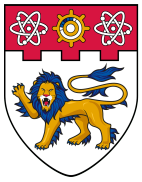

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

École Polytechnique Fédérale de Lausanne

Design and Benchmarking of Agentic Code Generation Systems

by Maxime Zammit

Master Thesis

Approved by the Examining Committee:

Prof. Edouard Bugnion
Thesis Advisor (EPFL - École Polytechnique Fédérale de Lausanne)

Asst. Prof. Dmitrii Ustiugov
Thesis Advisor (NTU - Nanyang Technological University)

Dr. Wenyan Chen
Thesis Supervisor

EPFL IC IINFCOM DCSL
INN 237 (Bâtiment INN)
Station 14
CH-1015 Lausanne

March 13, 2026

Acknowledgments

I would like to express my sincere gratitude to Professor Bugnion for his guidance, support, and trust throughout this thesis. I am also very grateful to Professor Ustiugov and the HyScale Lab at NTU Singapore for welcoming me during my time abroad. A special thank you goes to my supervisor, Dr. Wenyan Chen, for her guidance over the past six months, as well as to Colin Hong and Hongrui Liu for their dedicated mentorship and support during my research.

I extend my thanks to all the friends and colleagues who supported me during this project, with a special thanks to Alexandre and François, who made a stopover in Singapore to visit me.

Finally, my deepest appreciation goes to my parents, my sister, and my extended family for their unwavering support and encouragement. I would also like to dedicate a special mention to my grandmother, Lucie, whose resilience and values of kindness, hard work, and perseverance continue to inspire me.

Singapore, March 13, 2026 Maxime Zammit

Abstract

The rapid evolution of Large Language Models (LLMs) has resulted in the development of Autonomous Coding Agents, transitioning Artificial Intelligence (AI) from simple code autocompletion to executing complex, multi-step software engineering tasks. However, evaluating these agentic workflows remains an open challenge. Unlike traditional single-model inference, AI agents exhibit dynamic control flows, heterogeneous component interactions, and significant system-level overheads, mainly concerning Key-Value (KV) cache memory management.

This thesis presents a comprehensive analysis of the current coding agent landscape and introduces a custom framework designed to evaluate the end-to-end performance of agentic workflows. First, we attempt to clarify the fragmented ecosystem by proposing a coding agent blueprint designed as the least common multiple of existing state-of-the-art architectures. Second, we implement this blueprint, providing a benchmarking framework to evaluate both system performance metrics and coding capabilities.

Through this implementation, we investigate the memory management requirements of handling extensive context histories, identifying an optimization opportunity for the KV cache by actively managing and refining the context window. Finally, we empirically evaluate our proposed agent on the HumanEvalPlus benchmark, achieving an accuracy of 90.85%. This work provides a framework for comparing diverse coding agent architectures and settings, ultimately informing the design of more robust, efficient, and scalable LLM-serving infrastructures.

Contents

Chapter 1

Introduction

The public debut of ChatGPT[30] in November 2022 is widely regarded as a turning point in the artificial intelligence (AI) era [11]. While Large Language Models (LLMs) had been evolving within research laboratories for years, this specific release transitioned the technology from a subject of theoretical academic interest to a centerpiece of global infrastructure [43]. Initially, the high computational costs associated with LLMs created a significant barrier to entry for the academic community. A pivotal technical breakthrough for the open ecosystem emerged from UC Berkeley with the introduction of vLLM[38]. By implementing PagedAttention [22], which is a technique mirroring virtual memory management in traditional operating systems, researchers successfully addressed the "Key-Value (KV) cache" bottleneck. This innovation increased inference throughput by 2x to 4x by significantly reduced memory fragmentation, rendering the deployment of state-of-the-art models on standard academic hardware feasible, rather than necessitating massive industrial clusters. The convergence of efficient serving mechanisms, such as vLLM[38], and advanced deliberative reasoning strategies, such as Chain-of-Thought (CoT) [39], has facilitated the rise of Autonomous Coding Agents. In this context, the role of AI has transcended simple code autocompletion. Contemporary agents [41] are capable of navigating multi-file repositories, executing shell commands to diagnose environmental errors, and performing test-driven self-correction. By utilizing reasoning tokens to analyze a debugging cycle before generating code, these agents are redefining the software development lifecycle, shifting from passive assistants to autonomous technical partners.

AI is currently one of the most dynamic fields in modern technology. As with any rapidly evolving domain, key improvements and milestones occur frequently; however, quantifying these advancements is non-trivial. To objectively assess progress, robust metrics must be defined [24]. A system that excels according to one metric may underperform when evaluated against another. Furthermore, it is critical to eliminate human bias from performance measurement. This necessity underscores the importance of benchmarking. The primary objective of benchmarking is to establish a standardized testing framework that allows for the comparison of different settings or environments using identical datasets, thereby facilitating objective analysis. While standard

metrics for LLM performance exist, such as Time To First Token (TTFT) and Time Between Tokens (TBT), and system metrics like GPU and memory utilization are well-understood, there is a growing need for novel metrics designed specifically to assess the complex performance of autonomous agents.

The AI landscape in 2025 is increasingly defined by the proliferation of Agentic AI. A survey of the current state of the art [40] reveals a fragmented ecosystem where methodologies vary significantly, creating an appearance of disparity across systems. In the context of this thesis, the scope is narrowed specifically to Coding Agents, a subdomain anticipated to be a critical area of development in the coming years.

Given the significant heterogeneity among Coding Agents, this thesis aims to disambiguate the current landscape, describing existing options and identifying the common denominators across these systems. It is argued that following a year characterized by the frequent release of diverse systems, a comparative analysis is essential to understand the current technological baseline and identify future research directions for improving coding agents.

Consequently, this project seeks to deepen the understanding of the underlying mechanics of coding agents. The scope includes the analysis of existing tools, the development of a custom agentic code benchmarking framework, and an investigation into the memory management requirements of such tools.

Through the implementation of this custom benchmarking framework, our baseline agent achieved high accuracy on the HumanEvalPlus[25] benchmark. With a 90.85% resolution accuracy, our implementation ranks among the highest scores on the benchmark leaderboard. Furthermore, to address the memory bottlenecks resulting from the KV cache, we successfully reduced average memory utilization by 8.75% and peak utilization by 12.30%.

A key proposition of this work is to conduct a comparative analysis of existing AI agents, many of which have been constructed empirically. By identifying architectural similarities and highlighting key divergences, this analysis provides a theoretical foundation. This comparative study is complemented by the development of a custom agentic code benchmarking framework, which serves as a benchmark artifact to empirically evaluate model coding capabilities. Additionally, we attempt to address memory scarcity by implementing and evaluating an L_2 norm-based KV cache eviction strategy. The latter actively manage the OpenHands[35] context, this approach mitigates the critical memory bottlenecks inherent to agentic workflows.

Chapter 2

Background

This chapter introduces the foundational concepts underlying agentic coding workflows. These agents represent the latest evolution in a long lineage of developer assistance tools that have been refined since the early days of computer science. Software engineering is a foundational pillar of modern digital society, underpinning productivity and innovation across virtually all critical sectors. Given the disruption caused by AI across various industries, it is unsurprising that it has begun to fundamentally transform such a critical technical discipline.

