The Journal of Systems and Software 230 (2025) 112569

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A novel LLM-based classifier for predicting bug-fixing time in Bug Tracking Systems ☆

Pasquale Ardimento [a],*, Michele Capuzzimati [a], Gabriella Casalino [a], Daniele Schicchi [b], Davide Taibi [b]

[a] Department of Informatics, University of Bari Aldo Moro, Via Orabona 4, Bari, 70124, Italy
[b] Institute for Educational Technology, National Research Council of Italy, Via Ugo la Malfa 153, 90146 Palermo, Italy

ARTICLE INFO

Keywords:
Bug fixing time
Software maintenance and evolution
Large language models
Zero-shot learning

ABSTRACT

Predicting whether a newly submitted bug will be resolved quickly or slowly is a crucial aspect of the bug triage process, as it enables project managers to estimate software maintenance efforts and manage development workflows more effectively. This paper proposes a deep learning approach for classifying bug reports into two categories—*FAST* or *SLOW*—based on their expected fixing time. The method leverages a feature set composed of the bug description and reporter comments and adopts a transfer learning strategy using pre-trained Large Language Models (LLMs). The problem is framed as a supervised text classification task, where LLMs exploit their ability to learn rich contextual representations of language. We introduce a novel classification workflow that guides the LLM through a structured prompt, combining two design patterns: the persona pattern to contextualize the task and the input semantic pattern to organize textual information. The workflow relies on zero-shot learning to assess whether the intrinsic knowledge embedded in the LLMs is sufficient for this prediction task. We conducted a comprehensive evaluation of three state-of-the-art LLMs across multiple real-world datasets sourced from Bugzilla, encompassing a diverse range of software projects. The experimental results demonstrate that the proposed method is effective in accurately identifying fast-resolving bugs. Among the evaluated models, `LLaMA3-8B` consistently delivered superior performance. Additionally, the absence of statistically significant performance variations across datasets highlights the generalizability of the approach. Notably, the LLMs maintained strong performance even on small and imbalanced datasets, underscoring their robustness and practical applicability in real-world, data-scarce scenarios.

1. Introduction

Bug tracking systems (BTSs) play a crucial role in software development by allowing teams to systematically record, monitor, and manage software defects. One of the primary challenges in this process is accurately estimating the fixing time for reported issues, ensuring that the assigned timelines reflect the complexity of the problem. In this work, fixing time is defined as the number of days between the date the bug report is assigned to a developer and the last date on which it is marked as FIXED. This definition focuses on the actual period of developer activity, excluding the initial delays typically introduced by bug triage activities or the time between submission and assignment. This definition is consistent with literature that distinguishes fixing effort from broader resolution time, notably (Bhattacharya and Neamtiu, 2011; Gomes et al., 2021; Xia et al., 2016; Marks et al., 2011; Xuan et al., 2012).

Effectively assigning bug reports based on their priority can significantly improve the triage process, leading to more efficient defect management that improves the quality of software maintenance and contributes to the longevity and sustainability of software systems (Uddin et al., 2017). Incorporating artificial intelligence (AI) into BTS has greatly enhanced the efficiency and precision of defect management. This is achieved by automating essential tasks like bug triaging, priority assignment, and estimating fixing times. Using these technologies, AI reduces human biases and errors in manual bug classification, speeds up the software maintenance process, and optimizes resource allocation, leading to lower development costs and higher software quality (Nagwani and Suri, 2023; Bocu et al., 2023).

☆ Editor: Aldeida Aleti.

* Corresponding author.
E-mail addresses: pasquale.ardimento@uniba.it (P. Ardimento), m.capuzzimati1@studenti.uniba.it (M. Capuzzimati), gabriella.casalino@uniba.it (G. Casalino), daniele.schicchi@itd.cnr.it (D. Schicchi), davide.taibi@itd.cnr.it (D. Taibi).
URL: https://www.uniba.it/it/docenti/ardimento-pasquale (P. Ardimento).

https://doi.org/10.1016/j.jss.2025.112569
Received 23 February 2025; Received in revised form 6 June 2025; Accepted 8 July 2025
Available online 21 July 2025

Large Language Models (LLMs) have gained significant attention in software engineering, with research exploring their application across various tasks. Their ability to generate high-quality code and support automated development workflows has made them a focal point in the field. For instance, in code generation, LLMs like Codex and CodeT5 have been used to synthesize code snippets and complete programming tasks (Hou et al., 2023; Wang et al., 2021). In code summarization and translation, LLMs have shown promising results in transforming code into natural language descriptions and vice versa (Zheng et al., 2025). For vulnerability detection and repair, LLMs have been applied to automatically identify and fix security issues in source code (Sobo et al., 2025). In software documentation, LLMs can generate API and method-level descriptions (Liu et al., 2024), while in project support tasks like question answering and code evaluation, they assist developers by retrieving relevant knowledge or reviewing code quality (Chen et al., 2024). Furthermore, LLMs have been effectively used for classification problems across various domains, including bug report triage and fixing prediction (Chang et al., 2024; Schicchi and Taibi, 2024; McAvinue and Dev, 2025), making them strong candidates for predictive modeling in bug tracking systems.

In this study, we investigate the potential of LLMs to classify bug reports into two categories, FAST/SLOW, based on their expected fixing time, aiming to assess their effectiveness in providing a reliable solution for real-world software maintenance. Unlike traditional machine learning approaches that require extensive preprocessing and large labeled datasets, we propose a method that directly leverages raw bug reports, relying on the LLM's inherent knowledge to estimate fixing time. This eliminates the need for manual feature engineering and costly training, offering a scalable and low-overhead alternative for bug triage. To the best of our knowledge, this is the first attempt to automate fixing time classification using LLMs.

Although predicting exact bug fixing times (e.g., in minutes or hours) might seem more informative, such data is rarely available in public BTSs, and even in private contexts, it is often influenced by external factors such as developer availability, organizational constraints, or task switching. For this reason, we formulate the task as a binary classification problem: determining whether a bug is likely to be resolved quickly (FAST) or slowly (SLOW). This framing provides actionable insights to support prioritization and task allocation while ensuring greater robustness, interpretability, and generalizability across projects.

1.1. Contribution

The main contributions of this study are as follows:

- A novel deep learning workflow for predicting bug-fixing time using LLMs in a zero-shot learning setting, leveraging a feature set that includes issue descriptions and bug reporter comments;
- Two prompt engineering patterns (persona and input semantics) designed to guide LLM and improve classification consistency;
- An empirical evaluation of the proposed LLM-based workflow in realistic settings, including scenarios with limited and imbalanced data.

These contributions are significant for both researchers and practitioners. For researchers, the proposed methodology offers a novel application of LLMs in a domain where prompt engineering and zero-shot capabilities remain underexplored, enabling new studies on adaptable classification pipelines. For practitioners, our approach facilitates the early estimation of bug fixing times without costly fine-tuning, improving resource allocation, task prioritization, and overall project planning in real-world software maintenance scenarios. By reducing computational overhead and leveraging raw textual data, this method supports scalable and transferable adoption across different projects and BTS environments.

The remainder of this article is structured as follows. Section 2 provides an overview of the bug life cycle and LLMs. Section 3 summarizes relevant related work. In Section 4, we present our methodology for predicting bug-fixing time based on newly submitted bug reports. The empirical setting used to evaluate the proposed approach is detailed in Section 5. Sections 6 and 7 present the experimental results and their discussion. We outline potential threats to validity in Section 8 and conclude with a discussion on future research directions in Section 9.

2. Background

In this section, we outline the key background concepts of our study. We begin with an overview of the bug life cycle, analyzing how bugs evolve from initial reporting to fixing in both general BTSs and Bugzilla. This dual perspective ensures the generalizability of our approach while anchoring it in a widely adopted and well-documented system.

Next, we introduce LLMs, describing their core principles, architecture, and capabilities. We explain why LLMs are particularly suitable for bug-fixing time classification, especially given their strength in processing unstructured textual data and generating predictions in complex, language-rich domains.

2.1. Bug life cycle

Bug Tracking Systems are tools to manage and track software issues, defect and bugs, throughout the software development process. They provide functionalities to record, update, prioritize, and monitor issues collaboratively among team members. A typical BTS stores structured metadata such as the bug's status, severity, priority, and fixing, as well as unstructured textual fields such as the description and developer comments.

A standard bug life cycle includes several phases: a bug is reported, assigned to one or more developers, and updated over time until it is resolved and verified. During this process, metadata and textual information evolve, documenting the reasoning and activities that lead to fixing. Understanding the transitions and time intervals between these states is crucial for modeling and predicting bug-fixing effort.

2.1.1. Bugzilla

In this study, we use Bugzilla (Foundation, 2025), a widely adopted and well-documented BTS. Bugzilla was selected for several reasons. First, it is extensively used in open-source communities, including projects such as Mozilla and Eclipse, and is frequently employed in empirical software engineering research (Malhotra et al., 2018; Repository, 2015). Its widespread adoption ensures access to real-world, heterogeneous bug reports for analysis.

Beyond open-source ecosystems, Bugzilla has been adopted across a variety of industry segments—including large organizations (e.g., Red Hat, NASA, Novell), academic environments, and smaller teams. Its open-source nature, high customizability, and low entry barrier make it suitable for both enterprise-scale and lightweight deployments (Ayari et al., 2004; Bugzilla, 2025). These characteristics contribute to the practical relevance and generalizability of our study.

In addition, Bugzilla provides open access to historical bug data and offers a robust REST API (available since version 5.0), which facilitates automated data extraction and integration with external tools. These features make it particularly suitable for large-scale analysis and machine learning applications. Its long-standing adoption, dating back to 1998, ensures the availability of longitudinal datasets that are invaluable for studying trends in bug fixing practices.

In our study, we examined the bug life cycles in both Bugzilla and general BTSs. The Bugzilla life cycle, as detailed in the official documentation for release 5.0.4, follows a specific sequence of stages that a bug report passes through, from its initial reporting to its eventual fixing. Similarly, a generalized bug life cycle applicable to most BTS platforms also exists, though with some variations. Both general BTS

and Bugzilla allow users to report, track, describe, comment on, and classify bug reports. A typical bug report includes several predefined fields, such as the relevant product, version, operating system, and self-reported incident severity. Additionally, it contains critical free-form text fields like the bug title (referred to as *summary* in Bugzilla) and description. Users and developers can also contribute by adding comments and submitting attachments, which often include patches, screenshots, test cases, or other large or binary files.

When a bug is first reported, it typically enters an *unconfirmed* or *pending* state. At this stage, a triager assesses the report to determine whether it corresponds to a valid bug and whether it is not a duplicate of an existing report. After this initial assessment, bug reports may progress through various stages before being resolved. Once resolved, a bug report is closed with a specific status, such as *duplicate*, *invalid*, *fixed*, *wontfix*, or *worksforme*, each of which indicates the reason for closure. For instance, both *worksforme* and *invalid* suggest that the issue described in the report could not be reproduced by quality assurance. In some cases, a resolved bug report may need to be reopened, which, in general BTS, initiates a new cycle starting with the *REOPENED* status. The absence of this status in the default Bugzilla configuration is a notable difference between the two life cycles. However, Bugzilla can be customized to include a *REOPENED* status by adding a new option to the *STATUS* field through the configuration interface. For our study, we focused on Bugzilla installations where the *REOPENED* status was available, ensuring alignment with the generalized BTS life cycle.

2.2. Large language models

LLMS are a major breakthrough in generative artificial intelligence, pushing the boundaries of machine learning. The term *large* refers not only to the vast number of neurons and layers in their architecture but also to their expansive capabilities. Unlike traditional AI models, LLMs often go beyond specific tasks, demonstrating versatility across multiple functions.

2.2.1. Capabilities and applications of large language models

LLMs have been proven particularly efficient in natural language tasks such as understanding, generating, and manipulating human language. At the same time, they have been exploited in several other tasks, such as AI vision, task automation, and data processing (Chang et al., 2023). Most LLMs are founded on the transformer architecture that utilizes self-attention mechanisms to process data efficiently (Vaswani et al., 2017). The transformer's architecture has enabled the model's training parallelization, overcoming the difficulties encountered with previous models such as Recurrent Neural Networks (Medsker et al., 2001). Parallelizing the training phase has enabled the creation of bigger transformer-based models capable of gathering more knowledge and learning more skills (Bubeck et al., 2023).

