Structure-Level Interactions for LLM-Based Coding Agents

by
Russell Rozenbaum

advised by
Professor Cyrus Omar

A Thesis submitted in partial fulfillment of the requirements for
Honors in the College of LSA at the University of Michigan

Ann Arbor, Michigan
April 2025

Acknowledgements

I would like to express my sincere gratitude to my research advisor, Professor Cyrus Omar, for guiding me throughout the process of conducting and writing this thesis. Professor Omar, I am beyond grateful for the opportunities you and the Future of Programming Lab have brought me since joining a little less than a year ago. I have gained invaluable and transformative experience instrumental to the shaping of this thesis and my overall development as a researcher.

I would also like to show great appreciation to Andrew Blinn for assisting in the design and construction of this thesis. Andrew, your insightful feedback and deep intuition on CS research and programming languages are inspiring and have motivated me throughout my undergraduate research experience.

My thanks further extend to my research partner on this project, Cyrus Desai, whose collaboration, efforts, and intellectual contributions significantly enhanced this work.

Finally, I would like to thank Professor Ang Chen for being a reader and reviewer of this thesis.

Contents

Chapter 1

Introduction

The entrance of large language models (LLMs) [1] has yielded a new frontier in automated program synthesis, a longstanding goal of artificial intelligence [2] [4]. A variety of research within this domain has been conducted in LLM-based artificial intelligence methods and designs [4–6]. More specifically, recent research has shown promising results in utilizing LLMs with agentic approaches for programming-related tasks [3] [13] [18] [34] [50]. Existing work has achieved state-of-the-art results by equipping their agents with their own integrated development environment (IDE) [3] [13] [44], an approach referred to as the agent-computer-interface (ACI). These ACIs enable agents to create and edit code files, navigate code bases, execute tests, and ultimately interact with programs analogous to how humans interact with programs through IDEs.

ACIs come equipped with agent-facing tools, suitable for use by an LLM-based coding agent. Critically, existing works propose tools within their ACIs that strongly depend on code segmented solely by line numbers [4] [9] [14] [32] [34] [35]. We hypothesize that using such line-based architectures comes with several disadvantages. This hypothesis is based upon the fact that LLMs are trained and evaluated on large corpora of human-generated documents, which consequently leads to models learning patterns and behaviors from human code [45] [46]. It has been known that, for humans, indirection through purely text-based and line-based program interactions introduces fundamental complexity into the programming process [12]. We argue that LLM-coding agents need their ACI tools based similarly on what human programmers depend on. Strictly line-based edit actions are not entirely intuitive, in that they provide arbitrary, unhelpful information to the programmer, whether they be human or agentic.

When developing a program, the programmer rarely gives significant attention to the line numbers at which they edit. When debugging a program, the programmer may resort to line numbers to track down compilation errors made apparent by the console, but ultimately focus on the semantics and structure of the erroneous code itself. Line-based approaches further run the risk of random cutoffs during code segmentation, feeding the LLM potentially flagrant and performance-degrading windows of code. Furthermore, humans often apply varying degrees of code manipulation depending on the task, ranging from smaller, more frequent edit actions when developing code, to larger, less frequent edit actions when repairing defects [16]. This lends to our proposition of developing a **higher-level, structure-based action language**, not only allowing, but scaffolding the LLM-agent to "think" and "reason" about code generation at a deeper, nuanced level rooted in the semantics of the language and program, rather than the lexical, line-oriented syntax.

We have found that LLM-based coding agents typically focus on constructing architectures composed of the following schemata: **1. Perception** where the agent, aided by other tools such as a language server, gathers information about its environment, **2. Interpretation** where the LLM stores, processes, and reasons about relevant information, and **3.**

Interaction where the agent interacts with and makes changes upon the environment in which it operates [10] [34]. We propose novel design improvements to each of these three schemata. Most notably, we intend to equip our agent with a high-level, structure-based action language, as opposed to the preexisting lower-level, line-based action languages. Beyond this newly designed action language, we deploy our agent within Hazel, a live functional programming environment capable of structure-level edit and search actions, as well as extracting relevant context and rich semantic information via a language server [12] [11]. Hazel is capable of always producing semantically and syntactically meaningful program states, thus always able to derive AST information, even in potentially incomplete or ill-typed programs. Other languages face issues when programs are left in syntactically ill-formed, semantically meaningless states, where AST information fails to be extracted. Since two-thirds of edits that developers make leave syntactically incorrect source code [14], Hazel serves as the perfect candidate to alleviate such hindrances, allowing us to give our agent adequate AST information even when errors may be present in other parts of the program. Blinn et al. [13] previously found promising results in their work on ChatLSP, where they supplied the agent with a powerful IDE, equipped with a language server to effectively incorporate the type and binding structure of the language itself. This contextualization allows the system to prompt the LLM with codebase-wide contextual information that is not lexically local to the cursor, nor necessarily in the same file. We also design, build, and work with front-facing UI components, developing an interactive user interface within Hazel for human-agent communication and experimentation, drawing strongly off design principles used in modern, popular, industry AI coding assistants [1].

1.1 Intellectual Merit

This project extends prior efforts in AI-assisted coding by looking into how LLMs reason about code completion tasks. At its core, it contributes to the longstanding goal of automated program synthesis, dating back more than 70 years [2] [4]. Our efforts aim to extend the contributions made by ChatLSP through further utilizing the language server to allow for high-level, structure-based actions. The agent can use these actions to interact directly and indirectly with its environment during a task completion process [2]. This tight integration of the language server with the agent aims to further address the consequences assumed when critical task-relevant context comes from definitions that appear neither in the cursor window nor in the training data, also known as the **semantic contextualization problem** [13] [44] [51]. Exploring such hypotheses requires addressing questions such as: What are the benefits of equipping an agent with a high-level, structure-based action language? How can we efficiently produce dependable and credible performance metrics, avoiding the risks of data contamination, model discrepancies, etc? What does the human user need and not need when using such AI coding tools? These questions lie at the core of our research, and span a broad range of fields from prompt engineering [22] and programming languages theory

[1] Examples include Cursor (`https://www.cursor.com/`), Devin AI (`https://devin.ai/`), GitHub Copilot (`https://github.com/features/copilot`), and Windsurf (`https://codeium.com/windsurf`)

[2] It should be noted that the research done by [13] predominantly worked with code completion at single, empty hole, rather than what we plan to work with, task completion

to human-computer interactions (HCI) and UI design. We attempt to unveil answers to each of these questions through the implementation of backend and front-end agent architecture within the Hazel code base.

