CONSTRUCTOR UNIVERSITY

Reasoning for Repository-level Code Editing

by

Alexander Kovrigin

Bachelor Thesis in Computer Science

Submission: September 4, 2024

Supervisor: Timofey Bryksin
Industry Advisor: Aleksandra Eliseeva

Constructor University | School of Computer Science and Engineering

Statutory Declaration

Family Name, Given/First Name	Kovrigin, Alexander
Matriculation number	30006606
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

. . . .

Date, Signature

Abstract

In this work, we explore Large Language Model-based context retrieval strategies for repository-level code editing task. To achieve this goal, we first implemented a novel code editing system that incorporates AI agent features via the renowned LangChain framework and facilitates easy experimentation due to a modular architecture and rich configuration features. Next, we proposed numerous context retrieval strategies to investigate the impact of individual components on both context retrieval and downstream code editing performance. Finally, we ran a series of experiments with the proposed context retrieval strategies. Our findings suggest that agent-based methods for context collection perform better compared to unmodified queries to BM25. In addition, agents with additional reasoning perform more effectively than single-loop agents.

Contents

1 Introduction **1**

2 Statement and Motivation of Research **3**
2.1 Large Language Models . . . 3
2.2 Retrieval-Augmented Generation (RAG) . . . 3
2.3 BM25 . . . 4
2.4 AI Agents . . . 4
2.5 Code Editing with LLMs . . . 6
2.5.1 Non-agentic Code Editing . . . 6
2.5.2 AI Agents for Code Editing . . . 6

3 Description of the Investigation **7**
3.1 Code Editing System . . . 7
3.1.1 Requirements and Tasks . . . 7
3.1.2 Implementation . . . 8
3.1.3 Structure of the code editing system . . . 8
3.1.4 Implemented Code Editors . . . 9
3.2 Context Retrieval Variations . . . 9
3.2.1 Unmodified BM25 . . . 10
3.2.2 Agent . . . 10
3.2.3 Agent with Fixed Context Size . . . 10
3.2.4 Agent with Self-Correction . . . 11
3.3 Editing Component . . . 11

4 Evaluation of the Investigation **12**
4.1 Datasets . . . 12
4.2 Metrics . . . 13
4.2.1 Context Retrieval Metrics . . . 13
4.2.2 Code Editing Metrics . . . 14
4.3 Hyperparameters . . . 15
4.4 Results . . . 15
4.4.1 Context Retrieval . . . 15
4.4.2 Code Editing . . . 16

5 Conclusion **17**

A Full Evaluation Results **24**

1 Introduction

Software developers often need to modify existing codebases, either to accommodate evolving requirements, incorporate new features, or rectify bugs. These tasks are far from trivial and require a comprehensive understanding of the internal structure of the project. Consequently, both researchers and practitioners have devoted considerable efforts to exploring methods to automate code editing.

From the machine learning perspective, *code editing* can be formulated as automatically implementing changes for the existing project based on the given natural language instruction. Popular examples of real-world tools with such capabilities include GitHub Copilot [1] and Amazon CodeWhisperer [2]. From the research side, the most notable code editing approaches involve Large Language Models (LLMs), which play an essential role in the automation of many code-related tasks [5, 9, 16, 19, 26, 29, 34, 53].

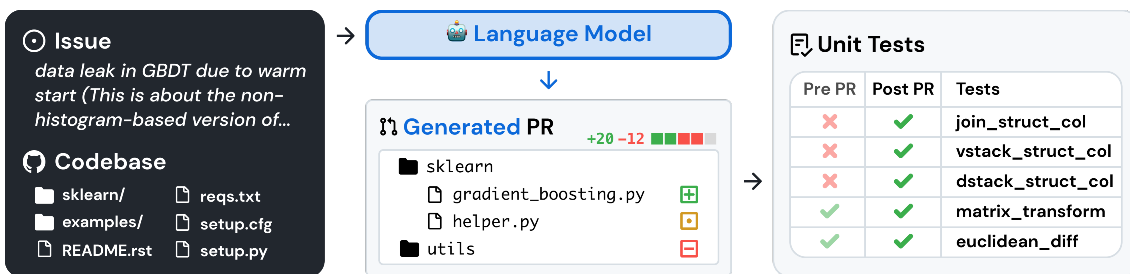

Figure 1: An overview of Code Editing task. The figure is taken from Jimenez et al. [20].

When modifying a function, developers must consider all instances of its usage in the code, a hard task due to the code dependencies and implicit code contracts. This highly intertwined nature of real-world projects presents difficulties for many automated systems, including LLMs, due to their context length limitation. However, in industrial software development, engineers often work on extensive projects with thousands of code files, each encompassing a large number of lines.

Consequently, several researchers are exploring the use of AI agents for code editing tasks [28, 33, 41, 54]. Moreover, in concurrent research, by incorporating Retrieval-Augmented Generation (RAG) and code search capabilities, developers equip AI agents to progressively edit the code [4, 7, 20, 25, 38, 47, 55].

The common implementation of AI agents for code editing includes two components:

Context Retrieval collects relevant code based on the developer's instruction. The use of LLMs is not mandatory here.

Editing modifies the relevant code fragments as per the developer's instruction. Given the complexity of this task, LLMs' coding and reasoning capabilities become essential.

Context retrieval, while it is arguably the simpler component, is still crucial for the overall performance of the system. Intuitively, if a vital code section isn't identified as the relevant context, then it is not possible to correctly execute the code editing. Similarly, if many sections of irrelevant code are retrieved, it increases the possibility of introducing irrelevant changes that might break the code. The experimental results of Jimenez et al. [20] also confirm that context is crucial for the end performance of code editing systems, with

oracle retrieval significantly increasing the number of correctly resolved GitHub issues for all models compared to simpler BM25 retrieval. The results of Phan et al. [30] align with these conclusions for a slightly different subset of coding tasks.

Therefore, it is of utmost importance to improve the quality of the context retrieval component of code editing systems. Although the code editing task has recently gained increased attention from the research community, the latest works focus on end-to-end AI agent code editing solutions, without careful consideration for the specific parts of the pipeline. Therefore, in this work, we pose the following research question with the aim of providing a deeper insight into the use of AI agents for code editing:

RQ: Does reasoning in agentic context retrieval improve code editing performance?

In this work, we study the different LLM-based agentic context retrieval options and how they affect the overall code editing process. Specifically, despite their coding ability, LLMs are notorious for hallucinations and inconsistencies [17]. Therefore, small changes in prompting and tool implementation can significantly change the quality of context retrieval and, consequently, the downstream code editing predictions. Reasoning and planning methods are universal and can be applied both to the context retrieval component (if implemented using agents) and to the editing component.

To answer our research question, we have formulated the following objectives for our research.

Propose context retrieval strategies Despite using the same retrieval mechanism, the context collection can be varied by altering the search queries and the amount of information retrieved. Hence, we experiment with various non-agentic (baseline) and agentic state-of-the-art approaches.

Experiment setup To facilitate our research, we needed to implement an experiment-focused code editing system. Unlike concurrent researchers, we have utilized the renowned *LangChain* [8] framework, which makes our system production-ready and easily extensible by other researchers. We have made our code publicly available on GitHub [1].

Define metrics and evaluate search strategies To assess the quality of agents' predictions, we have established an evaluation methodology for analyzing the predictions, as is done in other code editing systems evaluations [7, 20]. To answer our research question, we evaluated different context retrieval setups, including LLM and non-LLM methods, and experimented with applying reasoning and planning to the context retrieval component.

This work is structured as follows. Section 2 introduces the concepts used throughout the work and the subject area. Section 3 outlines our proposed approaches, our code editing system, and the setup of the experiments. Section 4 presents the evaluation methodology and results. Section 5 describes the conclusions of our work.

[1] https://github.com/JetBrains-Research/ai-agents-code-editing

2 Statement and Motivation of Research

In this section, we introduce all the necessary concepts and explain in detail how modern AI agents for coding tasks work.

2.1 Large Language Models

Language modeling is a crucial concept in the field of natural language processing (NLP). In general, language models (LMs) aim to estimate the likelihood of word sequences to predict the probabilities of future (or missing) tokens [57]. They are used in many NLP applications. Research on language models has progressed from statistical n-gram models [52], through Recurrent Neural Networks (RNNs) [21], to Transformer-based models [42].

