An LLM-Based Agent-Oriented Approach for Automated Code Design Issue Localization

Fraol Batole
Computer Science Department
Tulane University
New Orleans, LA, USA
fbatole@tulane.edu

David OBrien
Computer Science Department
Iowa State University
Ames, IA, USA
davidob@iastate.edu

Tien N. Nguyen
Computer Science Department
The University of Texas at Dallas
Dallas, TX, USA
tien.n.nguyen@utdallas.edu

Robert Dyer
Computer Science Department
University of Nebraska-Lincoln
Lincoln, NE,USA
rdyer@unl.edu

Hridesh Rajan
Computer Science Department
Tulane University
New Orleans, LA, USA
hrajan@tulane.edu

***Abstract*—Maintaining software design quality is crucial for the long-term maintainability and evolution of systems. However, design issues such as poor modularity and excessive complexity often emerge as codebases grow. Developers rely on external tools, such as program analysis techniques, to identify such issues. This work leverages Large Language Models (LLMs) to develop an automated approach for analyzing and localizing design issues.**

Large language models have demonstrated significant performance on coding tasks, but directly leveraging them for design issue localization is challenging. Large codebases exceed typical LLM context windows, and program analysis tool outputs in non-textual modalities (e.g., graphs or interactive visualizations) are incompatible with LLMs' natural language inputs.

To address these challenges, we propose LOCALIZEAGENT, a novel multi-agent framework for effective design issue localization. LOCALIZEAGENT integrates the specialized agents that (1) analyze code to identify potential code design issues, (2) transform program analysis outputs into abstraction-aware LLM-friendly natural language summaries, (3) generate context-aware prompts tailored to specific refactoring types, and (4) leverage LLMs to locate and rank the localized issues based on their relevance.

Our evaluation using diverse real-world codebases demonstrates significant improvements over the baseline approaches, with LOCALIZEAGENT achieving 138%, 166%, and 206% relative improvements in exact-match accuracy for localizing information hiding, complexity, and modularity issues, respectively.

***Index Terms*—Large Language Models (LLMs), Multi-Agent, Static Program Analysis, Code Design Issue Localization**

I. INTRODUCTION

Software design quality is crucial for the long-term maintainability, extensibility, and evolution of systems. As a project evolves, design issues such as poor modularity, tight coupling, and excessive complexity often emerge, hindering developer productivity and system quality. These issues, if left unaddressed, lead to the accumulation of technical debt, resulting in increased development costs and decreased software reliability [1, 2]. Thus, an approach that automatically addresses these design issues could be widely beneficial and adopted to reduce development costs and promote developer productivity. With recent advancements in code intelligence tasks since these prior works, the next logical step towards automated refactoring is design issue localization.

Large Language Models (LLMs) have demonstrated significant capabilities in code-related tasks [3, 4]. However, their potential for recognizing software design issues remains largely unexplored. Our rationale of leveraging LLMs is based on the fact that LLMs have been trained on a vast amount of source code and demonstrated strong capabilities in several code-semantics reasoning tasks involving intricate relationships in a codebase. We hypothesize that they can identify flawed relationships among components, such as misplaced methods or other irregular code structures, and suggest refactoring changes, e.g., moving methods to more appropriate locations without using thresholds or hard-coded rules.

However, LLM-based approaches face key limitations in this domain. First, source code is complex, and its inherent structure makes it difficult for LLMs to correctly parse and leverage effectively [5]. These struggles arise from the misalignment between the structured nature of programming languages and the non-structured natural-language (NL) texts that LLMs are primarily trained on. Second, our preliminary experiments indicate that LLMs tend to generate incorrect results when dealing with program semantics across abstractions. Thus, a more appropriate representation for LLMs should be both *natural language-friendly* and *abstraction-aware*.

This paper presents LOCALIZEAGENT, a novel agent-based framework that bridges this gap and explore LLMs' capabilities in improving software design. Inspired by recent works on LLM-based frameworks for other tasks [6, 7], LOCALIZEAGENT focuses on design issues identifiable within a single class (e.g., god classes and feature envy). It leverages the results of program analysis tasks to synthesize NL summaries of system designs fitting for LLMs. Our key insight is that by combining **multi-agent design**, **context-aware prompting**, and **LLM-friendly code representations**, we can overcome the limitations of current LLM-based approaches and enable more effective design issue localization. We design LOCALIZEAGENT with the following major ideas:

First, **Multi-Agent Collaborative Framework:** Recent works have started to investigate LLM-based agents that prepare task-specific data for LLMs [8, 9]. Motivated by such works, we introduce a novel multi-agent system consisting of LLM-based and program analysis-based agents responsible for collaborative tasks to predict future refactoring activity. This framework comprises of (1) a "Design Issue Analysis Agent" that leverages program analysis tools to identify potential design anomalies, (2) a "Program Analysis Agent" that extracts relevant context from the codebase using established and lightweight program analysis techniques, (3) a "Context-aware Prompt-building Agent" that dynamically constructs prompts based on previous analysis results, and (4) an "LLM-based Ranking Agent" that prioritizes localized design issues based on their impact and relevance. This ranking agent is inspired by the work on LLM-based software activity ranking [10].

Second, **LLM-Friendly Code Context Representation:** Inspired by NExT [11], on representing execution traces in a natural language, we propose a novel representation of program analysis outputs tailored for LLMs. Traditional program analysis output often comprises of massive amounts of program dependency graphs (PDGs) or full Abstract Syntax Trees (ASTs), which are difficult or infeasible representations for LLMs to effectively process [11, 12]. Our approach addresses this by (1) summarizing complex source code data into concise NL descriptions and (2) preserving code structure through NL contextual summaries. In brief, *program analysis can effectively summarize the heavily involved abstractions into a natural-language format more suitable for LLMs*.

Third, **Context-Aware Prompt Generation:** LOCALIZEAGENT introduces a prompt generation module that dynamically synthesizes prompts based on the specific refactoring context and code analysis summaries. Since our approach is designed as a framework, it is extensible to accommodate additional refactoring scenarios and program analysis tasks.

To evaluate LOCALIZEAGENT, we conduct an experiment on a diverse set of real-world codebases from a refactoring dataset [13]. Our results demonstrate that our approach achieves a 206% relative improvement in accuracy for localizing design issues, a 166% improvement for complexity issues, and a 138% improvement for information-hiding issues compared to the baseline LLM-based techniques. Moreover, our approach demonstrates significant gains across different refactoring types, with the best models outperforming the baselines by an average of 75% to 184% in Exach-Match@1.

In brief, the main contributions of this paper are as follows:

- To our knowledge, LOCALIZEAGENT is the first work proposing to leverage program analysis tasks to synthesize natural language summaries of relationships between system abstractions to facilitate design issue localization.
- We introduce a multi-agent, LLM-agnostic framework LOCALIZEAGENT, that is adaptable for future LLMs, refactoring activities, and program analysis tasks.
- A natural language-based representation of program analysis output that enhances LLMs' abilities to handle software design input more effectively than traditional static analysis-based representations.
- Experiments on a real-world dataset of refactorings resolving design issues demonstrate our tool's effectiveness and provide an evaluation benchmark for future research.

II. BACKGROUND

Large Language Models (LLMs) have emerged as powerful tools for various code intelligence tasks, demonstrating remarkable capabilities in code understanding and generation [3, 14, 15]. The application of LLMs has expanded rapidly across the software engineering domain, encompassing tasks such as code translation [16], automated program repair [8], code review [17], and fuzzing [18]. Pomian *et al.* [19] generates and ranks extracted method refactorings.

Recent advances in large language models introduce the concept of "agents", which are components that enhance the LLM's capabilities by providing the additional contexts for specific tasks. These interconnected agents enable LLMs to interact with external tools, effectively expanding their problem-solving abilities beyond the confines of their initial training data. For instance, researchers have developed agents that allow LLMs to explore codebases [8, 20], execute and validate patches [8], and formulate plans for complex tasks [8].