1.2 Broader Impact

Our proposed research serves to inject knowledge into an increasingly popular field of computer science and HCI research, fueled by the creation of powerful LLMs. Beyond injecting results into a growing field of research, we see an opportunity in increasing the learning and user experiences for programmers using Hazel. Since its creation, Hazel has been geared towards the use case of education on functional programming and type theory [11] [12]. Research shows that AI proves to be a useful tool in improving computational thinking, increasing student interest, and transforming programming education [36] [52]. Thus, implementing an LLM technology and an interactive UI within Hazel pushes Hazel towards its underlying goals in bettering student education.

1.3 Overview

The overall goals of our research lie in the following questions:

1. Is the use of a high, structure-level action language within an LLM-based coding agent tenable? Are its effects favorable, or even discernible, from existing action languages?

2. Do LLM-based agents prove to be a useful tool in human learning for lesser-known, low-resource programming languages such as Hazel?

We are currently still in the process of implementing our proposed agent architecture and evaluating these goals. This paper acts as a milestone update and a tangible layout of the methods we have laid out thus far. Specifically, the high-level action language we have formed, the front-facing UI components we have implemented, and the current state of agent architecture within Hazel.

Chapter 2

Background & Related Work

2.1 Hazel

Hazel [1] is a live programming environment based on a bidirectionally typed lambda calculus. It features error holes, automatically placing terms assigned a type inconsistent with the expected type inside a hole, a sort of membrane encapsulating potentially erroneous parts of the program. This defers type consistency checks until the term inside the hold is finished, allowing for incomplete programs not only to be checked for type consistency, but even evaluated [11] [12]. Uniquely, every complete and incomplete program that can be designed within Hazel has been proven to be statically and dynamically well-defined. Because of this, the language server need not be disabled for certain parts of the program, even when those parts may very well be unfinished or erroneous. This makes Hazel a star candidate for our research hypotheses, as in any state of a Hazel program, we can extract rich and relevant context to give to the LM-based coding agent via the language server. Because the foundations of Hazel lie in typed structure editing [12], this structure extraction is not only possible, but highly feasible.

2.2 Prompt Engineering

Prompt engineering has emerged in recent years as a design process capable of greatly leveraging the power of LLMs. Prompts are the sole way to interact with a trained LLM post-training, without opening up the black box itself and attempting to tweak its inner parameters. Research has led to at least 29 unique and transformative prompt engineering techniques, such as few-shot prompting, chain-of-thought reasoning, and retrieval augmented generation [22]. Given such a great quantity of successful prompting methods, LLMs, and consequently, LLM-based agents, are highly sensitive to exactly how a prompt is constructed. Our task is inherently one of prompt engineering, or at the very least, an analysis of the effects different pieces of information have on the LLM. Many methods in this field opt for the use of Chain-of-Thought reasoning (CoT), which has proved highly beneficial for LLMs in complex [30] [26], logical [29], and serial [42] reasoning tasks, especially for that of program synthesis [22] [3] [28] [5]. Other successful research finds sub-prompting to be beneficial in program synthesis via LLMs [41] [48]. Sub-prompting methods use a form of least-to-most reasoning to encourage the LLM to divvy up complex tasks into smaller, more isolated tasks. Few-shot prompting, where a few input-output examples for the task or role the agent has assumed are given in the prompt, [23] [22] has shown to be beneficial in tasks requiring LLMs to perform in low-resource domains.

[1] Hazel is publicly accessible at `https://hazel.org/`

2.3 AI Coding Agents

2.3.1 SWE-Agent

SWE-agent [3] has achieved state-of-the-art performance in software engineering tasks, evaluated on a benchmark drawn from real-world GitHub issues [9]. SWE-agent's most novel contributions introduce the idea of an ACI with LLM-friendly commands, enabling the agent to navigate repositories, search files, view files, and edit lines of code. Their ACI solves the shortcomings exhibited by agents operating within Linux shell environments [37]. They conclude that LLM-based agents need IDEs too (the ACI). Critically, SWE-agent's file viewer and file editing categories, as seen in A.2, harness commands that depend on line numbers to derive locality from the code base the agent operates within.

As with many LLM-based coding agents, SWE-agent rarely gets the code completion correct on the first run, achieving a pass@1 rate of 12.5% on SWE-bench [3] [9]. Due to this, error rounds, where the agent iteratively reattempts its code edits based on code linter errors or test cases, are needed. Importantly, they found that 51.7% of task completion trajectories have one or more failed edits, noting that while a majority of agents recover from failed edits, there exists diminishing returns with error rounds, as the agent recurringly fails at avoiding specific errors.

2.3.2 ChatLSP

ChatLSP [13] is a code completion agent [2] operating in Hazel, as opposed to SWE-agent, which is tasked with repository-level task implementations. They integrate the Hazel language server to adequately gather relevant type definitions and function headers of the hole that the programmer has requested to be filled by the agent. Likewise to SWE-agent, the ChatLSP agent iteratively refines code completions via error rounds, where the Hazel language server provides error localization [43], relevant information about the error(s). Their findings on the performance of error rounds align with those of [3] and SWE-agent, in that error rounds offer diminishing returns, finding a ratio of tests passing with versus without error rounds around 2. After evaluating on MVUBench (a benchmark consisting of model-view-update web applications), they conclude that by giving the agent relevant contextualization via the Hazel language server, the agent was able to perform on Hazel, a low-resource language for which very little documentation and publicly available training data exists, on-par with that of TypeScript, a relatively high-resource language.

2.3.3 Other Agents and Benchmarks

There exists a plethora of other LLM-based agents or agent-like frameworks specialized for coding tasks. These works include CodeCoT [35], CursorCore [31], PyCoder [14], and LDB [24]. Liu, J. et al. [34] offers a thorough literature review on state-of-the-art models and frameworks in agentic program synthesis.

[2] In their paper [13], Blinn et al. refer to their work as "static contextualization". We describe their work from a slightly different lens, viewing the process of the hole filling-language server-LLM interaction as one done by an, albeit liminal, agent.

ChatLSP isn't the first work to engage the semantic contextualization problem in program synthesis. CodeTrek [53] uses semantic contextualization for program repair tasks based on deep learning approaches. Pei et al. [51] contextualize Python function calls querying a static program analyzer to obtain relevant information, and IDECoder [44] utilizes static contexts available through an IDE to aid in LLM-based repository-level code completion.

There has also been significant work in developing benchmarks to evaluate LLM-based coding agents, such as SWE-Bench [9], SWE-lancer [21], CanItEdit [32], ML-Bench [7], MVUBench [13], and HumanEval [49]. The list of benchmarks is highly extensive, as it is oftentimes of interest for works in this area to craft freshly curated benchmarks to evade threats of validity stemming from publicly available benchmark tests appearing in LLM training data.

