Project Evolution-aware Prompting of LLMs for Just-in-time Defect Prediction in Edge-cloud Systems

Inseok Yeo[1], Sungu Lee[1], Duksan Ryu[2] and Jongmoon Baik[1,*]

[1]*Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea*
[2]*Jeonbuk National University, Jeonju, Republic of Korea*
E-mail: yinseok38@kaist.ac.kr; sungu0027@kaist.ac.kr; jbaik@kaist.ac.kr; duksan.ryu@jbnu.ac.kr
[*]*Corresponding Author*

Received 29 October 2025; Accepted 15 December 2025

Abstract

Edge-cloud systems, which bring computing, storage, and networking resources closer to end-users, offer significant advantages in reducing latency and enabling real-time data processing. These systems are increasingly deployed across diverse domains, such as smart manufacturing, autonomous vehicles, and large-scale IoT networks, to support big data-driven services that require continuous analytics and rapid response. Ensuring software reliability in these environments is critical, which has led to growing attention on just-in-time (JIT) defect prediction as an effective technique for prioritizing testing efforts by identifying code changes likely to introduce defects. However, existing techniques struggle to perform accurately on new or low-data projects due to insufficient training data.

In this paper, we propose PROPER-SDP, a prompt-based approach that leverages large language models. By incorporating project evolution data directly into prompts, our approach enables LLMs to effectively capture the contextual information essential for accurate JIT defect prediction. By doing so, we effectively address the cold-start problem, allowing accurate JIT defect prediction even in the absence of project-specific training data. Evaluation

Journal of Web Engineering, Vol. 25_3, 395–416.
doi: 10.13052/jwe1540-9589.2535

results demonstrate that our method significantly improves prediction performance, surpassing baseline methods by an average of 19.7% in F1-score. Our approach enables reliable JIT defect prediction even in rapidly evolving, resource-constrained edge-cloud systems.

Keywords: Just-in-time defect prediction, large language model, edge-cloud system.

1 Introduction

An edge-cloud system is a distributed computing architecture designed to bring cloud resources closer to end-users and devices by deploying computing, storage, and networking capabilities at the edge of the network. Unlike traditional cloud computing, which relies on centralized data centers, edge-cloud systems process data locally, significantly reducing latency [19].

With the rapid proliferation of data-intensive applications, such as real-time video analytics, smart manufacturing, autonomous driving, and large-scale IoT monitoring, edge-cloud infrastructures have become a key enabler of big data-driven services, allowing massive volumes of heterogeneous data to be processed and analyzed near their sources. This growing reliance on edge-cloud systems for critical, data-centric services has, in turn, motivated extensive research on ensuring their software reliability [8, 13, 20]. Among these, software defect prediction has emerged as a key technique to optimize testing strategies, enabling developers to prioritize the testing on defect-prone software components, ultimately reducing the cost of ensuring software reliability in edge-cloud environments.

Software defect prediction (SDP) is a crucial aspect of software engineering that aims to identify defective code components before deployment, improving software quality and reducing maintenance costs [1, 5]. Software defect prediction models predict potential defective software components [7]. The primary goal of SDP is to assist developers in prioritizing testing efforts, allocating resources efficiently, and mitigating risks associated with software failures [9]. Given the increasing complexity of modern software systems, accurate defect prediction models have gained significant attention, with researchers exploring various features, classifiers, and evaluation metrics to enhance predictive performance [14].

Just-in-time (JIT) defect prediction is the task of predicting whether a code change (e.g., a commit or pull request) will introduce a defect, so that potential bugs can be caught early, just in time before integration [27]. Unlike

traditional file-level or module-level defect prediction, JIT defect prediction operates at the change level, allowing developers to focus code review and testing efforts on the riskiest changes [6,27]. The advantage of JIT prediction has been highlighted in prior studies; by inspecting only a small fraction of commits (e.g., the top 20% most risky changes), it is possible to capture a significant portion (35%) of defects [10], enhancing the effectiveness of defect prediction.

Due to its advantages, JIT defect prediction has been applied in edge-cloud environments, where software is deployed across distributed edge devices and cloud backends. Testing resources and time are limited in edge-cloud environments [13], making it crucial to prioritize problematic commits. By directing scarce testing efforts to the most error-prone changes, JIT defect prediction significantly improves software reliability without requiring exhaustive testing of every modification.

However, building accurate JIT defect predictors for edge-cloud systems presents significant challenges. Training a defect prediction model requires a large amount of error data, but in real-world scenarios, machine learning models often struggle due to insufficient labeled defect data [25]. This challenge is even more pronounced in edge-cloud software projects, which are frequently new or rapidly evolving, making it difficult to collect enough labeled commit data for effective training.