Large Language Models (LLMs) are language models with billions of parameters, trained on a large amount of text data [6, 57]. Most LLMs are based on the Transformer architecture that was first introduced in the highly influential work of Vaswani et al. [42]. These models can generate human-like texts and perform downstream tasks such as translation, question answering, and summarization, making them versatile tools in the NLP area [22].

Furthermore, their remarkable capabilities extend beyond the text domain. By training LLMs on code, it has been shown that they achieve commendable scores for many coding tasks [9, 48], commonly outperforming any other non-Transformer approach.

2.2 Retrieval-Augmented Generation (RAG)

The quadratic complexity of the attention mechanism, one of the key components of the Transformer architecture, limits the maximum context length that Transformers can process [42]. Moreover, even models that support long input contexts show quality degradation when using information from the middle of the context [27].

Thus, Retrieval-Augmented Generation (RAG) has been developed to tackle this issue. It incorporates task-related knowledge from external databases into the LLM context on demand, allowing one to improve the generation quality without the need to load the entire database into the model's context. RAG has seen widespread adoption, becoming a key technology to improve the suitability of LLMs for real-world applications. The basic RAG process involves indexing, retrieval, and generation [12]:

- **Indexing** is performed by cleaning and converting the raw data into plain text format. To handle the language model's context size limitations, the text is broken down into chunks (*documents*) of manageable fixed length.
- **Retrieval** is performed by calculating similarity scores between the user query and the indexed documents, returning the k documents with the highest similarity.
- **Generation** is performed by feeding the chosen documents (*context*) and the initial task as input for the LLM. The model responds using its own knowledge and the information contained in the retrieved documents.

Retrieval for code tasks has been a major research area, driven by its tangible real-world applications [10]. The primary difficulty in efficient code retrieval is finding a semantic connection between natural language descriptions and code fragments. Numerous

approaches have been developed, including both classical ML and deep learning methods [14, 15, 18, 51].

2.3 BM25

Due to the long length of code documents, retrieval with dense methods (using latent space representations of queries and texts to construct semantic matching functions for relevance modeling) [24] is mostly ill-suited for repository-level tasks, given the high computational costs. Thus, we adopted the BM25 algorithm for our purposes, aligned with Jimenez et al. [20].

BM25, also known as Okapi BM25, is a ranking function based on the bag-of-words model, used in information retrieval to assess how relevant documents are to a specified search query [35, 40]. It ranks documents based on query terms, regardless of their position in the document. More precisely, BM25 is a family of scoring functions with variations in components and parameters. One of its forms found in Trotman et al. [39] and implemented in the `rank-bm25` Python package is defined in Equation 1.

$$rsv_q = \sum_{t \in q} \log\left(\frac{N}{df_t}\right) \cdot \frac{(k_1 + 1) \cdot tf_{td}}{k_1 \cdot \left(1 - b + b \cdot \frac{L_d}{L_{avg}}\right) + tf_{td}} \tag{1}$$

For a query q, the retrieval status value rsv_q is calculated as the sum of scores for each term t. Here, N represents the total number of documents, df_t the documents containing term t, and tf_{td} the frequency of term t in document d. L_d and L_{avg} denote the length of a document and the average document length, respectively.

2.4 AI Agents

Historically rooted in philosophical discussions, an *agent* generally refers to an entity with the capacity to act. For a more rigorous definition, within the area of Reinforcement Learning (RL), agents interact with an environment through observations and actions. Each interaction involves the agent receiving an input that indicates the current state of the environment and then choosing an action that alters the state of the environment [23].

Humanity has long pursued AI that matches or surpasses human intelligence, considering AI agents as a promising approach. The design of adaptable AI agents requires a versatile model. LLMs, known for their advanced reasoning [56], are viewed as potential catalysts for Artificial General Intelligence (AGI). LLMs have yielded significant results in the development of AI agents [46].

An AI agent commonly includes three components, as shown in Figure 2.

- **Tools**: Humans often use tools to perform tasks that exceed our physical and mental capabilities. Similarly, providing LLMs with external tools can greatly enhance the performance of these models [37]. Specialized tools improve the expertise, adaptability, and suitability of LLMs for domain-specific needs [32]. LLM-based agents not only need tools but are also well-suited for their integration. LLMs, enriched with pre-training and Chain-of-Thought (CoT) prompting, have shown significant reasoning and decision-making skills in complex environments [43].

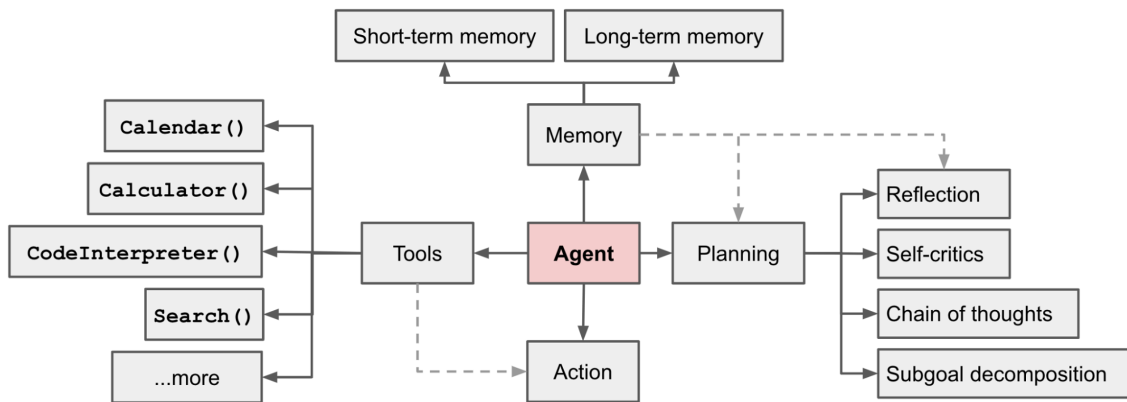

Figure 2: Overview of a LLM-powered autonomous agent system. The figure is taken from Weng [45].

- **Memory**: Agents based on LLM process past interactions in natural language, incorporating previous data into each input. As this data accumulates, it might exceed the capabilities of the Transformer architecture, potentially resulting in the truncation of content. Moreover, as agents accumulate historical observations and actions, the increasing memory load makes it harder to link related topics, possibly leading to misaligned responses. To address these challenges, memory stores environmental information and uses it to guide future actions [46].
- **Reasoning and Planning**: Although LLMs have extraordinary abilities in NLP tasks, they are also known for their hallucinations and inconsistencies [17, 36]. This is why methods have been developed to improve the reliability and logical consistency of the outputs. Reasoning refers to thinking about something systematically and logically to reach a conclusion or make a decision [56]. Planning refers to preparing a certain list of actions before actually executing them [17]. These linked concepts aim to improve the agent's output stability and significance, often discussed together in the context of the model's thought structuring abilities. Here are common reasoning and planning techniques:
  - **Chain-of-Thought (CoT)**: This approach uses a few 'chain of thought' examples, which are intermediate reasoning steps in natural language, requested by LLMs. In doing so, the model learns to provide a clear rationale before the final answer, significantly improving LLM performance on various reasoning tasks [44].
  - **Tree-of-Thought (ToT)**: a planning method that expands on CoT, using coherent text units (thoughts) as intermediate problem-solving steps. It considers various reasoning paths (chains) using the DFS or BFS algorithm, evaluates choices, and backtracks when needed [49].
  - **Self-Correction**: Language models can occasionally make mistakes during the reasoning process. The tools can correct reasoning errors and improve the accuracy of the answer [56]. The tools use evidence for self-correction of initial outputs [13]. For instance, the LLM's code can be additionally validated through the execution of the program [33].

- **ReAct** [50], unlike CoT, alternates between reasoning (Thought step) and planning (Action step), demonstrating significant improvements in planning capabilities [17].

2.5 Code Editing with LLMs

Given the introduction of LLMs and the paramount practical significance of code editing, it is only natural that this research direction is being investigated in depth.

