Wired for Reuse: Automating Context-Aware Code Adaptation in IDEs via LLM-Based Agent

Taiming Wang[†], Yanjie Jiang[‡*], Chunhao Dong[†], Yuxia Zhang[†], and Hui Liu[†]
[†]School of Computer Science and Technology, Beijing Institute of Technology, Beijing, 100081, China
[‡] College of Intelligence and Computing, Tianjin University, Tianjin, 300072, China
Email:{wangtaiming, dongchunhao22, yuxiazh, liuhui08}@bit.edu.cn, yanjiejiang@tju.edu.cn

***Abstract*—Copy-paste-modify is a widespread and pragmatic practice in software development, where developers adapt reused code snippets, sourced from platforms such as Stack Overflow, GitHub, or LLM outputs, into their local codebase. A critical yet underexplored aspect of this adaptation is *code wiring*: the context-aware process of substituting unresolved variables in pasted code with suitable variables or expressions from the surrounding context. Existing solutions either rely on heuristic rules or historical templates, often failing to effectively utilize contextual information, despite studies showing that over half of adaptation cases are context-dependent. In this paper, we introduce *WIRL*, an LLM-based agent for code wiring framed as a Retrieval-Augmented Generation (RAG) infilling task. *WIRL* combines an LLM, a customized toolkit, and an orchestration module to identify unresolved variables, retrieve context, and perform context-aware substitutions. To balance efficiency and autonomy, the agent adopts a mixed strategy: deterministic rule-based steps for common patterns, and a state-machine-guided decision process for intelligent exploration. We evaluate *WIRL* on a carefully curated, high-quality dataset consisting of real-world code adaptation scenarios. Our approach achieves an exact match precision of 91.7% and a recall of 90.0%, outperforming advanced LLMs by 22.6 and 13.7 percentage points in precision and recall, respectively, and surpassing IntelliJ IDEA by 54.3 and 49.9 percentage points. These results underscore its practical utility, particularly in contexts with complex variable dependencies or multiple unresolved variables. We believe *WIRL* paves the way for more intelligent and context-aware developer assistance in modern IDEs.**

***Index Terms*—Copy-paste-modify Practices, Code Reuse, Code Adaptation, Large Language Models, Agent.**

I. Introduction

Copy-paste-modify is a common and inevitable practice during development. The developers frequently reuse code snippets from online programming Q&A communities, e.g., Stack Overflow [1], open-source repositories, e.g., GitHub [2], or code generation of LLMs. However, in most cases the reused code snippets need additional adaptations to integrate to the local codebase. As reported, more than 85% code snippets from Stack Overflow need to be adapted (modified) before integration into the local code [3], [4]. Variable identifiers are usually under adaptation since undeclared or conflict identifiers would lead to compilation errors or potential bugs. Consequently, code wiring, replacing unresolved variables with existing ones from the local context, is one of the most prevalent forms of adaptation. An example of code wiring practice is presented in Fig. 1. Automatic code wiring can help developers get rid of the repetitive and error-prone processes during the adaptation and concentrate on the complex business logic. However, existing approaches are designed based on either simple heuristic rules [5] or templates extracted from historical modifications [6], leaving the context not effectively leveraged although 56.1% of the adaptation cases are dependent on the surrounding context as reported by Zhang et al. [3]. Advanced code editing approaches [7], [8] are dependent on a rich history of prior edits, information that is inherently scarce in the context of code wiring. Full-parameter LLMs have demonstrated promising performance; however, their high latency renders them impractical for real-world applications. On the other hand, distilled LLMs with smaller parameter sizes tend to under perform in terms of accuracy and robustness.

To this end, we present *WIRL*, an LLM-based agent for code **W**iring through **I**nfilling with **R**AG and **L**LMs. *WIRL* consists of three core components: an LLM, a customized toolkit, and an agent pilot. The customized toolkit offers three primary functionalities: (1) identifying and locating unresolved elements, (2) analyzing and collecting contextual information, and (3) infilling and recommending suitable substitutions. The agent pilot is responsible not only for initializing the prompt but also for coordinating the interaction between the LLM agent and the toolkit. It parses the LLM's output, updates the prompt dynamically, and invokes the appropriate tools as needed. With a dynamically updated prompt, *WIRL* incrementally gathers relevant context by invoking appropriate tools until the agent determines that sufficient information has been collected to make a final recommendation. To fully leverage the capabilities of LLMs, we reformulate the adaptation task as a retrieval-augmented generation (RAG)-based infilling task for the unresolved elements, framing it as a code completion problem which aligns more naturally with the strengths of LLMs. Empirical evidence from Zhang et al. [9] supports this reformulation, showing that LLM performance on code snippet adaptation tasks lags behind their performance on code completion and generation tasks. Moreover, by equipping with the RAG-based infilling strategy, even distilled LLMs with smaller parameter sizes can outperform full-parameter LLMs, thereby meeting both the latency and accuracy requirements of real-world software development scenarios. To enhance the efficiency of *WIRL*, we adopt a hybrid execution mode for

* Corresponding author: Yanjie Jiang (yanjiejiang@tju.edu.cn)

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00023

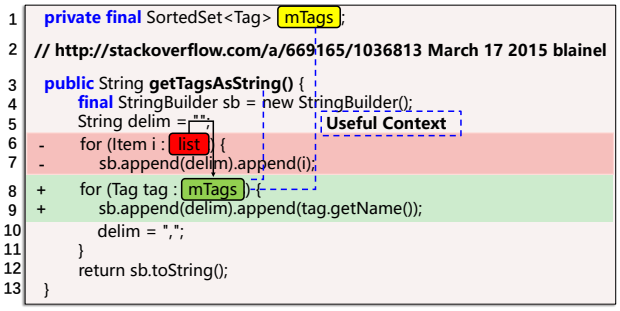

Fig. 1. Motivating Example

the agent, wherein essential steps are extracted and executed initially without invoking the LLM. In addition to this optimization, we introduce a state machine to guide the agent's exploration process, thereby reducing unnecessary or ineffective attempts and improving overall execution efficiency.

To assess the efficiency of *WIRL*, we manually curated a high-quality code wiring dataset derived from a publicly available code adaptation dataset [10]. Evaluated on this dataset, *WIRL* achieves an exact match precision of 91.7% and a recall of 90.0%. Notably, it outperforms advanced LLM baselines by 22.6 and 13.7 percentage points in precision and recall, respectively, and exceeds the performance of IntelliJ IDEA by 54.3 and 49.9 percentage points. These results highlight the effectiveness and practical advantage of *WIRL*. We also analyzed time efficiency and token consumption, demonstrating that *WIRL* meets real-time requirements with affordable computational cost.

The contributions of this paper are as follows:

- We propose *WIRL*, an LLM-based agent specifically designed for context-aware code wiring recommendations.
- We develop a customized toolkit that provides essential functionalities for precise and efficient code adaptation.
- We construct and release a high-quality dataset for code wiring evaluation, curated and validated through careful manual inspection.

The rest of this paper is structured as follows. Section II motivates this study. Section III illustrates the design of *WIRL*. Section IV presents experimental details and results. Section V first explores the broader implications and robustness of our findings through analysis of applicability and versatility. It then discusses potential threats to validity and the inherent limitations of our study. Section VI discusses the related works, and Section VII concludes this paper.

II. Motivating Example

To provide the intuition about how *WIRL* works and to define the key terminologies, we begin with a motivating example of a code wiring instance. Fig. 1 presents an example of copy-paste-modify (specifically code wiring) practice where a developer reused a piece of code snippet from Stack Overflow [11] and adapted it to the GitHub local repository [12]. The developer copied and pasted the line 6 and line 7 and then modified them into line 8 and line 9. For the sake of clarity, in this paper we call the code snippets in line 6 and line 7 ***isolated code*** since it is isolated before integration. The code snippets in line 8 and line 9 are called ***integrated code***, i.e., code after the integration. Specifically, in this example, local variable *"list"* (highlighted in red) is undefined and leads to syntax errors. The developers substituted it with the predefined field variable *"mTags"*(highlighted in yellow and green) to resolve the syntax errors. We call *"list"* an ***unresolved element***. It is worth noting that variables are not the only code entity that need to be modified, literal values or a method invocation expression could also be the target code entity. Consequently, we use *"element"* instead of only *"variable"*. *"mTags"* is called the ***context element***, i.e., the code elements in the context. Similarly, a *context element* could also be either a variable (including local variables, parameters, and fields) or an expression (e.g., method invocation).