2.4 Human vs LLM Programming

A great quantity of research has been conducted in the study of what does and does not work for human programmers. We highlight a few here, and analyze recent works which compare the programming behaviors between humans and LLMs, each of which may potentially demonstrate strong relevance to our research hypotheses.

2.4.1 Human vs LLM Attention

Humans and LLMs think differently. More complex models often produce more complex errors when writing programs, while simpler models produce more explainable errors. Kou, B. et al. [46] found that 31% of errors their CodeGen model produced could be explained by one of five attention patterns, while only 8% of GPT-4 errors could be described. Their work hints at LLMs deriving semantical understandings of computer programs at a more in-depth level. This supports the idea that LLM-based coding agents may benefit from more semantically rich and intuitive descriptions of programs, tasks, and contextualization. Work conducted by Licorish, S. et al. [45] further emphasizes a need for greater contextual awareness rates in LLMs. They show that 45% of GPT-4 generated code and 30% of human code had incorrect redefinitions from outer scope contexts, especially in scripts with nested functions or complicated loops. These findings highlight an inefficiency in LLM-based coding agents when it comes to properly considering and reasoning about code contexts and structures.

Afzal & Goues. [16] found that human programming behavior differs in nature depending on whether they are initially developing code or repairing defects. In particular, they found participants to "frequently apply small edits to the source code when they develop code, but apply less frequent or larger edits while repairing defects." From prior discussion on related work, we found that similar methods were used in LLM-based coding agents, such as splitting complex prompts into smaller, digestible prompts. We hope that extrapolating these findings to LLMs will help shed light on the efficacy of using higher, structure-level actions in LLM-based coding agents to mitigate bugs and boost performance.

2.4.2 Human vs LLM IDEs

Reliability, ease of use, and efficiency are among the most highly regarded traits of an IDE [17]. Work done by Kline & Seffah. [15] shows that a well-designed IDE with a strong focus on these traits enhances usability and learnability in novice and even expert programmers [15]. It also demonstrates that integrating learning within the IDE itself can reduce training time and costs.

In this work, we extend research done in this field to LLMs. Similar to human behavior, we want to avoid over-contextualizing and providing irrelevant context to the LLM. By supplying the LLM with a high, structure-level action language, relevant AST information, and ultimately precise context informed by the language semantics, we suspect a boost in performance of the LLM's code generation capabilities.

Chapter 3

Methods

3.1 Agent Architecture

In this paper, we acknowledge each of the three aforementioned components typical of a coding agent to comprise 3.1. We describe our architecture as follows:

1. **Perception.** We use Hazel's language server for context extraction and retrieval (such as gathering that of relevant types, function headers, etc, to conscientiously inform the agent of semantics and parts of the program not lexically-local to the cursor). To accomplish this, we make use of and build upon a prior implementation of ChatLSP [13]. We predominantly use ChatLSP during the construction of the task prompt itself. This prompt is then passed as input to the LLM, where it is handled in the interpretation stage. We then iterate on error rounds until either an error-round limit has been reached or the language server finds no type errors after the interaction stage.

2. **Interpretation.** It should be noted that we do not, and seemingly can not, directly manipulate this design principle. Recall that we've defined this process as the component "where the LLM stores, processes, and reasons about relevant information". It is where planning and memory take place [34] within the agent *through* the LLM, which we have no direct control over. Rather, we aim to indirectly affect this component through allowing the agent access to a high, structure-level action language, a language server, and meticulously crafted prompts. That is, we do not do anything with model internals; instead, we give it scaffolding for the model to reason with.

   We construct our own ACI toolkit A.1, basing it strongly on the work done by SWE-agent. Critically, however, we redesign commands in the file viewing and edit action

categories. Rather than solely depending on line numbers to derive lexical location within the code base, we look to the structure of the code itself. Line-based file viewing and code editing actions are meaningless beyond their purpose of providing lexical location. We reckon that successfully providing locality within the code base purely through higher-level semantics (such as function definitions, type definitions, and modules) leads to a decrease in hallucinatory responses and an increase in the quality of code produced. Furthermore, such actions provide high-level, semantic chunks to which meaning is ascribed by both programmers and the language server, such as definitions, type annotations, etc. Low-level, textual information, such as line indices, fails to capture this rich information.

Our agent can perform in one of two modes: simple and complex reasoning. In the simple reasoning setting, we feed the prompt into the LLM as-is. With complex reasoning triggered, we encourage the LLM to use CoT.

3. **Interaction.** We plan to implement the backend architecture for handling outputs given by the agent. This mainly involves extracting the action command from the agent's response, and performing the action on the code base as defined by the respective documentation in our action language A.1.

   To materialize and visualize the agent's process, we implement a front-end architecture, including a user-facing agent chat component, for human-agent and agent-computer interactions. In particular, we aim here to make our agent easily accessible by users of Hazel, and by us as evaluators of the agent.

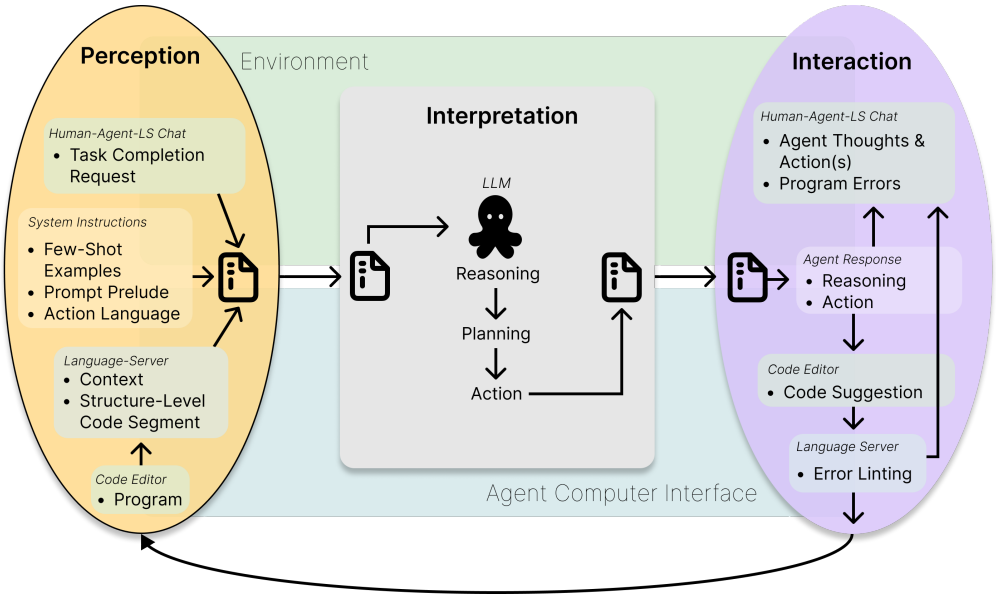

Figure 3.1: A visualization of our agent architecture.

