Automatic Summary of Code Contributions

MASTER THESIS

Luca Bretting

Submitted on 7 January 2026

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

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

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

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

Erlangen, 7 January 2026

License

Erlangen, 7 January 2026

Abstract

In collaborative software development, contributors frequently make code changes to a common code base. To avoid conflicts and contribute effectively, developers must keep up with the changes of others. Reading through code changes, pull request descriptions or issues can be time-consuming, particularly in large projects. While there is an abundance of research on generating commit messages and release notes, little attention has been devoted to generating developer-oriented summaries of contributions at the repository level. To address the research gap and help developers understand code changes more efficiently, this thesis proposes an approach to generate highly configurable, personalized summaries of code contributions from GitHub and GitLab repositories on-demand. The approach aggregates data from commits, pull requests and issues, analyzes the data using defined rules, and summarizes pull requests using Large Language Models (LLMs). The resulting summaries make it easier for readers to spot important contributions and allow them to understand the changes and their broader context faster.

Contents

List of Figures

Listings

List of Tables

Acronyms

NLP Natural Language Processing

LLM Large Language Model

PR Pull Request

VCS Version Control System

API Application Programming Interface

LOC lines of code

UI User Interface

AI Artificial Intelligence

REST Representational State Transfer

1 Introduction

Modern software is driven by continuous change. While traditional software development paradigms like Waterfall follow a linear structure where requirements are defined early and later change is discouraged, modern agile methodologies embrace it. In this paradigm, the ability to quickly adapt to changing requirements is seen as a competitive advantage (Mishra & Alzoubi, 2023). Although this adaptability allows better alignment with changing customer needs, it also introduces frequent code changes that developers in collaborative environments must keep up with.

To track and reduce the risk of changes, distributed Version Control Systems (VCSs) such as Git or Mercurial have become the standard in both industry and open source projects. These systems are crucial for collaboration as they prevent overwriting of other developers' changes when they are working on the same file, as well as maintaining a history of revisions of the software after each change to be able to compare versions and revert changes in case of unforeseen behavior (Zolkifli et al., 2018). Code hosting providers like GitHub have built on this foundation with features such as code review by peers, graphical user interfaces as opposed to the terminal user interface of Git, and issue tracking systems. On GitHub, a developer usually creates their own copy of a project (*a fork*), makes changes and then requests for the changes to be included into the main repository by creating a Pull Request (PR), optionally with a description of the changes. Figure 1.1 shows an example of a PR description. Once the project maintainers deem the changes to be ready for integration, they can merge the PR to make the changes part of the history and, as such, the basis for all subsequent changes (Kalliamvakou et al., 2014).

While these tools provide support in preventing conflicts, they only prevent concurrent modifications to the same file, and only after such a modification has occurred. More importantly, changes do not need to be in the same file to be conflicting - changes in other components, internal Application Programming Interfaces (APIs) or configurations can be just as harmful to an individual's work, yet could go unnoticed if the person is not made aware of them. As such, knowledge of what coworkers are doing is one of the most frequently needed types of

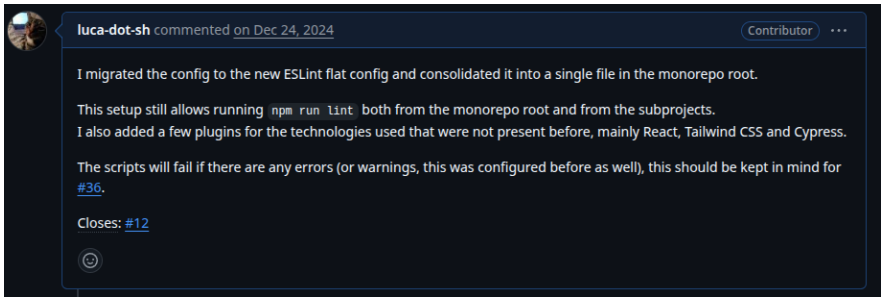


Figure 1.1: Pull Request Description with linked issues

information by developers. This information is obtained most often by asking a coworker directly, which is effective in terms of success rate (Ko et al., 2007), but can be time-consuming and can fail when coworkers are absent.

The GitHub and Git user interfaces for comparing revisions are useful when inspecting a single set of changes (*a commit*) or a PR, but lack in providing an overview of what is happening in the project. In projects with large amounts of changes, such an overview is needed to know where to look for changes affecting the contributor's own work. In 2014, Google reported around 15 million changed lines of code per week, which equates to an average of 600 changed lines for each of the 25000 developers they employed at the time, per week (Potvin & Levenberg, 2016).

At this scale, looking at individual commits or PRs can quickly become time-consuming and error-prone. Additionally, the context and reasoning behind changes is often scattered across multiple sources of data: Commit messages, PR titles and descriptions, issue trackers, comments on PRs, code comments, and ultimately the code itself. One developer might also work on many different projects, which breaks up these sources even more.
Software development teams lack an automated system to summarize code contributions and related context to help them keep track of each others changes, prevent conflicts, and reduce time spent reading source code changes that might not be relevant to them.

To address this problem, this thesis uses design science methodology to create a solution artifact to summarize code contributions. The design science paradigm is well-suited for this, since it is concerned with solving important, organizational problems by creating artifacts that address these problems (Peffers et al., 2007). The created artifact summarizes code changes in Git repositories on GitHub and GitLab, including related information such as PR descriptions and resolved issues into a single summary. It allows filtering changes by a given timeframe, author, and branch, and supports data from multiple repositories in the same summary. To provide an easy to read description of the changes, all of the

data associated with the PR is gathered and automatically summarized using a LLM. Additionally, changes are ranked by their importance and analyzed for noteworthy changes, such that the reader can quickly find what is most important to them. This summary is generated as Markdown, which easily integrates into different frontends, but could for example also be sent on a regular basis via e-mail to inform the team about the progress of their peers.

The overall structure of this thesis is derived from the Design Science Research Methodology for information systems research proposed by Peffers et al. (2007). In Chapter 2, the contribution process and the necessity of code change awareness is introduced in detail, motivating the need for a solution. To provide an overview of the existing solutions, approaches from similar summarization tasks such as Release Note Generation are presented and analyzed for their suitability for maintaining awareness. Having identified the problem, Chapter 3 defines goals and non-goals, potential target audiences and the requirements for a solution artifact derived from these. In Chapter 4, the overall architecture and design process of the artifact is presented, the implementation details of which are described in Chapter 5. Chapter 6 provides a demonstration of the solution on real data. Chapter 7 compares the requirements defined earlier with the demonstrated solution and evaluates whether they are satisfied. Chapter 8 reflects on the limitations of the approach and provides an outlook for future work that could build on this thesis.

2 Problem Identification

As discussed in the preceding chapter, modern software development introduces large amounts of changes and related information that developers need to keep up with on a regular basis. This chapter introduces the contribution process in detail, describes why code change awareness is required in collaborative software development, and defines the problem to solve. While generating developer-oriented summaries of code changes has received little attention, related tasks such as Release Note Generation share common problems and are therefore introduced in the following to provide an overview of the summarization techniques and data sources used.

2.1 Context

Version Control Systems are standard practice in professional software development, with more than 96 percent of professional developers using Git in 2022 (Stack Exchange, 2022). The separation of working copies in Git is done using branches, usually with a main branch from which the software releases are built and branches for features or bug fixes. Various branching strategies exist, e.g. using additional branches for the integration of changes that should not be released yet, which are then later merged into the main branch (Cortés Ríos et al., 2022).

In professional software development, code changes can often not directly be merged on the main branch by the author but must undergo a peer review process first. Many companies and large open source projects employ code review in their code contribution process to improve quality and distribute knowledge (Badampudi et al., 2023). Modern code review involves a multi-step process that is supported by platforms and tools like GitHub. After the changes have been made, the author usually writes a natural language description (e.g. PR description on GitHub) of the changes, which can also include rationale or implications for future modifications. When issue tracking systems are used, this description should also reference issues (i.e., bug tickets) that are solved or affected by the changes (Badampudi et al., 2023). GitHub supports this workflow by letting users

semantically connect PRs to issues using keywords, e.g. "Closes: #1". When the PR is merged, the associated issue (#1) is then automatically closed (GitHub, 2026b). In the remainder of this thesis, the issues linked this way will be referred to as *closing issues*.
After creating a description, the author makes a selection of peers to review the changes, which are then notified of being selected. The reviewer can leave comments, suggestions or point out issues that the author of the changes can use to iterate upon them. After their review, the reviewer approves the changes, requests further refinement, or outright rejects the PR (Badampudi et al., 2023). In addition to facilitating communication during the review process, supporting tools are also useful to understand the rationale and decisions that went into the change when revisiting it, as the trail of information left during the review process is available to other contributors as well.

2.2 Code Change Awareness

In collaborative software development, developers must keep up with the work of their peers, primarily to prevent unwanted interference with their own contributions (Codoban et al., 2015). Beyond facilitating feedback and suggestions, the previously described code reviews can also function as a communication mechanism for code changes that may be relevant to specific reviewers. By explicitly notifying reviewers of a PR, the direction of information flow is inverted: instead of passively waiting for others to discover and seek out relevant changes, reviewers are proactively informed of upcoming modifications by being added to the review. This shift, however, raises the question of who should be notified, i.e. who is likely to have an interest in or is affected by the proposed changes. Although prior work has explored approaches for recommending reviewers (Yu et al., 2016), providing a reliable answer to this question in practice typically requires knowledge of the current work items and responsibilities of all potential reviewers. Additionally, assigning many reviewers to a PR may create a bystander effect where reviewers might overly rely on others review changes, and, as such, may slow down the review process itself (Rigby et al., 2025). Kim (2011) found that developers may perceive e-mail notifications as disruptive, which can result in these notifications being ignored to maintain task focus. In the study, they found a preference for tools that would provide the information when the users request it.

Other than code reviews and communication with coworkers, developers most commonly keep up with changes by skimming the VCS. (Codoban et al., 2015). However, manually inspecting all PRs or commits for important changes is time-consuming, especially in large projects. When inspecting a PR, GitHub and GitLab User Interfaces (UIs) provide the description, commits of the PR and a line by line comparison to the target branch displaying the differences for each

modified file (GitHub, 2026a), also called a *diff*. This interface is suitable for reviewing, but insufficient for quickly searching through changes. If the author of the PR provided additional information about the PR's contents to help reviewers, it may be scattered across commit messages, code comments, the descriptions of linked issues and ultimately the PR description. This leaves the reader jumping between different sources to understand changes and their context. While reading commit messages is the most common strategy for understanding commits, their quality is reported to vary drastically (Codoban et al., 2015). Additionally, projects may use many distinct repositories for individual components of a system. This only potentiates the problem, as in this case, developers working on many repositories need check each one individually. As such, a system summarizing code contributions in projects could greatly reduce the amount of needed communication, time and effort needed to keep track of and understand code changes.

2.3 Existing solutions

There are several approaches that summarize code contributions, most notably with the goal of automatically generating release notes. This section will explore the existing solutions for inspecting and summarizing contributions to repositories, focusing on approaches that generate summaries of changes as opposed to techniques for summarizing source code.

2.3.1 Release Notes Generation

The term *Release Notes* refers to information that is often published along with the release of a new software version to inform stakeholders of the changes that were made to the software since the previous version. These release notes typically contain information about new features, bug fixes, and other changes made in the particular version (Abebe et al., 2016).
The creation of meaningful release notes can be a tedious and time-consuming process, with some practitioners reporting that it can require up to eight hours for a single release (Moreno et al., 2017). As such, automating this process is desirable from a productivity point of view. Numerous approaches have been proposed, which can be divided into two categories: rule-based approaches and machine learning based approaches. These two categories mostly use the same data sources: Metadata from Git history, e.g. commit messages or diffs, as well as PR and issue data from software development platforms like GitHub. The differences lie in how this data is processed and summarized. Comparison of the results of different approaches is generally hard as evaluation is often done manually against different baselines, which is why this section presents popular approaches compared by the techniques used, not by their results.

Rule-based approaches

Rule-based approaches generate release notes by extracting changes from VCSs and performing static code analysis, e.g. extracting modified methods or classes, as well as VCS metadata analysis. The extracted information is then used to generate text snippets to be included in the generated release notes.
The widely cited approach by Moreno et al. (2017) proposes a four-step pipeline for Java projects. In the first step, among others, changed files, methods, classes, variables, and deprecations are extracted. In addition to code modifications, changes to library dependencies, licensing and documentation are extracted as well. All of these require rules, i.e. the programming language syntax or the location of dependency definitions to be defined. Documentation changes, for example, are distinguished from other file changes by defining a set of common file extensions used for text files and declaring them as documentation. In the second step, closely related changes are consolidated to then be transformed into natural language descriptions. For instance, when a file is added that defines a new class, the class itself will be included rather than the file, avoiding redundant additions and improving conciseness. In a third step, the issues from an issue tracker, in their case Jira, are linked to commits. Finally, the fourth step assembles an HTML document from the collected data (Moreno et al., 2017).
Ali et al. (2020) proposes a similar pipeline, adapted for Node.js projects. In addition to the content of the release notes generated by Moreno et al. (2017), their approach also includes changes to the endpoints that serve requests.
A slightly different semi-automatic approach is suggested by Klepper et al. (2016). Their approach also collects data from a VCS and generates a textual summary; however, it includes a manual step in which a release manager is able to select changes to be displayed for different audiences and can manually add additional notes. As different stakeholders might require different information about a release, this additional step would allow keeping technical details from users, while providing them to developers who might benefit from them.

While academic research has proposed numerous approaches, several commercial solutions and integrations into software development platforms such as GitHub have emerged. GitHub provides an automatic release note generation feature that includes merged PRs, authors that contributed to a release and more, configurable via file in the repository. This configuration allows the definition of rules to include or exclude PRs based on assigned tags (GitHub, 2025a), which could allow a manual selection similar to Klepper et al. (2016). GitLab has a similar feature that uses commit messages to generate release notes and links their associated PRs (Ridley, 2025). Several open-source solutions exist, with a popular GitHub repository being *Conventional Changelog*, a tool that uses commit message prefixes found in the *Conventional Commits* specification (e.g. feat, fix) to group commits into categories (conventional-changelog Team, 2025).

Machine Learning based approaches

In recent years, many researchers in the field suggested using machine learning techniques such as text summarization models or LLMs to tackle the problem of generating release notes. In contrast to many rule-based approaches, these models are often generalized models and are therefore not tied to a specific programming language. DeepRelease (Jiang et al., 2021) was the first deep learning based approach and presents two models that are combined to generate release notes. The first one is a text summarization model that summarizes information from a PR, i.e. description, title and commit messages. The second model classifies the PR into one of four categories, namely fixes, features, non-functional changes and documentation, to be able to group them in the textual summary. Both of these models are trained on a self-created dataset of PRs from public GitHub repositories (Jiang et al., 2021).
A more recent, different approach proposed by Daneshyan et al. (2025) called SmartNote uses a general purpose LLM to generate summaries for the release notes. The approach uses a combination of data sources, namely commit messages, source code changes, PRs titles and descriptions as well as repository metadata from GitHub to provide context for the LLM. In their summarization of changes, they summarize individual commits by using the LLM. If commits are part of a PR, their summaries are aggregated by summarizing them again, creating a single entry in the release notes for the PR. Another unique feature is the usage of the LLM to reorder changes, prioritizing semantically important changes such as ones marked as breaking changes. The generated release notes are also tailored towards the individual project by identifying its domain and release note writing style and adapting to it. Using these and other techniques, they claim to be state-of-the-art (Daneshyan et al., 2025).

