Kamp,
Enhancing AI Coding Agents

# Verbesserung von AI Coding Agents zur Truffle-DSL Performance Optimierung

## *Enhancing AI Coding Agents for Truffle DSL Performance Optimization*

by

Antony Kamp

A thesis submitted to the
Hasso Plattner Institute
at the University of Potsdam, Germany
in partial fulfillment of the requirements for the degree of

Master of Science in IT-Systems Engineering

Supervisors

Prof. Dr. Robert Hirschfeld
Dr. Jens Lincke

Software Architecture Group
Hasso Plattner Institute
University of Potsdam, Germany

July 15, 2026

# Abstract

Building Domain-Specific Languages with the Truffle framework on GraalVM promises competitive performance, but achieving it requires Truffle-specific profiling and compiler knowledge that most language developers lack. Profiling tools like the CPU Sampler, Trace Compilation, and the Ideal Graph Visualizer demand deep framework expertise to interpret, leaving developers without effective guidance during performance optimization. Existing AI coding agents attempt to address this gap but optimize through static reasoning without benchmark validation, often introducing regressions rather than improvements.

To help language developers navigate Truffle's performance optimization workflow, this thesis proposes a layered knowledge enhancement architecture for AI coding agents with three levels: Tool Assistance via agent skills for individual profiling tools, Process Assistance via sub-agents for optimization process phases, and Full Orchestration. The architecture is implemented as a Claude Code plugin combining agent skills, sub-agents, and an MCP server for the "Are We Fast Yet" benchmark framework. It incorporates domain knowledge sourced from a contextual inquiry interview with a Truffle expert.

To evaluate whether this enhanced agent meaningfully supports developers, the evaluation contains a controlled experiment on an original and a degraded language implementation, comparing unenhanced and enhanced Agents in terms of speedup. It is extended with a qualitative user study of two developers, analyzed via reflexive thematic analysis. On the degraded Implementation, the enhanced Agent achieved a significantly higher speedup ($U = 96$, $p < 0.001$, $r = 0.92$). On the already well-optimized original Implementation, results were inconclusive ($U = 62$, $p = 0.192$, $r = 0.24$). The enhanced Agent followed a more hypothesis-driven and Truffle-focused approach at four to five times the API cost. Both study participants resolved a performance issue with the enhanced Agent while neither succeeded with the manual CLI setup.

These results demonstrate that layered knowledge enhancement is most impactful when developers face an unfamiliar or degraded codebase. The plugin reduced translation cost from thought to tool use. Together, these findings establish that AI agents equipped with domain-specific profiling knowledge can lower the barrier for language developers to achieve competetive performance.

# Zusammenfassung

Die Entwicklung von domänenspezifischen Sprachen mit dem Truffle Framework und der GraalVM verspricht wettbewerbsfähige Leistung, erfordert jedoch Truffle-spezifische Profiling- und Compilerkenntnisse, die den meisten Entwicklern fehlen. Profiling-Tools wie der CPU Sampler, Trace-Compilation und der Ideal Graph Visualizer erfordern tiefgreifende Framework-Kenntnisse zur Interpretation und lassen Entwickler ohne effektive Orientierung bei der Leistungsoptimierung. Bestehende AI Coding Agents versuchen diese Lücke zu schließen, optimieren jedoch durch statische Analyse ohne Benchmark-Validierung, was häufig zu Verschlechterungen statt Verbesserungen führt.

Um Sprachentwicklern bei der Navigation des Truffle Optimierungs Workflows zu helfen, schlägt diese Arbeit eine mehrschichtige Architektur zur Wissenserweiterung von AI Coding Agents mit drei Ebenen vor: Tool Assistance durch Agent Skills für einzelne Profiling-Tools, Process Assistance durch Subagenten für einzelne Optimierungsphasen, sowie Full Orchestration. Die Architektur ist als Claude Code plugin implementiert, das Agent Skills, Subagenten und einen MCP-Server für das AWFY Benchmarking Framework kombiniert. Sie beinhaltet zudem Domänenwissen, das aus einem Interview mit einem Truffle-Experten gewonnen wurde.

Um zu evaluieren, ob der erweiterte Agent Entwickler sinnvoll unterstützt, umfasste die Evaluierung ein kontrolliertes Experiment mit einer originalen und einer verschlechterten Sprachimplementierung, wobei die Beschleunigung von nicht erweiterten und erweiterten Coding Agenten verglichen wurde. Ergänzt wird sie von einer qualitative Nutzerstudie mit zwei Entwicklern, die mittels reflexiver thematischer Analyse ausgewertet wurde. Bei der verschlechterten Implementierung erzielte der erweiterte Agent eine signifikant höhere Beschleunigung ($U = 96, p < 0,001, r = 0,92$). Bei der bereits schnelleren originalen Implementierung waren die Ergebnisse nicht eindeutig ($U = 62$, $p = 0,192, r = 0,24$). Der erweiterte Agent verfolgte einen stärker hypothesengetriebenen und auf Truffle fokussierten Ansatz, verursachte jedoch das Vier- bis Fünffache der API-Kosten. Beide Studienteilnehmer konnten ein Leistungsproblem mit dem erweiterten Agent lösen, während die manuelle Einrichtung über die Befehlszeilenschnittstelle bei keinem von ihnen erfolgreich war.

Diese Ergebnisse zeigen, dass die mehrschichtige Wissenserweiterung am wirkungsvollsten ist, wenn Entwickler mit einer unbekannten oder verschlechterten Codebasis konfrontiert sind. Das Plugin reduzierte die Übersetzungskosten von der Idee bis zur Werkzeugnutzung. Diese Erkenntnisse zeigen gemeinsam, dass AI Agents mit domänenspezifischem Profiling-Wissen die Hürde für Sprachentwickler senken können, um wettbewerbsfähige Leistung zu erreichen.

# Acknowledgments

This thesis benefited from the support of several people. I would like to thank Prof. Dr. Robert Hirschfeld for the opportunity to conduct this work in the Software Architecture Group at the Hasso-Plattner Institute, and for the provided resources.

I want to thank my supervisor Dr. Jens Lincke for the discussions, the feedback, and the patience over the course of the project.

Dr. Tim Felgentreff provided the contextual inquiry interview that grounded chapter 3 and the agent design. I am grateful for the time, the openness and additional feedback to the architecture over time.

I further thank the Software Architecture Group for the feedback on the exposé, the BYOPL teaching team for the seminar implementations used as evaluation subjects, and the two study participants for their engagement.

# Contents

# List of Figures

# List of Tables

# List of Listings

# List of Algorithms

# List of Abbreviations

| | |
|---|---|
| DSL | Domain Specific Language |
| API | Application Programming Interface |
| AWFY | Are We Fast Yet |
| LLM | Large Language Model |
| AI | Artificial Intelligence |
| JIT | Just-In-Time |
| AOT | Ahead-Of-Time |
| AST | Abstract Syntax Tree |
| CLI | Command-Line Interface |
| BYOPL | Building Your Own Programming Language |
| VM | Virtual Machine |
| MCP | Model-Context Protocol |
| RAG | Retrieval-Augmented Generation |
| CDN | Cognitive Dimensions of Notations |
| CPU | Central Processing Unit |
| IGV | Ideal Graph Visualizer |
| IR | Intermediate Representation |

# 1. Introduction

Building your own programming language was a dark art for a long time. Writing a compiler by hand was required, which limited language development to compiler experts [2]. Nowadays, modern frameworks abstract compiler construction, which enables more developers to work on their own language [89].

In the industry, it allows companies to build their own Domain Specific Languages (DSLs) without requiring specialized developer teams. A DSL is a programming language that focuses on one problem area, the domain [30, 83]. The goal is to support the developers working within the domain. Each language offers domain-specific expressiveness at the cost of general applicability.

Multiple frameworks support DSL development. Among the publicly available DSL frameworks, GraalVM with the Truffle framework has emerged as a powerful choice [89]. Truffle abstracts low-level implementation concerns, allowing the domain developers to focus on domain-specific features. Using Truffle, the barrier to developing a new language continues to lower, making DSL development available to a broader range of developers and researchers.

However, while building a *functional* DSL is easier now, building a *fast* one is still a challenge. High performance is critical for DSLs to save energy, reduce computational costs [69], and provide a good user experience.

To support the optimization process, Truffle provides a range of profiling tools, like the CPU-Sampler [71] or the possibility to investigate the compiled code visually [38].

Language developers must still understand underlying concepts of the Truffle framework to interpret profiling results correctly and validate optimization hypotheses. Typically, regular domain developers lack this expertise because they focus on domain-specific features and not compiler details. This expertise gap between accessible language implementations and effective performance optimizations is the central challenge this thesis addresses.

This expertise gap requires guidance from experienced Truffle developers. IIdeally, domain developers would have access to experts to guide them in using profiling tools and interpreting the results. Experts can educate domain developers about the relative importance of different tools and show a reliable optimization process.

However, experts are expensive, scare, and not always available in practice, creating a need for scalable guidance mechanisms. Recent advances in Large

Language Models (LLMs) have enabled Artificial Intelligence (AI) coding agents that present a promising, always available alternative [21]. Coding agents are increasingly contributing to open-source software [86]. They offer an always accessible alternative to human experts.

This thesis investigates the use of AI coding agents for performance optimization of Truffle languages to support language developers. Optimizing a language implementation is a complex task. It requires framework-specific knowledge to interpret profiling data and it demands a multi-step reasoning process from hypothesis to verification. Peng et al. found empirically that AI agents are less likely to validate performance improvements through benchmarking, they rely instead on static reasoning [68]. This can have the opposite effect and reduce performance. It aligns with the developer perception: In the Stack Overflow 2025 developer survey, over 40% of professional developers rated AI as poor at handling complex tasks [1]. Therefore, existing agents require enhancement.

## 1.1. Research Scope and Contributions

This thesis investigates how AI coding agents can be enhanced to support performance optimization of Truffle languages. The figure 1.1 holds an overview over research scope and contributions.

Therefore, we conducted an interview with a Truffle expert to collect knowledge about his team's optimization process and tools. Afterwards, we enhanced an AI coding agent with that knowledge. Additionally, we evaluated how well an unenhanced and enhanced AI coding agent can improve a language implementation. For the enhancement, this thesis proposes a set of knowledge adapters — agent skills, sub-agents, and tool extensions — to guide Truffle performance optimization. These mechanisms are introduced in chapter 2.

In this context, we define *optimal runtime* with the "Are We Fast Yet" benchmark framework introduced by Marr, Daloze, and Mössenböck, because of its focus on the language implementation [50].

This research does not focus on the completeness of the profiling tools provided by GraalVM. Additionally, this research does not focus on comparing different coding agents. Instead, the thesis focuses on a generally applicable approach that can be adopted by multiple agents, because these mechanisms rely on standard prompting and tool-calling interfaces. We evaluate using Claude Code [23] with Claude Sonnet 4.5 model as the representative platform, due to its leading performance on SWE-bench Verified [11].

**Research Context:** This research uses implementations of the simple scripting language "Lox" [58] from the seminar Building Your Own Programming Language (BYOPL)[1] as evaluation subjects [48]. The quality between the implementations varies, which makes it suitable for evaluation. This variance provides a natural spectrum of optimization opportunities.

## 1.2. Thesis Structure

This work is structured as follows:

CHAPTER 2 (FOUNDATIONS) introduces foundational knowledge about GraalVM, Truffle, AI, coding agents and AI enhancement approaches.
CHAPTER 3 (UNDERSTANDING EXPERT OPTIMIZATION PRACTICE) contains an analysis of existing performance analysis tools and optimization strategies. It includes a structured interview with a Truffle expert.
CHAPTER 4 (DESIGNING THE OPTIMIZATION ASSISTANT) describes the solution derived from the interview.
CHAPTER 5 (IMPLEMENTING THE CLAUDE CODE PLUGIN) discusses implementation details of and learnings from the development.
CHAPTER 6 (STUDY DESIGN) describes the experimental design and user study methodology. Additionally, it discusses the threats to validity.
CHAPTER 7 (RESULTS & DISCUSSION) presents the results from the experiment and the user study. Interprets the results and discusses limitations.
CHAPTER 8 (RELATED WORK) Presents related work and compares approach with it.
CHAPTER 9 (CONCLUSION) summarizes all findings and contributions, and discusses future work.

[1] Hasso-Plattner Institute Potsdam, Winter semester 24/25 by Lincke et al.

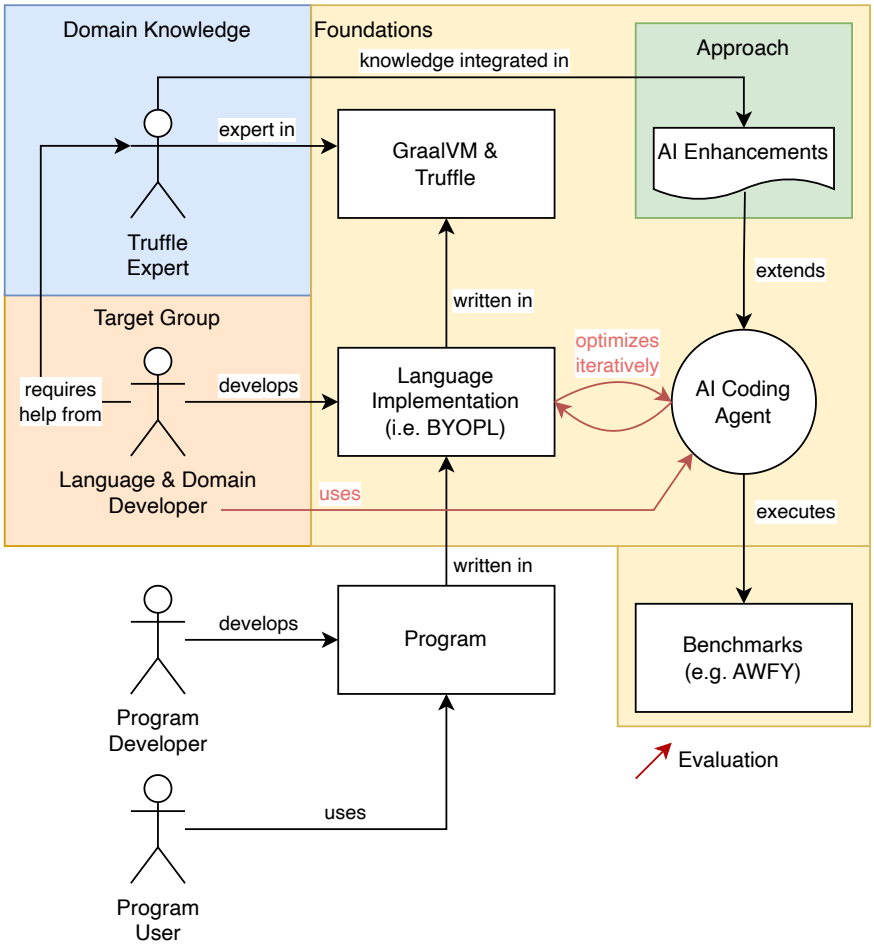



Figure 1.1.: Overview of the research scope and contributions. Visible are four actors, who use a different level of software. The thesis focuses on the language implementation level, developed by the language developer. The enhanced AI coding agent will optimize it with the help of benchmarks.

# 2. Foundations

This chapter introduces GraalVM and Truffle, detailing their architecture, applications and the running example of the Lox language. This chapter also presents the concept of profiling and analysis tools and the Are We Fast Yet benchmark framework, which are used to evaluate the language's performance. Since coding agents will play a significant role in this thesis, the chapter will explain the underlying concept of a LLM before discussing AI coding agents and their enhancement mechanisms. Additionally, it covers an overview of a vocabulary for evaluating developer experience, called Cognitive Dimensions of Notations (CDN).

## 2.1. Introduction of Graal & Truffle

GraalVM is a high-performance Java Virtual Machine (VM) implementation. Its relevance in the industry is underscored by the adoption rate, for instance, the community edition of JDK 21.0.2 has reached over 1.3 million total downloads via the GitHub API [63]. Written in Java, it handles both JVM and non-JVM languages, achieving high performance through speculative optimizations relying on profiling data [89].

The compiler is available as a Just-In-Time (JIT) compiler and an Ahead-Of-Time (AOT) compiler. The JIT compiler first runs code in interpreter mode and compiles it based on profiling data during execution. The AOT compiler compiles the program into a standalone executable before runtime. We focus on the JIT compiler because it is optimized for long-running systems, which language programs typically are [89].

For JIT compiler, GraalVM employs a multi-tier optimization system. Code initially runs in the interpreter. When a function is executed frequently, it moves to Low-tier and gets compiled. If the function is executed even more, it advances to High-tier and is compiled again using more aggressive optimization techniques. Optimizations rely on speculations and assumptions. If an assumption fails for compiled code, then it gets deoptimized and moved to the interpreter again, resulting in a massive performance regression [88].

At each compilation tier, the compiler transforms the code into an Intermediate Representation (IR), a graph-based structure of logic nodes and data flow edges. The IR is the artifact that the compiler optimizes at each tier. Understanding the IR becomes relevant when diagnosing compilation issues.

Truffle is a language implementation framework integrated into GraalVM. It enables developers to build their own programming languages and DSLs by providing reusable components. These languages are called guest languages and they are written in the host language Java. Two interpreter architectures are available. The Abstract Syntax Tree (AST) interpreter traverses the tree, while the Bytecode interpreter generates bytecode from an AST-like specification. Additionally, Truffle supports polyglot interoperability, which allows guest languages to call other languages and to share data between them. Truffle languages achieve high performance because of GraalVM's speculative optimizations, including Inlining, Constant Folding and Partial Evaluation [89].

A common example for a language implementation with Truffle is GraalPy, a high-performance Python implementation [66]. An example for a DSL written with Truffle is Apple Pkl, a configuration-as-code language [15, 39].

## 2.2. Profiling and Analysis Tools Concepts

Among the components developers can reuse from GraalVM and Truffle are various profiling tools. Profiling is a sub-process of optimizing the performance of a program. The goal is to understand where time is spent during the execution to find bottlenecks. A profiling tool tracks the behavior of a program, producing an output called a profile. Developers use the resulting profile for further analysis and optimizations [82].

There are two types of profiling tools available: samplers and tracers. Samplers are statistical profilers, which collect statistically random snapshots of the stack trace. From the captured executed code, the sampler derives metrics, such as the time spent in a function. These results can be visualized in a flamegraph [82].

In contrast, tracers are instrumentation profilers that track events during execution. Typical events include function calls or compilations. These captured events allow the tracer to derive metrics, such as the number of function calls [82].

Developers use profilers to find the spots where the program is slow. Complex issues related to compilation require deeper insights to identify the underlying cause. GraalVM provides a way to dump the IR between compilation phases. The graphs can be visualized with the Ideal Graph Visualizer (IGV) [38], where the compiled code is represented by "sea-of-nodes" containing logic nodes and data flow edges. Developers use it to identify where the compilation failed and, where performance slowed down [31].

Analyzing compiler graphs is a complex task, because users have to understand the IR and the compilation process. Consequently, they need support from experts. This research will investigate whether coding agents can lower this barrier by assisting developers with this task.

## 2.3. Are We Fast Yet Benchmark Framework

Benchmarks help developers to evaluate the execution performance of a program. The same applies to language implementation. Language developers can use standard benchmarks to evaluate the performance of the compiler. A prominent example is the Are We Fast Yet (AWFY) benchmark framework [50]. It consists of 14 benchmark programs and a methodology to measure compiler effectiveness across language implementations. Because it focuses on compiler behavior rather than language design, it can compare different compilers of the same or different languages. It achieves language agnosticism by requiring identical behavior in the benchmark's code.

Each benchmark focuses on one workload characteristic, which exercises compiler optimization techniques. For instance, the Sieve benchmark, which executes the Sieve of Eratosthenes algorithm, consists of many array operations, which exercises bounds-check elimination [50].

A harness script wraps the benchmark target function in two loops. The outer loop records the duration of the inner loop. The average, the fastest and the slowest inner iteration are relevant for further analysis. The goal of the loops is to obtain multiple durations and evaluate the stability of the measurements. Additionally, GraalVM requires several iterations before performance stabilizes because of its multi-tier optimization system. Algorithm 2.1 shows the inner loop has 100 and the other loop 3000 iterations [50, 89].

**Algorithm 2.1** AWFY methodology

```
1: procedure HARNESS(target)
2:     for i in [0, 3000] do
3:         before ← now()
4:         for j in [0, 100] do
5:             target()
6:         after ← now()
7:         print : after − before
```

Developers use the the geometric mean of the normalized execution times to summarize performance across multiple benchmarks into a single comparable metric [29]. The normalization happens over a baseline, for instance the runtime performance before a code change. In that case, it equals a performance speedup metric. The geometric mean has been criticized for lacking direct physical interpretation as an execution time [27]. Nevertheless, it remains the most appropriate summary statistic for our evaluation. The chapter 4 and chapter 6 will compare normalized performance ratios across benchmarks with different absolute runtimes.

## 2.4. Evaluation Subject: Lox Language

In this thesis, the development and evaluation are performed on the simple scripting language *Lox*. Lox supports basic arithmetic and logical operations, comparisons, variable declarations, control flow, and functions. It also includes more complex language features, such as classes and closures [58].

There are different implementations of the language publicly available, for instance a C and a Java implementation by the author Nystrom [57]. The Software Architecture group at the Hasso-Plattner Institute published an implementation written with Truffle and GraalVM called *swalox* [47].

In the seminar BYOPL in Winter Semester 2024/2025 at the Hasso-Plattner Institute, the students built their own Lox language implementations throughout the semester [43]. Even though they had the same material, the performance between the implementations differed by up to two orders of magnitude in the AWFY benchmark in our preliminary evaluation. This thesis will use the student implementations for development and evaluation purposes.

## 2.5. Large Language Model

Recent developments in LLMs have opened a new solution space for complex tasks. LLMs are large-scale neural language models that generate text by predicting the next most probable token given a sequence of input tokens [54].

The LLMs are trained on data from the internet, including code developer platforms. For example, the open-source model DeepSeek Coder used 87% code and 10% code-related language from GitHub in the pre-training step [40]. Because code is part of their training data, they can understand and generate code in many widely-used programming languages.

### 2.5.1. Limitations

LLMs have broad capabilities, but some inherent limitations. One limited factor is the context, which equals the full sequence of tokens visible to the model when generating a the next token. The context size is limited, and when the input exceeds the *context window*, it must be compressed, resulting in information loss [85].

Moreover, LLMs do not use long contexts uniformly. Liu et al. found that performance is highest when relevant information appears at the beginning or end of the input context, but degrades significantly when it is placed in the middle [49]. This positional bias also means that when long contexts contain irrelevant information, the LLM has difficulty identifying what is really relevant [49].

Additionally, Wang, Min, and Zou showed that LLM performance can collapse when input length reaches 40%−50% of the model's maximum context window [84]. This creates a design tension between enhancing knowledge and preserving quality. However, this threshold was established for the model Qwen2.5-7B and may not generalize to all architectures. These limitations are relevant for performance optimization tasks, where profiling outputs can span thousands of lines.

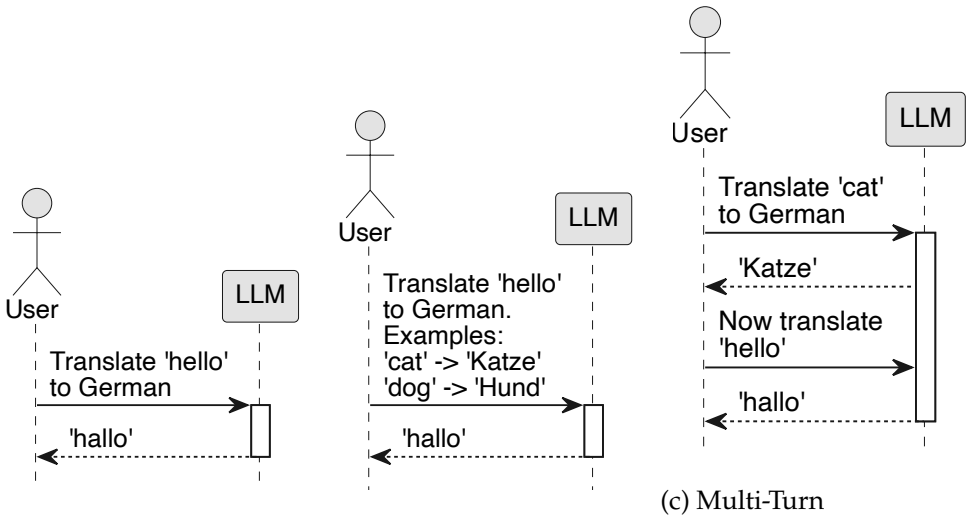
### 2.5.2. Prompting Patterns

Given these limitations, how the user utilizes the resulting model matters for the LLM's performance. Querying a trained LLM is called prompting. Given a prompt, the LLM generates the response by predicting the next most probable token over and over again. Different patterns for the interaction between user and LLMs exist [54]. The figure 2.1 contains examples for each of them.

Zero-Shot: Requires the LLM to answer a user input without any further information. Only the input is the context.

In-context Learning: In addition to the user input, the LLM receives desired input-output demonstrations. These examples together with the input form the context.

Multi-Turn Conversation: The LLM and the user have multiple iterations: The user's input and model's responses are the additional input for the next iteration's input. The previous conversation before an input is also part of the input's context.




(a) Zero Shot Prompting (b) In-Context Learning (c) Multi-Turn Conversation

Figure 2.1.: Examples of different prompting strategies. The User asks the LLM to translate into German.

Each pattern involves a trade-off between knowledge availability and context cost. Zero-shot prompting consumes minimal context but relies entirely on the model's training data, which may lack domain-specific knowledge. In-context learning injects relevant examples but increases token usage proportionally. Multi-turn conversation accumulates the richest context, but risks exceeding the context window over extended sessions. This trade-off between injecting sufficient domain knowledge and preserving context capacity is a recurring design tension throughout this thesis.

To support Multi-Turn Conversations, an application is needed to manage the conversation state across multiple turns. Such an application is called AI assistant. The assistant is re-sending the full conversation history as context for each new user input. Users pay for each processed token, which is the basic unit of text an LLM processes. Examples include Claude from Anthropic [22], ChatGPT from OpenAI [20] and Gemini from Google [36]. When an AI assistant is further extended with autonomous tool use and execution capabilities, it becomes an AI agent.

## 2.6. AI Coding Agents

AI assistants respond to prompts but cannot act on their own. For performance optimization of a Truffle language, responding is not enough. The system must run profiling tools, analyze results, and modify code in the filesystem. However,

the LLM on its own cannot execute the commands. But Schick et al. developed a mechanism to allow the LLM to inform the environment to invoke a tool.

For brevity, this thesis calls *AI coding agent*, *coding agents* or simply *agents* the combination of LLM and the executing environment. It uses *LLM* when referring specifically to the underlying language model.

Schick et al. introduced with Toolformer [75] a model that decides autonomously which tools to call, when to call them and with which arguments. Therefore, the LLM context must include which tools are available in the environment. The approach leverages a special character sequence to indicate a tool call, for instance the following bash command.

```
You can find in the current directory [Bash('ls', '-a') -> 'src/'].
```

The LLM generates the special character sequence, whenever it decides to call a tool, because they included it in the fine-tuning training data. Whenever the LLM generates a character sequence until the "`->`" arrow, the environment stops the generation process, executes the tool call, appends the response and lets the LLM continue generating tokens [75]. The tool call isn't restricted to bash commands, but also includes API calls. By now, the tool calling transitioned from Toolformer's special character sequence to JSON-formatted tool calls [12].

Common examples for AI coding agents are the open-source solution opencode [3] and the closed-source solutions Codex by OpenAI [60] and Claude Code by Anthropic [23]. Claude Code uses one of the Claude models under the hood, which are closed-source models. This thesis uses Claude Code with the Claude Sonnet 4.5 model as representative coding agents, due to its 82% accuracy on SWE-bench Verified ($N = 100$) [11]. SWE-bench is a benchmark for evaluating the LLM's capacity to solve real-world software issues [41].