III. MOTIVATION

A. Motivating Example

Let us use a real-world example to motivate our solution. Consider the example in Fig. 1. We have a real-world Java class that violates the *separation of concerns* design principle. The class handles multiple responsibilities, such as error handling and progress monitoring, leading to increased complexity and reduced maintainability. As a result, a software quality evaluation tool, named PMD [21], reports a high cyclomatic complexity score for the code before refactoring (Listing 1). Note that such a traditional software design evaluation tool, like PMD [21], is primarily used to detect high-level code smells using code metrics and rules (e.g., god classes). It cannot provide actionable suggestions on methods and classes such as code refactoring including splitting a god class or moving a method to another class, etc.

One of our primary objectives is to bridge the critical gap between high-level issue detection and actionable feedback. In our work, actionable feedback for fixing design issues and refactoring changes are treated as distinct tasks. Because refactoring may include changes that do not directly address design issues, some of them may also remain unresolved through refactoring (which falls beyond our scope). In brief, our focus is specifically on design issues that can be effectively resolved through refactoring.

Localizing and fixing such design issues can be challenging for developers, as it requires identifying the specific parts of the code that need to be changed. We hypothesize that

```
...
static void initialiseAndInstallRepository( IConfigurationElement remoteRepositoryElem
    , RepositoryPlugin localRepo,
    MultiStatus status, IProgressMonitor monitor) {
  SubMonitor progress = SubMonitor.convert(monitor, 3);
  - String implicitStr = remoteRepositoryElem.getAttribute("implicit");
  - String repoName = remoteRepositoryElem.getAttribute("name");
  - if (repoName == null) repoName = "<unknown>";
  - if ("true".equalsIgnoreCase(implicitStr))
    try {
      - RemoteRepository repo = (RemoteRepository)remoteRepositoryElem.createExecutableExtension("class");
      repo.initialise(progress.newChild(1));
      installRepository(repo, localRepo, status, progress.newChild(2));
    } catch (CoreException e) {
      String message = MessageFormat.format("Failed to initialise remote repository {0}.", repoName);
      if (status != null)
        status.add(new Status(IStatus.ERROR, Plugin.PLUGIN_ID, 0, message, e));
      Plugin.logError(message, e);
    }
    - else {
      - progress.worked(1);
    - }
} ...
```

Listing (1) Code Before Refactoring

```
...
static void initialiseAndInstallRepository( RemoteRepository repo, String repoName,  RepositoryPlugin localRepo,
   MultiStatus status, IProgressMonitor monitor) {
// Function body omitted for brevity
...
```

Listing (2) Code After Refactoring

Fig. 1: A Real-World Refactoring Instance Demonstrating Variable Parameterization

Note: In Listing 1, red lines indicate code removed after refactoring. Yellow highlighting in Listing 1 shows the parameters that were removed, and the green highlighting in Listing 2 shows the parameterized variables.

the Large Language Models (LLMs) are able to identify flawed relations among software components and program elements, such as misplaced methods and other irregular code structures, and suggest refactoring changes to fix those issues. However, our preliminary experiment with LLMs (e.g., GPT-4o) using naive prompts reveals their limitations in effectively identifying and localizing design issues in (Listing 2) [22].

Observation 1 [Limitations of Current LLM-based Approaches]. LLMs' ability to recognize and fix design issues is limited. Naive prompting techniques often fail to capture the *nuances of design principles*, leading to ineffective localization of design issues in existing codebases [22].

Observation 2 [Challenges in Processing Complex Code Representations]. LLMs require *global contextual information* to understand and analyze design issues in codebases effectively. However, the use of complex code representations, such as call graphs and program dependency graphs, can pose challenges for LLMs due to their limited understanding of programming languages [12] and the potential for inconsistent output due to unfamiliar input structure [11]. These limitations hinder LLMs' ability to make informed decisions based on the complex relations and dependencies in source code.

B. Key Ideas

From the observations, we have the following key ideas when designing LocalizeAgent:

1) Key Idea 1 [Multi-Agent Collaborative Design Issue Detection and Fixing Suggestions]: Recognizing design issues and deciding refactoring fixes is a complex task that requires analyzing various aspects of the codebase. Thus, we introduce a multi-agent learning framework in which specialized agents collaborate to achieve the tasks. We break down the problem into different subtasks, each handled by specialized agents designed to process information to identify and fix the issues.

2) Key Idea 2 [LLM-Friendly Code Context Representation]: To enable LLMs to analyze design issues effectively, we introduce a novel code context representation that is tailored to their strengths. Based on observation 2, traditional code representations, such as program dependence graphs (PDGs), are complex and can exceed the context window of LLMs, hindering their ability to process and understand the entire codebase. Thus, we propose a lightweight analysis technique that generates the concise natural language summaries of code metrics and dependencies. These summaries capture key project's structure, making it understandable for LLMs.

3) Key Idea 3 [Context-Aware Prompt Generation]: The effectiveness of LLMs heavily relies on the quality and relevance of the prompts [23, 24]. Generic or poorly constructed prompts often lead to suboptimal results, as they fail to capture the specific nuances and requirements of different scenarios. To address this limitation, we introduce a context-aware prompt generation technique that dynamically adapts the prompts based on the unique characteristics of each refactoring context. By leveraging the information obtained from our agents, we generate prompts that are tailored to the specific design issues and refactoring opportunities present within the codebase. For instance, as the above example requires a "parameterizing variable" refactoring, the LLM is prompted to identify both the variable and functions that need to be changed.

4) Key Idea 4 [LLM-based Ranking Agent]: While localizing design issues is crucial, prioritizing and ranking these suggestions is equally important. Without a proper ranking mechanism, developers may waste valuable time and resources addressing irrelevant or low-impact suggestions. We introduce an LLM-based Ranking Agent that leverages the collective knowledge and analyzing capabilities of LLMs and static analysis tools to prioritize the localized design issues.

C. Problem Formulation

We formulate the design issue localization using LLMs as follows. Given a codebase $\mathcal{C}_{\mathcal{B}}$ with potential design issues, we aim to leverage LLMs to identify the specific functions or methods that should undergo refactoring to resolve these issues. By localizing the design issues at the function level, we aim to provide developers with actionable insights into which parts of the code require attention and improvement. Formally, let $\mathcal{C}_{\mathcal{B}}$ be a codebase having a set of functions $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$. The goal is to identify a subset of functions $\mathcal{F}' \subseteq \mathcal{F}$ that exhibit design issues that can be fixed via refactoring. The LLMs are tasked with analyzing the codebase $\mathcal{C}_{\mathcal{B}}$ and generating a set of locations $\mathcal{L} = \{l_1, l_2, \ldots, l_m\}$, where each location l_i refers to a function $f_j \in \mathcal{F}'$ to be refactored.

IV. LocalizeAgent Approach

A. Overview

Figure 2 illustrates our approach for localizing design issues in source code. It uses a cooperative communication paradigm, where agents work together towards the shared goal of identifying design issues. The communication structure is centered around the **Planning Agent** serving as the central authority that manages the information flow and delegates tasks to the other agents. As seen in Fig. 2, the Planning Agent begins by invoking the **Design Issue Analysis Agent** to run a static analysis tool on the input code, identifying potential design issues. Using this information, the Planning Agent then gets the **Program Analysis Agents** to extract relevant code context by using lightweight analysis tools. These tools are used to create short summaries in natural language that capture important details about the code's structure, dependencies, metrics, and design attributes. Next, the **Context-aware Prompt Building Agent** creates customized prompts based on these summaries and types of refactoring identified by the Planning Agent. These prompts are given to an LLM to find specific design issues in the codebase and suggest refactorings. Finally, **LLM-based Ranking Agent** reviews the code, its context, and suggestions to prioritize the important issues.

B. Design Issues

Design issues in software systems manifest as structural problems that impede maintainability, extensibility, and code quality, typically emerging when code violates fundamental principles such as separation of concerns, information hiding, and modularity [25]. The design issues in our study is grounded in the seminal work of Hans Van Vliet [25]. For these issues, we employ PMD [21], a widely recognized static analysis tool to detect potential design issues in source code.

