# Code Review and Project Workflow Analysis for Git Data

Francesco Alzate, Davide Monaco, Massimo Francios
{s334061, s305133, s328914}@studenti.polito.it

January 29th, 2025

## 1 Introduction

In software development, tracking and understanding the evolution of a complex codebase is critical for maintaining code quality, guiding future development, and supporting informed decision-making.

Traditional code review processes, however, can struggle with the vast amount of changes, especially in complex projects with intricate histories, diverse team contributions, and numerous interdependencies. Large Language Models (LLMs) offer a promising approach to address this challenge by leveraging their capacity to analyze and interpret git commit messages, diffs, and tree structures, generating insights into the project's evolution and enhancing the code review process.

With their extensive language and pattern recognition capabilities, LLMs can process and analyze git data to summarize, categorize, and contextualize code changes over time. This provides development teams with structured, meaningful insights into commit intentions, the rationale behind changes, and the impact on the overall project. For instance, LLMs can identify and group commits related to specific features, highlight areas of frequent modification, or detect patterns in code refactoring and bug fixes, offering developers a clear view of how a project has evolved.

The main objectives of using LLMs for git analysis in complex projects include:

Commit Analysis and Summarization LLMs can review commit messages and code changes to create concise summaries of past work, simplifying the identification of significant updates and trends.

Evolutionary Insights and Reporting LMs can generate reports that outline the codebase's evolution, including insights into areas of technical debt, consistent patterns of code modification, and dependency trends.

This research aims to explore the use of LLMs for enhancing code review through git analysis, with the goal of improving developers' understanding of project evolution, facilitating more effective code reviews, and ultimately supporting the ongoing development and maintainability of complex software projects.

## 2 Research questions

RQ1: How effectively can LLMs summarize and interpret git commit messages to convey the intent behind code changes?

RQ2: How can LLMs facilitate better communication among team members by generating context-aware reports on project evolution?

# 3 Background

## 3.1 MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution

*MAGIS* is a framework that transforms the complex task of GitHub issue resolution into a collaborative effort, it incorporates 4 key roles for LLM-based agents working collaboratively in the workflow: **Manager** (responsible for team assembly, meeting organization, and plan formulation), **Repository Custodian** (tasked with identifying relevant files in the repository and tracking changes), **Developer** (responsible for code implementation) and **Quality Assurance (QA) Engineer** (ensuring code quality through review and feedback).

## 3.2 Can LLM Generate Regression Tests for Software Commits?

This study conducted by Liu et al. introduces *Cleverest*, a feedback-directed, zero-shot LLM-based approach for generating regression tests from code changes, such as commits and pull requests.

**Commit Analysis and Information Extraction:** *Cleverest* analyzes code changes by examining commit diffs and associated messages. This analysis focuses on identifying the intent behind the changes and the specific areas of the codebase affected.

**Prompt Generation for the LLM:** Based on the extracted information, *Cleverest* formulates prompts to guide the LLM in generating relevant test cases. These prompts are crafted to generate inputs that target the modified code paths, ensuring that the generated tests are effective in detecting potential regressions.

## 3.3 Code-Survey: An LLM-Driven Methodology for Analyzing Large-Scale Codebases

In this study conducted by Zheng et al. it's proposed *Code-Survey*, a methodology leveraging Large Language Models (LLMs) to analyze code changes through structured surveys. These surveys are designed to extract key information from commit messages, including the intent behind the changes, impacted files, and relevant metadata. The LLM uses these surveys as prompts to summarize the commit details, helping developers understand the purpose and scope of the changes.

# 4 Methodology

In this section we describe the details about the employed Language Models, Architectures and evaluation strategies. More about methodology, prompts and examples can be found in our repository [1].

## 4.1 Language Models

To evaluate the feasibility of using smaller models for git analysis, we employ **Llama 3.2-Instruct** [5] with 1 billion parameters as our main model. This decision aims to assess whether a lightweight

model could be useful in commit analysis while maintaining computational efficiency and low resource usage. Moreover, we try to balance efficiency and interpretability, ensuring good scalability for larger codebases.
To speed up the process of evaluating Llama results, we also employ **ChatGPT** [6] to quickly compare the actual output with the desired one, ensuring a more accurate and efficient revision. ChatGPT is used in a "fully supervised" setting, which means that ChatGPT outputs are cross-checked by human reviewers, addressing inconsistency or errors.