## 2.7. Domain-Specific AI Enhancement Approaches

While coding agents can execute tools, their effectiveness on specialized domains like Truffle optimization depends on their domain-specific knowledge from the training data. If the model lacks such expertise, knowledge enhancement techniques become relevant. Song et al. identified four approaches [79]:

Prompt Optimization The user prompt contains required information about the domain.

Dynamic Knowledge Injection Before passing the prompt to the LLM, a retrieval system fetches information relevant to the prompt. The additional information and the prompt are passed together to the LLM.

Static Knowledge Embedding The knowledge is integrated into the parameters of the LLM model, for instance by including domain-specific knowledge in the training data of the LLM.

Modular Knowledge Adapters The LLM loads an external knowledge base when needed during the message processing by the LLM, for instance with a tool call.

### 2.7.1. Prompt Optimization

Prompt Optimization enhancement mechanisms include adding domain-specific information to the prompt, for instance with In-Context Learning. This does not require tool calling, adjustments of the LLM or an external knowledge base. Instead, the user adds the required knowledge to the prompt, like compiler architecture examples for the compiler agent. This is a lightweight and effective technique [79].

A common way to operationalize this in practice is an `AGENTS.md` file to make injected knowledge reusable. `AGENTS.md` is a Markdown file in the project root, which gets injected into the context automatically on every new session. Developers use it to explain the project structure, build and test commands, code style guidelines and more to the coding agent. In the case of Claude Code, the file is called `CLAUDE.md` instead of `AGENTS.md` [10, 46].

However, with increasing size of the injected information, several issues arise. The injected knowledge counts into the context of the LLM. The section 2.5.1 discusses the limitations of LLMs in relation to the context. Additionally, the injected data is static and not adapted to the prompt.

### 2.7.2. Dynamic Knowledge Injection

Enhancement mechanisms of the Dynamic Knowledge Injection category work by retrieving information from an external knowledge base and passing it together with the prompt as input to the LLM. A widely adopted pipeline for this is Retrieval-Augmented Generation (RAG) [45]. Within a RAG pipeline, different retrieval techniques can be used, such as sparse keyword-based retrieval like BM25 [74] or dense semantic retrieval using embedding models [33].

A dense RAG consists of a knowledge base, an embedding model and a vector database. At build time, developers split the knowledge base into small document snippets and convert them into vector representations via the embedding model. When prompting, the system retrieves documents based on the prompt first. Afterwards, the LLM generates a response grounded in the retrieved documents rather than solely on knowledge learned during training.

Compared to other enhancement categories, Dynamic Knowledge Injection has the benefit of working with a potentially large knowledge base. Instead of injecting the full knowledge base, it only injects the relevant information. This keeps the prompt concise and within the LLM's context window limits. However, the output quality depends heavily on how accurately the retrieval step identifies relevant documents [33]. No fine tuning or retraining of the model is required, but the knowledge base and retrieval mechanism must be developed and maintained [45]. Additionally, asking unrelated questions to the specialized knowledge base may result in the injection of irrelevant information, which reduces the quality of the LLM output [79] and increases API costs.

### 2.7.3. Static Knowledge Embedding

Static Knowledge Embedding integrates domain knowledge into the model parameters via fine-tuning. While this avoids runtime dependencies, it requires retraining, which is expensive and resource-intensive [80]. More critically, it is inapplicable to closed-source models such as Claude Sonnet 4.5, where model weights are not accessible [79].

### 2.7.4. Modular Knowledge Adapters

Enhancement mechanisms of the Modular Knowledge Adapters category work by crafting a service that attaches to the model. The LLM decides autonomously when to use these *knowledge adapters*. Users of coding agents need to attach the available adapters to the agent [79].

With an increasing number of adapters, the context gets polluted, which negatively impacts the output quality. Because the adapters do not touch the model itself, they are reusable across different models [84].

This section introduces multiple examples of knowledge adapters, because this thesis will use different kinds of them in the approach. Commonly referenced examples are Model-Context Protocol (MCP), agent skills, and subagents.

**Model-Context Protocol (MCP)**
MCP is an open standard, initially developed by Anthropic [13]. MCP is a server-client pattern, where the agent is the client. The server consists of tools, resources and prompts. Users of coding agents can connect multiple MCP servers with an agent on startup. The server runs either locally or remotely.

The coding agent can call MCP tools, like all other local tools. They are added into the list of available tools on startup, including a description and the specification about the input and output schema.

However, the LLM decides autonomously when to call an MCP tool, which is based on the current context, the available information about the use cases of the tool from the description and the LLM reasoning. The MCP tool triggers the execution of a programmable function on the server, for instance executing complex calculations and returning it to the LLM. The figure 2.2 visualizes the example.

Resources represent data or files that an MCP client can read. The user of the coding agent can either inject resources into their prompts manually, or the agent can autonomously decide to load them. Additionally, the user can use predefined prompts provided by the server.

While MCP provides a great way to connect the agent with external systems, with an increasing number of MCP tools, the LLM has difficulty identifying the relevant tools [32]. Additionally, the tool descriptions risk context pollution, as described in section 2.5.1. Moreover, LLMs have difficulty generating structured data, which is required when providing the input data to an MCP tool [81]. Therefore, a suitable balance between number of tools and universality of the interface should be achieved when developing an MCP server.

**Agent Skills**

"Agent skills" are a recent knowledge adapter mechanism. They were initially developed by Anthropic, who published them as an open format [4]. The idea is to inject domain-specific knowledge into the context when the LLM needs it. An agent skill definition contains at least a `SKILL.md` Markdown file with reusable domain knowledge. It can also reference additional resources with Markdown links, for instance, documentation or Python scripts.

A *Skill Catalog* contains a list of all available agent skills with their names, descriptions and the absolute `SKILL.md` location. This catalog gets injected into the beginning of the context.

When generating a response, the coding agent can invoke an agent skill based on the current context, the skill description and the reasoning of the LLM. It calls a dedicated tool with the wanted skill name. When invoking a skill, the `SKILL.md` content is inserted into the context. The agent can load referenced resources as required. The figure 2.3 visualizes the process with an example.

However, depending on the length and content, the `SKILL.md` can risk context pollution, as described in section 2.5.1. Therefore, a suitable balance between size and detail should be achieved when developing an agent skill.

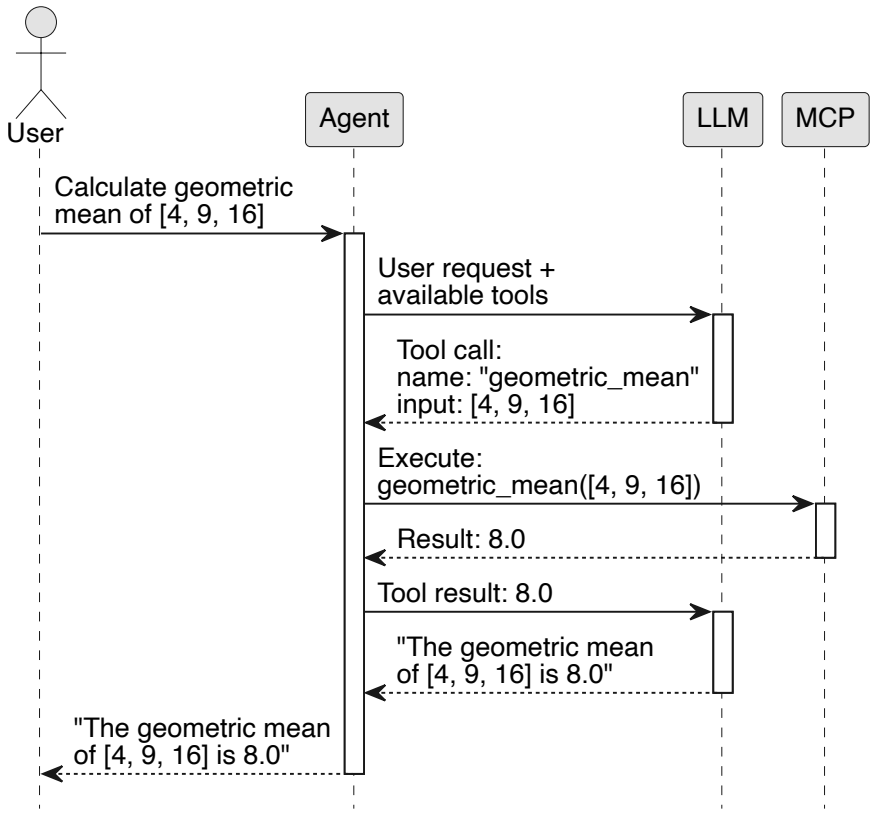



Figure 2.2.: Example for tool calling with MCP. The coding agent calls the MCP tool `mcp_geometric_mean` based on the LLM's decision. MCP client calls the function and passes the result to the LLM.

**Sub-Agent**

A custom "sub-agent" is a widely used knowledge adapter mechanism, adopted by most AI coding agents [9, 61]. The main agent delegates tasks to another specialized agent. Just as a human sends a prompt to the main agent, the main agent invokes the sub-agent with a prompt.

In Claude Code a sub-agent is defined by a Markdown file including name, description, persona, and model to use. Predefined sub-agents in Claude Code are `Explore` and `Plan`, which can be used to analyze a codebase or create an implementation plan [9].

The main agent decides autonomously when to invoke a specific sub-agent. This decision is based on the current context, the description of the agent, and the reasoning of the LLM. The user can give the LLM a hint by referencing the sub-agent in the prompt, but it is not guaranteed.

Invoking a sub-agent is implemented as another tool call, which takes the wanted sub-agent name and the prompt as an input. The environment starts

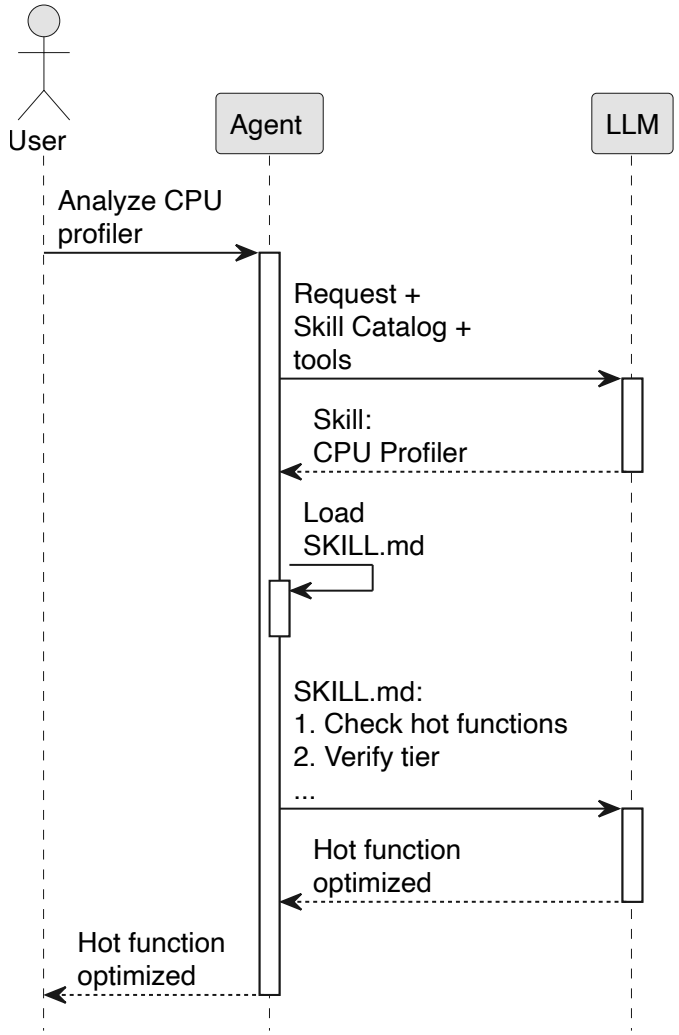



Figure 2.3.: Example for agent skill invocation. The LLM requests the environment to invoke the agent skill "CPU Profiler Analysis". The AI reads the `SKILL.md` and passes it to the LLM.

the sub-agent session with the persona, and the input prompt. The sub-session proceeds without further interaction with the user or the main agent. The sub-agent's response is inserted into the context of the main agent as tool result. The figure 2.4 visualizes the process with an example.

Sub-agents are particularly useful for delegating complex tasks, because the main agent only receives the final result and the intermediate steps are not in its context. However, the sub-agent needs to load the same information that the main agent has already loaded. This results in more tokens being used and higher overall costs, which is a well-documented overhead in multi-agent LLM systems [90].

The described architectures for MCP tools, agent skills, and sub-agents were empirically verified for Claude Code 2.1.31 by intercepting API requests with mitmproxy [53].

## 2.8. Claude Code Session

The following section introduces the structure of a Claude Code session, which is relevant for extracting usage information for the evaluation in chapter 6.

Claude Code is structured as a client on the developer's machine that communicates with Anthropic's server via the Messages API. A conversation session with the agent consists of a series of messages. When the user writes to the agent, the client sends the full conversation history as JSON object including the new message. Each request also contains available local tools with name, description, and input schema. The server responds with an event stream as the LLM produces the answer. This behavior follows the stateless design of the Messages API [12].

Each message in the conversation carries a `role` field and a `content` array. The `role` is either `user` or `assistant` depending on who is contributing the message. The `content` array contains objects with a `type` field indicating the structure and semantics of the array item. The Messages API defines multiple content types. The four most relevant content types for this thesis are shown in table C.1 [12].

When the LLM decides to call a tool, the server response contains a content item with the type `tool_use`. When the client receives a tool use response, it verifies the required permissions first and executes the tool call afterwards. The client sends a new request with the tool result and all previous messages to the server. A tool invocation can be, for instance, a bash command, the invocation of an agent skill, or a sub-agent. The figure 2.5 visualizes the process.

The server response also includes meta information, for instance the `usage`, described in detail in table C.2. Especially important for Claude performance and related costs are the cache fields. During a conversation, the conversation

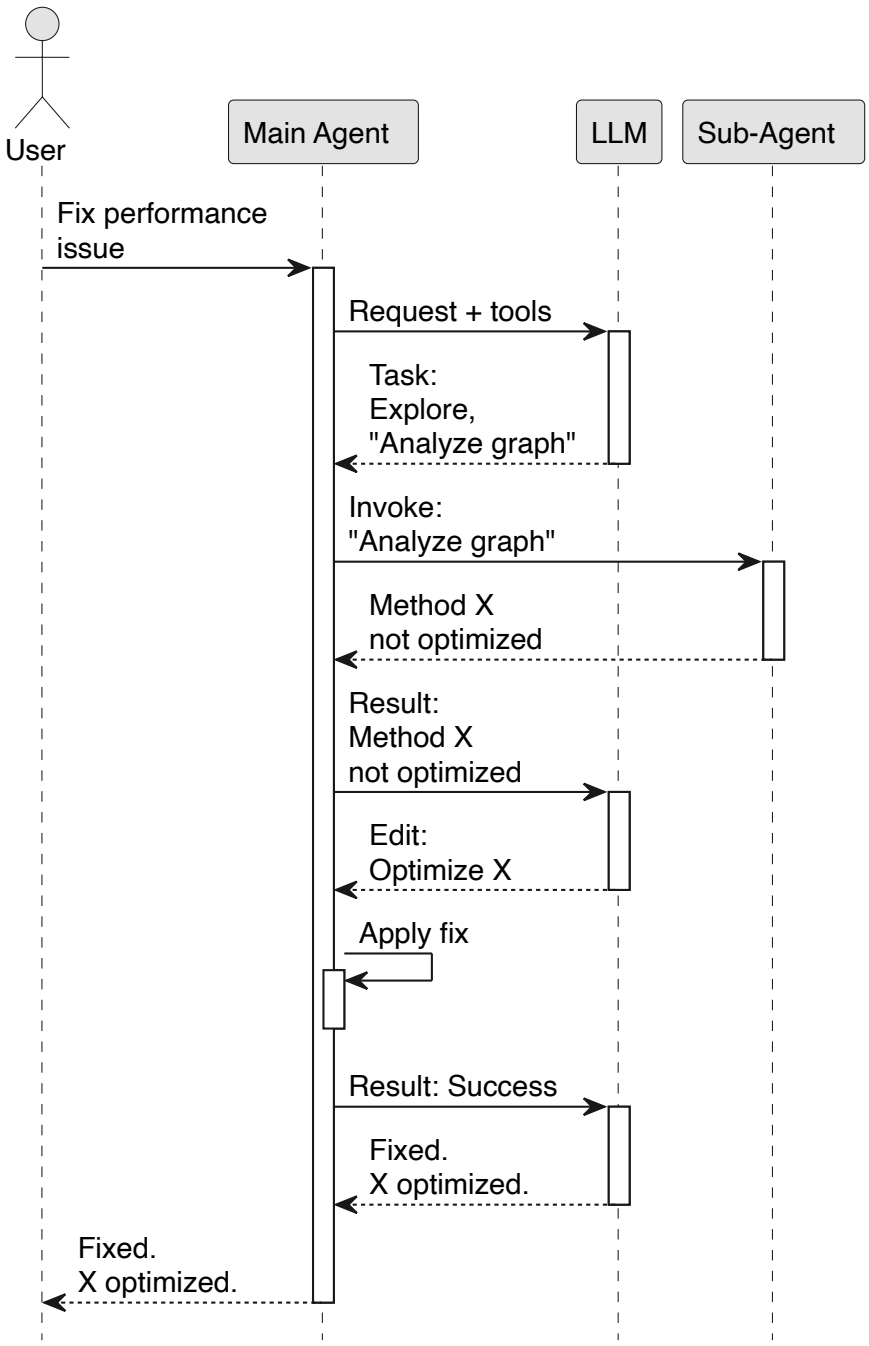



Figure 2.4.: Example for sub-agent invocation. The main agent invokes the sub-agent "Explore" to explore the codebase for performance issues.

history is passed to the LLM repeatedly. To avoid redundant computation, prompt caching allows the reusing already processed part of the conversation history, which costs 90% less [12, 14, 34]. The token counts are not only in the response, but also in the locally persisted session files `.jsonl`.

## 2.9. Cognitive Dimensions of Notations

To evaluate the effect of the approach on the developer, it is important to understand what kind of effects a tool can have on the developer. CDN provides a vocabulary for evaluating the usability of information artifacts, for instance programming languages [18]. It extends naturally to Command-Line Interface (CLI) tools and AI coding agents since both serve as information artifacts through which developers read and manipulate program state. It offers a set of named dimensions, each describing a distinct property of how a notation or tool interacts with human cognition.

In this thesis, the dimensions serve as a deductive coding framework for the thematic analysis of the user study in chapter 6. The individual dimensions are presented in table D.1.

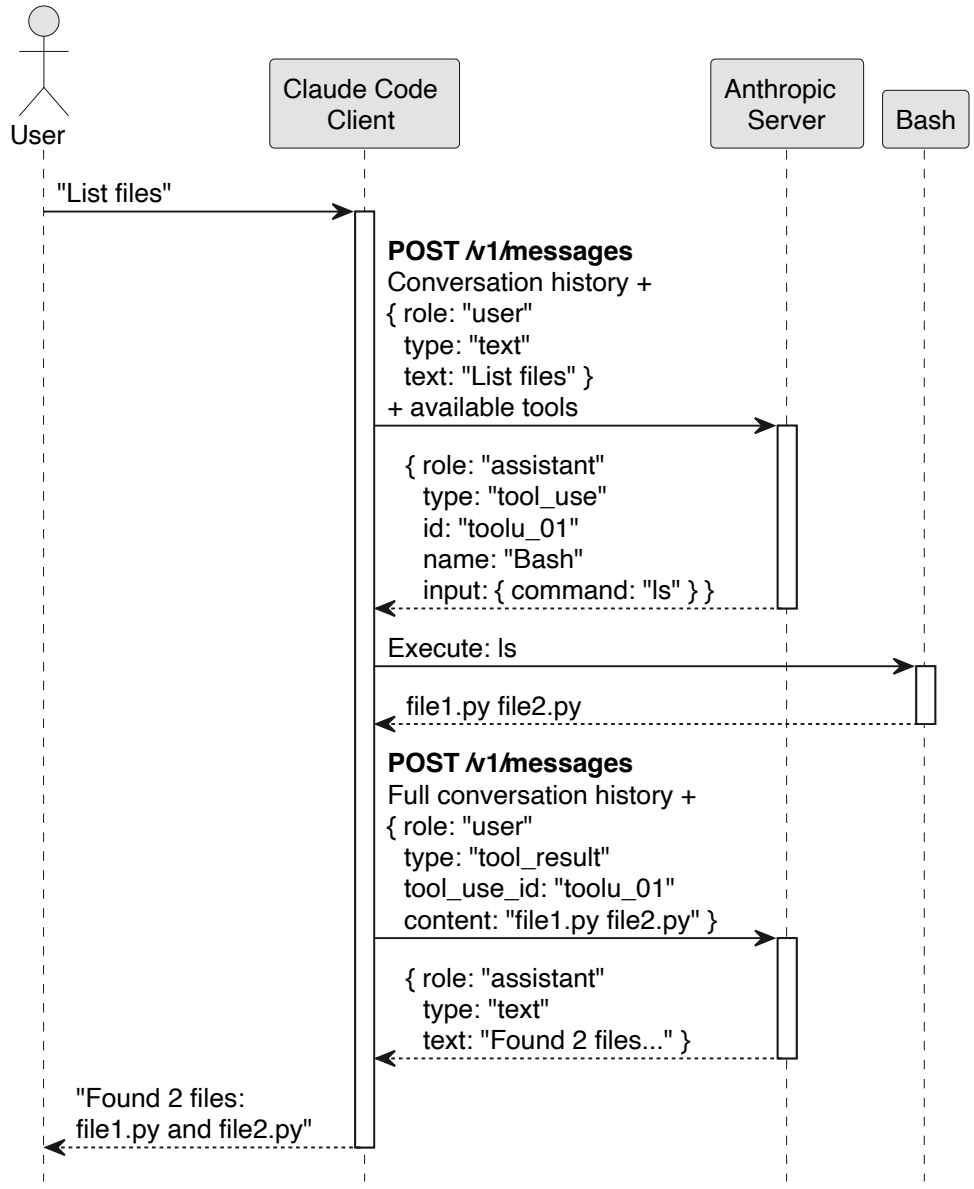



Figure 2.5.: Example for a response and request of the Anthropic Messages API. The LLM generates a tool call for the `Bash` tool. The client executes the command and passes the result to the LLM.

# 3. Understanding Expert Optimization Practice

The goal of this chapter is to understand how experts approach the task of optimizing a given language implementation. Therefore, we conducted a contextual inquiry interview with the Graal expert, Dr. Tim Felgentreff. From that interview, we outline a process for optimizing a language implementation running on Graal and Truffle to automate it in the next step. Additionally, this chapter describes a subset of available Graal profiling tools and collects the functional requirements for the final approach.

## 3.1. Contextual Inquiry Interview

We conducted a contextual inquiry interview [17] with Dr. Tim Felgentreff. The interview script is in Appendix E. He represents the collective knowledge of the Graal group. The interview method is suitable because it captures not only the expert's stated reasoning but also implicit decisions observed during actual tool use, which a standard interview would miss. The interview was conducted in German. Quotations are translated by us, and the original German text is provided in footnotes, including English translations.

The interview is split into three parts: introduction, code review and profiling tools as shown in figure 3.1. The discussion about the last two phases is described in section 3.2 and section 3.3.

In the first introduction phase, we asked him about his experience with Graal and Truffle. He has more than eight years of experience with Truffle languages[1] [28, 00:02:45]. In this time, he worked on building Truffle implementations for Squeak (TruffleSqueak), Python (GraalPy), and Lox (swalox)[2] [28, 00:03:28]. Notably, he is an expert on the language definition rather than on

[1] Original: *"Ich mache Truffle beruflich seit acht Jahren und zwei Monaten."*, Translated: "I have been working with Truffle professionally for eight years and two months."

[2] Original: *"Also angefangen hat es mit Truffle Squeak. Dann wurde ich angestellt. Dann habe ich GraalPy gestartet. Und irgendwann habe ich auch Lox gemacht, bevor die Vorlesung war."*, Translated: "So, it started with TruffleSqueak. Then I was hired. Then I started GraalPy. And at some point, I also worked on Lox before the lecture took place."

the compiler architecture side of Graal[3] [28, 00:03:42]. This focus matters, because this thesis also focuses on the language rather than the compiler part of Graal. To summarize, he brings a lot of prior domain knowledge with his experience in building Truffle languages and his work on the Lox language.

After the introduction, Felgentreff was asked to review a language implementation code base. He should feel as though he has been familiar with it all along. The codebase included the AWFY benchmarks alongside the language implementation itself. Afterwards, his task was to use Graal profiling tools to diagnose a performance issue. Felgentreff was asked to explain why he selected which tool and how he interpreted the tool output. In a short closing discussion, we asked him questions that came up while he was working on the tasks.

| **Interview (1h 20m)** | | |
|---|---|---|
| Intro (10m) | Code Review (20m) | Profiling Tools (50m) |

Figure 3.1.: Structure of the contextual inquiry interview with Felgentreff

## 3.2. Expert Optimization Workflow

Derived from the interview, this section discusses the expert's optimization workflow. It is reconstructed from this single session and should be understood as one expert's approach rather than a universal methodology. Felgentreff's extensive experience justifies treating a single session as a credible basis.

Felgentreff started with laying a baseline to understand where to start. Afterwards, he examined the codebase without any tools and collected theories for performance issues. In the following phase, he used Graal's profiling and analysis tools to verify a theory. After a successful verification, he planned the implementation, implemented a fix, validated it with benchmarks and tools and would iterate further. The figure 3.2 shows the workflow as a state diagram.

### 3.2.1. Establish Baseline

Felgentreff looked for existing benchmarks first and executed them to lay a baseline as starting point before any changes. In case of the AWFY benchmarks, each targets a specific concern of a language implementation. But without any

[3]Original: *"Aber eigentlich immer auf Language-Seite, nicht so viel auf Compiler-Seite bei Truffle."*, Translated: "But actually, it's almost always on the language side, not so much on the compiler side with Truffle."

cross-language comparison numbers, the benchmarks do not help to identify performance issues[4] [28, 01:16:01].

During the interview, we observed that Felgentreff consistently ran benchmarks with a minimal iteration count first to verify correctness before collecting performance data.

### 3.2.2. Static Theory Generation

In the second phase, Felgentreff examined the codebase by navigating top-down and looking for known Truffle anti-patterns. Examples of these anti-patterns are basic Java flaws, missing flags or missing or an unnecessary `@TruffleBoundary`[5] [28, 01:15:25]. Truffle Boundaries instruct Graal to not optimize a function and transfer to the interpreter [62]. Language developers use them for functions that are too complex for Partial Evaluation.

Starting with a first orientation, the expert looked for benchmarks, configuration files, syntax definition, test files, important Truffle files and he analyzed the folder layout. Afterwards, he continued with a code inspection starting with the Truffle Language file and the Bytecode Root Node file. These provide high-level definitions of the language interpreter. He continued with the implementation of data structures, for instance of classes and arrays, and custom AST nodes, followed by the neighboring code[6] [28, 01:15:00].

Throughout this inspection, Felgentreff collected theories for performance issues in a notes app. He created a map with locations and what is interesting regarding performance[7] [28, 01:15:13].

[4] Original: *"Ich habe halt dann in dem Moment nicht im Kopf, was die guten Zahlen sind. [...] Ich müsste ja jetzt immer auf einer anderen Sprache das auch nochmal ausführen."*, Translated: "At that moment, I simply don't have the right figures in mind. [...] After all, I would have to explain it all over again in another language."

[5] Original: *"Beim Bytecode scrolle ich einfach mal so ein bisschen durch, ob ich eine Truffle Boundary sehe oder ob ich sehe ich geunboxed Primitives oder nicht."*, Translated: "I'll just scroll through the bytecode a bit to see if I spot a Truffle boundary or whether I see unboxed primitives or not."