LLMs have shown great potential in various prediction and classification tasks in various domains. LLMs learn to solve a prediction task by leveraging large amounts of data, capturing complex patterns, and integrating contextual information. For example, in data science, LLMs have been customized to work with tabular data, enhancing tasks such as classification (Hegselmann et al., 2023). In autonomous driving, LLMs improve motion prediction by considering comprehensive traffic context (Ren et al., 2024), while in recommender systems, they enhance rating predictions through a combination of collaborative filtering and semantic understanding (Luo et al., 2024). These advancements demonstrate the versatility and effectiveness of LLMs in improving prediction accuracy and reliability across various applications.

Today, modern LLMs have demonstrated advanced capabilities in extracting language features encompassing both structural components and contextual meaning. These advancements enable them to excel in tasks centered around natural language, leveraging their capabilities in various language-based applications. Consequently, LLMs show great potential in predicting bug-fixing times. Their contextual understanding allows them to grasp a bug report's intricate relationships and nuances, providing a comprehensive view of the issue. They excel in pattern recognition, identifying correlations and trends from vast amounts of historical bug data to predict fix times based on similar past cases. Using natural language processing techniques, LLMs can extract critical entities such as specific modules, error codes, and user roles while also assessing the urgency or severity implied by the language used in the report. They understand grammatical structures, enabling them to identify essential actions and dependencies within the text. Their ability to interpret technical descriptions and comprehend code snippets, error messages, stack traces, and configuration details are crucial and are central to accurately gauging the complexity and effort required for bug fixing.

Furthermore, the vast amount of data used to train large language models allows them to synthesize knowledge from various sources, including extensive collections of coding practices, software development documentation, and historical bug-fixing patterns. This rich knowledge base enables them to identify recurring issues, recognize contextual relationships within code, and improve the predictive accuracy of bug detection and fixing.

The capabilities of LLMs and the nature of their training process encourage the development of this study, which aims to test their effectiveness in solving the bug-fixing prediction time problem.

2.2.2. Prompt engineering

Probabilistic methods govern the output generation process in LLMs. In this process, each element of the sequence is produced incrementally by selecting the most likely next element based on both the given input and the previously generated sequence. Additionally, contemporary LLMs utilize a parameter known as *temperature*, which enables the non-deterministic nature of the output generation process. Temperature parameter adjusts the level of randomness in the selection process, allowing for the production of more diverse outputs (Peeperkorn et al., 2024)

In response to the challenges posed by the uncertainty of the output process, the scientific community has initiated research into strategies for controlling the llms' sequence generation process. As a result, prompt engineering has gained significantly greater importance among LLMs' users. Prompt engineering (White et al., 2023a) is a methodological approach to formulating input prompts, a set of targeted instructions given to a Large Language Model. Prompts effectively program the LLM, customizing and enhancing its capabilities to meet specific user requirements (Liu et al., 2023). Through meticulous crafting and standardization of prompts, it is possible to mitigate randomness and improve the likelihood of obtaining specific types of responses. This technique brings to the efficient use of LLMs and finds application in various fields (White et al., 2023a). Precise prompts can significantly influence the accuracy and relevance of the model output, thus affecting the overall effectiveness of the interaction.

White et al. (2023a) introduce essential prompt patterns that establish the foundation for effective prompt engineering. These patterns enable systematic prompt engineering to attain various outputs and meet interaction goals when working with conversational LLMs. The use of prompt patterns can help tailor the types, formats, structure, and other features of the output generated by the LLM. They also enable control over how the LLM understands and processes the input to produce output. Moreover, prompt patterns can enhance the quality of input and output, manage user interaction with the LLM, and control the contextual information in which the LLM works.

3. Related work

While the Background Section introduced the fundamental concepts and technologies used in this study, this section focuses specifically on prior research addressing the prediction of bug-fixing time. We review

the evolution of this task from early machine learning approaches to probabilistic and sequential models, and finally to recent deep learning techniques, including transformer-based architectures and LLMs.

Bug-fixing time prediction has received considerable attention in software engineering, as it plays a critical role in project planning and resource allocation. Existing approaches can be broadly categorized into three groups: traditional machine learning techniques, probabilistic and time-aware models, and deep learning-based methods. The latter includes recent efforts that leverage pre-trained LLMs, highlighting a shift toward more flexible and semantically rich representations.

3.1. Early machine learning approaches

Initial studies on bug-fixing time prediction relied on classical machine learning models. Panjer (2007) explored various classification algorithms, including Naïve Bayes, logistic regression, and decision trees, to analyze bug reports from the Eclipse project. Despite achieving an accuracy of 34.9%, the study demonstrated the feasibility of statistical models for bug-fixing time estimation. Hooimeijer and Weimer (2007) extended this work by applying linear regression to a dataset of 27,000 bug reports from the Firefox project. Their findings indicated that textual features, such as the number of comments and attachments, correlated with longer bug-fixing times, suggesting that integrating text-based features could improve predictive performance.

Anbalagan and Vouk (2009) analyzed 72,482 bug reports from Ubuntu, identifying a linear relationship between the number of developers involved and bug fixing time. Bhattacharya and Neamtiu (2011) further refined these models by incorporating additional variables, including bug severity, developer involvement, and dependencies. However, these regression-based approaches struggled to capture the complexity of bug fixing processes, leading to limited predictive accuracy.

3.2. Probabilistic and sequential models

As machine learning techniques matured, researchers explored probabilistic models to enhance prediction reliability. Habayeb et al. (2018) introduced a Hidden Markov Model (HMM) to analyze the sequential nature of bug-fixing processes. Their method, tested on Firefox projects, outperformed traditional classifiers by leveraging the temporal dependencies of developer activities.

Zhang et al. (2013) proposed a Markov-based model to predict the number of monthly bug fixes in commercial software projects. Their results demonstrated the effectiveness of state-transition models in forecasting fixing trends. Akbarinasaji et al. (2018) extended this work by applying similar techniques to Bugzilla datasets from Firefox, reinforcing the robustness of Markov-based predictions.

3.3. Deep learning and large language models

With the advent of deep learning, transformer-based architectures revolutionized bug-fixing time prediction. Ardimento (2022) compared DistilBERT, a lightweight transformer, with Google's full BERT model. Their study demonstrated that while DistilBERT provided faster inference times, the full BERT model achieved superior accuracy due to its deeper architecture and richer contextual embeddings.

Further advancements were made by Ardimento and Mele (2020), who fine-tuned BERT for bug-fixing time prediction, leveraging its ability to capture semantic relationships within bug reports. Their findings showed significant improvements over traditional machine learning models, highlighting the capability of transformers to process complex textual data.

Beyond bug tracking, LLMs such as GPT-3 and T5 have been successfully employed in predictive modeling across various domains, including project management, healthcare, and software development (Gunasekaran et al., 2023; Tsabari et al., 2023). These models leverage pre-trained contextual embeddings to understand the nuances of textual data, making them particularly effective in estimating bug fixing times. Recent work by Colavito et al. (2025) provides a comprehensive evaluation of 22 large language models for automated issue report classification. Their study compares encoder-based and decoder-based LLMs in zero-shot and few-shot settings, highlighting the trade-offs between accuracy and computational efficiency. The findings of their work strength the applicability of LLMs for extracting semantically meaningful features from issue reports, a foundational capability for bug-fixing time prediction tasks. Gunasekaran et al. (2023) introduced a transformer-based approach for temporal classification of text, demonstrating its potential in estimating time-based attributes from documents. Similarly, Tsabari et al. (2023) applied transformer-based architectures to predict student bug-fixing times in educational programming tasks, reinforcing the adaptability of LLMs in diverse predictive applications.

Despite these advancements, existing studies have not fully explored the potential of LLMs in predicting bug-fixing time. While prior work has demonstrated the effectiveness of transformers in handling textual data, a gap remains in leveraging prompt engineering techniques and fine-tuning strategies to optimize LLM performance for bug-tracking systems. This study aims to address this gap by investigating the effectiveness of LLMs in predicting bug fixing times, with a specific focus on prompt engineering and input semantic optimization.

4. Proposed approach

Despite encouraging results, existing approaches to bug-fixing time prediction present several limitations. Traditional machine learning methods often rely on manual feature engineering or limited metadata, making it difficult to capture the full semantic richness of bug reports. Transformer-based models, though more expressive, typically require supervised fine-tuning on large labeled datasets, which is often infeasible in real-world bug tracking systems where labeled data is scarce or imbalanced. Moreover, although LLMs have demonstrated effectiveness in several domains, including software engineering, they have not yet been applied to predicting bug fixing times.

To address these challenges, we propose a novel method that directly leverages LLMs in a zero-shot setting. This strategy eliminates the need for task-specific training, making the approach more scalable and adaptable. We employ prompt engineering to guide the model's behavior through structured instructions. In particular, two patterns, persona and input semantics, are employed to enhance the interpretability and controllability of the output. Our method also integrates the bug report description with the bug reporter's comments, which are sometimes neglected despite their informative potential. Furthermore, we explicitly evaluate our approach under data-scarce and imbalanced conditions, which frequently characterize real-world bug tracking scenarios.

Fig. 1 illustrates the proposed workflow for classifying bug reports as either *FAST* or *SLOW*, based on their expected fixing time. The method is designed to be independent of any specific project or BTS, thereby ensuring generalizability across diverse software ecosystems. The workflow comprises three main phases: (i) *Data Collection*, (ii) *Corpus Creation*, and (iii) *Bug Fixing Time Classification*.

In the first phase, raw bug reports are collected from various sources and undergo a series of preprocessing steps designed to extract relevant features. These features serve a dual purpose: they are used to construct a textual corpus suitable for prompt-based inference and to assign a fixing time to each bug report. The fixing times are then discretized into the two classes *FAST* and *SLOW*, using a percentile-based threshold. In the final phase, LLMs are employed, utilizing prompt engineering techniques that guide the model's reasoning process, enabling the accurate classification of bug reports based on their predicted fixing time. Although the approach is general, in the following sections, we describe the first two phases with reference to Bugzilla, a well-documented and widely adopted BTS. This allows us to provide a concrete illustration of the process while preserving its generalizability to other platforms.

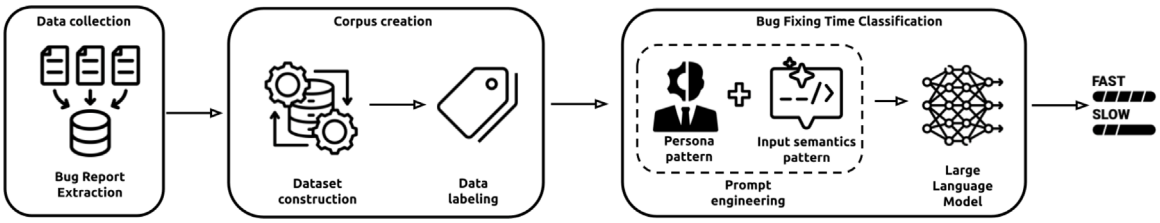

Fig. 1. Overview of the LLM-based classification workflow for predicting bug fixing time.

4.1. Data collection

The data collection process involved the extraction of bug reports from mature and actively maintained open-source software projects. Selection criteria for these projects included the availability of a long development history, a high volume of reported issues, and well-documented bug report metadata. These factors ensure a sufficient and diverse data foundation for studying bug-fixing time prediction.

To align with the objective of predicting bug-fixing time, only those bug reports that were confirmed and resolved were considered. Specifically, reports were selected if they were accepted as valid after submission and marked as fixed by developers. This approach ensured that each entry reflected a complete fixing cycle, which is essential for constructing a reliable historical record to support time-to-fixing prediction. Reports were included only if they had the status `RESOLVED` and the resolution field set to `FIXED`, or equivalent labels depending on the bug tracking system in use.

For each selected report, the initial bug description and the bug reporter comments were collected, as they provide valuable insights into the nature of the issue and the steps taken to resolve it.

This combination of structured and textual information constituted the foundation for building a dataset that captures both the contextual and technical aspects of software maintenance activities. The resulting dataset was subsequently used to construct model inputs for predicting bug-fixing time using LLMs.

4.2. Corpus creation

This phase focuses on preparing bug reports for processing by LLMs through a preprocessing pipeline consisting of two main stages: dataset construction and data labeling. The former transforms raw bug tracking data into a structured format, while the latter assigns class labels based on fixing time thresholds to support binary classification tasks.

4.2.1. Dataset construction

The objective of this phase is to transform raw bug report data into a structured format suitable for classification tasks using LLMs. The process begins with JSON-formatted records exported from the BTS. To ensure data quality, reports lacking critical information, such as a short description or bug reporter comments, are excluded, retaining only complete and informative instances.