## 4.2 Architecture

In this section we describe the architecture used for the git analysis commits and its main components. Our framework is illustrated in figure 1

**Commits extractor.** First of all, we extract the git commits from the repository and preprocess them to remove irrelevant information. To do so, we created specific function to filter trivial commits, such as minor changes, merges, or readme updates. After filtering commits, we normalized the commit messages, enhancing consistency and improving the model's ability to undestand messages.

**Categorization chain.** We implemented a categorization chain in which the model has to predict a category for a single commits, choosing from a fixed list of categories. In this phase the model is able to see all relevenat information regarding the commit, including author, message, changed files, lines of code added and removed. We tested the model in both zero-shot and few-shots settings and compare results to evaluate both approaches.

**Summarization chain.** We implemented a summarization chain where the model has to generate summaries for a single commit, given all relevant information. Summaries were generated at two different levels: the first one, called "summary" in this work, refers to a high-level description of the commit. The second, called "Technical summary", focused on changed lines of code. Only the few.shots setup was used for this task, as the zero-shot setup was not particularly suited for this kind of task, due to lack of guidance in the generation.

**Quality Assurance Framework.** To ensure quality of the technical summaries, we develope an iterative approach inspired by MAGIS [2]. This approach has an iterative structure where a first LLM agent is responsible for the summary generation, while a second agent is responsible for evaluating the summary and assigning an overall score on a scale from 0 to 10. We set an empirical threshold to 8 where summaries with a score below threshold were not accepted. In this way we achieved **Reflection** and **Self-criticism** capabilites, ensuring a more accurate and reliable output.

**Story Generation.** Finally, we developed a Story mechanism to describe the project evolution based on commits. A detailed description is provided in 4.3.

**Evaluation.** For the categorization task we employed **precision and recall** metrics. For the summarization task we develop a quality metric **UISC** similar to INVEST based on 4 aspects, each rated from 0 to 5: Unambiguous, Informative (depth of information), Significance (meaningful information) and Coherent (logical flow).

## 4.3 Story Generation

In the context of developing a Story application to track and describe project evolution through commit messages, we aim to extract meaningful summaries for each commit. This involves identifying the key elements from both the technical and user-friendly summaries, which are derived by

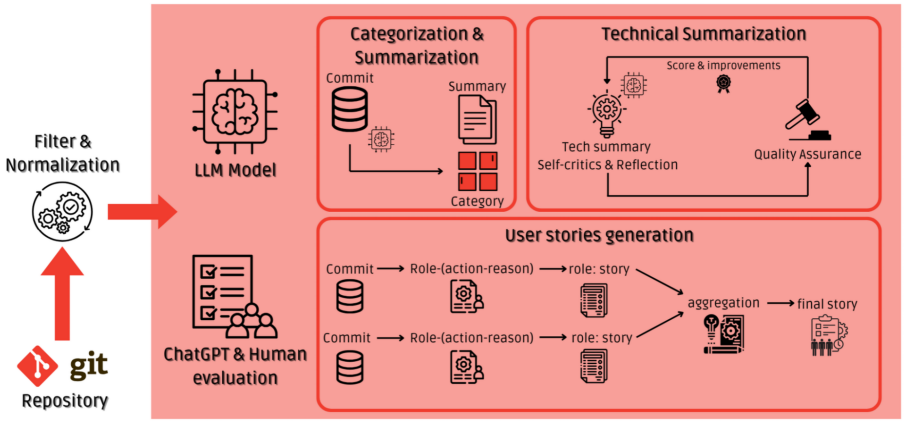


Figure 1: High-level Architectute developed to analyze Git Commits

translating code changes into concise descriptions. These elements include the subject ("who" the commit is about), the goal ("what" is being achieved by the commit"), and the reason or benefit ("why" the change is valuable"). By structuring commit messages in this way, the application generates summaries that clearly convey the project's technical advancements and broader purpose.

To achieve this, we implemented a prompt that analyzes input data, such as user-friendly or technical descriptions, and modifies it to generate concise stories. The prompt extracts the key elements—subject, action, and reason—and maps them to specific stakeholder perspectives (e.g., developer, tester, project manager). Each story is then presented in the format: **As a [role], I want [goal] so that [reason/benefit]**.