The timeline for AI and automation in software engineering predates the release of modern tools like GitHub Copilot[15]. For decades, Integrated Development Environments (IDEs) have employed heuristic, speculative, and early AI techniques to accelerate development and enhance programmer productivity. As early as 2005, researchers such as R. Holmes and G. C. Murphy [17] explored techniques to analyze codebases and provide context-aware recommendations for application programming interface (API) usage, refactoring and more. Initially, these tools were relatively rudimentary, often limited to basic syntactic autocomplete and inline documentation retrieval.

However, as these assistance features matured, they became central to developer workflows, significantly influencing the widespread adoption of specific tools. A notable historical example is the migration from the Eclipse IDE to IntelliJ IDEA[19] during the mid-2010s. This shift illustrates a core principle in software engineering: developers naturally gravitate toward tools that reduce cognitive load and provide reliable, automated assistance. By the mid-2000s, the open-source Eclipse IDE[12] was the established industry standard for Java development [14]. However, its fragmented, plugin-heavy architecture often caused environment instability and required significant manual configuration. In contrast, IntelliJ IDEA provided a unified, ready-to-use experience out of the box. Its built-in features enabled advanced, context-aware code completion, real-time inspections, and reliable project-wide refactoring. The release of IntelliJ's free community edition around 2010 accelerated a massive migration away from Eclipse [18]. This preference for highly integrated and

intelligent assistance remains clear today. JRebel's 2025 Java Developer Productivity Report shows that IntelliJ IDEA now dominates the market, serving as the primary development environment for 84% of Java developers [1].

While tools like IntelliJ revolutionized development through advanced static analysis and rigid, rule-based heuristics, the next major paradigm shift in automated assistance abandons deterministic rules in favor of probabilistic generation. This new frontier is driven by LLMs. To understand how modern agentic coding workflows operate, it is essential to first understand the underlying mechanics of these models and how they process source code.

2.1 LLM

Large Language Models (LLMs) are advanced deep learning algorithms fundamentally designed to understand and generate text by probabilistically predicting subsequent elements in a sequence. To comprehend how these models function, it is necessary to first look at how machines process text.

LLMs do not natively understand words; instead, they rely on a finite vocabulary of tokens. Tokenization is the process of dividing text into sub-word units, acting as a compression mechanism. A critical advancement in this area was the application of Byte Pair Encoding (BPE) to natural language processing in 2015, which significantly improved the handling of rare words and translation benchmarks [33]. Once text is tokenized, these tokens are mapped into high-dimensional mathematical vectors known as embeddings. The seminal Word2Vec research published by Mikolov et al. in 2013 [27] demonstrated that these vector representations capture deep semantic relationships. For example, Word2Vec showed that simple vector arithmetic could accurately predict relational logic between distinct entities:

$$v_{Madrid} - v_{Spain} + v_{France} \approx v_{Paris}$$

Figure 2.1: Exemple of Word2Vec[27]

In practical terms, this equation demonstrates that subtracting the vector representation of 'Spain' from 'Madrid' and adding the vector for 'France' yields a new mathematical vector. The closest corresponding token to this resulting vector in the embedding space is 'Paris.'

While embeddings and tokenization provided the foundational vocabulary, the architectural breakthrough that made modern LLMs possible occurred in 2017. Researchers at Google published the seminal paper "Attention Is All You Need"[37], introducing the Transformer architecture. By utilizing a self-attention mechanism, the Transformer allowed models to process entire sequences of tokens in parallel and capture long-range dependencies in text far more effectively than previous

sequential models.

The Transformer architecture provided the foundation for the Generative Pre-trained Transformer (GPT) lineage developed by OpenAI[30]. Beginning with GPT-1, these models demonstrated that scaling up both the parameter count and the volume of pre-training data led to highly adaptable and capable systems. This scaling trajectory culminated in the development of the GPT-3.5 model, which powered the public release of ChatGPT in late 2022. By pairing a massive, pre-trained model with an accessible conversational interface, ChatGPT popularized the capabilities of LLMs to the general public and ignited a massive, industry-wide acceleration in generative AI research and development.

2.2 Memory/KV Cache

Transformer-based autoregressive LLMs operate in two phases during inference: the Prefill phase and the Decoding phase. During Prefill, the model processes the initial input prompt. Because the entire input sequence is available concurrently, the computation of the Query (Q), Key (K), and Value (V) representations can be heavily. Consequently, this phase features high arithmetic intensity and is typically compute-bound.

To optimize the subsequent generation process, the model stores the computed K and V tensors for all prompt tokens in a memory pool known as the KV Cache.

The second phase, Decoding, generates output tokens autoregressively. Each new token n is conditioned on all previously processed tokens. During this phase, the model generates a new Query vector for the current token, but rather than recomputing the previous states, it retrieves the historical K and V tensors from the KV Cache to compute the self-attention scores. Because decoding processes only one token at a time, the arithmetic operations are minimal, but the entire KV Cache must be fetched from High Bandwidth Memory (HBM) to the processor's SRAM at every step. Consequently, the decoding phase is strictly memory-bandwidth bound.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Figure 2.2: Attention formula [37]

Agentic workflows iteratively append reasoning and tool outputs to shared prompts, necessitating efficient KV cache management. Reusing cached prompt prefixes across cycles eliminates redundant prefill computations, reducing TTFT and accelerating the agentic loop.

2.3 Attention Mechanism

To understand the mechanics of the KV cache, it is essential to examine the attention mechanism, which serves as the computational engine of modern LLMs. As previously mentioned, text generation is an autoregressive process, meaning each predicted token is conditioned on the entire sequence of preceding tokens.

Consider a user prompt containing the text "hello world". The first step is to convert this text into discrete integer IDs using the model's specific tokenizer. For instance, using OpenAI's tokenizer[36], this prompt yields the token sequence [24912, 2375]. The model maps these discrete IDs to dense vector embeddings.

During the Prefill phase, the model processes the input embeddings concurrently. It multiplies these embeddings by learned weight matrices to produce the Query (Q), Key (K), and Value (V) representations for each token. To eliminate redundant computation during subsequent generation steps, the K and V vectors for "hello" and "world" are stored in the GPU's memory, forming the KV cache. The attention mechanism then utilizes these computed values to calculate the sequence context and predict the first novel output token, such as an exclamation mark ('!').

During the subsequent Decoding phase, the model takes this newly generated token and computes only its specific Q, K, and V vectors. Rather than recomputing the representations for "hello" and "world", the attention mechanism retrieves their historical K and V tensors from the KV Cache. The model appends the new token's K and V vectors to the cache and computes the attention scores to predict the next token. By caching and retrieving these representations, the autoregressive generation avoids re-evaluating the entire prefix at every time step, significantly reducing computational overhead.

2.4 Paged Attention and vLLM

Early implementations of LLMs relied on a naive approach to KV caching that stored data in contiguous memory spaces. However, because the KV cache dynamically grows and shrinks over time[22], this contiguous allocation led to severe memory fragmentation. Kwon et al.[22] estimated that existing systems wasted between 60% and 80% of available memory due to this fragmentation and over-reservation.