2.3.2 Change Set Summarization

The summarization of a single set of source code changes is largely studied with the goal of generating commit messages. More recently, academia and industry have also developed approaches for generating PR descriptions from changes. While early, now considered outdated approaches for commit message generation used static rules and templates, more recent approaches have used machine learning techniques to translate different representations of the diff into commit messages (Y. Zhang et al., 2024). Similar to release note generation research, using LLMs has shown promising results, especially in human evaluation (Wu et al., 2025; L. Zhang et al., 2024).

Research on generating PR descriptions is dominated by machine learning based approaches using commit messages and added code comments to generate PR descriptions, notably without using diffs (Fang et al., 2022; Z. Liu et al., 2020; Sakib et al., 2024). A popular commercial solution is GitHub Copilot, an Artificial Intelligence (AI) coding assistant which provides a PR summary feature. While the source code is proprietary and its methodology is not known to the public, the documentation states "Copilot pull request summaries uses a simple-prompt flow . ... [which] utilizes the generic large language model" (GitHub, 2025c, About Copilot pull request summaries section). Part of this flow is the generation of a prompt with data from the code diffs of the PR. However, the documentation also states that the generated summaries can be inaccurate and incomplete at times (GitHub, 2025c). Despite its limitations and perceived simplicity, Xiao et al. (2024) found reduced review time as well as an increasing adoption in a study on the usage of this tool in practice.

2.3.3 Other approaches

De Miranda et al. (2024) used LLMs to summarize students contributions in university projects. They extract a `git blame`, related commit messages and the cyclomatic complexity for each file, which is used to first generate a description of the functionality of the file. This description is then used to generate a summary of an individual's contributions to the project. Additionally, they also generate a summary of the entire team's progress by using a project description as well as client requirements (De Miranda et al., 2024).

2.4 Differentiation and problems

While release notes are a way of summarizing code changes, they are insufficient for maintaining awareness of changes.
Firstly, release notes in general are targeted towards a diverse audience of stakeholders, with system internal changes or refactorings often being omitted. In practice, release notes focus on both fixed issues and added features, with only around 25 percent including system internal changes (Bi et al., 2022). These details may not be as important to a user of the software, but are crucial for developers working on the project. While some release note generation approaches (Ali et al., 2020; Moreno et al., 2017) include source code level changes by parsing source code, their approaches are specific to a single programming language, limiting their applicability. The other rule-based approaches solely rely on the reproduction of commit messages or PR titles, which creates a reliance on the presence of meaningful, expressive commit messages. When commit messages are too shallow, important details can be missed. In case of conventional-changelog Team (2025), the commit messages also need to follow a specific convention,

Publication/Tool	Languages	Data Sources
Klepper et al. (2016)	All	Manual input, commit messages, issue tracker
Moreno et al. (2017)	Java	Source code, commit messages, issue tracker
Ali et al. (2020)	Node.js	Source code, commit messages, issue tracker
GitHub RN Generator	All	PR Titles, PR Tags
GitLab RN Generator	All	Commit messages
Conventional Changelog	All	Commit messages
Jiang et al. (2021)	All	PR title, desc., commit messages
Daneshyan et al. (2025)	All	Git, GitHub API

Table 2.1: Overview of release note generation approaches with supported programming languages and data sources

limiting the applicability further. An overview of the supported programming languages and data sources of the presented approaches can be found in Table 2.1.

Secondly, the release notes are tied to a specific release as they are published along with a release (Abebe et al., 2016). As such, the points in time at which a developer can use the release notes to inspect changes are tied to the releases of the software. The findings of Kim (2011) suggest that developers prefer tools that provide information about the changes of their peers on demand, a criterion not satisfied by release notes.

Finally, release notes target a broad audience as opposed to a single individual. With the goal of summarizing of code changes, this limits the opportunities for personalization. By making such a summary ephemeral and targeted towards a particular user, new possibilities for drawing attention to changes that may be relevant to the individual emerge, e.g. by filtering changes to the users needs or removing their own changes from the summary. While Daneshyan et al. (2025) personalizes the release notes to the repository's conventions, none of the presented approaches are meant for or support user-specific personalization. Additionally, none of the presented approaches support summarizing contributions across multiple repositories.

Since existing commit message generation approaches aim to facilitate the process of writing commit messages, they could be used for summarizing individual contributions (even after the commit was made by ignoring its existing commit message) and assembling them into a summary of contributions. However, in the process, valuable, manually added context is lost, i.e. the commit message the author used or issues that were linked in it. Using PR description approaches

this way suffers from the same issue. The author might have written a comprehensive description of the PR, yet when using the PR description generation approaches as a sub task for repository level summarization, it is discarded, again with context that is only manually inferrable, such as linked issues.

The approach by De Miranda et al. (2024) focuses on generating summaries of the contributions of a single individual, and while they also generate summaries of the entire teams progress, this second step uses client's requirements as additional context. This focus is well suited for educational contexts where the goal is to assess an individual's performance, but is suboptimal to maintain awareness of code changes, as developers' interests can often extend beyond a single person. The use of clients' requirements is an interesting design decision not found in the other discussed approaches, but introduces a dependence on manual input and well written requirements.

To summarize, while there are existing solutions for summarizing code changes, they are not suitable for maintaining code change awareness. While release note generation approaches generate repository level summaries of changes, they often lack in depth, are programming-language specific, or solely reproduce commit messages and PR titles. Commit message generation and PR description generation approaches solve sub tasks, but their effectiveness is limited by their limited use of additional context manually provided by contributors. There exists a gap in the literature regarding approaches for summarizing code contributions and their context within a repository after the changes were made, specifically aimed at supporting and maintaining developers' awareness of code changes.

3 Objective Definition

This thesis aims to bridge the research gap identified in the previous chapter by designing an artifact that summarizes and provides an overview of contributions to support code change awareness. In this chapter, the objectives and requirements for this artifact will be defined.

3.1 Goals

The goals for the solution artifact are structured into three categories to address the identified common gaps in previous solutions: (1) summarization for developers, (2) wide applicability, and (3) personalization to the readers needs.

Summarization: Similar to previous approaches in release note generation and change set summarization, the artifact should generate textual summaries of change sets, e.g. PRs or commits. While release notes might omit internal details (Bi et al., 2022), the approach should explicitly include these details, e.g. file or class names, since it is heavily targeted towards developers working on the given project. To gather the entire context of a change and reduce the need for jumping between pieces of it, the summary of the change set should include all related context, e.g. PR descriptions, commit messages and linked issue description, in the summarization process. Notable changes should be highlighted to provide the reader with hints for prioritization, similar to Daneshyan et al. (2025) who highlight breaking changes by ordering them higher in the release notes for certain projects.

Applicability: Learning from the pitfalls of previous rule-based release note generation approaches (Ali et al., 2020; Moreno et al., 2017), the approach should be widely applicable, i.e. should not be limited to a set of programming languages. Another factor affecting applicability is the integration with Git hosting platforms and issue trackers. Since PR descriptions and linked issues provide valuable context and especially issues as context have received little attention in previous work, the approach should make use of them by integrating them into the summarization process. This requires integration with the respective Git hosting

platform, as both are not features of Git. Although not all Git hosting platforms can be supported, the most popular ones, GitHub and GitLab (JetBrains, 2021), should be supported as data sources.

Personalization: As identified in the previous chapter, existing solutions do not allow for the creation of summaries personalized to an individual reader, as they are generally meant for a broader audience. Given that the goal is summarization, the text should convey all relevant information as short as possible. Readers might deem certain information entirely irrelevant, e.g. when they are only interested in contributions from a certain time frame, and as such, the artifact should allow filtering contributions for user-specified filters. Codoban et al. (2015) found that users have increased interest in certain parts of the code base, which the user should also be able to express to further tailor the summary to their needs. This ability to parameterize, along with the preference for solutions that provide information when the user requests it found by Kim (2011), also necessitates the creation of summaries on-demand, in a timely manner.

3.2 Non-Goals

While the approach should provide summaries for change sets, the goal is not replace the inspection of line-by-line diffs, since the summary of the change sets should only provide a high level overview of the changes. Instead, it should help in the localization of interesting changes, filtering them from uninteresting ones and provide easy access to the diffs for closer inspection.

3.3 Target Audience

The artifact should generate summaries tailored towards internal stakeholders of collaborative projects, e.g. proprietary industry or open source projects. These projects are assumed to follow a minimum of good practices regarding the usage of Git and GitHub/GitLab, such as not having empty or meaningless commit messages (e.g. ”add code”) and optimally the use of issue trackers. Since GitHub themselves describe pull requests as their ”foundational collaboration feature” (GitHub, 2026a, About pull requests section), their usage in collaborative projects is also considered good practice here.

While a variety of stakeholders could benefit from these summaries, the primary audience are developers who want to maintain a high-level overview of the changes or catch up after coming back to the project after a prolonged period of time, e.g. a vacation or a sick leave. When a developer spots a change they deem interesting, they may want to be able to inspect the changes further, up to the source code level. Additionally, technical team leads could also use the summaries

to get an overview of the code changes their team made. While they might not be as interested in diving deeper, summarizing what a particular team member has done could help them track progress, especially across multiple repositories. Finally, quality assurance engineers, e.g. when developing test automation, could use the summary to see what parts of the code base changed to improve test coverage of these areas.

3.4 Requirements

From these goals, non-goals and target audience, the requirements are defined with their category of origin in Table 3.1. Each requirement is assigned a priority using the MoSCoW (Must, Should, Could, Won't) prioritization technique, which groups requirements into one of four priority categories (Kravchenko et al., 2022).

ID	Requirement	Priority	Category
RQ1	The artifact shall generate a structured textual summary in natural language.	Must	Summarization
RQ2	The artifact shall support filtering input data by timeframe, author, branch, and repository, and shall allow combining these filters in a single run.	Must	Personalization
RQ3	The artifact shall generate a summary containing all of the change sets and their closing issues from an issue tracking system according to the user's filters.	Must	Summarization
RQ4	For each change set (e.g. pull request, commit) included in the summary, the summary generated by the artifact shall provide a link to view the corresponding code diff in the respective platform (e.g., GitHub, GitLab).	Must	Non-Goals
RQ5	The artifact shall support GitHub and GitLab as data sources with feature parity.	Must	Applicability
RQ6	The artifact shall support multiple repositories as input for generating a single combined summary for them using one common set of input parameters.	Must	Applicability
RQ7	The summarization approach of the artifact shall be programming language agnostic.	Must	Applicability
RQ8	The artifact shall provide natural language summaries of PRs.	Should	Summarization
RQ9	The artifact shall aim to generate summaries in under 60 seconds.	Should	Personalization
RQ10	The artifact shall numerically analyze PRs for deviations from historical data or statically defined values and highlight these PRs in the generated summary.	Should	Personalization
RQ11	The artifact shall support configuring paths to specific parts of the code base and highlight any changes to these areas to the user.	Should	Personalization
RQ12	The artifact shall not rely on third party APIs other than GitHub and GitLab to provide its core functionality.	Could	Summarization

Table 3.1: Requirements for the solution artifact with priority and category of origin

4 Solution Design

4.1 Approach

In the design of a solution artifact, an important first choice is the granularity at which changes are summarized. Since the goal of the approach is not to replace inspection of line by line diffs, unstructured summarization (e.g. by using a LLM with all changes and without enforcing a structure) is not an option since traceability would be lost or changes could be omitted. However, if the summary were to simply list and summarize individual commits, it may quickly get too lengthy to save the user time. Drawing from previous solutions to this problem (Daneshyan et al., 2025), a hybrid approach was chosen. If commits are added as part of a PR, the relevant change set is the PR in which they are introduced. If they were not added to the branch by a PR, they will be handled in a separate section. This design allows using all context information of a PR, e.g. description, title and linked issues, in the summarization process, as well as being able to spot commits that may not have undergone review since they were directly committed onto the given branch. Additionally, it also allows the inclusion of currently open PRs targeting the given branch in the summary, which is crucial for spotting conflicting changes before they occur.

Similarly to Moreno et al. (2017), who identify documentation changes based on their file extensions, this approach has three sections for analysis of changes to individual files, namely configuration, documentation and the top five most changed files. The configuration and documentation sections are filled by classifying file changes by their file extension, with default values for file extensions that can be changed according to the user's needs. Although Moreno et al. (2017) also includes changes to dependencies, their definitions are programming language specific, which makes them unfeasible, since the goal here is a programming language agnostic approach.

From this split, the following basic structure is inferred:

- **Merged pull requests**: PRs that were merged onto the given branch or default branch, with additional information such as the issues it closes or commits it includes, to provide an overview of what has been done already.
- **Recently opened pull requests**: PRs that were newly opened and are not merged yet with the same additional information as merged PRs, to provide an overview of upcoming changes and work in progress.
- **Other commits**: Commits that were not part of a PR, i.e. directly pushed onto the branch.
- **File changes**: These have three subsections, which are derived over all commits in the given constraints.
  - **Most changed files**: Files that experienced the most changes in terms of lines of code, ordered by the number of changes.
  - **Documentation changes**: Changed files classified as documentation based on their file extension, ordered by the number of changes, configurable.
  - **Configuration changes**: Changed files classified as configurations based on their file extension, ordered by the number of changes, configurable.
- **One line summary**: An overview of statistics, such as the total number of lines added and removed, or the number of merged PRs.

As the goal is the creation of a summary, information density should be high by definition. To achieve this and better support the goal of identifying interesting changes faster, empty sections are omitted by default. However, a lack of data may also be relevant information for certain users, thereby this is a configurable option.

The following sections will introduce the high level architecture, detail the approaches taken generating for the contents of each section, as well as the techniques used to further highlight important changes.

4.2 Data Sources

As identified in the previous chapter, for the approach to be widely applicable, it should support GitHub and GitLab as sources of data. As such, data is extracted from these two platforms, namely from endpoints for PRs, issues and commits. It should be noted that there are overlaps in these endpoints, e.g. GitHub views PRs as issues as well, yet the separate endpoint for PRs is required as it provides

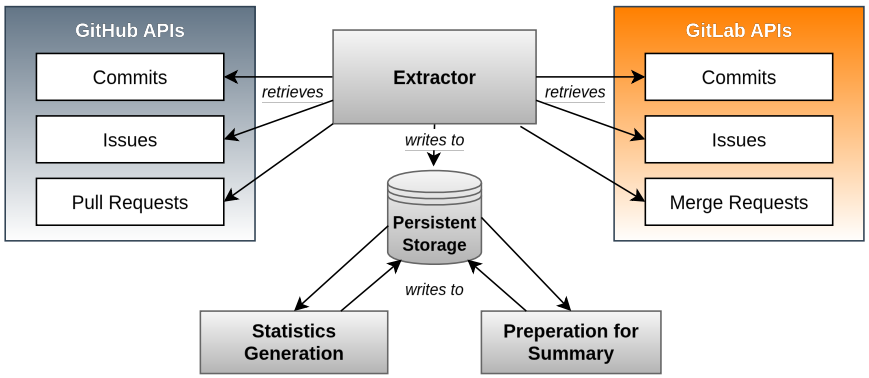


Figure 4.1: Overview of data sources and components of the batch job

additional information not provided for issues. For some features, cloning the repository is required, largely to prevent using up rate limits for features that do not require GitHub or GitLab specific information.