In the Readme.txt is defined how the prompt generates each story [1], both through a technical summary and a user-friendly version.

In this way, the prompt extracts the essence of the technical changes and presents them in terms that are understandable and actionable for each role involved in the project. After generating results in the format (Role, tuple(Action, Reason/Benefit)), an additional prompt is applied to transform these into concise short stories. These stories serve to further contextualize and personalize the role-based summaries, making them relatable and highlighting the positive outcomes of each action. By turning abstract or technical actions into narratives, this step humanizes technical data, creating a relatable and accessible presentation for all stakeholders. Each story is crafted with a role-centric perspective, centering on a specific role ( e.g developer or tester) and framing their contributions in a way that underscores their importance to the project. The narratives emphasize how actions lead to tangible improvements or benefits, reinforcing the value of the changes described in commits and providing a clear picture of their real-world impact.

#### 4.3.1 Story aggregator

Story Aggregation involves concatenating the sequential narratives of each stakeholder based on their history across commits. This approach effectively captures the chronological progression of changes, making it easier to trace how a project has evolved over time. However, it introduces several challenges. For instance, as each commit reflects a specific moment in the project's devel-

opment, changes in perspective, focus, or tone can result in a fragmented or disjointed narrative. Stakeholder roles may shift subtly or even drastically between commits, creating discordance in the story. Redundant or conflicting details may also arise, as subsequent commits refine or rework previous ideas without explicitly resolving discrepancies. Furthermore, the lack of contextual integration between commits can dilute the overall coherence of the narrative, making it harder to identify meaningful patterns or actionable insights. These limitations suggest that while Story Aggregation is straightforward, it may fail to produce a fluid, cohesive storyline that fully reflects the stakeholder's evolving role and goals within the project.

# 5 Results

In this section, we analyze the results of our architectures to analyze git commits. As a case study we used **MuJS** [7], a lightweight, embeddable Javascript interpreter written in C. MuJS is an open-source project with a public git repository. With a relatively compact codebase, it has a reasonable amount of commits and good complexity, making it a perfect candidate for our analysis, as it allows our architecture to deal with real-world scenarios.

## 5.1 Effectiveness in interpreting git commit messages

In this section we report the results of our experiments and we describe results related to Research question 1. In figure 2 we report the commit classification, where the model chooses between a fixed list of categories.

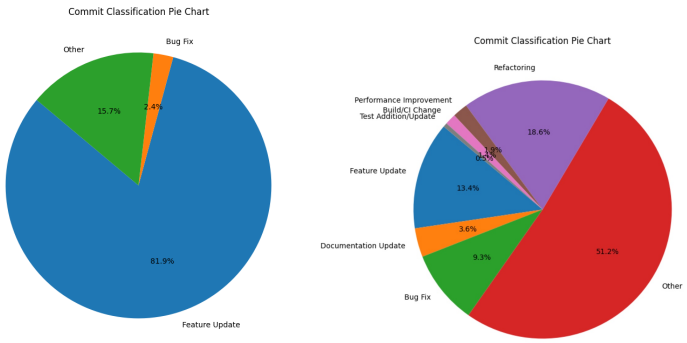


Figure 2: Zero vs Few-shots commits category distribution

Table 1 shows the precision and recall metrics. In the zero-shot setup we obtained a precision value of 0.44 and a recall value of 0.80: the model is less accurate in predicting relevant instances. A higher recall suggests that the model may include irrelevant instances, too. In zero-shot setup, the model relies on pre-trained knowledge, leading to more general outputs. In the few-shots setup, precision increases while recall is lower. This is probably due to the guidance of the prompt: the model prioritizes precision when fed with examples.

When evaluating summaries, we set 3 levels of quality based on our metric score $S$: mediocre , good and excellent quality. Table 2 reports the number of mediocre, good and excellent quality summaries for both levels. For this particular experiment, we employed only few-shots prompting,

| Setup | Precision | Recall |
|---|---|---|
| Zero-Shot | 0.44 | 0.80 |
| Few-Shot | 0.59 | 0.46 |

Table 1: Precision and Recall for Zero-Shot and Few-Shot categorization setups.

as we noticed zero-shot setup was not particularly suited for this kind of task, due to lack of guidance in the generation. We restricted our study to 100 commits from the MuJS repository.

| Type | **Mediocre** $S <= 12$ | **Good** $12 < S < 17$ | **Excellent** $S >= 17$ |
|---|---|---|---|
| Summaries | 18 | 39 | 43 |
| Technical Summaries | 16 | 28 | 56 |

Table 2: Number of mediocre, good, excellent quality summaries according to the UISC metric.