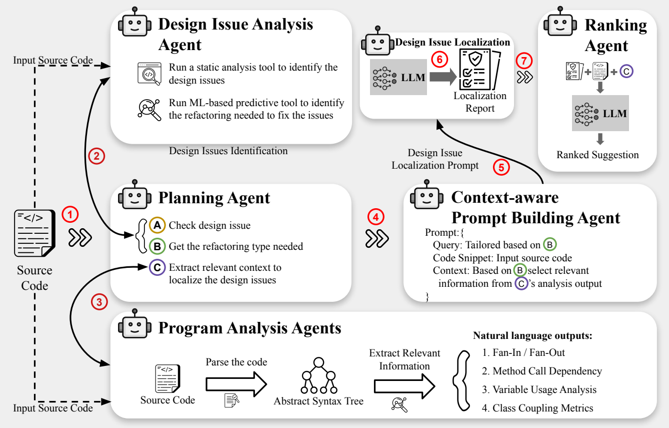

Input Source Code
Design Issue Analysis Agent
Run a static analysis tool to identify the design issues
Run ML-based predictive tool to identify the refactoring needed to fix the issues
Design Issues Identification
Design Issue Localization
LLM
6
Localization Report
7
Ranking Agent
C
LLM
Ranked Suggestion
Design Issue Localization Prompt
5
2
1
Source Code
Planning Agent
A Check design issue
B Get the refactoring type needed
C Extract relevant context to localize the design issues
4
Context-aware Prompt Building Agent
Prompt:{
Query: Tailored based on B
Code Snippet: Input source code
Context: Based on B select relevant information from C's analysis output
}
3
Program Analysis Agents
Source Code
Parse the code
Abstract Syntax Tree
Extract Relevant Information
Natural language outputs:
1. Fan-In / Fan-Out
2. Method Call Dependency
3. Variable Usage Analysis
4. Class Coupling Metrics
Input Source Code

Fig. 2: LocalizeAgent: Architecture Overview

A comprehensive review of PMD rules [21] was conducted to ensure alignment with the design categories listed in [25]. Multiple authors engaged in rigorous discussions to categorize the PMD's rules based on their descriptions. A specific evaluation scheme was implemented to ensure objectivity and reliability in the categorization process. Our criteria are the rule's primary focus, its impact on reuse and maintenance, and its relation to the design principles. This process yielded eleven PMD rules corresponding to the design categories. However, the abstraction and system structure issues were not addressed via refactoring. Thus, our analysis does not focus on three categories, resulting in a final set of eight PMD rules.

These eight rules address three crucial aspects of software design: **modularity**, **information hiding**, and **complexity**. Modularity, which promotes the separation of concerns, is assessed through rules detecting God Class, Coupling Between Objects, and Data Class [21]. Information hiding, essential for encapsulation and reducing dependencies, is evaluated using rules for Law of Demeter and Excessive Parameter List [21]. Complexity, which significantly impacts understandability and maintainability, is measured using rules for Too Many Methods, Excessive Public Count, and Cyclomatic Complexity [21].

C. Refactoring Types

Developers address the design issues through various mechanisms, e.g., architectural restructuring, design pattern solutions, or code refactoring. In our study, we focus on the historical refactoring changes that specifically address design issues. We combine refactoring with static analysis to ensure we study the cases where refactoring demonstrably resolved design issues, rather than considering all refactoring instances.

Our filtering on design issues in Section VIII-A2 resulted in the identification of four refactoring types that resolved those issues. That also ensures the ground truth for our experiments:

- *Parameterize Variable* (PV-Ref): This refactoring extracts a variable as a parameter, reducing coupling and increasing flexibility in design.

- *Inline Method* (IM-Ref): Inline Method replaces a method call with the method's contents, eliminating the need for a separate method and potentially reducing complexity.
- *Inline Variable* (IV-Ref): This refactoring replaces variable references with the variable's initializer expression, simplifying the code structure.
- *Move Method* (MM-Ref): Move Method moves a method to another class where it is more closely related, improving cohesion and reducing coupling between classes.

V. Planning and Analysis Agents

The Planning agent consists of two main components: the Design Issues Analysis agent and the Program Analysis agent, which work in tandem to identify design issues and extract relevant context for issue localization.

A. Design Issues Analysis Agent

The Design Issues Analysis agent is a specialized entity within our approach tasked with identifying design issues and the refactoring needed for a given codebase.

First, the agent leverages static analysis tools, such as PMD [21], to scan codebases and detect potential design anomalies. We use a set of rules from PMD for the tasks. For instance, PMD can detect coupling between objects and god classes, as shown in the example output below:

```
CouplingBetweenObjects: High amount of different
  objects as members denotes a high coupling

GodClass: Possible God Class (WMC=78, ATFD=36,
  TCC=8.000%)
```

Second, we integrate Aniche *et al.* [13], which trained a random forest algorithm on a dataset to predict the refactoring needs for a codebase. It takes the source code and predicts the refactoring types (e.g., inline method). For instance, for the example in Fig. 1, the tool would return that a parameterizing variable is needed to fix the issue.

The information gathered from this analysis is then used in the context-aware prompt-building phase (Fig. 2).

B. Context Extraction via Program Analysis Agents

The Program Analysis agents utilize static analysis tools to extract the relevant context from the codebase to aid LLMs in design issue localization. We propose a novel approach using lightweight analysis techniques as tools to generate concise, natural language summaries of key code metrics, dependencies, and design attributes.

1) Lightweight Code Analysis Tools: We utilize the following analysis techniques to demonstrate the effectiveness of using LLM-friendly program analysis outputs. The selection of these specific tools is based on their relevance to assist in identifying design issues that affect reuse and maintenance:

a) Fan-in and Fan-out Analysis ($A_{FI/FO}$): Fan-in denotes the number of times a method is called, and fan-out denotes the number of methods that are being called within a method [26]. Motivated by Schwanke *et al.* [26] and Layman *et al.* [27], we hypothesize that including a summary of this analysis task can indicate the coupling and complexity issues.

b) Call Relationships Analysis (A_{CR}): By identifying direct call dependencies between methods, this analysis helps detect potential coupling and cohesion issues.

c) Variable Usage Analysis (A_{VU}): This tracks variable usage patterns, which can indicate opportunities for refactoring that involve variables, such as parameter introduction.

d) Class Coupling Analysis (A_{CC}): By assessing class dependencies, this analysis provides insights into system modularity and potential areas for improving design quality.

2) Integration with LocalizeAgent*:* The Program Analysis agents form an integral part of the LocalizeAgent framework, bridging the gap between raw code and LLM-based analysis. The process works as follows:

1) Code Parsing: The agents parse the input codebase to construct an AST.
2) Analysis Application: Each lightweight analysis technique (A_i) is applied to the AST.
3) Summary Generation: The analyses produce natural language summaries (S) of relevant code properties.
4) LLM Integration: The Context-aware Prompt Building agent uses $\mathcal{S}$ to generate targeted prompts for the LLM.

These tools generate concise, natural language summaries, capturing key code structure and relationship aspects. Despite our current implementation, our approach is flexible, allowing the integration of additional analysis techniques for various design issues in the future. We use AST-based analysis to ensure accuracy and maintain code structural integrity.

VI. Context-aware Prompting Agent

The Context-aware Prompting agent bridges the gap between code analysis and LLM capabilities. It leverages the outputs from the Design Issue Analysis and Program Analysis agents to generate tailored prompts that guide the LLM in ranking design issues effectively. By incorporating relevant context into the prompts, we aim to enhance the LLM's ability to identify and suggest appropriate refactorings.