2.5.1 Non-agentic Code Editing

Some of the recent work on code editing does not involve AI agent approaches.

- **OctoPack** [29], **InstructCoder** [34], **DeepSeekCoder** [16] and **Coeditor** [19] propose novel datasets and fine-tune LLMs for function-level and file-level code editing. These works, although not directly applicable to repository-level code editing, are still of significant importance, as they can be used as editing tools for more complex agent systems.

- **CodePlan** [5] is an approach for repository-level code editing. The authors utilized static analysis for the code editing LLM workflow planning.

  Upon receiving an instruction (e.g., a function definition change), the method analyzes and identifies affected code chunks (e.g., external calls, inheritance). These chunks are added to the planning graph, linked to dependent nodes, and marked as pending.

  Nodes without pending dependencies are offered to a LLM for modification. The prompt for change includes the code chunk, spatial context (other methods in the same class or file), temporal context (the diff from the method's start to its current state), and causal context (reasons for node addition and changes to dependent nodes). The method then repeats the process for new changes.

  The resulting repository is passed to a testing mechanism. The compilation / type check is used for quality assurance. Errors in testing initiate a new iteration of the method, restarting the process.

  This method highlights the significance of static analysis and the benefits of an iterative approach to code editing.

2.5.2 AI Agents for Code Editing

Given the advances in the research related to AI agents, there has been a recent surge of papers that have applied them to code editing. The following represents a selection of such papers.

- **SWE-Agent** [47] is a novel end-to-end agent system for doing code editing and bug repair tailored towards GitHub issues. They use directory substring search for context retrieval, incorporate a linter for quality assurance, and employ a separate IDE-like editor for editing. It achieves commendable results in the evaluation benchmarks, showing the potential of AI agents code editing.

- **AutoCodeRover** [55] also proposes an agent system to solve Github issues. Like this concurrent work, the authors focus on the context retrieval for the code editing task to improve the downstream results. They employ a program representation (abstract syntax tree) rather than viewing a software project as just a collection of files.
- **AutoDev** from Tufano et al. [41] is an automated system for coding tasks. It provides a proof-of-concept solution for various codebase operations, including file editing, retrieval, build processes, execution, testing, and git operations. Its code search functionality is limited to `find`, `grep`, `ls` CLI tools. This work demonstrates practical applications of AI agents in coding.
- **MAGIS**, a multi-agent system, was developed as detailed in Tao et al. [38]. It is a software evolution framework with four distinct agents: Manager, Repository Custodian, Developer, and Quality Assurance Engineer. The framework focuses on agent collaboration in planning and coding. The Repository Custodian uses BM25 for retrieval and LLM for reranking.

In addition, approaches such as Luo et al. [28], Zhang et al. [53, 54] have explored adjacent tasks of repository-level documentation generation and code completion with AI agents.

3 Description of the Investigation

3.1 Code Editing System

3.1.1 Requirements and Tasks

To perform our experiments, we have to develop a code editing system (CES) with the following criteria:

- **AI Agent support**: To streamline experiments with AI agents, a CES must support working with them. This is not an easy task, since (as described in Section 2.4) the AI agents involve multiple complex components such as tool usage, reasoning, and memory. Moreover, there are also practical issues like error handling and working with different LLMs and their input formats.
- **Modular design**: To enable rapid experimentation, it is crucial that components like context retrieval, underlying retrieval mechanism, LLM provider, prompts, etc. can be swapped out interchangeably without modifying the whole system.
- **High extensibility**: To ensure maintainability and extensibility, we need our system to use industry-standard frameworks and tools. This way, future researchers will be able to build on top of our system
- **Multiprocessing**: Since LLM invocation is a notoriously computationally costly process, there should be an option to parallelize computations to ensure faster code editing. However, additional practical complications arise when running too many agents at once, most notably provider-specific issues like OpenAI's rate limit and GPU memory.

Unfortunately, modular design and high extensibility are not present in concurrent research into code editing [47, 55]. This drives us to develop our own CES that has these

qualities.

3.1.2 Implementation

In this section, we describe the code editing system (CES) that we have developed to perform our experiments.

We implemented the system in Python, a popular programming language for ML-related tasks because of its vast package ecosystem.

For code editing, we use LLMs from *OpenAI* and *HuggingFace*. These two companies are the leading LLM providers for proprietary and open-source models, respectively. To interact with these models, we use the corresponding Python packages: `openai` and `transformers`.

To work with AI Agents, we selected the *LangChain* framework, a leading toolkit for agent-related tasks. It simplifies working with LLMs by providing easy primitives for chaining models, creating state graphs, parsing the outputs, and more. This framework allows our code system to be more robust and simpler to maintain and extend.

To make our system easily configurable and facilitate reproducibility, we use the *Hydra* [2] package for configuration management. Due to a modular software design, we can control and swap out different components of the system using a single `yaml` file. This enables rapid experimentation with CES.

These choices make our system adherent to the criteria outlined previously. We have published the code on GitHub[3], making it accessible to researchers to use and extend.

3.1.3 Structure of the code editing system

The common algorithm for running inference via our proposed CES is the following:

1. **Dataset Loading**: A dataset containing the NL instruction and the codebases represented by `git` repositories are parsed. The codebase state is represented by a `git` commit. More details on the structure of the dataset are discussed in Section 4.
2. **Experiment Initialization**: During the CES inference loop, when the worker in the pool picks up a datapoint, the `git` repository is cloned and checked out at the corresponding commit in a temporary directory. Following this, the retrieval component indexes the code files in the repository and builds an index to perform code search queries.
3. **Code Editor Invocation**: The code editor is then called. The implemented variants of code editors are described in Section 3.1.4. A code editor accepts the instruction and the testbed and has to modify it according to the instruction. When the code editor has finished the execution, the predicted `diff` is calculated by the `git diff` command.
4. **Output to a prediction file**: At the end of inference the predictions are saved in a `.jsonl` file. It contains the predicted `diff` patch to be applied to the repository, the lines used for code editing (to evaluate context retrieval), and the data from the

[2] https://hydra.cc

[3] https://github.com/JetBrains-Research/ai-agents-code-editing

dataset (including the ground truth `diff`). The prediction files are then given to the evaluation subsystem to perform a quantitative analysis of the editing.

3.1.4 Implemented Code Editors

Code editors are the core abstraction for inference in our CES. We have developed multiple code editors to evaluate the code editing approaches as a whole.

1. **LLM**: The simplest code editor does not involve AI agents. It does not utilize code search and instead uses *oracle localization* as defined in Jimenez et al. [20]. Oracle localization refers to the perfect context retrieval when the editing component receives precisely the code sections referenced in the ground truth `diff`.

   This code editor, despite its simplicity, has multiple uses for our study. First, it serves as a reference point for context retrieval evaluations. Second, it allows us to evaluate the capabilities of different large language models in a reproducible and equal setting, rendering another use of our CES.

2. **End-to-End Agent**: This code editor is implemented by a single invocation of an agent with tools to view and edit the code. It mirrors the approaches used in the most prominent literature [3, 4, 25, 47]. Current tools include `code-search`, `view-file`, and `edit-file` (refer to Section 3.2.2).

   This method, while seemingly intuitive, complicates the evaluation and tuning of individual components. The intertwined usage of tools further hinders finding the agent's optimal configuration.

   Nevertheless, this code editor is useful for comparisons with ongoing research.

3. **Retrieve & Edit**: In this code editor, as discussed in the Introduction, context retrieval and editing are separated into two consecutive stages. This allows us to make an easier and more interpretable evaluation and experimentation with the context retrieval part.

   This setup currently achieves state-of-the-art results [55] on the renowned code editing benchmark *SWE-Bench* [20], prompting significant scientific interest in this approach.

   The result of the context retrieval is a set of code sections that the editing component should process.

3.2 Context Retrieval Variations

As discussed in the previous sections, the quality of the context retrieval stage is paramount for the downstream quality of code editing predictions.

To collect the *context* of relevant code we utilize retrieval mechanisms discussed in Section 2.2. To produce documents required for indexing, we split every code file into chunks of equal length with a small overlap.