To address this critical bottleneck, Kwon et al. introduced PagedAttention[22]. This mechanism adapts the concept of virtual memory and paging originally pioneered by Kilburn et al. in 1962[21] for LLM architecture. This implementation resulted in the development of vLLM[38], which has rapidly become an open-source industry standard for LLM inference platforms. By mitigating memory waste, vLLM delivered substantial performance improvements, achieving 2 to 4 times higher

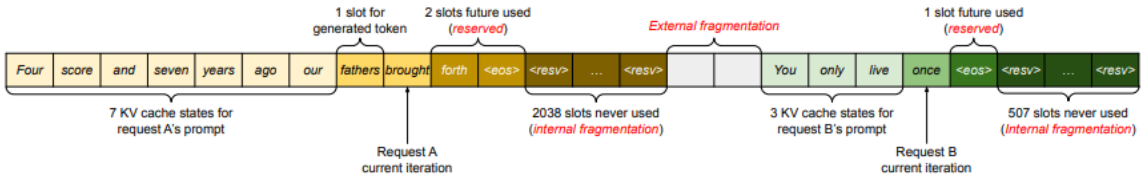

Figure 2.3: Memory Fragmentation[22]

throughput than prior systems and subsequently accelerating broader AI research and development.

2.5 Agentic AI

A dominant paradigm in 2025 artificial intelligence research is the transition toward agentic AI. Unlike traditional conversational models that passively generate responses based on a single prompt, agentic AI systems are defined by their ability to reason, plan, and autonomously interact with external environments and tools over multiple iterations. These multi-step, goal-driven workflows are currently being adopted across diverse industries. Applications range from autonomous fraud detection in finance to agentic coding, where agents iteratively write, test, and debug software, thereby significantly improving developer productivity and reducing the need for continuous human-in-the-loop oversight.

While agentic AI is a rapidly evolving concept with varying architectural implementations, the literature generally converges on a distinct set of characteristics that differentiate agents from standard generative models. As defined by Acharya et al. [2], these systems are designed to aim at those three primary core objectives:

- “Autonomy and Goal Complexity”[2]: The system does not merely execute predefined, rigid tasks. Instead, it actively formulates strategies to reach a high-level objective. For example, a specialized fraud-detection agent will not repeatedly apply a single heuristic; it will dynamically analyze a case, query necessary databases, and craft a bespoke evaluation path based on the specific context of the transaction.
- "Environmental and Operational Complexity"[2]: The agent can operate dynamically in evolving, uncontrolled environments, adjusting its execution strategy when it encounters unexpected situations or errors.
- "Independent Decision-Making and Adaptability"[2]: The architecture is specifically designed to handle long-horizon tasks that lack strict parameters or immediate end-states, successfully navigating them without requiring continuous human prompting. In addition, the system

possesses the self-sufficiency to autonomously manage its own computational resources, working memory, and external tool usage to accomplish its objectives.

2.6 L_2 Norm of key embeddings

One of the primary challenges currently faced by LLMs is the unconstrained growth of the KV cache. Processing long-context inputs results in a massive accumulation of cached keys and values, which frequently hits hardware memory limits and increases computational costs. Most existing KV cache compression methods either require modifying the model architecture (Ainslie et al., 2023)[3] or rely on computationally heavy retraining and fine-tuning (Nawrot et al., 2024)[29]. While some training-free methods exist, they often rely on complex algorithms that introduce significant computational overhead. To address this, Devoto et al. (2024)[10] demonstrated a strong negative correlation between the L_2 norm of cached keys and their corresponding attention scores. This finding allows for a highly effective KV cache compression strategy by keeping in memory only the keys with the lowest L_2 norms and their corresponding values. A major advantage of this technique is its seamless integration into existing pipelines . Computing the L_2 norm on key embeddings requires no model retraining and adds virtually zero computational expense.

Chapter 3

Design

This past year has seen the rapid rise of AI coding tools. Ranging from Cursor[5] to GitHub Copilot[15] and Claude Code[4], these have emerged as some of the industry leaders in this space. However, they are all closed-source, with their model-level mechanics kept as closely guarded secrets.

A primary objective of this project is to design a blueprint for an ideal coding agent. As discussed previously, an agentic workflow can be driven by a single LLM or an ensemble of multiple, potentially heterogeneous models. The blueprint presented below offers a flexible and generalized architecture designed to be adapted to best suit specific needs.

To develop an optimal agentic coding workflow, it is essential to comprehensively analyze both the existing landscape of AI tools and the traditional coding lifecycle employed by human software engineers. Because current coding agents vary significantly in their capabilities and execution, this section will first establish a taxonomy outlining the different types of agentic workflows currently in use.

3.1 Coding Agent Topology

Looking online and analyzing various coding agentic workflows, we found the following subcategories:

- Interactive Coding Assistants: Reactive, single-turn tools focused on autocomplete, inline documentation, and snippet generation. They lack long-term memory or the ability to autonomously plan multi-file changes. (e.g. GitHub Copilot (Classic)[15])
- Agentic IDEs & Workspaces (Human-AI Pair Programming): Deeply integrated into the editor. They bridge the gap between human-in-the-loop and full autonomy by utilizing context

engineering (reading the whole repository, understanding AGENTS.md files) and executing multi-file edits, terminal commands, and tool usage while keeping the developer in the driver's seat. (e.g. Cursor[5])

- Autonomous Software Engineers (End-to-End Agents): CLI-first or cloud-hosted systems designed to take a high-level issue, plan, execute, use tools (terminals, browsers), self-correct via "plan $\rightarrow$ execute $\rightarrow$ verify" loops, and open a pull request with minimal to zero human supervision. (e.g. Claude Code[4])
- Multi-Agent Collaborative Systems: Frameworks where multiple specialized AI agents collaborate, taking on different personas or roles (e.g., Planner, Coder, Reviewer, Tester) to execute complex pipelines concurrently. (e.g. MetaGPT[13])
- Continuous Integration: Asynchronous agents that live in your version control system (VCS) or CI/CD pipeline. They proactively review pull requests, flag architectural deviations, generate unit tests, and patch security vulnerabilities before code is merged. (e.g. CodeRabbit[9])
- DevOps & Site Reliability Agents: Agents that manage infrastructure, deploy code, monitor system health, and autonomously triage or resolve production incidents based on logs and metrics. (e.g. MistyAI[28])

In this project we are going to focus on autonomous coding systems which are very similar to the multi-agent way of working but for the sake of simplicity are implemented with a single LLM Model.

3.2 Coding Agent Architecture

In contemporary organizations, tasks are frequently siloed, leading to complex organizational structures. This specialization is rooted in Adam Smith's economic theory [34], which posits that individuals are significantly more productive when focused on a specific domain rather than acting as generalists. Designers of AI agents are increasingly adopting this paradigm, creating agentic architectures tailored for highly specific functions. A recent study by SkillsBench confirms this trend, demonstrating that generalist AI models underperform compared to those carefully curated for dedicated tasks [23].

Within the context of software engineering, we have defined a set of specialized agentic tasks that emulate the traditional Software Development Life Cycle (SDLC). This methodology follows established stages to ensure the generation of code that is functionally correct, secure, efficient, and strictly aligned with management requirements.