To address the data scarcity problem, cross-project defect prediction (CPDP) was proposed, where a model trained on one set of projects is applied to a new project [17]. However, CPDP often suffers from poor performance due to dataset shifts [10, 16]. Recent research on defect prediction in edge-cloud systems further highlights these challenges: while machine learning models perform well on the projects they were trained on, their effectiveness degrades significantly in cross-project scenarios [13]. In summary, existing JIT defect prediction methods for edge-cloud systems face two major limitations: they require impractically large labeled datasets for each project, and they struggle to maintain accuracy when applied across different projects.

In our previous work [23], we introduced PROPER-SDP, a prompt-based approach that leverages a project-evolution context to perform JIT defect prediction in edge-cloud systems. While the initial results were promising against CPDP baselines, the study had several limitations: the evaluation considered a limited set of LLMs and a smaller test set; comparisons did not include within-project (WPDP) settings, which constrained the validity and extent of the findings; and the prompt design conveyed only a partial view of the SDP objective, under-expressing rich project context and thereby limiting

the model's ability to reason about change-level risk in dynamic edge-cloud environments.

This paper extends the previous study by addressing its key limitations. We expand the experimental scope by incorporating 41% more test data and large language models (LLMs), introduce an extra research question by including WPDP baselines alongside CPDP, and perform statistical analyses to strengthen the validity and generalizability of our findings. Finally, we redesign the prompt to deliver more direct and information-dense context about the SDP objective for better guidance. PROPER-SDP enhances the LLM's predictions by incorporating the project's documentation and evolution data as contextual information concatenated to the defect prediction prompt. This enriched context allows the LLM to accurately capture project-specific nuances and evolution patterns, which are crucial for effective just-in-time defect prediction in software projects it has not been explicitly trained on. As a result, PROPER-SDP enables high-performance JIT defect prediction for edge-cloud systems, outperforming baseline models by an average of 19.7% in F1-score, without requiring any training data.

2 Background

2.1 Edge-cloud System and Go Language

In recent years, the adoption of edge-cloud architectures has become increasingly prevalent in software systems requiring low latency, scalability, and efficient resource utilization [19]. Edge-cloud systems combine centralized cloud infrastructure with decentralized edge nodes, enabling data processing closer to the source while leveraging the computational power and storage of the cloud. This hybrid model not only supports applications such as real-time analytics, IoT, and autonomous systems but also serves as a foundation for big data-driven services, allowing massive and diverse datasets to be processed efficiently near their sources.

Within this architectural shift, programming languages that support concurrency, performance, and portability have gained attention. The Go programming language, developed by Google, has emerged as a strong candidate for building edge-cloud services due to its lightweight runtime, efficient memory usage, and native support for concurrency. Go's simplicity and performance make it well-suited for implementing microservices, APIs, and lightweight daemons that can run reliably in both cloud servers and edge devices. Due to its popularity, Go is the main focus of this paper.

2.2 Software Defect Prediction and Just-in-time Defect Prediction

The goal of software defect prediction (SDP) is to aid developers in finding defects without testing. Traditional SDP techniques aim to identify defect-prone modules (files, classes, or functions) using static code metrics and historical defect data. These models are typically trained offline and used periodically during development to guide testing and code review efforts. While useful, conventional SDP approaches often operate at coarse granularity and may struggle to provide timely insights during fast-paced development cycles.

In response to these limitations, researchers have introduced JIT defect prediction techniques, which aim to predict whether certain software change is likely to introduce defects. JIT SDP techniques leverage fine-grained features such as code churn, change history, developer activity, and social metrics, related to target software change, allowing for predictions to be made immediately when changes are submitted. This enables more targeted reviews and resource allocation at the commit level.

2.3 Large Language Models

Large language models (LLMs) are transformer-based deep neural networks pre-trained on vast corpora of natural language and source code. Their primary strength lies in learning generalized representations that enable zero-shot or few-shot performance on downstream tasks without requiring task-specific training [15,26]. Unlike traditional deep learning based methods that depend on large labeled datasets and fine-tuning, LLMs can make predictions through prompt-based interactions, significantly lowering the barrier for practical deployment in data-scarce environments.

Recent work has explored various ways to enhance LLM performance in software engineering tasks such as defect prediction, fault localization, and code summarization [4]. These include integrating external tools or providing additional context [11, 24]. Such approaches have shown effectiveness in guiding LLMs toward more accurate and context-aware predictions across diverse downstream tasks.

3 Related Works

Kwon et al. [13] conducted one of the earliest studies on applying just-in-time software defect prediction to edge-cloud systems by leveraging

pre-trained deep learning models. In their work, they collected a large-scale dataset from GitHub using a GitHub Pull Request (GHPR)-based method, which enables automated identification and labeling of defective and clean code at the function level. They focused on three popular transformer-based models–CodeBERT, GraphCodeBERT, and UniXcoder–evaluating their predictive performance under both within-project defect prediction (WPDP) and cross-project defect prediction (CPDP) settings.