We first present the problem formulation, which serves as an important foundation of our approach. Assume the set of *unresolved elements* is denoted by $\mathcal{U} = u_i$ and the set of *context elements* by $\mathcal{C} = c_i$. The code wiring task can be formally defined as identifying an injective mapping $\mathcal{M} : \mathcal{U} \rightarrow \mathcal{C}$, where each $u_i \in \mathcal{U}$ is mapped to a distinct $c_i \in \mathcal{C}$. In this work, however, we reformulate code wiring as a retrieval-augmented generation (RAG)-based infilling task. Specifically, each $u_i \in \mathcal{U}$ is replaced with a placeholder in the *isolated code*. The objective then becomes to infill these placeholders using the retrieved context information, including the variable names themselves, which is motivated by the observation that literal similarity often provides valuable contextual cues. This reformulation is grounded in the insight that code completion is more naturally aligned with the capabilities of LLMs than explicit code adaptation. This design choice is further supported by recent findings by Zhang et al. [9], which highlight the superior performance of LLMs in code completion tasks relative to direct adaptation scenarios.

In this example, *WIRL* first identifies the unresolved variable *"list"*, and then proceeds to iteratively collect relevant contextual information such as available variables in the current scope, semantic hints indicating that *"list"* should refer to a collection object, and method names suggesting that *"Tags"* is the intended target. Then it assesses the suitability of candidate variables by invoking a set of external tools tailored for context analysis. Once the context is deemed sufficient, *WIRL* conduct infilling and *"mTags"* is filled as the appropriate variable here. Eventually, *WIRL* automatically applies the modification within the IDE for developers.

III. Approach

A. Overview

The overview of *WIRL* is illustrated in Fig. 2. *WIRL* consists of three core components: an *LLM*, a *customized toolkit* containing three types of tools, i.e., *locator*, *collector*, and *completer*, and an *agent pilot*, which orchestrates communication between the LLM and the toolkit by parsing outputs,

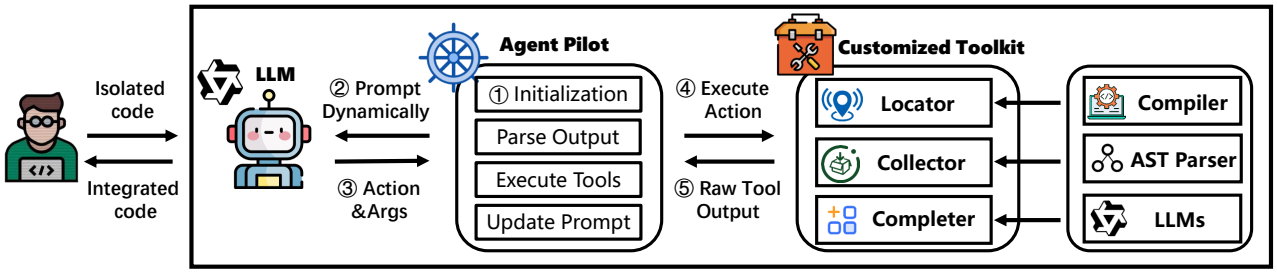

Fig. 2. Overview of WIRL

updating prompts, and executing tool invocations. Given a piece of *isolated code*, the *agent pilot* begins by initializing the prompt (*Step 1*). This prompt includes system roles, task descriptions, and instructions on how the task should be addressed using the available tools. It also embeds the location of *unresolved elements* and results from preliminary analyses. Using this prompt, the *agent pilot* issues a request to the LLM (*Step 2*). The LLM responds by specifying a tool to invoke from the customized toolkit (*Step 3*). The *agent pilot* then parses the response and executes the corresponding tool with the appropriate arguments (*Step 4*). The tool's output is subsequently parsed and integrated into the prompt (*Step 5*) for the next iteration. This loop continues iteratively until either the *completer* tool is executed, or a predefined computational budget is reached. The customized toolkit, implemented by the proposed approach, depends on high-level abstractions like compilers, AST parsers, and LLMs.

B. Dynamic Prompting

Adopting the *ReAct* framework [13], *WIRL* iteratively reasons and acts until the goal is met or the budget is exceeded. In each cycle, the *agent pilot* parses tool outputs and integrates them into the prompt for the next iteration, which consists of both static and dynamic sections.

1) Role (Static): The static section defines the agent's role as a Senior Java Development Engineer, tasked with autonomous, variable-level code integration based on the provided context, explicitly prohibiting any user intervention.

2) Goals (Static): Three primary goals to accomplish:

- ***Analyze context:*** Evaluate the current code context to determine whether it provides sufficient information to suggest a valid substitution.
- ***Collect more context:*** If the existing context is inadequate, invoke the appropriate tools from the provided toolkit to gather additional relevant information.
- ***Suggest a substitution:*** Once adequate context is obtained, leverage the agent's completion tool to recommend a suitable substitution for the unresolved element.

3) Guidelines (Static): Guidelines the agent should adhere:

- *Limited budget:* We emphasize that the budget is limited and it is important to be efficient to select the next steps.
- *Type should match:* We inform the agent that the basic principle is to make sure the retrieved *context element* is type-compatible with *unresolved element*.
- *Be careful to stretch:* More context typically leads to higher latency. Consequently, the agent is instructed to complete its recommendation using the minimal necessary context and to extend the context scope cautiously.
- *Extend scope:* We instruct the agent that in some cases, no suitable substitution candidates may be found within the immediate context. In such scenarios, the agent should expand its search scope beyond the current class, as the *context element* is more likely to be a method invocation expression rather than a simple variable.

4) State Description (Dynamic): To guide the agent in utilizing the *customized toolkit* in a meaningful and efficient manner, we define a state machine comprising three states: *initial state*, *insufficient context state*, and *sufficient context state*. Each state is associated with a set of available tools, as described in Section III-C, and the state description section of the prompt informs the agent of its current status. This mechanism is introduced because, in our preliminary experiments, we observed that the agent often engaged in aimless exploration when operating without explicit guidance. The transition rules between each state are presented in Table I. The agent begins in the *initial state*, where it identifies unresolved elements and performs a preliminary analysis of the context. It then transitions to the *insufficient context state* to iteratively gather necessary contextual information. The agent moves to the *sufficient context state* once enough information has been gathered or a predefined resource budget (i.e., iteration limit) is exhausted. In this state, it invokes the completer to generate the recommendation before terminating (*done state*). This structured workflow, with its built-in budget to handle errors and prevent infinite loops, ensures a focused and efficient decision-making process. It is important to note that while the state machine offers structured guidance, it does not enforce a fixed sequence of tool invocations. The selection and ordering of tools are left entirely to the agent's discretion.

5) Available Tools (Dynamic): This section illustrates the available tools (refer Section III-C) agent can call in each state.

6) Gathered Information (Dynamic): A key capability of *WIRL* lies in its ability to gather contextual information

TABLE I
TRANSITION RULES OF STATE MACHINE

From State	Transition Rules	To State
Initial	Initialization Finished	Insufficient Context
Insufficient Context	Budget Exhausted or Context Deemed Sufficient	Sufficient Context
Sufficient Context	Completer Executed	Done

through tool execution, which forms the foundation for generating accurate recommendations. To ensure the agent remains aware of previously executed tools and the context collected, this section of the prompt records the reasoning thoughts, the name of tool, its corresponding arguments (parsed from the LLM's output), along with the results returned by tool invocations (parsed from the tools' output).

7) Output Format (Static): To enable the *agent pilot* to effectively parse the output, we require all responses from the LLM be structured in JSON format. Each JSON object must contain three mandatory fields: *"thought"*, *"action"*, and *"action_input"*. The *"thought"* field captures the agent's reasoning and decision-making process based on the current context. The *"action"* field specifies the tools' name to be invoked next, while the *"action_input"* field provides necessary arguments for that tool, also formatted as a JSON object.

C. Customized Toolkit

One of the key novelties of *WIRL* is its ability to autonomously decide which tools to invoke based on the current state. The toolkit provided for *WIRL* (as shown in Table II) is carefully customized to facilitate the code wiring task.