Despite the same retrieval mechanism (BM25, in our case) used under the hood, there are numerous ways to implement the context collection, since one can change the queries used for the search and the number of documents retrieved per search. This is the focus of our investigation.

3.2.1 Unmodified BM25

The simplest baseline context retrieval method that does not involve any use of LLM. It assembles the context by running a single request to the underlying retrieval mechanism for the top documents using the natural language code editing instruction as the query.

Due to the TF-IDF-like mechanism of BM25, as discussed in Section 2.3, the retrieval mechanism is able to find the most relevant documents using rare meaningful tokens.

To make a valid comparison with the other context retrieval variations, we collect the context from the retrieved documents until the total context length has reached k tokens, where k is a hyperparameter.

3.2.2 Agent

Following the setups in the literature, we present the *Agent* context collection. The underlying agent has access to two tools:

- `code_search` tool searches for documents using the underlying retrieval mechanism using the given query. In addition, it accepts `limit` and `offset` integer parameters for additional flexibility for the agent. Returns the documents as code snippets with file path and line numbers added for subsequent use.
- `add_to_context` tool adds a given code snippet to the context. It accepts a file name, start, and end line numbers that the agent has learned from invocations of `code_search`. If some line is added twice, the second addition is ignored. In this setup, once added, a line cannot be removed from the context.

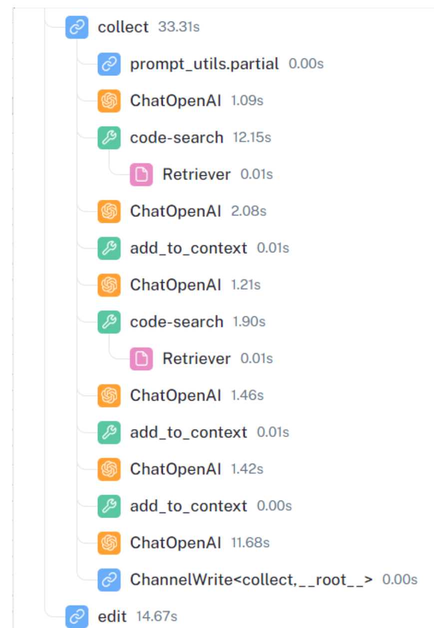

Figure 3: *Agent* Context Retrieval

When invoked, we prompt the agent to collect context for the code editing task using the tools provided. We employ a simple CoT technique to encourage the agent to break down his planned actions before performing them. Our prompt is easily configurable and is available in *LangSmith* [4]. An illustrative example of a typical agent workflow is presented in Figure 3 as a *LangSmith* log.

Once the agent decides to finish the context collection, we take the collected context lines from the `add_to_context` tool and pass them to the editing component.

3.2.3 Agent with Fixed Context Size

In our experiments, we have noted that the simple agent strategy usually collects not enough context in length. This is part of the importance of planning discussed in Sec-

[4] https://smith.langchain.com/hub/jbr-code-editing

tion 2.4. Hence, we propose an *Agent with Fixed Context Size* strategy, where the agent is forced to collect at least k tokens in the context.

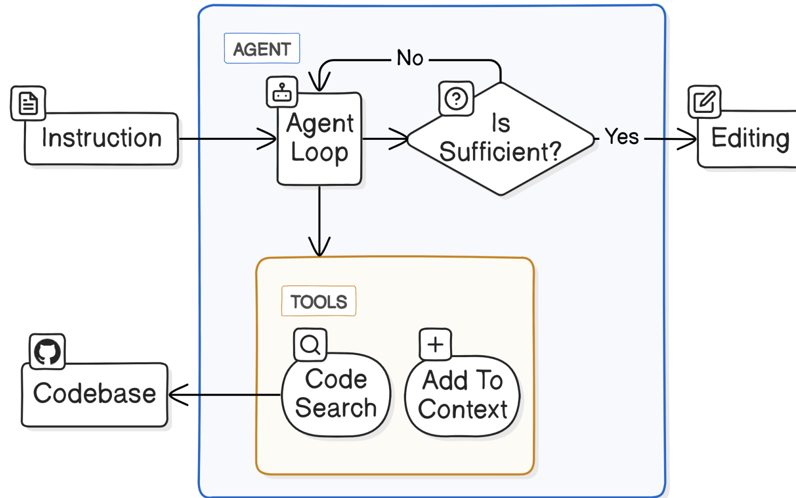

Figure 4: Agent with Review

To enable this strategy, a *review* step is added after the agent completes, checking if the context meets the required token count. If met, it proceeds to the editing subtask; otherwise, the agent is re-invoked until the condition is satisfied. Additionally, the agent prompt explicitly mentions how much context it is required to collect and how much it has collected so far. Obviously, agent memory should make it possible to remember previously seen documents. This setup is depicted in Figure 4.

3.2.4 Agent with Self-Correction

Self-Correction, as described in Section 2.4, is a universal reasoning method that can be applied to numerous areas. We use self-correction in our agent context collector to better reason the context length.

We replace the context length sufficiency check by having the same agent evaluate the collected lines to determine if the context *is sufficient* or if further collection is needed. The decision is made solely by the agent and is based only on its own understanding of the codebase.

3.3 Editing Component

Lastly, we describe the implementation of the editing component of our code editing system. Although it is not the target of our investigation, it is still the essential component to perform inference and evaluate the impact of context retrieval on code editing performance.

The editing component receives a list of lines grouped by files and outputs a `diff` patch file. The simplest way to generate predictions through LLMs would be to generate the entire file, but this would not fit the model context and could lead to hallucinations [17]. We could generate the diffs themselves; however, due to the complexities of the git `diff` format, the model could easily make a mistake, making it necessary to rerun the whole

generation. To combat this issue, we group the lines into consecutive code sections, process the code in these groups, and generate a new version for each of them.

In order to make the changes of different code sections coherent with one another (imagine two API calls to the same method being edited differently), we employ AI Agents once again. Their memory and reasoning components allow us to make better code edits.

The resulting algorithm is the following:

1. Given relevant lines of code in different files, split them into consecutive sections for each file.
2. For every section in every file invoke the AI Agent with the code section and the instructions in the input prompt.
3. After completion of the edit, replace the corresponding section in the original file.
4. When all sections have been processed, run `git diff` at the root of the testbed to generate a patch.

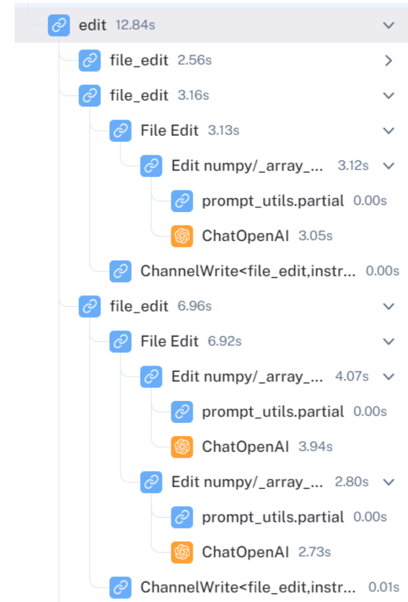

Figure 5: Editing Component Example

4 Evaluation of the Investigation

In this section, we describe the ways in which we evaluated our code editing systems and the data we used to perform these evaluations.

4.1 Datasets

Code editing requires the state of the repository prior to modification and the corresponding instruction. This is typically done by searching GitHub and similar services for meaningful commits in large projects. Each data point should include the following keys:

- `instruction`: Natural language description of the developer's intentions. Typically a Git commit message or GitHub issue body.
- `repo`: Git repository of the commit.
- `hash`: Git commit SHA where the code has been edited. Changes in code from the commit are regarded as the ground truth.
- `tests` (optional): A set of checks that verify the correctness of code edits or issue resolution.

From this data and the `git` tool, we can collect other necessary information for inference, such as true git diff, files at the base commit (before the change), and others.

For an accurate evaluation of the code editing system, it is important to choose data that represents real-world code well. With this in mind, we have used three high-quality code editing datasets in our evaluation.

LCA-CI-Fixing[5] is a dataset for the code repair task collected by JetBrains Research. To construct the instruction, we find the first mention of the `"error"` substring in the logs and give the model 10 lines around that mention with a prompt to fix this error.