These stages can be abstracted into four primary components commonly found in agentic

coding workflows: Architecture Generator, Code Generator, Reviewer, and Validator. Let us explore each of these components in detail:

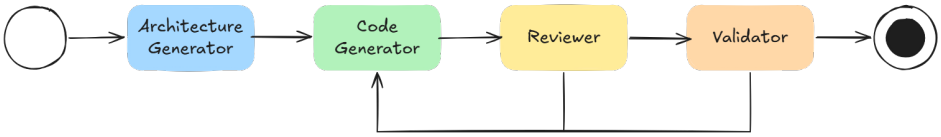

Figure 3.1: Global view of architecture

3.2.1 Architecture Generator

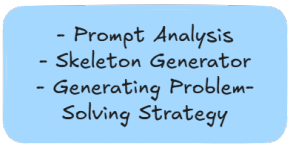

Figure 3.2: Architecture Generator

The primary objective of this stage is to formulate a strategic plan to resolve the given task. Drawing upon the fundamental computer science principle of "divide and conquer," this agent breaks down complex problems into manageable sub-tasks. Utilizing the initial prompt provided by the user alongside its underlying system instructions, the Architecture Generator produces a sequential list of implementation steps. Upon completing this architectural blueprint, the workflow transitions to the Code Generator.

3.2.2 Code Generator

Figure 3.3: Code Generator

As its name implies, the Code Generator is responsible for actively writing the software code until the defined plan is fully realized. During this phase, the agent leverages available tools to interface directly with the codebase. Once the agent determines the implementation is sound and the sub-tasks are complete, the process advances to the Reviewer phase.

Figure 3.4: Reviewer

3.2.3 Reviewer

The Reviewer's role is to critically analyze the generated code. It utilizes the LLM's reasoning capabilities to assess whether the task has been implemented correctly based on the initial architecture. It also employs specific tools to execute terminal commands, allowing it to interact with and observe the behavior of the newly produced code dynamically. If the code passes this internal LLM evaluation, the workflow proceeds to the Validator.

3.2.4 Validator

Figure 3.5: Validator

In contrast to the Reviewer, which relies primarily on the LLM's internal evaluation capabilities, the Validator ensures that the implemented solution meets the Definition of Done (DoD) of the task. This validation typically involves executing formal unit tests or employing other deterministic methods to verify requirements independently from the LLM's subjective assessment.

3.3 Reflective versus ReAct Framework

In the development of agentic AI, two fundamental paradigms have emerged to enhance model capabilities: the Reflection and Reasoning and Acting (ReAct) frameworks. Traditional Large Language Model (LLM) interaction typically relies on direct generation, where a user submits a query and the model outputs a final answer in a single computational pass. While effective for basic queries, this approach is often too simplistic for complex reasoning tasks.

To address these limitations, Madaan et al. (2023) demonstrated the efficacy of Reflection in their Self-Refine framework [26]. Rather than generating a single response, Reflection utilizes an

Figure 3.6: Direct LLM

iterative refinement loop. By prompting the model to generate an initial output, evaluate it for flaws, and subsequently refine it, the authors achieved performance improvements of up to 20% on specific reasoning tasks. This transforms the generation process from a single-pass prediction into a continuous quality enhancement loop.

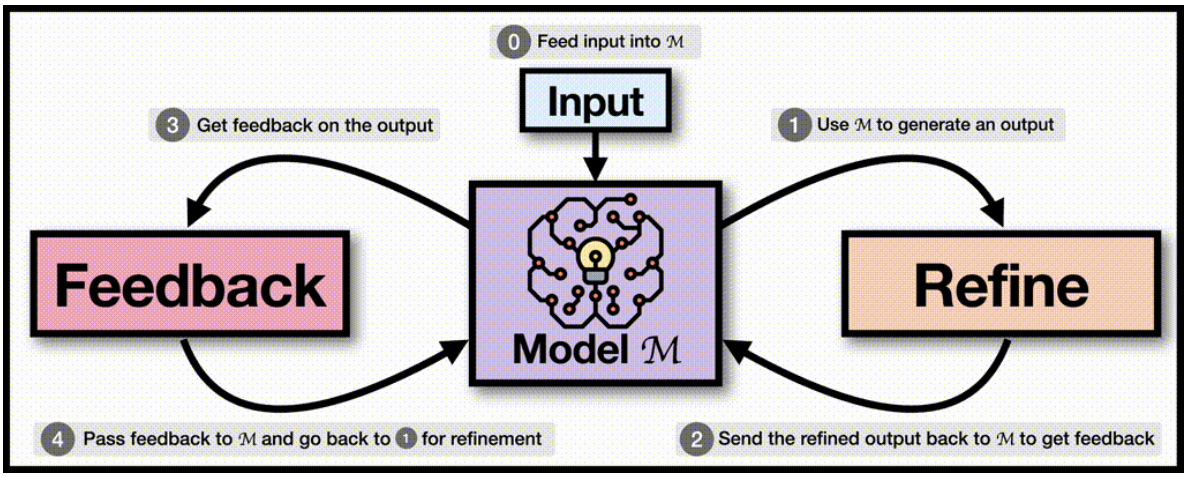

Figure 3.7: Self-Refine Framework[26]

Moreover, true agentic AI requires the ability to interact with external environments, a capability outside the scope of standalone text-generation weights. The ReAct framework bridges this gap. Because an LLM cannot directly interact with the real world, ReAct enables the model with a defined set of external tools (e.g. APIs, calculators, or search engines) [42]. The framework structures the LLM's output to interleave internal chain-of-thought reasoning with external action commands. The model reasons about a problem, executes a specific tool call, and ingests the resulting environmental observation. This dynamic reasoning-action-observation loop allows the agent to iteratively solve complex problems that require real-world grounding.

In our implementation, using OpenHands SDK[35], we follow the ReAct pattern using tools that are specialy curated for Agentic Coding Workflows.

3.4 Tools for Coding Agents

Tools are the primary mechanism through which coding agents interact with their environment. From utilizing the terminal and editing files to browsing the web, these interfaces allow LLM to interact with the external world. The OpenHands SDK provides three default tools utilized in this research: the TerminalTool, the FileEditorTool, and the TaskTrackerTool [16].

The specific capabilities of these three foundational tools are outlined below:

- **TerminalTool**: This tool enables the agent to execute commands within a Command Line Interface (CLI). Because the default implementation lacks built-in security constraints, its use should be restricted to isolated or controlled environments. Alternatively, it can be configured to restrict the agent to a predefined, safe subset of commands.
- **FileEditorTool**: Offering more granular control than standard terminal commands, this tool allows the agent to perform precise, line-by-line file edits. It also provides immediate feedback on the modifications made, ensuring the agent can verify its changes.
- **TaskTrackerTool**: Essential for workflow management within OpenHands, this tool allows the agent to process tasks sequentially. For more complex architectures, developers can utilize an alternative **DelegateTool**, which permits the primary agent to delegate complex subtasks to secondary agent instances, thereby balancing the workload.

In addition, the extensible architecture of OpenHands SDK simplifies the creation of custom interfaces, enabling the agent to interact with external software. For example, a custom web browser tool connected to a Kanban application could enable an agent in a coding workflow to autonomously read, update, and manage project tasks.