The Planning agent orchestrates the interaction between different components of our system. It triggers the Context-aware Prompting agent after receiving inputs from the Design Issues Analysis agent (identifying design issues and required refactoring types) and the Program Analysis agent (extracting relevant code context). This sequential process ensures that the prompts are informed by comprehensive code analysis.

a) Prompt Generation: Formally, let R=$\{r_1, r_2, \ldots, r_n\}$ be the set of identified refactoring types, and A = $\{a_1, a_2, \ldots, a_m\}$ be the set of analysis tools, each providing a summary s_i of code metrics and attributes. For a refactoring type $r_j \in R$, we define a prompt generation function f:

$$p_{r_j} = f(r_j, S_{r_j}) \tag{1}$$

S_{r_j} = $\{s_i \mid a_i$ is relevant to $r_j\}$ represents the relevant analysis summaries. The function f is implemented as a template-based system, augmented with heuristics that determine which code attributes are most relevant for each refactoring type. This approach allows for flexibility in prompt construction while maintaining consistency across different refactoring scenarios.

TABLE I: Program Analysis Tools Used for Each Refactoring Type

Refactoring Type	Tools	Rationale
Parameterize Variable	A_{CR}	Methods with high dependency on specific variables can be extracted as parameters to reduce coupling.
	A_{VU}	Usage patterns of variables across functions indicate candidates for parametrization.
Inline Method	$A_{FI/FO}$	Methods with low fan-in/out can be inlined to reduce unnecessary abstraction and simplify the code.
	A_{CR}	Direct call dependencies can help to identify methods that can be inlined to eliminate redundant calls.
	A_{CC}	Interdependencies between classes can indicate popular methods whose inlining can effectively reduce coupling.
Inline Variable	A_{VU}	Code can be simplified by inlining variables which are found to be frequently initialized.
Move Method	$A_{FI/FO}$	Methods with a high fan-out could reduce coupling by relocating them to a more relevant class.
	A_{CR}	Methods that are closely related to a different class should be moved to reduce cross-class dependencies.
	A_{CC}	Methods that contribute to high coupling should be relocated to more appropriate classes.

Note: These tools serve as a proof-of-concept for LOCALIZEAGENT's capabilities. Our approach can be extended to incorporate additional analysis tools.

b) Example of Prompt Generation: To illustrate this process, consider the prompt generated for *Parameterize Variable* shown in Listing 3. To aid the localization procedure, the prompt incorporates the contexts from the variable usage and call relationship summaries to guide the LLM.

Listing 3: Example of a prompt for issue localization

```
Query: Analyze the code snippet to identify
    variables that should be parameterized to
    reduce coupling.

Output Formatting: Don’t use ` or * on the
    response. Return the output using the
    following format only: Function:
    <func_name>, Variable: <var_name>.

Few-shot E.g: Function: get_user, Variable:
    user

Code Base: {Source Code}

Context: {
   Variable Usage Analysis (A_{VU})
   Class Coupling Analysis (A_{CC})
}
```

The context-aware prompt builder is performed for each refactoring type $r_j \in R$, generating a set of prompts P=\{p_{r_1}, p_{r_2}, ..., p_{r_n}\} that cover the various design issues identified in the codebase. Table I presents the analysis tools used for each refactoring type. The mapping of these tools is based on the characteristics of each refactoring type and the specific contextual information each analysis tool extracts (Table I).

VII. DESIGN ISSUE LOCALIZATION AND RANKING

This section details how we leverage LLMs to identify the specific locations of design issues and prioritize them.

A. Design Issue Localization

The localization process utilizes the context-aware prompts generated in the previous stage to guide an LLM in identifying specific functions or methods that exhibit design issues. This approach combines the strengths of static analysis tools with the contextual understanding capabilities of LLMs. Let $\mathcal{C_B}$ be the original codebase, and $A = \{a_1, a_2, \ldots, a_n\}$ be the set of code analysis tools, where each technique a_i generates a summary S_i. Given a refactoring type $r_j \in R$, we generate the prompt p_r, which contains the instruction, codebase $\mathcal{C_B}$, and relevant summaries $\mathcal{S}_r$. The LLM, denoted as $\mathcal{M}$, takes the prompt as an input. It then generates a set of locations $\mathcal{L}_r = \{l_1, l_2, \ldots, l_k\}$ that correspond to the design issues:

$$\mathcal{L}_r = \mathcal{M}(p_r) \tag{2}$$

Each location $l_i \in \mathcal{L}_r$ represents a function name in the codebase where a design issue related to the refactoring type r has been identified. The LLM achieves this by analyzing the code structure, dependencies, and metrics given in the context.

B. Prompting to Rank the Localized Design Issues

Listing 4: Prompt template for ranking suggestions

```
Given the following context and source code,
    rank the suggested functions in order of
    relevance. Don’t remove any functions from
    the list except for duplicates.

Context: {context}

Source Code: {Codebase}

Suggested Functions: {localized functions}

Return the ranked function as a list
Ranked functions:
```

While our localization process identifies potential areas for improvement, not all issues are equally significant or urgent. Thus, the prioritization of the identified design issues is crucial for efficient refactoring. This enables developers to optimize their limited time and resources by focusing on the most impactful refactorings. To address this need, we introduce an LLM-based ranking approach that leverages contextual code understanding to prioritize the identified design issues.

The motivation behind using an LLM for ranking stems from its ability to understand code relationships and recent success in ranking code fix suggestions [10]. Unlike rule-based systems, an LLM can potentially capture subtle interactions that influence the importance of a refactoring suggestion.

The ranking process begins by extracting the code context from the source code using the tools described in Section V-B. This context, along with the original code and the localized functions suggested for refactoring, is then used to create the prompt. The prompt instructs the LLM to rank the suggested functions based on their relevance for refactoring. The ranking process is guided by the prompt in Listing 4.

VIII. Evaluation

To evaluate LocalizeAgent, we seek to answer the following research questions:

a) **RQ1 (§ VIII-B) How effective is LocalizeAgent on localizing design issues?:** We assess the ability of LocalizeAgent to localize design issues in two scenarios:

(A) *Localizing a single design issue:* We evaluate LocalizeAgent's effectiveness in identifying the specific location of a single design issue within a codebase.

(B) *Localizing multiple design issues:* We investigate its performance in localizing multiple design issues.

b) **RQ2 (§ VIII-C) How sensitive is LocalizeAgent under different API settings when suggesting design issues refactoring?:** We perform a sensitivity analysis to evaluate how its performance varies under different settings.

c) **RQ3 (§ VIII-D) (Ablation Study) How does each component in LocalizeAgent contribute to the performance of localizing issues?:** We investigate the contribution of each key component in LocalizeAgent, namely the multi-agent learning framework, the natural language-based code analysis representation, and context-aware prompting.

d) **RQ4 (§ VIII-D) What is the time and budget to localize design issues?:** We evaluate LocalizeAgent's efficiency by measuring the time and financial cost to localize issues in the codebases of varying sizes.

A. Experimental Methodology

1) Dataset: We used the refactoring dataset released by Aniche *et al.* [13]. The dataset contains 11K diverse projects from Fdroid, Apache, and GitHub repositories. Following a similar methodology as NatGen [14], we randomly sampled only 6K projects due to the cost. These projects can contain multiple commits. Each commit contains the source code before and after refactoring, the refactoring types applied, and the function that changed, including its line number.

The authors used a static analysis tool, RefactoringMiner [28], to extract ground truth information on the refactoring types applied and the names of the refactored methods.