From each valid report, both the initial bug description and the bug reporter's comment are extracted.

As the fixing time is generally not provided explicitly in public BTS platforms, it must be derived. This is calculated as the number of days between the assignment date and the date the bug's status is updated to indicate that it had been fixed and validated.

All extracted information is consolidated into a unified data structure (e.g., a `data.frame`) and saved in formats appropriate for downstream processing, such as RData and JSON. This structured representation serves as the foundation for both the data labeling phase and the creation of input prompts for LLM-based classification.

4.2.2. Data labeling

Data labeling was a critical step in preparing the dataset for the classification task. Rather than predicting the exact fixing duration, each bug report was assigned to one of two categories, FAST or SLOW, based on its fixing time, as computed in the previous subphase. Given the complexity of the problem and the variability across projects, a binary classification framework was adopted. Class labels were defined using a percentile-based threshold, enabling a consistent and adaptable labeling strategy. By relying on relative rather than absolute fixing times, this approach ensures that the classification remains meaningful and comparable across heterogeneous datasets.

We adopt the 75th percentile as the threshold for distinguishing between *FAST* and *SLOW* bug reports. Reports resolved within a number of days less than or equal to this threshold (denoted as n) are labeled as *FAST*, whereas those requiring more than n days are labeled as *SLOW*. By definition, this implies that approximately 25% of the reports—those with the longest fixing times—are categorized as *SLOW*, reflecting their relative rarity and increased complexity.

This thresholding strategy is consistent with several prior studies that employ empirical distribution-based criteria to define classification labels (Zhang et al., 2013; Bhattacharya and Neamtiu, 2011; Ardimento and Boffoli, 2022; Ardimento et al., 2020; Gomes et al., 2021; Kholief et al., 2013; Rodrigues and Costa, 2024; Abdelmoez et al., 2012; Yücel and Tosun, 2022; Gong et al., 2013; Datta and Lade, 2015; Mihaylov, 2019; Bettenburg et al., 2012; Wang et al., 2019; Ardimento, 2023; Kawamura et al., 2023; Tsabari et al., 2023; Ardimento et al., 2016). In particular, Gomes et al. (2021) justify the use of the third quartile (75th percentile) as a threshold to separate short-lived from long-lived bugs, noting that it effectively captures the top 25% most time-consuming issues while reducing the influence of extreme outliers typically found in bug fixing data. Although the specific threshold value may vary depending on the organizational context, such as team efficiency, workflow complexity, or Service Level Agreements (SLAs), the percentile-based method ensures reproducibility and comparability by aligning the threshold with the characteristics of the dataset in use. This makes the approach both robust and flexible, as it can be adapted to different domains and data availability without changing the underlying modeling pipeline.

Moreover, in scenarios where fixing time thresholds are already defined by contractual obligations or internal policies (e.g., SLAs), our approach remains fully compatible. In such cases, a fixed threshold can be used to label the data, aligning the classification with operational requirements. The predictive pipeline does not require any architectural modification; only the labeling strategy changes, ensuring that the method remains effective, interpretable, and adaptable to both data-driven and policy-driven contexts.

4.3. Bug fixing time classification

The proposed approach addresses the task of classifying bug-fixing time using LLMs. In this work, we investigate the capability of LLMs to distinguish between bugs that require short or long fixing times, based solely on instructions provided through prompts. Our aim is to leverage the knowledge implicitly encoded within the models; therefore, no model fine-tuning or additional training is performed.

Specifically, we adopt a zero-shot learning setting, in which the LLM is prompted with a bug report and asked to predict its fixing time in terms of the two predefined classes: *FAST* and *SLOW*. To improve the classification process, we design detailed instructions using two prompt design patterns: the *persona pattern* and the *input semantic patterning*. These structured prompting schemes are intended to guide the model towards producing contextually appropriate and consistent outputs.

The combination of the persona pattern and the input semantics pattern is motivated by recent advances in prompt engineering for NLP tasks. In our approach, NLP plays a central role, as a pre-trained LLM is used to interpret and classify bug reports expressed in natural language. The persona pattern assigns the LLM the role of a domain expert, simulating the reasoning process of a software developer estimating fixing time. This role-based framing enhances the model's contextual understanding and response consistency. The input semantics pattern, on the other hand, organizes the bug report into structured fields, enabling the model to better identify and process relevant information. Prior work has shown that combining these patterns improves classification accuracy, interpretability, and generalization in tasks requiring domain-specific reasoning (White et al., 2023a,b; Chen et al., 2025).

4.3.1. Persona pattern

The Persona pattern is a strategic framework designed to enhance interactions with LLMs through role-playing. This method enables the customization of outputs from the language model by restricting the model's responses to a predetermined perspective. The persona pattern is developed to tailor outputs according to user expectations and requirements. Users may not always be cognizant of the specific output they need or the features the model should utilize to respond accurately to the query. However, they often have an idea of the type of expert who can address similar real-world problems. By implementing the Persona pattern, users can naturally express their needs, enabling them to articulate the type of assistance they require without needing to delve into the technical specifics of the desired output. In this work, the prompt instructs the LLM to act as a software engineering expert. This role-based framing encourages the model to adopt the reasoning style and decision-making process typical of an experienced developer. By simulating domain expertise, the LLM is more likely to focus on relevant aspects of the bug report, such as historical patterns or terminology commonly used by practitioners in the field. This enhances the contextual alignment of the output, resulting in more consistent and accurate predictions of bug fixing times.

4.3.2. Input semantics pattern

The input semantics pattern plays an indispensable role in prompt engineering. It enables the development of more precise and contextually appropriate prompts, significantly enhancing the quality of responses generated by language models. This aspect of prompt engineering highlights its significance in the evolving landscape of artificial intelligence, where a nuanced understanding and handling of input data can significantly impact the effectiveness and utility of language processing technologies.

This method involves categorizing inputs based on their inherent characteristics and relevance to the intended application context.

In this work, we provide the LLM with explicit instructions concerning the input structure, contextual cues to support interpretation, the classification criterion, and the expected output format. These elements are intended to guide the model's reasoning process and enhance the reliability of its predictions. Further details on the specific prompt design adopted are presented in the following sections.

In our experimental framework, each bug report is labeled as *FAST* or *SLOW* based on its fixing time relative to a project-specific threshold, defined as the 75th percentile of the fixing time distribution. Given a bug report included in the prompt, without its associated class, the LLM is expected to return the corresponding label, indicating whether the fixing time is *FAST* or *SLOW*.

Adopting input semantics in managing interactions with LLMs facilitates the optimization of response strategies, ensuring that the model's outputs are both timely and appropriate relative to the complexity of the inputs.

5. Experiment description

The experimentation is carried out using the approach described in Section 4 on a dataset composed of four open source projects. The following section reports the research questions, the description of the dataset, and the experimental settings.

5.1. Research questions

Based on the motivation outlined in Section 1, the research gaps identified in Section 3, and the proposed approach described in Section 4, we define the following research questions to investigate the predictive capabilities of LLMs in the context of bug fixing time classification. It is important to emphasize that the behavior and performance of LLMs are highly dependent on how they are employed. Therefore, the following research questions are designed to assess the effectiveness of LLMs within the specific design choices adopted in this study, namely, a zero-shot classification setting and the use of a structured prompt combining a persona pattern with input semantic patterning. These questions address critical challenges that have received limited attention in previous work, including the use of LLMs for classification tasks, the impact of data scarcity, and the presence of class imbalance. To answer these questions, we design and conduct a set of empirical experiments described in the following sections.

RQ1: To what extent can LLMs accurately classify bug reports as FAST or SLOW within the zero-shot workflow introduced in this study?

Rationale: Accurately predicting bug-fixing time is crucial for effective project management, as it enables better task prioritization and resource allocation. Large Language Models (LLMs), with their advanced natural language understanding capabilities, offer a promising alternative to traditional methods by analyzing complex textual information from bug reports and developer comments. However, the performance of LLMs is strongly influenced by how they are employed. In this study, we evaluate their effectiveness in a controlled setup within a zero-shot classification workflow, where LLMs are guided through specific prompt patterns to standardize reasoning and enhance output consistency. Classification is performed without task-specific training, relying solely on the intrinsic knowledge embedded in the pre-trained LLM.

RQ2: To what extent can LLMs accurately classify bug reports as FAST or SLOW within the zero-shot workflow introduced in this study, when only a limited number of bug reports are available?

Rationale: In many real-world scenarios, such as early-stage development, proprietary systems, or niche software domains, the availability of historical bug reports is often limited. This research question examines whether LLMs, when guided by the structured zero-shot workflow proposed in this study, can still achieve reliable classification performance under data-scarce conditions. Given their extensive pretraining on large-scale textual corpora, LLMs may be able to generalize effectively, making them a viable solution for contexts where traditional learning-based methods struggle due to data scarcity.

RQ3: To what extent does class imbalance impact the ability of LLMs to accurately classify bug reports as FAST or SLOW within the proposed zero-shot classification workflow?

Rationale: Real-world datasets are often imbalanced, with a majority of bugs resolved quickly and a minority taking significantly longer. Such skewed distributions may degrade model performance, particularly in classification tasks where the underrepresented class is of high practical relevance. This question explores whether LLMs, when used within the proposed zero-shot classification workflow, are robust

Table 1
Descriptive statistics of the collected datasets.

Dataset	Total bugs RESOLVED & VERIFIED	Discarded bugs	Full dataset
W3C	409	1	408
OpenXchange	520	0	520
Novell	32,143	4155	27,988
Mozilla	120,490	498	119,992

to class imbalance and can still generate reliable predictions across both categories. Understanding this behavior is crucial for practical deployment, as it helps assess the models' ability to generalize in realistic bug-tracking environments.

5.2. Selected projects

The experimental evaluation was conducted using bug reports extracted from Bugzilla, a widely adopted BTS, across four distinct software projects: Mozilla, Novell, OpenXchange, and W3C. These projects were selected to cover a diverse range of software development contexts, ensuring that the study's findings generalize across different domains.

- **Mozilla:** Includes bug reports related to the Firefox browser and its associated technologies. As one of the most active open-source projects, Mozilla provides a large volume of diverse and well-documented bug reports.
- **Novell:** A dataset originating from a former enterprise software company specializing in networking and security solutions. It offers insights into bug fixing processes in enterprise software.
- **OpenXchange:** Represents the largest independent email provider globally, contributing bug reports from its email and collaboration software platforms. This dataset introduces variability from cloud-based and enterprise communication systems.
- **W3C:** Contains bug reports related to web standards development and software tools used to maintain compliance with these standards.

Table 1 summarizes the main characteristics of the collected datasets. For each project, it reports the total number of bug reports marked as RESOLVED and VERIFIED, the number of discarded reports excluded during pre-processing and the final size of the dataset used in the analysis. Some bug reports were discarded primarily due to the absence of meaningful textual content, specifically, they lacked both a description and comments, rendering them unsuitable for the bug fixing time prediction task. Among the datasets, Mozilla offers the largest collection, with nearly 120,000 usable reports after filtering, while W3C provides the smallest, with just over 400 entries. Despite variations in size, all datasets share a consistent structure, allowing for the application of a unified preprocessing and analysis pipeline.

By incorporating datasets from different software ecosystems, this study aims to evaluate the robustness and adaptability of the proposed predictive models across diverse bug tracking environments.

5.3. Dataset collection

For each Bugzilla installation of selected projects, we retrieved structured data from all bug reports whose status was RESOLVED and VERIFIED, as these represent confirmed and resolved issues. The extraction process focused on collecting both textual and categorical fields relevant to the subsequent classification task. The textual fields extracted from each selected bug report are:

- the name of the selected project.
- Fixing days: number of days taken to fix the bug. This value is calculated as the number of days between the date the bug was assigned to a developer and the last date on which its status was set to FIXED in the Bugzilla metadata. This value is used to assign each bug to the FAST or SLOW class. As soon as a bug is marked as RESOLVED, the remaining time expected to fix the bug is considered zero. Although the specific fields used to determine this duration may vary between different BTSs, the underlying logic for estimating the fixing time remains consistent.
- Comment posted by the bug reporter. This comment, inserted by the user who created the report, usually consists of a long description of the bug and its characteristics.

5.4. Generation of experimental datasets

To address the research questions, we generated multiple datasets per project, varying in class balance and sampling strategy across different fixing-time thresholds. The original dataset is available at Ardimento et al. (2025).