3.2 A High, Structure-Level Action Language

As discussed in 2.3, many recent works have begun more strongly integrating the ACI within their agent frameworks. Yet, the use of line-based action languages remains prevalent in each of these related frameworks. To the best of our knowledge, it is unstudied whether this line-based approach is *ideal*. Our action language is made discernible from these action languages through altering line-based commands to instead use structure-based commands.

Larger, higher-level edit actions like these have been studied to effectively mitigate bugs in human programming [16]. It is also common practice for human programmers to use abstraction (chunking code into blocks, definitions, modules, etc.). Programmers will frequently use names as opposed to line locations to leverage the power of association when referring to the contents of a program. Thus, as a byproduct of training LLMs to pick up on patterns present in large corpora of human-generated code [45], we suspect that LLM-based coding agents could very well benefit from more actions based on the high-level structure of the code it works with rather than windows of this code, with arbitrary start and end points based on line indices.

We modify the action language defined by SWE-agent [3] A.2 A.1. Implementation and evaluation of this remains a future goal of ours.

3.3 Communicating with the Agent

Within Hazel, we craft a human-agent-LS interface. The accomplishments of this interface are twofold:

1. Hands the user control over the agent. Namely, the user can choose the LLM the agent is to use, what "mode" the agent is to operate within (discussed in 3.3.3), and communicate with the agent directly through a chat log.

2. Displays to the user the agent's thoughts, processes, and answers, as well as feedback from the language server and important system messages.

We develop this user interface with two types of end-users in mind. It is to aid us as researchers in the research process, allowing for accessible, transcribed, and thorough insights into the agent's thoughts and behaviors, the language server's feedback, as well as any system failures or bugs. Secondly, it is a way for new and existing users of the Hazel programming language to access and make use of LLM technology from directly within Hazel, helping us in addressing the first of our research goals. This interface is conveniently located in a sidebar menu adjacent to the editor 3.2.

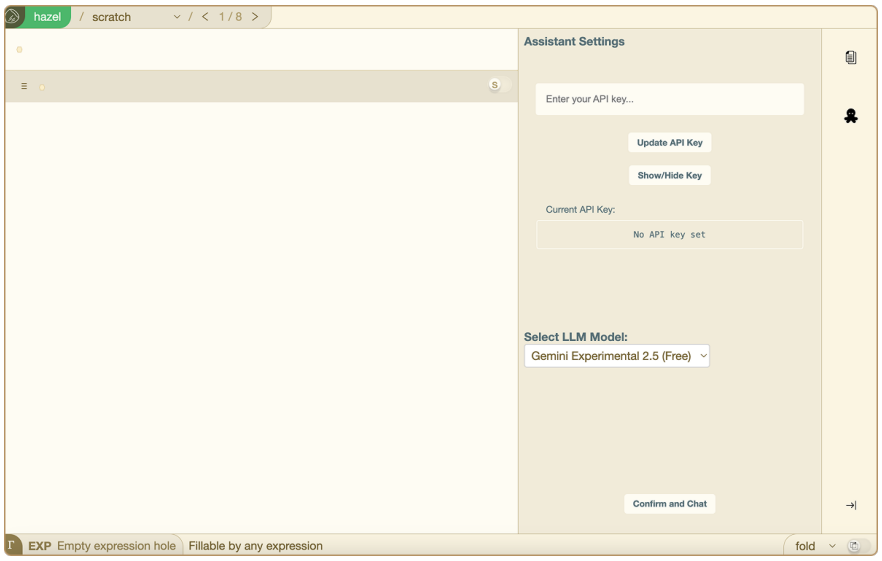

Figure 3.2: The agent sidebar is located on the right of the page, currently displaying the "Assistant Settings" menu.

3.3.1 Agent Settings

We currently make use of OpenRouter's unified LLM interface [1] to access a variety of available large language models. Users require only a single OpenRouter API key to access any of these models. In the agent's settings menu, the user can enter this API key and then select from a dropdown menu an LLM that we currently support on our backend for the agent to use. These API keys are free, and we support several free large language models available through OpenRouter. This simplifies the process of evaluating a variety of different LLMs, eliminating the need to hard-code API keys into Hazel's backend and making the agent easily usable by anyone with an OpenRouter API key. Once the user has entered an API key and selected the LLM they'd like the agent to use, they can click the "Confirm and Chat" button, taking them to the User-Agent-LS chat interface.

3.3.2 User-Agent-LS Chat Interface

Our user-agent-LS chat interface is where the conversations between the user, agent, and language server (and sometimes system messages regarding API call fails, error-round limits being reached, etc) are displayed. The user can create new chats with the "New Chat" button and reopen past chats from the history window by clicking the "Past Chats" button. The user can choose to go back to the settings menu if need be. At the bottom of the sidebar is a textbox where the user can send messages to the LLM. These messages typically undergo a variety of prompt-manufacturing processes, such as concatenating entire current chat logs to give the agent a sense of "memory" and default prompt preludes describing the agent's role and task, any relevant contexts from the language server, and few-shot prompting techniques, all dependent on which mode the user has the agent in. Finally, there are three tabs at the

[1] OpenRouter: `https://openrouter.ai/`

top of the sidebar representing each of the three agent modes, which the user can switch to, along with switching to the respective current chat for that mode by clicking on the mode's respective tab.

3.3.3 Three Modes

We offer specialized variations on the UI for three common use cases. To simplify the design and use process, we opt to separate distinct LLM/agent processes into three modes: Tutor, Suggest, and Compose.

i. Tutor

When operating in the tutor mode, there isn't so much the notion of an agent. Instead, the LLM acts as a helpful tutor to new or experienced users of the Hazel programming language. Prompted with the appropriate instructions regarding this role, Hazel's documentation slides, and a short blurb about Hazel, the agent can attempt to answer any questions the user has about Hazel. The agent responds with a concise, yet descriptive response to a user's inquiry and provides an optional code example. See A.2 for visual examples.

ii. Suggest

In this mode, we essentially make ChatLSP [13] accessible to the end-user through the front-end of Hazel. Here, the agent's goal is to provide suggested hole fillings in holes where the user types either '??' or '?a', the latter differing only in that it prompts the agent to use CoT. To construct the prompt in this mode, we describe the agent's basic instructions, incorporate few-shot prompting through input-output examples of correct hole fillings, gather all relevant context via the language server, and give a sketch of the code with the hole-to-be-filled in it.

If the agent responds with a valid filling free of syntax errors, the agent's code is suggested to the user as opaque code. The user can either accept this code by pressing tab on their keyboard or reject the agent's response by pressing anything other than tab. If the user changes their mind later about a suggestion, they can press a "resuggest" button below the respective code completion, where the suggestion will be reinserted into the editor as opaque code, given the hole still exists with the same unique identifier. See A.3 for visual examples.