Note that RefactoringMiner does not identify the design issues. Instead, it serves as a mining tool that, when provided with the code before and after modifications, detects the types of refactorings applied with +90% precision on most refactoring types. Moreover, those refactoring changes might contain fixes that are unrelated to design issues. Thus, we used RefactoringMiner along with the PMD tool to collect a dataset of *PMD-reported code smells* connected to *RefactoringMiner-identified refactoring types which resolved them as follows.*

2) Rule-Based Design Issues Filtering: Specifically, we used the PMD tool to filter our dataset and ensure that the refactoring addresses the design issues. Formally, let C be a commit representing a refactoring, and let C_{before} and C_{after} be the code snapshots before and after the refactoring, respectively. The set of design issues identified by the PMD tool [21] in a codebase C_b is denoted as $I(C_b)$. Each issue $i \in I(C_b)$ belongs to one of the design categories $D = \{modularity, information\_hiding, complexity\}$. We then apply the following criteria:

1) Remove duplicate code and code with longer lines than the LLMs context window to maintain dataset quality.
2) For each commit C, run PMD on C_{before} and C_{after} to obtain the sets of design issues $I(C_{before})$ and $I(C_{after})$.
3) Select commit C if and only if both:
   - $\exists i \in I(C_{before}) \cap D$: There exists at least one issue i in C_{before} that belongs to any of the design categories D.
   - $\exists i \in I(C_{before}) \setminus I(C_{after})$: There exists at least one issue i in $I(C_{before})$ that is not in $I(C_{after})$. Thus, the set of issues in C_{after} is a proper subset of those in C_{before}, meaning the refactoring has resolved at least one issue.

Note that the combination of RefactoringMiner and PMD cannot predict design issues in source code (because no change is provided in our design issue detection problem). They can only recover the past activities to build our oracle. PMD cannot be used on its own to solve our problem either as it detects only high-level code smells without fixing suggestions (Section III).

TABLE II: Statistics of the Evaluation Dataset

Refactoring Types	Design Issues							
	Modularity		Information Hiding		Complexity		Total	
	Single	Multi	Single	Multi	Single	Multi	Single	Multi
Parameterize Variable	18	1	9	12	11	5	38	18
Inline Method	237	43	48	83	409	102	694	228
Inline Variable	66	11	21	31	100	24	187	66
Move Method	459	424	119	338	519	580	1,097	1,342
Total	780	479	197	464	1,039	711	2,016	1,654

3) Dataset Statistics: Table II presents the statistics of our dataset that satisfies our filtering criteria in Section VIII-A2. We categorize the dataset based on the number of refactoring operations performed to address design issues in each code instance. If only a single refactoring operation (e.g., one "Move Method") is applied to fix the design issues in a codebase, we consider it a single issue. Conversely, if multiple refactoring operations of the same type (e.g., more than one "Move Method") are applied to a single codebase to resolve design issues, we classify it as having multiple issues.

4) Models: We leverage three widely used LLMs: *GPT-4o, Claude 3,* and *Gemini 1.0.* We chose them based on their accessibility through public APIs, cost-effectiveness, and demonstrated proficiency in coding tasks [29, 30]. GPT-4o, developed by OpenAI, offers enhanced reasoning capabilities with a 128K input token limit. Google's Gemini 1.0 pro provides improved reasoning within a 32K token limit. Claude 3.0 (Haiku) from Anthropic offers an extended 200K token limit, allowing for the processing of larger code segments. We utilize Python and the JavaLang parsers to implement lightweight static analysis techniques, generating concise natural language summaries from ASTs. We employ the PMD tool [21] to identify design issues and verify refactoring effectiveness.

TABLE III: Results for Design Issue Localization per *Design Issue Category* (RQ1)

Design Issues		Claude 3			ChatGPT 4o			Gemini 1.0			Baseline-LLM		
		EM@1	EM@5	EM@10	EM@1	EM@5	EM@10	EM@1	EM@5	EM@10	EM@1	EM@5	EM@10
Complexity	**Single**	24.6	56.0	66.0	**28.5**	**59.3**	**66.2**	22.9	44.7	50.9	10.7	42.5	54.3
	Multi	■	48.6	55.4	■	**60.3**	**69.8**	■	45.7	56.0	■	37.9	47.4
Information Hiding	**Single**	30.3	52.8	58.4	**40.3**	**61.0**	**62.8**	32.9	47.6	48.9	16.9	48.5	53.7
	Multi	■	**58.9**	**66.1**	■	56.5	62.3	■	50.0	54.5	■	50.8	60.2
Modularity	**Single**	16.0	50.2	59.0	**24.5**	**54.0**	**62.9**	19.8	51.0	61.6	8.0	34.8	49.2
	Multi	■	40.3	48.2	■	**43.2**	**50.9**	■	34.8	41.9	■	32.7	45.8

TABLE IV: Results for Design Issue Localization per *Refactoring Type* (RQ1)

Refactoring Types		Claude 3			ChatGPT 4o			Gemini 1.0			Baseline-LLM		
		EM@1	EM@5	EM@10	EM@1	EM@5	EM@10	EM@1	EM@5	EM@10	EM@1	EM@5	EM@10
Parameterize Variable	**Single**	16.2	**51.4**	54.1	**24.3**	**51.4**	**56.8**	13.5	37.8	51.4	10.8	35.1	48.7
	Multi	■	**35.7**	50.0	■	**35.7**	35.7	■	**35.7**	**57.1**	■	28.6	**57.1**
Inline Method	**Single**	21.2	54.9	**65.4**	**23.6**	**56.5**	63.6	18.1	40.7	48.1	8.4	39.6	51.9
	Multi	■	55.9	62.9	■	**59.2**	**67.1**	■	46.0	50.7	■	39.9	50.2
Inline Variable	**Single**	**47.7**	74.4	84.6	43.6	70.8	73.8	44.6	**78.5**	**88.7**	27.2	64.6	69.7
	Multi	■	44.4	54.0	■	49.2	60.3	■	49.2	**66.7**	■	**57.1**	**66.7**
Move Method	**Single**	19.3	54.0	62.6	**25.0**	**60.0**	**64.9**	22.6	51.7	60.5	8.8	36.7	49.7
	Multi	■	**47.6**	**54.9**	■	44.0	51.2	■	36.2	42.3	■	32.1	43.5

5) Metrics: We use the Exact-Match@K metric (EM@K) to measure the accuracy of the first K results in detection.

Let $\mathcal{L} = \{l_1, l_2, \ldots, l_K\}$ be the ranked set of the first K suggestions generated by a model for a given design issue, and let g_t be the ground truth localization of the design issue. We define the Exact-Match@K metric as follows:

$$\text{EM@}K = \mathbb{1}(\exists l \in \mathcal{L} : l = g_t) \tag{3}$$

where $\mathbb{1}(\cdot)$ is the indicator function that returns 1 if the condition inside the parentheses is true and 0 otherwise. That is, EM@K=1 if at least one of the first K suggestions exactly matches with a design issue in the ground truth.

The total Exact-Match@K (EM@K) across all design issues or refactoring types in the dataset is computed as:

$$\frac{1}{N}\sum_{i=1}^{N} \text{EM@}K_i \tag{4}$$

where N is the total number of design issues or specific refactoring types. We compute EM@K for different K values.

Note that the LLM outputs the function name and additional refactoring-specific information, such as variable names or rationale. While other information provides context for developers, our evaluation focuses solely on correctly identifying the specific function name due to ground truth limitations.

B. Effectiveness on Localizing Design Issues (RQ1)

1) Localizing Single Design Issues (RQ1-A):

a) Baseline: We used GPT-4o API (named as Baseline-LLM in Tables III and IV) with a naive prompt, i.e., prompt the model to suggest refactoring changes without additional context. The reason is that GPT-4o has consistently shown better performance in our evaluation than other models. Thus, the naive prompt can be fairly compared with others.

b) Procedure: To evaluate the models' effectiveness in localizing design issues, we conduct experiments on the dataset in Section VIII-A3. All APIs are run with temperature 0 and nucleus sampling (top p) as 1 to get a deterministic output for the reproducibility purpose. We present our results from two perspectives: 1) design issue categories (Table III) to assess LocalizeAgent's performance on broader software quality aspects, and 2) refactoring types (Table IV) to evaluate its effectiveness on specific code transformation tasks.

c) Empirical Results: As seen in Tables III and IV, LocalizeAgent achieves high-performance gains over the baseline across all design issue categories and refactoring types. For complexity, information hiding, and modularity issues, LocalizeAgent with GPT-4o outperforms the baseline on EM@1 by *166%, 138%, and 206%*, respectively. This improvement highlights the better effectiveness of our multi-agent learning framework and context-aware prompting.