3.5 Security and Safety

The cybersecurity risks associated with the integration of AI are increasingly significant. Two primary threat vectors have emerged: the unverified integration of LLM-generated code into production environments, and the deployment of agentic LLMs with the autonomy to execute system-level commands. Regarding the latter, improper configuration can allow the model to execute destructive commands within the project or, more severely, escape the project directory to modify sensitive host machine files.

In the first case, the risk comes from the fact that LLM models are mainly trained on historical codebases with historical programming practices. As Sebastiano P. highlights in his book chapter, the model will learn from obsolete data using techniques that are no longer best practices or that

even contain security flaws. The model might learn those patterns and propagate them in future code, which could multiply the chances of a cyberattack. We can sum up that this can cause the model to “favor older code libraries and repositories for learning, leading to an over-representation of deprecated and potentially risky coding paradigms” [31]. Therefore, code produced by an LLM inherently carries security risks and necessitates rigorous human review and automated testing before deployment.

To mitigate the active risks associated with agentic execution, it is imperative to enforce strict principles of least privilege. Furthermore, the agent’s workspace must be heavily isolated within a sandboxed or virtualized environment, such as a container, to prevent unauthorized system access.

Chapter 4

Implementation

As established in previous sections, the practical component of this research was implemented using the OpenHands SDK [35]. Iteratively developed from a baseline prototype, the system was extended into a more sophisticated agent architecture. Specifically, the software was augmented with several advanced capabilities: gathering real-time data during execution, implementing a Key-Value (KV) cache eviction mechanism based on the L_2 norm to optimize memory performance, sequentially executing batches of tasks fetched from a Git repository, validating agent behavior through automated testing, and generating structured execution feedback.

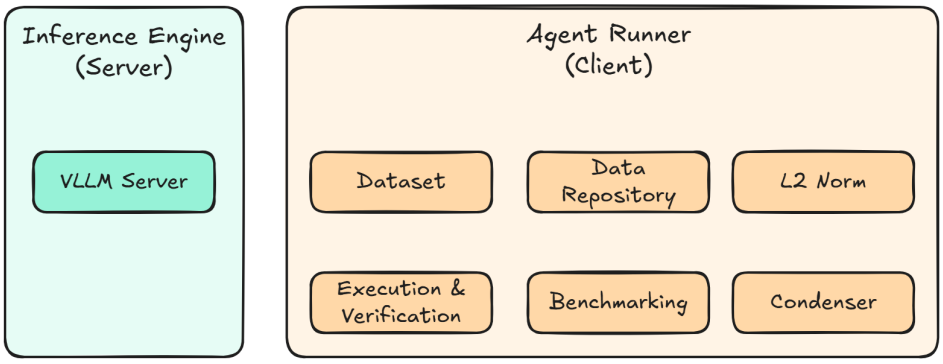

Figure 4.1: Global view of the Implementation

4.1 OpenHands SDK

The OpenHands Software Agent SDK [35] (formerly OpenDevin) is a modular framework designed to facilitate the development of autonomous, code-centric AI agents. Initially conceived as an open-source alternative to Devin AI, the project has evolved into a comprehensive SDK providing a model-

agnostic Application Programming Interface (API) for the creation of agent-driven software services. Highlighting its efficacy, OpenHands agents have demonstrated state-of-the-art performance on software engineering benchmarks, achieving over 71% resolution rates on SWE-bench when utilizing advanced backend models such as GPT-5[20].

The OpenHands SDK utilizes a decoupled, event-driven architecture built upon several core components:

- AI Agents: This module defines the cognitive structure and configuration of the agents. It manages interactions with the underlying LLM, frequently employing paradigms like CodeAct (Code as Actions) to dynamically generate executable scripts rather than relying strictly on predefined APIs.
- LLM Interface: Leveraging LiteLLM as a unified interface, this component abstracts the complexities of communicating with various models. It enables seamless integration with virtually any commercial or open-weight LLM API.
- Event Stream and State Management: Serving as the backbone of the architecture, this module manages conversation flow and maintains a persistent event storage system. It orchestrates the Action-Observation loop by routing agent generated actions to the environment and feeding the resulting observations back into the agent's context window.
- Agent Context and Skills: This component handles the injection of solving strategies, contextual data, and custom skills into the LLM prompt. Its primary function is to maintain a high signal-to-noise ratio within the context window.
- Tooling Integration: Provides the interfaces through which the agent interacts with external environments, encompassing capabilities such as shell execution, file parsing, web browsing, and code editing.
- Runtime Environment (Sandbox): Ensures secure and isolated code execution. By leveraging containerization technologies (e.g., Docker) or secure cloud-based runtimes, the framework guarantees that agent-generated code cannot compromise the host system.

While this project's implementation interacts with the majority of these components, the Sandbox runtime environment was intentionally omitted. Because the benchmark evaluations are executed within a strictly controlled and isolated environment, the additional layer of sandboxing was deemed unnecessary. However, transitioning this system to a production setting would necessitate the reintroduction of the sandbox feature, despite its computational overhead,to mitigate security risks and prevent unauthorized code execution.

4.2 Benchmark Handling

To benchmark agentic code generation, we needed to capture various metrics, including GPU utilization, memory utilization, KV cache utilization, TTFT, TBT, generation throughput, and the specific inputs and outputs of the model. These data points originate from several sources, which are detailed in this section.

The first step involves logging system-level data, such as GPU and memory usage. This information is collected by executing the *nvidia-smi* command on the node hosting the vLLM instance.

```
# --- MONITORING START ---
LOG_FILE="gpu_metrics_${PBS_JOBID}.csv"

METRICS="index,timestamp,utilization.gpu,memory.used,memory.total"

nvidia-smi --query-gpu="$METRICS" \
           --format=csv -l 1 > "$LOG_FILE" &
MONITOR_PID=$!

cleanup() {
    echo "Stopping GPU monitor (PID: $MONITOR_PID)..."
    kill $MONITOR_PID
}
trap cleanup EXIT
```

Figure 4.2: HPC Script to monitor the NVIDIA-SMI stats

Next, we implemented a subprocess that queries and records various vLLM metrics every second, such as the total number of tokens, TTFT, TBT, and the KV cache usage percentage. Finally, we overrode the main LLM class in OpenHands to create a subclass called *LoggingLLM*. This subclass saves the execution trace of the run to a file (including input and output sizes, as well as token IDs) and is also responsible for triggering the L_2 capture for each request made to vLLM.

4.3 Validator

A critical component of this thesis is the implementation of the validator mechanism previously described. While the framework is modular and compatible with multiple datasets, it was specifically designed around the HumanEvalPlus dataset [25]. This project utilizes the two distinct test splits

from HumanEvalPlus: the original tests from HumanEval [8] are employed during the validation process, while the newer, more rigorous tests are reserved for assessing final accuracy.

The primary objective of this design is to provide the agent with an iterative improvement pattern analogous to human learning. As established in foundational educational theory by Bloom (1956), cognitive learning progresses through a specific taxonomy: Knowledge, Comprehension, Application, Analysis, Synthesis, and Evaluation [7, p.18]. A direct parallel can be drawn between this taxonomy and the proposed agentic framework: Knowledge resides within the base model, Comprehension occurs upon receiving the prompt, Application is the generation of a solution, Analysis and Synthesis are performed by the reviewer, and Evaluation is executed by the validator. The goal is to recreate a pedagogical framework that enables the agent to iteratively refine and improve its code across multiple cycles.