1) Identifying unresolved elements: A prerequisite for effective *code wiring* is the accurate identification of *unresolved elements*. The *locator* tool (*identify_unresolved_elements*) leverages compiler information to detect unresolved variables and their references, and it is executed as the initial step. Since identifying and locating these elements is fundamental to addressing the task, the *agent pilot* invokes the *locator* automatically, without requiring input from the LLM.

2) Collecting Context Information: To minimize unnecessary LLM invocations and reduce iteration overhead, we provide two tools, i.e., *get_available_variables* and *get_unused_variables*, to help the agent efficiently construct a list of candidate variables. The *get_available_variables* tool parses the AST of the current Java file and collects all accessible local variables, method parameters, and class fields. To reflect realistic development scenarios, only local variables declared before the *unresolved element* are considered valid candidates. Once available variables are gathered, the *get_unused_variables* tool performs data-flow analysis to examine the usage of each variable. Variables with no detected references are marked as unused. To determine the syntactic role of an *unresolved element*, the tools *is_argument* and *is_receiver* inspect its AST context to identify whether it is enclosed within a method invocation. If the element appears as a method argument, its expected type can be inferred from the corresponding formal parameter, and the parameter name itself provides useful semantic cues. If the element is the receiver of a method call, the tool returns the *invoked method member* to support further analysis. The *reserve_type_compatible_ones* tool filters candidate variables by retaining only those whose types are compatible with that of the *unresolved element*. When the unresolved element is identified as a method receiver, the *retrieve_identical_function_call* tool takes the *invoked method member* as input and searches the current file for instances of the same member invocation, which may reveal contextually relevant patterns or variable usages. Finally, if no valid candidates are found in the current scope, the *get_method_names* tool is invoked. Given a class name, it returns the method members of that class that match the type of the *unresolved element*, providing additional opportunities for accurate substitution. The *sort_by_literal_similarity* tool is invoked when the agent determines that lexical similarity is a relevant factor. This tool computes the Levenshtein distance between the *unresolved element* and each candidate variable, ranking the candidates based on their literal similarity.

3) Infilling Isolated Code: Once the agent has collected sufficient contextual information, it invokes the *execution_completion* tool to infill the placeholders using the gathered data. This tool is implemented via an LLM call, where the prompt corresponds to the final version of the dynamically updated prompt that incorporates all previously collected context information.

D. Agent Pilot

The *agent pilot* orchestrating the communication between *LLM* and *customized toolkit*, playing a essential role in *WIRL*. Its responsibilities are outlined as follows:

1) Initializing Prompt: The *agent pilot* begins by initializing the prompt with static components such as the role definitions and task objectives. It then incorporates the input code (as described in Section IV-E1) into the prompt. Subsequently, the prompt is dynamically updated with the locations of *unresolved elements* and relevant contextual information, obtained by invoking the *Locator* (*identify_unresolved_elements*) and *Collector* tools (*get_available_variables*, *get_unused_variables*, *is_argument*, and *is_receiver*). This initialization process substantially reduces unnecessary LLM invocations and accelerates context analysis, thereby enhancing the overall efficiency of *WIRL*.

2) Parsing Output: Since the outputs of LLMs are returned in JSON format, they can not be directly used for tool execution. Accordingly, the second responsibility of the *agent pilot* is to validate and process these outputs. If an LLM response deviates from the expected structure, e.g., due to hallucinations or formatting errors, the *agent pilot* must attempt to correct the output or handle exceptions gracefully. In addition to output parsing and error handling, the *agent pilot* also maintains a memory of previously executed tools and their corresponding arguments to prevent redundant operations.

3) Executing Tools: With the parsed tool names and corresponding arguments, the *agent pilot* invokes the designated

TABLE II
AVAILABLE TOOLS FOR THE AGENT

Tools	Type	Applicable State	Description
identify_unresolved_elements	Locator	Initialization	Analyze compiler information to identify the *unresolved elements*.
get_available_variables	Collector	Initialization	Conduct code analysis and get the available variables in the current context scope.
get_unused_variables	Collector	Initialization	Conduct data flow analysis and get the unused variables in the current context.
is_argument	Collector	Initialization	Judge whether the *unresolved element* plays an argument in a method invocation.
is_receiver	Collector	Initialization	Judge whether the *unresolved element* plays a receiver in a method invocation.
retrieve_identical_function_call	Collector	Insufficient Context	Retrieve variables that invoke the identical function calls with *unresolved element*.
reserve_type_compatible_ones	Collector	Insufficient Context	Only reserve the variables that are type-compatible as the *unresolved element*.
sort_by_literal_similarity	Collector	Insufficient Context	Sort the available variables by their similarity with the *unresolved element*.
get_method_names	Collector	Insufficient Context	Collect method members with same type as *unresolved element* in the given class.
execute_completion	Completer	Sufficient Context	Invoke LLM to complete the code snippets with the collected context information.

tools and returns their outputs. Notably, all tool executions are performed within an isolated environment to ensure they do not interfere with the internal functioning of *WIRL*.

4) Updating Prompt: Once the tool output is available, the *agent pilot* updates all dynamic sections of the prompt in preparation for the next iteration. Specifically, it modifies the current state and the list of available tools, and appends the *"thought"*, *"action"*, *"action_input"*, and *"observation"* (i.e., the tool's output) to the accumulated context information.

IV. EVALUATION

A. Research Questions

- **RQ1:** How well does WIRL perform compared against baselines?
- **RQ2:** How well does WIRL perform regarding latency?
- **RQ3:** How well does WIRL perform regarding token consumption and monetary cost?
- **RQ4:** How do the design components contribute to the performance of WIRL?

RQ1 investigates the effectiveness of *WIRL* in comparison to selected baseline approaches for automatic code wiring. By addressing this question, we aim to understand how well *WIRL* performs in real-world development scenarios, as the evaluation leverages real data under realistic experimental conditions. Additionally, this analysis helps identify specific cases where *WIRL* outperforms the baselines. Gaining insights into the strengths of *WIRL* can guide future improvements and optimizations of the approach.

RQ2 examines the time efficiency of *WIRL* relative to baseline methods, with a particular focus on whether *WIRL* can meet the responsiveness requirements of integrated development environments (IDEs) and developers. To answer this question, we use the average time taken to complete a code wiring task as the primary metric. This allows us to assess whether *WIRL* delivers timely recommendations that support fluid and productive software development workflows.

RQ3 assesses the token consumption and associated monetary cost of *WIRL*. Specifically, we evaluate input/output token usage and the total cost incurred per code wiring instance. By comparing these metrics against those of the baseline approaches, we aim to determine the cost-efficiency of *WIRL* in delivering practical code wiring support at a reasonable computational expense.

RQ4 quantifies the contribution of each of *WIRL*'s design components, namely the Locator, Collector, and Completer tools, as well as its architectural design, to the system's overall performance. Understanding the individual impact of these elements is crucial for validating the novelty of our approach and justifying the design of the *WIRL* framework.

B. Dataset

To the best of our knowledge, there are no existing datasets specifically collected for the evaluation of automatic code wiring approaches. Recently, Zhang et al. [3] investigated context-based code snippet adaptation and constructed a high-quality dataset through a combination of automated processing and meticulous manual curation. The resulting dataset comprises 3,628 real-world code reuse cases, where code snippets were copied from Stack Overflow and subsequently adapted for use in GitHub projects. Their dataset was initially derived from the latest version of SOTorrent [14] (version 2020-12-31), available via Zenodo [15], and includes only those reuse cases in which GitHub files contain explicit references to Stack Overflow posts. Given the quality and relevance of this dataset, we adopted it as the foundation for our evaluation and performed additional filtering and manual curation. The construction of the evaluation dataset used in this paper involves the following steps:

- To prevent data leakage and potential overfitting, we first excluded the 300 sampled instances used by Zhang et al. [3], as the design of *WIRL* was partially inspired by their empirical observations.
- From the remaining 3,328 reuse cases, we performed a deduplication step to eliminate redundancy caused by forked repositories. Subsequently, we constructed a mapping between GitHub repositories and the corresponding Stack Overflow posts.
- As *WIRL* depends on compiler feedback and AST binding analysis, it requires the project to be resolvable. Therefore, we conducted a project-wise manual inspection of the remaining dataset. The mappings were sorted in descending order based on the frequency of references to each repository, prioritizing more frequently reused code.