[6] Original: *"Wenn ich mit Navigieren anfange, suche ich halt zuallererst nach der Truffle Language. Und dann suche ich nach etwas, was irgendwas mit Bytecode heißt. Und dann suche ich nach dem Kontext."*, Translated: "When I start navigating, the very first thing I look for is the Truffle language. Then I look for something with "bytecode" in the name. And then I look for the context."

[7] Original: *"Und dann versuche ich darüber mir eine Map zu erzeugen von wie sieht es denn aus, welche interessanten sozusagen Sachen sind denn da"*, Translated: "And then I try to build a mental map of what it looks like—what interesting things are there, so to speak."

At the end of this phase, he examined the language implementation and compiled a list of theories regarding performance issues. He prioritized them based on their criticality, according to his experience, and then proceeded to the next phase.

### 3.2.3. Tool-Assisted Verification, Fix, and Iteration

In the final phase, Felgentreff used Graal profiling and analysis tools to verify the most critical theory. The selection of tools is discussed in detail in section 3.3. As with benchmarks earlier, Felgentreff first ran the tools with a minimal example to verify correctness before continuing with the theory.

After successful validation, he planned a fix and implemented it to improve the performance and validated again. He re-profiled after each fix with benchmarks and tools to confirm the improvement. Afterwards, he continued with the next critical issue.

The order of processing the theories is critical. A critical performance issue can overlay other issues. Fixing the most critical first reduces the noise to see the other issues more clearly. For instance, according to Felgentreff, a missing Truffle Boundary can significantly increase the size of the IR Graph in the IGV[8] [28, 00:51:46]. With this noise, it is difficult to validate other issues[9] [28, 01:05:31].

At the end of the interview, if Felgentreff had had more time, he would have iterated, verified, and fixed more issues.

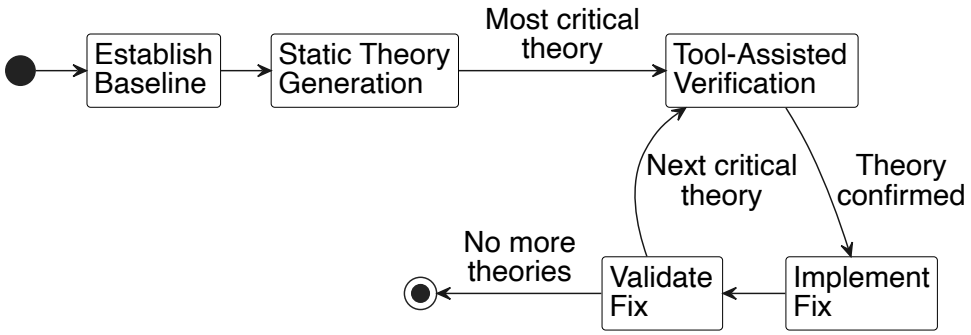



Figure 3.2.: State diagram for the expert's optimization workflow.

[8] Original: *"Ich vermute ganz stark, dass wenn man zum Beispiel allein schon mal das mit dem Lox Number macht, dann fällt hier viel weg."*, Translated: "I strongly suspect that if you were to just do the thing with the Lox Number, for example, a lot of this would fall away."

[9] Original: *"Warte mal, das mache ich jetzt mal ganz schnell, damit ich einfach was Besseres sehen kann."*, Translated: "Hang on, I'll just do this really quickly so I can see something better."

## 3.3. GraalVM Profiling Tools

This section describes how Felgentreff interacted with Graal analysis and profiling tools. It is extended with information from the Graal documentation and other literature. In the interview, the expert followed an escalation logic to select profiling tools.

Given a theory about an optimization problem, he always used the *CPU Sampler* on a benchmark that targets the problematic area [37]. With the tool, developers can examine where time is spent and how much time is allocated to each compilation tier. If a function does not run most of the time in high-tier, the expert used *Trace Compilation* to identify deoptimization loops[64]. If the functions run in high-tier most of the time but showed repeated invalidations in the *Trace Compilation* output, the expert used the *Trace Transfer To Interpreter Tool* to identify the reason for the deoptimization loops [64]. If the functions run stably in high-tier, the expert inspected the IR graph with IGV [38].

These four tools are introduced in detail below, while four additional tools are summarized in a table at the end of this section.

### 3.3.1. CPU Sampler

The CPU Sampler is a time-based sampling profiler that identifies where a program spends execution time. The expert recommended using the CPU Sampler first, because it is compact and gives a time distribution across functions. Additionally, it shows a compilation tier breakdown of the execution time. This helps to understand where time is spent and whether compilation works[10] [28, 00:55:50].

The table 3.1 shows the relevant flags to activate and control the CPU Sampler. The output consists of a table that includes the method name, total time spent in the function (including other function calls), time spent only in the function itself, and the percentage of time spent in each tier [37].

If the tier distribution shows a small time spent in high-tier and more time in interpreter mode and low-tier, it means that compilation issues exist. If the tier distribution shows most time spent in high-tier but the function is still slow compared to other methods, the IR graph should be inspected with IGV [64, 88]. To summarize, the CPU Sampler is a gateway tool, whose output determines the next step.

[10] Original: *"Das Coole an dem CPU-Sampler ist, es ist ziemlich kompakt und es gibt mir zwei Sachen. Es sagt mir, wo habe ich Zeit verbracht und... Wurde das, worin ich Zeit verbracht habe, compiled?"*, Translated: "The cool thing about the CPU sampler is that it's quite compact and gives me two pieces of information: it tells me where I spent time and... whether the code where I spent that time was compiled."

Table 3.1.: Most important CPU Sampler flags

| Flag | Description |
|---|---|
| `--cpusampler` | Enable the CPU Sampler. |
| `--cpusampler.ShowTiers` | Show per-tier compilation info (interpreter (tier 0), Low-tier (tier 1), High-tier (tier 2)). |

### 3.3.2. Trace Compilation

The Trace Compilation Tool is an event handler. It logs every compilation event with timing, tier levels, success or failure status, and invalidation reasons [64]. It provides visibility into what the compiler is doing. Developers can use it to verify hot code is compiling, diagnose compilation failures, and track deoptimization cycles. The tool is activated with the `--engine.TraceCompilation` flag.

Felgentreff used it when the CPU Sampler signaled that a method did not run fully in high-tier, which indicated a potential deoptimization cycle[11] [28, 00:55:50]. Two events raise concern when they appear in the event stream. When a compilation fails with the event `opt failed` the code may be too complex or contain unsupported patterns. The developer should check the included reason.

Additionally, the tool can indicate when a method has been repeatedly deoptimized with repeated `opt deopt` events. If a compilation event `opt done` follows, as shown in listing 3.1, it indicates a deoptimization cycle. The reason for the deoptimization can be observed with the Trace Transfer To Interpreter Tool.

Listing 3.1: Trace Compilation output (simplified). Normal compilation progresses from low-tier (tier 1) to high-tier (tier 2). Failed optimization shows `opt failed`. A deoptimization cycle shows repeated `opt deopt` and `opt done` events on the same method.

```
# Normal compilation progression
[engine] opt done sieve <tier1> |Time: 45ms
[engine] opt done sieve <tier2> |Time: 234ms

# Failed compilation
[engine] opt failed root sieve |Reason: Bailout
```

[11] Original: *"Wenn es nicht compiled wurde, habe ich schon mal das erste Problem. Dann muss ich Engine Trace Compilation benutzen [...] Da ist dann vielleicht ein Deopt-Loop oder sowas."*, Translated: "If it hasn't been compiled, I run into the first problem right there. Then I have to use engine trace compilation [...] There might be a deopt loop or something like that."

```
# Deoptimization cycle
[engine] opt deopt sieve |Reason: Assumption invalidated
[engine] opt done sieve <tier2> |Time: 234ms
[engine] opt deopt sieve |Reason: Assumption invalidated
```

### 3.3.3. Trace Transfer to Interpreter

While Trace Compilation reveals that a deoptimization occurred, Trace Transfer To Interpreter reveals why it got transferred to the interpreter. Developers can use it to diagnose the cause of the deoptimization [64]. Felgentreff used this tool for targeted theory testing rather than broad exploration.

The tool is activated with the `--engine.TraceTransferToInterpreter` flag. The output contains for each deoptimization the reason, location and a stack trace of the specific call.

With the reason, developers can pinpoint the deoptimization trigger, for instance Felgentreff received the reason "Assumption of root `moved` is invalidated." He used this tool after Trace Compilation revealed repeated deoptimization cycles, to identify the specific assumption that was violated[12] [28, 01:08:05].

### 3.3.4. Compiler Graph Dumping and IGV

In comparison to the Trace Transfer To Interpreter tool, analyzing the compiler graph is not about compilation events, but the compilation artifact. It allows an inspection of the compiler's IR graph. Felgentreff used it when the compilation happened successfully but the compiled code was still slow.

The tool is activated with the `--vm.Djdk.graal.Dump` flag. It expects as argument a specification of which intermediate steps of the compilation process should be dumped. The results are `.bgv` files, which can be analyzed visually with IGV [38].

IGV allows to navigate through a graph of IR nodes. Felgentreff mapped the IR nodes back to source code sequentially by comparing them side-by-side. A disproportionate graph size for a small function reveals a performance issue, for example a missing `@TruffleBoundary`[13] [28, 00:51:27]. Felgentreff judged graph size relative to source complexity based on experience. He did not apply a fixed threshold but relied on recognizing disproportionate patterns.

[12] Original: *"Es gibt ein Tool, was mir das printet. [...] will diese Assumption in Validation haben. Das wollte ich haben. Assumption of root moved is invalidated."*, Translated: "There's a tool that prints this for me. [...] I want this assumption in the validation. That's what I wanted. Assumption of root moved is invalidated."

[13] Original: *"Das ist ja eine ganz kleine Funktion. [...] Und diese Graphen sind riesig dafür."*, Translated: "That is a really small function. [...] And these graphs are huge for it."

Other indicators for optimization potential are `BoxNode` and `UnboxNode` nodes, which indicate boxing of values. An `InvokeNode` implies an unspecialized method call outside of the Graal optimizations, for instance in figure 7.2. When escape analysis fails, the compiler graph contains `CommitAllocationNodes`.

### 3.3.5. Additional Profiling Tools

Graal provides more profiling and analysis tools than observed in the interview, they are relevant to the approach in chapter 4. These were added from the public documentation afterwards. The table 3.3 contains additional tools and a brief description, extracted from the Graal documentation [64, 65].

Table 3.3.: GraalVM Truffle profiling and tracing tools

| **Name** | **Description** |
|---|---|
| Trace Performance Warnings | Detects optimization barriers during Truffle compilation, for instance virtual calls, non-constant stores, unresolved type checks, and Truffle Boundary issues that prevent peak performance.<br>Flag: `--compiler.TracePerformanceWarnings=all` |
| Memory Tracer | Experimental allocation profiler tracking memory allocations at guest-language level. Identifies unnecessary object creation and allocation-heavy hot loops.<br>Flag: `--memtracer` |
| CPU Tracer | Counts exact execution frequencies at function and statement level, not the execution time.<br>Flag: `--cputracer --cputracer.TraceTiers=true` |
| Trace Inlining | Shows inlining decisions during compilation with call tree structure and reasons for inline/don't-inline decisions.<br>Flag: `--engine.TraceInlining` |

## 3.4. Functional Requirements

This chapter reconstructed an expert optimization workflow from a single contextual inquiry session and described the profiling tools involved. From the interview and additional research, three functional requirements for an enhanced coding agent emerge:

R1 – Workflow. The agent must replicate the expert's phased workflow: establish a performance baseline, generate theories from static code analysis, verify theories with profiling tools, create an implementation plan, implement, and iterate (section 3.2).

R2 – Tooling. The agent must invoke and interpret GraalVM profiling tools, including executing them, parsing their output, and reasoning about the results (section 3.3). Tool selection must follow a context-dependent escalation pattern: the agent selects the next tool based on prior results rather than following a fixed sequential order.

R3 – Comparison. The agent requires reference performance data from other language implementations to contextualize benchmark results. Without cross-language comparison, raw benchmark numbers alone do not reveal whether a result indicates a performance issue.

Some aspects of the expert workflow resist automation, for instance the experience-based prioritization of theories and the implicit pattern library built over years of Truffle development.

# 4. Designing the Optimization Assistant

This chapter derives the design requirements from the expert workflow and presents the layered architecture that realizes them.

## 4.1. Usability Requirements

The functional requirements 1–3, derived from the expert workflow in section 3.4, describe what the system must do. This section adds two usability requirements that emerged from iterative prototyping:

R4 – Integration. The enhancements should integrate into coding assistants that developers already use in their daily work, rather than requiring a separate tool.

R5 – Autonomy. The enhancements should support varying levels of autonomy. It should support a range from assisting with individual tool execution and interpretation to conducting the full optimization workflow autonomously.

## 4.2. Approach Overview

The approach contains three levels: "Tool Assistance", "Process Assistance" and "Full Orchestration". All levels use modular knowledge adapters from section 2.7.4.

The Tool Assistance level contains mostly agent skills and focuses on the *Tooling* requirement. Each agent skill contains the instructions to run and interpret one mentioned profiling or analysis tool from section 3.3. The exception is compiler graph dumps, which are implemented as a sub-agent, due to the large context required for graph analysis.

The Process Assistance level consists of process helper sub-agents and focuses on the *Workflow* requirement. Each sub-agent supports the developer to execute one step of the optimization process introduced in section 3.2. They use the agent skills from the Tool Assistance level.

The Full Orchestration level is another agent skill and focuses on the *Workflow* requirement. It contains all instructions to execute the full optimization process from laying the baseline to verifying an implementation. While the Process Assistance level handles individual steps, the Orchestration level runs the entire workflow autonomously. Therefore, it uses the sub-agents from the Process Assistance level.

To fulfill the *Comparison* requirement, the coding agent is extended with two tools to compare benchmark runs. The figure 4.1 gives a visual overview of the approach.

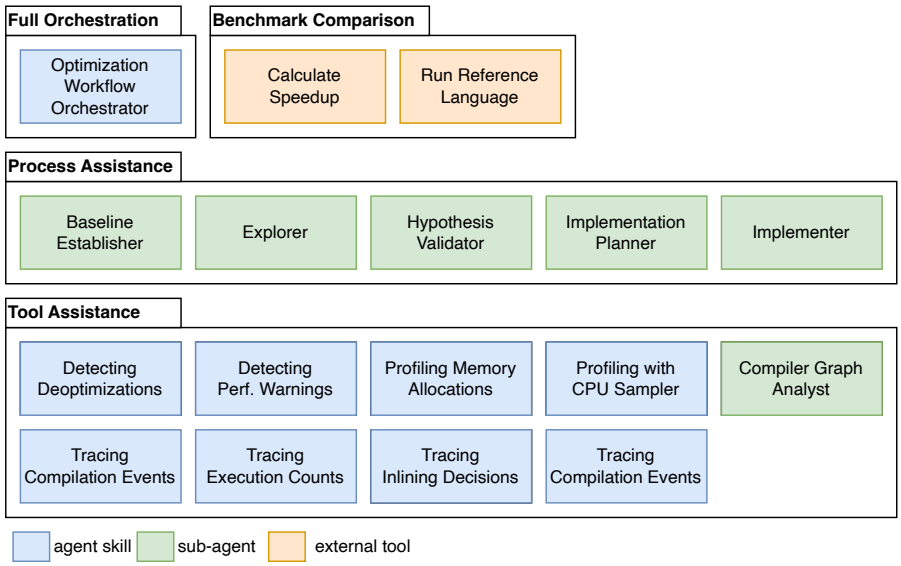



Figure 4.1.: Overview of the layered agent architecture for performance optimization assistance. Shows the levels Full Orchestration, Process Assistance and Tool Assistance. Benchmark Comparison shows the tool extensions shared across multiple levels.

The three knowledge adapter types used — agent skills, sub-agents, and external tool integration — are supported by the six most widely adopted coding agents as shown in table 4.1 [87]. The approach is therefore portable beyond Claude Code without architectural changes, supporting the *Integration* requirement by design.

One major constraint of the approach design is token usage. To keep the agent focused it is important to keep the context small and not pollute it as discussed in section 2.5.1. This is relevant when discussing our decision on which enhancement technique to use for which level.

Table 4.1.: Support for knowledge adapter mechanisms across the most widely adopted coding agents [87] (as of June 2026). *On-Demand Instructions* inject domain knowledge into the context when needed with agent skills. *Task Delegation* spawns isolated sub-sessions for specific tasks with sub-agents. *External Tools* integrate programmatic tool calls.

| **Coding Agent** | **On-Demand Instructions** (agent skills) | **Task Delegation** (sub-agents) | **External Tools** (i.e. MCP) |
|---|---|---|---|
| GitHub Copilot [51] | ✓ | ✓ | ✓ |
| Claude Code [6] | ✓ | ✓ | ✓ |
| Codex [60] | ✓ | ✓ | ✓ |
| Junie [42] | ✓ | ✓ | ✓ |
| Antigravity [35] | ✓ | ✓ | ✓ |
| Cursor [25] | ✓ | ✓ | ✓ |

## 4.3. Benchmark Comparison Tools

Two tools enhance the functionality of the coding agent with tools, which are used at multiple levels. The first tool *Run Reference Language* generates the execution time of predefined AWFY benchmarks for a reference language implementation, for instance CPython, to fulfill the *Comparison* requirement. The benchmarks should run on the developer machine, because the execution times are hardware dependent.

The second tool *Calculate Speedup* deals with the comparison of two sets of execution times of the same benchmarks of different implementations. It allows the cross-language comparison of benchmark execution times and fulfills the *Comparison* requirement. This is useful to evaluate if a code edit improved the overall performance. As mentioned in section 2.3, the appropriate metric is the geometric mean of the normalized times.

**Design Alternatives** For both benchmark comparison tools, several alternative approaches were evaluated and rejected. The agent could download and execute the benchmark code or calculation scripts directly, either from the official AWFY GitHub repository[1] or through an agent skill referencing a local script. These approaches were rejected due to high token costs and unnecessary information for the LLM context risking context pollution as described in section 2.5.1. The direct tool integration removes the LLM from the execution loop entirely and returns structured data, eliminating hallucination in the computation step. This also saves tokens, costs, and context.

[1] https://github.com/smarr/are-we-fast-yet

For *Run Reference Language* specifically, downloading pre-computed reference data from the repository was rejected, because comparing locally collected times against data from different hardware would produce misleading speedup ratios. For *Calculate Speedup*, letting the LLM compute the geometric mean through token prediction was rejected, because it approximates the result from learned patterns rather than applying the formula.

## 4.4. Tool Assistance Level

The Tool Assistance level contains one agent skill for each profiling and analysis tool introduced in section 3.3 to fulfill the *Tooling* requirement. The individual invocation of the agent skills fulfills the *Autonomy* requirement.

Each skill contains a description, instructions when to use it, a quick start with command line examples, command line options, output interpretation and patterns to look for, and related skills to continue the analysis further. Additionally, it includes a Fermi Verification[2] Step that checks whether the tool produced plausible output before the agent continues the analysis. In this step, the agent is asked to come up with a simple example, predict the tool output autonomously and execute the tool with the example. If the result differs too much in magnitude, the tool appears to be malfunctioning. This reflects the behavior of Felgentreff in section 3.2.

The skill provides verified tool knowledge, so the agent does not need to recall these from training data. The LLM then interprets the tool output in the context of the current codebase and hypothesis. This separates factual accuracy, encoded in the agent skill, from analytical reasoning, which remains with the LLM.

**Design Alternatives** Four alternative approaches were evaluated and rejected. Including all tool knowledge directly in the prompt risks context pollution as described in section 2.5.1. Implementing each tool as a sub-agent incurs unnecessary token costs and prevents the combination of multiple tools in a single analysis step. Implementing each tool as an external tool, for instance with MCP, requires detailed descriptions of tool input and output for each tool. The full documentation of the tools is always loaded due to protocol constraints and would pollute the context.

The fourth alternative was a RAG for tool knowledge retrieval. The tool documentation is preprocessed, split into chunks and stored in a vector database via an embedding model. A coding agent can use the RAG as, for

[2] Fermi estimates are back-of-the-envelope calculations based on logical assumptions. They quickly verify the plausibility of complex quantitative claims.

instance, an MCP tool to query it and receive chunks. This approach was prototyped and evaluated.

The benefit of this approach is that a single prompt can retrieve documentation from multiple tools, allowing combinations of tools. Additionally, it only returns the sections of the tool documentation, that are relevant to the query.

On the other hand, it is not guaranteed that all relevant sections are retrieved and invoking multiple agent skills allows the combination of multiple tools too. Moreover, the results heavily depend on how the agent formulates its queries, which varies across runs and decreases reproducibility for consistent outcomes. Additionally, the tool knowledge base is small and the infrastructure is disproportionate, but extendable for new tools.

The prototype confirmed these concerns. Agent skills provide complete, reproducible tool knowledge with lower infrastructure cost.

### Compiler Graph Analysis

Compiler graph analysis is implemented as a sub-agent instead of an agent skill. The reason is that the dumped compiler graphs are not terminal output. Instead, they are dumped as binary `.bgv` files that need transformation into a queryable structure before analysis. Then the agent searches for the anti-patterns, for instance big compiler graphs, `BoxNode`, or `InvokeNode`.

To summarize, the sub-agent expects a question about compilations from the main agent, generates, transforms, and queries the graph data to answer the prompt.

**Design Alternative** As alternative, an agent skill was evaluated and rejected. The transformation, the size of the transformed output and the inherit complexity of compilation graphs risk context pollution described in section 2.5.1. Only the answer to the question is relevant for the main agent.

## 4.5. Process Assistance Level

Each step of the process, derived in section 3.2, is realized as its own sub-agent, collectively fulfilling the *Workflow* requirement. The table 4.2 shows an overview. The exception is the verification step, where the expert ran all the benchmarks again to verify the optimization success, which remains with the main agent. Re-delegating it to a sub-agent would incur unnecessary token costs from re-reading the data, and the decision of how to proceed after validation depends on the result, making it architecturally part of the orchestration layer.

Table 4.2.: Mapping between sub-agents and each step in the expert's optimization process

| **Sub agent** | **Process step** |
|---|---|
| Baseline Establisher | Run AWFY benchmarks with the language implementation in question and a reference language implementation |
| Explorer | Read the codebase structurally and identify potential performance issues, developing hypotheses |
| Hypothesis Validator | Take the most critical hypothesis and verify or falsify it with given tools |
| Implementation Planner | Given a successfully validated hypothesis, create an implementation plan to fix the underlying performance issue |
| Implementer | Given an implementation plan, implement the plan and do a short validation if the change was successful. |

The goal of the architecture is context isolation. Each sub-agent only has the required knowledge in the context for its specific step in the process, avoiding context pollution as described in section 2.5.1. Additionally, the architecture allows the developer to invoke each sub-agent, and with it each step, individually, fulfilling the *Autonomy* requirement.

Moreover, sub-agents that require less reasoning tasks do not require large models. For instance, a sub-agent that runs benchmarks and summarizes the results can save tokens by using a cheaper model with fewer reasoning capabilities than the one validating hypotheses. This allows saving tokens and costs while leveraging sub-agents at the same time.

**Design Alternatives** Two alternative approaches were evaluated and rejected. Either the full process with each step could be passed to the main agent in one prompt or each process step is realized as its own agent skill. The prompt idea violates the *Autonomy* requirement, because if the language developer wants to execute a single step, it loads the full process. Agent skills fulfill the *Autonomy* requirement. However, when invoking multiple skill steps sequentially, the main agent's context accumulates the content of all previously loaded skills, polluting the context for subsequent steps as described in section 2.5.1.

### 4.5.1. Sharing Memory Across Sub-Agents

Before describing each sub-agent, this section introduces the memory mechanism they share. One goal of the overall approach is context isolation, each sub-agent's context should only contain the required information to fulfill the task. However, it is still necessary to share information between sub-agents, for instance to communicate the hypothesis collected by the *Explorer* with the *Hypothesis Validator*.

Two techniques are used to exchange information. The main agent forwards responses received from one sub-agent to the next. However, the main agent should focus on orchestrating and discussing results rather than serving as an information relay. This technique is still relevant to pass small information to the next sub-agent, for instance for which verified hypothesis the *Implementation Planner* should develop a solution for.

The primary mechanism is a set of shared memories. A memory is a named, persistent storage unit that any sub-agent can read from or write to during the optimization process. They are designed as human-readable that both sub-agents and the developer can inspect and edit it at any time. Since no separate memory management application is required, the mechanism integrates naturally into existing file-based development workflows, supporting the *Autonomy* requirement.

Transparency is an important design goal. The developer should be able to read, verify, and, if necessary, correct any memory content between optimization steps. This rules out opaque storage mechanisms such as databases or vector stores, which would require additional tooling to inspect.

As an example for memories, the *Baseline Establisher* summarizes its results in a *Benchmark Baseline* memory. All other sub-agents can read the benchmark execution time baseline from this memory autonomously. The approach contains four memories, each corresponding to an artifact produced by one process step and consumed by subsequent steps.

Benchmark Baseline Memory contains the benchmark execution time baseline. Used whenever sub-agent wants to compare execution times from optimized language implementation with the baseline.

Hypothesis Memory includes all currently relevant hypotheses of performance issues with plan to validate each.

Implementation Plan Memory contains the plan developed by the *Implementation Planner* to fix a verified performance issue.

Learned Lessons Memory includes what was attempted, what succeeded, and what did not. The goal is to prevent the agent from re-attempting falsified hypotheses or failed optimization strategies.

### 4.5.2. Sub-Agents

In the following, each sub-agent is introduced individually, including their required input form the main agent, purpose and response to the main agent.

**Baseline Establisher Sub-Agent**
The purpose of the *Baseline Establisher* is creating reference execution times for future optimizations. The sub-agent requires some benchmarks of the AWFY framework set up and written in the language in question. The sub-agent runs the benchmarks with predefined inner and outer iteration count, as seen in table A.1, and collects minimal, maximal, average and total execution time. The benchmark configuration does not equal the original one from the AWFY framework, because the number of iterations is higher and so the execution times longer, which makes it less interactive and useful for development purposes. The current iteration numbers are derived from prototyping.

The same benchmarks with the same configurations are executed for a reference language via the *Run Reference Language* tool too. The data of both runs is summarized in the *Benchmark Baseline Memory* and fulfills the *Comparison* requirement.

**Explorer Sub-Agent**
The *Explorer* sub-agent's task is to read the language implementation systematically and extract hypotheses for performance issues ranked by confidence. Therefore, it reads from the *Learned Lessons Memory* and *Benchmark Baseline Memory* first. Afterwards, it follows the top-down inspection process by Felgentreff in section 3.2. The agent looks for common GraalVM and Truffle anti-patterns or where the implementation is not Truffle idiomatic. It writes two to three most critical hypotheses to the *Hypothesis Memory*, based on the LLM's reasoning about potential performance impact. It includes a short description how potentially the hypothesis could be validated.

**Hypothesis Validator Sub-Agent**
The *Hypothesis Validator* sub-agent's purpose is to validate a hypothesis for a performance issue. It receives the most critical unverified hypothesis from the *Hypothesis Memory* and applies one of three validation approaches. The selection is based on the hypothesis description, the current context and LLM reasoning.

The first approach is direct experimentation. If a potential fix of a hypothesis to validate is very small, then the sub-agent can try it out and verify a performance improvement through benchmarking. This approach minimizes token usage. An example is adding or removing a Truffle Boundary decorator.

The second approach is writing a small custom benchmark. It is selected when the suspected issue can be isolated in a small program. For instance, the addition and subtraction operations should take similar time. A custom benchmark can compare the execution times of these operations, and a significant difference would confirm a hypothesis.

The third approach uses Graal profiling and analysis tools. It selects a tool from the Tool Assistance level and runs at least one tool to validate the hypothesis. The selection is based on the hypothesis description and the tool descriptions. For instance, this approach is useful to verify a deoptimization loop. Graph analysis requires a separate delegation path because nested sub-agent invocation is not universally supported across coding agents.

The sub-agent returns the result of the validation to the main agent and writes the validation result and evidence to the *Hypothesis Memory*. The figure 4.2 visualizes the full process of the *Hypothesis Validator* sub-agent.