LCA-Code-Editing[6] is a dataset for the code editing task with the focus on longer code edits, also collected by JetBrains Research. Figure 6 shows the distribution of the instruction types in the dataset.

SWE-Bench Lite[7] is one of the most frequently reported datasets in the literature. It consists of real-world GitHub issues in Python projects. The Lite version of the SWE-Bench [20] consists of 300 rows, as opposed to 2.3k in the original version.

The detailed breakdown of the datasets used in the investigation is presented in Table 1.

Dataset	Data Points	Context Length (tokens)	Has Tests
`lca-ci-fixing`	144	512	✓
`lca-code-editing`	119	1200	✗
`SWE-Bench_Lite`	300	120	✓

Table 1: The information about the considered datasets.

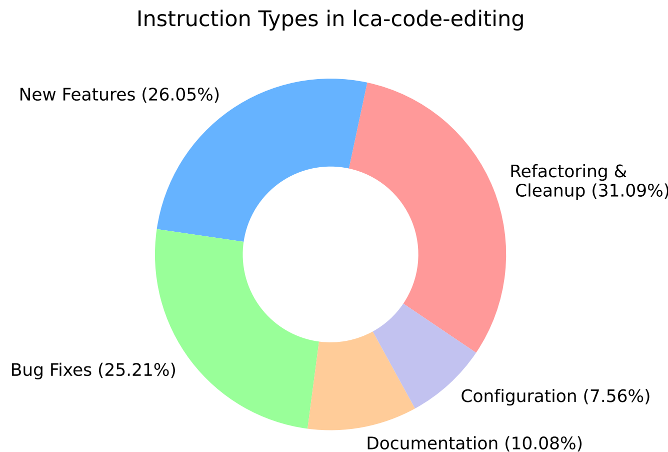

Figure 6: The distribution of instruction types in the `lca-code-editing` dataset.

4.2 Metrics

4.2.1 Context Retrieval Metrics

To evaluate the different context retrieval mechanisms, we can calculate Recall, Precision, and F1 of the collected context, comparing it with the ground truth diff, treating it as a binary classification problem. The objects of this classification can be files, Python

[5] https://hf.co/datasets/JetBrains-Research/lca-ci-fixing
[6] https://hf.co/datasets/JetBrains-Research/lca-code-editing
[7] https://swebench.com

scopes (classes, functions), or lines. The formulas for the Python scope level are shown in Equations 2, 3, and 4.

$$\text{precision} = \frac{|\{\text{relevant scopes}\} \cap \{\text{retrieved scopes}\}|}{|\{\text{retrieved scopes}\}|} \tag{2}$$

$$\text{recall} = \frac{|\{\text{relevant scopes}\} \cap \{\text{retrieved scopes}\}|}{|\{\text{relevant scopes}\}|} \tag{3}$$

$$F_1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{4}$$

4.2.2 Code Editing Metrics

In addition, to evaluating context retrieval, we are also interested in the performance of the downstream code editing task. Hence, to perform a quantitative analysis of our code edits, we employ two groups of metrics: execution-based metrics and text-based metrics.

Execution-based metrics. If the dataset contains tests for each datapoint to verify that the instruction has been followed correctly, then a meaningful metric to calculate is `pass@k`. It represents the percentage of data points where tests pass at least once during k tries. Out of the datasets we considered, covered in Section 4.1, execution metrics are applicable to SWE-bench Lite and LCA-CI-Fixing.

In the literature related to code editing, only `pass@1` or *pass rate* is commonly reported [47, 55]. In addition, it is significantly cheaper in terms of computational resource. This is why we have decided to only calculate the `pass@1` metric.

Text-based metrics. Text-based metrics are calculated by analyzing the similarity of ground truth and predicted changes between diffs or between the resulting code.

We have selected the following metrics:

- **CodeBertScore** from Zhou et al. [59] represents the cosine similarity between predicted and ground truth code in the embedding space. It uses an encoder-only Transformer CodeBERT [11] to produce the embeddings, hence the name.
- **Character n-gram F-score (chrF)** evaluates machine translation by comparing character n-grams between the output and a reference [31].
- **LLM-based Evaluation** can be used to judge the quality of the predictions. The larger the model and the higher its reasoning abilities, the better. [58]. LLM can be used to predict a score for each prediction individually, but comparing predictions in a pairwise fashion is usually more robust. In the latter case, the performance can then be numerically quantifiable by the win rate.

Localization metrics. Since we have a ground truth diff, we are interested in understanding whether the predicted diff references the same files, lines, or Python scopes. To achieve this, we treat the code editing as a binary classification task, where the task is for the git diff to reference the same set of files/lines/scopes. Therefore, we are able to calculate the recall, precision and F1 of this classifier.

4.3 Hyperparameters

For our experiments, we used the BM25 retrieval mechanism. We are using OpenAI GPT-3.5 Turbo as LLM for the AI agent. We have split the code into chunks of 512 tokens with overlap of 128 tokens (as obtained from the GPT-3.5 Turbo tokenizer). This roughly corresponds to 10 lines of code.

We have evaluated the following variations of strategies introduced in Section 3.2: *Unmodified* with 200, 500 total tokens, *Agent*, *Agent Fixed* with 500, and *Agent Self-Correction* for SWE-Bench experiments. For LCA datasets, we have also included 2000 tokens variations.

4.4 Results

4.4.1 Context Retrieval

Strategy	Recall	Precision	F_1	Context Length (tokens)
Baseline				
Unmodified 200	0.14	0.08	0.09	**218**
Unmodified 500	0.17	0.05	0.07	487
Agentic Retrieval				
Agent	0.15	0.11	0.11	**247**
Agent Fixed 500	**0.25**	0.05	0.08	894
Agent Self-Correction	0.18	**0.12**	**0.13**	**253**

Table 2: Context Retrieval Metrics

We present the most notable results in Table 2 — it includes metrics calculated for the `SWE-Bench_Lite` dataset on the Python Scope level. Full results are provided in the Appendix A.

Furthermore, we can examine the distribution of the context lengths collected for the SWE-Bench dataset. Agent Fixed 500 exhibits a higher context length than anticipated due to the sufficiency condition being greater than or equal, with no restrictions on large additions by the model.

We observe that *Agent* and *Agent Self-Correction* collect around 220 tokens on average since they have free choice. It is a relatively small amount of information, and yet quite close to the true average context size of 120 for this dataset, as detailed in Table 1.

Moreover, we note that agentic methods are comparable to non-agentic methods with similar context length. Moreover, *Unmodified* with 500 tokens is comparable to *Agent Self-Correction* with a much lower context length of about 200 tokens. Moreover, *Agent Fixed* 500 shows a markedly higher recall than the *Unmodified* alternative, underscoring effective agent-formed queries.

Furthermore, the precision of context retrieval is significantly higher for agentic approaches (*Agent* and *Agent Self-Correction*) with a "free choice" of context length compared to other methods. This indicates that the agent ceases to retrieve when the context be-

comes filled with irrelevant tokens, aligning with the recall-precision trade-off. This trade-off is highlighted in the F_1 column.

These results suggest the following conclusions, supported by statistical tests:

1. *Unmodified* BM25 is inferior to agent-based methods for context retrieval in terms of localization metrics.
2. The Agentic retrieval with free context length choice collects a context of appropriate length.
3. The Agentic retrieval with an additional sufficiency condition marginally surpasses the single-loop agentic retrieval.

It should be noted that the context lengths for *Agent* and *Agent Self-Correction* have minimal differences. Our investigation shows that, on average, the standard Agent conducts 1.6 searches, while the Agent Self-Correction performs 2.6 searches and 1.5 invocations. The sufficiency criterion may need future modifications to improve context collection.

4.4.2 Code Editing

Strategy	Recall	Precision	F_1	chrF
Baseline				
Unmodified 200	0.14	0.08	0.09	0.064
Unmodified 500	0.17	0.05	0.07	0.061
Agentic Retrieval				
Agent	0.17	0.11	0.12	0.065
Agent Fixed 500	**0.22**	0.04	0.07	0.056
Agent Self-Correction	0.19	**0.13**	**0.14**	**0.074**

Table 3: Code Editing Metrics

To check the downstream code editing performance, we have conducted an evaluation using the code editing metrics introduced in Section 4.2.2.