4.3 High Level Architecture

Speed is important for the approach, since the generation of the summary should happen when the user requests it to prevent interrupting their workflow (Kim, 2011). However, the extraction of data can be a time-consuming process, which is furthermore slowed down by GitHub's rate limits (GitHub, 2022a), especially when a large amount of changes is present.

To ensure a timely and reliable generation of summaries, their is split into two parts. First, the extraction of data from Git, GitHub and GitLab and the preprocessing of this data is done as a batch job detailed in Chapter 4.3.1. This batch job has less strict time constraints, as it can be run at times of no development activity, e.g. at night. In a final step, the data is then stored for later retrieval. The second part then uses the retrieved data to generate the summary by linking, formatting and amending it with information dependent on the parameters of the summary, which are discussed in section 4.3.2. This split enables fast and personalized on-demand generation of summaries, as well as a reduced number of calls to GitHub and GitLab APIs when multiple summaries are generated. It would also allow the two parts to run on entirely different machines.

4.3.1 Batch Job

Given a list of repositories from GitHub and GitLab, the first step queries the respective endpoints for commits, issues, and PRs, as shown in figure 4.1. The data is first stored in its raw response form and then cleaned, e.g. by removing un-

needed fields and flatting structured fields. Once the data is retrieved, additional steps that use it but do not need runtime input parameters can be performed. One such step is the derivation of historical repository statistics from commits, issues, and PRs to fit the summary to project-specific conventions when analyzing a PR, rather than using fixed values as thresholds for decisions. In another step, the person who last modified each file prior to a PR (called the *file blame* in the following) is determined and stored for each PR by analyzing commit history. This information is used at runtime to highlight modifications to a user's last modified files. Storing this information with each PR is a tradeoff between storage requirements and runtime speed, and since one of the primary goals is timely generation on a user's request, runtime is prioritized here.

4.3.2 On-demand generation

When the data is successfully extracted, a summary of code contributions can be generated on demand. This stage has stricter time constraints, as the user has actively triggered it and is waiting for the summary. Since one of the goals defined in Chapter 3 was the ability to personalize the summary, a central configuration is created that allows setting optional parameters for the generation of the summary. These parameters include filters such as a timeframe, formatting options on what to include or enabling the usage of LLMs. An example of a full configuration can be found in Appendix E.

All configurable parameters are optional and have default values or fallbacks. For filters, the default value is no filtering. To ensure feature parity between GitHub and GitLab data, a first step transforms GitLab's naming schema into the GitHub one if GitLab is chosen as a platform. After this step, it is transparent to the remainder of the pipeline which platform was chosen. The result of this pipeline is a summary in Markdown, as it is easy to convert into other formats such as HTML. If the input contains data from multiple repositories, one summary of the previously described structure is generated per repository and the results are concatenated in the same output file, using the same configuration options.

4.4 Code Contribution Summary

Having established a structure of the summary and high level architecture of the pipeline, the defined sections must now be filled. In Chapter 2, two paradigms for release note generation were identified, rule-based and machine learning based. Rule-based approaches suffered from one of two problems, they either were programming language specific or simply reproduced commit messages or PR titles, however excel in producing predictable summaries. The machine learning based approaches do not suffer from these problems, but require complex training of custom models (Jiang et al., 2021) or use general purpose LLMs (Daneshyan et al.,

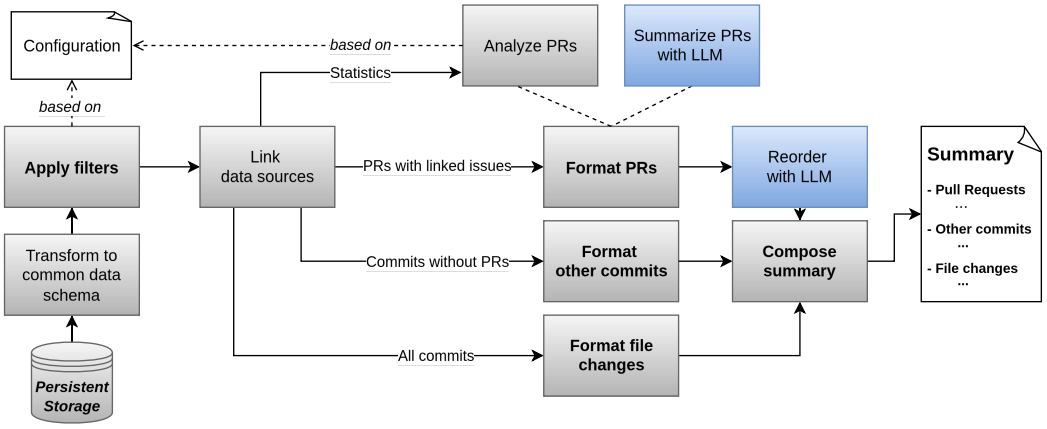


Figure 4.2: Overview of the steps for on-demand summary generation. The steps in blue are skipped when LLMs are disabled.

2025), which are often hosted by third parties (Liagkou et al., 2024), potentially limiting the applicability in environments with data protection concerns.

To combine the strengths and mitigate the weaknesses of both paradigms, a hybrid approach is created. The basis of the summary is a rule-based approach, which fetches, filters and analyzes data locally (called the *rule-based summary* in the following), providing predictable, deterministic structure. However, the previously identified restriction of rule-based approaches to individual programming languages underscores the need for a machine learning based approach to summarizing code changes. Prior work in release note generation and change set summarization revealed two directions for machine learning based approaches, namely custom trained models (Jiang et al., 2021; Z. Liu et al., 2020) and generic LLMs (Daneshyan et al., 2025; Wu et al., 2025; L. Zhang et al., 2024). Custom trained models require datasets, which are easier to obtain for tasks such as commit message generation, as public data contains ground truth (e.g. commit messages) a dataset can be built with. For generating summaries of change sets *including* existing related information such as linked issues, PR description or commit messages, it is hard as there is no document in public repositories that would represent the result of such a process and could serve as ground truth.

Combined with the promising results of recent publications using LLMs for these tasks (Jiang et al., 2021; L. Zhang et al., 2024), the route of LLMs is chosen as they do not need to be trained on custom datasets and are easy to adapt to novel tasks (Wu et al., 2025). Due to limited local compute resources, the OpenAI API providing access to LLMs is used. As this is a third party service that could raise data protection concerns for some users, the usage of LLMs is optional. If the user chooses to allow it, the rule-based summary is extended by generating natural language summaries of individual PRs with their context and reordering

changes using a LLM (called the *LLM-enhanced summary*). The design of the LLM integration ensures that the used third party service can be replaced with a self hosted, on-premise LLM, mitigating potential data protection concerns. The following sections will introduce both variants, while a combined overview of the steps can be found in Figure 4.2.

4.4.1 Rule-based summary

As identified in Chapter 2, prior work focused on generating release notes for a broad audience, limiting the possibilities for personalization when using the generated summaries for code change awareness. In the proposed approach, the summaries are viewed as ephemeral and generated for a single individual. With this shift in focus, the first step is giving the user the ability to specify what they do not want to see included, which is also a common first step when skimming VCS manually (Codoban et al., 2015). As such, the following configuration options are created and the respective filters are applied during the generation of the summary:

- **Timeframe**: Allows setting inclusive start and end dates to constrain PRs and commits to a given timeframe, e.g. to summarize team progress when returning from a vacation.
- **Branch**: As described in Chapter 2, Git allows for different workflows and branching strategies, with some having separate branches for changes to be integrated, but not ready for production yet (Cortés Ríos et al., 2022). This filter allows specifying such branches, with the default value being the default branch of the repository.
- **Repositories**: As the input can contain data from multiple projects, one or more repositories can be specified to limit the summary.
- **Author**: Allows filtering for a specific PR or commit author.

The configuration also allows specifying the user's own GitHub or GitLab username, and if set, they can choose if they want to hide their own contributions in the summary to focus on others work. After filtering, the PRs and issues are joined for entry into the PR sections of the summary. Each PR section consists of:

- A **title** consisting of the PR number, title, author, number of additions and deletions and a link to the GitHub/GitLab UI.
- A collapsible section for it's **commits**, with each commit's message header, age, number of additions and a link to the GitHub/GitLab UI.
- A collapsible section of **issues** the PR closes, with each issue's number, title and link to the GitHub/GitLab UI, omitted if none were linked.

To improve the discovery of noteworthy changes, statistics are computed and compared to either manually configured values or historical statistics for each PR. If this threshold is surpassed, a warning will be added to the respective PR in the summary. These metrics include:

- **Number of comments**: Many potential problems reviewers found and pain points of the changes.
- **Number of commits**: A large amount of individual changes.
- **Number of files changed**: Broad changes to the code base.
- **Number of lines changed**: Large changes to the code base.
- **Number of referenced issues**: Interconnected changes or changes affecting many issues.

Based on the findings of Codoban et al. (2015), which show that developers consider changes in specific code regions or entities more important than other changes, an additional configuration is introduced that allows configuring paths to important files (e.g. environment templates or important API definitions). When these files are modified in a pull request, a warning section will be added to the respective PR to highlight these changes to readers. To support specifying code regions, the configuration also allows the use of wildcards in the beginning or end of the path, which allows users that only work on a particular region of a repository to highlight changes to it. A developer responsible for the build files of a Gradle project could use this feature to highlight changes to Gradle build files, e.g. by setting the configuration to `["*.gradle*"]`. (Gradle, 2025)

Building on the goal of allowing personalization, the identity of the person requesting the summary can be provided in the configuration to analyze PRs for changes to files that were previously modified by them. To do this, the previously generated file blame is retrieved and compared with the files changed in the respective PR. The summary then contains a section for each PR with matches, listing the files and their change types, e.g. "You added `README.md` and this PR modified it". Using this section, readers can quickly spot other's changes that might alter the behavior of their previous contributions and assumptions about the code base. Although previous work has tried to analyze and describe changes with greater granularity, e.g., by comparison on a method or class level (Moreno et al., 2017) or by highlighting changes to endpoints (Ali et al., 2020), the file level was chosen here because it is programming language agnostic and computationally inexpensive. Parsing each file in the repository would also require cloning the repository, while on a file level, the already retrieved commit history is sufficient. The same is true for analyzing dependencies between files.

After all PRs are amended with the additional information described, they are sorted in descending order by the total lines of code (LOC) changed.

4.4.2 LLM-enhanced summary

If the usage of LLMs is enabled in the configuration, the rule-based summary will be extended with additional features. The central configuration shown in Listing 4.1 regulates the usage of LLMs, their provider and model, as well as giving users the ability to limit the cost of a summary.

```
1  # Global configuration for LLMs
2  LLM:
3    # disable/enable LLM usage
4    enabled: false
5    # Configuration for the LLM provider (see LLM.md)
6    provider: openai
7    # Model to use, depends on provider
8    model: null
9    # Maximum cost in USD per run, null = unlimited
10   max cost per run: null
11   # Enable verbose logs
12   verbose: false
```

Listing 4.1: LLM Configuration

The basis of the additional LLM-based features is a generic interface to be implemented for each LLM provider the operator wants to use. By only using this interface in the generation of the summary, the provider or model can easily be switched, e.g. in case of rising costs of a single provider. This architecture is especially important for users with privacy and data protection concerns as it would also allow to the usage of self-hosted LLMs. Two versions of the interface exist, one for synchronous and one for asynchronous requests to leverage the speedup of parallel requests if the backing implementation supports them. For this thesis, the only implementation of this interface uses the OpenAI API, which was implemented for both synchronous and asynchronous requests.

Pull Request Summary

Similar to Daneshyan et al. (2025), who use the README and the project's description to determine the project domain, a summary of the project context is generated for each repository first to be added to the context in later requests. To achieve this, all documentation files (identified by configurable file extensions) are gathered, along with the file tree of the repository and processed by the LLM with the goal of summarizing the project's purpose, architecture, main features and unique aspects in a few sentences. The resulting summary is cached and can be deleted to regenerate it if the project has changed drastically. If the user would rather write this description themselves, they can add a file with their version to

the corresponding repository and reference its path in the configuration.
For each PR, a summary will be generated to quickly get an overview of its contents. The context for this summary includes the previously generated project description, the title and description of the PR, titles and descriptions of the issues it closes as well as diffs for each modified file. Since breaking changes are considered particularly important by developers (Codoban et al., 2015), the prompt includes instructions to highlight them. To further personalize the summary to each reader, the previously defined configuration of important files is translated into natural language and incorporated into the prompt, instructing the LLM to give priority to these files and explain their changes in greater detail. The resulting summary is added to the respective PR as a collapsible section.

Cost Limiter

As the size of requests increases for larger PRs and is unknown to the user before generating the summary, they might want to protect against the unwanted costs of LLM usage. As such, the configuration allows setting an upper limit on the amount of money to be spent during the generation of a summary. Before each request, the cost of the request is estimated using the number of input characters and pricing data of the respective provider. After the request is made, the exact number of tokens and thus the resulting price of the request is added to a sum accumulating costs for the summary. If a request would or has exceeded the limit, the request and all following requests will result in an exception. None of the presented approaches in Chapter 2 implemented similar safeguards.

Semantic Ordering

In the rule-based summary, the PRs in the respective section are ordered by LOC changed. This approach leaves room for improvement, as a large number of LOC changed does not directly imply an important change. For example, modifications to dependency lock files like `package-lock.json` for the Node Package Manager (Karrys, 2022) can be very large, yet might not be as important to some readers as changes to a public API. Daneshyan et al. (2025) used a similar approach to reorder categories of changes, e.g. by moving breaking changes to the top for library projects.
To better capture the semantics of the changes in the ordering of the pull requests, the LLM is instructed to first reason about the potential impact and importance of each PR and then determine an order from most to least important. For this prompt, the input is the content of each PR section as it appears in the summary, which also includes the previous LLM-generated PR summary and personalized sections. Therefore, the semantic ordering uses far less input tokens than the generation of the PR summaries as the content is already condensed and can make use of the user's preferences in the ordering.

5 Implementation

5.1 Development Environment

The implementation is part of the backend of MECOIS, a research project of the supervising chair. As per existing project setup, Python 3.12 was used along with Apache Spark, an analytics system for large amounts of data (The Apache Software Foundation, 2025). Data storage and retrieval is handled by Delta Lake, a storage layer for Lakehouse architecture compatible with Apache Spark (Lee et al., 2024, p. 6).

5.2 Data Sources

The data sources to extract are configured in a central configuration file that specifies the sources to fetch and the respective repositories. As per existing project setup, all extracted data is pre-processed before it is used in the summary. Pre-processing includes the flatting of structured fields, removal of various unneeded fields, converting time fields into a common date type. Additionally, user-identifying information is entered into a separate database that can be used to merge the user identities from separate sources, and the respective column values are replaced with an ID assigned by the database.

5.2.1 GitHub

GitHub offers a Representational State Transfer (REST) and GraphQL API, which differ in the information they provide. For the implementation, both are used as some of the information is only provided by the GraphQL API, while the REST API provides an easy way to retrieve raw data for later processing. The GitHub REST API (Version 2022-11-18) endpoints for repositories follow the URL schema `/repos/{owner}/{repo}/{endpoint}`, where `owner` and `repo` identify the repository uniquely and `endpoint` is the respective endpoint, e.g. `commits`. As all the used endpoints are for repositories, only the `endpoint` part of the URL is given when discussing endpoints.