When the agent responds with code involving syntax errors, the language server notifies the agent of these errors, where the agent then attempts to provide a new suggestion. By default, we set the error round limit to two.

iii. Compose

Here is where we plan to develop the task completion mode for the agent. The user should be able to describe a task in a specification-like manner, where the agent then, given our proposed structure-level action language, can take over and attempt to compose an implementation accordingly.

Chapter 4

Discussion

LLMs offer a vast range of possibilities in automatic program synthesis. With the creation of transformers taking place only 8 years ago [1], this area of study in coupling LLMs with agentic programming is relatively new, but is rapidly gaining popularity. We believe that equipping an agent with a high, structure-level action language will provide important semantical information and context relevant to the program and successful completion of user-provided specifications. CMTT [54] and gradual type theory [55] offer foundational groundwork in providing *typing contexts* via the language server. Hazel, being strongly rooted to gradual CMTT [12], offers the prime environment for gathering semantic-rich contexts and structures via its language server.

Our future intentions with this project lie in the implementation and evaluation of an LLM-based coding agent equipped with a high, structure-level action language, and tasked with the completion of user-provided specifications. Our prior work in the front-end development of an informative user interface and back-end integration of LLM API calls, language server static context retrieval, and prompt construction, set up a solid foundation to continue working from.

Chapter 5

Threats to Validity

The design of an LLM-based agent is akin to that of any other engineering task. Each component requires careful consideration and thoughtful construction.

5.1 Prompt Engineering

The prompt fed into an LLM can have significant effects on the output of an LLM, and ultimately, the agent itself [28] [40] [25]. Research supports Chain-of-Thought (CoT) reasoning in LLM prompt design [30] [22], especially in that of LM-based coding agents [35] [34] [3]. Our work is inherently one of prompt design, therefore even the slightest tweaks in a prompt may lead to substantial variations in the quality of results and performance of our agent. Furthermore, given the abundance of prompt engineering methods [22], choosing the right combination of techniques is by no means a trivial task. We plan to alleviate such threats to validity by incorporating modes in which our agent is prompted to use CoT reasoning or

not. From here, we can determine the effects that such known prompt designs have on the quality of the agent's performance.

5.2 LLMs are *Not* Human

We base much of our work on the idea that qualities which benefit human programmers may very well benefit LLM-based coding agents. However, it has been shown that evaluating LLMs as if they are humans can lead to misconstrued results, and they should rather be treated as their own system [38]. Humans and LLMs, by nature, differ in characteristics, training objectives, specialities, and limitations, and thus it follows that the design process for LLM-based coding agents can be seen as a meticulous analysis, which may very well unveil relationships between the two along the way [3]. The nature of attention in humans vs LLMs in programming tasks has also been shown to yield an attention misalignment between the two, suggesting that ultra-large language models tend to learn more deep and complex reasoning strategies compared to that of human programmers [46]. Ultimately, though our work on sculpting higher-quality agent-computer interactions derives much of its reasoning based on research in what has and has not worked for human programmers, our work has the potential to reveal connections, or a lack thereof, between human and AI programming.

5.3 Hazel

Hazel is a low-resource language, with very little data and documentation available. This makes it substantially more difficult for LLMs to perform well on coding tasks presented within this language, hence our resorting to few-shot prompting to aid the model in syntactical, structural, and semantical understandings of the language. Nevertheless, it is still unfavorable to extrapolate any findings found within this study to higher-resource languages. One approach used by [13] to condone the comparison between such a study on a low-resource language like Hazel was to also test, to an extent, on TypeScript. TypeScript presents notable parallels between Hazel and is a far higher resource language, thus making findings relative to it more appropriate to extend to other programming paradigms.

Chapter 6

Future Work

6.1 Current Work

6.1.1 Creating a Robust Benchmark

Prior benchmarks, such as that of SWE-Bench [9], are composed of real-world problems drawn predominantly from open-source Python repositories. Python is one of the highest-resource programming languages, especially with respect to Hazel, meaning the LLMs that coding agents make use of can far more easily pick up on patterns of the language itself directly from training data. Indeed, Python lacks the tools required by our agent to extract contextual information via the language server due to its dynamic type system and lack of a strict syntactic structure during editing. Moreover, SWE-bench makes use of open-source code drawn from 12 popular Python repositories. We ideally would like to generate our own benchmark of coding tasks within the Hazel language, and perhaps also a higher-resource language offering a language server with similar capabilities compared to that of Hazel's. Creating such a benchmark completely from scratch also helps to avoid data contamination stemming from the direct training of LLMs on code within the benchmark. MVUBench [13] offers a great stepping stone of a benchmark to continue building upon.

6.1.2 Implementing and Evaluating an Agent for Task Completion

Much of the research done in this paper has yet to be fully completed, and there remains work to be done in evaluating the architecture and methods laid out. To offer a holistic answer to each of our research questions posed earlier 1.3, we plan to conduct a user study on the impacts LLM-based agents have on learning the Hazel programming language and functional programming as a whole. Furthermore, we aim to implement and evaluate our proposed high, structure-level action language within Hazel, building upon the architecture currently laid out.

6.2 On-The-Fly Code Completion

Hazel presents to be a very strong candidate for implementing a coding agent capable of on-the-fly code completion. Takerngsaksiri, W. et al. [14] found that on-the-fly code completion is largely limited by a lack of syntactic information, as "for every two-thirds of characters that developers type, AST [information] fails to be extracted because it requires the syntactically correct source code". Due to Hazel's typed holes, cursor, and bidirectional type system, its programming states are always meaningful and syntactically valid [11] [12]. The research discussed in this paper and existing research performed by Blinn, A. et al. [13] on ChatLSP

set up solid foundations to study the efficacy of on-the-fly code completion and suggestions in the Hazel programming environment.

References

[1] Vaswani, A. et al. Attention is all you need. (2017).

[2] Backus, J. et al. The Fortran Automatic Coding System. (1957).

[3] Yang, J. & Jimenez, C. et al. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. (2024)

[4] Austin, & Odena, et al. Program Synthesis with Large Language Models. (2021).

[5] Guo, D. & Zhu, Q. et al. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. (2024).

[6] Fan, Z. et al. Automated Repair of Programs from Large Language Models. (2023).

[7] Tang, X. et al. ML-BENCH: Evaluating Large Language Models and Agents for Machine Learning Tasks on Repository-Level Code. (2024).

[8] Lai, Y. et al. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. (2022).

[9] Jimenez, C. & Yang, J. et al. SWE-BENCH: CAN LANGUAGE MODELS RESOLVE REAL-WORLD GITHUB ISSUES? (2024).