Their results showed that UniXcoder generally performed best in the WPDP scenario, thanks to its abstract syntax tree-based representation learning. However, they also highlighted a key limitation: all three models exhibited poor generalization in CPDP scenarios, largely due to project-specific code characteristics and data distribution shifts, which hindered cross-project transferability.

Following this, Sharma et al. [2] highlighted the importance of addressing evolving datasets in defect prediction, where software is frequently updated. Although their study did not focus on edge-cloud systems, this challenge is relevant, due to environment that often involve frequent deployment, server reboots, etc. [21] They proposed a dynamic fine-tuning strategy that allows large language models to adapt to new versions while retaining prior knowledge using elastic weight consolidation (EWC) and memory replay. This continual learning approach improves model stability over time but still requires repeated fine-tuning and high computational cost, making it less practical for systems that demand lightweight, fast, and scalable defect prediction solutions.

Building upon this foundation, our work proposes PROPER-SDP, a novel LLM-based approach that addresses the data scarcity and cross-project generalization challenges. Instead of relying on model fine-tuning, we employ a prompt-based strategy using large language models (LLMs) enhanced with project evolution data such as README modifications and file structure changes. This approach enables effective zero-shot JIT defect prediction and reduces reliance on labeled training data, offering a practical and scalable solution for dynamic edge-cloud environments.

4 Methodology

In this section, we present PROPER-SDP. It leverages the predictive power of large language models incorporated with valuable project evolution data to assess software changes and identify potential defects at an early stage. We first outline the data preparation process that captures project evolution data,

Table 1 Characteristics of subject projects

Name	Stars	Forks	PRs	Short Description
EdgeX-go(EdgeX) [3]	1.1K	0.4K	2.3K	An open-source framework providing a unified platform for building and deploying IoT solutions.
Kubeedge(Kube) [12]	5.5K	1.4K	2.6K	A community-driven project that extends Kubernetes capabilities to edge environments.
Openshift [18]	1.3K	1.2K	5.3K	A Red Hat-managed application platform supporting deployment and orchestration across hybrid, multi-cloud, and edge environments.
Traefik [22]	40.5K	4.4K	4.2K	A cloud-native reverse proxy and load balancer designed for modern infrastructures, integrating smoothly with Docker and Kubernetes.

followed by a detailed overview of proposed approach. The detailed overall approach is illustrated in Figure 1.

4.1 Data Preparation

Given a JIT defect dataset initially collected through GitHub Pull Requests (GHPR), we conducted an additional data preparation phase aimed at effectively integrating project evolution context into the large language model (LLM). To accurately capture this evolutionary context, we specifically identified and analyzed the key differences between the current and previous software versions associated with each pull request.

We defined project evolution context using three criteria: changes in the main project README files, modifications to local README files, and adjustments to the local file structure. These were chosen because they are the key representative indicators of meaningful changes in project. The main README often reflects high-level updates to the system's functionality or architecture. Local README changes capture information closer to the target function, such as usage details or implementation notes, offering fine-grained context. File structure changes, including added or removed files in the same directory, may signal refactoring or feature updates that affect code behavior. Together, these elements provide both global and local context to support more accurate defect prediction.

Using these criteria, we filtered the GHPR dataset to include only entries that showed at least one type of evolutionary change. Compared to our

Algorithm 1 Generate JIT Dataset

```
function GENERATE_JIT_DATASET
    jit_commits ← collect_jit_commits(ghpr_data)
    for each commit in jit_commits do
        prev_commit ← get_previous_commit(commit)
        cur_readme ← get_main_readme(commit)
        prev_readme ← get_main_readme(prev_commit)
        l_readme_cur ← get_local_readme(commit)
        l_readme_prev ← get_local_readme(prev_commit)
        fs_current ← get_file_structure(commit)
        fs_prev ← get_file_structure(prev_commit)
        main_readme_change ← diff(prev_readme, cur_readme)
        l_readme_change ← diff(l_readme_prev, l_readme_cur)
        fs_change ← diff(fs_prev, fs_current)


         ###Checking if change data exists
        if main_readme_change or
          local_readme_change or
          file_structure_change then
            changes ← (
                main_readme_change,
                local_readme_change,
                file_structure_change)
            Add (commit, changes) to enriched_dataset
        end if
    end for


     ###Data balancing
    while size(buggy_data) < size(clean_data) do
        x_aug ← augment(x) for some (x, y) ∈ buggy_data
        Add (x_aug, buggy) to buggy_data
    end while
    return enriched_dataset
end function
```

Figure 1 Overall approach of PROPER-SDP.

previous work [23], which relied on elimination-based filtering, we additionally applied data augmentation to address class imbalance and improve the generalization of the results. The statistics of the resulting test dataset are summarized in Table 2.