As a general reference, we adopted the 75th percentile of the fixing time distribution as the threshold to distinguish between the *FAST* and *SLOW* classes for each project. As detailed in Section 4.2.2, this choice aligns with prior studies and was systematically applied to the full dataset of each project. Table 1 reports the total number of bug reports retained in each dataset after filtering, under the *Full Dataset* condition.

Moreover, for each project, we created *sampled datasets* by randomly extracting 300 samples and repeating this process 50 times. This approach enables experimentation on smaller subsets while preserving statistical significance. We primarily used the 75th percentile as the classification threshold to ensure comparability with the full datasets.

To further assess the impact of different threshold choices on the classification performance of LLMs, and to evaluate the generalizability and robustness of our results, we also considered the 50th and 90th percentiles when generating the sampled datasets. These two additional thresholds represent extreme cases of class imbalance: the 50th percentile typically yields more balanced class distributions and a low absolute value (in days) for *FAST* bugs, capturing very short fixing times, as shown in Table 2. However, it also increases variability within the *SLOW* class, which may include both moderately and highly time-consuming bugs. In contrast, the 90th percentile results in a highly imbalanced distribution, focusing on a small set of rare, long-fixing cases that define the *SLOW* class, thus simulating a more challenging classification scenario.

Moreover, for both the full and sampled datasets, we considered both balanced and imbalanced class distributions.

To summarize, the datasets generated for our experiments are briefly described below. The same dataset names will be consistently used in the subsequent section to refer to the corresponding experimental configurations.

- *Full Data, Balanced Classes*: Only for the 75th percentile, we labeled the entire dataset using the corresponding threshold and applied random undersampling to balance the classes. The larger class was reduced to match the size of the smaller one;
- *Sampled Data, Balanced Classes*: For each selected percentile (50th, 75th, 90th), we randomly selected 50 subsets of 300 bug reports from the original dataset. Each sample was labeled using the fixed threshold and balanced via random undersampling;
- *Full Data, Unbalanced Classes*: Only for the 75th percentile, the entire dataset was labeled using the corresponding threshold, and the original class imbalance resulting from this threshold was preserved
- *Sampled Data, Unbalanced Classes*: For each selected percentile (50th, 75th, 90th), we randomly selected 50 subsets of 300 bug reports. A fixed threshold, calculated on the full dataset, was used to label each bug report as FAST or SLOW. In this configuration,

Table 2
Class distributions in sampled datasets labeled using a fixed global threshold. The unbalanced values reflect observed distributions without any post-labeling class balancing.

Collection	Balance	Percentile	Threshold (Res. days)	# of fast	# of slow
Mozilla	Unbal.	50th	12	181	119
	Bal.	50th	12	119	119
	Unbal.	75th	52	214	86
	Bal.	75th	52	86	86
	Unbal.	90th	176	263	37
	Bal.	90th	176	37	37
Novell	Unbal.	50th	6	163	137
	Bal.	50th	6	137	137
	Unbal.	75th	33	230	70
	Bal.	75th	33	70	70
	Unbal.	90th	130	272	28
	Bal.	90th	130	28	28
OpenXchange	Unbal.	50th	8	150	150
	Bal.	50th	8	150	150
	Unbal.	75th	79	226	74
	Bal.	75th	79	74	74
	Unbal.	90th	830	274	26
	Bal.	90th	830	26	26
W3C	Unbal.	50th	23	152	148
	Bal.	50th	23	148	148
	Unbal.	75th	38	219	81
	Bal.	75th	38	81	81
	Unbal.	90th	95	266	34
	Bal.	90th	95	34	34

no balancing was applied after labeling. As a result, even for the 50th percentile, which globally represents the median, the class distribution within each subset is not necessarily balanced. We chose to use a fixed global threshold and not to balance the subsets to simulate realistic conditions, where the imbalance between fast and slow bug fixes occurs naturally. This allowed us to evaluate the robustness of the models in a scenario closer to real-world use.

Table 2 reports the number of *FAST* and *SLOW* bug reports for each project, based on sampled datasets generated using three percentile thresholds (50th, 75th, and 90th). For each threshold, a representative sample of 300 bug reports is presented, considering both unbalanced and balanced class distributions. The specific threshold value (in days) used to label the bug reports is also indicated for each configuration.

It is worth noting that, for a given percentile threshold, the fixing times vary significantly across projects. For instance, using the 50th percentile to identify *FAST* bug reports, the minimum number of days observed is 6 for Novell and 23 for W3C. These differences become even more pronounced at higher percentiles and do not increase linearly within individual projects. At the 75th percentile, the threshold values are 33 days for Novell, 38 for W3C, 52 for Mozilla, and 79 for OpenXchange. To explore extreme cases, we also analyzed the 90th percentile. Here, the thresholds are 95 days for W3C, 130 for Novell, 176 for Mozilla, and as high as 830 for OpenXchange. Such variability highlights the strong dependence of bug fixing time on the specific project. It also indicates that a fixed threshold across projects would be inadequate. Moreover, the distribution of fixing times differs markedly between projects with more homogeneous processes, such as W3C and Novell, and those like OpenXchange, which exhibit much higher variability. These findings confirm the appropriateness of using a percentile-based thresholding strategy, as it enables meaningful comparisons across heterogeneous datasets by normalizing for project-specific characteristics.

5.5. LLMs

Our objective was to evaluate how predictive performance varies depending on the choice of the large language model. To this end, we selected three state-of-the-art open-weight LLMs: Llama 2:70B, Llama 3:8B, and Mistral 7B (ver. 0.2). Llama 2:70B, developed by Meta, is a large-scale model with 70 billion parameters known for its strong text generation and reasoning capabilities (Touvron et al., 2023). Llama 3:8B is the updated version of LLama2, a more compact yet optimized variant with 8 billion parameters, designed to strike a balance between efficiency and performance (Dubey et al., 2024). Mistral, on the other hand, is a smaller but highly efficient model. Leveraging 8 billion parameters and advanced architectural optimizations, it achieves faster and more adaptable language processing (Jiang et al., 2023).

At the time of our experiment, these models were among the most prominent open-weight LLMs freely available, making them suitable choices for our study. We deliberately excluded proprietary pay-per-use models, prioritizing transparency, reproducibility, and accessibility. Additionally, given our computational constraints, we set an upper limit of 70 billion parameters, selecting models within this range to ensure feasible experimentation.

Recognizing the strong performance of smaller models in recent studies, we also evaluated an 8-billion-parameter model to assess whether a more compact architecture could deliver competitive results for predictive bug fixing. This allowed us to explore potential trade-offs between model size, efficiency, and predictive accuracy.

5.6. Prompt

In this work, we employed zero-shot learning to evaluate the ability of LLMs to classify bug fixing times without relying on fine-tuning or task-specific examples. This setup allowed us to directly assess whether the knowledge encoded in the models is sufficient for performing this classification task.

To make the task comprehensible and operational for the LLMs, we designed the prompt by combining two key strategies: the persona pattern and the input semantic pattern, as illustrated in Fig. 2 and detailed in Section 4.3. This combination was instrumental in clearly specifying the task while also simulating realistic developer behavior and input structure.

The first section of the prompt adopts the persona pattern, instructing the model to assume the role of an expert software developer. This role-based framing promotes domain-consistent reasoning, aligning the model's behavior with the type of cognitive processing expected in software engineering tasks, such as bug triage and estimating fixing times.

The second section details the task instructions provided to the large language model. It introduces a semantic input pattern that assigns labels—'A' for *FAST* and 'B' for *SLOW*—to facilitate binary classification. This labeling schema is designed to scaffold the model's understanding by embedding explicit symbolic cues within the prompt. By reducing semantic ambiguity and introducing a structured form of supervision, this setup helps guide the model toward producing more consistent and interpretable outputs. In addition, a decision criterion is incorporated based on a fixed temporal threshold, denoted by the placeholder `{days}`. This threshold is expressed in days and acts as a parameter for defining the classification boundary. The use of a placeholder makes the prompt parametrizable, enabling flexible reuse across multiple datasets and configurations while maintaining a uniform and transparent decision protocol.

In the third section, the bug report is structured into standardized fields—namely, the project name (`dataset`) and a description (`comments`). This structured input improves interpretability, guiding the LLM to process each component effectively. It also mirrors how developers typically read and interpret bug reports, reinforcing the realism of the task formulation.

The fourth section specifies a strict output format, requiring the model to return only the classification label ('A' or 'B') without additional commentary. This constraint ensures consistency across model outputs and simplifies post-processing during evaluation.

```
1  "Start thinking as if you were an expert software developer"

2  "### Instruction:"
   "A = 'FAST' and B = 'SLOW' you can prompt in output only A or B, predicting the
   resolution time of a bug by saying those that can be resolved in LESS than {days} DAYS
   in the 'A' label and those that can be resolved in MORE than {days} DAYS in the 'B'
   label."

3  "### Input:"
   BUG: comes from the {dataset} BTS (Bug Tracking System) Dataset, about this bug we
   know that: {comments}"

4  "### Output format: Your reply must be "A" or "B", do not write any other comments."
```

Fig. 2. Template of the LLM prompt used for bug fixing time classification.

While this study does not aim to systematically evaluate the effectiveness of prompt engineering techniques, the use of prompt design patterns, specifically the Persona Pattern and Input Semantic Pattern, was intentional and guided by insights from existing literature. These patterns were incorporated to improve the alignment of large language model outputs with the task of bug-fixing prediction. The Persona Pattern helps constrain the model's behavior by simulating a specific role or expertise (e.g., that of a professional software engineer), which has been shown to enhance output relevance and consistency. The Input Semantic Pattern, on the other hand, structures the input in a way that emphasizes critical contextual information, thereby helping the model better understand and process the task.

Although we do not present exploratory or comparative results on the effectiveness of these patterns in isolation, their inclusion is motivated by prior empirical findings that demonstrate how carefully crafted prompts can significantly influence LLM performance across a range of tasks (White et al., 2023a; Salmi et al., 2025). By adopting these patterns, we aim to reduce variability in model behavior and improve the quality and interpretability of the generated predictions. We consider a more in-depth investigation of prompt design and its isolated effects on performance as a promising avenue for future research.

5.7. Classification performance metrics

For each bug report, given the ground truth class labels (FAST or SLOW), evaluated as described in Section 4.2.2, and the predicted label produced by the LLM, we are able to quantitatively assess model performance using standard classification metrics and interpret the results in the context of bug fixing. Our analysis focuses in particular on the correct identification of FAST bugs, which is especially valuable in real-world settings where early identification of quickly resolvable issues can significantly streamline the software maintenance process.

Accuracy reflects the overall proportion of correct predictions made by the model, including both fast and slow bug reports. However, it can be misleading in imbalanced scenarios, where one class dominates.

Precision, in our context, refers to the percentage of bugs predicted to be fast that are actually resolved quickly. High precision indicates that when the model identifies a bug as fast, it is generally a reliable prediction, helping avoid the risk of prioritizing complex bugs under the mistaken assumption that they are easy to resolve.

Recall measures the model's ability to identify all bugs that truly have a short fixing time. A high recall means that most fast-resolving bugs are correctly detected, ensuring these issues are promptly addressed and not overlooked.

The F1-score provides a balanced measure by combining precision and recall. It is particularly useful when dealing with class imbalance, as it captures both the model's ability to avoid false alarms and its capacity to identify all relevant fast-resolving bugs.

In addition to predictive performance, we also recorded the execution time of each model, measured in seconds. This enables us to evaluate the trade-off between accuracy and computational efficiency, a crucial consideration when integrating LLMs into practical software engineering pipelines.

5.8. Empirical settings

We designed three experimental settings to evaluate the ability of LLMs to predict bug fixing times under different data conditions. The experiments aim to assess both predictive performance and computational efficiency across varying dataset sizes and class distributions.

In the first experiment, *Full Data – Balanced Classes*, we evaluated the overall effectiveness of open-source LLMs in predicting fixing times using the complete dataset for each project. To ensure an unbiased evaluation, the two classes, *FAST* and *SLOW*, were balanced via random undersampling of the majority class. This setup provides a comprehensive benchmark for model performance across full-scale datasets.

The second experiment, *Sampled Data – Balanced Classes*, investigates the feasibility of using smaller, more practical datasets. For each project, we generated sampled datasets containing 300 bug reports. This sample size reflects real-world scenarios where predictions must be produced efficiently, making large-scale inference impractical. The class distribution was balanced to present the models with a simplified classification task. To account for variability due to sampling, each experiment was repeated 50 times with different random seeds, and the results were averaged. Furthermore, to explore the impact of class definition on model performance, we varied the classification threshold using the 50th, 75th, and 90th percentiles of the fixing time distribution. This enabled us to simulate varying degrees of class imbalance and examine the adaptability of LLMs under different labeling strategies.