4.4 Custom Context Condenser

To optimize KV cache performance, we implemented an optional, custom context condenser. This mechanism evaluates L_2 norms on a turn-by-turn basis, where each conversational turn is divided into two segments: an Action and an Observation. To determine the retention value of these segments, we adapted the L_2 norm-based key cache compression strategy proposed by Devoto et al. (2024)[10], scaling their token-level findings to a segment-level eviction.

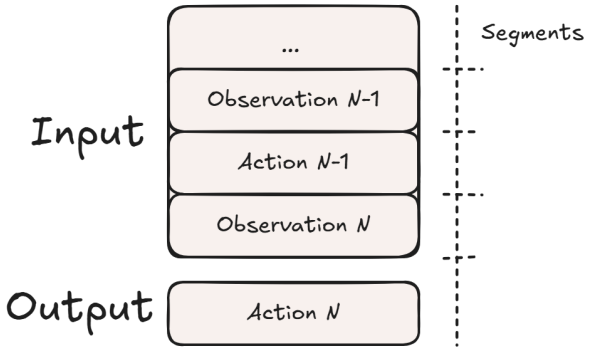

Figure 4.3: Condenser on segments

Devoto et al. demonstrated an inverse correlation between the L_2 norm of a key embedding and its resulting attention score; thus, tokens with lower L_2 norms are generally more critical for generation. Formally, let a segment S be composed of a sequence of tokens $\{t_1, t_2, \ldots, t_m\}$. For each token t_k, we compute the L_2 norm of its corresponding representation, denoted as $\|k_{t_k}\|_2$. We assume that the most significant token within a segment dictates the overall importance of that segment. Therefore, we define the segment-level significance score, v_S, as the minimum L_2 norm among its constituent tokens:

$$v_S = \min_{t_k \in S} \| k_{t_k} \|_2$$

When the condenser identifies a pool of segments, it compares their respective v_S scores. Eviction is performed on the segments that exhibit the largest v_S values. In other words, if the "most important" token of a segment is still relatively insignificant compared to the most important tokens of other segments, that entire segment is deemed non-essential and is targeted for eviction.

Once identified, these non-essential segments are batch-evicted simultaneously on both the client and server sides. This dual-sided eviction is strictly necessary; because the client natively transmits the entire accumulated context from the beginning of the conversation, the server would otherwise be forced to process redundant data. Finally, to minimize the computational overhead of recomputing the KV cache after a deletion, these evictions are executed so that they align as closely as possible with the existing block boundaries.

Chapter 5

Evaluation

The evaluation of this framework is structured around two primary objectives. First, we aim to demonstrate the system's functional effectiveness by measuring its generation accuracy and iterative improvement on the HumanEvalPlus benchmark. Second, we assess the underlying computational efficiency of the architecture. Specifically, we evaluate the custom context condenser to quantify its ability to optimize KV cache memory usage without compromising the quality of the generated outputs.

5.1 Setup

Our experiments were conducted on the NSCC HPC Aspire-2A cluster [6], utilizing the Qwen/Qwen3-Coder-30B-A3B-Instruct model [32]. The workload was distributed across two NVIDIA A100-SXM4 GPUs, providing a combined total of 80GB of High Bandwidth Memory.

To prevent out-of-memory errors and maintain a functional buffer, the vLLM memory utilization factor was configured to 0.9, allocating approximately 72GB for active computational use. The model weights account for 57GB of this allocation (28.5GB per GPU). Of the remaining 15GB, approximately 4GB (2GB per GPU) is consumed by execution overheads and optimizations, such as CUDA graphs. Consequently, the system yields a final, usable KV cache capacity of approximately 11GB (5.5GB per GPU).

5.2 Accuracy measurements

To establish a performance baseline against the public leaderboard, the agent was evaluated on the complete HumanEvalPlus dataset. The execution was configured with a batch size of 5, enabling

five concurrent threads to process distinct tasks in parallel via the shared vLLM server.

Out of the 164 total tasks composing the HumanEvalPlus benchmark, the agent successfully resolved 149 (90.85%). A task was strictly considered resolved only if it passed all associated unit tests, which range from 100 to 1,000 tests per task. Examining the granular, test-level statistics across all combined tasks, the generated code successfully passed 114'277 out of 124'253 individual test cases, resulting in a 91.97% strict test pass rate.

Furthermore, hardware telemetry collected throughout the evaluation demonstrates that utilizing tensor parallelism effectively and evenly distributed the computational workload across both GPUs for the duration of the dataset run.

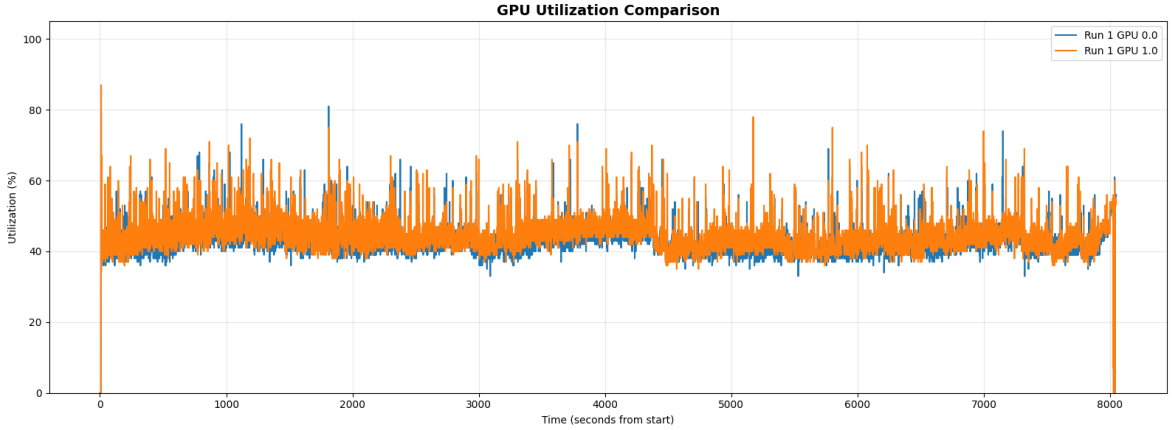

Figure 5.1: GPU Utilization

As discussed previously, TTFT represents the duration of the prefill phase, in which the model processes the entirety of the input prompt to compute the initial KV cache states. The recorded TTFT begins at 165ms and gradually stabilizes between 235ms and 240ms. Unlike the decoding phase, the prefill phase is fundamentally compute-bound and scales relative to the sequence length of the input prompt.

The gradual latency increase observed from 165ms to 240ms over the duration of the test corresponds to the system's ramp-up phase. This climb is attributed to the dynamic batching of five parallel requests. As these requests accumulate, the growing context of each sequence progressively consumes more GPU compute and KV cache resources.