To systematically and efficiently assemble this enriched dataset, we leveraged the RESTful GitHub API, automating the identification and extraction of project evolution data. Consequently, the final dataset is enriched

Table 2 Comparison of number of bugs in original and new datasets

Project	Original	New
EdgeX	1148	1078
Kube	1139	1376
Openshift	5541	5545
Traefik	1080	976

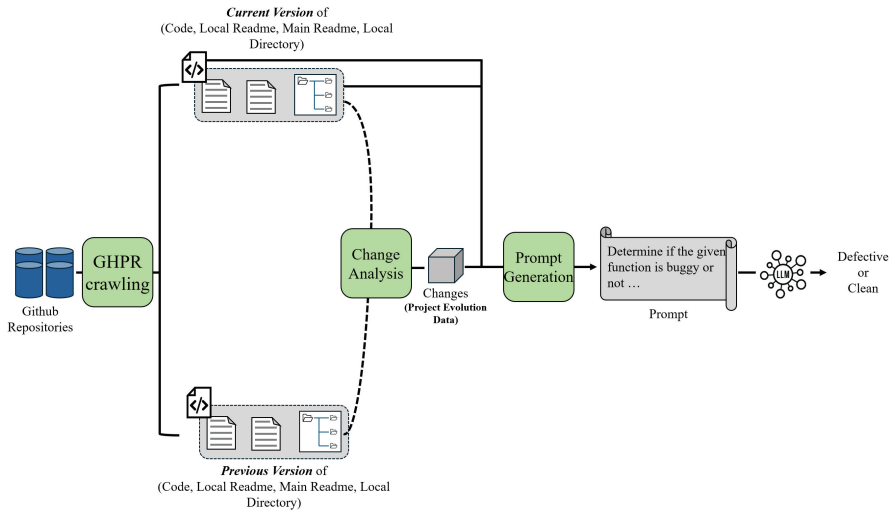

Figure 2 Dataset collection algorithm.

with comprehensive contextual details, including the target function's name and code, pull request messages, main and local README modifications, and specific changes to the local file structure. This structured, context-rich dataset significantly enhances the capability of the LLM in accurately predicting defect-related changes within edge-cloud system projects.

The detailed algorithm used for data preparation is provided in Figure 2. It begins by collecting JIT commits from the GHPR dataset (lines 1–2) and iteratively processes each commit (line 3) to retrieve both the current and previous versions of the corresponding files. From lines 4–13, the algorithm extracts three primary sources of evolution data: the main README, local README, and local file structure. These are compared using the diff() function to identify meaningful modifications between the two versions. Lines 16–18 check whether at least one of these change types exists; if so, the corresponding commit and its contextual data are added to the enriched dataset. Finally, from lines 23–27, a data balancing stage is performed, where

additional buggy samples are generated through augmentation until the ratio of buggy and clean instances becomes balanced. This two-phase process–context extraction followed by class balancing–ensures that the resulting dataset captures both structural evolution and label distribution, thereby improving the reliability and representativeness of the JIT defect prediction dataset.

4.2 Overview of PROPER-SDP

Following the data preparation phase, we conduct defect prediction by integrating both static code attributes and evolutionary project context to improve predictive accuracy. The defect prediction process consists of two stages: change analysis and prompt generation. In the change analysis stage, we extract information from the target commit that contains the function to be classified, as well as from its immediately preceding commit. This allows us to capture both the current state and recent changes to the code. In the prompt generation stage, we construct a carefully organized prompt that conveys the most relevant information–highlighting both code differences and contextual evolution–to ensure the LLM receives the appropriate signals and nuance for accurate defect prediction.

4.2.1 Change analysis

The first stage of PROPER-SDP involves analyzing the change between commits to extract all relevant information for defect prediction. Specifically, we identify the target commit, which includes the function whose defect status needs to be predicted, as well as its immediate preceding commit. From these two snapshots, we extract the function's source code before and after the change, its name, and its location in the project. We also gather metadata from the associated pull request, such as the title and description, which often reflect the developer's intent or rationale behind the change.

To provide additional context about how the project is evolving, we incorporate project evolution data across three dimensions: (1) modifications to the main project-level README file, (2) changes to local README files located in the same directory as the target function, and (3) structural changes in the local file system, such as added or removed files. For file structure changes, we list file names without including the full source code to keep the input compact. For README files, if no change is detected, we explicitly indicate this; otherwise, we include both the full "before" and "after" versions rather than line-by-line diffs, allowing the model to capture broader semantic changes.

4.2.2 Prompt generation

The prompt generation strategy is illustrated in Figure 3. Once the change-related and contextual data have been collected, we proceed to the prompt

”-Objective: Determine if the given function reflects the software state before the pull request(Buggy) or After the pull request (Non-buggy)

-Provided Information:
- ****Project****:
{projects [index]}
- ****Function Name****:
{function_name}
- ****Source Code****:
{source_code}
- ****PR Message****:
{PR_message}
- ****PR Body****:
{PR_body}
- ****README (Current)****:
{cur_main_readme}
- ****README (Previous)****:
`{'[No change]' if cur_main_readme == prev_main_readme else prev_main_readme}`
- ****Deleted Files****:
`{' , '.join(deleted_files) if deleted_files else 'None'}`
- ****Added Files****:
`{' , '.join(added_files) if added_files else 'None'}`
- ****Local README (Current)****:
{cur_readme if cur_readme else 'None'}
- ****Local README (Previous)****:
`{'[No change]' if cur_readme == prev_readme else prev_readme}`

Use the provided project evolution context — including updates to the README and file structure — to understand how recent changes might have introduced potential defects. Consider whether the modification aligns with the project's overall design intent and stability. Use this reasoning to decide if the change is bug-inducing or non-bug-inducing.