Recognizing that perfectly balanced datasets are uncommon in real-world applications, the third experiment, *Unbalanced Classes*, introduces natural class imbalance to better reflect the dynamics of actual bug fixing processes. The goal was to assess whether the performance of LLMs degrades under imbalanced conditions or remains robust. This experiment was conducted on both the full datasets and the sampled datasets. For the sampled case, the same procedure used in the second experiment was adopted: 50 repetitions were performed for each configuration, and results were averaged. As before, we evaluated model behavior under all three threshold conditions (50th, 75th, and 90th percentiles).

All experiments were conducted on a high-performance computing server equipped with an Intel i7-11700K 3.7 GHz CPU, two NVIDIA

GeForce RTX 3080 GPUs, 16 GB DDR4 RAM, and two Samsung 980 EVO Plus 1 TB SSDs, running Ubuntu 22.04. The experimental pipeline was implemented in Python. We used Docker for containerization and GPU acceleration to ensure reproducibility and efficiency. The Ollama framework was employed to locally deploy and query generative models via their respective APIs. Additionally, we used GNU Screen to manage long-running Python processes in the background, ensuring resilience in the face of network interruptions, as the total processing time could extend over several days.

6. Results

In this section, we present the results obtained from each experiment.

6.1. Experiment 1 - Full data balanced classes

Table 3 presents the classification performance of the three LLMs across the four complete and balanced datasets, constructed as described in Section 5.4. Among the tested models, `LLaMA3-8B` consistently outperforms both `LLaMA2-70B` and `Mistral`, achieving accuracy values approximately 15%–20% higher. This result highlights the importance of model-specific pre-trained knowledge in driving classification effectiveness, with `LLaMA3-8B` emerging as the most suitable model for bug fixing time classification in this setting. With only minor variations across datasets—typically within a few percentage points—each LLM exhibits stable performance, suggesting that the ability of these models to solve the task is relatively unaffected by differences in dataset size or domain. The highest accuracy, 71%, is achieved by `LLaMA3-8B` on the OpenXchange dataset.

An analysis of precision and recall values reveals that recall remains consistently high across all datasets, ranging between 90% and 100%. This indicates that the models are highly effective in identifying instances of the *FAST* class, suggesting an ability to recognize textual patterns associated with bugs that are resolved quickly. Notably, performance remains stable even across projects of varying scale and complexity, further supporting the hypothesis that LLMs are able to generalize their predictions based on the underlying semantic structure of the input.

However, the comparatively lower precision scores suggest that a substantial number of *SLOW* bugs are misclassified as *FAST*, resulting in a high false positive rate. While this does not impact the model's ability to detect fast-resolving bugs, it may compromise the reliability of task prioritization strategies by overestimating the ease of fixing for more complex issues.

The elevated rate of false positives, although lowest in `LLaMA3-8B` (approximately 40%), reflects a broader challenge across all models. A key difficulty lies in the intrinsic nature of the *SLOW* class, which includes bugs whose fixing time exceeds the 75th percentile of the distribution. These cases represent the most time-consuming and heterogeneous 25% of bug reports and often lack consistent linguistic patterns, making them difficult to recognize through language alone.

Unlike *FAST* bugs, which are more frequent and tend to follow recognizable, often repeated descriptions, *SLOW* bugs vary widely depending on contextual factors such as codebase complexity, development bottlenecks, or undocumented technical dependencies. The notion of a "long" fixing time is inherently context-dependent and not easily mapped to consistent surface-level features that an LLM can reliably detect.

This limitation is particularly evident in a zero-shot learning setting, where models rely solely on their pre-trained general-purpose knowledge without access to labeled task-specific data. Because fast-resolving bugs are more frequently represented in publicly available training corpora, LLMs may have internalized more reliable cues for identifying such instances. In contrast, slow-resolving bugs—being rarer and more context-sensitive—are likely underrepresented, making them harder to model. Consequently, when faced with uncertainty, the models tend to overgeneralize and favor the *FAST* prediction, increasing the likelihood of false positives.

Table 3
Computational performance of LLMs on the complete datasets, showing classification metrics for the Fast class (balanced).

Data	LLM	Accuracy	Precision	Recall	F1
W3C	Llama2:70b	52%	52%	98%	67%
	Llama3:8b	69%	62%	100%	77%
	Mistral	52%	52%	100%	68%
OpenXchange	Llama2:70b	53%	52%	95%	67%
	Llama3:8b	71%	64%	97%	77%
	Mistral	50%	50%	97%	67%
Novell	Llama2:70b	54%	53%	95%	68%
	Llama3:8b	67%	61%	99%	75%
	Mistral	51%	51%	99%	67%
Mozilla	Llama2:70b	54%	54%	90%	66%
	Llama3:8b	69%	62%	95%	75%
	Mistral	54%	52%	95%	68%

Kruskal–Wallis test results: F = 8.1889, p = 0.0167.

In summary, the results of Experiment 1 demonstrate that `LLaMA3-8B` is capable of effectively classifying bug fixing time using balanced full datasets, yielding strong performance across all evaluation metrics. The other two models, `LLaMA2-70B` and `Mistral`, perform significantly worse, suggesting that their internal knowledge representations may be less aligned with the requirements of this specific task. These findings validate the potential of LLMs in supporting bug triage but also point to current limitations in handling infrequent and heterogeneous classes such as *SLOW*.

6.2. Experiment 2 - sampled data balanced classes

Table 4 presents the average classification performance of the three LLMs over 50 runs using sampled datasets, under balanced class conditions, and varying the threshold used to define the *FAST* and *SLOW* classes. Each threshold setting corresponds to a different percentile of the bug fixing time distribution (50%, 75%, 90%).

We first focus on the results obtained using the 75th percentile as a threshold, to enable direct comparison with Experiment 1, which employed the same criterion on full datasets. As expected, the overall performance slightly decreases with respect to full data. This outcome is unsurprising, as the sampled setting utilizes a smaller amount of data, despite employing multiple random repetitions to ensure generalizability. Recall remains consistently high, with only a marginal decline, confirming that the models are still capable of correctly identifying nearly all bugs classified as *FAST*. This suggests that even when limited data is available, LLMs can effectively detect textual patterns associated with fast fixing times. However, a drop in precision is observed across all models, highlighting increased difficulty in correctly identifying *SLOW* bugs. This confirms the tendency of LLMs to produce a relatively high number of false positives when faced with edge or ambiguous cases. `LLaMA3-8B` and `LLaMA2-70B` deliver comparable performance in this setting, whereas `Mistral` reports consistently low values in both precision and recall. This renders it unreliable for classifying bug fixing times. Importantly, we observe no significant variation in performance across the different datasets, reinforcing the idea that model behavior is stable across heterogeneous projects. This supports the generalizability of the approach and suggests that dataset-specific characteristics have a limited impact on LLM performance under balanced conditions.

In addition to predictive performance, we also examined the computational efficiency of each model. As shown in Tables 7 and 8, processing time scales substantially with dataset size. For example, processing the complete Mozilla dataset with `LLaMA3-8B` requires approximately eight hours, compared to just three to four minutes

when using sampled data. The difference is even more pronounced for the other two models: LLaMA2-70B requires up to seven days, and Mistral up to two days for the same dataset. These computational costs make the use of full datasets impractical in real-time or resource-constrained environments.

Given that performance on sampled data remains largely comparable to that observed with full datasets—especially when using stronger models such as LLaMA3-8B—while offering a substantial reduction in processing time, the proposed approach is well-suited for real-world applications. In practical settings, where only a limited number of bug reports are submitted daily and timely decision-making is essential, using small, representative samples enables efficient and accurate triage.

We also investigated the impact of varying the threshold used to define class membership. In addition to the standard 75th percentile (as discussed in Section 4.2.2), we evaluated the effect of using the 50th and 90th percentiles, again under balanced class conditions. The results show no substantial difference in performance across the different thresholds. This suggests that the predictive behavior of the models is influenced more by the relative proportion of samples assigned to each class than by the absolute threshold value. These findings indicate that, although the models were primarily evaluated using the 75th percentile as a threshold, their performance remains stable across different threshold settings. This suggests that the approach is generalizable and can be effectively adapted to alternative threshold values, depending on the specific requirements of the application domain. In practice, this flexibility enables practitioners to tailor the classification boundary according to operational needs or risk tolerance, without compromising the overall reliability of the model.

6.3. Experiment 3 - unbalanced classes

In this experiment, we evaluated the performance of LLMs on unbalanced data, considering both the complete and sampled datasets. For the sampled case, each experimental configuration was repeated 50 times to ensure statistical robustness. The classification threshold was varied (50%, 75%, and 90%) to assess the impact of different class definitions. The corresponding results are reported in Tables 5 and 6 for the complete and sampled datasets, respectively. Due to the inherent bias of accuracy in imbalanced settings—where it tends to reflect the majority class—we chose to exclude it from the core analysis, although it is reported in the tables for completeness and consistency with previous experiments. Instead, we focused primarily on the F1 score as a more reliable indicator of predictive performance in the presence of class imbalance.

Focusing first on the results obtained from the complete datasets (Table 5), we observe that recall remains high across all models and datasets, with values consistently ranging between 90% and 100%. This confirms that the models continue to effectively identify bugs belonging to the *FAST* class, even when trained and tested on imbalanced data. Precision values show a substantial improvement compared to Experiments 1 and 2, reaching levels between approximately 75% and 90%. This improvement is expected, as the *FAST* class—being the majority class in this configuration—results in a lower number of false positives. In other words, fewer *SLOW* bugs are misclassified as *FAST*, leading to higher precision scores.

Among the models evaluated, LLaMA3-8B consistently achieves the best performance, with F1 scores ranging between 92% and 95%. These results confirm its reliability and suitability for predicting bug fixing time classes, particularly in scenarios with imbalanced data. While some minor dataset-specific variations are observed, such as slightly lower recall for the largest dataset, Mozilla, and slightly higher performance for the smallest dataset, W3C, no significant performance gaps emerge. This suggests that neither the scale of the dataset nor the specific content of the project strongly influences model effectiveness in this context.

Turning to the results obtained from the sampled datasets (Table 6), and focusing again on the 75th percentile threshold to enable comparison with the full data condition, we observe a slight decrease in both recall and precision. Precision values stabilize around 75%, and the overall performance, as reflected by the F1 score, decreases by approximately 10 percentage points for both LLaMA3-8B and LLaMA2-70B, from around 95% to 85%. Mistral, on the other hand, exhibits poor performance, with F1 scores consistently falling between 50% and 60%, indicating limited suitability for this task in unbalanced settings.

As previously discussed, computational time remains a critical limiting factor when processing large datasets. Full data processing with certain models is impractical for real-time applications. However, the use of sampled data significantly reduces computational cost while still achieving competitive performance. Despite the observed drop in F1 scores compared to the full datasets, the results remain comparable to those obtained in balanced settings, confirming the applicability of sampling strategies for real-world deployment. These findings further reinforce the suitability of LLaMA3-8B for practical use, particularly in operational contexts where most bug reports are resolved quickly. In such settings, the prevalence of *FAST* bugs naturally enhances predictive performance, as the model is better aligned with the dominant patterns in the data.

Finally, we analyzed the impact of the classification threshold on model performance. While recall remains consistently high across thresholds, demonstrating the models' ability to reliably identify *FAST* bugs (except for Mistral), precision values vary depending on the selected threshold. This confirms the trends observed in previous experiments and supports our hypothesis that LLMs operating in a zero-shot setting face difficulties in identifying *SLOW* bugs, which are more heterogeneous and less represented in the model's pre-training data. As the proportion of *SLOW* bugs increases (e.g., with the 50th percentile threshold), precision drops accordingly. Conversely, when using higher thresholds such as the 90th percentile, which define *SLOW* bugs as only the most extreme cases, precision improves substantially, reaching values around 90%.

6.4. Significance testing of model performance

The Kruskal–Wallis test (McKight and Najab, 2010) was employed to examine the statistical significance of differences among the performance metrics of various LLMs across multiple datasets. Before conducting the Kruskal–Wallis test, we assessed the normality of the data distributions within each group using the Shapiro–Wilk test. The results indicated deviations from normality, confirming the need for a non-parametric approach. Since the Kruskal–Wallis test does not assume normality, it provided a robust alternative for comparing model performance.

This non-parametric method is advantageous when comparing three or more independent groups without assuming normality, making it an appropriate choice given the nature of our data. The Kruskal–Wallis test determines whether at least one group distribution significantly differs from the others by ranking the data and analyzing the variance of ranks among groups.