[10] Wang, X. et al. OPENHANDS: AN OPEN PLATFORM FOR AI SOFTWARE DEVELOPERS AS GENERALIST AGENTS (2024).

[11] Omar, C. et al. Toward Semantic Foundations for Program Editors (2017).

[12] Omar, C. et al. Hazelnut: A Bidirectionally Typed Structure Editor Calculus (2017).

[13] Blinn, A. et al. Statically Contextualizing Large Language Models with Typed Holes (2024).

[14] Takerngsaksiri, W. et al. Syntax-aware on-the-fly code completion (2024).

[15] Kline & Seffah. Evaluation of integrated software development environments: Challenges and results from three empirical studies (2005).

[16] Afzal & Goues. A Study on the Use of IDE Features for Debugging (2018).

[17] Bergström, A. A Survey on Developers' Preferences in Integrated Development Environments (2018).

[18] Abramovich, T. et al. EnIGMA: Enhanced Interactive Generative Model Agent for CTF Challenges (2024).

[19] Wu, Z. et al. OS-COPILOT: TOWARDS GENERALIST COMPUTER AGENTS WITH SELF-IMPROVEMENT (2024).

[20] Jiang, N. et al. KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair (2023).

[21] Miserendino & Wang et al. SWE-Lancer: Can Frontier LLMs Earn $1 Million from Real-World Freelance Software Engineering? (2025).

[22] Sahoo, P. et al. A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications (2024).

[23] Brown, Mann, Ryder, & Subbiah et al. Language Models are Few-Shot Learners (2020).

[24] Zhong, L et al. Debug like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step by Step (2024).

[25] Lunemark, A. `https://www.cursor.com/blog/prompt-design` (2023).

[26] Gu, Y. et al. Middleware for LLMs: Tools Are Instrumental for Language Agents in Complex Environments (2024).

[27] Li & Zhang et al. More Agents Is All You Need (2024).

[28] Shinn, N. et al. Reflexion: Language Agents with Verbal Reinforcement Learning (2023).

[29] Sprague, Z. et al. TO COT OR NOT TO COT? CHAIN-OF-THOUGHT HELPS MAINLY ON MATH AND SYMBOLIC REASONING (2024).

[30] Wei, J. et al. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models (2023).

[31] Jiang, H. et al. CURSORCORE: ASSIST PROGRAMMING THROUGH ALIGNING ANYTHING (2024).

[32] Cassano, F. et al. Can It Edit? Evaluating the Ability of Large Language Models to Follow Code Editing Instructions (2024).

[33] Gauthier, P. Aider is ai pair programming in your terminal `https://github.com/paul-gauthier/aider` (2024).

[34] Liu, J. et al. Large Language Model-Based Agents for Software Engineering: A Survey (2024).

[35] Huang & Bu et al. CodeCoT: Tackling Code Syntax Errors in CoT Reasoning for Code Generation (2024).

[36] Yilmaz, R. et al. The effect of generative artificial intelligence (AI)-based tool use on students' computational thinking skills, programming self-efficacy and motivation (2023).

[37] Yang, J. et al. Intercode: Standardizing and benchmarking interactive coding with execution feedback (2023).

[38] McCoy, T. et al. Embers of Autoregression: Understanding Large Language Models Through the Problem They are Trained to Solve (2023).

[39] LaToza, T. et al. Explicit Programming Strategies (2019).

[40] Author Unkown. Prompt engineering overview `https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/overview` (2024).

[41] Zhou, D. et al. LEAST-TO-MOST PROMPTING ENABLES COMPLEX REASONING IN LARGE LANGUAGE MODELS (2023).

[42] Li, Z. et al. Chain of Thought Empowers Transformers to Solve Inherently Serial Problems (2024).

[43] Zhao, E. et al. Total Type Error Localization and Recovery with Holes (2024).

[44] Li, Y. et al. Enhancing LLM-Based Coding Tools through Native Integration of IDE-Derived Static Context (2024).

[45] Licorish, S. et al. Comparing Human and LLM Generated Code: The Jury is Still Out! (2025).

[46] Kou, B. et al. Do Large Language Models Pay Similar Attention Like Human Programmers When Generating Code? (2024).

[47] Bansal, A. et al. Towards Modeling Human Attention from Eye Movements for Neural Source Code Summarization (2023).

[48] Nijkamp, Pang, & Hyashi. et al. CODEGEN: AN OPEN LARGE LANGUAGE MODEL FOR CODE WITH MULTI-TURN PROGRAM SYNTHESIS (2023).

[49] OpenAI. HumanEval: Hand-Written Evaluation Set `https://github.com/openai/human-eval` (2021).

[50] Yang & Liu. et al. If LLM Is the Wizard, Then Code Is the Wand: A Survey on How Code Empowers Large Language Models to Serve as Intelligent Agents (2024).

[51] Pei, H. et al. Better Context Makes Better Code Language Models: A Case Study on Function Call Argument Completion (2023).

[52] DeOrio, A. GenAI in computer science education: Friend or foe? (2024).

[53] Pashakhanloo, P. et al. CODETREK: FLEXIBLE MODELING OF CODE USING AN EXTENSIBLE RELATIONAL REPRESENTATION (2022).

[54] Nanevski, A. et al. Contextual Modal Type Theory (2008).

[55] Siek, J. et al. Refined Criteria for Gradual Typing (1998).

Appendix A

Appendix

A.1 ACIs

Table A.1: The action language we propose in this paper. Required arguments are enclosed in <>. The last column shows the documentation presented to the LLM.

Category	Command	Documentation
File Viewer	**goto_definition** `<variable_name>`	Places the cursor at the nearest definition of the matched `variable_name`.
	goto_type_definition `<variable_name>`	Places the cursor at the nearest type definition for the matched `variable_name`.
	show_references `<variable_name>`	Displays a list of definitions where the variable matching `variable_name` is referenced.
	scroll_down	Moves cursor to preceding definition, if one exists.
	scroll_up	Moves cursor to succeeding definition, if one exists.
File Editing	**replace** `<replacement_text>`	Replaces the text of the current definition with `replacement_text`
Task	**submit**	Ends the iterative task completion process.

Table A.2: The action language used in SWE-agent's ACI. Required arguments are enclosed in <> and optional arguments are in []. The last column shows the documentation presented to the LLM. Directly sourced from Yang, J. & Jimenez, C. et al. [3].