-Task:

Predict the function's state relative to the PR:
Now, let's begin."

Figure 3 Example prompt for PROPER-SDP.

generation stage. In this stage, the extracted information is organized into a structured, natural-language prompt designed to emphasize both static code features and the evolution context surrounding the target function. Each component, such as the pull request title, function path, before/after code, README changes, and file structure updates, is clearly labeled to help the LLM interpret the role and relevance of each element.

To conserve input space and focus on meaningful content, the main project-level README and local README files from the previous version are included only if changes are detected compared to the current version. If no differences are found, a brief note indicating "no change" is provided instead of repeating unchanged content.

In our previous work [23], the prompt offered only a partial description of the SDP objective, limiting the model's ability to fully capture how project evolution affects defect risk. In this study, we redesigned the prompt to be more explicit and information-dense, guiding the LLM to reason directly about whether a given change is likely to introduce a defect. This enhancement provides clearer task framing and richer contextual grounding, contributing to the improved predictive performance observed in this work.

This comprehensive prompt is then passed to the LLM, which performs a binary classification–predicting whether the target function is buggy (pre-change) or clean (post-change). The resulting predictions are stored and evaluated using standard classification metrics, primarily the F1-score.

5 Experimental Setup

5.1 Research Questions

We set the following research questions:

- **RQ1:** What is the performance of PROPER-SDP compared to CPDP baselines?
- **RQ2:** What is the performance of PROPER-SDP compared to WPDP baselines?
- **RQ3:** What is the optimal LLM model for PROPER-SDP?
- **RQ4:** What is the effect of different model design choices on PROPER-SDP?

5.2 Dataset

We utilized the GHPR edge-cloud defect dataset introduced by Kwon et al. [13]. The final dataset encompasses data from four open-source

edge-cloud projects, whose characteristics are detailed in Table 1. From this dataset, we specifically selected entries containing at least one type of evolutionary context information. Compared to our previous work that relied on data filtering, we applied data augmentation to address class imbalance problem. Detailed statistics of the resulting dataset are presented in Table 2. The final dataset encompasses data from four open-source edge-cloud projects.

5.3 Used Models and Evaluation Metrics

To determine the most optimal model for JIT defect prediction in edge-cloud systems, we evaluated multiple state-of-the-art large language models (LLMs). Specifically, we compared the performance of GPT-3.5, GPT-4o-Mini, Gemini-2-Flash, and Gemini-2-Flash-lite, which are widely utilized in various natural language processing tasks. Compared to our previous work, we additionally used Gemini-2-Flash-lite to evaluate the generalizability of PROPER-SDP. The experiments were conducted using their respective APIs to ensure a standardized evaluation framework.

For performance assessment, we primarily employed the F1-score, a widely recognized metric in defect prediction. Unlike learning-based models that may output probabilistic or multi-class predictions, PROPER-SDP strictly adheres to binary classification. The F1-score is particularly suitable in this context as it balances precision and recall, making it an effective metric for evaluating defect prediction accuracy.

5.4 Baselines

We compared PROPER-SDP with the fine-tuned, learning-based defect prediction method proposed by Kwon et al. [13], hereafter referred to as FT-SDP. To effectively evaluate defect prediction performance in edge-cloud systems with limited training data, we compared our results with the cross-project defect prediction outcomes from that study.

The previous work evaluated defect prediction performance using three pre-trained models: CodeBERT, GraphCodeBERT, and UnixCoder. Each model was trained on three different source projects and tested on four target projects. The authors noted that they excluded Traefik as a source project due to differences such as project length and structural variations. However, for a fair evaluation of defect prediction in scenarios lacking test data, we included Traefik as a source project, training it on the CodeBERT model.

To simplify comparisons, we presented the previous work's results using both its best-performing scenario and its average performance. This

approach ensures a more comprehensive evaluation of our defect prediction methodology.

6 Results

6.1 RQ1: JIT-SDP Performance of Proposed Approach Compared to CPDP Baselines

To evaluate the effectiveness of PROPER-SDP, we compared its defect prediction performance with FT-SDP's cross-project defect prediction (CPDP) result including the best-performing result and the average performance of various models. Table 3 presents the F1-scores for four edge-cloud projects: EdgeX, Kube, Openshift, and Traefik.