Table IV shows LocalizeAgent's effectiveness for different refactoring types. Notably, it shows good performance on the IV-Ref refactoring type, with EM@1 scores surpassing the baseline by *60% to 75%* across all LLMs. This suggests that the proposed approach is particularly adept at handling inline variable refactorings, possibly due to the specific code context and patterns associated with this refactoring type.

The results also highlight some areas for improvement. The Parameterize Variable refactoring type exhibits lower performance compared to others, with EM@1 scores ranging from 13.5% to 24.3%. Despite this, LocalizeAgent still outperforms the baseline from *25% to 125%* for Parameterize Variable refactoring, indicating the potential for further improvement with additional context or refined prompts.

Comparing the results in Tables III and IV, we can observe the trends in the relative performance among the LLMs. *GPT-4o consistently outperforms the other models across both design issue categories and refactoring types.* In contrast, Gemini 1.0 shows the lowest performance among the LLMs used with LocalizeAgent. However, even the least performing LLM with LocalizeAgent surpasses the baseline in all types of refactoring, demonstrating its robustness across all the fixing changes. The Baseline-LLM only exhibits better results than Gemini 1.0 in the Parameterize Variable refactoring type, while

LOCALIZEAGENT with Gemini 1.0 outperforms the baseline in all the remaining refactoring types.

Furthermore, our experiments reveal significant gains when K increases from 1 to 10, with EM scores improving by up to 43% (i.e., for Modularity, from 16% to 59%), demonstrating that LOCALIZEAGENT captures correct issues within its top recommendations. While this validates the framework's detection capabilities, enhancing the ranking mechanism remains important for prioritizing the most critical suggestions.

d) A Case Study: We evaluate our approach using the motivating example in Section III that needs Parameterize Variable refactoring. Our result shows an improved performance by incorporating the relevant context, i.e., A_{CR} and A_{VU}. LOCALIZEAGENT successfully identified the function `initialiseAndInstallRepository` within the top five suggestions, indicating that the function requires parameterizing variables. In contrast, the naive prompt approach failed to localize the issue within the top 10 suggestions [22].

2) Localizing multiple design issues (RQ1-B):

a) Baseline: We used the same baseline as RQ1-A.

b) Procedure: To measure correctness, we check if all the ground truth issues are within the top K results. Since evaluating multiple suggestions with EM@1 would be infeasible, we used only EM@5 and EM@10 for evaluation.

c) Empirical Results: The results in Tables III and IV show that localizing multiple design issues is more challenging than single issues, with the consistently lower EM@5 scores. However, the Inline Method refactoring is an exception. The model exhibits a comparative or even better accuracy for multi-issue localization. Interestingly, the baseline approach demonstrates a smaller performance gap between the single-issue and multiple-issue localization scenarios compared to LOCALIZEAGENT. While GPT-4o dominated for single-issue localization, Claude-3 showed better performance for multiple-issue localization, with 2.4 and 3.8 percentage points increasing in the information hiding category. Despite the challenges, LOCALIZEAGENT still outperforms the baseline in most design issue categories and refactoring types.

C. Sensitivity Analysis (RQ2)

a) Procedure: To assess the sensitivity of LOCALIZEAGENT to different API settings, we sampled 37 codebases from each refactoring category in our dataset. We varied the temperature parameter, which controls the randomness in the LLM's output. While our main experiments used temperature 0 to ensure reproducible results, we also evaluated the temperatures of 0.5 (T-0.5) and 0.9 (T-0.9) to analyze how output diversity affects performance. These values were chosen as they represent common settings: T-0.5 balances creativity with consistency, while T-0.9 maximizes exploration of solutions.

b) Empirical Results: Fig. 3 shows the heatmap of the differences in EM scores between the temperature settings of T-0.5 and T-0.9 for three LLMs and four refactoring types. The heatmap visually represents these differences; red indicates a performance decrease, and blue signifies a performance increase when changing from T-0.5 to T-0.9.

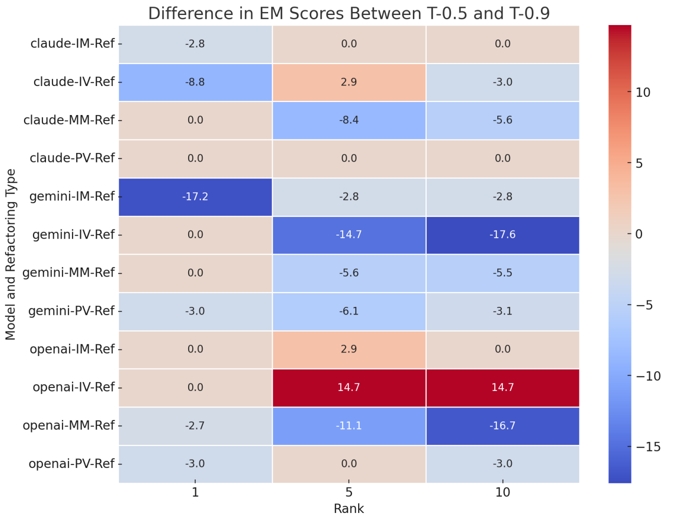


Fig. 3: Model Temperatures and Exact-Match Accuracy (RQ2)

Notably, an increase in temperature from T-0.5 to T-0.9 typically decreases the EM scores across most settings, *suggesting that higher randomness might dilute the precision of the suggestions*. For instance, the Gemini model for the Inline-Method refactoring experienced a sharp decrease in EM@1, highlighting its particular sensitivity to increased temperature. Conversely, some refactoring types, such as the Inline-Variable refactoring with the Openai model, show a slight improvement with a higher temperature. This indicates that *some refactoring types might benefit from the increased exploration, enabling the model to capture broader solution patterns*. These findings illustrate the nuanced impact of temperature on model performance and underscore the necessity of tuning this parameter.

D. Ablation Study (RQ3)

a) Procedure: To assess the contribution of each component in LOCALIZEAGENT, we conduct an ablation study by evaluating the performance under different settings, focusing on (1) the Program Analysis Agent and (2) the Ranking agent. First, we evaluate the effectiveness of LOCALIZEAGENT without including any context (i.e., without $A_{FI/FO}$, A_{CR}, A_{VU}, A_{CC}). Next, we study the contribution of the Ranking Agent.

b) Empirical Results: Table 5 presents our ablation study results. In the case of the Parameterize-Variable and Inline-Variable refactorings, our *context-aware prompting significantly contributes to performance across all LLMs*. For instance, the EM@1 scores drop from 2.9 to 11 percentage points for Parameterize-Variable refactoring for Claude 3 and GPT-4o. For the Inline-Variable refactoring, without context, the results show 1.6–10.2 percent decrease across all models. Moreover, the ranking component also shows a positive contribution, with EM@1 improvements ranging from 4.6 to 10.2 percentage points for Inline-Variable. GPT-4o benefits the most from context and ranking, achieving the highest EM@1 scores of 24.3% and 43.6% for the Parameterize-Variable refactoring and the Inline Variable refactoring, respectively.