We present the localization metrics for the performed edits and chrF scores calculated for `SWE-Bench_Lite` in Table 3 (full results in Appendix A). We can observe that they, to a great extent, follow the findings based on the context retrieval results described in Section 4.4.1. This may be due to the code editor tending to edit regardless of the relevance of the code. Introducing a reasoning step before editing might mitigate this issue, although it is outside the scope of this study.

Moreover, the results show that the *Agent Self-Correction* marginally outperforms the other strategies in terms of the chrF score between edited code sections in the predicted diff versus the ground truth diff. This underscores the importance of precision and the advantages of an agentic approach.

5 Conclusion

In this work, we present our investigation of the LLM-based context retrieval agents for code editing. We evaluated several context retrieval strategies similar to those found in the literature, focusing on their performance both in standalone context retrieval and in code editing.

To facilitate our experiments, we implemented a novel code editing system optimized for rapid experimentation and published it on GitHub[8].

We have developed a methodology for evaluation by collecting relevant data and establishing meaningful metrics.

Our evaluation leads to the following conclusions regarding our research question:

1. Agent-based methods for context collection significantly outperform unmodified BM25 in both context retrieval and code editing metrics;
2. Agents that assess the sufficiency of the context collection perform better than those that do not — the reason for this is that an excessive amount of context could otherwise accumulate;
3. Agents that perform several iterations of context collection and employ reasoning techniques (such as Self-Correction) show marginally better results than those with a single iteration.

Further Work

The area of AI agents for code editing is both practically important and inherently challenging, necessitating further work. In the following, we outline several potential directions for future work.

Firstly, code editing with our context retrieval approaches has shown low pass rates in code editing tasks, highlighting the critical need to improve its performance.

Secondly, many advanced context retrieval methods without the use of AI agents have been developed recently. In particular, the use of static analysis tools seems to be a promising research direction [5, 30, 55]. Investigating optimal integration of such methods into the AI Agent workflow is crucial.

Finally, it is crucial to enhance the agent's understanding of context sufficiency. Our experiments using `lca-code-editing` dataset revealed a significant lack of retrieved context, indicating a poor grasp of the related code.

[8] https://github.com/JetBrains-Research/ai-agents-code-editing

References

[1] 2022. GitHub Copilot, your AI pair programmer. https://github.com/features/copilot/

[2] 2023. Amazon CodeWhisperer. https://aws.amazon.com/codewhisperer/

[3] 2024. Devika - Agentic AI Software Engineer. https://github.com/stitionai/devika

[4] 2024. OpenDevin: Code Less, Make More. https://github.com/OpenDevin/OpenDevin

[5] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. 2023. CodePlan: Repository-level Coding Using LLMs and Planning. https://doi.org/10.48550/arXiv.2309.12499 arXiv:2309.12499 [cs]

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[7] Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Anton Lozhkov, Carolyn Jane Anderson, and Arjun Guha. 2024. Can It Edit? Evaluating the Ability of Large Language Models to Follow Code Editing Instructions. https://doi.org/10.48550/arXiv.2312.12450 arXiv:2312.12450 [cs]

[8] Harrison Chase. 2022. *LangChain*. https://github.com/langchain-ai/langchain

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[10] Luca Di Grazia and Michael Pradel. 2023. Code Search: A Survey of Techniques for Finding Code. *ACM Comput. Surv.* 55, 11, Article 220 (feb 2023), 31 pages. https://doi.org/10.1145/3565971

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL]

[12] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv:2312.10997 [cs.CL]

[13] Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2024. CRITIC: Large Language Models Can Self-Correct with Tool-Interactive Critiquing. arXiv:2305.11738 [cs.CL]

[14] Wenchao Gu, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, and Michael R. Lyu. 2021. CRaDLe: Deep code retrieval based on semantic Dependency Learning. *Neural Networks* 141 (Sept. 2021), 385–394. https://doi.org/10.1016/j.neunet.2021.04.019

[15] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), 933–944. https://api.semanticscholar.org/CorpusID:47021242

[16] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. https://doi.org/10.48550/arXiv.2401.14196 arXiv:2401.14196 [cs]

[17] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2023. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. arXiv:2311.05232 [cs.CL]

[18] Hamel Husain, Hongqiu Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *ArXiv* abs/1909.09436 (2019). https://api.semanticscholar.org/CorpusID:202712680

[19] Jiayi Wei, Greg Durrett, and Işıl Dillig. 2023. Coeditor: Leveraging Contextual Changes for Multi-round Code Auto-editing. *arXiv.org* (2023). https://doi.org/10.48550/arxiv.2305.18584

[20] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770 [cs]

[21] Kun Jing and Jungang Xu. 2019. A Survey on Neural Network Language Models. *ArXiv* abs/1906.03591 (2019). https://api.semanticscholar.org/CorpusID:182952667

[22] Jean Kaddour, J. Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. 2023. Challenges and Applications of Large Language Models. *ArXiv* abs/2307.10169 (2023). https://api.semanticscholar.org/CorpusID:259982665

[23] L. P. Kaelbling, M. L. Littman, and A. W. Moore. 1996. Reinforcement Learning: A Survey. arXiv:cs/9605103 [cs.AI]

[24] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Yu Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. 2020. Dense Passage Retrieval for

Open-Domain Question Answering. *ArXiv* abs/2004.04906 (2020). https://api.semanticscholar.org/CorpusID:215737187

[25] Cognition Labs. 2024. Devin, AI software engineer. https://www.cognition-labs.com/introducing-devin.

[26] Changshu Liu, Pelin Cetin, Yogesh Patodia, Saikat Chakraborty, Yangruibo Ding, and Baishakhi Ray. 2024. Automated Code Editing with Search-Generate-Modify. https://doi.org/10.48550/arXiv.2306.06490 arXiv:2306.06490 [cs]

[27] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the Middle: How Language Models Use Long Contexts. arXiv:2307.03172 [cs.CL]

[28] Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. 2024. RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation. https://doi.org/10.48550/arXiv.2402.16667 arXiv:2402.16667 [cs]

[29] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. OctoPack: Instruction Tuning Code Large Language Models. https://doi.org/10.48550/arXiv.2308.07124 arXiv:2308.07124 [cs]

[30] Huy N. Phan, Hoang N. Phan, Tien N. Nguyen, and Nghi D. Q. Bui. 2024. RepoHyper: Better Context Retrieval Is All You Need for Repository-Level Code Completion. arXiv:2403.06095 [cs.SE]

[31] Maja Popović. 2015. chrF: Character n-Gram F-score for Automatic MT Evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, Ondřej Bojar, Rajan Chatterjee, Christian Federmann, Barry Haddow, Chris Hokamp, Matthias Huck, Varvara Logacheva, and Pavel Pecina (Eds.). Association for Computational Linguistics, Lisbon, Portugal, 392–395. https://doi.org/10.18653/v1/W15-3049

[32] Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shihao Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bowen Li, Ziwei Tang, Jing Yi, Yuzhang Zhu, Zhenning Dai, Lan Yan, Xin Cong, Yaxi Lu, Weilin Zhao, Yuxiang Huang, Junxi Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. 2023. Tool Learning with Foundation Models. arXiv:2304.08354 [cs.CL]

[33] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. https://doi.org/10.48550/arXiv.2307.16789 arXiv:2307.16789 [cs]

[34] Qisheng Hu, Kaixin Li, Xu Zhao, Yuxi Xie, Tiedong Liu, Hui Chen, Qizhe Xie, and Junxian He. 2023. InstructCoder: Empowering Language Models for Code Editing. *arXiv.org* (2023). https://doi.org/10.48550/arxiv.2310.20329

[35] Stephen E. Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3 (2009), 333–389. https://api.semanticscholar.org/CorpusID:207178704

[36] Stephen Roller, Emily Dinan, Naman Goyal, Da Ju, Mary Williamson, Yinhan Liu, Jing Xu, Myle Ott, Kurt Shuster, Eric M. Smith, Y-Lan Boureau, and Jason Weston. 2020. Recipes for building an open-domain chatbot. arXiv:2004.13637 [cs.CL]