- Following this order, two authors independently reviewed each code reuse case to determine whether it qualifies as a *code wiring* instance. In cases of disagreement, discussions were held until a consensus was reached. Ultimately, we identified 100 *code wiring* cases involved with 221 pairs of *unresolved elements* and *context elements* from Stack Overflow to GitHub. For ease of reference, we refer to this curated evaluation dataset as $CW Evaluation$.

C. Selected Baselines

1) SOTA Approaches: Since code editing task is potential to solve the code wiring problem, we include two state-of-the-art approaches in code editing, e.g., GrACE [7] and CoEdPilot [8].

We selected *ExampleStack* [16] as a baseline because it represents the state-of-the-art in automatic code wiring. *ExampleStack* is a Google Chrome extension designed to assist developers in adapting and integrating online code examples into their own code repositories. The authors provide comprehensive reproduction instructions, which allowed us to successfully set up and run *ExampleStack* in our evaluation.

Besides that, we selected the widely adopted industrial IDE, *IntelliJ IDEA* [5], as one of our baselines. The renaming recommendation support in IDEA is both practical and suitable to address the code wiring problem.

2) Raw LLMs: We take the following raw LLMs as our baselines: Distilled models, i.e., *GPT-4o-mini* [17], *Qwen2.5-Coder-14B* [18], and *Qwen2.5-Coder-32B* [19]; Full-parameter models, i.e., *DeepSeek-V3* [20] and *Qwen-max* [21].

D. Metrics

We adopted the evaluation metrics used by Wang et al. [22] to assess the performance of *WIRL* and the baselines. The definitions are introduced in the following:

- *#Total Cases:* the number of cases involved in the evaluation, i.e., number of *unresolved elements*.
- *#Recommendation (#Rec):* the number of cases where the evaluated approaches make a recommendation for the developers. Note that the number of recommendations (#recommendation) may not always match the total number of cases (#total cases).
- *#Exact Match (#EM):* the number of cases where the recommended code elements are identical to the ground truth (i.e., the existing names in context).
- $EM_{Precision}$ *(EM_P):* the number of exact matches divided by the number of recommendations, i.e.,

$$EM_{Precision} = \frac{\#Exact\ Match}{\#Recommendation} \tag{1}$$

- EM_{Recall} *(EM_R):* the number of exact matches divided by the number of cases involved in the evaluation.

$$EM_{Recall} = \frac{\#Exact\ Match}{\#Total\ Cases} \tag{2}$$

E. Experimental Setup

1) Input and Output Format for LLMs: To better simulate real-world development scenarios, the input code format is configured as follows:

- For the LLM baselines, the entire class containing the *isolated code* is provided as input. The *isolated code*, representing the segment requiring adaptation, is explicitly marked using a pair of control tokens, $< start >$ and $< end >$, to delineate the adaptation region. The processed data format and the complete prompt format used for the LLM baselines are available in our public repository [23].
- To enhance efficiency, the input to *WIRL* consists solely of the method declaration containing the *isolated code*. *WIRL* is designed to operate with the minimal necessary context and selectively expand the context scope when needed.
- Furthermore, we assume that any code following the *isolated code* is unavailable, reflecting the common top-down coding pattern observed during software development. In this scenario, only the code preceding the adaptation region, which within the same method declaration, is accessible as context.

For the sake of clarity, we constrained the LLM baselines to output only the *unresolved elements* along with their corresponding *context elements*.

2) LLM Configurations: To promote self-consistency and mitigate output variability, we set the temperature to 0. The top-p is set to 0.2, and the other parameters are set to default. Furthermore, each evaluated approach was executed five times, and the most frequently generated output was selected as the final result, following the protocol outlined by Chen et al. [24].

3) Baseline Configurations: As *IDEA*, *ExampleStack*, and *CoEdPilot* produce ranked lists of candidate substitutions, we only considered their top-ranked recommendation as the final answer to ensure a fair and consistent comparison across all evaluated approaches. For *CoEdPilot*, we also equipped it with our Locator module to maximize its performance.

Due to the deprecation of the *code-davinci-002* model, which was originally used in the zero-shot implementation of *GrACE*, we sought a suitable replacement. OpenAI officially discontinued support for the Codex API, including *code-davinci-002*, on March 23, 2023, and recommended transitioning to more advanced models. To this end, we opted to use a more advanced model, *GPT-4o*, for our implementation of *GrACE*.

4) Implementations: To implement *WIRL*, we built a plugin for IntelliJ IDEA using the *IntelliJ PlatForm Plugin SDK* [25] and the Langchain4J [26] framework to interface with LLM APIs. While *WIRL* is designed to be compatible with any advanced LLMs, in this study we use *Qwen2.5-Coder-14B* as the backend model for two key reasons. First, if *WIRL* demonstrates superior performance over state-of-the-art full-parameter LLMs while relying on a smaller model, it further highlights the strength of our design. Second, models with fewer parameters typically respond faster [27], [28], making them more suitable for real-time usage in development

TABLE III
COMPARISON AGAINST BASELINES

Baselines	#EM	#Rec	EM_P	EM_R	Latency
IDEA	79	189	41.8%	35.7%	**3.7ms**
ExampleStack	4	10	40.0%	1.8%	729.1ms
CoEdPilot	21	32	65.6%	9.5%	2,417.4ms
GrACE	163	184	88.6%	73.8%	2,873.4ms
GPT-4o-mini	101	145	69.7%	45.7%	3,930.4ms
Qwen2.5-Coder-14B	110	157	70.1%	49.8%	4,624.5ms
Qwen2.5-Coder-32B	149	197	75.6%	67.4%	6,799.4ms
Qwen-max	124	159	78.0%	56.1%	5,393.6ms
DeepSeek-V3	129	171	75.4%	58.4%	11,489.9ms
WIRL	**199**	**217**	**91.7%**	**90.0%**	5,255.6ms

environments. In our current implementation, we utilize the *Program Structure Interface* (PSI) [29], a powerful toolkit for syntactic and semantic analysis, to gather compiler information for the *Locator* and perform AST analysis for the *Collector*.

In addition, *WIRL* employs an iterative strategy to perform context-aware code wiring. To balance latency, computational cost, and effectiveness, we empirically set the maximum number of iterations at two.

F. RQ1: Outperform the SOTA

In Table III, the first column lists the evaluated approaches. The second and third columns indicate, respectively, the number of cases in which the recommended code elements exactly match the ground truth and the number of cases in which a recommendation is made by the approach. The fourth and fifth columns, $EM_{Precision}$ and EM_{Recall}, report how often the recommendations are correct and how many of the correct substitutions are successfully identified, respectively (see Section IV-D for detailed definitions). From Table III, we make the following observations:

- *WIRL* outperforms all selected baselines by a significant margin in both EM_{Recall} and $EM_{Precision}$. It produces the highest number of recommendations (217) while also achieving the highest number of exact matches (199). The resulting EM_{Recall} and $EM_{Precision}$ scores of 90.0% and 91.7%, respectively, reflect its strong and reliable performance in real-world development scenarios.
- Compared to *IDEA*, *WIRL* achieves improvements of 54.3 percentage points in EM_{Recall} and 49.9 percentage points in $EM_{Precision}$. Against *ExampleStack*, it shows even greater gains: 88.2 and 51.7 percentage points in EM_{Recall} and $EM_{Precision}$, respectively. Compared to advanced code editing approaches, i.e., *GrACE* and *CoEdPilot*, *WIRL* achieves improvements of 16.2 and 80.5 percentage points in EM_{Recall} and 3.1 and 26.1 percentage points in $EM_{Precision}$, respectively. When benchmarked against the advanced LLM model *Qwen2.5-Coder-32B*, *WIRL* still delivers notable improvements of 22.6 and 13.7 percentage points, further demonstrating its superior effectiveness.
- *WIRL* also substantially improves upon the performance of directly using a raw LLM (*Qwen2.5-Coder-14B*). Without the academic design described in Section III, the baseline LLM achieves only 49.8% EM_{Recall} and 70.1% $EM_{Precision}$. With the integration of the external toolkit and structured prompt design, *WIRL* improves EM_{Recall} and $EM_{Precision}$ by 40.2 and 21.6 percentage points, highlighting the value of its architectural innovations.