TABLE V: Ablation Study for Localizing Design Issues per Refactoring Type (RQ3)

Refactoring Type	Claude 3			ChatGPT 4o			Gemini 1.0		
	EM@1	EM@5	EM@10	EM@1	EM@5	EM@10	EM@1	EM@5	EM@10
Parameterize Variable	16.2	**51.4**	**54.1**	24.3	**51.4**	**56.8**	13.5	**37.8**	**51.4**
↪ **w/o context**	13.3	38.3	40.0	13.3	35.7	43.3	**21.7**	33.3	41.7
↪ **w/o ranking**	**18.9**	45.9	51.4	**27.0**	45.9	**56.8**	13.5	35.1	48.6
Inline Method	21.2	**54.9**	**65.4**	23.6	**56.5**	**63.6**	18.1	40.7	48.1
↪ **w/o context**	23.4	54.6	61.0	21.7	45.6	51.0	**21.7**	**45.7**	**51.0**
↪ **w/o ranking**	**28.8**	54.6	64.2	**27.0**	51.3	62.9	17.4	36.5	43.3
Inline Variable	**47.7**	**74.4**	**84.6**	**43.6**	**70.8**	**73.8**	**44.6**	**78.5**	**88.7**
↪ **w/o context**	38.0	57.4	61.0	42.0	63.6	71.7	34.4	45.6	48.2
↪ **w/o ranking**	38.0	61.5	70.8	39.0	63.6	67.7	34.4	60.5	67.7
Move Method	19.3	**54.0**	**62.6**	25.0	60.0	64.9	22.6	51.7	60.5
↪**w/o context**	**28.4**	54.6	61.4	**29.1**	**61.5**	**66.8**	**28.5**	**54.6**	**61.3**
↪ **w/o ranking**	20.3	51.8	59.9	27.3	60.6	65.3	24.3	52.0	59.6

In contrast, *the impact of context-aware prompting is less substantial* for the Inline-Method and Move-Method refactorings. For Inline-Method, Claude 3 and Gemini 1.0 show slight improvements in EM@1 without context, while GPT-4o's performance drops by 3.4 points, suggesting that the provided context might be excessive in some cases. For the Move-Method refactoring, which relies the least on context, i.e., $A_{FI/FO}, A_{CR}, A_{CC}$ summary, context removal results in the smallest impact on EM@1, with the drop from 1 to 2.3 points. Our ranking shows the mixed impact, with the improvements for Claude 3 and Gemini 1.0 but a slight decrease for GPT-4o.

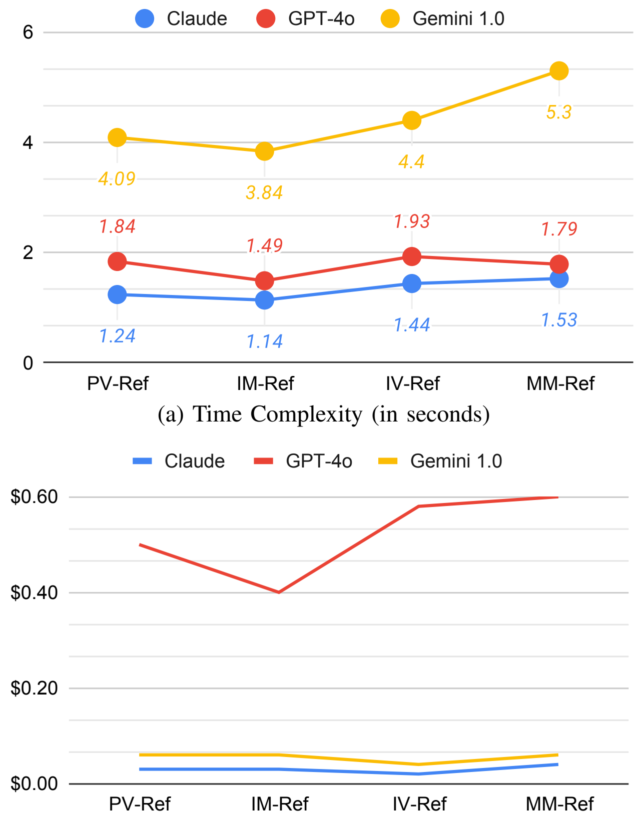
The model's effectiveness is tied to the complexity of the refactoring type and the LLM's understanding of the associated design principles. For refactoring tasks that involve methods like the Inline Method and Move Method refactorings, the LLMs can rely on inherent patterns and simpler rules, reducing the necessity for detailed contextual information. In these cases, excessive context might introduce noise, leading to a slight decline in performance. This is observed with the Inline-Method refactoring for Claude 3 and Gemini 1.0, where removing context even slightly improves performance, indicating these models' efficiency in handling common refactoring tasks without additional context. This limitation indicates that our approach may not be universally optimal across all refactoring scenarios. Future work should explore adaptive context mechanisms that can dynamically adjust the amount of the context based on the complexity of the refactoring task.

E. Time Complexity and Cost of APIs (RQ4)

a) Procedure: We assessed the time complexity and cost of LLM APIs for each refactoring type by sampling 37 codebases from each category. This approach maintains consistency for fair comparisons. Time complexity was gauged by the end-to-end duration—from sending the request to receiving LLM-generated refactoring suggestions. Costs were calculated considering the number of API calls, input/output sizes, and the pricing plans of each provider. The total cost was derived from these factors according to the respective pricing models.

b) Empirical Results: As seen in Fig. 4, the results highlight the differences in time complexity and cost among LLM APIs for design issue localization. *Claude 3 (Haiku) displays the lowest time complexity and API cost, proving the most efficient for quick feedback and budget-limited scenarios. GPT-4o offered a balanced performance in speed but was more expensive and suitable where trade-offs between speed and expense are acceptable*. Gemini 1.0, while slowest, maintained a moderate cost, fitting for balancing multiple factors.

MM-Ref refactoring consistently shows higher time complexity and costs across all LLMs. This might be expected as it uses three different contexts ($A_{FI/FO}$, A_{CR},A_{CC}), increasing processing time. Yet, the Inline-Method refactoring, also using three contexts, has lower time complexity, suggesting other factors influence processing speeds. The Inline-Variable refactoring, despite its smallest input size, has high time complexity.



(a) Time Complexity (in seconds)

(b) Costs of APIs

Fig. 4: Time Complexity and LLM Costs per Refactoring

This indicates that the inherent complexity of refactoring types and the efficiency of the LLMs in handling certain contexts significantly impact the overall performance.

IX. Limitations and Threats to Validity

Data leakage: We could face the risk of data leakage in the evaluation dataset. If the LLMs were pre-trained on the same or similar projects as those used in our evaluation, it could lead to an inflated performance metric. To mitigate this risk, we carefully selected a diverse set of real-world codebases from the refactoring dataset in Aniche *et al.* [13].

Non-determinism of LLMs: LLMs exhibit non-deterministic behaviors, which can lead to variability in the generated refactoring suggestions. This non-determinism affects the reproducibility and consistency of our results. The lack of explainability further compounds this issue, as it makes it difficult to understand why different suggestions are made across runs. For mitigation, we used a temperature setting of 0 for more stable predictions. We experimented with different settings in RQ2 (Section VIII-C).

Rule-based filtering criteria: We applied rule-based filtering criteria using PMD [21] to focus on specific refactoring types that successfully resolved design issues and contained the necessary ground truth. While these criteria help us evaluate the effectiveness in a controlled setting, they may limit the generalizability of our findings to other refactoring types or design issues and introduce false positives. To address this, we plan to use ML filtering techniques and expand our approach to a broader range of refactoring types and design issues.

X. Related Work

A. LLM for Code Generation and Understanding

Researchers explored LLMs for code understanding and generation tasks [3, 14, 15]. Recent works leverage emergent capabilities of LLMs, such as In-Context Learning for code intelligence tasks [11, 16, 17, 18, 31], while others explore the use of multi-agents and LLMs [8]. Dilhara *et al.* [32] combines static analysis and LLM to improve the effectiveness of transformation by example. For refactoring tasks, Pomian *et al.* [19] proposes a technique to query and rank LLM-suggested extract method refactorings. Shirafuji *et al.* [33] introduces a few shot-learning technique to apply refactoring fixes on code snippets. In contrast, our work explores how static analysis agents can supplement LLMs to suggest refactorings that improve software design [34]. Towards detecting and fixing software design issues, previous works have explored approaches for predicting refactoring activity, often at the function level [13].

B. LLM-based Agents for Software Engineering

Bouzenia *et al.* [8] presents an approach where an LLM, tasked with program repair, is augmented with “agents” that utilize external tools to explore the codebase, test patches, and plan goals. Zhang *et al.* [20] helps an LLM agent explore a codebase via provided code search APIs to resolve an issue autonomously. To our knowledge, our work is the first to leverage a multi-agent framework to detect design issues with LLM-friendly code analysis summary, and context-aware prompts.