Additionally, we evaluated the assumption of equal variances using Levene's test, as homogeneity of variances can impact the interpretation of parametric tests. The results revealed heterogeneity across groups, further reinforcing the appropriateness of the Kruskal–Wallis test. In addition, the Kruskal–Wallis test is less sensitive to equal variance assumptions, making it a reliable choice for our analysis.

The Kruskal–Wallis test revealed statistically significant differences among the groups, indicating that variations in model performance were unlikely to be attributed to random chance.

Table 4
Computational performance of LLMs on sampled datasets, with average classification metrics for the *FAST* class over 50 runs, using classification thresholds of 50%, 75%, and 90% under balanced class conditions.

Data	LLM	Percentile	Accuracy	Precision	Recall	F1
W3C	Llama2:70b	50	45.91%	46.1%	99.02%	62.91%*
		75	50.72%	51.91%	97.8%	66.99%*
		90	52.62%	52.59%	99.52%	68.78%*
	Llama3:8b	50	46.16%	46.21%	99.38%	63.09%*
		75	50.15%	50.08%	99.75%	66.68%*
		90	51.1%	51.13%	99.03%	67.43%*
	Mistral	50	51.36%	48.3%	71.73%	57.71%*
		75	51.56%	52.18%	65.76%	57.86%*
		90	49.14%	51.51%	54.81%	52.93%*
OpenXchange	Llama2:70b	50	50.5%	50.85%	98.66%	66.84%*
		75	50.42%	50.48%	99.11%	66.87%*
		90	50.5%	50.26%	99.93%	66.88%*
	Llama3:8b	50	50.3%	50.15%	99.28%	66.64%*
		75	50.94%	50.62%	99.31%	67.05%*
		90	49.46%	49.72%	98.29%	66.04%*
	Mistral	50	49.64%	50.98%	45.72%	48.07%*
		75	47.64%	48.58%	38.68%	42.89%*
		90	49.7%	49.73%	55.17%	52.19%*
Novell	Llama2:70b	50	51.69%	51.7%	99.27%	67.99%*
		75	50.6%	50.67%	99.21%	67.08%*
		90	58.67%	58.52%	99.77%	73.76%*
	Llama3:8b	50	52.59%	52.16%	99.74%	68.5%*
		75	51.39%	51.07%	99.18%	67.42%*
		90	58.77%	58.61%	99.49%	73.76%*
	Mistral	50	47.73%	49.35%	42.58%	45.68%*
		75	47.28%	47.54%	38.15%	42.28%*
		90	48.87%	58.43%	42.44%	49.04%*
Mozilla	Llama2:70b	50	53.83%	53.8%	99.03%	69.72%*
		75	58.17%	58.21%	99.08%	73.32%*
		90	49.6%	49.8%	98.07%	66.05%*
	Llama3:8b	50	54.3%	54.1%	98.07%	69.72%*
		75	58.09%	58.15%	98.24%	73.05%*
		90	49.5%	49.74%	98.0%	65.99%*
	Mistral	50	48.01%	51.48%	52.91%	52.16%*
		75	47.43%	54.91%	50.98%	52.81%*
		90	49.37%	49.37%	58.0%	53.27%*

* p-value < 0.05.

Table 5
Computational performance of LLMs on the complete datasets, showing classification metrics for the Slow and Fast classes (unbalanced).

Data	LLM	Accuracy	Precision	Recall	F1
W3C	Llama2:70b	77%	77%	98%	86%
	Llama3:8b	92%	90%	100%	95%
	Mistral	78%	78%	99%	87%
OpenXchange	Llama2:70b	74%	76%	95%	84%
	Llama3:8b	85%	89%	91%	90%
	Mistral	74%	74%	98%	85%
Novell	Llama2:70b	75%	77%	94%	85%
	Llama3:8b	91%	90%	99%	94%
	Mistral	74%	75%	98%	85%
Mozilla	Llama2:70b	72%	76%	90%	83%
	Llama3:8b	88%	91%	94%	92%
	Mistral	73%	76%	93%	84%

Kruskal–Wallis test results: F = 7.7607, p = 0.0206.

In the experiments involving sampled bugs, the performance metrics were averaged across the 50 generated samples, each consisting of 300 examples. To assess the statistical significance of the observed differences among experimental conditions, a one-way Analysis of Variance (ANOVA) test was conducted. This test determines whether there are statistically significant differences between group means, assuming normality and homogeneity of variances. No post-hoc analysis was performed, as the ANOVA results were used solely to determine the presence or absence of overall significant effects.

7. Discussion

RQ1: To what extent can LLMs accurately classify bug reports as FAST or SLOW within the zero-shot workflow introduced in this study?

The results indicate that LLMs, when used within the proposed zero-shot classification workflow, can be reasonably effective in predicting bug fixing times, with their performance varying according to the specific model employed. Among the models tested, `LLaMA3-8B` consistently demonstrated the most reliable predictive capabilities, outperforming the others across all experimental settings. The consistently high recall values suggest that LLMs are highly effective at identifying patterns associated with fast-resolving bugs, a property that is particularly valuable for supporting task prioritization in bug-tracking systems.

However, the observed lower precision highlights a tendency to misclassify some slow-resolving bugs as fast. This issue is particularly pronounced in the balanced data setting, where the number of *SLOW* and *FAST* bugs is artificially equalized, a condition that does not typically reflect real-world distributions. The balanced setting was intentionally constructed to stress-test the models under controlled conditions, rather than to mirror operational scenarios.

In conclusion, while LLMs are not without limitations, particularly in distinguishing between rare or complex cases, their predictive capabilities have demonstrated promise in enhancing estimation processes

Table 6
Computational performance of LLMs on sampled datasets, with average classification metrics for the *FAST* class over 50 runs, using classification thresholds of 50%, 75%, and 90% under not balanced class conditions.

Data	LLM	Percentile	Accuracy	Precision	Recall	F1
W3C	Llama2:70b	50	48.48%	48.67%	98.79%	65.2%*
		75	73.15%	73.63%	98.9%	84.41%*
		90	89.63%	90.27%	99.21%	94.53%*
	Llama3:8b	50	48.8%	48.85%	99.5%	65.52%*
		75	73.63%	73.75%	99.71%	84.79%*
		90	89.57%	89.91%	99.57%	94.49%*
	Mistral	50	52.47%	50.97%	72.12%	59.72%*
		75	58.39%	74.64%	65.92%	69.99%*
		90	55.45%	90.58%	56.56%	69.6%*
OpenXchange	Llama2:70b	50	51.17%	52.1%	97.14%	67.5%*
		75	75.62%	76.74%	98.1%	86.0%*
		90	90.29%	90.88%	99.27%	94.89%*
	Llama3:8b	50	51.87%	51.7%	99.35%	68.0%*
		75	76.43%	76.85%	99.1%	86.57%*
		90	89.3%	90.57%	98.46%	94.34%*
	Mistral	50	49.22%	50.79%	45.28%	47.84%*
		75	40.7%	74.29%	35.78%	48.12%*
		90	51.29%	90.06%	52.1%	65.99%*
Novell	Llama2:70b	50	52.25%	52.2%	99.36%	68.44%*
		75	75.11%	75.42%	99.39%	85.76%*
		90	91.01%	91.78%	99.06%	95.28%*
	Llama3:8b	50	53.1%	52.69%	99.69%	68.94%*
		75	75.33%	75.64%	99.22%	85.84%*
		90	91.07%	91.77%	99.15%	95.32%*
	Mistral	50	47.82%	50.05%	42.14%	45.71%*
		75	42.29%	72.77%	37.42%	49.39%*
		90	40.03%	91.43%	38.15%	53.79%*
Mozilla	Llama3:8b	50	54.63%	54.52%	99.09%	70.34%*
		75	78.41%	79.13%	98.73%	87.85%*
		90	86.99%	88.19%	98.43%	93.03%*
	Llama3:8b	50	54.96%	54.76%	98.0%	70.25%*
		75	78.0%	79.12%	97.96%	87.54%*
		90	86.53%	88.07%	97.96%	92.75%*
	Mistral	50	47.96%	52.03%	52.57%	52.27%*
		75	48.9%	76.67%	50.64%	60.96%*
		90	55.81%	87.06%	58.47%	69.93%*

* p-value < 0.05.

Table 7
Computational times (in seconds) of the three LLM with the four sampled datasets, balanced and unbalanced.

Data	Balanced	LLama3:8b	LLama2:70b	Mistral
W3C	Yes	148	2266	239
	No	148	2242	247
OpenXchange	Yes	148	2266	239
	No	148	2242	247
Novell	Yes	111	1702	187
	No	110	1670	184
Mozilla	Yes	245	3101	388
	No	216	3249	361

Table 8
Computational times (in seconds) of the three LLM with the four complete datasets, balanced and unbalanced.

Data	Balanced	LLama3:8b	LLama2:70b	Mistral
W3C	Yes	157	2378	263
	No	307	3061	358
OpenXchange	Yes	243	3659	403
	No	469	4679	853
Novell	Yes	10,611	160,584	17,685
	No	10,601	118,045	13,579
Mozilla	Yes	91,374	591,977	152,290
	No	91,100	564,218	141,110

during bug triage. These findings suggest that LLMs can support more efficient software maintenance practices, provided that well-optimized models are selected to ensure both accuracy and robustness.

RQ2: To what extent can LLMs accurately classify bug reports as FAST or SLOW within the zero-shot workflow introduced in this study, when only a limited number of bug reports are available?

Our experiments show that LLMs can still provide reasonable predictions even when trained on a limited number of bug reports. Although classification performance slightly decreases with reduced data, the models retain their predictive capacity, suggesting that their extensive pre-training allows them to generalize well even in data-scarce scenarios. This is particularly relevant for bug-tracking systems where historical data is limited. Additionally, smaller datasets significantly reduce computational time, making real-time or near-real-time bug fixing predictions more feasible. These results suggest that LLMs can be effectively utilized even in environments where only a small number of bug reports are available, supporting efficient decision-making in software maintenance.

RQ3: To what extent does class imbalance impact the ability of LLMs to accurately classify bug reports as FAST or SLOW within the proposed zero-shot classification workflow?

In real-world Bug Tracking Systems, fast-resolving bugs occur more frequently than slow ones, leading to naturally imbalanced datasets. Our findings show that LLMs, when used within the proposed zero-shot classification workflow, are influenced by this imbalance and tend to

favor the majority class. Specifically, the models perform better at identifying fast-resolving bugs, likely due to their greater prevalence in both real-world scenarios and the corpora used during pre-training. This pre-existing knowledge benefits the zero-shot setting, where models rely exclusively on internal representations without task-specific fine-tuning. Conversely, the classification of slow-resolving bugs remains more challenging, as such cases are underrepresented in both LLMs' training data and real BTSs, and are often more heterogeneous in nature.

This bias is evidenced by the higher precision values obtained in experiments on unbalanced datasets—both complete and sampled—where the proportion of *FAST* bugs was higher. As the number of examples from the *FAST* class increases, the predictive performance of the LLMs improves. While this confirms a limitation in recognizing rare and complex bug reports, it is worth noting that such an imbalance reflects the real-world distribution of bugs.

Therefore, although the lower precision observed for the *SLOW* class highlights a current limitation, its practical impact is mitigated by the relatively low frequency of such cases in operational contexts. Consequently, the proposed LLM-based workflow remains a viable and reliable solution. Furthermore, while using full datasets can improve performance, the associated computational costs may hinder scalability. Our results confirm that sampling represents an effective compromise, as it substantially reduces computational time while maintaining acceptable predictive accuracy. This reinforces the applicability of LLM-based approaches for predicting large-scale bug fixing, even under class imbalance.

8. Threats to validity

This section outlines the potential threats to the validity of the research, categorized according to Runeson and Höst (2009) guidelines.