1) Analysis Over IDEA: First, IDEA's suggestion mechanism, designed for renaming, lacks effective ranking mechanism for code wiring. Our analysis revealed that while IDEA's suggestion list contained the correct substitution in 84.1% of cases (159 out of 189), it was often buried among a large number of irrelevant options, sometimes over 100, requiring significant developer effort to locate. Only 49.7% candidates in the top position are correct answers. This increases cognitive load and slows down the adaptation process. In contrast, *WIRL* provides a single, high-confidence recommendation for each unresolved element with 91.7% precision, turning a tedious search into a simple validation step. Second, IDEA failed to generate any recommendations in 14.5% of cases. These failures typically occurred when the unresolved elements are involved with complex expressions rather than simple variable s, a scenario that falls outside the scope of its renaming functionality. *WIRL*, by leveraging static code analysis, successfully handles these more complex cases, demonstrating its broader applicability and robustness.

2) Analysis Over ExampleStack: The core limitation of *ExampleStack* is its dependency on a pre-collected database of historical code modifications. Its effectiveness is therefore constrained to scenarios that have been seen before, which significantly hampers its performance on novel problems common in real-world development. In our evaluation, only 6 code wiring instances within $CWEvaluation$ had similar historical modifications in *ExampleStack*'s database [6]. Across all test cases, *ExampleStack* attempted 10 recommendations, with only 4 being correct. In contrast, *WIRL* leverages the reasoning capabilities of LLMs to generate solutions dynamically and from scratch for each specific context. This design allows *WIRL* to operate independently of historical data, giving it far greater generalization and effectiveness across diverse and unseen code wiring scenarios.

3) Analysis Over Code Editing Methods: As presented in Table III, *WIRL* surpasses the performance of advanced code editing methods like *GrACE* and *CoEdPilot*. This superiority can be attributed to two primary factors: First, these code editing methods are fundamentally designed to leverage a rich history of prior code edits. The code wiring scenario, however, involves pasting code from external sources, a context that inherently lacks such a historical record. This foundational mismatch significantly curtails the effectiveness of code editing methods. Second, the core methodologies of these tools are not well-suited for code wiring. *CoEdPilot*'s key modules for analyzing prior and subsequent edits are ineffective in this context, as there is little relevant edit history to analyze. Consequently, its generation module, which is fine-tuned specifically for edit-history-rich tasks, is ill-equipped for the distinct challenge of code wiring. *GrACE*, utilizing GPT-4o in a zero-shot setting, achieves impressive results

```
public void refreshBlocksForCommentStatus(CommentStatus)
-     int start = list.getFirstVisiblePosition();
-     for(int i=start, j=list.getLastVisiblePosition();i<=j;i++)
-       if(target==list.getItemAtPosition(i)){
-         View view = list.getChildAt(i-start);
-         list.getAdapter().getView(i, view, list);
+     int firstPosition = listView.getFirstVisiblePosition();
+     int endPosition = listView.getLastVisiblePosition();
+     for (int i = firstPosition; i < endPosition; i++) {
+       if (mCommentListPosition == i) {
+         View view = listView.getChildAt(i - firstPosition);
+         listView.getAdapter().getView(i, view, listView);
      }}}
Qwen2.5-Coder-14B: list->listView
WIRL: list->listView list->listView target->mCommentListPosition list-
>listView list->listView list->listView list->listView
```

Fig. 3. Example of Identification Failure for Raw LLM

```
import java.nio.charset.Charset;
public static String[] readFileIntoStringArr(final String path)
                throws IOException {
        ......
-     List<String> lines = Files.readAllLines(Paths.get(path), encoding);
+     final List<String> lines = Files.readAllLines(Paths.get(path),
+     Charset.defaultCharset());
  }
Qwen2.5-Coder-14B: encoding->lines
WIRL: encoding->Charset.defaultCharset()
```

Fig. 4. Example of Recommendation Failure for Raw LLM

```
public DBMSConnection(DBMSConnectionProperties properties) throws
SQLException, InvalidDBMSConnectionPropertiesException {
try {  ......
       DriverManager.setLoginTimeout(3);
       DBMSConnection.getAvailableDBMSTypes();
-    this.connection=DriverManager.getConnection(url1,user,password);
+    this.connection=DriverManager.getConnection(properties.getType(
+    ).getUrl(properties.getHost(), properties.getPort(),
+    properties.getDatabase()), properties.getUser(),
+    properties.getPassword())
}
WIRL + Qwen2.5-Coder-14B / Qwen2.5-Coder-32B / DeepSeek-V3 :
url -> properties.getUrl(),
user -> properties.getUser(),
Password -> properties.getPassword()
```

Fig. 5. Example of Recommendation Failure for WIRL

by combining the powerful completion capabilities of LLMs with a carefully engineered prompt. This design philosophy is highly analogous to that of *WIRL*, and its success further validates the effectiveness of our approach.

4) Analysis Over Raw LLMs: *WIRL*'s superiority over raw LLMs is evident in both recall and precision. The lower EM_{Recall} of raw LLMs can be attributed to two main factors. First, their fixed context windows prevent them from processing large classes that exceed token limits, resulting in complete failure. Second, they often fail to identify all unresolved elements within the code. As illustrated in Fig. 3, *Qwen2.5-Coder-14B* successfully identified *"list"* and recommended *"listView"*, but it failed to detect *"target"*. In contrast, *WIRL* uses compiler information to reliably identify all unresolved elements, overcoming these limitations.

The reduced $EM_{Precision}$ of raw LLMs stems from their difficulty in analyzing complex contexts, especially when the substitution is not a simple variable but requires synthesizing code, e.g., a method call. As shown in Fig. 4, a raw LLM is unlikely to deduce the correct substitution *"Charset.defaultCharset()"* because it lacks awareness of the class's available static methods. In contrast, *WIRL* addresses this limitation by leveraging its toolkit to first identify the required type (e.g., *"Charset"*) and then retrieve relevant methods (e.g., *"defaultCharset()"*), supplying the LLM with the necessary information to make a correct recommendation. Additionally, raw LLMs sometimes generate extraneous substitutions, increasing false positives and further lowering precision.

5) Failure Analysis for WIRL: While *WIRL* demonstrates strong performance, analyzing its failures reveals its current limitations. We use a representative case testing with different base models (Fig. 5) to illustrate this boundary. In this case, *WIRL* must resolve three elements: *"url1"*, *"user"*, and *"password"*. It successfully maps the simpler elements, *"user"* and *"password"*, to their corresponding context variables, *"properties.getUser()"* and *"properties.getPassword()"*. This highlights its proficiency in handling direct mappings. The failure occurred with the element *"url1"*. Its correct substitution is not a simple variable but a deeply nested chain of method invocations. Constructing this requires advanced, multi-step compositional reasoning, a known frontier challenge for current LLMs and agent systems. The model needs to synthesize a complex code structure rather than just perform a direct mapping. This failure across various base models indicates that the limitation lies not only in the LLM's capabilities but also in the current agent-tool interaction paradigm, which is less effective for tasks requiring the synthesis of complex, multi-step code from scratch.

G. RQ2: Time Efficiency

In this research question, we examine the time efficiency of *WIRL* compared to other baseline approaches on the $CW\,Evaluation$. The comparison results are summarized in Table III, where the last column reports the average execution time per *code wiring* instance. It is worth noting that the reported time costs of raw LLMs are averaged over five independent runs to ensure robustness and consistency. From Table III, we make the following observations:

- *WIRL* required only marginally more time (approximately 0.6 seconds) to generate a recommendation compared to *Qwen2.5-Coder-14B*, and it demonstrated greater time efficiency than all other evaluated LLMs except *GPT-4o-mini*. This efficiency can be attributed to its use of a relatively compact LLM (14B parameters), whereas other models incur longer response times due to their significantly larger parameter sizes. For example, *DeepSeek-V3*, with 671B parameters, exhibited the highest latency, averaging 11.5 seconds per recommendation.
- Although *IDEA* and *ExampleStack* achieved superior performance in terms of time cost, they fall short in ensuring recommendation accuracy. We argue that *WIRL* offers a more practical trade-off between recommendation quality

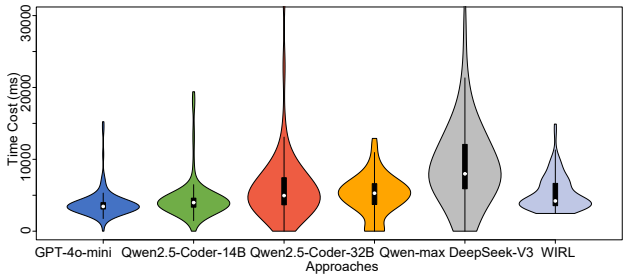

Fig. 6. Time Cost Distribution

Fig. 7. Token Number Distribution

and acceptable latency, making it well-suited for real-world development workflows.