XI. Discussion and Future Work

Large Language Model: The effectiveness of LocalizeAgent relies on the code generation capabilities of the underlying LLM. While we use generic LLMs like GPT-4, Claude, and Gemini, which outperform code-specific models [29, 30], evaluating other LLMs provides good insights. Moreover, we use an LLM-based refactoring-agnostic ranking algorithm. Future research could also explore developing ranking algorithms tailored to specific refactoring types as in Pomian *et al.* [19]. Our agent-based framework allows researchers to easily adapt and improve it.

IDE Integration: We could enhance developer tools and workflows by integrating with IDEs. Similar to recent advancements in IDE-integrated AI assistants [19, 35], future research could focus on embedding our output into IDEs to provide developers with context-aware, prioritized suggestions.

Expanding Design Issue Coverage: We currently target three design categories: modularity, information hiding, and complexity. Expanding this scope to include additional design principles, such as abstraction and system structure, would broaden the range of design issues our tool can identify. Future work should explore alternative static analysis tools beyond PMD to identify a wider range of design issues.

Supporting More Refactoring Types: We currently focus on local design issues within a class. Extending to handle global refactoring issues across multiple classes/modules presents challenges, such as developing techniques to capture inter-class relations and dependencies. Additionally, the lack of comprehensive benchmarks for project-level refactoring is an obstacle. Creating benchmarks through collaborative efforts or by mining large repositories is essential for future research.

XII. Conclusion

This paper introduces LocalizeAgent, a novel approach that leverages LLMs to effectively localize design issues in evolving codebases. Its key innovations include the leverage of multi-agent learning for collaborative refactoring, generating natural-language summaries of code metrics and dependencies to reduce input data loads and overcome the LLMs' context window limitations, and employing context-aware prompting tailored to specific refactoring types. Evaluated on real-world codebases, LocalizeAgent achieves 206%, 166%, and 138% relative improvements in accuracy for identifying design issues compared to baseline LLM techniques.

XIII. Data Availability

A replication package, including the source code and dataset, is available at [36].

Acknowledgment

This work was supported by the National Science Foundation under Grants: 25-12857, 25-12858, 15-18897, 15-13263, 21-20448, 19-34884, and 22-23812. The third author was supported in part by 21-20386 and the National Security Agency grant NCAE-C-002-2021. All opinions are those of the authors and do not reflect the views of sponsors. Generative AI was used to revise sections of this paper.

References

[1] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 47–52. [Online]. Available: https://doi.org/10.1145/1882362.1882373

[2] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "The financial aspect of managing technical debt," *Inf. Softw. Technol.*, vol. 64, no. C, p. 52–73, aug 2015. [Online]. Available: https://doi.org/10.1016/j.infsof.2015.04.001

[3] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, "CodeT5+: Open code large language models for code understanding and generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 1069–1088. [Online]. Available: https://aclanthology.org/2023.emnlp-main.68

[4] "OpenAI," https://openai.com/.

[5] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. Shujie, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2021.

[6] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, jul 2024. [Online]. Available: https://doi.org/10.1145/3660810

[7] R. Bairi, A. Sonwane, A. Kanade, V. D. C., A. Iyer, S. Parthasarathy, S. Rajamani, B. Ashok, and S. Shet, "CodePlan: Repository-Level Coding using LLMs and Planning," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, jul 2024. [Online]. Available: https://doi.org/10.1145/3643757

[8] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, April 27-May 3, 2025 2025.

[9] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh *et al.*, "OpenDevin: An Open Platform for AI Software Developers as Generalist Agents," *arXiv preprint arXiv:2407.16741*, 2024.

[10] N. Wadhwa, J. Pradhan, A. Sonwane, S. P. Sahu, N. Natarajan, A. Kanade, S. Parthasarathy, and S. Rajamani, "CORE: Resolving Code Quality Issues using LLMs," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 789–811, 2024.

[11] A. Ni, M. Allamanis, A. Cohan, Y. Deng, K. Shi, C. Sutton, and P. Yin, "Next: Teaching large language models to reason about code execution," in *International Conference on Machine Learning (ICML'24)*. PMLR, 2024.

[12] A. Hooda, M. Christodorescu, M. Allamanis, A. Wilson, K. Fawaz, and S. Jha, "Do large code models understand programming concepts? Counterfactual analysis for code predicates," in *Proceedings of the 41st International Conference on Machine Learning (ICML'24)*, vol. 235. PMLR, Jul 2024, pp. 18 738–18 748. [Online]. Available: https://proceedings.mlr.press/v235/hooda24a.html

[13] M. Aniche, E. Maziero, R. Durelli, and V. H. S. Durelli, " The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring ," *IEEE Transactions on Software Engineering*, vol. 48, no. 04, pp. 1432–1450, Apr. 2022. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TSE.2020.3021736

[14] S. Chakraborty, T. Ahmed, Y. Ding, P. T. Devanbu, and B. Ray, "NatGen: generative pre-training by "naturalizing" source code," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: ACM Press, 2022, p. 18–30. [Online]. Available: https://doi.org/10.1145/3540250.3549162

[15] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations (ICLR'23)*, 2023. [Online]. Available: https://openreview.net/forum?id=iaYcJKpY2B_

[16] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: ACM Press, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639226

[17] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng, "Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: ACM Press, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623306

[18] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4All: Universal Fuzzing with Large Language Models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: ACM Press, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639121

[19] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, T. Bryksin, and D. Dig, "Together We Go Further: LLMs and IDE Static Analysis for Extract Method Refactoring," 2024. [Online]. Available: https://arxiv.org/abs/2401.15298

[20] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: ACM Press, 2024, p. 1592–1604. [Online]. Available: https://doi.org/10.1145/3650212.3680384

[21] PMD, "PMD Static Analysis Tool," https://docs.pmd-code.org/latest/pmd_rules_java_design.html, Accessed 2024.

[22] annonymous, "ChatGPT Prompt of the Motivating Example," 2024, https://chatgpt.com/share/2986fa56-357d-4ea9-a821-9e8607389d9b.

[23] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, "Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks," *arXiv preprint arXiv:2310.10508*, 2023.

[24] T. Ahmed, K. S. Pai, P. Devanbu, and E. Barr, "Automatic semantic augmentation of language model prompts (for code summarization)," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: ACM Press, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639183

[25] H. Van Vliet, H. Van Vliet, and J. Van Vliet, *Software engineering: principles and practice*. John Wiley & Sons Hoboken, NJ, 2008, vol. 13.

[26] R. Schwanke, L. Xiao, and Y. Cai, "Measuring architecture quality by structure plus history analysis," in *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, 2013, pp. 891–900.

[27] L. Layman, G. Kudrjavets, and N. Nagappan, "Iterative identification of fault-prone binaries using in-process metrics," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '08. New York, NY, USA: ACM Press, 2008, p. 206–212. [Online]. Available: https://doi.org/10.1145/1414004.1414038

[28] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180206

[29] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, B. Zhu, H. Zhang, M. I. Jordan, J. E. Gonzalez, and I. Stoica, "Chatbot arena: An open platform for evaluating llms by human preference," in *ICML*, 2024. [Online]. Available: https://openreview.net/forum?id=3MW8GKNyzI

[30] "Chatbot arena leaderboard," 2024, https://huggingface.co/spaces/lmsys/chatbot-arena-leaderboard.

[31] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao, "Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: ACM Press, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3608134

[32] M. Dilhara, A. Bellur, T. Bryksin, and D. Dig, "Unprecedented code change automation: The fusion of llms and transformation by example," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 631–653, 2024.

[33] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe, "Refactoring programs using large language models with few-shot examples," in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2023, pp. 151–160.

[34] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo, "Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 5, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3643674

[35] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, A. Sokolov, T. Bryksin, and D. Dig, "EM-Assist: Safe Automated ExtractMethod Refactoring with LLMs," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: ACM Press, 2024, p. 582–586. [Online]. Available: https://doi.org/10.1145/3663529.3663803

[36] anonymous, "LocalizeAgent Reproducibility Package," 2024, https://github.com/fraolBatole/LocalizeAgent.