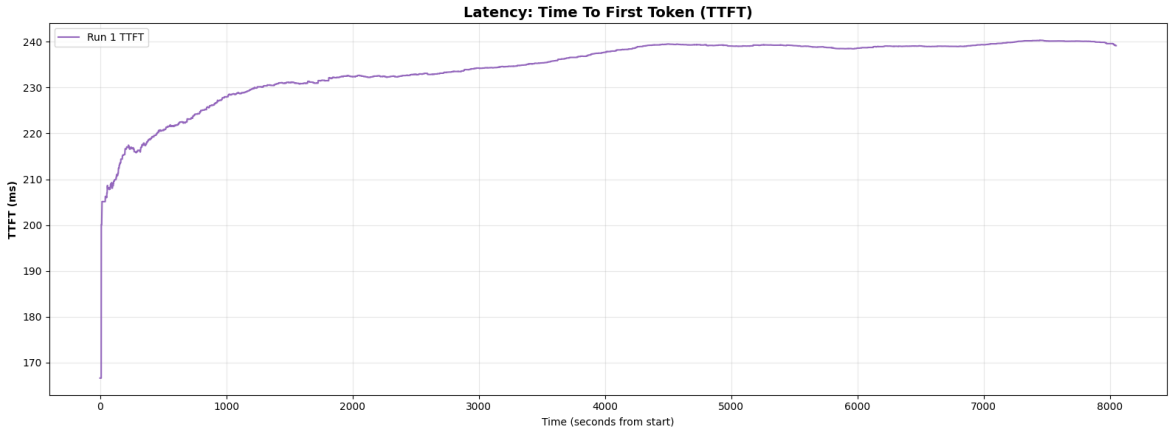

Figure 5.2: GPU Utilization

Inter-Token Latency (ITL), interchangeably referred to as TBT, measures the duration required to generate each subsequent token during the autoregressive decoding phase. As depicted in the empirical data, the ITL initiates at approximately 20ms and exhibits an asymptotic climb, ultimately plateauing near 80ms. At the onset of the workload, the system processes a single isolated request, reaching at the initialization a latency of 20ms per token. As the queue populates to the target batch size of five, the GPUs execute forward passes for multiple sequence states simultaneously.

System logs indicate that the global generation throughput stabilizes at approximately 62.5 tokens per second under full load. Assuming an even distribution of memory bandwidth and compute resources across the five concurrent requests, the theoretical per-request throughput is calculated as follows:

$$\text{Throughput}_{\text{user}} = \frac{62.5 \text{ tokens/s}}{5 \text{ concurrent requests}} = 12.5 \text{ tokens/s}$$

The corresponding ITL is the reciprocal of this per-request throughput:

$$\text{ITL} = \frac{1}{12.5 \text{ tokens/s}} = 0.080 \text{ seconds (80 ms)}$$

The empirical plateau at 80ms precisely mirrors this theoretical capacity. This confirms optimal memory bandwidth utilization across the tensor-parallel GPU configuration, indicating that the hardware is fully saturated without suffering from performance degradation or queuing bottlenecks.

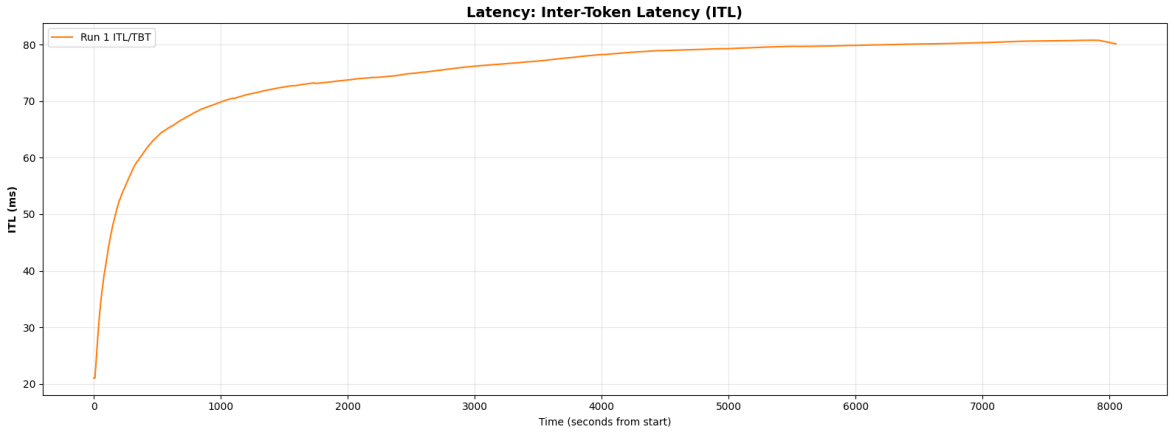

Figure 5.3: GPU Utilization

5.3 L_2 Norms Eviction

To evaluate the effectiveness of our L_2 norm-based KV Cache eviction approach, we want to assess whether the custom context condenser outperforms the standard baseline. For this evaluation, we curated a specific subset of 10 tasks from the HumanEvalPlus benchmark. We monitored several key metrics throughout the execution: KV cache utilization (both average and peak), execution time, inference throughput, and overall accuracy.

The eviction policy was configured as follows: after an initial accumulation of five context segments, each subsequent step evaluates the newly appended segment. We identify the segment exhibiting the largest minimal L_2 norm (utilizing the min-pooling technique described previously) as a candidate for eviction. Once a threshold of at least two—and occasionally three, depending on process synchronization within the OpenHands framework—eviction candidates is reached, the custom condenser is triggered. This process actively purges the identified segments from both the local context condenser and the KV cache on the vLLM server.

To establish a comparative analysis, we benchmarked our custom L_2 eviction mechanism against the unmodified baseline. The results are summarized in the following table.

Our findings demonstrate that the L_2 norm eviction strategy successfully optimizes memory footprint without degrading the quality of the generated code. Specifically, we achieved an 8.75% relative reduction in average KV cache usage and a 12.30% relative reduction in peak KV cache consumption while maintaining a perfect 100% accuracy on the evaluated subset. The primary trade-off observed was a runtime increase of approximately four minutes, which is largely attributable to the computational overhead and synchronization delays introduced by the active eviction process.

Metric	**Baseline**	L_2 **Norm Eviction**	**Impact**
Accuracy	100%	100%	No degradation
Throughput (avg.)	48.52 tokens/s	48.50 tokens/s	Negligible change
Avg. KV cache usage	9.60%	8.76%	8.75% relative reduction
Peak KV cache usage	18.21%	15.97%	12.30% relative reduction
Execution time	15 min 10 s	19 min 15 s	+4 min 5 s runtime

Table 5.1: Performance comparison between the baseline and the custom L2-norm eviction condenser.

Chapter 6

Future Work

Following this thesis work, we have identified several promising directions for future research to further enhance the efficiency and capabilities of our coding agent.