Table 3 F1 score comparison of PROPER-SDP and CPDP baselines

	PROPER-SDP	FT-SDP(CPDP-best)	FT-SDP(CPDP-average)
EdgeX	**0.701**	0.667	0.586
Kube	**0.7**	0.688	0.583
Openshift	**0.736**	0.666	0.594
Traefik	**0.719**	0.667	0.623

Across the evaluated projects, PROPER-SDP consistently achieved higher F1-scores than FT-SDP (CPDP) on average and outperformed its best-performing configuration in three out of four cases. Specifically, for the EdgeX, Kube, and Openshift projects, our approach surpassed the best FT-SDP (CPDP) results, while achieving comparable performance on the Traefik project. Furthermore, PROPER-SDP significantly exceeded the FT-SDP (CPDP) average across all projects, improving the F1-score by approximately 19.7% on average and demonstrating the effectiveness of prompt-based LLM defect prediction in data-scarce edge-cloud environments.

6.2 RQ2: JIT-SDP Performance of Proposed Approach Compared to WPDP Baselines

As shown in Table 4, we compared the performance of PROPER-SDP against FT-SDP (WPDP) results. PROPER-SDP slightly outperformed the FT-SDP (WPDP) average and achieved comparable results to its best-performing configuration, recording the highest F1-scores in two out of four projects. Notably, unlike FT-SDP (WPDP), which requires training on the same project, PROPER-SDP performs prediction without any in-project training data–demonstrating its competitiveness in data-scarce environments.

Table 4 F1-score comparison of PROPER-SDP and WPDP baselines

	PROPER-SDP	FT-SDP(WPDP-best)	FT-SDP(WPDP-average)
EdgeX	**0.701**	**0.702**	0.686
Kube	0.7	**0.79**	0.745
Openshift	0.736	**0.831**	0.825
Traefik	**0.719**	0.67	0.645

Table 5 F-test comparison of variances between PROPER-SDP and WPDP baselines

Comparison	F-value	p-value
PROPER-SDP vs WPDP(best)	19.28	0.037
PROPER-SDP vs WPDP(average)	20.96	0.033

Furthermore, Table 5 presents the results of an F-test analysis comparing performance variance. PROPER-SDP demonstrated significantly lower variance than both WPDP (best) and WPDP (average), with F-values of 19.28 and 20.96, and p-values below 0.05. This indicates that PROPER-SDP not only provides comparable predictive performance but does so more stably, reinforcing its robustness even without access to project-specific training data.

6.3 RQ3: JIT-SDP Performance of Different LLM Models

To identify the most effective large language model (LLM) for JIT defect prediction, we evaluated four commonly used LLMs: `gpt-3.5-turbo`, `gemini-2.0-flash`, `gpt-4o-mini`, and `gemini 2.0 flash-lite`. Table 6 presents the F1-scores for each model across the four target edge-cloud projects.

Table 6 F1-score comparison between different LLM models

	gpt-3.5-turbo	gemini 2.0 flash	gpt-4o-mini	gemini 2.0 flash-lite
EdgeX	**0.701**	0.545	0.648	0.619
Kube	**0.700**	0.566	0.552	0.381
Openshift	**0.736**	0.439	0.486	0.399
Traefik	**0.719**	0.530	0.589	0.516

Among the evaluated models, `gpt-3.5-turbo` consistently achieved the highest F1-scores across all projects, demonstrating superior capability in understanding code changes and contextual project evolution. The performance gap between `gpt-3.5-turbo` and the other models was especially prominent in the Openshift and Kube projects, where its F1-scores surpassed the others by more than 0.1 in some cases.

In addition to predictive accuracy, we also compared the inference cost and latency of each model. Table 7 summarizes the average monetary cost per thousand predictions in dollars and time spent per prediction in seconds.

Table 7 Cost comparison between different LLM models

Model	Money ($)	Time (s)
gpt-3.5-turbo	1.52	1.01
gemini-2.0-flash	0.33	1.13
gpt-4o-mini	0.37	0.87
gemini 2.0 flash-lite	0.30	1.12

The results show that `gpt-3.5-turbo` achieved the highest prediction accuracy but also required the most resources–it was both the most expensive and one of the slowest models. In contrast, `gpt-4o-mini` provided the best balance between cost and speed, offering faster responses at a lower price. While `gpt-3.5-turbo` consistently produced the highest F1-scores across all projects, its high computational cost may limit its use in cost-sensitive environments. The two Gemini-based models were the cheapest options, but their predictive performance was the weakest.

Overall, when accuracy is the top priority, `gpt-3.5-turbo` remains the most effective choice. However, `gpt-4o-mini` serves as a practical and cost-efficient alternative in scenarios where faster inference and lower cost are more important than maximum accuracy.

6.4 RQ4: Impact of Project Evolution Context Components

To understand the impact of different project evolution context components on performance, we performed an ablation study by removing one component at a time: main README changes, local README changes, and file structure changes. The results can be seen in Table 8, with the best-performing F1-scores shown in bold and the second-best F1-scores underlined.