The results related to the summaries demonstrates good capabilites of the model to summarize information when guided from examples in the prompt. The iterative approach from MAGIS [2] used for tehcnical summaries paid off as we obtained an higher number of excellent summaries. Regardless the type of prompt, the model can make mistakes and provides irrelevant information. This could be due to an high number of lines of code in the prompt.

### 5.2 Effectiveness in generating correct descriptions of applied modifications to codebases

To track project evolution, commit messages from 10 commits are analyzed to generate both user-friendly and technical stories for each role. These stories are organized in a dictionary where each key represents an author. For every author, the dictionary contains user-friendly and technical summaries, each highlighting the main stakeholder and listing the stories corresponding to each commit. Below, we describe the results of the **Story Aggregation** [1]. It is clear that the model generalizes well across different commits, providing user-friendly summaries even when the commits belong to distinct categories. The bullet points highlight changes in the commit messages or subjects. However, despite the variation in the subject of each bullet point, the story lacks fluency and does not follow a clear timeline of events. This issue presents a significant limitation of the current method. For the technical summaries, three examples are described. In the case of the software architect and tester roles, we observe concise and clear stories. This may be due to these roles being less prominent in the 10 commits analyzed, leading to fewer samples. On the other hand, the developer role presents a story that stands out as an outlier. This could be because the generated response reflects the prompt used to create the story. Specifically, in the few-shot approach, the method likely copied this behavior, structuring the answer in the form of examples rather than following a more generalized pattern.

## 6 Discussion

In this section we describe results, the limitation of our approach, and future investigation that could enhance solutions for the problem.

**Conclusions.** Our experiments demonstrates that LLMs have a good potential in analyzing git data, generating commit categories, and summaries, allowing to track project evolution. However, our results prove that there are some limitations and possible improvements: we achieved satisfying results in few-shots setup for categorization and summarization but, narratives for tracking project evolution leave room for improvements.

In particular, our approach is able to capture individual commit details, but often it leads to irrelevant outputs when trying to combine them together into a global narrative. This limitation is related to the fact that the examined model is relatively small and distilled from a larger one. This is even more problematic when the examined codebase gets larger, leading to scalability and coherence issues.

Our approach exhibits variability across different task: while it is more consistent in few-shots setup it still provides irrelevant or redundant information.

**Future investigations.** A first possible improvements could be leveraging a larger LLM model to conduct experiments, as it could improve overall accuracy for categorization, coherence between narratives and better summaries. With our choice we wanted to prioritize computational efficiency, but employing a model with higher parameter number allows a better understanding of git commit's context. Similar to the technical summaries framework, leveraging more than one agent for reviewing stories could enhance the ability of creating narratives to describe project evolution. A promising direction for improving story generation could be implementing a **RAG** architecture to minimize misleading outputs in narratives.

Overall, our approach demonstrates encouraging outcomes, but the problem remains partially solved. With future investigations, LLM agents can evolve and become necessary tools for improving workflows and project development.

# References


[1] Alzate, Francios, Monaco. "Code Review and Project Workflow Analysis for Git Data." Available at: `https://github.com/maxfra01/code-review-and-project-workflow-analysis-for-git-data`

[2] Tao, Wei, et al. "MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution." Available at : `https://arxiv.org/pdf/2403.17927`

[3] Liu, Jing, et al. "Can LLM Generate Regression Tests for Software Commits?" Available at: `https://nimgnoeseel.github.io/resources/paper/llm-regression-paper.pdf`

[4] Zheng, Yusheng, et al. "Code-Survey: An LLM-Driven Methodology for Analyzing Large-Scale Codebases." Available at: `https://arxiv.org/html/2410.01837v1`

[5] Meta AI. "Llama 3.2 Instruct." Available at: `https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct`.

[6] OpenAI. "ChatGPT." Available at: `https://openai.com/chatgpt`.

[7] Tor Andersson and contributors. "MuJS: An Embeddable JavaScript Interpreter in C." Available at: `https://github.com/ccxvii/mujs`.