- Code editing methods, such as *GrACE* and *CoEdPilot*, exhibit moderate time efficiency, placing them between the fast, heuristic-based methods and the slower, raw LLM baselines. This performance is attributable to their reliance on pretrained language models, which are more computationally intensive than simple heuristics.

To gain deeper insights into the time cost distribution of *WIRL* and the raw LLMs, we visualized the results using a violin plot, as shown in Fig. 6. The plot reveals that *WIRL* is capable of handling the majority of *code wiring* instances in under 5 seconds. Its median time cost is 4,201.5ms, which is comparable to that of *Qwen2.5-Coder-14B* (3,977.5ms), and substantially lower than that of *DeepSeek-V3* (7,983.5ms). This demonstrates *WIRL*'s ability to deliver high-quality recommendations with competitive latency.

H. RQ3: Token Consumption and Monetary Cost

In this research question, we evaluate the token usage and monetary cost of *WIRL* compared to other LLM-based baselines on the $CWEvaluation$. The comparison results are summarized in Table IV. Columns 2-4 report the average number of input tokens, output tokens, and total tokens per code wiring instance, respectively, while the last column shows the corresponding average monetary cost. All figures are averaged over five independent experimental runs.

As shown in Table IV, *WIRL* consumes slightly more tokens than *Qwen2.5-Coder-14B*, with an average increase of 237.3 input tokens and 80.2 output tokens. This results in a marginally higher monetary cost, i.e., approximately 0.0001 USD more per instance. Surprisingly, although *GPT-4o-mini* incurs the lowest monetary cost, its performance is relatively poor (refer Table III). The increase in token usage is attributed to *WIRL*'s iterative design, where it progressively collects and integrates contextual information into the prompt before generating a final recommendation. As the output from one iteration is passed as input to the next, the cumulative token count naturally exceeds that of the single-pass raw LLMs.

To further examine the distribution of token consumption for *WIRL* and the LLM baselines, we present a violin plot in Fig. 7. The results indicate that *WIRL* handles the majority of cases using fewer than 2,500 tokens, with a median token count of 2,306.5 per code wiring instance, which is comparable to that of *Qwen2.5-Coder-14B* having a median of 1,962.5 tokens. Additionally, both *WIRL* and *Qwen-max* exhibit stable token usage, with minimal outlier points. Notably, *Qwen-max* shows consistent performance in both time and token consumption, largely because it omits cases that exceed its maximum token limit.

I. RQ4: Ablation Study

To investigate the impact of each design choice on overall performance, we conducted an ablation study by creating several variants of *WIRL*. We first individually removed the three types of customized tools, *Locator*, *Collector*, and *Completer*, to assess their respective contributions (Rows 3-5 in Table V). Next, we evaluated the impact of key architectural components. We replaced the state machine with a fully agent-guided exploration strategy and individually removed the filtering tools (i.e., reserve_type_compatible_ones and sort_by_literal_similarity), and the retrieval tools (retrieve_identical_function_call and get_method_names). From Table V, we make the following observations:

- **Tool Contributions:** Three tool types contribute differently to *WIRL*'s performance. The *locator* tools are the most critical, followed by the *completer* and then the *collector* tools. The paramount importance of the *locator* tools is because both the *collector* and *completer* depend on the precise location information they provide. Without *locator*, the system effectively degrades to a raw LLM.
- **Architectural Necessity:** All architectural components are integral to the overall performance. The data clearly shows that removing any component, the state machine, filtering tools, or retrieval tools, results in a performance degradation. This highlights their crucial roles in providing valid contextual information to the agent. While the time efficiency remains stable across most variants, the removal of the state machine presents a notable exception.
- **State Machine Efficiency:** The moderate latency increase observed when removing the state machine is a direct result of a proactive design choice to prevent aimless exploration, a common pitfall in agentic systems. We constrained the maximum iteration number to two, which limits the observable latency penalty in this study. Without this constraint, removing the state machine would lead to prolonged, multi-

TABLE IV
TOKEN CONSUMPTION, AND MONETARY COST

LLMs	Avg. #Input Tokens	Avg. #Output Tokens	Avg. #Total Tokens	Avg. Monetary Cost (USD)
GPT-4o-mini	2,769.4	27.4	2,796.8	0.0004
Qwen2.5-Coder-14B	2,769.4	28.3	2,797.7	0.0008
Qwen2.5-Coder-32B	2,769.4	43.8	2,813.2	0.0008
Qwen-max	2,769.4	24.9	2,794.3	0.0010
DeepSeek-V3	2,769.4	22.0	2,791.4	0.0008
WIRL	3,006.7	108.5	3,115.2	0.0009

TABLE V
ABLATION STUDY

Variants	#EM	#Rec	EM_P	EM_R	Latency
WIRL	**199**	**217**	**91.7%**	**90.0%**	5,255.6 ms
w/o Locator	110	157	70.1%	49.8%	**4,624.5 ms**
w/o Collector	169	190	88.9%	76.5%	5,048.2 ms
w/o Completer	146	189	77.2%	66.1%	4,827.1 ms
w/o State Machine	198	217	91.2%	89.6%	7,297.1 ms
w/o Filtering Tools	179	205	87.3%	81.0%	5,679.7 ms
w/o Retrieval Tools	189	216	87.5%	85.5%	5,820.3 ms

step exploration cycles and a significantly more substantial, and likely unacceptable, increase in latency.

The ablation study results confirm that each component is integral to *WIRL*'s performance and the architecture designs are also the key contributors to the efficiency of *WIRL*.

V. DISCUSSION

A. Placing WIRL in the Broader Context of Code Adaptation

Despite its specific focus, code wiring represents a frequent, critical, and foundational aspect of code adaptation. This claim is supported by two key factors:

- **Prevalence in Adaptations:** The empirical study by Zhang et al. [3] on real-world code adaptations provides strong evidence for this. Their study revealed that a remarkable 88.0% of code reuses require adaptation. Through our further analysis, over one quarter (27.3%) of those adaptations involve code wiring operation, underscoring its high prevalence. Additionally, the study highlights that variable references and method invocations are central to the adaptation process, with 77.2% of statement-level code adaptations involving method invocations or variable references.
- **Task Complexity:** While direct variable-to-variable mapping is relatively straightforward, the core challenge lies in mapping a variable to a complex expression, such as a deeply nested method invocation. This elevates the task from simple pattern matching to one requiring deeper program semantic understanding. A complex example is presented in Fig. 5 and the analysis is presented in Section IV-F5.

B. Applicability Analysis

We evaluate WIRL's real-world applicability across three key dimensions:

- **Accuracy:** With a precision of 91.7%, WIRL significantly reduces the developer's cognitive load by turning a tedious and error-prone manual adaptation process into a simple inspection and approval task in nine out of ten cases. This accuracy substantially outperforms the 71.8% of the previous state-of-the-art approach [9] in code snippet adaptation.
- **Latency:** WIRL introduces a minimal computational overhead of only 0.6 seconds on top of the base model's inference time. The average 5.25-second recommendation time, inflated by our experimental setup using online API calls, would be significantly reduced in a production environment with a locally deployed model or a dedicated API, making it well-suited for practical development workflows. To support self-hosted models or private APIs, the implementation of *WIRL* [30] allows for the configuration of a custom endpoint URL via the IDEA settings panel at File - Settings - Tools - WIRL AI Settings.
- **Cost:** Economically, WIRL is highly efficient, with an average cost of just $0.0009 per recommendation based on public API pricing. This negligible cost yields a high return on investment by saving valuable developer time.