The following extraction processes are repeated for each specified repository owner combination.

Commits

To extract commits, a bare clone of the repository is created or the latest changes from the repository are fetched if it is already cloned. After cloning, the output of a `git log` command is parsed to extract the hashes of all commits in the repository. This command includes the `--all` flag to extract commits from all branches. For each extracted hash, a request is sent to the `/commits/{ref}` endpoint, where `ref` is the hash of the commit (GitHub, 2022b). While the API also offers the `/commits` endpoint to list commits, requesting data for individual commits provides additional information, such as file changes or statistics required for the generation of the summary. The API is also not used for retrieving the commit hashes as it would require separate requests for all branches, compared to a single command when cloning the repository. To later be able to filter changes by branch and perform analysis, branch information is attributed to each commit. To achieve this, the `git rev-list` command, which lists all commit hashes in a branch, is run for every branch in the repository. The results are used to build a map from each hash to its corresponding branches, which is then used to add branch information to each commit.

Issues

Similar to commits, GitHub provides endpoints for listing issues and acquiring detailed information about an issue (GitHub, 2022c). As only basic information such as the title and description is required, there is no need to request individual issues and as such, the `/issues` endpoint for listing issues is used. The requested data is stored in its original form.

Pull Requests

GitHub regards pull requests as issues and while the `/issues` endpoint includes them, it does not provide PR-specific information such as the target branch of the PR. Therefore, GitHub offers a separate `/pulls` endpoint, which is used here to retrieve a list of all PRs in the repository that contains basic information such as title, but also the mentioned PR specific information (GitHub, 2022d).
However, neither this endpoint nor the endpoint to retrieve a single PR provides information about linked issues. There is a REST endpoint for retrieving commits of a PR (`/pulls/{pull_number}/commits`), however, this endpoint lacks information such as the number of changed lines for each commit. The previously extracted commits data from the repository cannot be used here as the commits might be from a fork and therefore not captured in the data.

For these reasons, the GitHub GraphQL API is used to retrieve additional detailed information about commits and linked issues to add to the data from the REST API. For each PR, the following is information is retrieved:

- Commits with their hash, message (in plain text and HTML), URL, author name and date as well as statistics such as number of lines added. The issues a commit message references can be extracted searching the HTML commit message for `<a>` tags of the class `issue-link`, where the `href` attribute then contains a link ending with the issue number.
- The identifiers of issues the PR closes.
- The identifiers of issues the PR description mentions, these can be extracted the same way as those mentioned in commit messages by retrieving and searching the HTML version of the PR description.
- The number of comments.
- Statistics such as the number of lines added by the PR

The exact GraphQL query can be found in Appendix A. As the `pullRequests` endpoint enforces pagination, the multiple requests may be required until all PRs are retrieved.

Lastly, to retrieve the PR's modified files and the respective diffs, GitHub provides the `/pulls/{pull_number}/files` REST endpoint. An example of the information returned per changed file from this endpoint can be seen below (links are omitted for space reasons).

```
1 {
2     "sha": "951a56fa986d2753f9b828adfddda6f36f5856a8",
3     "filename": "rust/backend/daemon/src/main.rs",
4     "status": "modified",
5     "additions": 1,
6     "deletions": 1,
7     "changes": 2,
8     "patch": "@@ -27,5 +27,5 @@ async fn main() {\n     //
          apparently needed...\n     helpers::bump_rlimit();\n \n
          -    server::serve_forever().await;\n+    server::
          serve_forever_socket().await;\n }"
9 },
```

Listing 5.1: Response from GitHub Pulls Files REST API

This endpoint comes with several limitations. Firstly, although this behavior is not documented (GitHub, 2022d), the `patch` field is omitted for very large file changes, which was observed for various files with over 2000 lines of changes.

More importantly, when files are moved or renamed, the `status` for the respective file will be `renamed`, but no information is provided about the old location or name of the file before the modification. This information is required for later analysis of the history and as such, a different approach was chosen.
The previously extracted basic PR information contains a link to the GitHub Web UI for the respective PR. When appending `.diff` to this link, GitHub will provide a `diff` for each file that can be parsed to extract information about the change to the file. An example of such a diff, which is provided for each changed file in the PR, can be seen below.

```
1  diff --git a/rust/backend/daemon/src/main.rs b/rust/backend/
       daemon/src/main.rs
2  index b2e06a25..951a56fa 100644
3  --- a/rust/backend/daemon/src/main.rs
4  +++ b/rust/backend/daemon/src/main.rs
5  @@ -27,5 +27,5 @@ async fn main() {
6       // apparently needed...
7       helpers::bump_rlimit();
8
9  -    server::serve_forever().await;
10 +    server::serve_forever_socket().await;
11   }
```

Listing 5.2: GitHub Pull Request Raw Diff

To parse the list of these diffs, the file is read line by line. The diff command in line 1, which can be recognized using a regular expression (`diff --git a/(.*) b/(.*)`) marks the beginning of a new file. From there, the first line starting with `---` or `+++` marks the source and destination respectively. These can be used to recognize the change type (called `status` in the GitHub API response) by removing the `a/` and `b/` prefixes and comparing the filenames. An overview of how change types are identified using source and destination filenames is given in Table 5.1. For renamed files, the source also identifies the old filename, which is the missing information in the above mentioned REST API for listing file changes. The remainder until the next encounter of the diff command is the `patch` section from the API, which is collected as well. For each line starting with `+` or `-`, an additions or deletions counter is increased respectively for the current file. The total amount of `changes` is derived simply by adding additions and deletions. When the next diff command or the end of the response is encountered, the collected information is stored and the parsing of the next file starts. In addition to including the old filenames for renamed files, this approach also does not suffer from the size limitations encountered when using the REST API, as it includes the `patch` regardless of the size.

Source	Destination	Derived change type
Path to file	Path to file	Modified
Path to old file	Path to new file	Renamed
`/dev/null`	Path to file	Added
Path to file	`/dev/null`	Removed

Table 5.1: Change types by source and destination

File Blame

To be able to determine which files were modified by a given user before a PR to highlight changes to these files, the user that last modified each file is precomputed for each PR to reduce runtime during the generation of the summary. For each PR, the target branch and the base commit hash, i.e. the hash of the last commit before the commits introduced via the PR start, are obtained. The previously extracted commits from all branches are then filtered for the given target branch and sorted by the time at which they were committed in order from oldest to newest. Iterating over the resulting commits, a map is built that maps each filename to the latest encountered change, including information such as the change type or author name. The map is updated with the following rules:

- If a file was added, it is added to the map.
- If a file was modified, its entry in the map is overridden.
- If a file was removed, it is removed from the map.
- If a file was renamed, the entry for the old filename is removed from the map and the change is stored under the new filename. Additionally, the entry will include the old filename.

After the base commit hash is encountered and processed, the iteration is stopped and the resulting map is stored along with the PR.

Statistics

While GitHub offers an API for retrieving repository statistics (GitHub, 2022e), this API is unsuitable for this approach for multiple reasons. Firstly, the documentation states the statistics data might not be available immediately if it is not cached and should be retrieved later when its computation is finished. More importantly, the endpoints focus solely on statistics related to commits, but later analysis of PRs primarily requires statistics related to PRs, which is not provided by existing endpoints. To obtain these and stay flexible with the metrics used, all statistics are computed locally by analyzing the previously extracted commits, issues and PRs. PRs are grouped by repository owner combination and then aggregated. For the number of comments and total lines changed, the mean value

for each group can directly be computed. For files, commits and issues the PR closes or references, the size of the respective array is computed and averaged. The resulting data is written into a separate dataframe where each row contains the statistics for one repository.

Rate Limits

GitHub enforces a rate limit of 5000 requests per hour for the REST API (GitHub, 2022a). The GraphQL API uses a point system, which can make requests more expensive, but each request costs at least one point (GitHub, 2025b). When extracting data for larger repositories, it is possible for these rate limits to be hit. To deal with this problem, both the REST and GraphQL API requests include two headers in each response: `x-ratelimit-remaining`, which specifies the number of remaining requests or points and `x-ratelimit-reset`, which is the time in seconds until the rate limits resets (GitHub, 2022a). To prevent exceptions or invalid data during the extraction process, the `x-ratelimit-remaining` header is checked on every request and if it is lower than 1, the extraction process waits the duration specified by `x-ratelimit-reset`.

5.2.2 GitLab

Similar to GitHub, GitLab offers a REST and GraphQL API (GitLab, 2025b), but as the REST API offers all the functionality needed to ensure parity with the data from GitHub, the GraphQL API is not used . Interaction with the REST API is implemented using the `python-gitlab` library (python-gitlab Team, 2025). The REST API uniquely locates projects by a numeric identifier in the URL, which has to be given in the configuration for the repositories to extract. Another way to locate them is a combination of a namespace and project path, which can be regarded similar to owner and repository for GitHub. To match the GitHub data's schema for repository identification, the namespace and project path are requested and stored as owner and repository respectively for all data points from the sources below.

Commits

The Commits API (GitLab, 2025a) is used to retrieve all the commits from the repository. In contrast to GitHub, this API allows setting a parameter to retrieve all commits, which makes cloning the repository obsolete for extracting commits from all branches. Additionally, it provides an endpoint to list the branches that contain a given commit, which for GitHub also involves cloning the repository. As such, all commits for the given repositories are retrieved, along with the information which branches they are on and the files that were modified in each commit.

Issues

Issues are simply retrieved using the Issues API (GitLab, 2025c) and stored without any additional modification.

Merge Requests

In GitLab terminology, PRs are called merge requests and can be retrieved using the Merge Requests API (GitLab, 2025d). To obtain the basic information for each merge request, the full list of merge requests is first retrieved to get their IDs. Then, each merge request is queried individually to collect detailed data, as the initial list does not include certain required details that are only available when requesting a single merge request. Retrieving the issues a PR closes and references is possible via two endpoints of the Merge Requests API, the IDs of which are added to the retrieved data.
For simplicity and parity in the data schema, the `raw_diffs` Merge Requests API endpoint (GitLab, 2025d) is used to obtain the list of diff command outputs for each file in the same format GitHub returns when appending `.diff` to a PR link (shown in Listing 5.2). The response can then be parsed with the same technique described for parsing the GitHub PR diffs. For a merge request's commits, the list of commits is gathered and their details are requested for each commit. Additionally, the changed files are requested for each commit to add the number of changed files to the commit data.

GitLab to GitHub Transformation

To be able to use GitLab and GitHub data transparently in the later stages of the pipeline, the used fields need to have common names. As such, GitLab field names are transformed to their GitHub counterparts when needed. For simplicity, this transformation only includes the fields used, not the entire data scheme. While many of the column names differ, their semantics are identical and only renaming is required. These renamings can be found in Appendix G. For others, the values differ, e.g. the `state` of open merge requests has the value `opened` in GitLab, while GitHub uses `open`, and as such, these values need to be remapped to be able to use them. Another special case are the modified files of commits. While GitHub has a `status` field that specifies the change type (e.g. added, modified) for each changed file, GitLab uses three booleans `new_file`, `renamed_file` and `deleted_file`. These can be used to derive the `status` field: If `renamed_file` is true, the value of `status` is `renamed`, if `deleted_file` is true, it is `removed`, if `new_file` is true, it is `added` and finally, if all are false, it is `modified`.

File Blame, Statistics and Rate Limits

Due to the previous transformation to common names, the implementation of the statistics and file blame generation for GitHub can be entirely reused for GitLab data. For all requests, rate limits are handled by the `python-gitlab` library by default.

5.3 Rule-based summary

In the rule-based summary, the previously extracted data is loaded and filtered, adapted to the parameters specified in the `summary-config.yml` and formatted into a Markdown document. An example for the configuration can be found in Appendix E.

5.3.1 Filtering input data

The first step is the validation of the retrieved data, since data extraction is separate from the generation of the summary. As empty dataframes or missing tables could lead to cryptic errors in the remainder of the process, the boundary conditions are checked at start with clear error messages to debug errors more easily. These checks include verification of all the dataframes (i.e. PRs, issues, commits and statistics) being loaded and having more than zero columns.

After successful validation, the loaded data is filtered according to the user-defined configuration. The user can specify one or more repositories, as the input might contain data from multiple repositories. It is also possible to specify only the owner, e.g. to filter by an organization. If specified, the data is filtered using the previously added `owner` and `repository` fields.
To filter for a specific branch, the commits and PRs are filtered for the specified value if given. As the commits dataframe contains the information which branches a commit is part of, the filter checks whether each commit is part of the specified branch. For PRs, the configured value is compared against the target branch of the PR. If the user does not specify a branch to filter, the repository's default branch is used for filtering.
The timeframe configuration consists of separate start and end dates, which can be adjusted independently to include only those data points that fall within the specified range. For commits, the relevant timestamp is the time at which the commit was authored. For PRs, the relevant timestamp depends on the state: if the PR is merged, its merge time is used; if it is still open, the creation time is used. The issues dataframe is not filtered as it will later be linked with the PRs and the issues closed by a PR might be older.
Finally, an author can be set in the configuration by specifying a username. As the data contains the IDs from the identity database, the data could be directly

filtered by finding the ID from the database and specifying it in the configuration. For convenience, this step is done by the filter and the user can directly specify a more readily available field, the username. The filter then searches the database for identities with the username, caches the results and filters the data for all identities that match this username. For commits, the author of the commit (as opposed to the commiter, which can be a different person) is used; for PRs, the creator of the PR is used. Again, the issues are not filtered by author as they might have been created by someone else, but are closed by a PR by the given author.

5.3.2 Linking issues to pull requests

After filtering, the different data sources need to be combined to be used in the generation of the summary. As one summary is generated per repository, the first step here is grouping by owner repository combination. For each PR, the issues it closes are given as IDs, which are used to join them with the detailed issue information from the issues dataframe to later display it in the summary. As the information about a PR's commits is already part of the PR dataframe, there is no need to join them, but not all commits need to be introduced via PRs, e.g. could also have been pushed directly to the branch in question. To cover this case, the commits dataframe is filtered to only contain commits that are not part of any PR. This is done by creating a superset of all commit hashes belonging to PRs and then excluding those hashes from the commits dataframe.

5.3.3 Summarizing pull requests

The summary has two sections related to PRs, "Merged pull requests" and "Recently opened pull requests". The PRs in these sections undergo the same steps, but are pre-filtered. To determine whether a PR is merged, the PRs can be filtered for `state==closed` and `merge_commit_sha is not None`. The `state` only would not suffice as PRs can also be rejected, which also results in a `closed` state, but these are excluded here. For opened pull requests, it is sufficient to filter for `state==open`. In both sections, the PRs are ordered by number of lines changed in descending order.

Adding basic information

Each section has a title consisting of the number, i.e. identifier of the PR, its title as well the number of additions and deletions in the PR. The PR number is formatted as a link to the GitHub/GitLab UI to view its contents in detail if desired. An example can be seen in Listing 5.3.

```
1    #81: "Deployment fixes and generation of test data" (301
        additions, 851 deletions):
```

Listing 5.3: Examplary section title line in the summary

Below are collapsible sections for commits and linked issues. These sections are created using the `<details>` and `<summary>` HTML elements, which allow the creation of sections that open upon clicking them. For the linked issues, such a section titled "Issues closed by this PR" is created containing a bullet point list of the issues with their title, issue number and a link to the GitHub/GitLab UI for viewing the respective issue. If there are no issues linked as being closed by the PR, this section is omitted. The "Commits" section includes a bullet point list of commits, with the first line of each commit message, the age in days from now, the number of added and deleted lines as well as a link to view the commit in detail.