**Implementation Planner Sub-Agent**

The *Implementation Planner* and *Implementer* sub-agents correspond to the "Fix" and "Iteration" steps from Felgentreff's process in section 3.2. They are split because decomposing complex tasks into specialized subtasks with role-specific agents has been shown to improve task completion over single-agent approaches [72].

The *Implementation Planner's* purpose is to create an implementation plan to fix a verified performance issue. To do so, it reads from *Hypothesis Memory*, *Benchmark Baseline Memory* and *Learned Lessons Memory* first to understand the current state of the optimization process. Afterwards, it is asked to create an implementation plan for a confirmed hypothesis with related benchmarks and write it to the *Implementation Plan Memory*.

**Implementer Sub-Agent**

The purpose of the *Implementer* sub-agent is to implement the plan from the *Implementation Planner* sub-agent and verify that the behavior of the language implementation did not change.

Therefore, it reads the *Implementation Plan Memory*, implements the plan step by step and verifies the behavior by checking build process and automated tests. Afterwards, it runs the verification benchmarks specified in the plan to confirm the change works as expected, otherwise it iterates.

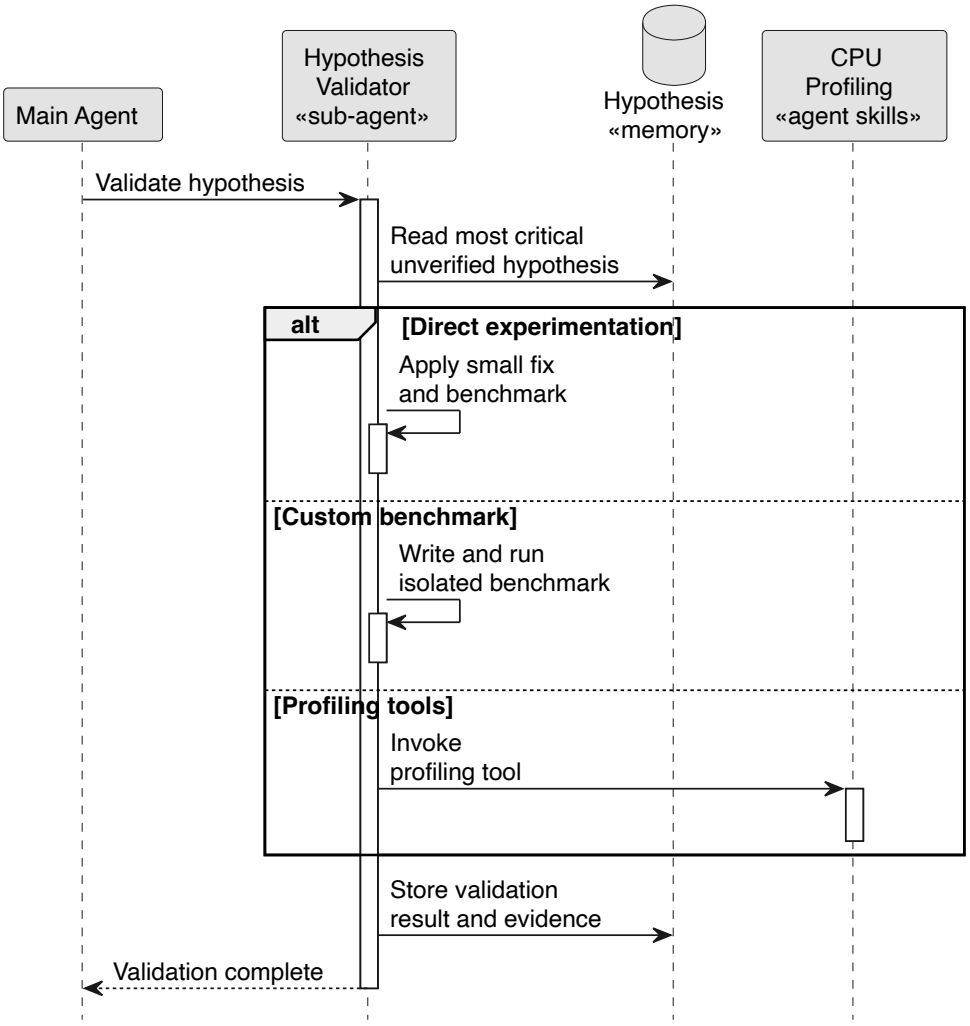



Figure 4.2.: Interaction of the *Hypothesis Validator* sub-agent with external systems, such as the CPU profiling agent skill

## 4.6. Full Orchestration Level

The Full Orchestration level allows executing the full optimization process autonomously. This realizes the *Workflow* requirement fully and the *Autonomy* requirement to the highest autonomy level. Previously, in the Process Assistance level, the developer decides which step to run and interprets the response. These decisions now lie in the definition of the third level.

The process is described in an agent skill, including which sub-agent to invoke, their order, and how to continue with the response. This only requires a static process description, loaded on demand. The process requires invoking the Process Assistance level sub-agents, which ruled out a sub-agent definition for this level as not every AI coding agent supports nested sub-agents.

An alternative is a prompt. The agent skill content is essentially a process prompt, but it can be invoked on demand. The agent can decide autonomously, if the orchestrator agent skill is required. This reduces the risk of context pollution described in section 2.5.1.

With this mechanism in place, the optimization process is split into the phases from the expert's process in section 3.2. Each phase delegates the task to the corresponding sub-agent from the Process Assistance level. The main agent with the invoked agent skill does not perform analysis or implementation tasks. Instead, it is a pure coordinator, which reads the results from the memories and the responses and decides the next action. This prevents irrelevant information from having an effect on decisions.

It starts by creating a benchmark baseline with the *Baseline Establisher* sub-agent if the *Benchmark Baseline Memory* does not exist. It then continues by exploring the codebase and developing hypotheses with the *Explorer* sub-agent. The main agent decides given the most critical hypothesis, whether the validation proceeds with the *Hypothesis Validator* or the *Compiler Graph* sub-agent. This decision is based on the LLM's reasoning, the hypothesis context and sub-agent descriptions. For instance, if a hypothesis concerns compilation behavior visible in the compiler graph, the agent routes to the Compiler Graph sub-agent.

If a hypothesis was validated successfully, it passes the hypothesis to the *Implementation Planner* sub-agent. The main agent reviews the resulting plan and may request revisions. Afterwards, the *Implementer* sub-agent implements the plan and the main agent validates the performance improvement by running the benchmarks again. It compares the execution time results to the baseline with the *Calculate Speedup* tool, commits, and refreshes the baseline.

If the hypothesis validation fails, the main agent continues with the next hypothesis. If all fail, it reruns the *Explorer* sub-agent. If the final speedup validation fails, it reverts to the exploration sub-agent again.

A single execution of the optimization process rarely exhausts all improvement potential. Multiple iterations of this process are recommended. But the coding agent may re-attempt previously failed approaches, because of the context isolation in Process Assistance level sub-agents. The orchestrator is designed to execute multiple iterations without repeating itself. Therefore, it maintains the *Learned Lessons Memory*, which persists across iterations. After failed hypothesis validations and failed or successful implementation attempts, it writes generally applicable learnings about Truffle and performance optimizations to *Learned Lessons Memory*.

To prevent context pollution as described in section 2.5.1, after each iteration the orchestrator removes intermediate artifacts, like tool outputs, compiler graphs, *Implementation Plan Memory* and the *Hypothesis Memory*. Only the *Benchmark Baseline Memory* and the *Learned Lessons Memory* remain. The next iteration should only start with cumulative knowledge not the residual data from previous attempts.

The figure 4.3 shows the complete process of the agent skill.

As an example, consider a single iteration on a Lox implementation whose `permute` benchmark runs slower than the CPython reference. The *Baseline Establisher* runs the benchmarks for both language implementations and writes the execution times to the *Benchmark Baseline Memory*. The *Explorer* reads the memory, inspects the codebase top-down, and writes two hypotheses to the *Hypothesis Memory*. The top hypothesis suspects a missing Truffle Boundary on a string-manipulation helper. The main agent routes this hypothesis to the *Hypothesis Validator* rather than the *Compiler Graph Analyst*, because the symptom concerns a deoptimization loop rather than the IR shape. The *Hypothesis Validator* selects its profiling tool branch and invokes the *Tracing Compilation Events* agent skill. It observes repeated opt deopt events on the suspected method and confirms the cause with the *Trace Transfer to Interpreter* agent skill. Afterwards, the *Implementation Planner* drafts the fix and writes it to the *Implementation Plan Memory*. The *Implementer* applies the plan and runs the verification benchmarks. Finally, the main agent calls the *Calculate Speedup* tool on the new benchmark run, confirms the improvement, commits, and refreshes the baseline.

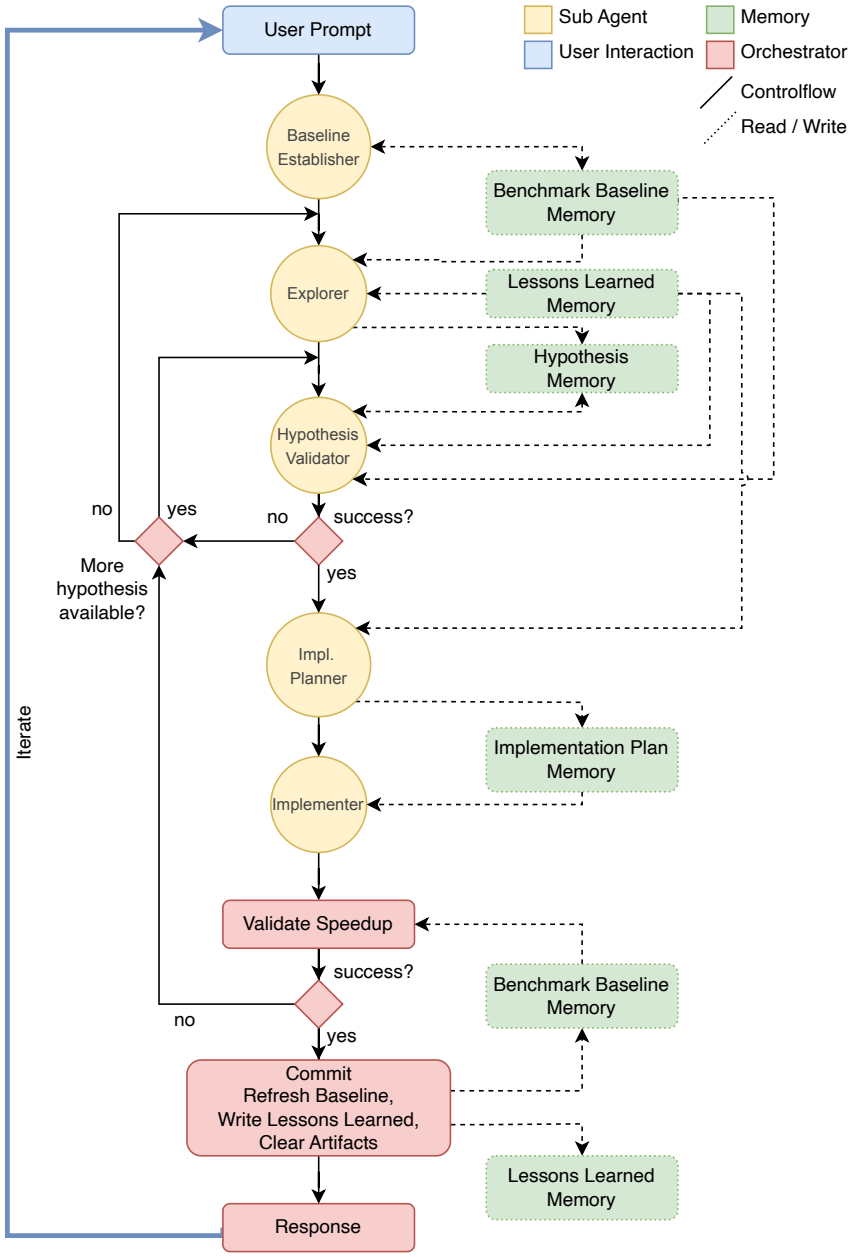



Figure 4.3.: Diagram of the process at the Full Orchestration level. The sub-agents come from the Process Assistance level. The same memory name indicates the same memory instance.

## 4.7. Design Trade-offs

The optimization process encoded in the Full Orchestration level is derived from a single contextual inquiry session with one expert. This grounds the design in real practice, but it encodes one specific workflow as the canonical sequence. A different expert might prioritize tools differently or skip phases entirely.

Additionally, several critical decisions in the approach are delegated to the LLM's reasoning. The *Explorer* ranks hypotheses by criticality, the orchestrator selects the validation approach, and the *Hypothesis Validator* interprets whether evidence confirms or falsifies a hypothesis. These decisions are not deterministic, because LLM reasoning is non-deterministic. The same input can produce different rankings or validation decisions across runs, which affects the reproducibility of the full optimization process. But, encoding these decisions as deterministic rules would require anticipating the full range of hypothesis types and codebase characteristics across language implementations. This results in a complex rule set that would need continuous maintenance.

Moreover, the approach relies heavily on sub-agents for the Process Assistance level. This optimizes context usage by isolating each process step, but this isolation comes at a cost. Each sub-agent starts with an empty context and must re-read the codebase, memories, and relevant artifacts from scratch. This results in high token usage, because the same files are read and processed multiple times across a single optimization run. This means, that the execution time and API costs increase substantially. Context isolation improves the accuracy of individual steps by removing irrelevant information, but duplicates work that a single-agent architecture would perform only once.

Additionally, the benchmark configuration used in the *Baseline Establisher* was adopted from the seminar setup rather than the original AWFY reference configuration. The key difference is the number of outer iterations: 20 compared to 3,000 in the original configuration [50]. While the total invocation count exceeds Graal's default compilation threshold of 1,000, the reduced number of outer iterations means that warm-up iterations occupy a larger share of the measurement. These deviations are applied consistently across all conditions.

Another trade-of is that the Tool Assistance level is tightly coupled with the GraalVM version 24 and not directly applicable to other DSL frameworks or older versions with other profiling tools. However, the Process Assistance and Full Orchestration level can be reused for other frameworks.

# 5. Implementing the Claude Code Plugin

This chapter describes how the system designed in chapter 4 was realized using Claude Code, including packaging, mapping of the approach components to the realizations and key implementation decisions.

## 5.1. Development Environment

The implementation used Claude Code version 2.1.31, Claude Sonnet 4.5 model version 20250929, and Claude Haiku 4.5 model version 20251001, Java 24.0.1 on GraalVM CE, GraalVM version 24.2.1 with Truffle and Bytecode Interpreter enabled, and ANTLR version 4.12.0. For the language implementation in question, it requires the use of GraalVM version 24 at least and the presence of the AWFY micro benchmarks.

During the development of the Claude Code extension, we used the project of the group 2 from the BYOPL seminar in the winter semester 2024/2025. It contains an implementation of the Lox Language, which was described in section 2.4. It includes the implementations of the AWFY benchmarks towers, sieve, queens, permute, list from the seminar. We added bounce and storage implementations.

The implementation utilizes Claude Code and integrated tools to develop agent skills and sub-agents. It was tested against multiple Lox implementations, each level individually, and refined iteratively based on observed agent behavior.

## 5.2. System Overview

To allow an easy integration of the contribution into an existing Claude Code setup, we decided to use a Claude Code plugin, introduced in version 2.0.12 [5] and constructed as a folder. The plugin is called `cc-truffle-performance-plugin`.[1]

Claude Code plugins allow the packaging of an MCP server, definitions of agent skills and sub-agents into a single deliverable. The figure F.1 shows the

[1] https://github.com/hpi-swa-lab/cc-truffle-performance-plugin

final folder structure of the plugin "Truffle Performance Plugin". Developers can use the plugin by downloading it, for instance to `./cc-plugins/`, and starting Claude Code with an additional flag:

```
claude --plugin-dir=./cc-plugins/cc-truffle-performance-plugin
```

Add a skill to a plugin requires a new folder in the `skills/` directory with the skill name.Inside, there must be at least a `SKILL.md` file with a name and description in YAML front matter and Markdown instructions. The name and description are used to determine when the skill should be invoked and the Markdown instructions get inserted into the LLM context on demand. The skill folder can also bundle additional resources, for instance a `PATTERN.md` file referenced in the `SKILL.md` of the *Profiling with CPU Sampler* skill.

A custom sub-agent consists of only one file in the `agents/` folder with a YAML frontmatter and Markdown instructions without any subfolders. The frontmatter should include the name, description of the sub-agent, the model and optionally permission restrictions.

A user can invoke sub-agents and agent skills directly as slash command with the plugin name and invocable name in the prompt, for instance:

```
/cc-truffle-performance-plugin:baseline-establisher
```

MCP server is registered in the `.mcp.json` file. It allows the registration of local MCP servers inside the plugin directory.

## 5.3. Component Realization

This section maps each component of chapter 4 to the concrete file in the plugin.

Therefore, the agent skills of the Tool Assistance level are added to the `skills/` directory, each following the same template. The agent skill of the Full Orchestration level is also added to the same directory containing the process description and the connected tasks.

The sub-agents from the Tool Assistance level and the Compilation Graph Analyst are added to the `agents/` folder, whose Markdown instructions follow the description from chapter 4.

The memories, which are a named, persistent storage units that any sub-agent can read from or write to, are realized as plain Markdown files in the project directory. These are readable by both the agent and the developer. For instance, the *Baseline Establisher* sub-agent writes the `BENCHMARK_BASELINE.md` file for the *Benchmark Baseline Memory*.

The table 5.1 provides the full mapping between the approach component and their plugin files. Together, the plugin realized all five requirements identified in chapter 4.

Table 5.1.: Mapping between approach component and plugin file path in the Claude Code plugin.

| **Approach Component** | **Plugin Path** |
|---|---|
| *Benchmark Comparison Tools* | |
| Calculate Speedup | `mcp-awfy/` → `compute_geomean_speed_up` |
| Run Reference Language | `mcp-awfy/` → `run_python_benchmark` |
| *Tool Assistance level* | |
| Detecting Deoptimizations | `skills/detecting-deoptimizations/` |
| Detecting Performance Warnings | `skills/detecting-performance-warnings/` |
| Profiling Memory Allocations | `skills/profiling-memory-allocations/` |
| Profiling with CPU Sampler | `skills/profiling-with-cpu-sampler/` |
| Tracing Compilation Events | `skills/tracing-compilation-events/` |
| Tracing Execution Counts | `skills/tracing-execution-counts/` |
| Tracing Inlining Decisions | `skills/tracing-inlining-decisions/` |
| Compiler Graph Analyst | `agents/compiler-graph-analyst.md` |
| *Process Assistance level* | |
| Baseline Establisher | `agents/baseline-establisher.md` |
| Explorer | `agents/explorer.md` |
| Hypothesis Validator | `agents/hypothesis-validator.md` |
| Implementation Planner | `agents/implementation-planner.md` |
| Implementer | `agents/implementer.md` |
| *Full Orchestration level* | |
| Optimization Workflow Orchestrator | `skills/optimization-workflow-orchestrator.md` |

## 5.4. Profiling and Analysis Skills

All agent skills of the Tool Assistance level follow the same template. The listing G.1 shows the full *Profiling with CPU Sampler* skill as example.

The description in the YAML frontmatter highlights the purpose of the Graal tool and when to use it. The Markdown body uses headings to separate sections, code blocks for CLI commands, and checklists for multi-step verification procedures.

At the beginning, each skill starts with a list of scenarios when to use it. A quick start guide contains the concrete CLI commands with the flags in a code block. A list of all relevant options and command line flags, their descriptions and input schema follows.

Afterwards, the Fermi Verification step from section 4.4 is realized as a Markdown checklist. The agent is asked to predict the result of a smoke test. After running the smoke test, it compares the prediction with the real result. The checklist helps the agent to follow the structure, which listing 5.1 shows for the *Profiling Memory Allocation* skill as an example.

Listing 5.1: Fermi Verification checklist from the Profiling Memory Allocation skill.

```
## !REQUIRED: Fermi Verification (Every Tool Invocation)
**Before running**:
- [ ] Pre-calculate: Expected allocation count
    (estimate based on loops/iterations)
- [ ] Smoke test: `<launcher> -{}-memtracer -c 'var a = [1, 2, 3];'`
    →Verify shows allocations
**After running**:
- [ ] Validate: Actual vs estimate within 1 order of magnitude? YES / NO
- [ ] If NO: **STOP** - Debug tool before proceeding
    (test with known allocating code)
- [ ] Save output: `tool-outputs/memtracer-[benchmark].txt`
**Gate**: All boxes checked? →Proceed to analysis
```

The output section includes steps to interpret the output, patterns to look out for, and thresholds. The patterns are extracted from online documentation, the expert interview and our observations during prototyping. If the list of patterns is too long, then they are referenced in a `PATTERN.md` file. It is recommended to write the tool output to a file in the local folder `tool-output/`. For tools with large output, the skill instructs the agent to pipe results through `grep` to extract relevant lines before reading the full output. This prevents large profiling output from consuming the context window. Afterwards, a list of related skills recommends next steps based on the interpreted output and identified potential issues.

The following observations emerged during iterative development of the skills. It turned out that the length of the skill description matters, which was already described in section 2.5.1. The solution was to split the material into separate files, for example a new `PATTERN.md`. The agent loads these files when needed. But based on informal observations across multiple runs, the agent only reads the file sporadically, even if the skill highly recommends the referenced file.

Additionally, prototyping revealed that the Markdown instructions should be imperative, step-by-step procedures rather than declarative descriptions [4]. A prototype version described the tools like documentation, for instance "X can be used to check Y". When switched to "Run X, then check for Y," the agent began following the instruction. One possible explanation is that the additional reasoning step to select X in a declarative description is an additional hurdle to select it. The listing 5.2 shows two formulations as an example for the *Profiling Memory Allocation* skill.

Finally, the introduction of the Fermi Verification helped interpret the output of malfunctioning tools correctly. For instance, the memory tracer requires setup in the language implementation, which is not present in the given project. As a result, the memory tracer responds with zero detected allocations. The agent concludes that the compiler performs excellently. With the Fermi Verification it flags that zero allocations are implausible for an array initialization.

Listing 5.2: Comparison between declarative and imperative style for the Profiling Memory Allocation skill.

```
// Declarative
Profiling Memory Allocation skill can be used to identify objects that
escape hot loops. Escape analysis effectiveness can be verified by
examining allocation counts relative to iteration counts.

// Imperative
If hot loop shows many allocations:
1. Check if objects escape the loop
2. Verify escape analysis is running
3. Consider restructuring to avoid allocation
```

## 5.5. Compiler Graph Analyst

The *Compiler Graph Analyst* is implemented as a sub-agent, which was explained in the chapter 4. The following section focuses on the realization. The main agent invokes the sub-agent with a question about the compilation, which can include a hypothesis. Afterwards, it generates compiler graphs by running a specified program with the suitable command line flag.

The output consists of binary `.bgv` files, which cannot be processed by an agent without postprocessing. As mentioned in section 3.3.4, developers visualize the compiler graph as a colored graph in IGV. This format is not optimal for agents to read. That means it requires a tool to parse and transform the binary data into an agent-readable format that the agent can use to interpret the compiler graph. Shopify developed "seafoam" [78], a command line tool to work with compiler graphs and `.bgv` files. We use the version 0.17. It was selected as the only actively maintained open-source tool for `.bgv` processing, aside from IGV as graphical interface. It allows traversing the graph via the CLI or converting it into a `JSON` file with the `bgv2json` function. A coding agent can use either the seafoam CLI or query the `JSON` file with `jq` as CLI commands.

The system prompt instructs the agent to look for disproportionate graph size relative to source complexity, specific node types indicating missed optimizations or deoptimization points visible in the graph structure. The sub-agent specification includes seafoam commands and `jq` queries, while also

allowing the agent to create its own queries. Therefore, the sub-agent definition provides `bgv2json` output format so the agent knows what to query.

The sub-agent responds to the main agent with an answer to the original question, evidence, and examples.

Because Claude Code is one of the coding agents, which does not support sub-agents calling other sub-agents as of now, asking the *Compiler Graph Analyst* sub-agent is another way to validate a hypothesis, next to the *Hypothesis Validator*.

## 5.6. Benchmark Comparison MCP Server

The tools *Calculate Speedup* and *Run Reference Language* are implemented as MCP tools. Therefore, the Claude Code plugin contains an MCP server implementation, which uses Python 3.13 with `uv`, fastmcp 2.14 as the MCP server framework, and pydantic 2.12 to validate the input and response.

In the plugin the local server is registered with a `.mcp.json` file configuring the server start command like in listing F.1. The server starts as local process when Claude Code initializes a new session.

The MCP server provides two tools: `run_python_benchmark` implements *Run Reference Language*. Given a name of an AWFY benchmark, inner, and outer iteration count, it runs the benchmark locally. The package contains all original AWFY Python implementations, looks them up, and runs them on request. It returns the minimum, maximum, sum, and average of all execution times (in milliseconds), together with the input.

The second tool `compute_geomean_speed_up` implements *Calculate Speedup*. As input it expects a list of baseline execution times, a list of execution times to compare with, and the list of benchmark names in the same order. After input validation, it calculates the speedup ratio for each benchmark and an overall speedup with the geometric mean. The response contains the benchmark-wise speedup, the global speedup, and whether the execution times being compared are faster overall.

One crucial learning is that the sub-agent and agent skill description must explicitly call the MCP tool. Otherwise, even though the description highly recommends using it for comparisons, the agent calculated the speedup on its own with the average. This was observed informally during prototyping.

# 6. Study Design

The following chapters evaluate whether the approach can achieve a better performance improvement when using the agent with the knowledge enhancements and if the full system assists the Truffle-inexperienced developers. This chapter focuses on how to measure it.

## 6.1. Research Approach

The evaluation uses a mixed methods approach with a qualitative and a quantitative part to answer each question.

The quantitative part consists of controlled experiments with Lox implementations. It investigates if a coding agent with the knowledge enhancements achieves greater optimization than one without the enhancements. This is related to the functional requirements. In the experiment both agent configurations are asked to optimize a Lox implementation.

The qualitative part includes a user study with students from the BYOPL seminar, accompanied by a thematic analysis. The goal is to determine whether the agent and its enhancements together compensate for the lack of expertise compared to using only the provided GraalVM profiling tools. It relates to the *Integration* and *Autonomy* requirements.

## 6.2. Experimental Study Design

The goal of the experiments is to evaluate the effect of the knowledge enhancements on the resulting optimization, when the agent is asked to optimize a given language implementation.

The independent variables in the experiments have two conditions.:

Unenhanced Agent: Claude Code with a `Claude.md`
Enhanced Agent: Claude Code with a `Claude.md` and the developed Claude Code plugin

Both conditions include read and write access to the given language implementation, including the micro benchmarks from AWFY benchmark suite. Additionally, they have also access to run CLI commands and to do web

searches. The language implementation comes with unit tests provided by the seminar. The same versions from the development setup in chapter 5 are used. The Claude Code internal memories feature is disabled to prevent knowledge sharing across experiments.

### 6.2.1. Experiment Data

The experiment requires representative language implementations as input data. Two Lox implementations from the BYOPL seminar serve as datasets:

Original Implementation (OI): The Lox Language implementation from group 7 of the BYOPL seminar.
Degraded Implementation (DI): A modified variant tge Lox Language implementation from group 7 of the BYOPL seminar with deliberately introduced performance issues.

The implementation of group 7 was selected, because it is correct based on the provided unit tests and the performance compared to the other groups is already mature, but not as fast as the reference implementation *swalox*. That means it still includes potential for improvement. The modifications in the degrated implementations are representatives of common Truffle anti-patterns presented in BYOPL seminar and target four categories:

Missing specialization: We collapse binary, arithmetic, and comparison operations into monomorphic fallbacks instead of individual specializations per type (example: listing I.1).
Missing Inline cache: We disabled inline caching for function calls and method look up to prevent efficient lookup (example: listing I.2).
Data structure regressions: We replaced the efficient array representation with a slower `ArrayList` (example: listing I.3).
Misplaced Truffle Boundaries: We added Truffle Boundaries to prevent efficient compilation of frequently called code (example: listing I.4).