Category	Command	Documentation
File viewer	**open** <path> [<line_number>]	Opens the file at the given path in the editor. If line_number is provided, the window will move to include that line.
	goto <line_number>	Moves the window to show line_number.
	scroll_down	Moves the window up 100 lines.
	scroll_up	Moves the window down 100 lines.
Search tools	**search_file** <search_term> [<file>]	Searches for search_term in file. If file is not provided, searches in the current open file.
	search_dir <search_term> [<dir>]	Searches for search_term in all files in dir. If dir is not provided, searches in the current directory.
	find_file <file_name> [<dir>]	Finds all files with the given name in dir. If dir is not provided, searches in the current directory.
File editing	**edit** <n>:<m> <replacement_text> **end_of_edit**	Replaces lines n through m (inclusive) with the given text in the open file. All of the replacement_text will be entered, so make sure your indentation is formatted properly. Python files will be checked for syntax errors after the edit. If an error is found, the edit will not be executed. Reading the error message and modifying your command is recommended as issuing the same command will return the same error.
	create <filename>	Creates and opens a new file with the given name.
Task	**submit**	Generates and submits the patch from all previous edits and closes the shell.

A.2 LLM-based Tutor in Hazel

New and experienced users of Hazel can interact with an LLM to navigate their way through the syntactical and semantical forest of Hazel using the agent's "Tutor" mode. The following examples feature the user asking a relatively basic programming-related question, where the

LLM then responds with an informative, but concise response and an optional code example. These examples were made using the Claude Sonnet 3.5 large language model.

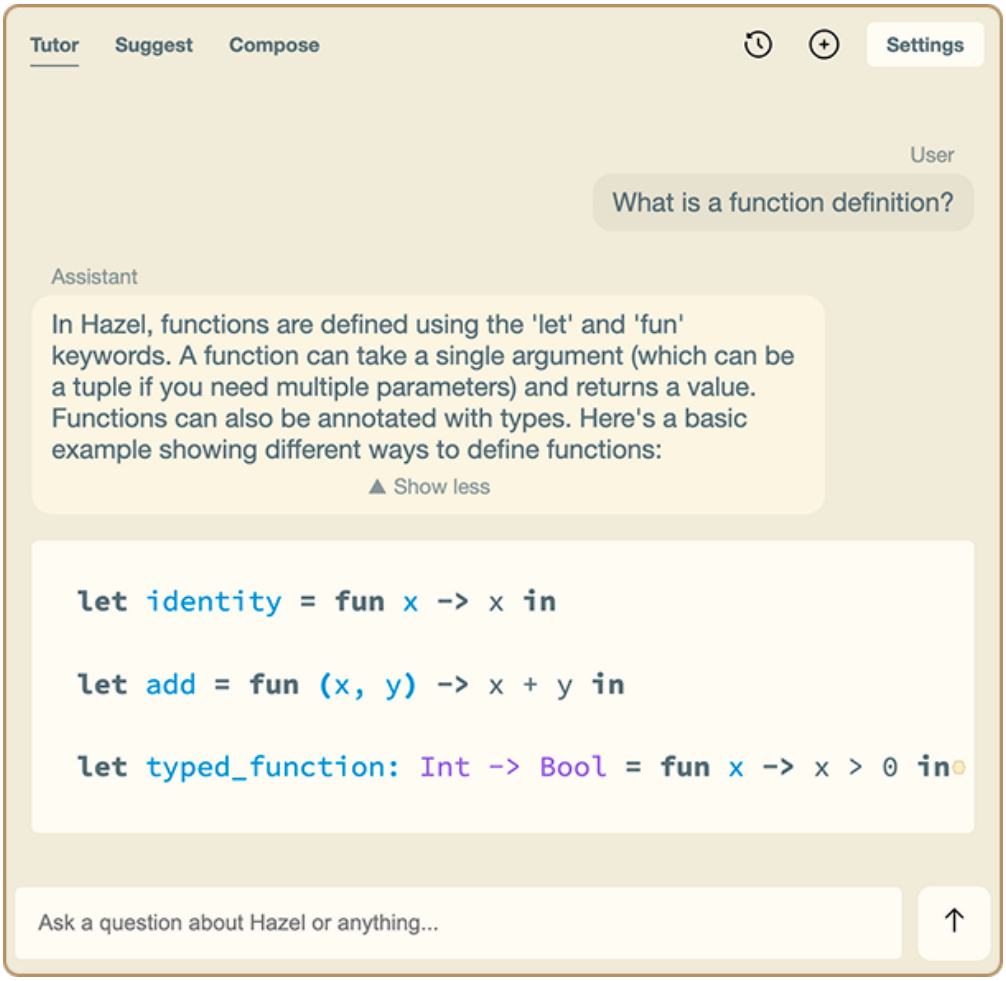

Figure A.1: A simple question about function definitions. The agent describes how function definitions work in Hazel, followed by an example code snippet containing three example use cases.