Analyzing for outliers

Each PR section includes warnings if a metric (e.g. files changed) surpasses a certain threshold. The threshold is either set statically in the `summary-config.yml` or derived from the statistics. For each threshold, the value from the configuration is used if it is not `null` or else the derived historical mean value from the statistics is used. This threshold is then compared against each PR, and if the PR exceeds the threshold, a warning is added to highlight the outlier, e.g. "This PR changed a lot of files (78 changed)".

In addition to numeric analysis, the configuration also allows specifying important files such as public API definitions, with support for wildcards as described in Chapter 4. To check whether a file declared as important was modified by a PR, all of it's file changes are compared with the definitions in the configuration. The type of comparison depends on the wildcards used and is shown in Table 5.2. If one or more files in the PR match, a collapsible section is added that lists these files, including each file's change type as well as the number of lines added and removed.

Pattern	`"file*"`	`"*file"`	`"*file*"`	`"file"`
Match type	Prefix	Suffix	Substring	Exact
Python operator	`startswith`	`endswith`	`contains`	`==`

Table 5.2: Supported wildcard pattern types and their corresponding Python string comparison operators

Personalizing the summary

To allow for personalization of the summary, the identity of the person requesting it must be known. This identity can be configured by specifying the GitHub/GitLab username in the `summary-config.yml`. If the approach were to be integrated into a frontend with identity management, it could also be derived from the logged in user. Similar to the author filter, the username first needs to be mapped to an ID from the mentioned identity database as author information is stored in IDs. In a first step, all of the requestor's own changes (i.e., PRs and commits) are excluded from the summary by default when the username is specified, using the same methods applied when filtering by a specific author. This behavior can be configured to turn off the filter.
In the second step, the previously generated file blame for the PR is filtered to contain only files that the requesting user has previously modified. For every file changed in the PR, this filtered map is then checked to see if it includes the corresponding filename, indicating that the PR has modified a file that the requestor has previously touched. For files that were renamed in the PR, the old filename is checked as the previous modification to a file would be stored under the old name. If there is more than one match, another collapsible section "Modified files you made recent changes on" is added with bullet points for each affected file, containing the previous and the PR's modification, e.g. "This PR **modified** `server/src/Models/User.ts`, a file you recently **added**."

5.3.4 Analyzing overall file changes

The summary contains three sections for changes to individual files, namely (1) the files with the most line changes, (2) changes to configuration files and (3) changes to documentation. To extract these changes, the file changes from all PRs and commits are gathered, grouped by filename, and the sum of their additions, deletions, and total changes (i.e. sum of additions and deletions) are computed, respectively.
For the most changed files, the resulting list is ordered by total changes in descending order. The five most changed files are then formatted into a bullet point list, with each line containing the filename, deletions, additions, and a link to the GitHub/GitLab UI for file history to allow fast access to the commits that changed the file.
For the configuration and documentation changes, the `summary-config.yml` allows specifying a list of file extensions identifying the respective file type, with defaults (found in Appendix E) for common types such as `.md` for Markdown documentation files or `.yml` for configuration files. The ordered list of files is then filtered to only contain files that end with one of the specified file extensions and a bullet point list is created for both documentation and configuration files, with the same formatting as before.

5.3.5 Formatting

After the content of all individual sections has been generated, the final step assembles the Markdown document from them. However, when many repositories are summarized, e.g. in environments where a single author might be working on many at once repositories, more sections might be left empty as was no activity by a given author. To reduce the amount of empty sections, the configuration allows setting a flag to remove them from the summary entirely and only show sections with data. This is configurable as a lack of data might also be of interest to some users.
For multiple repositories, a summary is generated for each repository and concatenated with the other repositories, resulting in a single Markdown file. To distinguish the individual summaries, a title is added containing the repository name as well as the filters set for it's generation to improve the reader's understanding when the summary is read at a later point in time, e.g. "Summary of repositoryname from 2025-11-22 to 2025-12-05 for authorname". Finally, the resulting Markdown file is written in the current working directory. If this approach were to be extended, this final step could also be replaced by an alternative delivery method, e.g. sending an e-mail to a development team or uploading the file to a shared file storage.

5.4 LLM-enhanced summary

When the usage of LLMs is enabled in the configuration, the summary includes additional sections and features enabled by their capabilities. For this implementation, OpenAI's API was used with the `openai-python` library (OpenAI, 2025e), which requires the user to provide an API key.

5.4.1 LLM interface

To reduce dependence on a single vendor, the interface for LLMs was designed such that it can be replaced by other remote APIs or self-hosted LLM deployments. The following sections will describe the components created as part of this process.

LLM Requestor

The central interface to be used by clients and to be implemented for each vendor or self-hosted LLM is the `LLMRequestor` interface. The interface has three methods for conversation-based interaction, shown in Listing 5.4.

```
1 class LLMRequestor:
2     def start_conversation(self, message: str,
          system_prompt=None) -> LLMConversationState
3     def send_message(self, conversation:
          LLMConversationState, message: str) ->
          LLMConversationState
4     def batch_start_conversation(self, messages: list[str
          ], system_prompt=None) -> list[
          LLMConversationState]:
```

Listing 5.4: LLM Requestor interface

To start a conversation, `start_conversation` takes a first message and an optional system prompt, which can be used to give instructions and context of the task to the LLM. The return value is a `LLMConversationState`, a data class containing the conversation with all it's messages as well as convenience methods to retrieve the last response or print the conversation in a easy to read format for debugging. `send_message` allows the continuation of a previous conversation with another message, the response to which is appended to the previous conversation in the `LLMConversationState`. Whether the previous conversation is sent to the server on each `send_message` call is an implementation detail, as a vendor might also offer server-side handling of conversation state.
The function `batch_start_conversation` offers the same functionality as `start_conversation`, but allows providing multiple messages for each of which a conversation is started, e.g. when multiple independent data points should be processed with the same system prompt. This allows for an implementation that processes the requests in parallel, which can drastically speed up the process for large amounts of input messages compared to sequential processing. When more flexibility is needed for the client, there is also the option to implement the `LLMRequestorAsync` interface, which has `async` versions of `start_conversation` and `send_message` that allow running requests concurrently using Python's coroutines.
To enable clients to deal with input errors independently of vendor-specific exceptions, implementations should catch and wrap them with the implemented generic exceptions. These exceptions include the `LLMContextWindowTooLargeError` that should be raised when the size of the input exceeds the models context window, and `LLMCostLimitExceededError` which should be raised when either a server-side cost limit is hit or the locally configured one, the latter one will be discussed shortly.

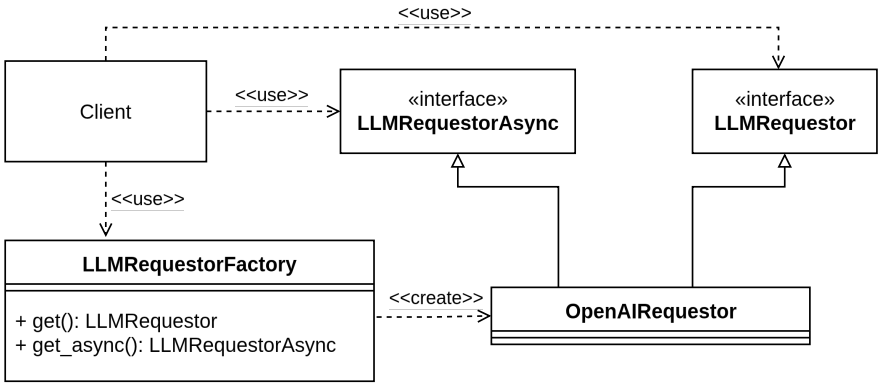


Figure 5.1: UML Diagram of the LLM interfaces

LLM Requestor Factory

An instance of a `LLMRequestor` implementation is obtained using the `LLMRequestorFactory`. Which provider and model is chosen is dependent on the configuration shown in Listing 4.1. This approach allows switching all LLM requests between different providers and models with just two lines of configuration changes, given an implementation of the interfaces already exists. It also allows disabling or enabling LLMs for the entire project by setting `enabled` to `false`, e.g. in environments where LLM usage is forbidden due to data protection concerns. If a client then tries to obtain an `LLMRequestor` using the factory, a `LLMUsageForbiddenError` is raised. Additionally, the configuration also allows setting a verbose log mode for debugging purposes. Figure 5.1 shows a UML diagram of the relationships between the `LLMRequestorFactory` and other components.

OpenAI Client

The `OpenAIRequestor` implements the `LLMRequestor` and `LLMRequestorAsync` interfaces by wrapping the OpenAI Responses API (OpenAI, 2025b). The openai-python library (OpenAI, 2025e) provides interfaces for both synchronous and asynchronous requests, which are used in the respective implementations of the interfaces. For `batch_start_conversation`, the implementation uses the asynchronous version to schedule all requests in parallel to speed up the total processing time. To handle rate limits, which become relevant when processing many concurrent requests, a retry mechanism for error conditions is implemented with an exponentially increasing wait time and a maximum of ten retries. This retry mechanism does not apply when the context window is exceeded, as this is an input error. OpenAI allows managing conversation state server-side by returning an ID with each response that can be provided on subsequent requests to maintain conversation state. The ID is stored in `LLMConversationState` and is

provided when `send_message` is called.

The implementation uses GPT-5-mini, the medium-sized model from OpenAI's frontier model family. The choice was made for two reasons. Firstly, as of December 2025, API pricing for input tokens when using GPT-5-mini is five times cheaper than for the larger model GPT-5, reducing the cost of the summarizing large PRs. More importantly, GPT-5-mini is claimed by OpenAI to be faster in serving requests, which is crucial for the execution speed of summary (OpenAI, 2025a). The models accept inputs of up to 400,000 tokens (OpenAI, 2025a), which corresponds to approximately 1.6 million characters of English text (OpenAI, 2025d).

Cost Limiter

The configuration optionally allows setting an upper limit in US dollars on how much a summary should cost. For this purpose, a `CostLimiter` component is created that is used to track spending and ensure the cost stays under the given limit. As costs are vendor-specific, they must be entered manually in a YAML configuration for each vendor and model to be used, as shown in Listing 5.5.

```
1 openai:
2   gpt-5-mini:
3     input: 0.25
4     cached: 0.025
5     output: 2.0
```

Listing 5.5: Vendor Cost Table

While retrieving the costs of each model automatically would be the optimal solution, OpenAI lacks an API that would provide such functionality, which may also be the case for other LLM providers. Although OpenAI offers an endpoint to retrieve costs in a given timeframe (OpenAI, 2025c), this endpoint also only allows the retrieval of costs after the request has already been billed and would have to be queried after each request. The response of requests to the Responses API (OpenAI, 2025b) also does not provide information about the cost of the request, but includes the number of input, output and cached tokens billed. Together with the cost table, these can be used to calculate the cost of the request. The cost of the output tokens is only known after the request, but the cost of input tokens can be estimated before to try to prevent requests exceeding the cost limit before they occur. A rough estimation can be made by dividing the number of input characters by four (OpenAI, 2025d) and multiplying it with the cost per token. If the sum of the accumulated cost and this estimate exceeds the cost limit, a `LLMCostLimitExceededError` is raised. After the request, the exact costs are calculated based on the token counts in the response and added to the

accumulated cost for the summary. Using this method, the maximum error is one output context window times the cost per token, assuming an average of four characters per token is accurate.

5.4.2 Project Summary

Before summarizing pull requests, a description of the project is generated. While Daneshyan et al. (2025) only used the README and the project's description to determine its domain, this implementation uses additional context if available by gathering documentation files and the file tree of the project. To do so, the repository's default branch is cloned and all files ending with the documentation file extensions specified in the configuration are collected and read. Additionally, the following Git command is run, which outputs a list of all files in the repository with their sizes.

```
1 git -C path/to/repo ls-tree --format "%(path) (%(
    objectsize) bytes)" -r HEAD
```

Listing 5.6: Git file tree command

The LLM is then given the repository name and owner, the files in the repository with their sizes, as well as all the documentation content with the goal of generating a short description of the project's purpose, main features, architecture and other unique aspects standing out from the documentation. The full system prompt can be found in Appendix B.
As the resulting query can get large, the response is cached in persistent storage and reused for the following requests. If the user wants to regenerate the response, they can delete the cached one to do so. If they want to specify a project description themselves or point to a specific file best summarizing the project, the configuration allows setting a path to such a project summary from the given repository.

5.4.3 Pull Request Summary

To generate a summary for a PR, all gathered context is given to the chosen LLM in a structured form. As LLM requests can take multiple seconds to process and the PRs are independent from each other, summarizing them is done in parallel using the `LLMRequestorAsync` interface. The system prompt, which can be found in full in Appendix C, instructs the LLM to shortly summarize the PR by first describing its overall changes and their impact, followed by grouping changes to individual parts of the project (e.g. frontend, backend) and summarizing them. Additionally, a section for breaking changes should be included if there are any in the given PR.

When the user has specified files they deem important in the `summary-config.yml`, the LLM is instructed to prioritize changes to these files in the summary. This is done by extending the system prompt with a section that both communicates this goal and lists the specified files in natural language, e.g. the pattern `"build/*"` is rendered in this list as "Files starting with build/". The instructions also state that reasoning of the changes from issues or the PR description should be included, but only if explicitly stated. Additional directives include formatting instructions, such as the usage of Markdown elements for class names, shortening of file paths and the usage of technical terms.

For each PR, this system prompt is sent with a message containing the following:

- The previously generated project summary for context.
- The PR title and description.
- The commit messages of the PRs commits.
- The title and description of the issues the PR closes.
- The PR's changed files, with each file's path, change type (e.g. added, modified) and diff (the previously extracted `patch` field).

For very large PRs or when using models with a small context window, the context may not fit into the context window of the model. To prevent crashes in this case, the resulting `LLMContextWindowTooLargeError` is caught and the summary is replaced with a message explaining the summary could not be generated and providing the PR description as a fallback. If the cost limit is exceeded (i.e., a `LLMCostLimitExceededError` is raised), this exception is handled in the same manner.

5.4.4 Semantic Ordering

After summarizing and formatting a section for PRs (i.e. Merged PRs or Open PRs), the LLM is instructed to first reason about the importance of each PR and then create an order from most important to least important. Since sections are provided to LLM the same way as they are provided to the user, this ordering can also make use of personalized warnings and highlighting. In the end of the system prompt (found in Appendix D), the LLM is instructed to output the ordered list of PR numbers in a JSON-style format (`{"order": [] }`) to facilitate the parsing of the response. An alternative solution would be to instruct the LLM to output the given inputs in the suggested order, but since this requires more output tokens and leaves room for erroneous reproduction of the input, the JSON formatted list was chosen as the preferred solution. After receiving a response, the string between the last occurrence of `"order":` and the following `}` is extracted, followed by removing all whitespace, newline and array delimiting characters

(`[` and `]`). The remaining string should now only contain the numbers delimited by a comma, which can now be used to create the order given by the LLM. If the response fails to include all PRs from the input, the missing PRs are appended at the end to ensure completeness. If the response exceeded a cost limit, context window, or the parsing of the response failed, the order remains unchanged.