### 6.2.2. Experiment Procedure

The experiment procedure consists of ten runs ($n = 10$) each with five sequential iterations. Five iterations were chosen based on observations during prototyping, where performance gains typically stagnated after three to four optimization cycles. Ten runs per condition balance the need to capture LLM output variance with the practical constraints of experiment budget.

One iteration is one zero-shot prompt invocation of an agent with the described conditions. The prompt asks the agent to identify and fix exactly one performance issue per iteration without user interaction. After each iteration, it is asked to commit the changes for traceability. It instructs a self-correction mechanism, to fix broken build and tests, limits the agent to one issue and recommends file-based memory across iterations. We added one line to the prompt for the Enhanced Agent invoking the orchestrator skill, which isolates the plugin's effect. The full prompts are documented in Appendix J.

The Claude session runs inline with a special flag to enable unattended execution. For example, it disables all human verification of Claude Code actions. This allows evaluating only Claude Code capabilities.

```
claude --dangerously-skip-permissions
```

One iteration lasts a maximum of 1.5 hours, a reasonable amount of time a developer would take for this task from observations during prototyping. If the session is not finished after this time, the iteration is canceled, the changes reverted to the previous iteration and the next iteration starts.

A run lasts a maximum of 4 hours, because this equals a half working day for a full time developer. That means, it is possible that not all iterations finish.

Before each run and after each iteration the experiment environment builds the project, executes the benchmarks as described in table A.2, and extracts the execution times for further analysis. We decided to exclude Mandelbrot and nbody from the benchmark. Unlike the remaining seven micro benchmarks, which originate from the SOM Smalltalk benchmark suite, both are derived from the Computer Language Benchmark Game and carry a different design heritage. Additionally, they are outliers within the suite and prevent a uniform analysis. The benchmark configuration also differs and uses smaller iteration counts, because of limited resources.

If the build or a benchmark fails, the same behavior as for a timeout is applied. This also reflects the behavior in a real-world scenario.

The full experiment procedure is reflected in the Algorithm B.1 and Algorithm B.2. This experiment procedure is executed for each combination of agent condition and implementation. They run on a MacBook Pro 2023, with an Apple M3 Pro chip and 18 GB memory.

### 6.2.3. Variables and Metrics

For the analysis of the experiments, we defined two dependent variables. The most relevant dependent variable is the geometric mean speedup $S$ over all AWFY micro benchmarks, where $S > 1$ means it is faster than the baseline and $S < 1$ means it is slower. If $b_i$ is the baseline execution time and $a_i$ is the optimized execution time for benchmark $i$, with $|B|$ benchmarks from the set $B$, then $S$ is defined as

$$S = \left( \prod_{i=1}^{|B|} \frac{b_i}{a_i} \right)^{\frac{1}{|B|}} \tag{6.1}$$

The section H.1 contains an example calculation.

The second dependent variable is the token cost $C$, measured in US dollars. It measures how much a full optimization session would cost for the end user, if they paid for the tokens instead of using the subscription model. It can be calculated based on the usage field in the server responses, which are summarized in table C.2. Let $M$ be the set of models used during a run. For each model $m \in M$, let $r_m^{\text{base}}$, $r_m^{\text{write}}$, $r_m^{\text{hit}}$, and $r_m^{\text{out}}$ denote the per-token prices for base input, cache writes, cache hits, and output respectively. For each API request $j$ using model $m$, the response contains four token counts: base input tokens $t_j^{\text{base}}$, cache write tokens $t_j^{\text{write}}$, cache hit tokens $t_j^{\text{hit}}$, and output tokens $t_j^{\text{out}}$. The total cost of a run with $k$ requests is defined the following, where $m_j \in M$ is the model used for request $j$.

$$C = \sum_{j=1}^{k} \left( t_j^{\text{base}} \cdot r_{m_j}^{\text{base}} + t_j^{\text{write}} \cdot r_{m_j}^{\text{write}} + t_j^{\text{hit}} \cdot r_{m_j}^{\text{hit}} + t_j^{\text{out}} \cdot r_{m_j}^{\text{out}} \right) \tag{6.2}$$

The per-token prices are based on Anthropic's published pricing at experiment time [14]. The usage information is derived from the `.jsonl` Claude Code session files. The section H.2 contains an example calculation.

The analysis combines descriptive metrics and inferential testing. The inferential component compares the geometric mean speedup after five iterations between conditions with statistical testing. The descriptive component reports additional metrics without statistical testing, because the sample size ($n = 10$) limits the power of inferential tests for secondary metrics.

The inferential component addresses whether the enhanced agent achieves higher geometric mean speedup than the unenhanced agent on AWFY micro-benchmarks. The hypothesis are defined as the following:

$H_0$ The Enhanced Agent does not achieve a higher geometric mean speedup than the Unenhanced Agent.

$H_1$ The Enhanced Agent achieves a higher geometric mean speedup than the Unenhanced Agent.

The comparison is conducted per implementation between the Unenhanced and Enhanced Agent using a one-tailed Mann-Whitney U test ($\alpha = 0.05$) over the geometric mean speedup values of the $n = 10$ runs per condition. Effect size is reported as rank-biserial correlation $r$. Mann-Whitney U is chosen because the small sample size ($n = 10$ per condition) does not guarantee normal distribution of speedup values. The test is one-tailed because the research question asks specifically whether the Enhanced Agent outperforms the Unenhanced Agent, not whether they differ.

As a first descriptive metric, the completion rate indicates how often each condition produces a usable result. We track the number of iterations with failed builds or benchmarks, as well as the count of started, finished, and language-modified finished iterations. If completion rates differ substantially between conditions, the inferential comparison of speedup values must be interpreted with that context.

As a second descriptive metric, we compare the per-iteration improvement distribution.Therefore, we count the number of iterations with code changes that result in performance improvements, no performance change, or regressions. We define a performance improvement as $S > 1.03$ to account for the benchmark runtime variance. We define a performance regression as $S < 0.97$.

As a third descriptive metric, we report the development of the performance improvements over iterations. Specifically, for each run how did the geometric mean speedup $S$ at each iteration mean developed over the iterations compared to the start.

As a fourth descriptive metric, we classify for each iteration the focused component of the included code change, either Truffle-focused or Java-focused. Truffle-focused changes require knowledge about Truffle, while Java-focused changes do not. This reveals whether the enhanced agent finds qualitatively different issues rather than just more of the same.

Additionally, as fifth metric, we compare the final performance of both conditions against the reference implementation *swalox* and CPython 3.13 with the geometric mean speedup $S$.

Finally, as sixth descriptive metric, we want to report the geometric mean speedup $S$ over the total costs per condition $C$. A discussion about the efficiency of the knowledge enhancement will take place in that context.

## 6.3. User Study Design

The goal of the user study is to find out which aspects of the expertise gap the enhanced coding agent addresses and which ones it does not. Therefore, the study focuses on how developers with limited Truffle optimization knowledge approach the task with and without the enhanced coding agent.

The selected method includes the think-aloud protocol, where the participants work on Truffle language optimization and speak their thoughts out loud [16]. The sessions are transcribed and the screen of the developer recorded with OBS Studio [59]. The think-aloud protocol is the right study style, because it allows capturing the developer's workflow and behavioral differences between the two conditions. The study script is available in Appendix K.

The participants are two master's-degree students, who attended the BYOPL seminar in the winter semester 2024/2025. Given the time passed since then, both know the used frameworks but have already forgotten details. The small participant group is justified by limited experiment resources and limited number of available participants, who know parts of the Truffle framework and Lox language, but are not experts in it. This mirrors the situation of a domain developer with limited Truffle experience, who wants to improve the language implementation performance.

### 6.3.1. User Study Setup

In preparation of the user study, we prepared two participant setups:

Participant Setup Manual (PS-M) The participant has access to the codebase, CLI, and online resources.

Participant Setup Agent (PS-A) The participant has access to the codebase, CLI, online resources, Claude Code with the plugin, and the corresponding user manual.

Both participant setups correspond to the study conditions. The manual setup deliberately omits Claude Code, because the quantitative analysis targets the effect of the full system and whether the enhanced agent closes the knowledge gap compared to the developer working alone.

The participants work on two different language implementations: one is the degraded implementation from the experimental study, and the other is a degraded variant, DI-2, of the original implementation with the same performance issue categories but different concrete performance issues. The second variant prevents participants from benefiting from seeing the same bug twice and keeps the two parts balanced.

### 6.3.2. User Study Procedure

One session starts with answering demographic questions, followed by two main parts. In both parts, the participant starts with familiarizing themselves with a given Lox implementation, either of the first or second variant of the degraded implementation. Afterwards, they have to find and fix a performance issue in the language implementation with a given manual or agent setup. The structure of one session is described in figure 6.1.

To minimize the impact of learning between the phases of the experiment, both participants receive the setups in a counterbalanced order. That way, they work on different codebases per setup as visualized in table 6.1. Implementation order is not counterbalanced, because both degraded implementations contain performance issues from the same categories with comparable difficulty, making order effects unlikely.

Table 6.1.: Assignment of participant setups and implementations to participants.

| | Part 1 | Part 2 |
|---|---|---|
| Participant A | PS-M on DI | PS-A on DI-2 |
| Participant B | PS-A on DI | PS-M on DI-2 |

| **Study Session (2h 50min)** | | | | | |
|---|---|---|---|---|---|
| Demo-graphics | **Part 1** (1h 20min) | | **Part 2** (1h 10min) | | Finishing Up |
| (10min) | Recap (20min) | Optimization (1h) | Recap (10min) | Optimization (1h) | (10min) |

Figure 6.1.: Structure of a user study session

### 6.3.3. User Study Evaluation

The sessions were conducted and transcribed in German with podwriter [44] as auto-transcription tool. The results were reviewed and corrected manually afterwards. Quotations are translated by us and the original German text is provided in footnotes.

The analysis of the sessions is a reflexive thematic analysis on the transcriptions by Braun and Clarke [19], which focuses on depth instead of generalizability. A qualitative thematic analysis was chosen because the small participant group does not permit quantitative analysis, but does support pattern identification and cross-condition contrasts. Thematic analysis allows a single coder, and does not require blinding or external coders.

The initial codes of the thematic analysis are structured around the CDN framework by Blackwell et al. [18] with deductive codes derived from its dimensions. The analysis is also open for inductive codes for developer experience aspects, which are not covered by CDN. The open-source qualitative analysis tool Taguette is used as coding tool [73].

Codes derived from the CDN dimensions allow a good starting point for analyzing the developer experience, because it focuses on the interaction with information artifacts, for instance CLI profiling tools and enhanced coding agents. Alternative frameworks, for instance the Nielsen heuristics[55] are either Graphical User Interface (GUI) specific or disproportionately heavy for this study's scope.

In addition to the thematic analysis, the progress of the participants in the optimization process will be reported.

## 6.4. Threats to Validity

The threats to validity are split into four groups based on the four validity types from Cook and Campbell [24], later refined by Shadish, Cook, and Campbell [76]: Internal, external, construct and conclusion validity.

### 6.4.1. Internal Validity

One of the most impactful threats to internal validity is the non-deterministic nature of the LLM. Not only do different models produce different answers, but even the same Claude Sonnet model yields varying outputs. This has an effect on both study parts. In the quantitative part, we mitigate this with repeated runs ($n = 10$) to capture output variance.

A second internal threat is the use of hosted models by Anthropic. We do not have control over consecutive changes to the models by Anthropic on their

servers, whose accuracy fluctuates over time. Moreover, Anthropic suffers from recurring incidents, which forced us to stop the evaluation. Because of the cost constraints, we cannot run the experiments in parallel, but on different days with potentially different versions. This also has an effect on both study parts. We tried to minimize this effect with pinned Claude Code, Claude Sonnet and Claude Haiku versions.

Another threat is the effect of the used tokens in the quantitative part. If the Enhanced Agent consumes more tokens and produce greater performance improvements, the speedup could partly reflect higher computation effort and not structured guidance. Capping the token budget as mitigation strategy is not an option, because it would stop the agent in the middle of the process without any result. The the speedup-to-cost ratio must be interpreted within that context and only a descriptive analysis will be made.

A further concern is, that the run and iteration time limits in the quantitative part are pragmatic, but arbitrary. Longer or shorter runs might yield different results.

For the qualitative part, the participant in the user study knows that we built the plugin and that we observe them during the study. This may lead to self-censored criticism or perform artificial enthusiasm, even unconsciously. To minimize this effect, we reminded the participants that honest feedback is more valuable to the study, though this cannot fully eliminate the effect.

### 6.4.2. External Validity

A threat to external validity is the small participant group size in the qualitative part, due to pool of potential participants and limited resources. Additionally, no pilot study for the qualitative study was conducted. The timing, and task difficulty are untested, potentially leading to floor or ceiling effects or unclear instructions.

Another threat, as Felgentreff mentioned in the interview[1] [28, 01:14:34], is that Lox is a toy language and the implementation complexity is not comparable with sophisticated programming languages.

A further concern is, that in the quantitative part only one group of LLM models is tested, which limits generalizability to other LLM families.

For both parts, another aspect is that the language implementations from the BYOPL seminar use the Truffle Bytecode interpreter instead of the more common Truffle AST interpreter. The results may reflect the interpreter's immaturity rather than the effectiveness of the enhancements.

[1] Original: *"Lox ist halt noch klein genug, vielleicht kannst du ja alles reinhauen [...] aber [andere Sprachen sind] halt nicht klein genug"*, Translated: "Lox is still small enough, maybe you can just throw everything in there [...] but [other languages] simply aren't small enough."

### 6.4.3. Construct Validity

The construct validity threats are related to the benchmarks in the quantitative part, whether micro-benchmarks are representative of real optimization. The AWFY benchmark framework focuses specifically on measuring compiler effectiveness across language implementations. The inner and outer iteration counts differ from those used by the original AWFY authors, which may affect comparability. However, the configuration is applied consistently across all conditions and accounts for compiler warm-up time and stable performance.

Another concern is, that the think-aloud protocol may alter natural behavior of the participants.

Moreover, measuring the token cost in the quantitative part is not the only user cost. The user time and other required subscriptions also contribute to that.

### 6.4.4. Conclusion Validity

A threat to conclusion validity is the small number of runs in the quantitative part, which also limits statistical power to detect effects of the knowledge enhancements. $n = 10$ is underpowered for Mann-Whitney U to detect moderate effects. The high LLM output variance may increase the effect. To partially address this, Mann-Whitney U was chosen specifically for small samples, and effect size is reported as rank-biserial correlation to supplement significance testing.

A further concern is that the absence of inter-rater reliability checks means the coding reflects one analyst's interpretation, while reflexive thematic analysis by Braun and Clarke explicitly supports single-coder analysis.

# 7. Results & Discussion

This chapter reports and discusses the results of the experiment and the user study. The results of the experiments of the different implementations will be evaluated separately based on the introduced metrics in chapter 6. For the user study, the resulting themes from the thematic analysis will be discussed. A summary collects the main findings at the end.

## 7.1. Experiment Results

This section discusses the results of original and degraded implementation individually. This includes an evaluation of the completion rate, the impact of the iterations, an analysis of the kinds of issues the agent fixed, comparison of the performance improvements over the iterations, a short discussion of outliers, a comparison with the reference implementation swalox and a cost comparison.

### 7.1.1. Original Implementation (OI)

Starting with the original implementation and the iteration completion rate, table 7.1 shows the count of started, finished and finished iteration with language code changes. No iteration result contained a build error or broken benchmarks.

The reason for the lower number of completed iterations is the timeouts. We can see that multiple iterations did multiple implementation attempts and did not result in performance improvements. As a result, iteration and runs timed out and the remaining iterations did not start. The iteration without code changes only documented its findings for further iterations.

This is consistent with a more hypothesis-driven approach, as the agent invested more time per iteration rather than committing a change at every step. It is directly connected to the *Workflow* requirement.

The table 7.2 shows the distribution of iteration outcomes by performance impact measured with the geometric mean speedup $S$. Even though the Enhanced Agent completed fewer iterations, it implemented more code changes with greater impact on performance. Additionally, the number of neutral performance changes is smaller than the one from the Unenhanced Agent.

Table 7.1.: Iteration completion rate for the original implementation across all runs. Started iterations involve the coding agent receiving the prompt. Finished iterations are those where the agent completes its task. Finished iterations with code change are a subset of finished iterations with language implementation modification.

| Iteration State | Unenhanced Agent | Enhanced Agent |
|---|---|---|
| Started | 50 | 46 |
| Finished | 50 | 34 |
| Finished without errors | 50 | 34 |
| Finished with code change | 50 | 22 |

This indicates that the Enhanced Agent identified and fixed more critical performance issues. The Unenhanced Agent's changes, individually seen, had less impact on the performance.

Table 7.2.: Distribution of iteration outcomes by performance impact for the original implementation. Iterations are classified as improvement ($S > 1.03$), regression ($S < 0.97$), or neutral. Each iteration reflects the geometric mean speedup across all AWFY benchmarks.

| Iteration State | Unenhanced Agent | Enhanced Agent |
|---|---|---|
| Improvement ($S > 1.03$) | 8 | 11 |
| Neutral ($1.03 \geq S \geq 0.97$) | 38 | 10 |
| Regression ($0.97 > S$) | 4 | 1 |

However, the types of performance issues the agent fixed can be divided into Truffle-focused and Java-focused bugs. Fixing Truffle-focused bugs requires using Truffle functions and annotations. However, fixing Java-focused bugs does not require any Truffle knowledge. The distribution of performance issues fixed by the agent across iterations supports the claim that the Enhanced Agent adopted a more Truffle-focused strategy, as shown in table 7.3.

The ratio of Truffle-focused changes to Java changes is larger for the Enhanced Agent. The Unenhanced Agent lacks the domain-specific context about Truffle internals and the GraalVM profiling tools, falling back on general Java optimization. This is consistent with the knowledge enhancements influencing the Enhanced Agent's optimization strategy in the desired direction.

When looking at the performance improvements the coding agents achieved over iteration in figure 7.1, no clear trend emerges across runs. The variation

Table 7.3.: Distribution of fixed performance issues for the original implementation. Truffle-focused changes require Truffle functions and annotations. Java-focused changes do not require truffle knowledge to make the change.

| Performance issue | Unenhanced Agent | Enhanced Agent |
|---|---|---|
| Truffle-focused | 33 | 16 |
| Java-focused | 17 | 6 |

is driven primarily by a small number of outliers, discussed below. The Mann-Whitney U test ($\alpha = 0.05$) revealed no statistically significant difference in geometric mean speedup between Unenhanced and Enhanced Agent ($U = 62$, $p = 0.192$, one-tailed, $r = 0.24$, small effect, $n = 10$ per group). That means $H_0$ is not rejected. One larger and two smaller outliers exist with the Enhanced Agent in the positive and one outlier with the Unenhanced Agent in the negative direction. The other runs do not show a significant change in performance.

The small observed effect combined with the limited sample means the study was underpowered to detect a difference of this magnitude. A larger sample would be required to draw concrete conclusions, because the outliers suggest that individual runs vary. This makes a deeper look into the outliers especially interesting.

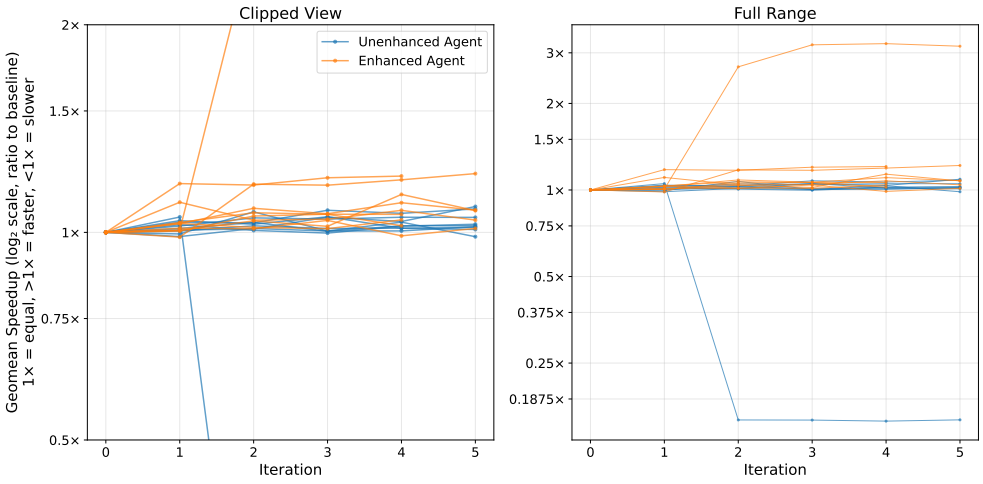



Figure 7.1.: Speedup progression per run for the original implementation, split into two panels with a clipped and a full range.

The figure L.1 shows the individual benchmark performance for the outlier Run 3 of the Enhanced Agent. Iteration two reveals a significant performance improvement, where the coding agent increases the cache limit of read and write property operation from one to three. This improves the handling of polymorphic input patterns. If the same object receives different types as properties, it was forced to fallback to a slow implementation, because it cached the fast implementation only for one type. After the change, it caches the implementation of three input types without using the fallback implementation.

The figure L.2 shows the individual benchmark performance for the outlier Run 9 of the Unenhanced Agent. Iteration two reveals a significant performance regression, where the coding agent used the Java function `String.intern()` in the read and write property operation. The goal was to add the property name to the Java String constant pool on the heap. This should allow fast comparison by identity instead of value. But the function call `String.intern()` is represented as `InvokeNode`, one of the slow node types identified in the chapter 3 as problematic. The figure 7.2 shows an excerpt of the compiler graph with the node.

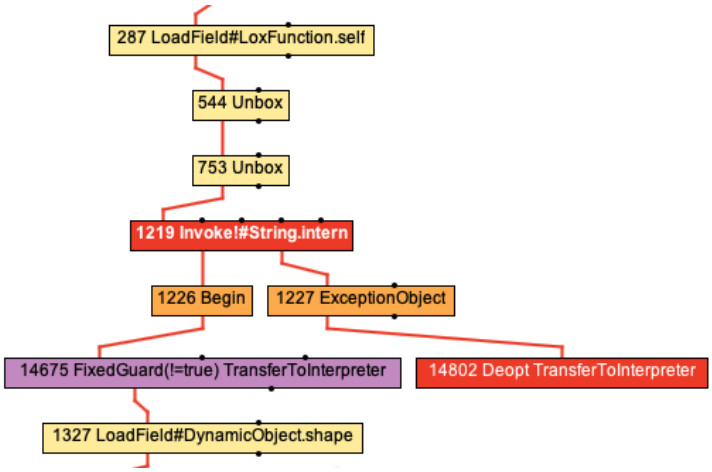



Figure 7.2.: Intermediate Representation of `String.intern()` in Run 9 for the original implementation with "Unenhanced Agent".

The outlier supports that *Tooling* requirement is fulfilled and the agent executed tools and interpreted results correctly.

Returning to the total runs, comparing the final implementations with swalox reveals significant optimization potential still exists. The figure 7.3 shows that the resulting implementation of both Agents is still slower than swalox. The result of the Unenhanced Agent is even slower on average than before using the agent. The final language implementation of the Enhanced

Agent performs better on average. But these statements should be treated with caution, as the errors of both agents are too large and the outliers too significant.

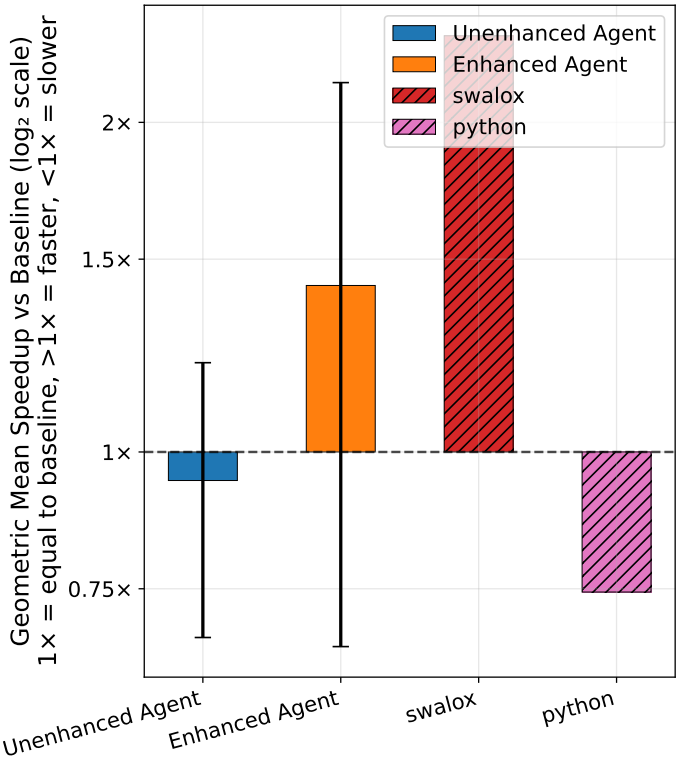



Figure 7.3.: Average geometric mean speedup for the original implementation compared against the swalox and CPython 3.13. The baseline is the unoptimized implementation prior to any agent. Unenhanced and Enhanced Agent values represent the mean geometric mean speedup across all runs. Error bars indicate one standard deviation across runs.

When comparing the API costs $C$ of coding agent sessions with the speedup in figure 7.4, it reveals that the Enhanced Agent costs four to five times what the Unenhanced Agent costs without significant performance gain. A likely explanation, consistent with the design discussions, is use of sub-agents, which start with a fresh and empty context and must repeatedly read the codebase from scratch. This consumes a disproportionate number of tokens per iteration and discussed in chapter 4. While sub-agents were chosen to provide context isolation for theory generation and implementation, this isolation comes at a significant cost in redundant context loading. Additionally, the structured hypothesis-driven approach requires more steps and more tokens.

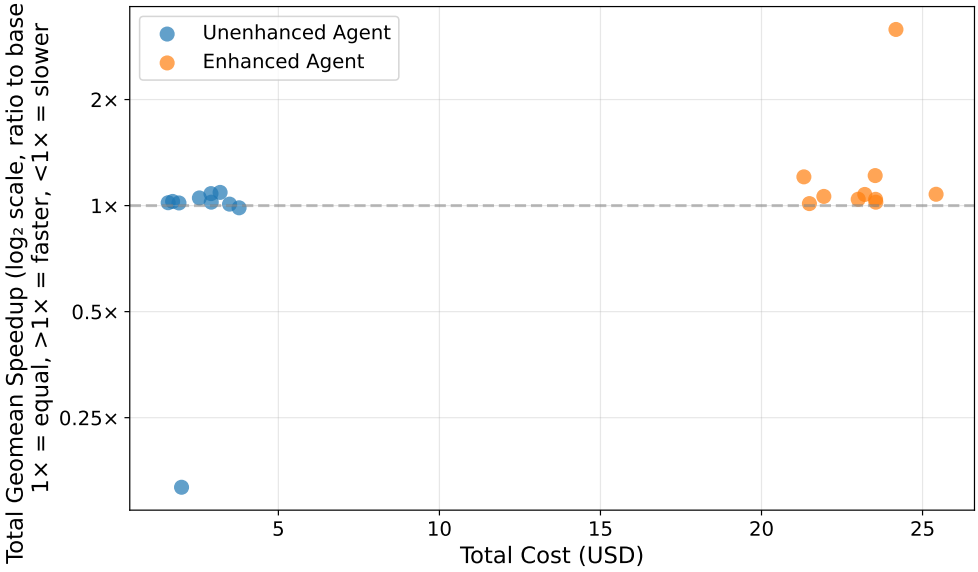



Figure 7.4.: API cost $C$ over total geometric mean speedup $S$ in relation to the baseline before optimization per run for the original implementation

### 7.1.2. Degraded Implementation (DI)

Continuing with the analysis of the degraded implementation results. This modified variant of the original implementation contains deliberately introduced performance issues. The table 7.4 shows the distribution of the started, finished and finished iterations with code changes.