Table 8 Ablation study (F1-score)

	Original	No local RM	No file structure	No main RM
EdgeX	**0.701**	<u>0.656</u>	0.649	0.652
Kube	**0.7**	<u>0.667</u>	0.604	0.612
Openshift	**0.736**	0.641	<u>0.666</u>	0.649
Traefik	**0.719**	0.526	0.595	<u>0.656</u>

The local README changes had the most significant impact on model performance. This suggests that local README files, which are typically

situated closest to the target function, provide the most directly relevant context for defect prediction, making them a critical source of information. In contrast, changes to the main README had the smallest impact. This may be because the main README is often located farther from the specific code being modified, tends to contain high-level or general project information, and does not change frequently–reducing its value in predicting function-level defects.

These results suggest that combining both global (main README) and local (local README and file structure) context is important for effective LLM-based JIT defect prediction in edge-cloud systems.

7 Threats to Validity

7.1 Internal validity

The internal threat to validity is potential bias in the project evolution data extraction process. PROPER-SDP relies on automated retrieval of project-specific context, including README changes and file structure modifications. However, incomplete or erroneous extraction could introduce inconsistencies, affecting the model's defect prediction performance. We mitigated this threat by expanding the dataset, incorporating additional LLMs, and conducting more experiments to improve validity and generalizability.

7.2 External validity

The LLMs used in our study are pre-trained on publicly available code, and their effectiveness in predicting defects for private or domain-specific projects, which may follow different coding standards, has not been assessed. This threat was mitigated by expanding the dataset, employing multiple LLMs, and conducting additional experiments to enhance the validity and generalizability of the results. Future studies should evaluate the generalizability of our method in broader software development contexts.

7.3 Construct validity

We compared our method against baseline models trained with cross-project learning as differences in training data distributions may affect the fairness of comparisons. Further validation using industry benchmarks and real-world defect reports would provide a more comprehensive assessment of the model's effectiveness in just-in-time defect prediction in edge-cloud systems.

8 Conclusion

In this paper, we proposed a novel prompt-based approach to just-in-time (JIT) defect prediction for edge-cloud systems using large language models. Our PROPER-SDP incorporates project-specific evolutionary context, such as changes to README files and file structures, into the input prompts, enabling accurate defect prediction without requiring extensive labeled datasets or model fine-tuning. Experimental results across four real-world edge-cloud projects demonstrate that PROPER-SDP consistently outperforms traditional cross-project learning methods in predictive accuracy, particularly when using GPT-3.5-turbo, and have comparable result to within-project defect prediction model. Furthermore, our ablation study highlights the importance of contextual information, especially local README changes, in improving model performance. This study suggests that prompt-based LLMs provide a scalable and adaptable solution for enhancing software reliability in dynamic, data-scarce edge-cloud environments. Future work will explore the integration of domain-specific knowledge and the applicability of this approach in industrial and proprietary settings.

Acknowledgements

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2026-RS-2020-II201795, 50%) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation) and the Institute of Information & Communications Technology Planning & Evaluation (IITP)-Innovative Human Resource Development for Local Intellectualization program grant funded by the Korea government (MSIT) (IITP-2026-RS-2024-00439292, 50%).

References

[1] Akimova, E.N., Bersenev, A.Y., Deikov, A.A., Kobylkin, K.S., Konygin, A.V., Mezentsev, I.P., Misilov, V.E.: A survey on software defect prediction using deep learning. Mathematics **9**(11), 1180 (2021).

[2] Bhutamapuram, U.S., Chonari, F., K Anilkumar, G., Konchada, S.K.: Llms for defect prediction in evolving datasets: Emerging results and future directions. In: Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering. p. 520–524. FSE

Companion '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3696630.3728491

[3] EdgeX Foundry: edgex-go: EdgeX Foundry Go Services. https://github.com/edgexfoundry/edgex-go (2025).

[4] Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., Zhang, J.M.: Large language models for software engineering: Survey and open problems. In: 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE). pp. 31–53 (2023). https://doi.org/10.1109/ICSE-FoSE59343.2023.00008

[5] Giray, G., Bennin, K.E., Ömer Köksal, Önder Babur, Tekinerdogan, B.: On the use of deep learning in software defect prediction (2022). https://arxiv.org/abs/2210.02236

[6] Guo, Y., Gao, X., Jiang, B.: An empirical study on jit defect prediction based on bert-style model. arXiv preprint arXiv:2403.11158 (2024).

[7] Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.: A systematic literature review on fault prediction performance in software engineering. IEEE Transactions on Software Engineering **38**(6), 1276–1304 (2011).

[8] Hong, H., Lee, S., Ryu, D., Baik, J.: Enhancing software defect prediction in ansible scripts using code-smell-guided prompting with large language models in edge-cloud infrastructures. In: International Conference on Web Engineering. pp. 30–42. Springer (2024).