6 Demonstration

To demonstrate the capabilities of the implementation, a summary will be generated for an exemplary GitHub repository. The used repository is from the AMOS project, an university course by the supervising chair that introduces students to agile practices with a semester long industry software development project. The specific project consists of an Android Automotive application designed for system level observability which communicates with a backend component implemented in Rust (The AMOS Projects, 2025). This repository was chosen because (a) it is a publicly available project, (b) meets the minimum standards defined in Chapter 3, and (c) has had sufficient amount of changes and participants to be considered realistic. The primary use case defined in Chapter 3 from the perspective of a developer will be demonstrated.

At the time of retrieval (18th of December 2025), the chosen repository has a total of 675 unique commits across its branches, 629 of which are on the main branch. The issue tracker contains are total of 114 issues, 108 of which have been closed. 150 PRs have been created and closed during the project, there are no open PRs at the time of retrieval. The project uses a branching strategy where features and fixes are merged onto an integration branch `dev` first and after each week long sprint, this branch is merged into the `main` branch of the repository from which a release is created.

As the batch job step also generates historical statistics for the repository, they are presented in Table 6.1. The execution environment is an Ubuntu 24.04 ma-

Metric	Mean	Minimum	Maximum
Changed lines	1090.4	0	12911
Changed files	21.2	1	169
Commits	8.4	1	94
Comments	2.2	0	23
Closing issues	0.1	0	3
Linked issues	0.4	0	4

Table 6.1: Statistics for demonstration repository rounded to one decimal place

chine with an AMD Ryzen 5 2600 processor and 32 gigabytes of memory. All components of the pipeline are run locally.

The demonstrated use case is from the perspective of a frontend developer that wants to know about the work of their peers during the last sprint. It is assumed that the data from GitHub has already been fetched overnight using the methods described in Chapter 5. To create this summary, the following configuration options are changed from their defaults:

- LLM usage is enabled.
- The `Branch` is set to `dev`, since this is the branch PRs are targeting.
- `From` and `To` are set to the beginning and end date of the particular sprint
- `Important Files` is set to `["frontend/*"]` as this is the repository's frontend directory and the frontend developer might be particularly interested in changes to their part of the project.
- `Requestor Username` is set to the GitHub username of the developer in question.
- All warning thresholds are set to `null` to use the historical statistics in Table 6.1 as a reference.

The full configuration used for generating the summary can be found in Appendix E. Running the pipeline with these parameters took 96 seconds on the described test setup and cost 0.021 US dollars of OpenAI credits. To preserve space, individual sections of the summary will be highlighted here, while a screenshot of the entire summary can be found in Appendix F.

The first line of the summary is the title, which includes the parameters set in the configuration and the repository's name.

```
Summary of amos2024ws03-android-zero-instrumentation from
2024-11-20 up to 2024-11-27 on branch dev
```

Listing 6.1: Summary title

The first section is the one line summary titled *TL;DR* (too long, didn't read), containing aggregated statistics of the summary's contents.

```
In 10 merged pull requests, a total of 3303 lines were added
and 1010 lines were removed. There are currently 0 open pull
requests.
```

Listing 6.2: One line summary

Figure 6.1: Pull request with highlighted file changes. Each entry consists of the filename, type of change, the number lines added and deleted.

Scrolling through the listed PRs in the merged PRs section, the a PR is highlighted as modifying files specified as important, and as such, it is inspected further to see how the changes affected the developer's part of the project. The preserve the Markdown style, this PR can be found in figure 6.1.

Figure 6.2: Summary of pull request #108

Since LLM usage is enabled, the developer can now learn about the changes that were made in context with other changes of the project. Although the author has only provided a short description of the PR referencing another PR that is stacked on top of it, the approach still generates a comprehensive summary of the changes, which can be seen in Figure 6.2. From this summary, the developer can now learn about the changes to the frontend part of the project and how they relate to the rest of the PR, i.e. the integration of the new feature into the frontend's

configuration management with a default value of `null` for compatibility. Since the prompt includes the important files specified in the configuration, changes to the frontend are moved to the top and explained file by file, with more detail than other changes. Additionally, the last section highlights breaking changes, which for PR #108 include changes to Protobuf schemas.

If the developer wants to dive deeper into these changes, they can click the PR number which is formatted as a link to the GitHub UI for inspecting the diff. Additionally, the commits are listed in a collapsible commits section, for this PR with only one commit.

```
- feat: sendmsg collection and config (393 days ago, +277 , -41 , Link)
```

Listing 6.3: Commits section of PR #108

Since the PR has not linked any issues as being closed by the PR, this section is omitted. Other than highlighted modified files in Figure 6.1, there were no warnings generated for the PR as it does not exceeded any of the thresholds inferred from the historical statistics of the project.

By default, empty sections of the summary are hidden and as such, the "Open PRs" and "Other commits" sections are omitted after filtering since all PRs were merged and all commits were introduced to the `dev` branch via PRs. If this option were to be changed to show them, the summary would include these sections with a text stating the absence of data.

As LLM usage was enabled in the configuration, the entries in the merged PRs section are also reordered to reflect their semantics. The following listing shows the first and last three entries of the ten merged PRs.

```
· #106: "Configuration rework" (360 additions, 304 deletions) by REDACTED
· #105: "88 vfs write new" (326 additions, 43 deletions) by REDACTED
· #108: "feat: sendmsg collection and config" (277 additions, 41 deletions) by
  REDACTED

                                ··· omitted entries ···

· #125: "feat: implement demo program for sendmsg" (486 additions, 1 deletions) by
  REDACTED
· #96: "chore: sprint-05 deliverables" (63 additions, 2 deletions) by REDACTED
· #94: "Sprint 05" (42 additions, 0 deletions) by REDACTED
```

Listing 6.4: First and last three entries of the summary

The first entry is large refactoring in the backend, while the following two are new features, with the latter being the one presented in detail earlier. It can also be seen that the ordering does not rely on lines of code changed, as PR #108 has less changes than #125, yet the latter is a demo program separate from the main code base and as such was given a lower priority. The last two PRs in the

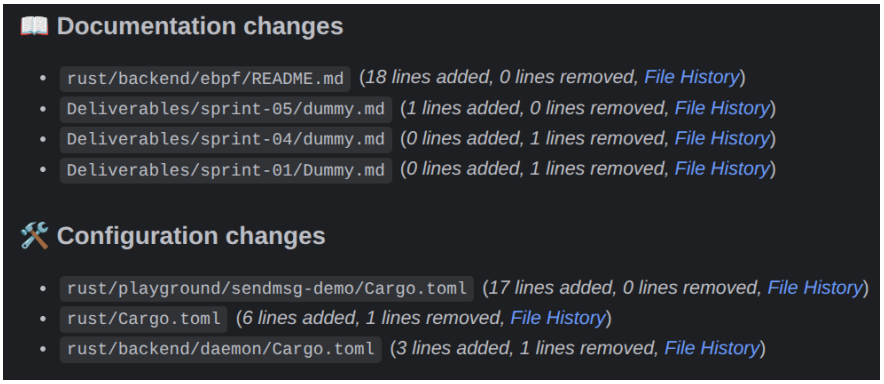


Figure 6.3: Documentation and configuration changes sections

summary include deliverables such as screenshots and PDF files mandatory in this university course, but no code changes, explaining their lower priority.

The last section of the summary is made up of the three sections for accumulated file changes. Focusing on the documentation and configuration changes, Figure 6.3 shows both sections with the respective files that were modified, given the configured filters. The documentation changes include two READMEs of the project as well as the removal and addition of dummy files for the mentioned deliverables. Configuration changes include the addition of a new Rust build file for the demo program as well as changes to existing ones. If the developer wants to further inspect the changes, they can do so by clicking the ”File History” link, taking them to the GitHub/GitLab UI that lists the commits that affected the respective file.

Since the input also included data from another repository and no repository filter was configured, it is summarized in the same file as well. However, after applying the timeframe filter, no records remained, and as such, its summary only consists of the one line summary section indicating the lack of data, as seen in Appendix F.

7 Evaluation

Following the methodological framework of Peffers et al. (2007), the constructed and demonstrated artifact must be evaluated against the objectives and requirements defined in Chapter 3.

Beginning with the requirements related to **summarization** and the content of the summary: the core requirement of structured textual summaries (RQ1) is clearly met. The previous chapter shows this by presenting the individual sections of the summary and their textual form. Another requirement concerned the completeness of the summary, defined in RQ3 as including all change sets and their associated closing issues that fall within the specified filters. Since the summary is based on a rule-based mechanism that does not exclude changes unless they are filtered out, the summary includes all changes, thereby avoiding any omissions that might arise when using a LLM to generate the structure. For the demonstration, the summary entries were manually cross-checked with the GitHub UI, confirming their completeness. RQ8 addressed the summarization of PRs, which was accomplished by using an LLM to summarize them, as demonstrated and illustrated in Figure 6.2, thus fulfilling this requirement. Lastly, RQ12 limited the use of third party APIs to GitHub and GitLab. Although the demonstrated artifact currently relies on OpenAI APIs to summarize PRs, the implemented generic interface enables the integration of self-hosted LLMs, thus eliminating the hard dependency on external APIs. Consequently, this requirement will be regarded as partially fulfilled.

Turning to the requirements concerning **applicability**, RQ5 required support for GitHub and GitLab, the two most popular code hosting platforms (JetBrains, 2021). By implementing an extractor for both platforms and transforming the data into a unified schema, the artifact supports both platforms, fulfilling the requirement. If the input contains data from multiple repositories, the artifact generates summaries for each repository within the same output, in a single run and configuration, as required by RQ6. Finally, the approach should not be constrained to a set of programming languages (RQ7). This has been achieved by relying solely on metadata from GitHub and GitLab for the rule-based summary and by using a generic LLM for additional summarization, which is again not

tied to any specific set of languages. The demonstration also shows this in Figure 6.2 by presenting a summary of a PR containing changes to Rust, Kotlin and Protobuf files.

Lastly, the goal of **personalization** introduced user-configurable filters. All filters specified in RQ2 were implemented, with the author and time frame filters showcased in the previous chapter. RQ10 and RQ11 further required the analysis and highlighting of change sets to draw attention to specific changes. The artifact addresses this in two ways. It first generates historical statistics to identify outliers in PRs, and highlights them in the summary, as can be seen for PR #106 seen in Appendix F, thus fulfilling RQ10. Secondly, it allows users to specify configurations for highlighting changes to particular files, as demonstrated for frontend files in the previous chapter (RQ11). Due to the personalized and on-demand nature of the summaries, RQ9 specified a runtime limit of under 60 seconds. This requirement was not met in the demonstration, as generating the summary took more than 60 seconds. To investigate why this was the case, the runtime of the individual steps is measured in order to identify steps for potential optimization.

The measurements show that most of the runtime is spent waiting for LLM requests. The parallel summarization of ten PRs took 30.7 seconds, while the following reordering took 31.8 seconds, surpassing 60 seconds on LLM requests alone. Logging the response times of individual requests shows that some requests can take as little as five seconds, but larger PRs can take more than 30 seconds. The reordering step involves reasoning about the impact of each change to improve the quality of the result. However, this reasoning leads to a longer processing time as more output tokens need to be generated, thus taking around 31 seconds to process the single request. While the implementation has been optimized to improve the speed of these requests, e.g. by parallelizing PR summaries and using the smaller GPT-5-mini model (OpenAI, 2025a), the potential for optimization is limited by the processing time of single requests.

To summarize, twelve of the 13 requirements specified in Chapter 3 were fulfilled. All requirements classified as ”Must” or ”Should” were implemented successfully, with the exception of discussed runtime requirement, which was prioritized as ”Should”. To accelerate the generation of the summary, future work could investigate the performance on even smaller models, evaluate processing times on self-hosted LLMs, or introduce caching mechanisms for generated PR summaries.

8 Conclusions

In summary, this thesis presents the creation of an artifact for generating personalized, developer-focused code contribution summaries from GitHub and GitLab repositories and their issue trackers, helping developers keep up with the work of their peers. In this chapter, limitations and constraints to the applicability of the approach will be presented, as well as an outlook on possible directions for future work.

8.1 Limitations

8.1.1 Usage of LLMs

While the usage of LLMs in the summary is optional, some of the features are missing when disabling them. This dependence on LLMs limits the applicability of the approach in environments with proprietary data, as organizations might be concerned about sending data from their development process to a third party application. Although some LLMs can be self-hosted and the design of the artifact supports their integration, costs can be higher for small-scale applications compared to using an API with pricing per token (Liagkou et al., 2024).
Another problem originating from the usage of LLMs is their potential to summarize the provided information in an incorrect, yet plausible sounding way. These so called ”hallucinations” (Huang et al., 2025) are inherently endangering the reliability of the approach as users might overly rely on the provided summary of a PR. As such, critical information may be overlooked if it was omitted, and readers might develop a wrong understanding of a PR's contents if it was incorrectly summarized.
Hallucinations could originate from a lack of context, as the only source code provided to the LLM for summarization is the diff, which includes only the lines surrounding the changed lines. However, some changes might require the context of the entire file or other files to be correctly summarized. While this is mitigated by providing additional context such as PR descriptions or commit messages and including instructions to not infer context that is not given in the prompt, there is no guarantee that the model will follow these instructions.

Finally, LLMs have a model-specific context window that limits the number of input tokens the model can handle (N. F. Liu et al., 2024). While there were no cases where the context window was exceeded during the testing of the artifact, the size of the context window differs and models with a smaller context window may not be able to summarize larger PRs.

8.1.2 Assumptions about repositories

During the generation of the summary, some assumptions are made about the quality of the input data in regards to common practices. Many of the features of the summary center around pull requests, thus limiting functionality of the approach in environments that do not use pull requests for contributions. The quality of the summary degrades when commit messages and PR titles are meaningless, PR descriptions are left empty or when PRs are not linked to the respective issues in the issue tracker. When the project does not have any documentation, the approach cannot determine the project context and architecture, which could result in worse quality of the LLM-generated PR summaries. While these are not hard constraints, they limit the effective use of the approach in environments with a lack of good practices regarding PRs, commit messages and issue trackers.

8.1.3 Lack of external validation

Peffers et al. (2007) suggests a variety of possible forms of evaluation for design science artifacts. In this thesis, the evaluation is intentionally limited to a comparison of the defined objectives and requirements with the artifact's demonstrated functionality. This evaluation is supported by manual comparison with data sources and quantitative analysis of runtime, focusing on adherence to the defined goals and requirements as derived from the identified problem and existing solutions. Other evaluation methods proposed by Peffers et al. (2007), such as satisfaction surveys that involve practitioners would address questions of perceived usefulness, usability, and effectiveness. While such methods could provide empirical evidence to support claims regarding feasibility of the approach for maintaining code change awareness, they were considered out of scope for this thesis and remain for future work.

8.2 Future work

8.2.1 Integration with other software development tools

While the market share of Git is large enough to cover the vast majority of users (Stack Exchange, 2022), the landscape of Git hosting platforms and especially

issue tracking systems is more diverse and can also lead to different combinations. In addition to GitHub and GitLab, which are the platforms supported by the artifact presented here, Bitbucket is another widely used Git hosting service, particularly in corporate environments (JetBrains, 2021). In regard to issue tracking systems, Jira is an established solution in organizations which the artifact does not support (JetBrains, 2021). As users might want to have a separate issue tracker from their Git hosting service, future work could implement integrations for these platforms and investigate their interoperability, for example when using GitHub as a Git hosting service with Jira as the issue tracker.