The Enhanced Agent completed fewer iterations than the Unenhanced Agent. The reason for the lower number is timeouts again, suggesting a more hypothesis-driven approach. However, in comparison to the numbers of the original implementation the finished iteration with code change increased. This suggests that the Enhanced Agent validated more issues successfully in the degraded implementation.

The table 7.5 shows the distribution of iteration outcomes by performance impact, either improvements or regressions. The same observation holds as for the original implementation. Although the Enhanced Agent completed fewer iterations, more of them produced a substantial performance impact.

This supports the claim that the Enhanced Agent identified and fixed more critical performance issues than the Unenhanced Agent. The knowledge enhancements improve the iteration precision, while the Unenhanced Agent commits changes without measurable effect.

However, the types of performance issues the agent fixed can be divided into Truffle-focused and Java-focused bugs. Fixing Truffle-focused bugs requires using Truffle functions and annotations. However, fixing Java-focused bugs

Table 7.4.: Iteration completion rate for degraded implementation across all runs. Started iterations involve the coding agent receiving the prompt. Finished iterations are those where the agent completes its task. Finished iterations with code change are a subset of the finished iterations with language implementation modification.

| Iteration State | Unenhanced Agent | Enhanced Agent |
|---|---|---|
| Started | 50 | 49 |
| Finished | 49 | 38 |
| Finished without errors | 49 | 38 |
| Finished with code change | 49 | 38 |

Table 7.5.: Distribution of iteration outcomes by performance impact for the degraded implementation. Iterations are classified as improvement ($S > 1.03$), regression ($S < 0.97$), or neutral. Each iteration reflects the geometric mean speedup across all AWFY benchmarks.

| Iteration State | Unenhanced Agent | Enhanced Agent |
|---|---|---|
| Improvement ($S > 1.03$) | 26 | 31 |
| Neutral ($1.03 \geq S \geq 0.97$) | 20 | 6 |
| Regression ($0.97 > S$) | 3 | 1 |

does not require any Truffle knowledge. The distribution of performance issues fixed by the agent across iterations is similar for both agents, as shown in table 7.6.

Compared with the original implementation, the proportion of Truffle-focused changes increased. This suggests that the number of performance issues in the implementation also affects the focus of both coding agents.

Table 7.6.: Distribution of fixed performance issues for the degraded implementation. To fix Truffle-focused bugs, it requires Truffle functions and annotations. Java-focused bugs do not require truffle knowledge to fix it.

| Performance issue | Unenhanced Agent | Enhanced Agent |
|---|---|---|
| Truffle-focused | 48 | 35 |
| Java-focused | 1 | 3 |

When looking at the performance improvements the coding agents achieved over iterations in figure 7.5, a clear trend emerges. Enhanced Agent achieved in more cases a higher overall performance improvement than the Unenhanced Agent. The Mann-Whitney U test ($\alpha = 0.05$) revealed a statistically significant improvement in geometric mean speedup $S$ for Enhanced Agent over Unenhanced Agent ($U = 96$, $p < 0.001$, one-tailed, $r = 0.92$, large effect, $n = 10$ per group). This means $H_0$ can be rejected.

These findings confirm that the knowledge enhancements have an impact on the overall performance improvement of the coding agent. However, this claim only holds for a degraded language implementation, because the tests were inconclusive for the original implementation. That means the enhancements are most valuable when there are clear detectable performance issues.

Additionally, the Enhanced Agent implemented fewer Truffle-focused changes, but achieved a higher overall performance. This may indicate that the additional knowledge helped the Enhanced Agent prioritize the impact of performance issues.

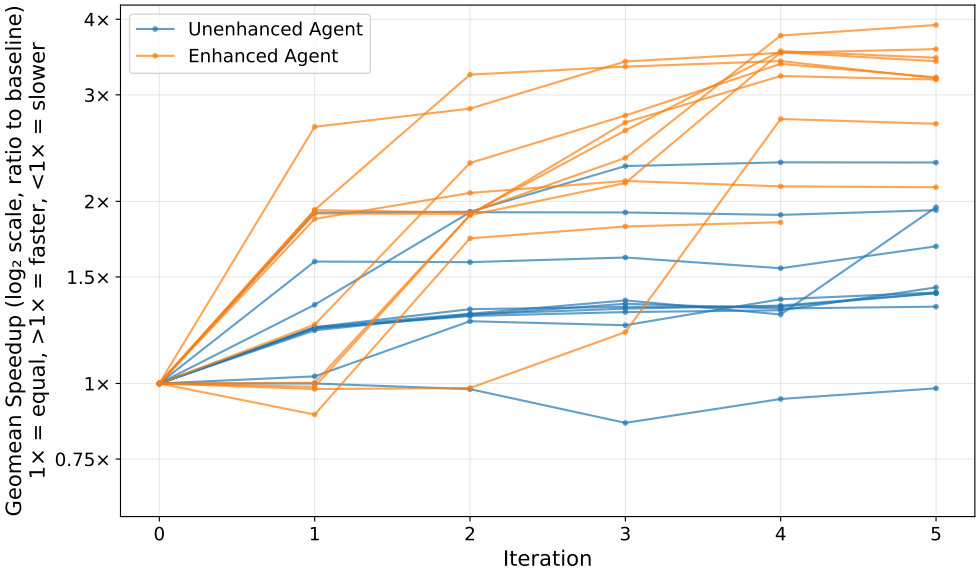



Figure 7.5.: Speedup progression per run for the degraded implementation.

Nevertheless, we want to discuss one outlier in the Enhanced Agent runs deeper. In figure 7.5 is visible, that in one run the performance decreased in the first iteration. The figure L.3 shows the individual benchmark performances over the iterations.

When looking into the concrete Claude Code session, it reveals that the first iteration ran into a timeout and got canceled, which is consistent with the revert branch in Algorithm B.2. That means the iteration should not include any

source code changes that affect the performance. However, the last executed command was a build command that never completed. Seemingly, killing the Claude Code process did not kill the build process, which still ran when the benchmarks executed.

This is not an insight into the tool, but into the evaluation design itself. Canceling the Claude Code process is not enough when it has run into a timeout. We would have to kill all other GraalVM related processes too. Despite this confound, the remaining runs still produced a statistically significant result, which suggests the finding is robust to this noise.

Returning to the total runs, comparing the final implementations with swalox reveals similar results to the original implementation. A performance optimization potential still exists, which could be reduced in further iterations though. The figure 7.6 shows that the resulting implementation of both agents is still slower than swalox. However, the enhanced agent is, on average, faster than Python. Additionally, the error size reduced for both cases, which suggests convergent behavior and more stable and repeated optimization workflows.

When comparing the API costs $C$ of coding agent sessions with the speedup in figure 7.7, it reveals slightly different results from the original implementation. The Enhanced Agent costs more than the Unenhanced Agent, but with a higher speed up. However, the Enhanced Agent costs less than on the original implementation. A possible reason is that the agent reaches a valid theory faster and needs fewer attempts. One interpretation is that cost scales with problem difficulty. When clear performance issues exist, the agent converges on a valid theory in fewer attempts.

### 7.1.3. Supplementary Cross-Project Experiment

We conducted a supplementary experiment using multiple language implementations from the BYOPL seminar. For each project, we executed a smaller version of the quantitative study setup. We ran one fully autonomous optimization iteration with the Unenhanced and Enhanced Agents. We repeated these runs three times, and figure 7.8 shows the average speedup. The projects are selected based on the number of passed unittests in the seminar.

The results show that, in most cases, the Enhanced Agent performed better than the Unenhanced Agent. However, because the sample size was very small, these results are not representative.

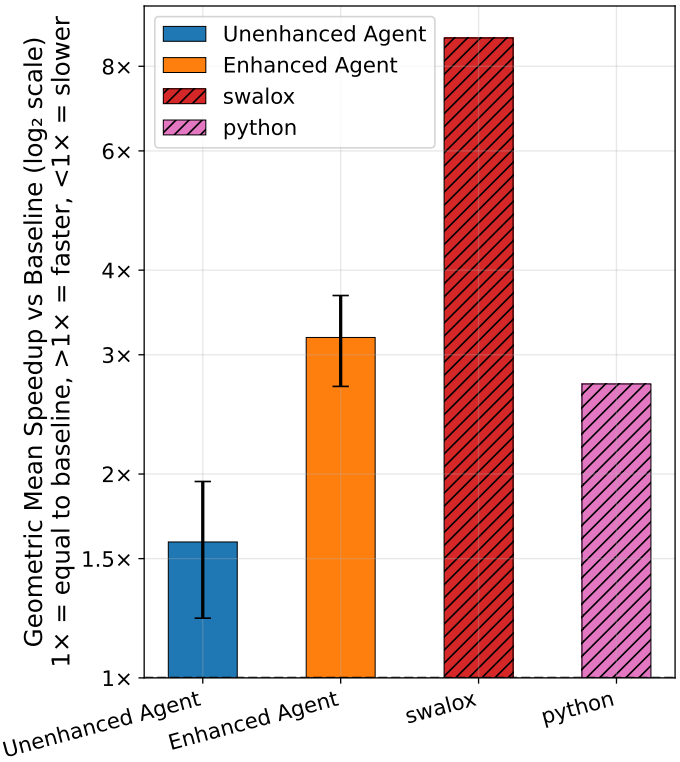



Figure 7.6.: Average geometric mean speedup for the degraded implementation compared against the swalox and CPython 3.13. The reference is the unoptimized implementation prior to any agent. Unenhanced and Enhanced Agent values represent the mean geometric mean speedup across all runs. Error bars indicate one standard deviation across runs.

## 7.2. User Study Results

This section focuses on the results of the user study. It discusses the progress of the participants first. Afterwards, it presents the derived themes of the thematic analysis of the study transcripts.

### 7.2.1. Participants Progress

When examining the language implementation with the manual setup, both participants did not find any performance issue.

When examining it with the agent coding setup, both participants identified and fixed a real performance issue. Participant A used the Full Orchestration agent skill, while Participant B used the Process Assistance level sub-agents.

This suggests that the Enhanced Agent supported the participants to improve the language implementations more reliably.

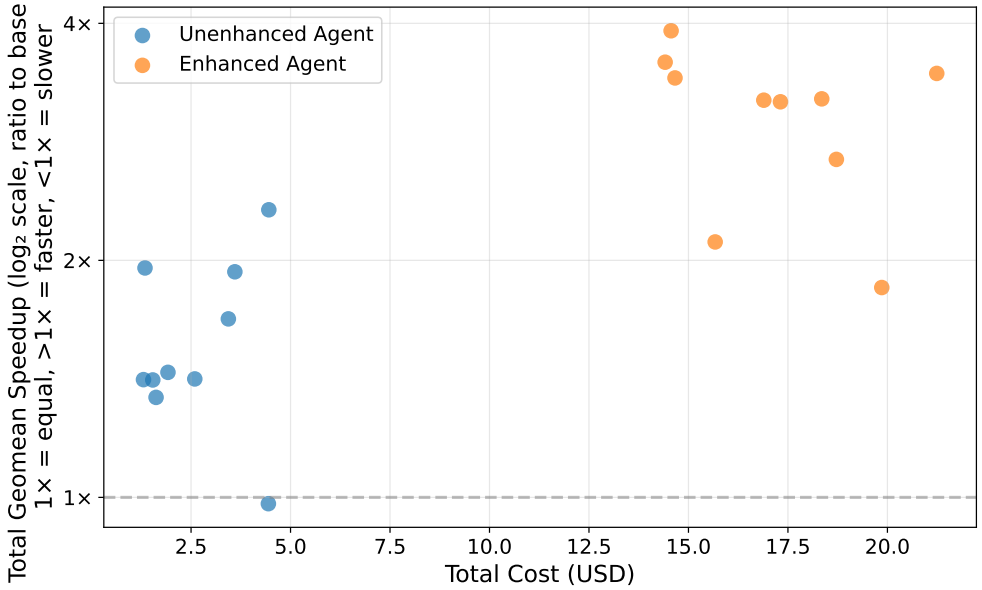



Figure 7.7.: API cost $C$ over total geometric mean speedup $S$ in relation to the baseline before optimizations per run for the degraded implementation

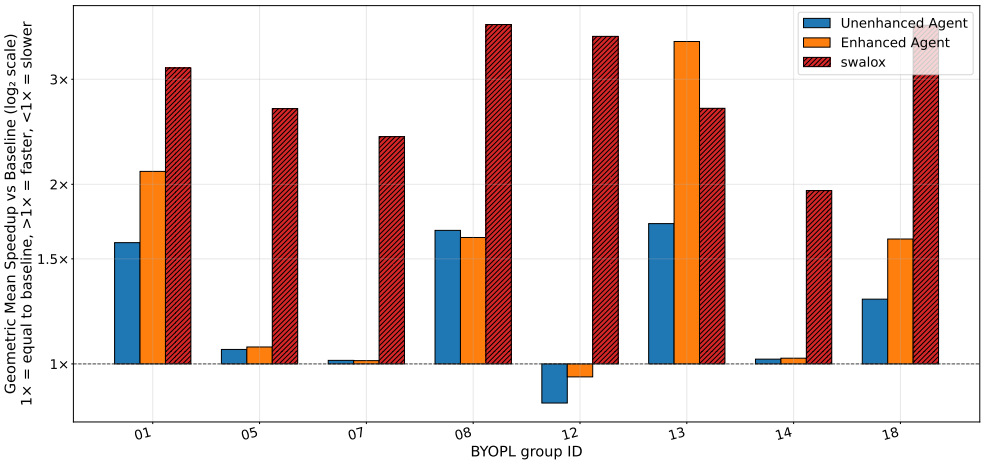



Figure 7.8.: Results of a supplementary experiment. It shows the average effect of one iteration on ten different projects from the BYOPL seminar. We repeated the experiment three times for each project. We omit error bars, because of the small sample size.

### 7.2.2. Thematic Analysis Findings

We derived three major themes from the thematic analysis. We will define each of them, connect them to a CDN dimension and give an example from the user study.

**Theme 1: Translation between developer idea and expression in tools**
This deductive theme captures the translation cost between how developers think about performance problems and how they must express those thoughts to tools. With the manual setup, participant had to translate natural reasoning into low-level CLI syntax. This required searching documentation and experimenting with parameter variations. The agent setup reduced these costs by accepting natural language. However, minor friction persists: the participant had to load the plugin manually and write down their thoughts.

The primary CDN dimension in focus was "Closeness of mapping". The natural language representation was closer to the developer's thoughts than the CLI commands.

We observed in the study one participant, who mentioned this directly. They could happily insert their thoughts directly into Claude Code instead of searching for a CLI command or flag first.[1]

**Theme 2: Interpretation Bottleneck**
This deductive theme captures the gap between having profiling data and understanding what it means for code changes. With the manual setup, tools produce rich output that participants cannot interpret without deep framework expertise. With the agent setup, its structured analysis bridges this gap, but introduces a new bottleneck. Participants cannot verify whether the agent's interpretation is correct. The theme excludes difficulties in expressing intent to use tools (Theme 1).

The primary CDN dimension in focus was "Hard mental operations". The interpretation of the output requires expert knowledge, same as the validation of the agent's interpretation.

[1] Participant B, (PS-M, DI-2, 01:10:45): "*ich konnte einfach meine Gedanken halt in Claude eintippen und musste mir nicht irgendwelche Commands raussuchen. So, und das war irgendwie halt, also jetzt so spontan, ein echt großer Unterschied.*", Translated: "I could simply type my thoughts into Claude without having to look up any specific commands. And that—thinking about it off the top of my head—really made a huge difference."

We observed one participant with the manual setup, who mentioned that they were frustrated. They know that the profiling tools are powerful, but they cannot understand it well enough.[2] With the agent setup, they mentioned the difficulties validating the agent's output.[3]

**Theme 3: From Craftsperson to Reviewer**
This inductive theme captures how the plugin transforms the developer's role from active problem-solver to passive reviewer and what that transformation feels like. It includes the identity and engagement consequences of automation, for instance loss of ownership, boredom and disengagement. The coping strategy for one participant was parallel work. However, the structural mechanisms, like process transparency, confirmation of tool execution, *Learned Lessons Memory* and code difference visualization help the reviewer role.

The theme consists mostly of inductive codes, meaning no CDN dimension was in focus here.

We observed, that one participant was deeply involved in the optimization problem with the manual setup. Their frustration was about the task difficulty instead of their role.[4] With the agent setup, the agent actually produced results, but the participants lost the engagement. They stopped identifying with the solution.[5]

[2] Participant A (PS-M, DI, 01:15:04): "*Es ist ein Guessing. Es ist ein bisschen frustrierend, dass man dann vor diesem mächtigen Tool steht und nicht so wirklich Dinge deuten kann.*", Translated: "It involves guesswork. It is a bit frustrating to stand there in front of this powerful tool and not really be able to interpret things."

[3] Participant A (PS-A, DI-2, 00:11:51): "*das ist schön und gut, dass du mir hier deine Lessons learned aufschreibst, aber ich habe keine Ahnung, ob das wirklich richtig ist oder nicht.*", Translated: "That's all well and good—you writing down your lessons learned here—but I have no idea whether it's actually correct or not."

[4] Participant B (PS-M, DI, 01:13:01): "*ich finde es schade, dass ich da jetzt nichts Großes hinbekommen habe*", Translated: "It's a shame I didn't manage to pull off anything big there."

[5] Participant A (PS-A, DI-2, 00:56:31): "*Ich identifiziere mich gar nicht mehr mit der Lösung*", Translated: "I no longer identify with the solution at all."

## 7.3. Summary

In summary, the experiment reveals a mixed picture. For both implementations, the Enhanced Agent used a more hypothesis-driven optimization approach and the ratio of critical performance fixes to completed iterations is higher. The knowledge enhancements influenced the agent's optimization strategy in the intended direction and helped prioritize the impact of performance issues.

However, for both implementations the API cost of the Enhanced Agent is significantly larger and it did not reach the full performance potential in comparison to swalox.

The final speedup depends in which shape the language implementation is already. For the well-optimized original implementation the Mann-Whitney U test found no statistically significant difference between the two agents. The benefit in terms of measurable speedup remains inconclusive for a well-optimized implementation such as this one. For the degreaded implementation however, the enhancements had a significant effect and a more consistent output overall. The enhancements help when there are obvious issues, but not measurably when the implementation is already mature.

In the user study, both participants fixed a performance issue within the time frame with the agent, which indicates a faster development process than without it.

Additionally, the costs for the translation of thoughts into concrete tool use decreased together with the costs of interpreting the results. The frustration increased, because of the developer's role switch, which is unrelated to the agent enhancement.

# 8. Related Work

This chapter discusses publications that were not already introduced in chapter 2, but are related to Truffle language performance optimization and coding agent enhancements. Each section focuses on a specific component of the thesis approach, introducing publications first and discussing the difference afterwards.

## 8.1. Performance Analysis Tools for Truffle

This section discusses publications that focus on performance analysis of Truffle languages. Gaikwad, Nisbet, and Luján explain in "Performance Analysis for Languages Hosted on the Truffle Framework" [31] how to visually analyze Truffle-hosted language performance using flamegraphs. Therefore, they used the Linux `perf` tool together with `perf-map-agent` to map sampled call-stacks back onto guest-language source code.

Niephaus et al. took a different angle in "Example-Based Live Programming for Everyone" [56] and built a language-agnostic live programming environment on top of Truffle to give developers immediate runtime feedback during development. The live programming environment used the Truffle instrumentation to obtain runtime information. The goal was to lower the effort to build runtime-feedback tools across the Truffle ecosystem.

Both works improve the interface to Truffle tooling for human experts, who already understand the framework. Neither prior work addresses the expertise gap between accessible language implementations and effective performance optimization. This thesis discusses an approach to support Truffle beginners in optimizing their language implementation without knowing the Truffle details. Therefore, we embed that expert knowledge into an AI coding agent.

## 8.2. AI Agents for Code Performance Optimization

This section discusses publications on how coding agents can be used for local code optimizations. Peng et al. propose in *Beyond Local Code Optimization* [67] a multi-agent framework that decomposes the optimization problem into four coordinated agent roles, similar to the Process Assistance level from

section 4.5. In contrast to their approach, which derives these roles from architectural signals, our roles are derived from an expert interview.

Peng et al. target a general software system with microservices and focus on the throughput instead of the runtime performance. This thesis targets the Truffle internals instead of a microservice architecture. Additionally, we collected domain-specific knowledge and extended the coding agent with it. The prior work does not address knowledge enhancements.

## 8.3. Alternative Agent Enhancement Mechanisms

This section discusses different approaches to extend a coding agent with domain-specific knowledge. Prior work has explored knowledge enhancement at two layers: model parameters and runtime tooling.

Peters et al. show in *Knowledge Enhanced Contextual Word Representations* [70] a practical example how to embed multiple knowledge bases into the model parameters. Their model, KnowBert, integrated both WordNet [52] and a subset of Wikipedia into the pre-trained language model BERT [26]. The resulting model improved fact recall while maintaining runtime comparable to the base model.

Gan and Sun propose in *RAG-MCP* [32] a retrieval-augmented approach, which embeds tool documentation into a vector store and retrieves relevant chunks at runtime. This thesis discusses RAG as an alternative to agent skills in chapter 4.

Shi et al. present AutoTools in *Tool Learning in the Wild* [77], a knowledge adapter that lets developers transform public documentation into a unified representation understandable to a LLM.

To build the representation, an agent parses a library's documentation. In a second step, it transforms it into Python functions. The generated function library is inserted into the execution environment. The coding agent is then equipped with the created function library and generates executable programs.

In contrast, this thesis proposes an approach, which does not require an open-weight model. Instead, sub-agents, agent skills and MCP servers are reusable across any MCP-compatible coding agent and can be updated independently of the underlying model. KnowBert requires weight access, which is inapplicable when using leading closed-source models like Claude Sonnet 4.5. AutoTools is not easily integrable into existing AI coding agents, as it is designed as a self-contained research pipeline rather than a reusable developer tool.

# 9. Conclusion

## 9.1. Research Summary

This research focused on the expertise gap when domain developers with less Truffle knowledge wanted to optimize their DSLs. These developers lacked the knowledge about underlying concepts of the Truffle framework to identify performance flaws with the integrated profiling tools. They needed support from Truffle experts to execute the tools and interpret their output correctly.

This research examined whether an enhanced AI coding agent could close the gap. Therefore, we compared the agent in an experiment with an unenhanced agent and in a user study with the unguided manual profiling process.

We developed the approach in three steps. First, we synthesized expert knowledge from a contextual inquiry interview with a Truffle expert, Dr. Tim Felgentreff, discussed in chapter 3. Afterwards, we introduced in chapter 4 a layered knowledge adapter design to support the domain developers with different granularity: Tool Assistance, Process Assistance or Full Orchestration. We implemented it as a Claude Code plugin in chapter 5 and conducted an experiment and a user study chapter 6 and 7.

In a degraded language implementation, the enhanced agent showed a statistically significantly higher speedup than an unenhanced agent ($U = 96$, $p < 0.001$, $r = 0.92$). For an already well-optimized implementation the results were inconclusive ($U = 62$, $p = 0.192$, $r = 0.24$). In both cases, the enhanced agent produced higher API costs. However, both participants resolved a performance issue with the enhanced agent, while neither completed one with the manual process. Additionally, the costs for the translation of thoughts into concrete tool use decreased together with the costs of interpreting the results.

## 9.2. Contribution

This thesis includes different kinds of contributions. The following list and figure 9.1 give an overview.

Optimization Workflow We conducted a contextual inquiry interview and reconstructed in chapter 3 an optimization workflow for Truffle languages. It includes the steps baseline generation, static theory generation, tool-assisted verification, fix and iterate.

Truffle Tools We collected resources about Truffle profiling and optimization tools, like CPU Sampler, Trace Compilation, Trace Transfer to Interpreter, Compiler Graph Dumping, and IGV.

Agent-agnostic Enhancements We synthesized the optimization workflow and tool knowledge in agent skills, sub-agent definitions, and tool extensions in chapter 4. It includes one agent skill for each profiling tool, one sub-agent per workflow phase, one full orchestration skill. The enhancements are designed to be portable to other coding agents.

Claude Code Plugin We bundled the enhancements in a Claude Code plugin `cc-truffle-performance-plugin`,[1] described in chapter 5.

Experiments We evaluated this layered architecture in a controlled experiment with $n = 10$ runs $\times$ 5 iterations $\times$ 2 implementations $\times$ 2 conditions in chapter 6 and 7.

User Study We also conducted a user study with two participants from the BYOPL seminar and derived insights with a thematic analysis in chapter 6 and 7.

## 9.3. Implications for Practice

The findings carry several implications for practice. For language developers, the enhanced agent compensates for missing Truffle expertise, especially on implementations with clear performance issues. The benefit on already mature implementations is measurably limited. The enhanced agent follows a more hypothesis-driven and Truffle-focused approach.

The user study suggests that the mental cost for translating thoughts into tool executions decreases with the enhanced agent. But the verification of the agent's interpretations and code changes is still hard for non-experts.

However, the API costs increase in comparison to an unenhanced agent. This is the case for both mature and degraded implementations, where the cost scales with problem difficulty.

[1] `https://github.com/hpi-swa-lab/cc-truffle-performance-plugin`

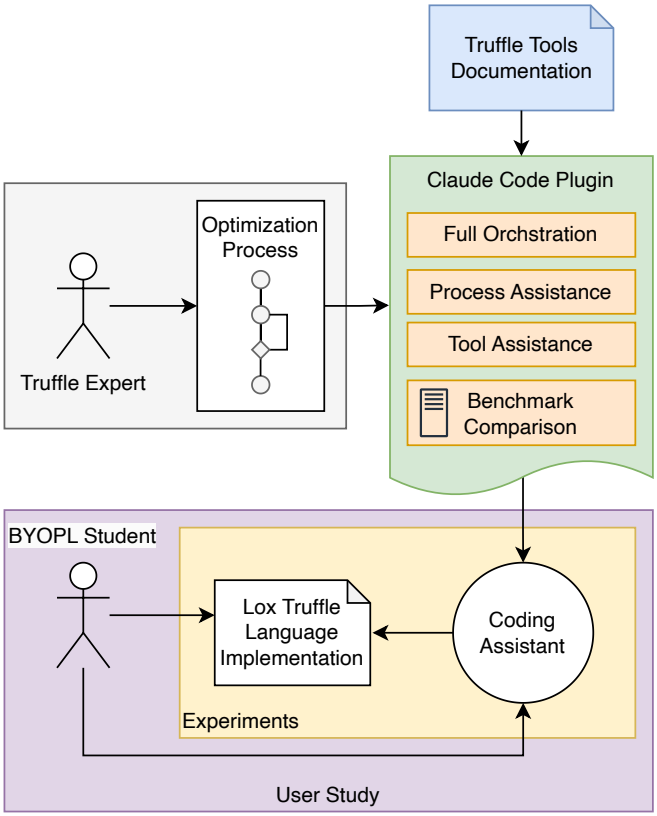



Figure 9.1.: Overview of contributions. Each colored item represents an individual contribution.

For coding agent developers, this research introduces a layered approach to enhance an AI agent with a closed-weight model with domain knowledge. It includes considerations when choosing an enhancement technique, such as using sub-agents to reduce context pollution.

## 9.4. Future Research Directions

Several directions for future research emerge from this work. Researchers could extend the tool catalogue in chapter 3 and apply the layered architecture to other coding agents, like Codex, Copilot, or OpenCode. OpenCode would enable a comparison with open-weight models.

Another potential research direction is the reduction of the sub-agent token overhead. The context isolation results in redundant codebase reloading in the sub-agents. Future work could evaluate different approaches, for instance introducing a shared context, by splitting a conversation instead of using a

sub-agent. Alternatively, only a selection of the codebase could be passed to the sub-agent to reduce the potential context size.

Other research directions focus on the validity threats. Researchers could repeat the user study with a larger participant group. They could also repeat the experiment and user study with other Truffle implementations beyond Lox. Alternatively, they could repeat the experiment with the original implementation using a different prompt strategy or longer iterations, because the results were inconclusive. Additionally, the evaluation can be repeated with the AST interpreter instead of the Bytecode interpreter and AOT instead of JIT compilation.