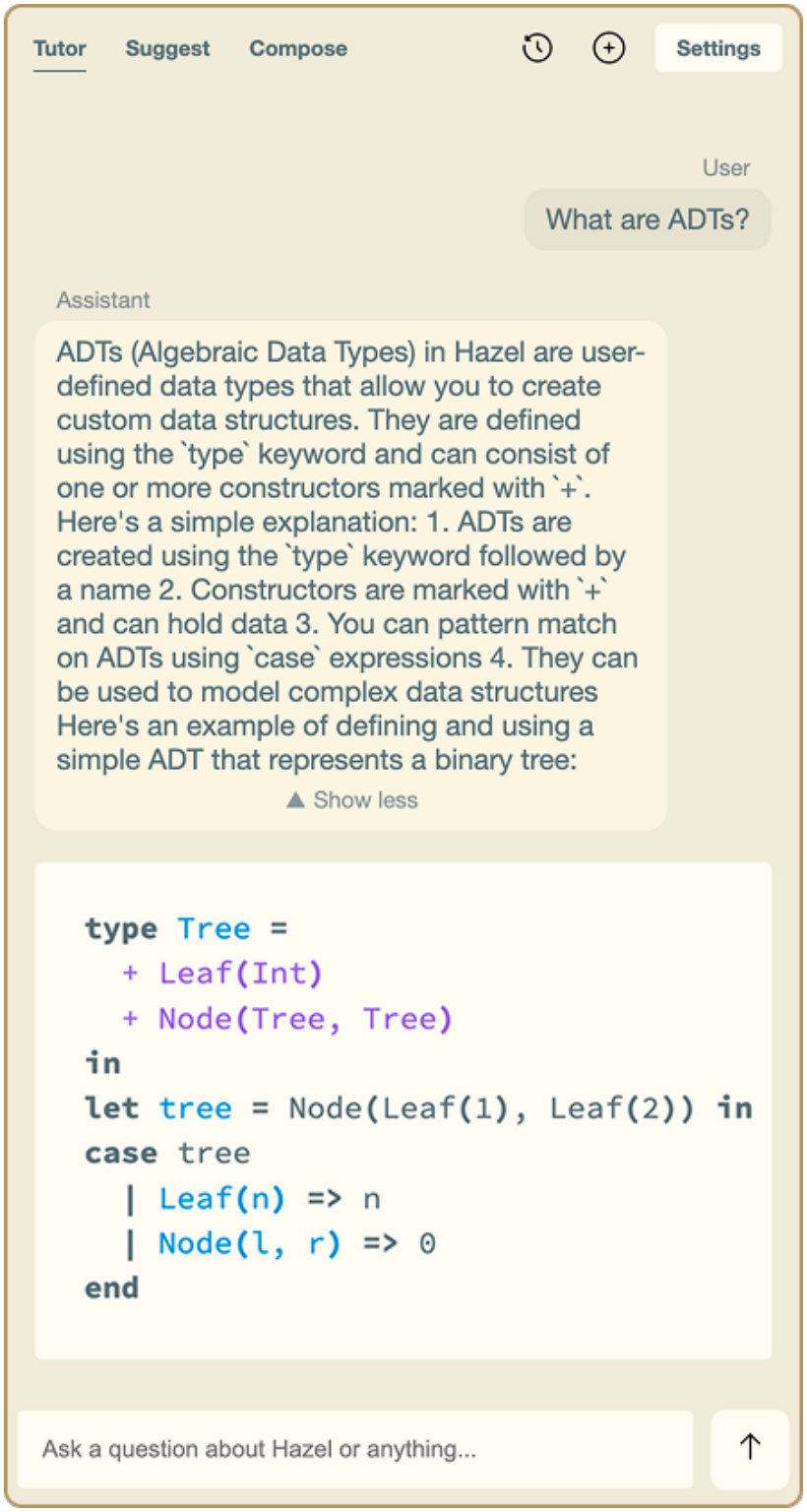

Figure A.2: A simple question about abstract data types (ADTs). The agent describes how ADTs are used in Hazel, followed by an example code snippet.

A.3 Agentic Code Suggestions in Hazel

The following code suggestions were completed using the Claude Sonnet 3.5 large language model. These examples are quite trivial, thus don't require a lot of context from the language server or reasoning from the LLM. Rather, they serve as a visual demonstration of the user interface that users can directly interact with from Hazel's front-end.

Figure A.3: The user requests a hole filling after the definition of the rec_fib variable. The prompt is concatenated with relevant static context via the language server (collapsed for brevity) and a program sketch. The user uses '??', thus not prompting the LLM to use CoT. The LLM responds with the suggested filling for the hole at the location of the '??'. The user may press tab to accept this request, which pastes the suggestion into the editor.

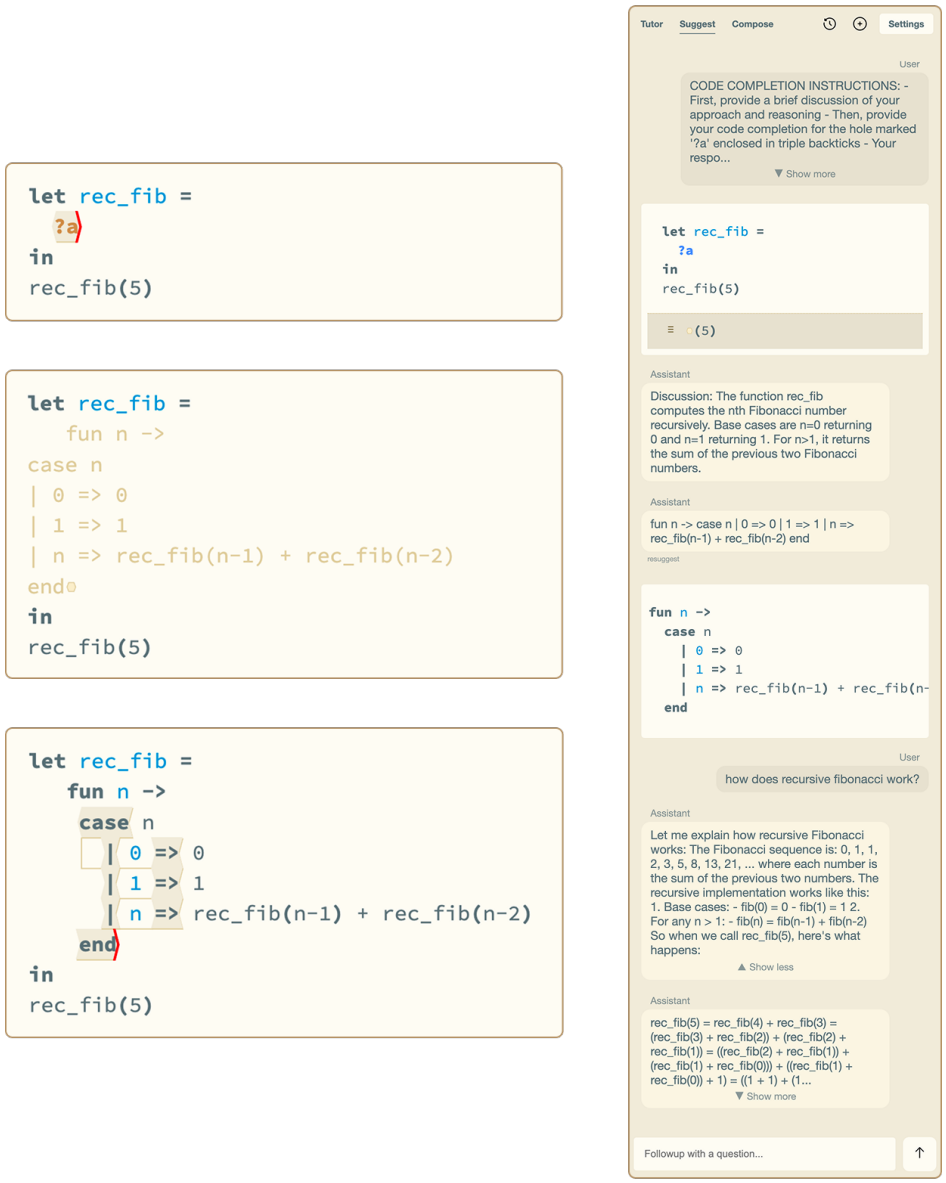

Figure A.4: The user requests a hole filling for the definition of the rec_fib variable. The prompt is concatenated with relevant static context via the language server (collapsed for brevity) and a program sketch. The user uses '?a', prompting the LLM to use CoT. The LLM's response is parsed using regular expressions into a brief discussion containing its CoT reasoning and the suggested filling for the hole at the location of the '?a'. The user may press tab to accept this request, which pastes the suggestion into the editor. In this example, the user follows up with a question about the recursive Fibonacci function, where the agent then provides an answer and worked example (collapsed and cutoff for brevity).