8.2.2 Improved personalization

In the presented approach, PRs are analyzed for changes to files that the summary requestor has previously modified, while also supporting the manual selection of relevant files. Although this strategy is computationally inexpensive, more sophisticated techniques could increase precision and reduce the reliance on manual configuration. Future work could try improving the granularity of how prior modifications by a given requestor are detected, for example by comparing changes line by line using the output of `git blame`, which would likely increase precision for large files. Another potential direction is to determine an "owner" for files that are frequently, primarily, or exclusively edited by a single person, or for which that person has contributed a majority of the number of lines. This ownership information could then be used to automatically derive a baseline set of important files that currently must be specified manually, reducing manual configuration.

8.2.3 Retrieval of additional context at runtime

There have been several approaches to providing LLMs with the ability interact with their environment at runtime, e.g. using calls to tools that allow web searches (J. Liu et al., 2025). During the design of the LLM-enhanced summary, a prototypical implementation of such a system was tested using the Model Context Protocol (Anthropic, 2024) with servers for Git and file system access, meaning the LLM was able to self issue Git commands or view the content of additional files in the summarization process. Although this approach was ultimately deemed unsuitable due to its substantially higher costs and, in particular, runtime, it nevertheless represents a worthwhile direction for future work under less restrictive time and cost constraints. A vast body of research on these so-called LLM-based agents has demonstrated significant potential in other software engineering tasks (J. Liu et al., 2025), indicating a promising avenue for future work in automated code contribution summarization.

Appendices

A GitHub GraphQL API Query

The following is the used for extracting data using the GitHub GraphQL API. The fields marked as "FILLED" are filled during the data retrieval, e.g. with the repository name and owner and during iteration over the paginated endpoint.

```
query {
    repository(owner: "FILLED", name: "FILLED") {
        pullRequests(first: FILLED, after: FILLED) {
            nodes {
                    title,
                    bodyHTML,
                    number,
                    additions,
                    deletions,
                    totalCommentsCount,
                    closingIssuesReferences(first:10) {
                      nodes {
                        number,
                        repository {
                            name,
                            owner {
                                login
                            }
                        }
                      }
                    },
                    commits(first: 100) {
                      nodes {
                            commit {
                                author {
                                  name
                                },
                                oid,
                                message,
                                messageHeadlineHTML,
                                messageBodyHTML,
                                url,
                                changedFilesIfAvailable,
                                authoredDate,
                                additions,
                                deletions
                            }
```

```
                        }
                    }
                },
                pageInfo {
                    hasNextPage,
                    endCursor
                },
                totalCount
            }
        }
}
```

B Project Summary system prompt

```
You are given basic information about a software project, the
file tree and the contents of its documentation files.
Summarize the project in a few sentences, focusing on its
purpose, main features, architecture and any unique aspects
that stand out from the documentation.
Do not derive any information that is not explicitly stated in
the provided data or obvious.
Be concise and informative.
```

C Pull Request Summary system prompt

In the following, the `[PERSONALIZED]` placeholder is replaced by the personalized section for important files described in Chapter 5.

```
You are given a pull request, along with its commits, file
    changes and linked issues.
Your task is to shortly summarize the pull request.
# Contents:
- It is very important to keep the summary short, but
    informative.
- First provide a short summary of what the pull request
    changes in 2-3 sentences, then explain changes to its
    individual parts shortly.
- Try not to summarize each file individually, but group
    related changes together.
- If the pull request description or issues contain the
    reasoning or additional information to the changes,
  focus on that, but do no try to guess the reasoning if it is
     not present.
- Explicitly mention breaking changes, but only if the pull
    request introduces some.
[PERSONALIZED]
# Formatting:
- Only answer with the summary, do not include any other text.
- Try to shorten paths instead of providing full paths.
- Use markdown formatting in your response, but do not add a
    headings or titles.
- Try to use code blocks for class names, filenames, or other
    technical terms where appropriate.
```

D Semantic Ordering system prompt

```
You are given a summary of pull requests and your job is to
    rank the pull requests in a order from most important to
    least important.
Prioritize changes marked as breaking.
Do not only rely on lines of code changed.
Please provide reasoning and in the end, output the numbers in
     order that you ranked them in JSON format like this: {"
    order": [] }.
```

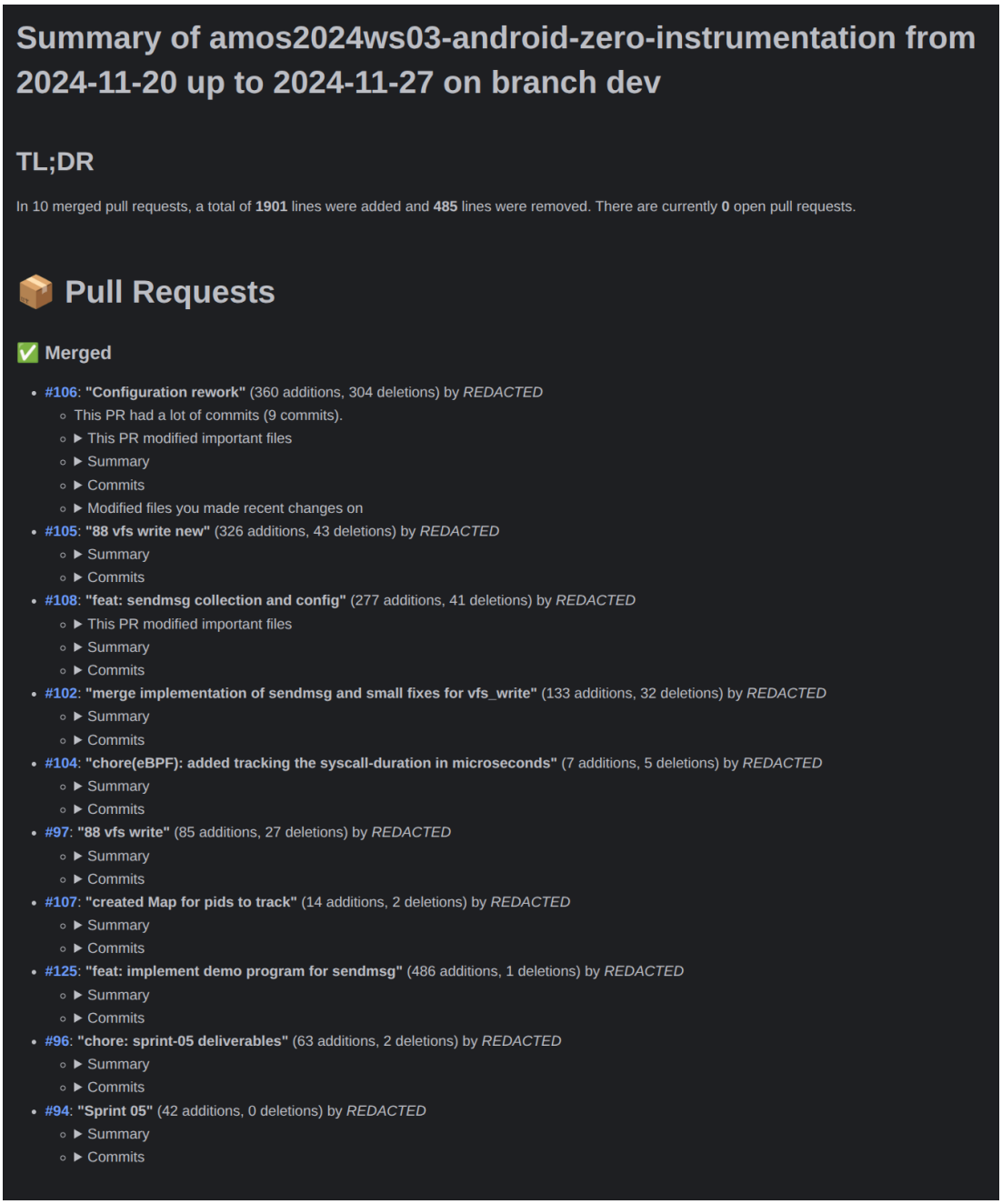
E Summary configuration used for demonstration

```
# Configuration for the pipeline's filters, null = all items are included
General:
    # The source of the data, can be either github or gitlab
    Source: github

    # Requestor username for personalized summaries, null = no personalization
    Requestor Username: luca-dot-sh

# Settings for LLM usage in the summary
LLM:
    Enabled: true

Filters:
    # Owners and repositorys to use in the summary
    # If this is an empty list, all repositories in the data lake will beused
    # If you want to specify repositories, you must specify Owner and Repo
    # Example:
    # Repositories:
    #   - Owner: Mecois
    #     Repo: mecois
    Repositories: []

    # Branch to use, if this is set to null, the default branch of the repository
        will be used
    Branch: dev

    # Start date filter data from in format: Year-Month-Day, null = no lower bound
    From: 2024-11-20

    # End date filter data from in format: Year-Month-Day, null = no upper bound
    To: 2024-11-27

    # Author to filter by (GitHub/Gitlab username, matching if author if is a
        substring), if null all authors are included
    Author: null


# Formatting settings for the summary
Formatting:
    # If true, empty sections (f.e. no pull requests) will be hidden in the summary
    Hide Empty Sections: true

    # If Requestor Username is set and this is set to true, hide contributions made
        by the requestor in the summary
    Hide Contributions by Requestor: true

# Settings that can be adapted for specific projects and their conventions
Project Specific:

    # Path to a project information file in the repository to summarize
    Project Summary Path: null

    # Define file extensions for different file types in the repository
    File Extensions:
        Documentation: [ ".md", ".adoc", ".rst", ".txt", ".html", ".tex"]
        Configuration: [ ".json", ".yaml", ".yml", ".xml", ".properties", ".conf", ".
            gitignore", ".env", ".toml" ]

    # Thresholds and paths for generating warnings in the summary
    # If github_repo_statistics is present in the datalake, these thresholds
```

```
# can be set to null to use the repository's average values
Warning Flags:
    Num Comments Threshold: null
    Num Commits Threshold: null
    Num Files Changed Threshold: null
    Num Lines Changed Threshold: null
    Num Referenced Issues Threshold: null

    # Specify important files that will be flagged when changed. This supports
        wildcards at the start and beginning
    # *name -> files that start with name, name* -> files that end with name, *
        name* -> files that contain name
    Important Files: ["frontend/*"]
```

F Overview of full summary from demonstration

For readability, the screenshot is split into two figures.



Summary of amos2024ws03-android-zero-instrumentation from 2024-11-20 up to 2024-11-27 on branch dev

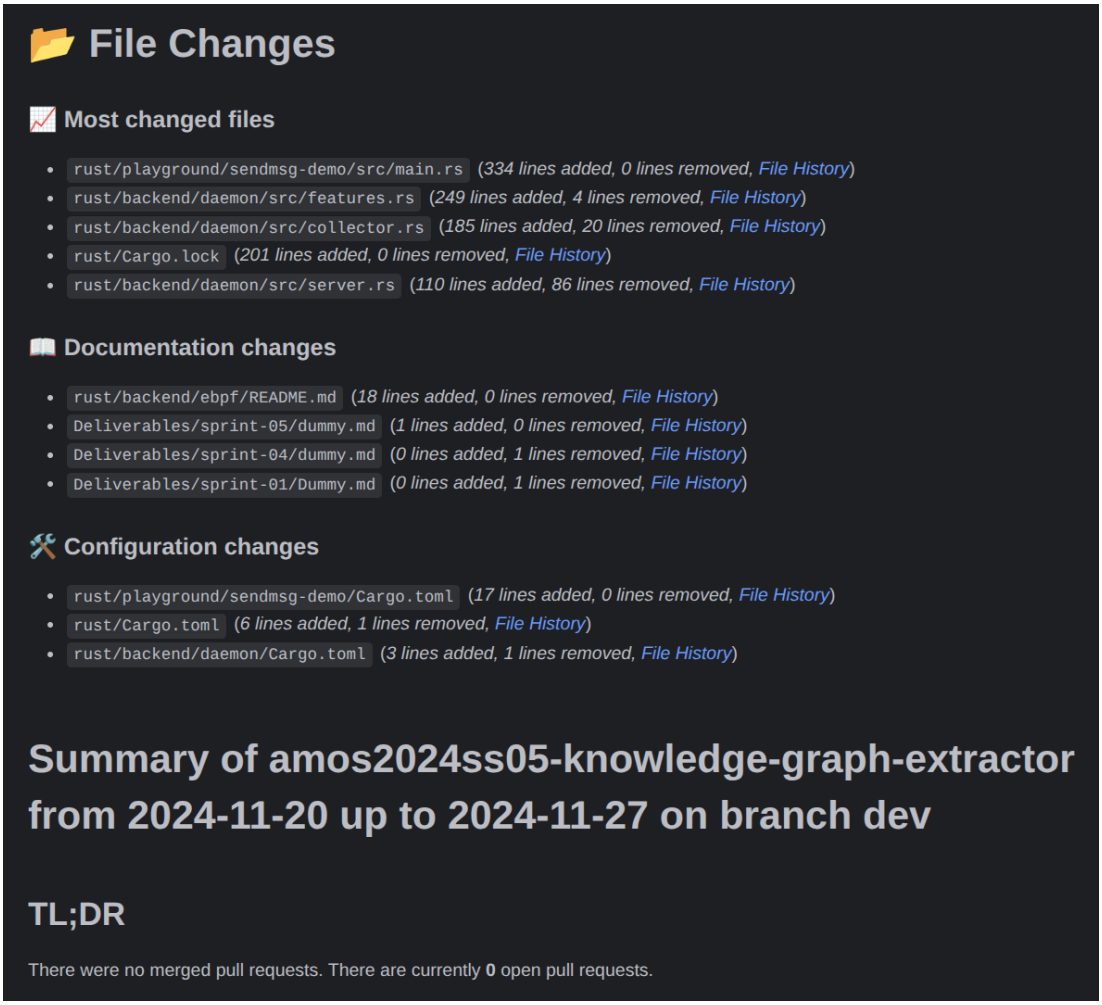
TL;DR

In 10 merged pull requests, a total of **1901** lines were added and **485** lines were removed. There are currently **0** open pull requests.