Another potential direction focuses on the interpretation bottleneck, derived in the thematic analysis in chapter 7. Future research could develop verifiable evidence mechanisms for non-experts to review the agent's output.

Another direction is changing the degree of freedom in the decisions during the optimization process. Stricter or looser descriptions of the optimization process can save token costs and affect the optimization results.

Further global developments during the work on this thesis reveal more directions. Newer models with larger context windows, for instance Claude Fable 5 [7] or Claude Opus 4.8 [8], allow working on more complex solutions without any enhancements. Comparing their abilities with a smaller model and the developed plugin can reveal further insights.

Each of these directions brings domain developers one step closer to autonomous Truffle optimization.

# Bibliography


[1] *2025 Stack Overflow Developer Survey*. URL: https://survey.stackoverflow.co/2025/ (visited on 2026-01-08).

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Reprinted, with corr., [36. Druck]. Addison-Wesley Series in Computer Science. Reading, MA, USA: Addison-Wesley, 2002. 796 pages. ISBN: 978-0-201-10088-4.

[3] Anomaly. *OpenCode*. URL: https://opencode.ai (visited on 2026-01-29).

[4] Anthropic. *Agent Skills Specification*. Technical Specification. Anthropic, 2025.

[5] Anthropic. *Claude Code Cangelog v2.0.12*. Claude Code Docs. URL: https://code.claude.com/docs/en/changelog (visited on 2026-04-03).

[6] Anthropic. *Claude Code Overview*. Claude Code Docs. URL: https://code.claude.com/docs/en/overview (visited on 2026-04-07).

[7] Anthropic. *Claude Fable 5*. URL: https://www.anthropic.com/claude/fable (visited on 2026-07-07).

[8] Anthropic. *Claude Opus 4.8*. URL: https://www.anthropic.com/claude/opus (visited on 2026-07-07).

[9] Anthropic. *Custom Subagents*. URL: https://code.claude.com/docs/en/sub-agents (visited on 2026-03-17).

[10] Anthropic. *How Claude Remembers Your Project*. URL: https://code.claude.com/docs/en/memory (visited on 2026-03-15).

[11] Anthropic. *Introducing Claude Sonnet 4.5*. URL: https://www.anthropic.com/news/claude-sonnet-4-5 (visited on 2026-01-29).

[12] Anthropic. *Messages API*. URL: https://platform.claude.com/docs/en/api/messages (visited on 2026-03-10).

[13] Anthropic. *Model Context Protocol Specification*. Technical Specification 2025-11-25. Anthropic, Nov. 2025.

[14] Anthropic. *Pricing*. URL: https://platform.claude.com/docs/en/about-claude/pricing (visited on 2026-03-10).

[15] *Apple/Pkl*. Version 0.30.2. Apple Inc., Dec. 15, 2025.

[16] K. Baxter, C. Courage, and K. Caine. "During Your User Research Activity". In: *Understanding Your Users*. Elsevier, 2015, pages 158–189. ISBN: 978-0-12-800232-2. DOI: 10.1016/B978-0-12-800232-2.00007-9.

[17] H. Beyer and K. Holtzblatt. *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufmann, 1998.

[18] A. F. Blackwell, C. Britton, A. Cox, T. R. G. Green, C. Gurr, G. Kadoda, M. S. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, and R. M. Young. "Cognitive Dimensions of Notations: Design Tools for Cognitive Technology". In: *Cognitive Technology: Instruments of Mind*. Edited by M. Beynon, C. L. Nehaniv, and K. Dautenhahn. Volume 2117. Berlin, Heidelberg: Springer, 2001, pages 325–341. ISBN: 978-3-540-42406-2 978-3-540-44617-0. DOI: 10.1007/3-540-44617-6_31.

[19] V. Braun and V. Clarke. "Using Thematic Analysis in Psychology". In: *Qualitative Research in Psychology* 3.2 (Jan. 2006), pages 77–101. ISSN: 1478-0887, 1478-0895. DOI: 10.1191/1478088706qp063oa.

[20] *ChatGPT*. URL: https://chatgpt.com/ (visited on 2026-01-10).

[21] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. *Evaluating Large Language Models Trained on Code*. July 14, 2021. DOI: 10.48550/arXiv.2107.03374. arXiv: 2107.03374 [cs]. URL: http://arxiv.org/abs/2107.03374 (visited on 2026-01-08). Pre-published.

[22] *Claude*. URL: https://claude.ai/new (visited on 2026-01-10).

[23] *Claude 3.7 Sonnet and Claude Code*. URL: https://www.anthropic.com/news/claude-3-7-sonnet (visited on 2026-01-08).

[24] T. D. Cook and D. T. Campbell. *Quasi-Experimentation: Design & Analysis Issues for Field Settings*. Boston, MA, USA: Houghton Mifflin, 1979. 405 pages. ISBN: 978-0-395-30790-8.

[25] *Cursor: The Best Coding Agent*. Cursor. June 18, 2026. URL: https://cursor.com/ (visited on 2026-06-23).

[26] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. May 24, 2019. DOI: 10.48550/arXiv.1810.04805. arXiv: 1810.04805 [cs]. URL: http://arxiv.org/abs/1810.04805 (visited on 2026-03-15). Pre-published.

[27] L. Eeckhout. "R.I.P. Geomean Speedup Use Equal-Work (Or Equal-Time) Harmonic Mean Speedup Instead". In: *IEEE Computer Architecture Letters* 23.1 (Jan. 2024), pages 78–82. ISSN: 1556-6056, 1556-6064, 2473-2575. DOI: 10.1109/LCA.2024.3361925.

[28] T. Felgentreff. *Contextual Inquiry Interview about Graal and Truffle*. Potsdam, Germany, Dec. 16, 2026.

[29] P. J. Fleming and J. J. Wallace. "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results". In: *Communications of the ACM* 29.3 (Mar. 1986), pages 218–221. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/5666.5673.

[30] M. Fowler. *Domain-Specific Languages*. The Addison-Wesley Signature Series. Upper Saddle River, NJ, USA: Addison-Wesley, 2011. 1 page. ISBN: 978-0-321-71294-3 978-0-13-210754-9.

[31] S. Gaikwad, A. Nisbet, and M. Luján. "Performance Analysis for Languages Hosted on the Truffle Framework". In: *Proceedings of the 15th International Conference on Managed Languages & Runtimes*. ManLang '18. New York, NY, USA: ACM, Sept. 12, 2018, pages 1–12. ISBN: 978-1-4503-6424-9. DOI: 10.1145/3237009.3237019.

[32] T. Gan and Q. Sun. *RAG-MCP: Mitigating Prompt Bloat in LLM Tool Selection via Retrieval-Augmented Generation*. May 6, 2025. DOI: 10.48550/arXiv.2505.03275. arXiv: 2505.03275 [cs]. URL: http://arxiv.org/abs/2505.03275 (visited on 2026-03-17). Pre-published.

[33] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang. *Retrieval-Augmented Generation for Large Language Models: A Survey*. Mar. 27, 2024. DOI: 10.48550/arXiv.2312.10997. arXiv: 2312.10997 [cs]. URL: http://arxiv.org/abs/2312.10997 (visited on 2026-03-15). Pre-published.

[34] I. Gim, G. Chen, S.-s. Lee, N. Sarda, A. Khandelwal, and L. Zhong. *Prompt Cache: Modular Attention Reuse for Low-Latency Inference*. Apr. 25, 2024. DOI: 10.48550/arXiv.2311.04934. arXiv: 2311.04934 [cs]. URL: http://arxiv.org/abs/2311.04934 (visited on 2026-03-12). Pre-published.

[35] *Google Antigravity*. Google Antigravity. URL: https://antigravity.google/ (visited on 2026-06-23).

[36] *Google Gemini*. URL: https://gemini.google.com (visited on 2026-01-10).

[37] GraalVM. *CPUSampler*. URL: https://www.graalvm.org/tools/javadoc/com/oracle/truffle/tools/profiler/CPUSampler.html (visited on 2026-01-20).

[38] GraalVM. *Ideal Graph Visualizer*. URL: https://www.graalvm.org/latest/tools/igv/ (visited on 2026-01-08).

[39] *GraalVM Use Cases*. URL: https://www.graalvm.org/use-cases/ (visited on 2026-01-10).

[40] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. Jan. 26, 2024. DOI: 10.48550/arXiv.2401.14196. arXiv: 2401.14196 [cs]. URL: http://arxiv.org/abs/2401.14196 (visited on 2026-01-29). Pre-published.

[41] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* Nov. 11, 2024. DOI: 10.48550/arXiv.2310.06770. arXiv: 2310.06770 [cs]. URL: http://arxiv.org/abs/2310.06770 (visited on 2026-01-29). Pre-published.

[42] *Junie, the AI Coding Agent by JetBrains*. JetBrains. URL: https://www.jetbrains.com/junie/ (visited on 2026-06-23).

[43] A. Kamp and O. Heß. *Hpi-Swa-Teaching/Byopl-02*. Version 1.1.0. Berlin, Germany: HPI Software Architecture Group Lab, 2025.

[44] R. Krahn. *Podwriter.Io: Video, Audio Und Podcast Transkription*.

[45] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. Apr. 12, 2021. DOI: 10.48550/arXiv.2005.11401. arXiv: 2005.11401 [cs]. URL: http://arxiv.org/abs/2005.11401 (visited on 2026-01-10). Pre-published.

[46] LF Projects LLC. *AGENTS.Md*. URL: https://agents.md/ (visited on 2026-03-15).

[47] J. Lincke. *Hpi-Swa-Lab/Swalox*. Version 1.0.0. Berlin, Germany: HPI Software Architecture Group Lab, 2025.

[48] J. Lincke, T. Felgentreff, F. Niephaus, and R. Hirschfeld. *Build Your Own Programming Language*. Seminar, Hasso Plattner Institute, University of Potsdam. 2024.

[49] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang. *Lost in the Middle: How Language Models Use Long Contexts*. Nov. 20, 2023. DOI: 10.48550/arXiv.2307.03172. arXiv: 2307.03172 [cs]. URL: http://arxiv.org/abs/2307.03172 (visited on 2026-02-03). Pre-published.

[50] S. Marr, B. Daloze, and H. Mössenböck. "Cross-Language Compiler Benchmarking: Are We Fast Yet?" In: *Proceedings of the 12th Symposium on Dynamic Languages*. SPLASH '16: Conference on Systems, Programming, Languages, and Applications: Software for Humanity. Amsterdam, Netherlands: ACM, Nov. 2016, pages 120–131. ISBN: 978-1-4503-4445-6. DOI: 10.1145/2989225.2989232.

[51] Microsoft. *GitHub Copilot*. GitHub. 2026. URL: https://github.com/features/copilot (visited on 2026-04-07).

[52] G. A. Miller. "WordNet: A Lexical Database for English". In: *Communications of the ACM* 38.11 (Nov. 1, 1995), pages 39–41. ISSN: 0001-0782. DOI: 10.1145/219717.219748.

[53] *Mitmproxy - an Interactive HTTPS Proxy*. URL: https://www.mitmproxy.org/ (visited on 2026-03-10).

[54] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian. *A Comprehensive Overview of Large Language Models*. Oct. 17, 2024. DOI: 10.48550/arXiv.2307.06435. arXiv: 2307.06435 [cs]. URL: http://arxiv.org/abs/2307.06435 (visited on 2026-01-22). Pre-published.

[55] J. Nielsen. "Enhancing the Explanatory Power of Usability Heuristics". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI94: ACM Conference on Human Factors in Computer Systems. Boston, MA, USA: ACM, Apr. 24, 1994, pages 152–158. ISBN: 978-0-89791-650-9. DOI: 10.1145/191666.191729.

[56] F. Niephaus, P. Rein, J. Edding, J. Hering, B. König, K. Opahle, N. Scordialo, and R. Hirschfeld. "Example-Based Live Programming for Everyone: Building Language-Agnostic Tools for Live Programming with LSP and GraalVM". In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. SPLASH '20: Conference on Systems, Programming, Languages, and Applications, Software for Humanity. Virtual, USA: ACM, Nov. 18, 2020, pages 1–17. ISBN: 978-1-4503-8178-9. DOI: 10.1145/3426428.3426919.

[57] B. Nystrom. *Munificent/Craftinginterpreters*. Aug. 2, 2024.

[58] R. Nystrom. *Crafting Interpreters*. 2021. 1 page. ISBN: 978-0-9905829-3-9.

[59] OBS Project. *OBS Studio*. Version 32.0.4. Dec. 13, 2025.

[60] OpenAI. *ChatGPT Codex*. URL: https://openai.com/codex/ (visited on 2026-01-08).

[61] OpenAI. *Subagents*. URL: https://developers.openai.com/codex/subagents/ (visited on 2026-03-17).

[62] Oracle. *CompilerDirectives.TruffleBoundary (GraalVM Truffle Java API Reference)*. URL: https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/CompilerDirectives.TruffleBoundary.html (visited on 2026-03-24).

[63] Oracle. *Graalvm/Graalvm-Ce-Builds*. Version 21.0.2. 2024.

[64] Oracle. *Optimizing Truffle Interpreters*. URL: https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/Optimizing/ (visited on 2026-03-24).

[65] Oracle. *Profiling Command Line Tools*. URL: https://www.graalvm.org/22.3/tools/profiling/ (visited on 2026-03-24).

[66] *Oracle/Graalpython*. Version 25.0.1. Oct. 21, 2025.

[67] H. Peng, P. V. Patil, A. Z. Qiu, G. K. Thiruvathukal, and J. C. Davis. *Beyond Local Code Optimization: Multi-Agent Reasoning for Software System Optimization*. Mar. 16, 2026. DOI: 10.48550/arXiv.2603.14703. arXiv: 2603.14703 [cs]. URL: http://arxiv.org/abs/2603.14703 (visited on 2026-04-30). Pre-published.

[68] H. Peng, A. Zhong, R. A. C. Méndez, K. G. Kalu, and J. C. Davis. *How Do Agents Perform Code Optimization? An Empirical Study*. Version 1. 2025. DOI: 10.48550/arXiv.2512.21757. arXiv: 2512.21757 [cs]. URL: http://arxiv.org/abs/2512.21757 (visited on 2026-01-09). Pre-published.

[69] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva. "Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate?" In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. Sle 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pages 256–267. ISBN: 978-1-4503-5525-4. DOI: 10.1145/3136014.3136031.

[70] M. E. Peters, M. Neumann, R. L. Logan, R. Schwartz, V. Joshi, S. Singh, and N. A. Smith. *Knowledge Enhanced Contextual Word Representations*. Oct. 31, 2019. DOI: 10.48550/arXiv.1909.04164. arXiv: 1909.04164 [cs]. URL: http://arxiv.org/abs/1909.04164 (visited on 2026-01-10). Pre-published.

[71] *Profiling Command Line Tools*. URL: https://www.graalvm.org/latest/tools/profiling/ (visited on 2026-01-08).

[72] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, and M. Sun. *ChatDev: Communicative Agents for Software Development*. June 5, 2024. DOI: 10.48550/arXiv.2307.07924. arXiv: 2307.07924 [cs]. URL: http://arxiv.org/abs/2307.07924 (visited on 2026-03-31). Pre-published.

[73] R. Rampin. *Taguette*. 2018. URL: https://www.taguette.org/ (visited on 2026-04-10).

[74] S. Robertson and H. Zaragoza. "The Probabilistic Relevance Framework: BM25 and Beyond". In: *Foundations and Trends in Information Retrieval* 3.4 (2009), pages 333–389. ISSN: 1554-0669, 1554-0677. DOI: 10.1561/1500000019.

[75] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. *Toolformer: Language Models Can Teach Themselves to Use Tools*. Feb. 9, 2023. DOI: 10.48550/arXiv.2302.04761. arXiv: 2302.04761 [cs]. URL: http://arxiv.org/abs/2302.04761 (visited on 2026-01-10). Pre-published.

[76] W. R. Shadish, T. D. Cook, and D. T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Nachdr. Belmont, CA, USA: Wadsworth Cengage Learning, 2002. 623 pages. ISBN: 978-0-395-61556-0.

[77] Z. Shi, S. Gao, L. Yan, Y. Feng, X. Chen, Z. Chen, D. Yin, S. Verberne, and Z. Ren. *Tool Learning in the Wild: Empowering Language Models as Automatic Tool Agents*. Mar. 4, 2025. DOI: 10.48550/arXiv.2405.16533. arXiv: 2405.16533 [cs]. URL: http://arxiv.org/abs/2405.16533 (visited on 2026-01-10). Pre-published.

[78] Shopify. *Seafoam*. Shopify, Mar. 21, 2026.

[79] Z. Song, B. Yan, Y. Liu, M. Fang, M. Li, R. Yan, and X. Chen. *Injecting Domain-Specific Knowledge into Large Language Models: A Comprehensive Survey*. Version 2. 2025. DOI: 10.48550/arXiv.2502.10708. arXiv: 2502.10708 [cs]. URL: http://arxiv.org/abs/2502.10708 (visited on 2026-01-10). Pre-published.

[80] E. Strubell, A. Ganesh, and A. McCallum. "Energy and Policy Considerations for Deep Learning in NLP". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. Florence, Italy: Association for Computational Linguistics, 2019, pages 3645–3650. DOI: 10.18653/v1/P19-1355.

[81] Z. R. Tam, C.-K. Wu, Y.-L. Tsai, C.-Y. Lin, H.-y. Lee, and Y.-N. Chen. "Let Me Speak Freely? A Study On The Impact Of Format Restrictions On Large Language Model Performance." In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*. Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track. Miami, FL, USA: Association for Computational Linguistics, 2024, pages 1218–1236. DOI: 10.18653/v1/2024.emnlp-industry.91.

[82] M. Van De Vanter, C. Seaton, M. Haupt, C. Humer, and T. Würthinger. "Fast, Flexible, Polyglot Instrumentation Support for Debuggers and Other Tools". In: *The Art, Science, and Engineering of Programming* 2.3 (Mar. 29, 2018), page 14. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2018/2/14.

[83] A. Van Deursen, P. Klint, and J. Visser. "Domain-Specific Languages: An Annotated Bibliography". In: *ACM SIGPLAN Notices* 35.6 (June 2000), pages 26–36. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/352029.352035.

[84] W. Wang, J. Min, and W. Zou. *Intelligence Degradation in Long-Context LLMs: Critical Threshold Determination via Natural Length Distribution Analysis*. Jan. 7, 2026. DOI: 10.48550/arXiv.2601.15300. arXiv: 2601.15300 [cs]. URL: http://arxiv.org/abs/2601.15300 (visited on 2026-02-03). Pre-published.

[85] X. Wang, Z. Chen, Z. Xie, T. Xu, Y. He, and E. Chen. *In-Context Former: Lightning-fast Compressing Context for Large Language Model*. Nov. 5, 2024. DOI: 10.48550/arXiv.2406.13618. arXiv: 2406.13618 [cs]. URL: http://arxiv.org/abs/2406.13618 (visited on 2026-02-03). Pre-published.

[86] M. Watanabe, H. Li, Y. Kashiwa, B. Reid, H. Iida, and A. E. Hassan. *On the Use of Agentic Coding: An Empirical Study of Pull Requests on GitHub*. Version 2. 2025. DOI: 10.48550/arXiv.2509.14745. arXiv: 2509.14745 [cs]. URL: http://arxiv.org/abs/2509.14745 (visited on 2026-01-09). Pre-published.

[87] *Which AI Coding Tools Do Developers Actually Use at Work?* The JetBrains Blog. Apr. 2, 2026. URL: https://blog.jetbrains.com/research/2026/04/which-ai-coding-tools-do-developers-actually-use-at-work/ (visited on 2026-06-23).

[88] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. "Practical Partial Evaluation for High-Performance Dynamic Language Runtimes". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '17: ACM SIGPLAN Conference on Programming Language Design and Implementation. Barcelona, Spain: ACM, June 14,

2017, pages 662–676. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062381.

[89] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. "One VM to Rule Them All". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. SPLASH '13: Conference on Systems, Programming, and Applications: Software for Humanity. Indianapolis, IN, USA: ACM, Oct. 29, 2013, pages 187–204. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509581.

[90] G. Zhang, Y. Yue, Z. Li, S. Yun, G. Wan, K. Wang, D. Cheng, J. X. Yu, and T. Chen. *Cut the Crap: An Economical Communication Pipeline for LLM-based Multi-Agent Systems*. Oct. 3, 2024. DOI: 10.48550/arXiv.2410.02506. arXiv: 2410.02506 [cs]. URL: http://arxiv.org/abs/2410.02506 (visited on 2026-03-17). Pre-published.

# Appendix A.

# Are We Fast Yet Configurations

Table A.1.: Configuration of outer and inner iterations of the AWFY benchmarks in *Baseline Establisher* Sub-Agent. Default configuration applies to all other not mentioned benchmarks.

| **Benchmark** | **Outer iteration** | **Inner iteration** |
|---|---|---|
| permute | 20 | 10000 |
| queens | 20 | 3000 |
| sieve | 20 | 10000 |
| *default* | 20 | 100 |

Table A.2.: onfiguration of outer and inner iterations of the AWFY benchmarks in Evaluation.

| **Benchmark** | **Outer iteration** | **Inner iteration** |
|---|---|---|
| bounce | 20 | 100 |
| list | 20 | 100 |
| queens | 20 | 30000 |
| sieve | 20 | 10000 |
| towers | 20 | 100 |
| storage | 20 | 100 |
| permute | 20 | 100 |

# Appendix B.

# Evaluation Experiment Algorithms

**Algorithm B.1** Experiment procedure

```
1: procedure Experiment
2:     for impl ∈ {OI, DI} do
3:         for cond ∈ {Unenhanced Agent, Enhanced Agent} do
4:             for run ← 1 to 10 do
5:                 ExecuteRun(impl, cond)
6:                 remove Claude Code session memory
```

**Algorithm B.2** Single run execution

```
1: procedure ExecuteRun(impl, cond)
2:     baseline ← BuildAndBenchmark(impl)
3:     for iter ← 1 to 5 do                          ▷ max 4 h per run
4:         snapshot ← current source state
5:         invoke agent with cond prompt              ▷ max 1.5 h
6:         if API error then
7:             revert to snapshot
8:             abort experiment
9:         else if timeout then
10:            revert to snapshot
11:        else
12:            result ← BuildAndBenchmark(impl)
13:            if result failed then
14:                revert to snapshot
15:            else
16:                record result
```

# Appendix C.

# Claude Session Fields

Table C.1.: Relevant content types in the Claude Messages API.

| Type | Role | Description |
|---|---|---|
| `text` | both | Plain natural language entered by the user or generated by the assistant. |
| `thinking` | assistant | Internal reasoning trace of the assistant. Visible to the user but does not contain final information. |
| `tool_use` | assistant | Tool invocation with tool name and input arguments. Associated with a `tool_use_id`. |
| `tool_result` | user | Output returned after executing a tool call. References a `tool_use_id` to link the result with its invocation. |

Table C.2.: Token usage fields returned in the Claude Message API response metadata.

| Field | Description |
|---|---|
| `input_tokens` | Number of tokens in the input prompt sent to the API and not yet cached. |
| `output_tokens` | Number of tokens generated by the LLM. |
| `cache_creation_input_tokens` | Number of tokens written to the cache on this request. |
| `cache_read_input_tokens` | Number of tokens read from the cache. |

# Appendix D.

# Cognitive Dimensions of Notations

Table D.1.: Overview of all CDN dimensions.

| **Dimension** | **Definition** |
|---|---|
| Viscosity | How much effort is needed to make a change. |
| Visibility | How much of the relevant information is visible at once. |
| Premature Commitment | Whether the user is forced to make decisions before they have enough information. |
| Hidden Dependencies | Whether changes in one place silently affect other places. |
| Role-Expressiveness | How clearly each part communicates what it is for. |
| Error-Proneness | How easily the notation leads the user into making mistakes. |
| Abstraction Gradient | How easily the user can move between different levels of detail. |
| Secondary Notation | Whether the user can add meaning beyond the formal syntax, for example through naming or layout. |
| Closeness of Mapping | How closely the notation matches the way the user thinks about the problem. |
| Consistency | Whether similar things look and behave similarly throughout the notation. |
| Diffuseness | How much space or text is needed to express an idea. |
| Hard Mental Operations | How much mental effort is required to read or work with the notation. |
| Progressive Evaluation | Whether the user can check their work at any point, without having to finish first. |

# Appendix E.

# Interview Script

# Interview Script

Antony Kamp

December 2025

## 1 Introduction

Welcome to this Interview. Before we get started please read through the following points carefully.

- The interview is planned to take no longer than 60 minutes. It can finish earlier if all tasks are completed.
- You may withdraw from the interview at any point.
- You may also take a break at any point.
- During the study we would kindly ask you to think out loud as much as possible.
- The interview is held in German
- The interview happens in the regular working environment of the interviewee

### Conductor Tasks

The following tasks need to be completed by the study conductor before continuing.

- Handout the consent form and answer any questions that may arise.
- Get the form signed and start the recordings
- Ask if the participant still wants to be interviewed

## Consent

I agree to participate in this interview conducted by *Antony Kamp* as part of their Master's Thesis at HPI. I understand that during this interview, audio and the computer screen will be recorded for research purposes. I acknowledge that my participation is voluntary, and I have the right to withdraw at any time without penalty. I understand that my data will be kept confidential and used only for research purposes. All my questions related to the collection, processing, and publishing of data in this study have been answered. By signing below, I consent to the recording of audio and my actions on the computer screen for the duration of the study

_____

Place, Date & Signature

100

## Introduction of Participant

Start the computer screen and audio recording after the consent is given.
The interviewee's task is to introduce themselves, especially their experience with GraalVM, Truffle and Truffle Languages.

## Code Review

The interviewee task is to download and set up the project "byopl24-02"[1] from group 2 of the seminar "Build Your Own Programming Language" in the winter semester 2024/2025. It contains a Truffle language implementation of the "Lox" language. Afterwards, they are asked to get familiar with the codebase. They have 20 minutes.
The conductor collects follow-up questions and reminds the participant to speak their thoughts out loud if they remain silent for over 2 minutes: "Please keep talking through your thought process."

## Profiling and Analysis Tools

The interviewee task is to use Graal profiling and analysis tools to diagnose a performance issue. The interviewee is asked to explain why they selected which tool and how they interpreted the tool output. They have 30 minutes.
The conductor collects follow-up questions and reminds the participant to speak their thoughts out loud if they remain silent for over 2 minutes: "Please keep talking through your thought process."

## Follow-up question

The conductor asks the follow-up question, he collected during the interview. Afterwards, the conductor tanks the interviewee for making time to participate in this interview.