[9] Hosseini, S., Turhan, B., Gunarathna, D.: A systematic literature review and meta-analysis on cross project defect prediction. IEEE Transactions on Software Engineering **45**(2), 111–147 (2017).

[10] Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., Hassan, A.E.: Studying just-in-time defect prediction using cross-project models. Empirical Software Engineering **21**, 2072–2106 (2016).

[11] Kang, S., An, G., Yoo, S.: A quantitative and qualitative evaluation of llm-based explainable fault localization. Proc. ACM Softw. Eng. **1**(FSE) (Jul 2024). https://doi.org/10.1145/3660771

[12] KubeEdge Authors: KubeEdge: Kubernetes Native Edge Computing Framework. https://github.com/kubeedge/kubeedge (2025).

[13] Kwon, S., Lee, S., Ryu, D., Baik, J.: Pre-trained model-based software defect prediction for edge-cloud systems. Journal of Web Engineering **22**(2), 255–278 (2023).

[14] Malhotra, R.: A systematic review of machine learning techniques for software fault prediction. Applied Soft Computing **27**, 504–518 (2015).

[15] Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., Gao, J.: Large language models: A survey (2025). https://arxiv.org/abs/2402.06196
[16] Nam, J., Pan, S.J., Kim, S.: Transfer defect learning. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 382–391 (2013). https://doi.org/10.1109/ICSE.2013.6606584
[17] Pal, S., Sillitti, A.: Cross-project defect prediction: a literature review. IEEE access **10**, 118697–118717 (2022).
[18] Red Hat, Inc.: OpenShift Installer. https://github.com/openshift/installer (2025).
[19] Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: Vision and challenges. IEEE internet of things journal **3**(5), 637–646 (2016).
[20] Soualhia, M., Fu, C., Khomh, F.: Infrastructure fault detection and prediction in edge cloud environments. In: Proceedings of the 4th ACM/IEEE Symposium on Edge Computing. pp. 222–235 (2019).
[21] Souza, P.S., Ferreto, T.C., Rossi, F.D., Calheiros, R.N.: Location-aware maintenance strategies for edge computing infrastructures. IEEE Communications Letters **26**(4), 848–852 (2022). https://doi.org/10.1109/LCOMM.2022.3150243
[22] Traefik Labs: Traefik: The Cloud Native Application Proxy. https://github.com/traefik/traefik (2025).
[23] Yeo, I., Lee, s., Ryu, D., Baik, J.: Proper-sdp: Prompt-based project evolution-aware software defect prediction for edge-cloud systems. The 5th International Workshop on Big data driven Edge Cloud Services (BECS 2025) Co-located with the 25th International Conference on Web Engineering (ICWE 2025), June 30-July 3, 2025, Delft, Netherlands.
[24] Yeo, I., Ryu, D., Baik, J.: Improving llm-based fault localization with external memory and project context. arXiv preprint arXiv:2506.03585 (2025).
[25] Z. Wan, X. Xia, A.E.H.D.L.J.Y., Yang, X.: Perceptions, expectations, and challenges in defect prediction (2020).
[26] Zhao, W.X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J.Y., Wen, J.R.: A survey of large language models (2025), https://arxiv.org/abs/2303.18223
[27] Zhao, Y., Damevski, K., Chen, H.: A systematic survey of just-in-time software defect prediction. ACM Computing Surveys **55**(10), 1–35 (2023).

Biographies

Inseok Yeo received his bachelor's degree in computer science from Hanyang University in 2024. He is a master's student in computer science at KAIST. His research areas include software analytics, software engineering based on AI and LLMs.

Sungu Lee received his bachelor's degree in mathematics from KAIST in 2021 and his master's degree in software engineering from KAIST in 2022. He is a doctoral student in software engineering at KAIST. His research areas include software analytics based on AI, software defect prediction, mining software repositories, and software reliability engineering.

Duksan Ryu earned his bachelor's degree in computer science from Hanyang University in 1999 and his master's dual degree in software engineering from KAIST and Carnegie Mellon University in 2012. He received his Ph.D. degree from the school of computing at KAIST in 2016. His research areas include software analytics based on AI, software defect prediction, mining software repositories, and software reliability engineering. He is currently an associate professor in software engineering department at Jeonbuk National University.

Jongmoon Baik received his B.Sc. degree in computer science and statistics from Chosun University in 1993. He received his M.Sc. degree and Ph.D. degree in computer science from University of Southern California in 1996 and 2000 respectively. He worked as a principal research scientist at Software and Systems Engineering Research Laboratory, Motorola Labs, where he was responsible for leading many software quality improvement initiatives. His research activity and interests are focused on software six sigma, software reliability and safety, and software process improvement. Currently, he is a full professor in the school of computing at Korea Advanced Institute of Science and Technology (KAIST). He is a member of the IEEE.