C. Dataset Representativeness Analysis

Although our dataset, $CWEvaluation$, is modest in size, its quality and representativeness are justified by:

- **Ecological Validity:** $CWEvaluation$ is derived from genuine code reuse instances from the SOTorrent database filtered and validated by Zhang et al. [3], ensuring it reflects authentic development practices.
- **Project and Domain Diversity:** It includes 29 distinct open-source projects from a wide range of domains, e.g., foundational libraries and popular mobile applications, enhancing the generalizability of our findings.
- **Structural Diversity:** The code snippets' structural properties, e.g., LOC, are comparable to those in larger datasets [3], and a distributional analysis can be found in our online website [31]. Besides, $CWEvaluation$ covers both file-level and project-level adaptation scenarios, ensuring its diversity and complexity.

D. Versatility Analysis

WIRL is designed to be language-agnostic. Its core LLM-based reasoning and agentic control flow are language-independent. To adapt WIRL to a new language, only two

language-specific components in its toolkit require modification: the *Locator*, which uses compiler information to find unresolved variables, and the *Collector*, which leverages Abstract Syntax Tree (AST) analysis to gather contextual information. This modular design ensures high generalizability.

E. Threats to Validity

The primary threat to **external validity** is the dataset's size. We mitigate this by using a public, manually inspected dataset [3] and by open-sourcing our data and results [23] to encourage replication. A threat to **construct validity** is potential bias in identifying *code wiring* instances. To address this, two authors independently inspected the data, achieving a high inter-rater agreement (Cohen's kappa of 0.81), with all discrepancies resolved through discussion.

F. Limitations

WIRL has three main limitations. First, in the current implementation, if WIRL's analysis does not identify any suitable existing variables with high confidence, it will not make a replacement recommendation. Instead, it will inform the developer that no suitable match was found, implicitly guiding them to declare a new variable. For future work, we plan to enhance WIRL to not only detect this scenario but also suggest contextually appropriate names and types for the new variable declaration. Second, our dataset, $CWEvaluation$, lacks more complex wiring scenarios, such as those with semantic ambiguities or many-to-many mappings, due to the laborious nature of manual collection. We plan to develop a semi-automated approach to expand the dataset in the future. Finally, *WIRL* is currently implemented and evaluated only for Java, and extending it to other languages remains future work.

VI. Related Work

A. Code Snippet Adaptation

Research into code snippet adaptation began with empirical studies classifying the reuse and modification patterns developers employ when integrating external code [3], [4], [32]. Early automated tools like *CSNIPPEX* [33] and *NLP2Testable* [34] addressed unresolved variables by declaring them with default values, a solution insufficient for complex, context-dependent adaptations. A more significant body of work leverages code clones and existing examples. Tools such as *SniptMatch* [35], *ExampleStack* [16], *CCDemon* [36], *MICoDe* [37], and *EUKLAS* [38] recommend adaptations by matching snippets to local or online code repositories. However, their reliance on finding similar, previously adapted code limits their applicability to novel scenarios. Another strategy frames adaptation as a de-obfuscation task, masking and recovering identifiers using context-aware models [39], [40]. Other tools address related but distinct goals, such as assessing code compatibility [41] or converting snippets into methods [42].

In contrast, *WIRL* focuses on resolving unresolved variables in new contexts without depending on prior examples.

B. Code Editing

Recent advancements in code editing also inform our approach. Some models are pre-trained specifically for code modification, such as CoditT5 [43], which models edits directly, and CCT5 [44], which learns from code change datasets. Other methods leverage edit history and structural context: C3PO [45] represents edits as paths in the Abstract Syntax Tree, while Overwatch [46] learns from sequences of developer actions. More advanced LLM-based approaches like GrACE [7] and CoEdPilot [8] condition edit generation on prior associated edits to better capture developer intent.

While powerful, these methods differ from *WIRL* in two key ways. First, the edit location of code wiring problem is often a deterministic problem guided by compiler errors, whereas general code editing involves less predictable changes across multiple files. Second, advanced code editing approaches are dependent on a rich history of prior edits, information that is inherently scarce in the context of code wiring.

C. LLM-based Agents for Software Engineering

LLM-based agents are increasingly used to automate complex software engineering tasks through iterative reasoning and tool use [47]–[49]. They have been successfully applied to code generation and testing [50]–[53], code maintenance and repair [54]–[56], and automated issue resolution [57].

Despite these advancements, no prior work addresses context-aware code wiring for pasted snippets. *WIRL* fills this gap as the first framework to integrate agent-based planning and a customized toolkit for automated variable resolution.

VII. Conclusions and Future Work

We introduce *WIRL*, a novel LLM-based agent designed to facilitate context-aware code wiring, an essential but underexplored aspect of the copy-paste-modify paradigm in software development. Unlike prior approaches , *WIRL* keeps a dynamically updated prompt to collect and record useful context information through a hybrid architecture that combines structured tool invocation with autonomous reasoning capabilities of LLMs. By formulating the adaptation task as an RAG infilling problem, *WIRL* aligns closely with the natural strengths of LLMs in code completion. Our evaluation demonstrates that *WIRL* significantly outperforms commercial IDEs, advanced code editing approaches, and advanced LLMs.

Looking ahead, we envision *WIRL* as a stepping stone toward more general-purpose adaptation agents that can support a broader range of integration and transformation tasks.

VIII. Data Availability

The replication package, including tools and data, is publicly available in [23].

Acknowledgment

The authors thank the reviewers for their insightful comments and constructive suggestions. This work was partially supported by the National Natural Science Foundation of China (62232003 and 62172037).

REFERENCES

[1] "StackOverflow," 2025. [Online]. Available: https://stackoverflow.com

[2] "GitHub," 2025. [Online]. Available: https://github.com

[3] T. Zhang, Y. Lu, Y. Yu, X. Mao, Y. Zhang, and Y. Zhao, "How do Developers Adapt Code Snippets to Their Contexts? An Empirical Study of Context-Based Code Snippet Adaptations," *IEEE Transactions on Software Engineering*, vol. 50, no. 11, pp. 2712–2731, 2024.

[4] J. Chen, V. Tech, and V. Tech, "How Do Developers Reuse StackOverflow Answers in Their GitHub Projects ?" *2024 39th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pp. 146–155, 2024.

[5] "IntelliJ IDEA," 2025. [Online]. Available: https://www.jetbrains.com/idea/

[6] "ExampleStack-Artifact," 2025. [Online]. Available: https://github.com/tianyi-zhang/ExampleStack-ICSE-Artifact

[7] P. Gupta, A. Khare, Y. Bajpai, S. Chakraborty, S. Gulwani, A. Kanade, A. Radhakrishna, G. Soares, and A. Tiwari, "Grace: Language models meet code edits," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 1483–1495. [Online]. Available: https://doi.org/10.1145/3611643.3616253

[8] C. Liu, Y. Cai, Y. Lin, Y. Huang, Y. Pei, B. Jiang, P. Yang, J. S. Dong, and H. Mei, "Coedpilot: Recommending code edits with learned prior edit relevance, project-wise awareness, and interactive nature," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 466–478. [Online]. Available: https://doi.org/10.1145/3650212.3652142

[9] T. Zhang, Y. Yu, X. Mao, S. Wang, K. Yang, Y. Lu, Z. Zhang, and Y. Zhao, "Instruct or Interact? Exploring and Eliciting LLMs' Capability in Code Snippet Adaptation Through Prompt Engineering," 2024. [Online]. Available: https://arxiv.org/abs/2411.15501

[10] "Code Adaptation Dataset," 2025. [Online]. Available: https://figshare.com/s/741503454b274cd85094

[11] "Stack Overflow URL for Motivating Example," 2025. [Online]. Available: https://stackoverflow.com/questions/668952/the-simplest-way-to-comma-delimit-a-list/669165#669165

[12] "GitHub URL for Motivating Example," 2025. [Online]. Available: https://github.com/CMPUT301W15T10/301-Project/blob/master/src/com/cmput301/cs/project/models/Claim.java

[13] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," pp. 1–33, 2023. [Online]. Available: http://arxiv.org/abs/2210.03629

[14] S. Baltes, L. Dumani, C. Treude, and S. Diehl, "SOTorrent: Reconstructing and analyzing the evolution of stack overflow posts," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018, pp. 319–330.