[37] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. arXiv:2302.04761 [cs.CL]

[38] Wei Tao, Yucheng Zhou, Wenqiang Zhang, and Yu Cheng. 2024. MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution. https://doi.org/10.48550/arXiv.2403.17927 arXiv:2403.17927 [cs]

[39] Andrew Trotman, Xiangfei Jia, and Matt Crane. 2012. Towards an Efficient and Effective Search Engine. In *OSIR@SIGIR*. https://api.semanticscholar.org/CorpusID:9444322

[40] Andrew Trotman, Antti Puurula, and Blake Burgess. 2014. Improvements to BM25 and Language Models Examined. *Proceedings of the 19th Australasian Document Computing Symposium* (2014). https://api.semanticscholar.org/CorpusID:207220720

[41] Michele Tufano, Anisha Agarwal, Jinu Jang, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. 2024. AutoDev: Automated AI-Driven Development. https://doi.org/10.48550/arXiv.2403.08299 arXiv:2403.08299 [cs]

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL]

[43] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:2203.11171 [cs.CL]

[44] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. https://doi.org/10.48550/arXiv.2201.11903 arXiv:2201.11903 [cs]

[45] Lilian Weng. 2023. LLM Powered Autonomous Agents. https://lilianweng.github.io/posts/2023-06-23-agent/.

[46] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, and Tao Gui. 2023. The Rise and Potential of Large Language Model Based Agents: A Survey. arXiv:2309.07864 [cs.AI]

[47] John Yang, Carlos E. Jimenez, Alexander Wettig, Shunyu Yao, Karthik Narasimhan,

and Ofir Press. 2024. SWE-agent: Agent Computer Interfaces Enable Software Engineering Language Models.

[48] Ke Yang, Jiateng Liu, John Wu, Chaoqi Yang, Yi R. Fung, Sha Li, Zixuan Huang, Xu Cao, Xingyao Wang, Yiquan Wang, Heng Ji, and Chengxiang Zhai. 2024. If LLM Is the Wizard, Then Code Is the Wand: A Survey on How Code Empowers Large Language Models to Serve as Intelligent Agents. arXiv:2401.00812 [cs]

[49] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. https://doi.org/10.48550/arXiv.2305.10601 arXiv:2305.10601 [cs]

[50] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL]

[51] Xin Ye, Hui Shen, Xiao Ma, Razvan C. Bunescu, and Chang Liu. 2016. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (2016), 404–415. https://api.semanticscholar.org/CorpusID:4280759

[52] ChengXiang Zhai. 2008. Statistical Language Models for Information Retrieval: A Critical Review. *Found. Trends Inf. Retr.* 2 (2008), 137–213. https://api.semanticscholar.org/CorpusID:61572040

[53] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. https://doi.org/10.48550/arXiv.2303.12570 arXiv:2303.12570 [cs]

[54] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges. https://doi.org/10.48550/arXiv.2401.07339 arXiv:2401.07339 [cs]

[55] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. arXiv:2404.05427 [cs.SE]

[56] Zhuosheng Zhang, Yao Yao, Aston Zhang, Xiangru Tang, Xinbei Ma, Zhiwei He, Yiming Wang, Mark Gerstein, Rui Wang, Gongshen Liu, and Hai Zhao. 2023. Igniting Language Intelligence: The Hitchhiker's Guide From Chain-of-Thought Reasoning to Language Agents. arXiv:2311.11797 [cs.CL]

[57] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. arXiv:2303.18223 [cs.CL]

[58] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. arXiv:2306.05685 [cs.CL]

[59] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. https://doi.org/10.48550/arXiv.2302.05527 arXiv:2302.05527 [cs]

A Full Evaluation Results

In this section we present the full experiment results. In Tables 4, 5, and 6, the plus-minus term denotes the 95% confidence interval of the mean estimator.

Among the code editing metrics not included in Table 3, CodeBertScore and LLM evaluation proved to be ineffective in our experiments. The pass rates for all approaches using `SWE-Bench_Lite` datasets were low, indicating the need to improve our context retrieval and editing components. For the `lca-ci-fixing`, we observe very low successful test passes, so there is not enough data to draw conclusions.

As for the other datasets, we note that the results for the `lca-ci-fixing` are quite similar to those for the `SWE-Bench_Lite`. For the `lca-code-editing`, the agentic methods with variable context length show significantly reduced performance. This may be due to the longer ground truth context in this dataset compared to others (see Table 1). Thus, this highlights a valuable research direction: optimizing the agent's understanding of the necessary context.

name	Agent	Agent Fixed 500	Agent SC	BM25 200	BM25 500
Line CR Recall	0.046 ± 0.013	0.100 ± 0.02	0.054 ± 0.014	0.065 ± 0.018	0.081 ± 0.019
Line CR Prec	0.040 ± 0.012	0.0137 ± 0.003	0.048 ± 0.014	0.030 ± 0.009	0.0148 ± 0.004
Line CR F1	0.034 ± 0.010	0.023 ± 0.005	0.039 ± 0.010	0.037 ± 0.010	0.024 ± 0.006
Line AE Recall	0.049 ± 0.015	0.078 ± 0.019	0.072 ± 0.018	0.071 ± 0.019	0.076 ± 0.019
Line AE Prec	0.033 ± 0.011	0.0107 ± 0.003	0.044 ± 0.012	0.031 ± 0.009	0.0186 ± 0.006
Line AE F1	0.034 ± 0.011	0.0178 ± 0.005	0.046 ± 0.012	0.038 ± 0.011	0.026 ± 0.007
PyScope CR Recall	0.149 ± 0.02	0.25 ± 0.03	0.176 ± 0.03	0.136 ± 0.02	0.175 ± 0.03
PyScope CR Prec	0.106 ± 0.019	0.048 ± 0.007	0.124 ± 0.02	0.081 ± 0.016	0.048 ± 0.008
PyScope CR F1	0.113 ± 0.019	0.075 ± 0.010	0.131 ± 0.020	0.094 ± 0.017	0.071 ± 0.011
PyScope AE Recall	0.174 ± 0.03	0.22 ± 0.03	0.193 ± 0.03	0.144 ± 0.03	0.173 ± 0.03
PyScope AE Prec	0.112 ± 0.019	0.048 ± 0.009	0.126 ± 0.02	0.078 ± 0.015	0.051 ± 0.009
PyScope AE F1	0.124 ± 0.019	0.071 ± 0.011	0.138 ± 0.02	0.093 ± 0.017	0.072 ± 0.012
File CR Recall	0.36 ± 0.04	0.53 ± 0.04	0.38 ± 0.04	0.23 ± 0.03	0.29 ± 0.04
File CR Prec	0.23 ± 0.03	0.111 ± 0.013	0.24 ± 0.03	0.141 ± 0.02	0.091 ± 0.013
File CR F1	0.27 ± 0.03	0.173 ± 0.016	0.28 ± 0.03	0.169 ± 0.03	0.133 ± 0.018
File AE Recall	0.34 ± 0.04	0.40 ± 0.04	0.34 ± 0.04	0.23 ± 0.03	0.27 ± 0.04
File AE Prec	0.24 ± 0.03	0.112 ± 0.015	0.24 ± 0.03	0.147 ± 0.02	0.098 ± 0.016
File AE F1	0.27 ± 0.03	0.163 ± 0.019	0.27 ± 0.03	0.172 ± 0.03	0.137 ± 0.020
chrF	0.065 ± nan	0.056 ± 0	0.074 ± nan	0.064 ± nan	0.061 ± 0
Context	247 ± 41	894 ± 87	253 ± 35	218 ± 4	487 ± 6