101

[1] https://github.com/hpi-swa-teaching/byopl24-02

# Appendix F.

# Claude Code Plugin

Listing F.1: Configuration of the `mcp-awfy` MCP server in `.mcp.json`.

```
{
  "mcpServers": {
    "awfy": {
      "command": "uv",
      "args": [
        "run",
        "--directory",
        "${CLAUDE_PLUGIN_ROOT}/mcp-awfy",
        "mcp-awfy"
      ]
    }
  }
}
```

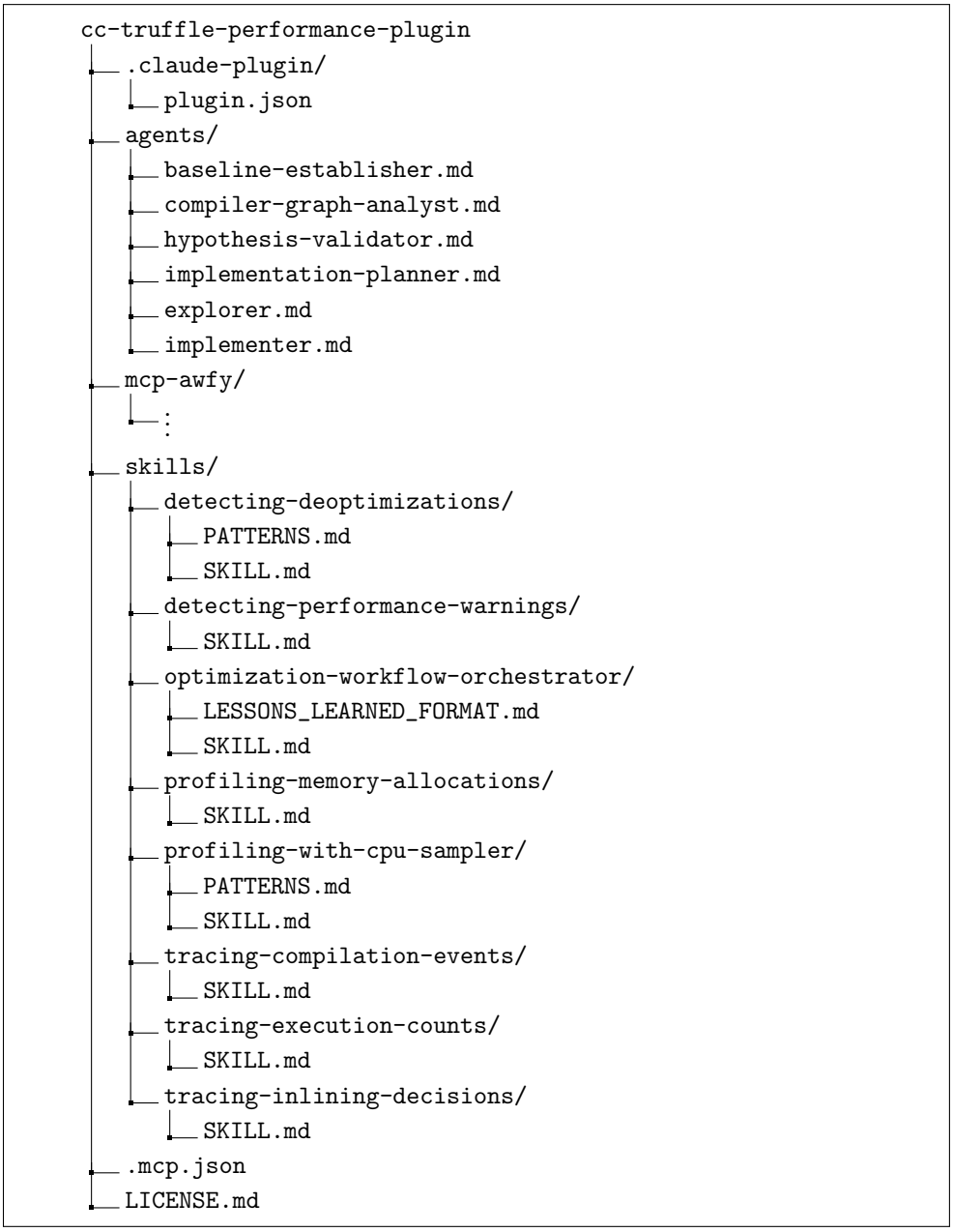



Figure F.1.: Full Claude Code Truffle Performance plugin folder structure

# Appendix G.

# Agent Skill "Profiling with CPU Sampler"

Listing G.1: Agent Skill "Profiling with CPU Sampler" from `cc-truffle-performance-plugin` Claude Code plugin

````
---
name: profiling-with-cpu-sampler
description: Time-based sampling profiler showing WHERE execution time
     is spent (wall-clock time, not frequency). Provides histogram
    with self/total time, compilation tiers (T0/T1/T2), and flame
    graphs. Use as FIRST step to identify hot functions consuming most
     time. Low overhead, suitable for longer runs. Pair with tracing-
    execution-counts to understand time-per-execution vs execution
    frequency.
---

# Profiling with CPU Sampler

Time-based sampling profiler that identifies WHERE your program spends
     execution time. Shows wall-clock time distribution across
    functions with compilation tier breakdown.

## When to Use This Skill

**Use FIRST** when investigating any performance issue:

- Identify hot functions consuming most time
- Verify code is compiling (tier distribution)
- Measure time spent in interpreter vs compiled code
- Generate flame graphs for visualization

## Quick Start

```bash
# Basic profiling
<launcher> --cpusampler <program>

# With tier breakdown (RECOMMENDED)
<launcher> --cpusampler --cpusampler.ShowTiers=true <program>

# Skip warmup (recommended for steady-state analysis)
<launcher> --cpusampler --cpusampler.Delay=5000 --cpusampler.ShowTiers
    =true <program>
```
````

```
## REQUIRED: Fermi Verification (Every Tool Invocation)

**Before running**:

- [ ] Pre-calculate: Expected hot functions (1-5 names), T0/T1/T2
    split
- [ ] Smoke test: `<launcher> --cpusampler -c 'print 1;'` →Verify
    output format

**After running**:

- [ ] Validate: Actual vs estimate within 1 order of magnitude? YES /
    NO
- [ ] If NO: **STOP** - Debug tool before proceeding (run `--help:
    cpusampler`, test on known-good input)
- [ ] Save output: `tool-outputs/cpu-sampler-[benchmark].txt`

**Gate**: All boxes checked? →Proceed to analysis

## Key Options

| Option | Description | Recommended Value |
| ------------------------------- | ----------------------- |
    ---------------------- |
| `--cpusampler.ShowTiers=true` | Show T0/T1/T2 breakdown | **Always
    use** |
| `--cpusampler.Delay=<ms>` | Skip warmup | 2000-10000 |
| `--cpusampler.Period=<ms>` | Sample interval | 10 (default) |
| `--cpusampler.Output=calltree` | Output format | Default: histogram
    |
| `--cpusampler.OutputFile=<file>` | Save to file | For later analysis
     |
| `--cpusampler.SampleInternal=true` | Include internal frames | For
    Truffle debugging |

## Understanding Output

### Tier Columns

| Tier | Meaning | Target |
| ---- | ------------------- | ---------------------- |
| T0 | Interpreter | <10% for hot functions |
| T1 | First-tier compiled | Transitional |
| T2 | Fully optimized | >80% for hot functions |

### Sample Output

```
Sampling Histogram. Recorded 412 samples with period 10ms.
  Self Time: Time spent in function (excluding callees)
  Total Time: Time in function including callees

Name || Total Time || Self Time || T0 | T1 | T2
queens || 1850ms 88.0% || 1850ms 88.0% || 5.2% | 3.1% | 91.7%
hasConflict || 250ms 11.9% || 250ms 11.9% || 8.8% | 4.2% | 87.0%
```

## Interpretation Guidelines
```

````
### Good Performance

- Hot functions show >80% T2 time
- <10% T0 (interpreter) time
- Time concentrated in expected hot functions

### Performance Problems

- >30% T0 time →Compilation issues
- High T1 but low T2 →Optimization barriers
- Time in unexpected functions →Algorithm issues

## Integration with Other Skills

**Next steps based on findings**:

| Finding | Next Skill |
| ---------------------- | ------------------------------- |
| High T0 time | `tracing-compilation-events` |
| Optimization barriers | `detecting-performance-warnings` |
| Memory issues suspected | `profiling-memory-allocations` |
| Inlining problems | `tracing-inlining-decisions` |

## Related Skills

- `tracing-execution-counts` - Execution frequency (not time)
- `detecting-performance-warnings` - Find optimization barriers
- `tracing-compilation-events` - Compilation behavior

## Reference

```bash
# Full help
<launcher> --help:cpusampler
```

See [PATTERNS.md](PATTERNS.md) for common problem patterns and
    solutions.
````

# Appendix H.

# Independent Variables Examples

## H.1. Geometric Mean Speedup $S$

To illustrate how the geometric mean speedup $S$ is obtained from the raw execution times, consider a run evaluated on a subset of $|B| = 5$ AWFY micro benchmarks. For each benchmark $i \in B$, the baseline execution time $b_i$ and the optimized execution time $a_i$ are measured in milliseconds. The per-benchmark speedup is then the ratio $b_i/a_i$. The measured times and the resulting ratios are summarized in table H.1.

Table H.1.: Baseline and optimized execution times in milliseconds for the example subset of AWFY micro benchmarks, together with the per-benchmark speedup $b_i/a_i$.

| Benchmark | $b_i$ (ms) | $a_i$ (ms) | $b_i/a_i$ |
|---|---|---|---|
| Bounce | 120 | 95 | 1.263 |
| List | 88 | 80 | 1.100 |
| Mandelbrot | 210 | 150 | 1.400 |
| NBody | 64 | 70 | 0.914 |
| Towers | 155 | 120 | 1.292 |

Four of the five benchmarks are faster than the baseline ($b_i/a_i > 1$), while NBody regresses ($b_i/a_i < 1$), as its optimized time exceeds the baseline. Substituting the ratios into the definition of $S$ gives

$$\begin{aligned} S &= \left(\frac{120}{95} \cdot \frac{88}{80} \cdot \frac{210}{150} \cdot \frac{64}{70} \cdot \frac{155}{120}\right)^{\frac{1}{5}} \\ &= \left(1.263 \cdot 1.100 \cdot 1.400 \cdot 0.914 \cdot 1.292\right)^{\frac{1}{5}} \\ &= 2.297^{\frac{1}{5}} \approx 1.18. \end{aligned} \tag{H.1}$$

The run therefore achieves a geometric mean speedup of $S \approx 1.18$, meaning the optimized version is faster than the baseline across the five benchmarks.

Because $S$ is a geometric rather than an arithmetic mean, each benchmark contributes multiplicatively, so the single regression on NBody dampens the overall result but does not dominate it, and a benchmark that is twice as fast offsets one that is half as fast.

## H.2. Token Costs $C$

To illustrate how the token cost $C$ is derived from the raw usage fields, consider a short optimization run consisting of $k = 3$ API requests. The run uses two models, $M = \{\text{Sonnet}, \text{Haiku}\}$: the main agentic loop is served by Claude Sonnet 4.5, while a single delegated subtask is routed to the cheaper Claude Haiku 3.5. The per-token prices $r_m$ follow directly from Anthropic's published per-million-token (MTok) pricing [14], that is $r_m = p_m/10^6$ for a listed price $p_m$ in US dollars per MTok. The relevant prices are summarized in table H.2.

Table H.2.: Per-model prices in US dollars per million tokens (MTok) used in the example, following Anthropic's published pricing [14]. Cache writes refer to the 5-minute cache tier.

| Model | $p_m^{\text{base}}$ | $p_m^{\text{write}}$ | $p_m^{\text{hit}}$ | $p_m^{\text{out}}$ |
|---|---|---|---|---|
| Sonnet 4.5 | 3.00 | 3.75 | 0.30 | 15.00 |
| Haiku 3.5 | 0.80 | 1.00 | 0.08 | 4.00 |

The four token counts reported in the `usage` field of each server response are listed in table H.3. The first request writes the system prompt and the initial project context into the cache, so its cache write count is large. The second request reuses that context, which is reflected in a high cache hit count. The third request is a short, self-contained call to Haiku that neither reads from nor writes to the cache.

Table H.3.: Token counts per request $j$ as reported in the Claude Code session files.

| $j$ | $m_j$ | $t_j^{\text{base}}$ | $t_j^{\text{write}}$ | $t_j^{\text{hit}}$ | $t_j^{\text{out}}$ |
|---|---|---|---|---|---|
| 1 | Sonnet 4.5 | 2 000 | 25 000 | 0 | 800 |
| 2 | Sonnet 4.5 | 1 500 | 3 000 | 27 000 | 1 200 |
| 3 | Haiku 3.5 | 500 | 0 | 0 | 300 |

Substituting the token counts and per-token prices into the cost definition and factoring out $10^{-6}$ from the MTok prices yields

$$\begin{aligned} C &= \big(2\,000 \cdot 3.00 + 25\,000 \cdot 3.75 + 0 \cdot 0.30 + 800 \cdot 15.00\big) \cdot 10^{-6} \\ &\quad + \big(1\,500 \cdot 3.00 + 3\,000 \cdot 3.75 + 27\,000 \cdot 0.30 + 1\,200 \cdot 15.00\big) \cdot 10^{-6} \\ &\quad + \big(500 \cdot 0.80 + 0 \cdot 1.00 + 0 \cdot 0.08 + 300 \cdot 4.00\big) \cdot 10^{-6} \\ &= \big(111\,750 + 41\,850 + 1\,600\big) \cdot 10^{-6} \\ &= 155\,200 \cdot 10^{-6} \approx \$0.155. \end{aligned} \tag{H.2}$$

The full run therefore costs approximately \$0.16 if the consumed tokens were billed at the pay-as-you-go API rates. As expected, the first Sonnet request dominates the total because of its large cache write, while the delegated Haiku request contributes only a negligible fraction of the overall cost.

# Appendix I.

# Truffle Anti-Pattern Examples

## I.1. Missing Specialization

Listing I.1: One megamorphic fallback instead of per-type specializations

```
@GenerateUncached
@GenerateInline(true)
public abstract class AddNode extends BinaryOperationNode {

    @TruffleBoundary
    @Specialization
    protected Object doOp(Node node, Object left, Object right)
        {
        if (left instanceof Long l && right instanceof Long r) {
            return Math.addExact(l, r);
        } else if (left instanceof Double l && right instanceof
            Double r) {
            return l + r;
        } else if (left instanceof Long l && right instanceof
            Double r) {
            return (double) l + r;
        } // ... 11 further instanceof branches ...
        else {
            throw new LoxRuntimeError("'" + left + " + " + right +
                 "' is invalid", node);
        }
    }
}
```

## I.2. Missing Inline Cache

Listing I.2: Indirect call target lookup on every function call

```
@GenerateUncached
@GenerateInline(true)
public abstract class LoxCallFunctionNode extends Node {
    public abstract Object execute(Node node, Object left,
        Object right);

    @Specialization
    static Object doCall(LoxFunction function, @Variadic Object
        [] arguments) {
        // No @Cached("function.getCallTarget()") CallTarget +
            DirectCallNode:
        // every call resolves and invokes the target indirectly.
        return function.getCallTarget().call(function.
            createArguments(arguments));
    }
}
```

## I.3. Data Structure Regressions

Listing I.3: ArrayList backing instead of a native array

```
public class LoxArray {
    private final ArrayList<Object> elements; // was: Object[]
        elements

    public LoxArray(Object[] array) {
        elements = new ArrayList<>();
        for (Object obj : array) {
            elements.add(obj); // element-wise copy + boxing
        }
    }
    // ...
}
```

## I.4. Misplaced Truffle Boundaries

Listing I.4: Array element access cannot be compiled/inlined

```
@TruffleBoundary
public Object get(int index) {
    if (index < 0 || index >= elements.size()) {
        return Nil.INSTANCE;
    }
    var result = elements.get(index);
    return result != null ? result : Nil.INSTANCE;
}

@TruffleBoundary
public void set(int index, Object value) {
    while (index >= elements.size()) {
        elements.add(null);
    }
    elements.set(index, value);
}
```

# Appendix J.

# Experiment Prompts

Listing J.1: Prompt for Unenhanced Agent.

```
Task: Analyze and improve the performance of this Lox language
    implementation. Fix exactly one performance issue fully. Return a
    summary.

Sidenote: If your changes break the build or tests, fix them before
    continuing.

Sidenote: Do NOT stop to ask me questions. Make reasonable decisions
    on your own and continue until the task is complete. If you
    encounter ambiguity, use your best judgment and document your
    choices. Only stop
if you hit a truly unrecoverable error.

Sidenote: Document your findings and reuse information from previous
    iterations in the working directory itself only.
```

Listing J.2: Prompt for Enhanced Agent.

```
Task: Analyze and improve the performance of this Lox language
    implementation. Fix exactly one performance issue fully. Return a
    summary.

  Important: Use the /optimization-workflow-orchestrator skill to
      drive this process

  Sidenote: If your changes break the build or tests, fix them before
      continuing.

  Sidenote: Do NOT stop to ask me questions or for reviews. Skip all
      approvals. Make reasonable decisions on your own and continue
      until the task is complete. If you encounter ambiguity, use your
       best judgment
  and document your choices. Only stop if you hit a truly
      unrecoverable error.

  Sidenote: Document your findings and reuse information from previous
       iterations in the working directory itself only.
```

# Appendix K.

# Study Script

# Study Script

Antony Kamp

Feburary 2026

## 1 Introduction

Welcome to this User Study. Before we get started please read through the following points carefully.

- The study is planned to take no longer than 180 minutes. It can finish earlier if all tasks are completed.
- You may withdraw from the study at any point.
- You may also take a break at any point. When we reach the halfway mark there will be a short break.
- There are no wrong answers. We are **not testing you**.
- All collected data (if published) will be anonymous.
- During the study we would kindly ask you to think out loud as much as possible.

### Conductor Tasks

The following tasks need to be completed by the study conductor before continuing.

- Handout the consent form and answer any questions that might arise.
- Get the form signed and start the recordings
- Ask if the participant still wants to participate
- Give out reimbursement / explain how the process will work
- Make sure the participant answers the demographic questions on the next page
- While the participant fills out answers, make sure the study setup is ready to go

## Consent

I agree to participate in this user study conducted by *Antony Kamp* as part of their Master's Thesis at HPI. I understand that during this study, audio and the computer screen will be recorded for research purposes. I acknowledge that my participation is voluntary, and I have the right to withdraw at any time without penalty. I understand that my data will be kept confidential and used only for research purposes. Furthermore, I am aware that any published data will be anonymous. All my questions related to the collection, processing, and publishing of data in this study have been answered. By signing below, I consent to the recording of audio and my actions on the computer screen for the duration of the study



Place, Date & Signatur

# Demographic Interview

Before starting with the first task please answer the following questions.
**Important**: All are optional and can be left blank if you'd rather not answer.

- What is your current age? _____
- Are you enrolled in any studies at the moment?:
  - ☐ Bachelor
  - ☐ Master
  - ☐ PhD
  - ☐ other: _____
- Years of experience in software development? _____

Please note down (in bullet points) what ideas, projects, or tasks (if any) you have worked on with the following tools:

| **Tool** | **Experience** |
|---|---|
| GraalVM | |
| Truffle | |
| AI | |
| AI Coding Assistance | |

### Almost ready to start

Start recording computer screen and audio after consent is given.

# 2 Task 1

Before starting, study conductor fills out the following form:

- Available Resources
  - ☐ CLI (subsection 6.1)
  - ☐ Claude Code (subsection 6.2)

## 2.1 Phase 1: Recap (20min)

#### Conductor Task

Prepare the task by setting up the "Lox" language Implementation byopl24-07-A from Seminar "Build Your Own Programming Language" at Hasso-Plattner Institute from the wintersemester 2024/2025. Open it in Visual Studio Code and the Shell.

#### Participant Task

Please take a look at the language implementation of the simple script language "Lox". Get familiar with the code base, the GraalVM and Truffle and tools provided by Truffle. Use the provided Resources. You have 20 minutes.

### Phase 2: Optimization

#### Participant Task

You task is to find performance issues in the language implementation, validate them and fix them. You can use the provided resources. You have one hour.
Remember: We are not testing you. There are no wrong answers. You can also take a break at any point. Please speak your thoughts and feeling during this task out loud.

#### Conductor Task

Remember the participant to speak thoughts and feelings out loud if they remain silent for over 2 minutes: "Please keep talking through your thought process. I want to hear what you are looking for, what you are confused by, and what you are reading"

## 3 Break

Let's have a break for 10min.

## 4 Task 2

Before starting, study conductor fills out the following form:

- Available Resources
  - ☐ CLI (subsection 6.1)
  - ☐ Claude Code (subsection 6.2)

The study constructor ensures that the participant uses a different set of resources in tasks 1 and 2.

### 4.1 Phase 1: Recap (10min)

#### Conductor Task

Prepare the task by setting up the "Lox" language Implementation byopl24-07-B from Seminar "Build Your Own Programming Language" at Hasso-Plattner Institute from the wintersemester 2024/2025. Open it in Visual Studio Code and a Shell.

#### Participant Task

Please take a look at the language implementation of the simple script language "Lox". Get familiar with the code base. Use the provided Resources. You have 10 minutes.

### Phase 2: Optimizaiton

#### Participant Task

You task is to find performance issues in the language implementation, validate them and fix them. You can use the provided resources. You have one hour.
Remember: We are not testing you. There are no wrong answers. You can also take a break at any point. Please speak your thoughts and feeling during this task out loud.

#### Conductor Task

Remember the participant to speak thoughts and feelings out loud if they remain silent for over 2 minutes: "Remember the participant to speak thoughts and feelings out loud if they remain silent for over 2 minutes: "Please keep talking through your thought process. I want to hear what you are looking for, what you are confused by, and what you are reading"

122

# 5 Finishing up

## Conductor Task

Ask the following questions before completing the study:

- Before concluding the study, is there anything you want to add or that you wish we could have asked?
- Do you have any questions for us?

Thank you for making time to participate in this study.

# 6 Definition of Available Resources

This section defines the available resources the participant can have to complete the task.

## 6.1 CLI

The participant has the following resources to complete the task:

- Command Line Shell without Claude Code
- Unrestricted Internet Access
- IGV open
- Additional Material
  - Graal Optimization Guide (link)
  - Slides "Tool Support" from Seminar "Build Your Own Programming Language" at Hasso-Plattner Institute, by Jens Lincke, Tim Felgentreff, Fabio Niephaus in Wintersemester 2024/2025
  - Slides "Bytecode DSL Update" from Seminar "Build Your Own Programming Language" at Hasso-Plattner Institute, by Christian Humer in Wintersemester 2024/2025
- Visual Studio Code
- Short description of the code base

## 6.2 Claude Code

The participant has the following resources to complete the task:

- Claude Code with Extension
- Unrestricted Internet Access
- IGV open
- Additional Material
  - Graal Optimization Guide (link)
  - Slides "Tool Support" from Seminar "Build Your Own Programming Language" at Hasso-Plattner Institute, by Jens Lincke, Tim Felgentreff, Fabio Niephaus in Wintersemester 2024/2025
  - Slides "Bytecode DSL Update" from Seminar "Build Your Own Programming Language" at Hasso-Plattner Institute, by Christian Humer in Wintersemester 2024/2025
  - Claude Plugin user manual
- Visual Studio Code
- Short description of the code base

123

# Appendix L.

# Speedup Progression per Benchmark Diagrams

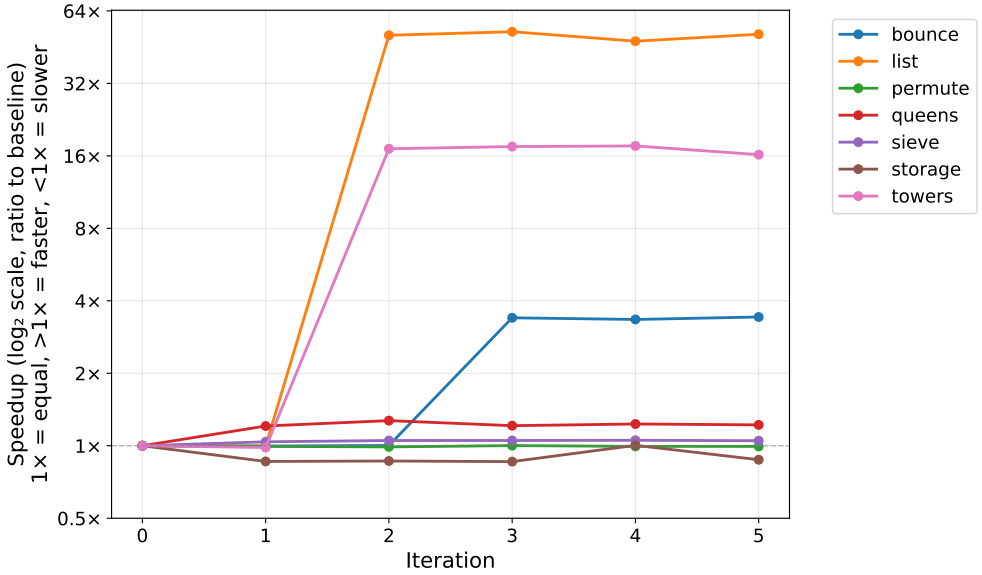



Figure L.1.: Speedup progression per benchmark for Run 3 for "Original Implementation" with "Enhanced Agent".

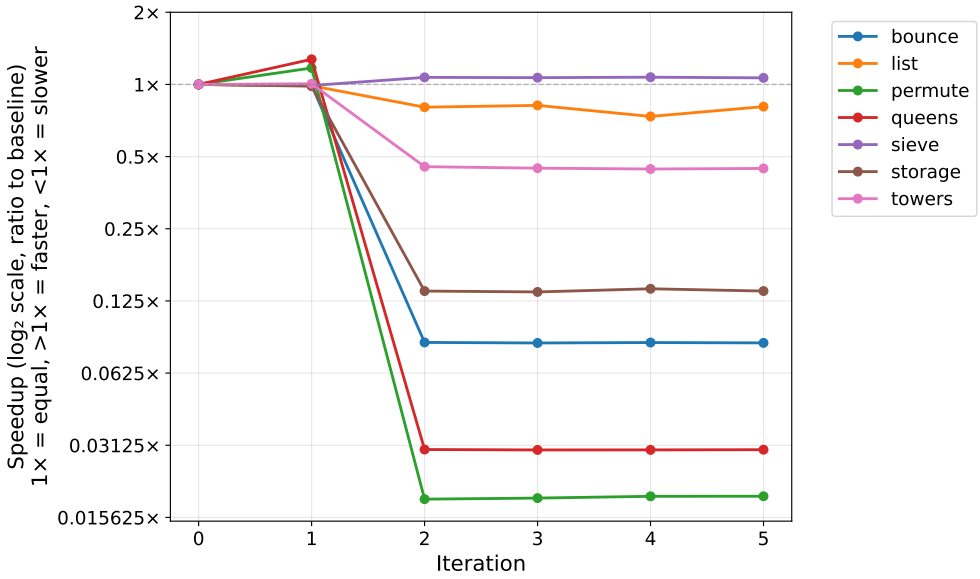



Figure L.2.: Speedup progression per benchmark for Run 9 for "Original Implementation" with "Unenhanced Agent".

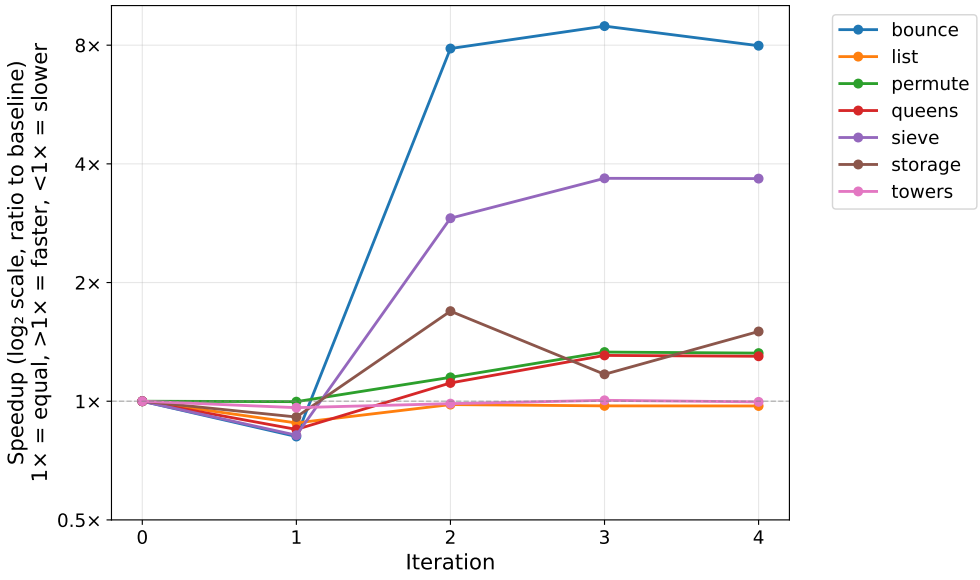



Figure L.3.: Speedup progression per benchmark for Run 8 for "Degraded Implementation" with "Enhanced Agent".

# Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst sowie keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe.

Potsdam, den 15. Juli 2026

Antony Kamp