- **Construct Validity**. The research relies on estimating bug-fixing time using a uniform distribution of developer work hours. Since the actual time spent and daily distribution of hours are not publicly available, this assumption may not accurately reflect the real effort expended by developers. This discrepancy can impact the construct validity of the model, as the estimations may not fully capture the complexity of bug-fixing processes. Field Selection: The choice of input features used by the model is critical to its performance. While LLMs are capable of processing textual data, the relevance and quality of selected fields directly affect the model's ability to interpret and predict bug-fixing times accurately. Including irrelevant or less significant fields might introduce noise and potentially skew predictions, affecting the construct validity of the model.
- **Internal Validity**. The datasets used may contain outliers in bug-fixing time, which can distort the training process and affect the accuracy of predictions. While removing outliers can enhance data quality, it also risks excluding valuable data points that contribute to a comprehensive understanding of bug-fixing dynamics. This management of outliers could impact the internal validity by introducing potential biases in the model's training and evaluation. Furthermore, our study does not account for the possibility of bug reports being reopened after closure. Reopened bugs were treated as new reports, which could invalidate the calculation of bug-fixing time in some cases. This oversight could lead to inaccuracies in the dataset and affect the internal validity of the predictions, as the model may not fully represent the true nature of bug fixing.
- **External Validity**. The experiments were conducted using datasets from four Bugzilla instances. While these projects are significant, they do not encompass the entire spectrum of open-source software. Therefore, the results may not generalize to other open-source projects or software development environments, limiting the external validity of the findings. Furthermore, the proposed model was tested only on open-source projects. It is unclear whether the model would perform effectively on proprietary software, which often involves different team structures and predefined criteria for bug handling. This limitation affects the external validity, as the model's effectiveness in proprietary settings remains uncertain.
- **Reliability**. The use of a single prompt for generating predictions might narrow the interpretation of input data, potentially limiting the model's ability to generalize across diverse bug reports. This reliance on a single prompt could introduce bias and affect the robustness and accuracy of the predictions. To enhance reliability, employing a more varied set of prompts or methods could provide a broader perspective and improve the model's consistency.

9. Conclusion

This study investigated the effectiveness of Large Language Models in classifying bug reports according to their expected fixing time within a novel zero-shot workflow. The proposed approach integrates a prompt design based on a combination of persona patterns and input semantics to guide the reasoning of the model. Specifically, we evaluated the ability of LLMs to categorize bug reports as either *FAST* or *SLOW*, using a predefined temporal threshold.

The results demonstrate that LLMs are capable of extracting meaningful information from textual input and can reliably predict whether a bug is likely to be resolved quickly, offering valuable support for task prioritization in bug-tracking systems. Among the models evaluated, `LLaMA3-8B` consistently delivered the best performance, while other models showed greater variability across settings.

Although all models performed well in detecting fast-resolving bugs, their ability to correctly identify slow-resolving ones was more limited, particularly under the balanced data setting, which was artificially constructed to stress-test the models and does not reflect real-world distributions. In contrast, in the more realistic unbalanced scenarios, where fast-resolving bugs are the majority—the models, and especially `LLaMA3-8B`, achieved high predictive performance, with F1 scores reaching up to 95% on full datasets. These findings highlight that, while misclassifications (notably false positives) remain a limitation, their practical impact may be reduced in operational contexts where slow bugs are less frequent.

Importantly, this work represents one of the first empirical evaluations of LLMs for bug fixing time prediction in a zero-shot setting, i.e., without additional training or external supervision. The aim was to provide a foundational assessment of their capabilities using only their intrinsic knowledge. The results are promising and suggest that LLMs can serve as effective tools even in early stage or low-resource software projects.

Another notable finding is the robustness of LLMs in data-scarce settings: their predictive performance remained stable when applied to limited subsets of bug reports. This suggests strong generalization capabilities inherited from pretraining, making them well-suited for domains where historical bug data is sparse or inconsistently available.

However, computational cost remains a critical constraint. Running inference on large-scale datasets with state-of-the-art LLMs can be time-intensive, limiting their real-time applicability. Our experiments show that sampling offers a viable compromise, significantly reducing inference time while maintaining competitive predictive accuracy, thus enhancing the feasibility of deployment in production systems.

We adopted the commonly used 75th percentile threshold to define fixing time classes, in line with existing literature. However, experiments with varying thresholds confirmed the generalizability of the approach, demonstrating that the workflow can be adapted to specific application contexts by tuning the classification threshold.

Future work will focus on addressing the identified limitations by enhancing model precision and reducing the incidence of false positives, particularly in the classification of slow-resolving bugs. One promising direction is the refinement of prompt design through prompt engineering techniques, which can improve both predictive accuracy and output interpretability. In addition, injecting external task-specific knowledge into LLMs may offer significant benefits. This can be achieved through strategies such as few-shot learning, providing the model with curated examples to guide its reasoning, and/or fine-tuning on domain-specific datasets to better align the model's internal representations with the nuances of bug fixing scenarios. Hybrid approaches that combine LLMs with traditional machine learning methods can also be explored to leverage complementary strengths.

Furthermore, extending the evaluation to include proprietary software systems and various bug tracking platforms will be essential to validate the robustness and portability of the approach. Lastly, optimizing inference through lightweight architectures or more efficient pipelines will be critical to supporting real-time decision-making in resource-constrained production environments.

CRediT authorship contribution statement

Pasquale Ardimento: Validation, Methodology, Conceptualization, Software, Supervision, Data curation, Writing – original draft. **Michele Capuzzimati:** Formal analysis, Investigation, Data curation, Software. **Gabriella Casalino:** Writing – original draft, Conceptualization, Validation, Methodology, Supervision. **Daniele Schicchi:** Writing – original draft, Supervision, Methodology, Software, Conceptualization, Validation. **Davide Taibi:** Writing – original draft, Resources.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Dataset is available.

References

Abdelmoez, W., Kholief, M., Elsalmy, F., 2012. Bug fix-time prediction model using Naïve Bayes classifier. In: 2012 22nd International Conference on Computer Theory and Applications. ICCTA, pp. 1–6. http://dx.doi.org/10.1109/ICCTA.2012.6523564.

Akbarinasaji, S., Caglayan, B., Bener, A., 2018. Predicting bug-fixing time: A replication study using an open source software project. J. Syst. Softw. 136, 173–186. http://dx.doi.org/10.1016/J.JSS.2017.02.021.

Anbalagan, P., Vouk, M., 2009. On predicting the time taken to correct bug reports in open source projects. In: 2009 IEEE International Conference on Software Maintenance. IEEE, pp. 523–526. http://dx.doi.org/10.1109/ICSM.2009.5306337.

Ardimento, P., 2022. Predicting bug-fixing time: Distilbert versus google BERT. In: Taibi, D., Kuhrmann, M., Mikkonen, T., Klünder, J., Abrahamsson, P. (Eds.), Product-Focused Software Process Improvement - 23rd International Conference, PROFES 2022, Jyväskylä, Finland, November 21-23, 2022, Proceedings. In: Lecture Notes in Computer Science, vol. 13709, Springer, pp. 610–620. http://dx.doi.org/10.1007/978-3-031-21388-5_46.

Ardimento, P., 2023. Enhancing bug-fixing time prediction with LSTM-based approach. In: Kadgien, R., Jedlitschka, A., Janes, A., Lenarduzzi, V., Li, X. (Eds.), Product-Focused Software Process Improvement - 24th International Conference, PROFES 2023, Dornbirn, Austria, December 10-13, 2023, Proceedings, Part II. In: Lecture Notes in Computer Science, vol. 14484, Springer, pp. 68–79. http://dx.doi.org/10.1007/978-3-031-49269-3_7.

Ardimento, P., Bilancia, M., Monopoli, S., 2016. Predicting bug-fix time: Using standard versus topic-based text categorization techniques. In: Calders, T., Ceci, M., Malerba, D. (Eds.), Discovery Science - 19th International Conference, DS 2016, Bari, Italy, October 19-21, 2016, Proceedings. In: Lecture Notes in Computer Science, vol. 9956, pp. 167–182. http://dx.doi.org/10.1007/978-3-319-46307-0_11.

Ardimento, P., Boffoli, N., 2022. A supervised generative topic model to predict bug-fixing time on open source software projects. In: Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering. ENASE, SCITEPRESS, pp. 233–240. http://dx.doi.org/10.5220/0011113100003176.

Ardimento, P., Boffoli, N., Mele, C., 2020. A text-based regression approach to predict bug-fix time. In: Appice, A., Ceci, M., Loglisci, C., Manco, G., Masciari, E., Ras, Z.W. (Eds.), Complex Pattern Mining - New Challenges, Methods and Applications. In: Studies in Computational Intelligence, vol. 880, Springer, pp. 63–83. http://dx.doi.org/10.1007/978-3-030-36617-9_5.

Ardimento, P., Casalino, G., Schicchi, D., Taibi, D., 2025. Bug reports from bug tracking systems. http://dx.doi.org/10.17632/st94bmxfs4.1.

Ardimento, P., Mele, C., 2020. Using BERT to predict bug-fixing time. In: 2020 IEEE Conference on Evolving and Adaptive Intelligent Systems, EAIS 2020, Bari, Italy, May 27-29, 2020. IEEE, pp. 1–7. http://dx.doi.org/10.1109/EAIS48028.2020.9122781.

Ayari, K., Abran, A., Desharnais, J.-M., 2004. Eclipse vs. Mozilla: A comparison of two large-scale open source problem report repositories. In: Proceedings of the 7th International Workshop on Principles of Software Evolution. IEEE, pp. 100–103. http://dx.doi.org/10.1109/HASE.2015.45.

Bettenburg, N., Robbes, R., Hassan, A.E., Robillard, M.P., 2012. Bug-fix time prediction models: Can we do better? In: Proceedings of the 8th Working Conference on Mining Software Repositories. MSR, pp. 207–210. http://dx.doi.org/10.1109/MSR.2012.6224280.

Bhattacharya, P., Neamtiu, I., 2011. Bug-fix time prediction models: can we do better? In: Proceedings of the 8th Working Conference on Mining Software Repositories. pp. 207–210. http://dx.doi.org/10.1145/1985441.1985472.

Bocu, R., Baicoianu, A., Kerestely, A., 2023. An extended survey concerning the significance of artificial intelligence and machine learning techniques for bug triage and management. IEEE Access 11, 123924–123937. http://dx.doi.org/10.1109/ACCESS.2023.3329732.

Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y.T., Li, Y., Lundberg, S., et al., 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. arXiv preprint arXiv:2303.12712.

2025. Bugzilla: List of users. https://www.bugzilla.org/installation-list/. (Accessed 14 May 2025).

Chang, Y., Wang, X., Wang, J., Wu, Y., Yang, L., Zhu, K., Chen, H., Yi, X., Wang, C., Wang, Y., et al., 2024. A survey on evaluation of large language models. ACM Trans. Intell. Syst. Technol. 15 (3), 1–45.

Chang, Y.-C., Wang, X., Wang, J., Wu, Y., Zhu, K., Chen, H., Yang, L., Yi, X., Wang, C., Wang, Y., Ye, W., Zhang, Y., Chang, Y., Yu, P.S., Yang, Q., Xie, X., 2023. A survey on evaluation of large language models. ACM Trans. Intell. Syst. Technol. 15, 1–45. http://dx.doi.org/10.1145/3641289.

Chen, B., Zhang, Z., Langrené, N., Zhu, S., 2025. Unleashing the potential of prompt engineering for large language models. Patterns (ISSN: 2666-3899) 6 (6), 101260. http://dx.doi.org/10.1016/j.patter.2025.101260, URL: https://www.sciencedirect.com/science/article/pii/S2666389925001084.

Chen, Y., Zhang, K., Wang, W., Wang, L., 2024. LLM agents for software engineering: From capabilities to guidelines. arXiv preprint arXiv:2402.14332.

Colavito, G., Lanubile, F., Novielli, N., 2025. Benchmarking large language models for automated labeling: The case of issue report classification. Inf. Softw. Technol. 184, 107758. http://dx.doi.org/10.1016/j.infsof.2025.107758.

Datta, S., Lade, P., 2015. Will this be quick? A case study of bug resolution times across open source projects. In: Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. MSR, pp. 134–143. http://dx.doi.org/10.1145/2723742.2723744.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al., 2024. The llama 3 herd of models. arXiv preprint arXiv:2407.21783.

Foundation, M., 2025. Bugzilla @ Mozilla. https://bugzilla.mozilla.org/. (Accessed 06 May 2025).

Gomes, P., Valente, M.T., Bigonha, M.A., 2021. On the prediction of long-lived bugs. Inf. Softw. Technol. 129, 106412. http://dx.doi.org/10.1016/j.infsof.2020.106412.

Gong, L., Abdelmoez, W., Kholief, M., 2013. Predicting bug-fixing time: An empirical study of commercial software projects. In: Proceedings of the 2013 International Conference on Software Engineering Research and Practice. SERP, pp. 1–7.