Table 4: Full `SWE-Bench_Lite` Results

name	Agent	Agent Fixed 2000	Agent Fixed 500	Agent SC	BM25 2000	BM25 500
Line CR Recall	0.064 ± 0.03	0.183 ± 0.05	0.117 ± 0.05	0.074 ± 0.03	0.159 ± 0.06	0.109 ± 0.05
Line CR Prec	0.122 ± 0.05	0.0190 ± 0.008	0.022 ± 0.008	0.117 ± 0.05	0.0102 ± 0.004	0.023 ± 0.010
Line CR F1	0.063 ± 0.03	0.025 ± 0.008	0.028 ± 0.010	0.066 ± 0.03	0.0163 ± 0.006	0.030 ± 0.013
Line AE Recall	0.104 ± 0.04	0.173 ± 0.06	0.137 ± 0.05	0.109 ± 0.04	0.116 ± 0.05	0.106 ± 0.05
Line AE Prec	0.109 ± 0.04	0.0190 ± 0.009	0.024 ± 0.010	0.107 ± 0.04	0.0159 ± 0.009	0.032 ± 0.017
Line AE F1	0.087 ± 0.03	0.025 ± 0.009	0.032 ± 0.011	0.089 ± 0.04	0.023 ± 0.013	0.038 ± 0.020
PyScope CR Recall	0.140 ± 0.05	0.32 ± 0.07	0.22 ± 0.06	0.190 ± 0.06	0.24 ± 0.07	0.176 ± 0.06
PyScope CR Prec	0.175 ± 0.06	0.041 ± 0.011	0.046 ± 0.013	0.177 ± 0.05	0.023 ± 0.006	0.052 ± 0.018
PyScope CR F1	0.134 ± 0.05	0.058 ± 0.012	0.063 ± 0.017	0.153 ± 0.05	0.036 ± 0.010	0.066 ± 0.02
PyScope AE Recall	0.187 ± 0.06	0.28 ± 0.07	0.24 ± 0.06	0.20 ± 0.06	0.21 ± 0.06	0.155 ± 0.06
PyScope AE Prec	0.171 ± 0.06	0.040 ± 0.012	0.045 ± 0.013	0.168 ± 0.05	0.026 ± 0.010	0.052 ± 0.02
PyScope AE F1	0.152 ± 0.05	0.054 ± 0.012	0.063 ± 0.017	0.159 ± 0.05	0.040 ± 0.015	0.062 ± 0.02
File CR Recall	0.25 ± 0.07	0.43 ± 0.07	0.34 ± 0.07	0.28 ± 0.07	0.32 ± 0.07	0.23 ± 0.07
File CR Prec	0.21 ± 0.06	0.080 ± 0.017	0.102 ± 0.03	0.22 ± 0.06	0.042 ± 0.010	0.085 ± 0.03
File CR F1	0.21 ± 0.06	0.119 ± 0.02	0.133 ± 0.03	0.23 ± 0.06	0.070 ± 0.016	0.112 ± 0.03
File AE Recall	0.24 ± 0.07	0.36 ± 0.07	0.30 ± 0.07	0.27 ± 0.07	0.28 ± 0.07	0.194 ± 0.06
File AE Prec	0.22 ± 0.06	0.081 ± 0.019	0.104 ± 0.03	0.23 ± 0.06	0.054 ± 0.016	0.096 ± 0.04
File AE F1	0.21 ± 0.06	0.116 ± 0.02	0.130 ± 0.03	0.23 ± 0.06	0.083 ± 0.02	0.112 ± 0.04
Pass Rate	0.0069 ± 0	0.0139 ± 0	0.0069 ± 0	0.021 ± nan	0.021 ± nan	0.0069 ± 0
chrF	0.040 ± nan	0.054 ± nan	0.048 ± 0	0.041 ± nan	0.035 ± nan	0.033 ± nan
CodeBertScore	0.552 ± nan	0.556 ± nan	0.554 ± nan	0.553 ± nan	0.552 ± nan	0.552 ± nan
Context	171 ± 35	2058 ± 205	805 ± 81	194 ± 48	1800 ± 54	488 ± 11

Table 5: Full `lca-ci-fixing` Results

name	Agent	Agent Fixed 2000	Agent Fixed 500	Agent SC	BM25 200	BM25 2000	BM25 500
Line CR Recall	0.0179 ± 0.010	0.098 ± 0.03	0.055 ± 0.020	0.022 ± 0.011	0.0153 ± 0.008	0.060 ± 0.02	0.034 ± 0.014
Line CR Prec	0.117 ± 0.05	0.055 ± 0.02	0.072 ± 0.03	0.117 ± 0.05	0.075 ± 0.04	0.034 ± 0.014	0.062 ± 0.03
Line CR F1	0.028 ± 0.014	0.058 ± 0.020	0.055 ± 0.019	0.033 ± 0.016	0.024 ± 0.012	0.040 ± 0.015	0.041 ± 0.016
Line AE Recall	0.042 ± 0.03	0.098 ± 0.04	0.063 ± 0.03	0.042 ± 0.02	0.023 ± 0.02	0.067 ± 0.03	0.042 ± 0.03
Line AE Prec	0.118 ± 0.05	0.050 ± 0.019	0.068 ± 0.03	0.118 ± 0.05	0.091 ± 0.04	0.036 ± 0.016	0.069 ± 0.03
Line AE F1	0.046 ± 0.02	0.054 ± 0.019	0.052 ± 0.02	0.046 ± 0.02	0.028 ± 0.016	0.040 ± 0.017	0.038 ± 0.017
PyScope CR Recall	0.070 ± 0.03	0.185 ± 0.04	0.124 ± 0.03	0.078 ± 0.03	0.050 ± 0.02	0.158 ± 0.04	0.096 ± 0.03
PyScope CR Prec	0.21 ± 0.06	0.113 ± 0.03	0.144 ± 0.03	0.21 ± 0.06	0.156 ± 0.05	0.075 ± 0.016	0.127 ± 0.03
PyScope CR F1	0.089 ± 0.03	0.117 ± 0.03	0.114 ± 0.03	0.094 ± 0.03	0.065 ± 0.02	0.089 ± 0.018	0.094 ± 0.03
PyScope AE Recall	0.092 ± 0.04	0.188 ± 0.05	0.141 ± 0.04	0.098 ± 0.04	0.057 ± 0.03	0.158 ± 0.05	0.101 ± 0.04
PyScope AE Prec	0.21 ± 0.06	0.111 ± 0.03	0.145 ± 0.04	0.21 ± 0.06	0.157 ± 0.05	0.073 ± 0.019	0.124 ± 0.04
PyScope AE F1	0.095 ± 0.03	0.110 ± 0.03	0.114 ± 0.03	0.103 ± 0.04	0.068 ± 0.03	0.084 ± 0.02	0.087 ± 0.03
File CR Recall	0.22 ± 0.06	0.42 ± 0.07	0.36 ± 0.07	0.21 ± 0.06	0.175 ± 0.05	0.42 ± 0.07	0.25 ± 0.06
File CR Prec	0.30 ± 0.07	0.188 ± 0.04	0.23 ± 0.04	0.30 ± 0.07	0.23 ± 0.06	0.120 ± 0.02	0.188 ± 0.04
File CR F1	0.22 ± 0.06	0.22 ± 0.04	0.24 ± 0.04	0.22 ± 0.05	0.170 ± 0.05	0.167 ± 0.03	0.189 ± 0.04
File AE Recall	0.22 ± 0.06	0.37 ± 0.07	0.33 ± 0.07	0.192 ± 0.06	0.143 ± 0.05	0.33 ± 0.07	0.21 ± 0.06
File AE Prec	0.31 ± 0.07	0.196 ± 0.04	0.24 ± 0.05	0.31 ± 0.07	0.22 ± 0.06	0.133 ± 0.03	0.188 ± 0.05
File AE F1	0.22 ± 0.06	0.22 ± 0.04	0.23 ± 0.05	0.20 ± 0.05	0.150 ± 0.05	0.160 ± 0.03	0.165 ± 0.04
chrF	0.040 ± 0	0.059 ± nan	0.042 ± nan	0.034 ± nan	0.032 ± nan	0.071 ± 0	0.052 ± 0
CodeBertScore	0.554 ± nan	0.568 ± nan	0.561 ± nan	0.552 ± nan	0.548 ± nan	0.560 ± nan	0.556 ± nan
Context	196 ± 22	2589 ± 490	846 ± 112	198 ± 26	212 ± 6	1837 ± 38	487 ± 10

Table 6: Full `lca-code-editing` Results