. 3, pp. 305–321, Jul. 1997. [Online]. Available: http://doi.org/10.1002/(SICI)1234-981X(199707)5:3<305::AID-EURO184>3.0.CO;2-4

[34] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach*, 3rd ed. Boca Raton, FL, USA: CRC Press, 2014. [Online]. Available: https://learning.oreilly.com/library/view/software-metrics-3rd/9781439838228/?ar=

[35] I. Sommerville, *Software engineering*, 10th ed. Harlow, England: Pearson Education, 2016.

[36] DORA | DORA Guides. Accessed: May. 05, 2026. [Online]. Available: https://dora.dev

[37] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: the contributor's perspective," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 285–296. [Online]. Available: https://doi.org/10.1145/2884781.2884826

[38] chaoss/grimoirelab-sortinghat. Accessed: May. 05, 2026. [Online]. Available: https://github.com/chaoss/grimoirelab-sortinghat

[39] OSCI – Open Source Contributor Index. Accessed: May. 05, 2026. [Online]. Available: https://opensourceindex.io/

[40] OSSInsight — AI Agent Analytics & Open Source GitHub Intelligence. Accessed: May. 05, 2026. [Online]. Available: https://ossinsight.io

Acronyms and Abbreviations

AIC Akaike Information Criterion.

API Application Programming Interface.

ARIMA Autoregressive Integrated Moving Average.

AUC Area Under the Curve.

BIC Bayesian Information Criterion.

CHAOSS Community Health Analytics Open Source Software.

CV cross-validation.

DORA DevOps Research and Assessment.

DSR Design Science Research.

ETL Extract, Transform, Load.

LOC Lines of Code.

LR Logistic Regression.

MAE Mean Absolute Error.

ML Machine Learning.

OS Open Source.

OSCI Open Source Contributor Index.

OSS Open Source Software.

PCR Peripheral Contribution Rate.

R^2 R-squared.

REST Representational State Transfer.

RF Random Forest.

RMSE Root Mean Squared Error.

SVM Support Vector Machine.

UUID Unique User Identity.

VCS Version Control Systems.

List of Figures

1 Key Factors for Adopting Inner Source 3
2 The onion model of an OS community 4
3 Data Processing Flow 26
4 ML Flow 27
5 Supabase and Frontend Flow 29
6 Dashboard - Top navigation bar and current reality cards 40
7 Dashboard - Scenario levers and prediction model selection 41
8 Dashboard - Prediction Visualisations 42
9 PCR for two classified repositories 48
10 Funnel width for classified top and bottom quartile repositories 49
11 Data Lake Structure 60
12 Dashboard - Overview 62

List of Tables

1 Identity resolution across the evaluation dataset. 46
2 Proxy filtering across the evaluation dataset. 47
3 Random Forest regression performance on the held-out test set. 50
4 Exemplary extract of all RF predictions compared to true window B PCR. 51
5 Normalised feature importances from the cross-project RF model. . . . 51
6 Performance of L1- and L2-regularised LR models on test samples. . . . 52
7 Performance of the Linear SVM classifier on test samples. 53
8 Coverage of valid window A/B splits across repositories. 63
9 Median review density (comments per pull request and time-to-merge grouped by pull request type. 63
10 Random Forest cross-validation results on the training set. 63
11 Logistic Regression cross-validation results for Lasso (L1) and Ridge (L2) regularisation on the training set. 64
12 Standardised Logistic Regression coefficients for L1 and L2 regularisation. 64
13 Linear Support Vector Machine coefficients. 65

Listings

1 Exemplary configuration of the entire Inner Source prediction module. 60