Gunasekaran, K.P., Babrich, B.C., Shirodkar, S., Hwang, H., 2023. Text2Time: Transformer-based article time period prediction. In: 2023 IEEE 6th International Conference on Pattern Recognition and Artificial Intelligence. PRAI, IEEE, pp. 449–455. http://dx.doi.org/10.1109/PRAI59366.2023.10331985.

Habayeb, M., Murtaza, S.S., Miranskyy, A., Bener, A.B., 2018. On the use of hidden Markov model to predict the time to fix bugs. IEEE Trans. Softw. Eng. 44 (12), 1224–1244. http://dx.doi.org/10.1109/TSE.2017.2757480.

Hegselmann, S., Buendia, A., Lang, H., Agrawal, M., Jiang, X., Sontag, D., 2023. Tabllm: Few-shot classification of tabular data with large language models. In: International Conference on Artificial Intelligence and Statistics. PMLR, pp. 5549–5581.

Hooimeijer, P., Weimer, W., 2007. Modeling bug report quality. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA. ACM, pp. 34–43. http://dx.doi.org/10.1145/1321631.1321639.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., Wang, H., 2023. Large language models for software engineering: A systematic literature review. arXiv preprint arXiv:2308.10620.

Jiang, A.Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D.S., Casas, D.d.l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al., 2023. Mistral 7B. arXiv preprint arXiv:2310.06825.

Kawamura, A.N., et al., 2023. Deep learning and gradient-based extraction of bug report features for bug fixing time prediction. Front. Comput. Sci. 5, 1032440. http://dx.doi.org/10.3389/fcomp.2023.1032440.

Kholief, M., Elsalmy, F., Abdelmoez, W., 2013. Improving bug fix-time prediction model by filtering out outliers. In: Proceedings of the 2013 International Conference on Technological Advances in Electrical, Electronics and Computer Engineering. TAEECE, pp. 1–6. http://dx.doi.org/10.1109/TAEECE.2013.6557306.

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., Neubig, G., 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM Comput. Surv. 55 (9), 1–35.

Liu, H., Zhang, M., Lin, Z., Zhao, Z., 2024. An empirical study of LLMs for code documentation generation. arXiv preprint arXiv:2403.01986.

Luo, S., Wang, J., Zhou, A., Ma, L., Song, L., 2024. Large language models augmented rating prediction in recommender system. In: ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing. ICASSP, IEEE, pp. 7960–7964.

Malhotra, R., Kaur, A., Sibal, R., 2018. A systematic literature review on the usage of bug tracking systems in software development. J. Syst. Softw. 137, 320–336. http://dx.doi.org/10.1016/j.jss.2017.12.002.

Marks, D.J., Devanbu, P.T., Filkov, V., 2011. Developer beliefs about bug fixing and productivity. In: Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement. ESEM, IEEE, pp. 1–10. http://dx.doi.org/10.1109/ESEM.2011.36.

McAvinue, S., Dev, K., 2025. Comparative evaluation of large language models using key metrics and emerging tools. Expert Syst. 42 (2), http://dx.doi.org/10.1111/exsy.13719, URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85203066672&doi=10.1111/fexsy.13719&partnerID=40&md5=31d9a42926818640d3ea3b96babd1661.

McKight, P.E., Najab, J., 2010. Kruskal–Wallis test. Corsini Encycl. Psychol. 1–1.

Medsker, L.R., Jain, L., et al., 2001. Recurrent neural networks. Des. Appl. 5 (64–67), 2.

Mihaylov, M., 2019. Predicting the resolution time and priority of bug reports: A Deep Learning Approach (Master's Thesis). University of Strathclyde, Master's Thesis.

Nagwani, N.K., Suri, J.S., 2023. An artificial intelligence framework on software bug triaging, technological evolution, and future challenges: A review. Int. J. Inf. Manag. Data Insights 3 (1), 100153. http://dx.doi.org/10.1016/J.JJIMEI.2022.100153.

Panjer, L.D., 2007. Predicting eclipse bug lifetimes. In: Fourth International Workshop on Mining Software Repositories. MSR'07: ICSE Workshops 2007, IEEE, http://dx.doi.org/10.1109/MSR.2007.25, 29–29.

Peeperkorn, M., Kouwenhoven, T., Brown, D., Jordanous, A., 2024. Is temperature the creativity parameter of large language models? arXiv preprint arXiv:2405.00492.

Ren, Y., Chen, Y., Liu, S., Wang, B., Yu, H., Cui, Z., 2024. TPLLM: A traffic prediction framework based on pretrained large language models. arXiv:2403.02221. URL: https://arxiv.org/abs/2403.02221.

Repository, P., 2015. PROMISE - predictive models in software engineering. https://openscience.us/repo/. (Accessed 06 May 2025).

Rodrigues, B.R.d., Costa, J.M., 2024. Predicting bug-fixing time with rating features. An. Comput. Beach 15, 30–36. http://dx.doi.org/10.14210/acotb.v15n1.p30-36.

Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. 14, 131–164. http://dx.doi.org/10.1007/S10664-008-9102-8.

Salmi, L., Lewis, D.M., Clarke, J.L., Dong, Z., Fischmann, R., McIntosh, E.I., Sarabu, C.R., DesRoches, C.M., 2025. A proof-of-concept study for patient use of open notes with large language models. JAMIA Open 8 (2), ooaf021.

Schicchi, D., Taibi, D., 2024. Redefining education: A personalized AI platform for enhanced learning experiences. In: Taibi, D., Schicchi, D., Temperini, M., Limongelli, C., Casalino, G. (Eds.), Proceedings of the Second International Workshop on Artificial INtelligent Systems in Education Co-Located with 23rd International Conference of the Italian Association for Artificial Intelligence (AIxIA 2024), Bolzano, Italy, November 26, 2024. In: CEUR Workshop Proceedings, vol. 3879, CEUR-WS.org, pp. 1–10, URL: https://ceur-ws.org/Vol-3879/AIxEDU2024_paper_37.pdf.

Sobo, A., Mubarak, A., Baimagambetov, A., Polatidis, N., 2025. Evaluating LLMs for code generation in HRI: A comparative study of ChatGPT, Gemini, and Claude. Appl. Artif. Intell. 39 (1), http://dx.doi.org/10.1080/08839514.2024.2439610.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al., 2023. Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288.

Tsabari, S., Segal, A., Gal, K., 2023. Predicting bug-fix time in students' programming with deep language models. In: Proceedings of the 16th International Conference on Educational Data Mining. EDM, pp. 396–405. http://dx.doi.org/10.5281/zenodo.8115733.

Uddin, J., Ghazali, R., Deris, M.M., Naseem, R., Shah, H., 2017. A survey on bug prioritization. Artif. Intell. Rev. 47, 145–180.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. Adv. Neural Inf. Process. Syst. 30.

Wang, Y., Kang, S., Luo, S., Wang, Y., Sun, Y., 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859.

Wang, A.N., et al., 2019. Predicting bug fix time using word embedding and deep learning. IET Softw. 13 (6), 523–530. http://dx.doi.org/10.1049/iet-sen.2019.0260.

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., Schmidt, D.C., 2023a. A prompt pattern catalog to enhance prompt engineering with chatgpt. arXiv preprint arXiv:2302.11382.

White, J., et al., 2023b. Evaluating persona prompting for question answering tasks. https://www.dre.vanderbilt.edu/~schmidt/PDF/Evaluating_Personified_Expert_Effectiveness_Conference.pdf.

Xia, X., Lo, D., Wang, X., Li, B., Xing, Z., 2016. Accurate developer recommendation for bug resolution. ACM Trans. Softw. Eng. Methodol. (TOSEM) 25 (1), 1–55. http://dx.doi.org/10.1145/2808791.

Xuan, J., Jiang, H., Ren, Z., Luo, Z., Wu, X., Zhang, W., 2012. Towards effective bug triage with software data reduction techniques. In: Proceedings of the 2012 ACM SIGSOFT Symposium on the Foundations of Software Engineering. FSE, ACM, pp. 1–11. http://dx.doi.org/10.1145/2393596.2393660.

Yücel, G., Tosun, A., 2022. Predicting bug-fixing time with machine learning - a collaborative filtering approach. J. Softw.: Evol. Process. 34 (5), e2350. http://dx.doi.org/10.1002/smr.2350.

Zhang, H., Gong, L., Versteeg, S., 2013. Predicting bug-fixing time: an empirical study of commercial software projects. In: 2013 35th International Conference on Software Engineering. ICSE, IEEE, pp. 1042–1051. http://dx.doi.org/10.1109/ICSE.2013.6606654.

Zheng, Z., Ning, K., Zhong, Q., Chen, J., Chen, W., Guo, L., Wang, W., Wang, Y., 2025. Towards an understanding of large language models in software engineering tasks. Empir. Softw. Eng. 30 (2), 50. http://dx.doi.org/10.1007/S10664-024-10602-0.

Pasquale Ardimento is an Associate Professor at the University of Bari Aldo Moro, Italy. His research spans software engineering and educational technologies, with a current focus on integrating AI into software engineering education to enhance learning through intelligent tools and analytics. He has developed AI-powered systems such as UML Miner, a Visual Paradigm plugin that provides personalized feedback to students learning UML modeling. He has authored over 90 peer-reviewed publications and actively contributes to the academic community as a reviewer, workshop organizer, and program committee member for international conferences in software engineering and technology-enhanced learning. He is also involved in national and EU-funded research projects and is a member of the Italian National Interuniversity Consortium for Informatics (CINI).

Michele Capuzzimati is a master's student in Computer Science at the University of Bari Aldo Moro, with a strong passion for Software Engineering. His expertise extends to Agile Integration, focusing on developing microservices in Java and Software Integration, with a particular emphasis on Enterprise Service Bus (ESB). His research interests range from software engineering to machine learning and cybersecurity.

Gabriella Casalino is currently an Assistant Professor (Tenure Track) at the Computational Intelligence Laboratory (CILab) of the Informatics Department of the University of Bari. Her research is focused on Computational Intelligence methods for interpretable data analysis. She is actively involved in eHealth, Data Stream Mining, and eXplainable Artificial Intelligence. Her work primarily focuses on the medical and educational domains. She holds membership in the IEEE Task Force on Explainable Fuzzy Systems, the Interdepartmental Center for Telemedicine of the University of Bari- CITEL, and the HELMeTO Task Force, which concentrates on Higher Education Learning Methodologies and Technologies Online. She is an active member of the computer science community and contributes by organizing committees of workshops and special sessions in prestigious international conferences such as ECAI and IEEE WCCI. Additionally, she is an Associate Editor for the international journals Scientific Reports, IEEE Transactions on Computational Social Systems, and Soft Computing. She is a Guest Editor for

several special issues in IEEE SMC Magazine, IEEE Transactions on Computational Social Systems, and IEEE Systems Journal. She is a Senior member of the IEEE Society and has received several awards for her research, including the prestigious FUZZ-IEEE Best Paper award.

Daniele Schicchi, Ph.D., is a researcher specializing in Information and Communication Technology, with extensive experience in academic and research institutions, including the University of Palermo and the Institute for Educational Technology at National Research Council of Italy. His research focuses on Artificial Intelligence, particularly its applications in education and natural language processing, with expertise in Learning Analytics, Educational Data Mining, and Text Simplification through Machine Learning techniques. Throughout his career, he has taken on leadership roles, serving as the Chair of multiple international conferences and workshops on AI and education. Additionally, he has contributed as a Technical Program Committee member for numerous international conferences on Artificial Intelligence. As a speaker, he has presented at various international conferences. His research output includes significant contributions to AI applications in language processing and education, with notable publications in journals and conference proceedings.

Davide Taibi is a Senior Researcher at the Institute for Educational Technology of the National Research Council of Italy (CNR-ITD). His research primarily focuses on the application of innovative technologies to support education at various levels, covering areas such as Ubiquitous Learning, Semantic Web and Linked Data, Learning Analytics, Augmented and Virtual Reality for education, and Artificial Intelligence in Education. He has contributed extensively to these research fields, authoring over 100 publications in peer-reviewed journals and international conference proceedings and as Co-Editor of the diamond open access Italian Journal of Educational Technology (IJET). Moreover, he has coordinated two EU funded projects in the field of Data Literacy: DATALIT (Data Literacy at the interface of higher education and business), and DEDALUS (DEveloping DAta Literacy courses for University Students). Currently, he is leading two EU funded project named SMERALD (SMEs – Raising Awareness and Learning on Digital data, data analysis and artificial intelligence) and IDEAL (Integrating Data Analysis and AI in Learning experiences) concerning Artificial Intelligence in Education with a specific focus on AI literacy. He is also a contract professor of Open Data Management at the University of Palermo.