[15] "Zenodo," 2025. [Online]. Available: https://zenodo.org/

[16] T. Zhang, D. Yang, C. Lopes, and M. Kim, "Analyzing and Supporting Adaptation of Online Code Examples," in *Proceedings - International Conference on Software Engineering*, vol. 2019-May. IEEE, 2019, pp. 316–327.

[17] "GPT-4o-mini," 2025. [Online]. Available: https://platform.openai.com/docs/models/gpt-4o-mini

[18] "Qwen2.5-coder-14b," 2025. [Online]. Available: https://bailian.console.aliyun.com/?tab=model#/model-market/detail/qwen2.5-coder-14b-instruct

[19] "Qwen2.5-coder-32b," 2025. [Online]. Available: https://bailian.console.aliyun.com/?tab=model#/model-market/detail/qwen2.5-coder-32b-instruct

[20] "DeepSeek-V3," 2025. [Online]. Available: https://bailian.console.aliyun.com/?tab=model#/model-market/detail/deepseek-v3

[21] "Qwen-max," 2025. [Online]. Available: https://bailian.console.aliyun.com/?tab=model#/model-market/detail/qwen-max?modelGroup=qwen-max

[22] T. Wang, H. Liu, Y. Zhang, and Y. Jiang, "Recommending variable names for extract local variable refactorings," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 6, Jul. 2025. [Online]. Available: https://doi.org/10.1145/3712191

[23] "WIRL," 2025. [Online]. Available: https://github.com/Michaelll123/WIRL

[24] D. Z. Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, "SELF-CONSISTENCY IMPROVES CHAIN OF THOUGHT REASONING IN LANGUAGE MODELS," *ICLR*, vol. 98004, pp. 1–18, 2023.

[25] "IntelliJ PlatForm Plugin SDK," 2025. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/welcome.html

[26] "Langchain4J," 2025. [Online]. Available: https://docs.langchain4j.dev/intro/

[27] J. Li, J. Xu, S. Huang, Y. Chen, W. Li, J. Liu, Y. Lian, J. Pan, L. Ding, H. Zhou, Y. Wang, and G. Dai, "Large language model inference acceleration: A comprehensive hardware perspective," 2025. [Online]. Available: https://arxiv.org/abs/2410.04466

[28] X. Zhu, J. Li, Y. Liu, C. Ma, and W. Wang, "A survey on model compression for large language models," *Trans. Assoc. Comput. Linguistics*, vol. 12, pp. 1556–1577, 2024. [Online]. Available: https://doi.org/10.1162/tacl_a_00704

[29] "Program Structure Interface," 2025. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/psi.html

[30] "WIRL Implementation," 2025. [Online]. Available: https://github.com/Michaelll123/WIRL-With-License

[31] "LOC Distributional Analysis," 2025. [Online]. Available: https://github.com/Michaelll123/WIRL/blob/master/LOCAnalysis.png

[32] Y. Wu, S. Wang, C. P. Bezemer, and K. Inoue, "How do developers utilize source code from stack overflow?" *Empirical Software Engineering*, vol. 24, no. 2, pp. 637–673, 2019.

[33] V. Terragni, Y. Liu, and S. C. Cheung, "CSNIPPEX: Automated synthesis of compilable code snippets from Q&A sites," in *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 118–129.

[34] B. Reid, C. Treude, and M. Wagner, "Optimising the fit of stack overflow code snippets into existing code," *GECCO 2020 Companion - Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, no. 1, pp. 1945–1953, 2020.

[35] D. Wightman, Z. Ye, J. Brandt, and R. Vertegaal, "Snipmatch: using source code context to enhance snippet retrieval and parameterization," in *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 219–228. [Online]. Available: https://doi.org/10.1145/2380116.2380145

[36] Y. Lin, X. Peng, Z. Xing, D. Zheng, and W. Zhao, "Clone-based and interactive recommendation for modifying pasted code," in *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*, 2015, pp. 520–531.

[37] Y. Lin, G. Meng, Y. Xue, Z. Xing, J. Sun, X. Peng, Y. Liu, W. Zhao, and J. Dong, "Mining implicit design templates for actionable code reuse," in *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 394–404.

[38] C. Dörner, A. R. Faulring, and B. A. Myers, "EUKLAS: Supporting copy-and-paste strategies for integrating example code," in *PLATEAU 2014 - Proceedings of the 2014 ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools, Part of SPLASH 2014*, 2014, pp. 13–20.

[39] M. Allamanis and M. Brockschmidt, "SmartPaste: Learning to Adapt Source Code," 2017. [Online]. Available: https://arxiv.org/abs/1705.07867

[40] X. Liu, J. Jang, N. Sundaresan, M. Allamanis, and A. Svyatkovskiy, "AdaptivePaste: Intelligent Copy-Paste in IDE," 2023, pp. 1844–1854.

[41] R. Cottrell, R. J. Walker, and J. Denzinger, "Jig saw: A tool for the small-scale reuse of source code," in *Proceedings - International Conference on Software Engineering*, 2008, pp. 933–934.

[42] V. Terragni and P. Salza, "APIzation: Generating Reusable APIs from StackOverflow Code Snippets," in *Proceedings - 2021 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021*. IEEE, 2021, pp. 542–554.

[43] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "Coditt5: Pretraining for source code and natural language editing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556955

[44] B. Lin, S. Wang, Z. Liu, Y. Liu, X. Xia, and X. Mao, "CCT5: A code-change-oriented pre-trained model," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 1509–1521. [Online]. Available: https://doi.org/10.1145/3611643.3616339

[45] S. Brody, U. Alon, and E. Yahav, "A structural model for contextual code changes," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 215:1–215:28, 2020. [Online]. Available: https://doi.org/10.1145/3428283

[46] Y. Zhang, Y. Bajpai, P. Gupta, A. Ketkar, M. Allamanis, T. Barik, S. Gulwani, A. Radhakrishna, M. Raza, G. Soares, and A. Tiwari, "Overwatch: learning patterns in code edit sequences," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, pp. 395–423, 2022. [Online]. Available: https://doi.org/10.1145/3563302

[47] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, and H. Chen, "From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future," 2024. [Online]. Available: https://arxiv.org/abs/2408.02479

[48] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin *et al.*, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, 2024.

[49] X. Chen, X. Hu, Y. Huang, H. Jiang, W. Ji, Y. Jiang, Y. Jiang, B. Liu, H. Liu, X. Li, X. Lian, G. Meng, X. Peng, H. Sun, L. Shi, B. Wang, C. Wang, J. Wang, T. Wang, J. Xuan, X. Xia, Y. Yang, Y. Yang, L. Zhang, Y. Zhou, and L. Zhang, "Deep learning-based software engineering: progress, challenges, and opportunities," *SCIENCE CHINA Information Sciences*, vol. 68, no. 1, p. 111102, 2025. [Online]. Available: https://doi.org/10.1007/s11432-023-4127-5

[50] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, and H. Cui, "AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation," 2024. [Online]. Available: https://arxiv.org/abs/2312.13010

[51] F. Lin, D. J. Kim, Tse-Husn, and Chen, "SOEN-101: Code Generation by Emulating Software Process Models Using Large Language Model Agents," 2024. [Online]. Available: https://arxiv.org/abs/2403.15852

[52] 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: Association for Computing Machinery, 2024, p. 1592–1604. [Online]. Available: https://doi.org/10.1145/3650212.3680384

[53] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024, pp. 50 528–50 652.

[54] F. Batole, D. OBrien, T. N. Nguyen, R. Dyer, and H. Rajan, "An llm-based agent-oriented approach for automated code design issue localization," in *ICSE2025: The 47th International Conference on Software Engineering*, April 27-May 3 2025.

[55] I. Bouzenia, P. Devanbu, and M. Pradel, "RepairAgent: An Autonomous, LLM-Based Agent for Program Repair," 2024. [Online]. Available: http://arxiv.org/abs/2403.17134

[56] K. Ke, " NIODebugger: A Novel Approach to Repair Non-Idempotent-Outcome Tests with LLM-Based Agent ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 762–762. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00226

[57] W. Tao, Y. Zhou, Y. Wang, W. Zhang, H. Zhang, and Y. Cheng, "Magis: Llm-based multi-agent framework for github issue resolution," in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024, pp. 51 963–51 993.