📦 Pull Requests

✅ Merged

- #106: **"Configuration rework"** (360 additions, 304 deletions) by *REDACTED*
  - This PR had a lot of commits (9 commits).
  - ▶ This PR modified important files
  - ▶ Summary
  - ▶ Commits
  - ▶ Modified files you made recent changes on
- #105: **"88 vfs write new"** (326 additions, 43 deletions) by *REDACTED*
  - ▶ Summary
  - ▶ Commits
- #108: **"feat: sendmsg collection and config"** (277 additions, 41 deletions) by *REDACTED*
  - ▶ This PR modified important files
  - ▶ Summary
  - ▶ Commits
- #102: **"merge implementation of sendmsg and small fixes for vfs_write"** (133 additions, 32 deletions) by *REDACTED*
  - ▶ Summary
  - ▶ Commits
- #104: **"chore(eBPF): added tracking the syscall-duration in microseconds"** (7 additions, 5 deletions) by *REDACTED*
  - ▶ Summary
  - ▶ Commits
- #97: **"88 vfs write"** (85 additions, 27 deletions) by *REDACTED*
  - ▶ Summary
  - ▶ Commits
- #107: **"created Map for pids to track"** (14 additions, 2 deletions) by *REDACTED*
  - ▶ Summary
  - ▶ Commits
- #125: **"feat: implement demo program for sendmsg"** (486 additions, 1 deletions) by *REDACTED*
  - ▶ Summary
  - ▶ Commits
- #96: **"chore: sprint-05 deliverables"** (63 additions, 2 deletions) by *REDACTED*
  - ▶ Summary
  - ▶ Commits
- #94: **"Sprint 05"** (42 additions, 0 deletions) by *REDACTED*
  - ▶ Summary
  - ▶ Commits



📂 File Changes

📈 Most changed files

- `rust/playground/sendmsg-demo/src/main.rs` (*334 lines added, 0 lines removed, File History*)
- `rust/backend/daemon/src/features.rs` (*249 lines added, 4 lines removed, File History*)
- `rust/backend/daemon/src/collector.rs` (*185 lines added, 20 lines removed, File History*)
- `rust/Cargo.lock` (*201 lines added, 0 lines removed, File History*)
- `rust/backend/daemon/src/server.rs` (*110 lines added, 86 lines removed, File History*)

📖 Documentation changes

- `rust/backend/ebpf/README.md` (*18 lines added, 0 lines removed, File History*)
- `Deliverables/sprint-05/dummy.md` (*1 lines added, 0 lines removed, File History*)
- `Deliverables/sprint-04/dummy.md` (*0 lines added, 1 lines removed, File History*)
- `Deliverables/sprint-01/Dummy.md` (*0 lines added, 1 lines removed, File History*)

🛠️ Configuration changes

- `rust/playground/sendmsg-demo/Cargo.toml` (*17 lines added, 0 lines removed, File History*)
- `rust/Cargo.toml` (*6 lines added, 1 lines removed, File History*)
- `rust/backend/daemon/Cargo.toml` (*3 lines added, 1 lines removed, File History*)

Summary of amos2024ss05-knowledge-graph-extractor from 2024-11-20 up to 2024-11-27 on branch dev

TL;DR

There were no merged pull requests. There are currently **0** open pull requests.

G Field Remapping from GitLab to GitHub

The following table shows the mapping of GitLab column names to GitHub column names for PRs/Issues and Commits. Structured fields have been flattened using underscores. The list only includes columns that were renamed without changing their values from the respective REST APIs.

GitLab	GitHub	Data Source
`iid`	`number`	PRs & Issues
`description`	`body`	PRs & Issues
`web_url`	`html_url`	PRs & Issues
`diff_refs_base_sha`	`base_sha`	PRs & Issues
`target_branch`	`base_ref`	PRs & Issues
`id`	`sha`	Commits & Commits in PRs
`message`	`commit_message`	Commits & Commits in PRs
`authored_date`	`commit_author_date`	Commits & Commits in PRs
`web_url`	`html_url`	Commits & Commits in PRs
`author_email`	`commit_author_email`	Commits & Commits in PRs
`additions`	`stats_additions`	Commits & Commits in PRs
`deletions`	`stats_deletions`	Commits & Commits in PRs

References

Abebe, S. L., Ali, N., & Hassan, A. E. (2016). An empirical study of software release notes. *Empirical Software Engineering*, *21*(3), 1107–1142. https://doi.org/10.1007/s10664-015-9377-5

Ali, M., Aftab, A., & Buttt, W. H. (2020). Automatic release notes generation. *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*, 76–81. https://doi.org/10.1109/ICSESS49938.2020.9237671

Anthropic. (2024). *Introducing the Model Context Protocol*. Retrieved November 22, 2025, from https://www.anthropic.com/news/model-context-protocol

Badampudi, D., Unterkalmsteiner, M., & Britto, R. (2023). Modern code reviews—survey of literature and practice. *ACM Trans. Softw. Eng. Methodol.*, *32*(4). https://doi.org/10.1145/3585004

Bi, T., Xia, X., Lo, D., Grundy, J., & Zimmermann, T. (2022). An empirical study of release note production and usage in practice. *IEEE Transactions on Software Engineering*, *48*(6), 1834–1852. https://doi.org/10.1109/TSE.2020.3038881

Codoban, M., Ragavan, S. S., Dig, D., & Bailey, B. (2015). Software history under the lens: A study on why and how developers examine it. *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 1–10. https://doi.org/10.1109/ICSM.2015.7332446

conventional-changelog Team. (2025). *Conventional-changelog*. Retrieved December 20, 2025, from https://github.com/conventional-changelog/conventional-changelog

Cortés Ríos, J. C., Embury, S. M., & Eraslan, S. (2022). A unifying framework for the systematic analysis of git workflows. *Information and Software Technology*, *145*, 106811. https://doi.org/10.1016/j.infsof.2021.106811

Daneshyan, F., He, R., Wu, J., & Zhou, M. (2025). Smartnote: An llm-powered, personalised release note generator that just works. *Proc. ACM Softw. Eng.*, *2*(FSE). https://doi.org/10.1145/3729345

De Miranda, F., Ferrao, R. C., Soler, D. P., & Vieira Graglia, M. A. (2024). Llm-based individual contribution summarization in software projects. *Proceed-*

ings of the 2024 on ACM Virtual Global Computing Education Conference V. 2, 307–308. https://doi.org/10.1145/3649409.3691092

Fang, S., Zhang, T., Tan, Y.-S., Xu, Z., Yuan, Z.-X., & Meng, L.-Z. (2022). Prhan: Automated pull request description generation based on hybrid attention network. *Journal of Systems and Software*, *185*, 111160. https://doi.org/10.1016/j.jss.2021.111160

GitHub. (2022a). *Rate limits for the REST API*. Retrieved November 29, 2025, from https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api?apiVersion=2022-11-28

GitHub. (2022b). *REST API endpoints for commits*. Retrieved November 25, 2025, from https://docs.github.com/en/rest/commits/commits?apiVersion=2022-11-28

GitHub. (2022c). *REST API endpoints for issues*. Retrieved November 25, 2025, from https://docs.github.com/en/rest/issues/issues?apiVersion=2022-11-28

GitHub. (2022d). *REST API endpoints for pull requests*. Retrieved November 25, 2025, from https://docs.github.com/en/rest/pulls/pulls?apiVersion=2022-11-28

GitHub. (2022e). *REST API endpoints for repository statistics*. Retrieved November 27, 2025, from https://docs.github.com/en/rest/metrics/statistics?apiVersion=2022-11-28

GitHub. (2025a). *Automatically generated release notes*. Retrieved August 15, 2025, from https://docs.github.com/en/repositories/releasing-projects-on-github/automatically-generated-release-notes

GitHub. (2025b). *Rate limits and query limits for the GraphQL API*. Retrieved November 29, 2025, from https://docs.github.com/en/graphql/overview/rate-limits-and-query-limits-for-the-graphql-api

GitHub. (2025c). *Responsible use of GitHub Copilot pull request summaries*. Retrieved December 21, 2025, from https://docs.github.com/en/enterprise-cloud@latest/copilot/responsible-use-of-github-copilot-features/responsible-use-of-github-copilot-pull-request-summaries

GitHub. (2026a). *About pull requests*. Retrieved January 4, 2026, from https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests

GitHub. (2026b). *Linking a pull request to an issue*. Retrieved January 5, 2026, from https://docs.github.com/en/issues/tracking-your-work-with-issues/using-issues/linking-a-pull-request-to-an-issue

GitLab. (2025a). *Commits API | GitLab Docs*. Retrieved December 3, 2025, from https://docs.gitlab.com/api/commits/

GitLab. (2025b). *Get started extending GitLab | GitLab Docs*. Retrieved December 3, 2025, from https://docs.gitlab.com/api/get_started/get_started_extending/#step-3-use-the-apis

GitLab. (2025c). *Issues API | GitLab Docs*. Retrieved December 3, 2025, from https://docs.gitlab.com/api/issues/#list-issues

GitLab. (2025d). *Merge requests API | GitLab Docs*. Retrieved December 3, 2025, from https://docs.gitlab.com/api/merge_requests/

Gradle. (2025). *Gradle kotlin DSL primer*. Retrieved December 23, 2025, from https://docs.gradle.org/current/userguide/kotlin_dsl.html

Huang, L., Yu, W., Ma, W., Zhong, W., Feng, Z., Wang, H., Chen, Q., Peng, W., Feng, X., Qin, B., & Liu, T. (2025). A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Trans. Inf. Syst.*, *43*(2). https://doi.org/10.1145/3703155

JetBrains. (2021). *The State of Developer Ecosystem in 2020*. Retrieved November 19, 2025, from https://www.jetbrains.com/lp/devecosystem-2021

Jiang, H., Zhu, J., Li, Y., Liang, G., & Zuo, C. (2021). Deeprelease: Language-agnostic release notes generation from pull requests of open-source software, 101–110. https://doi.org/10.1109/APSEC53868.2021.00018

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

Karrys, L. (2022, October). *Package-lock.json*. Retrieved November 4, 2025, from https://docs.npmjs.com/cli/v8/configuring-npm/package-lock-json

Kim, M. (2011). An exploratory study of awareness interests about software modifications. *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, 80–83. https://doi.org/10.1145/1984642.1984662

Klepper, S., Krusche, S., & Bruegge, B. (2016). Semi-automatic generation of audience-specific release notes. *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, 19–22. https://doi.org/10.1145/2896941.2896953

Ko, A. J., DeLine, R., & Venolia, G. (2007). Information needs in collocated software development teams. *29th International Conference on Software Engineering (ICSE'07)*, 344–353. https://doi.org/10.1109/ICSE.2007.45

Kravchenko, T., Bogdanova, T., & Shevgunov, T. (2022). Ranking requirements using moscow methodology in practice. In R. Silhavy (Ed.), *Cybernetics perspectives in systems* (pp. 188–199). Springer International Publishing. https://doi.org/10.1007/978-3-031-09073-8_18

Lee, D., Wentling, T., Haines, S., & Babu, P. (2024). *Delta lake: The definitive guide* (1st ed.). O'Reilly Media. Retrieved December 29, 2025, from https://www.oreilly.com/library/view/delta-lake-the/9781098151935/

Liagkou, V., Filiopoulou, E., Fragiadakis, G., Nikolaidou, M., & Michalakelis, C. (2024). The cost perspective of adopting large language model-as-a-service. *2024 IEEE International Conference on Joint Cloud Computing (JCC)*, 80–83. https://doi.org/10.1109/JCC62314.2024.00020

Liu, J., Wang, K., Chen, Y., Peng, X., Chen, Z., Zhang, L., & Lou, Y. (2025). Large language model-based agents for software engineering: A survey. https://arxiv.org/abs/2409.02977

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2024). Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, *12*, 157–173. https://doi.org/10.1162/tacl_a_00638

Liu, Z., Xia, X., Treude, C., Lo, D., & Li, S. (2020). Automatic generation of pull request descriptions. *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, 176–188. https://doi.org/10.1109/ASE.2019.00026

Mishra, A., & Alzoubi, Y. I. (2023). Structured software development versus agile software development: A comparative analysis. *International Journal of System Assurance Engineering and Management*, *14*(4), 1504–1522. https://doi.org/10.1007/s13198-023-01958-5

Moreno, L., Bavota, G., Penta, M. D., Oliveto, R., Marcus, A., & Canfora, G. (2017). Arena: An approach for the automated generation of release notes. *IEEE Transactions on Software Engineering*, *43*, 106–127. https://api.semanticscholar.org/CorpusID:15433385

OpenAI. (2025a). *GPT-5 mini model | OpenAI API*. Retrieved December 30, 2025, from https://platform.openai.com

OpenAI. (2025b). *Responses | OpenAI API Reference*. Retrieved December 6, 2025, from https://platform.openai.com/docs/api-reference/responses

OpenAI. (2025c). *Usage | OpenAI API Reference*. Retrieved December 6, 2025, from https://platform.openai.com/docs/api-reference/usage/costs

OpenAI. (2025d). *What are tokens and how to count them?* Retrieved December 6, 2025, from https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them

OpenAI. (2025e, December). *Openai/openai-python*. Retrieved December 6, 2025, from https://github.com/openai/openai-python

Peffers, K., Tuunanen, T., Rothenberger, M., & Chatterjee, S. (2007). A design science research methodology for information systems research. *J. Manage. Inf. Syst.*, *24*(3), 45–77. https://doi.org/10.2753/MIS0742-1222240302

Potvin, R., & Levenberg, J. (2016). Why google stores billions of lines of code in a single repository. *Commun. ACM*, *59*(7), 78–87. https://doi.org/10.1145/2854146

python-gitlab Team. (2025). *Python-gitlab v7.0.0*. Retrieved December 3, 2025, from https://python-gitlab.readthedocs.io/en/stable/

Ridley, B. (2025). *Tutorial: Automate releases and release notes with gitlab*. Retrieved August 15, 2025, from https://about.gitlab.com/blog/tutorial-automated-release-and-release-notes-with-gitlab/

Rigby, P. C., Rogers, S., Saleem, S., Suresh, P., Suskin, D., Riggs, P., Maddila, C., Nagappan, N., & Mockus, A. (2025). Improving code reviewer recom-

mendation: Accuracy, latency, workload, and bystanders [Just Accepted]. *ACM Trans. Softw. Eng. Methodol.* https://doi.org/10.1145/3736405

Sakib, M. N., Islam, M. A., & Arifin, M. M. (2024). Automatic pull request description generation using llms: A t5 model approach. *2024 2nd International Conference on Artificial Intelligence, Blockchain, and Internet of Things (AIBThings)*, 1–5. https://api.semanticscholar.org/CorpusID:271693486

Stack Exchange. (2022). *Stack Overflow Developer Survey 2022*. Retrieved November 15, 2025, from https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022#version-control-version-control-system-prof

The AMOS Projects. (2025). *Amosproj/amos2024ws03-android-zero-instrumentation*. Retrieved December 17, 2025, from https://github.com/amosproj/amos2024ws03-android-zero-instrumentation/

The Apache Software Foundation. (2025). *Apache spark™ - unified engine for large-scale data analytics*. Retrieved December 29, 2025, from https://spark.apache.org/

Wu, Y., Wang, Y., Li, Y., Tao, W., Yu, S., Yang, H., Jiang, W., & Li, J. (2025). An empirical study on commit message generation using LLMs via in-context learning. *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, 553–565. https://doi.org/10.1109/ICSE55347.2025.00091

Xiao, T., Hata, H., Treude, C., & Matsumoto, K. (2024). Generative ai for pull request descriptions: Adoption, impact, and developer interventions. *Proc. ACM Softw. Eng.*, *1*(FSE). https://doi.org/10.1145/3643773

Yu, Y., Wang, H., Yin, G., & Wang, T. (2016). Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology*, *74*, 204–218. https://doi.org/10.1016/j.infsof.2016.01.004

Zhang, L., Zhao, J., Wang, C., & Liang, P. (2024). Using large language models for commit message generation: A preliminary study [ISSN: 2640-7574]. *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 126–130. https://doi.org/10.1109/SANER60148.2024.00020

Zhang, Y., Qiu, Z., Stol, K.-J., Zhu, W., Zhu, J., Tian, Y., & Liu, H. (2024). Automatic commit message generation: A critical review and directions for future work. *IEEE Transactions on Software Engineering*, *50*(4), 816–835. https://doi.org/10.1109/TSE.2024.3364675

Zolkifli, N. N., Ngah, A., & Deraman, A. (2018). Version control system: A review. *Procedia Computer Science*, *135*, 408–415. https://doi.org/10.1016/j.procs.2018.08.191