- **Advanced Inference Optimization:** Because agentic workflows naturally require maintaining extensive context histories across multiple iterations, future work will focus on further refining KV cache management. Optimizing how the context is stored and which part can be evicted will be key to improve further the metrics of our system. In addition it would be even more precise making our eviction mechanism aware of the type of the message and being able to evict repetitive or stale memory.
- **Native vLLM Integration:** Currently, our caching strategies are handled at the client and server level. To enhance the scalability and portability of our approach, we propose implementing these custom caching strategies directly into the vLLM backend. This native integration would make the optimizations client-agnostic, allowing any external system interfacing with the vLLM server to benefit from the reduced overhead.
- **Transition to Repository-Level Evaluation:** While our initial baselines and validations were successfully established using function-level datasets like HumanEvalPlus, real-world software engineering is much more complex. Future evaluations will transition to repository-level benchmarks, such as SWE-Bench[20], to test the agent's ability to navigate, understand, and resolve complex issues across more complex codebases.
- **Cross-Model Benchmarking:** To ensure our agentic blueprint is both robust and economically viable, we aim to conduct comprehensive cross-model evaluations. Assessing the agent's performance and cost-efficiency across a diverse range of open-source and proprietary foundation models will help identify the optimal trade-offs between computational cost and coding accuracy.

Chapter 7

Conclusion

This thesis project has been both challenging and highly rewarding. Prior to this semester, our experience with LLMs was strictly from an end-user perspective. The immersive six-month period spent in Singapore provided us with profound insights into the MLSys research environment, as well as the broader AI landscape in Asia.

We initiated this research period with an intensive learning phase, striving to comprehensively understand the trade-offs between the various available models and configurations. Concurrently, we began running initial models, applying multiple modifications to the standard setups. This was followed by the implementation of our first dataset to run experiments and attempt to reproduce established baselines. During this process, we analyzed existing systems to design a blueprint for an ideal coding agentic workflow. Once the blueprint was established, we developed our proposed coding agent based on this architecture. We continuously enhanced this agent by integrating new features, ranging from verification and batching to recent attempts at cache optimization.

Ultimately, this project has significantly expanded our knowledge of AI, Large Language Models, and agentic workflows. Approaching these complex topics through hands-on, project-based research proved to be highly engaging, and the resulting insights will be highly relevant given the rapidly growing importance of AI in the industry. Finally, we would like to express our sincere gratitude to all our supervisors and advisors for their guidance and assistance throughout the semester.

Bibliography

[1] *2025 Java Developer Productivity Report*. https://www.jrebel.com/resources/java-developer-productivity-report-2025. [Accessed 17-02-2026].

[2] Deepak Bhaskar Acharya, Karthigeyan Kuppan, and B. Divya. "Agentic AI: Autonomous Intelligence for Complex Goals—A Comprehensive Survey". In: *IEEE Access* 13 (2025), pp. 18912–18936. DOI: 10.1109/ACCESS.2025.3532853.

[3] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints". In: *The 2023 Conference on Empirical Methods in Natural Language Processing*. 2023. URL: https://openreview.net/forum?id=hmOwOZWzYE.

[4] Anthropic. *Claude Code*. https://claude.com/product/claude-code. Agentic AI coding tool. 2025.

[5] Anysphere. *Cursor: The AI Code Editor*. https://cursor.com/. AI-powered code editor. 2023.

[6] *ASPIRE 2A | NSCC*. https://www.nscc.sg/aspire-2a/. [Accessed 08-03-2026].

[7] Benjamin Samuel Bloom. *Taxonomy of educational objectives: Affective domain*. Vol. 2. Longmans, Green, 1956.

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

[9] CodeRabbit. *CodeRabbit: AI Code Reviews*. https://www.coderabbit.ai/. AI-powered code review platform. 2023.

[10] Alessio Devoto, Yu Zhao, Simone Scardapane, and Pasquale Minervini. "A Simple and Effective L_2 Norm-Based Strategy for KV Cache Compression". In: (2024). URL: https://arxiv.org/abs/2406.11430.

[11] Yogesh K. Dwivedi, Nir Kshetri, Laurie Hughes, Emma Louise Slade, Anand Jeyaraj, Arpan Kumar Kar, Abdullah M. Baabdullah, Alex Koohang, Vishnupriya Raghavan, Manju Ahuja, Hanaa Albanna, Mousa Ahmad Albashrawi, Adil S. Al-Busaidi, Janarthanan Balakrishnan, Yves Barlette, Sriparna Basu, Indranil Bose, Laurence Brooks, Dimitrios Buhalis, Lemuria Carter, Soumyadeb Chowdhury, Tom Crick, Scott W. Cunningham, Gareth H. Davies, Robert M. Davison, Rahul Dé, Denis Dennehy, Yanqing Duan, Rameshwar Dubey, Rohita Dwivedi, John S. Edwards, Carlos Flavián, Robin Gauld, Varun Grover, Mei-Chih Hu, Marijn Janssen, Paul Jones, Iris Junglas, Sangeeta Khorana, Sascha Kraus, Kai R. Larsen, Paul Latreille, Sven Laumer, F. Tegwen Malik, Abbas Mardani, Marcello Mariani, Sunil Mithas, Emmanuel Mogaji, Jeretta Horn Nord, Siobhan O'Connor, Fevzi Okumus, Margherita Pagani, Neeraj Pandey, Savvas Papagiannidis, Ilias O. Pappas, Nishith Pathak, Jan Pries-Heje, Ramakrishnan Raman, Nripendra P. Rana, Sven-Volker Rehm, Samuel Ribeiro-Navarrete, Alexander Richter, Frantz Rowe, Suprateek Sarker, Bernd Carsten Stahl, Manoj Kumar Tiwari, Wil van der Aalst, Viswanath Venkatesh, Giampaolo Viglia, Michael Wade, Paul Walton, Jochen Wirtz, and Ryan Wright. "Opinion Paper: "So what if ChatGPT wrote it?" Multidisciplinary perspectives on opportunities, challenges and implications of generative conversational AI for research, practice and policy". In: *International Journal of Information Management* 71 (2023), p. 102642. ISSN: 0268-4012. DOI: https://doi.org/10.1016/j.ijinfomgt.2023.102642. URL: https://www.sciencedirect.com/science/article/pii/S0268401223000233.

[12] Eclipse Foundation. *Eclipse IDE*. https://eclipseide.org/. Integrated Development Environment. 2001.

[13] FoundationAgents. *MetaGPT: The Multi-Agent Framework*. https://github.com/FoundationAgents/MetaGPT. Open-source multi-agent framework. 2023.

[14] David Geer. "Eclipse becomes the dominant Java IDE". In: *Computer* 38.7 (2005), pp. 16–18.

[15] GitHub. *GitHub Copilot: Your AI Pair Programmer*. https://github.com/features/copilot. AI-powered code completion tool. 2021.

[16] *Hello World - OpenHands Docs*. https://docs.openhands.dev/sdk/guides/hello-world#ready-to-run-example. [Accessed 07-03-2026].

[17] Reid Holmes and Gail C. Murphy. "Using structural context to recommend source code examples". In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO, USA: Association for Computing Machinery, 2005, pp. 117–125. ISBN: 1581139632. DOI: 10.1145/1062455.1062491. URL: https://doi.org/10.1145/1062455.1062491.

[18] *IntelliJ IDEA Dominates the IDE Market*. https://snyk.io/fr/blog/intellij-idea-dominates-the-ide-market-with-62-adoption-among-jvm-developers/. [Accessed 28-02-2026].

[19] JetBrains. *IntelliJ IDEA*. https://www.jetbrains.com/idea/. Integrated Development Environment. 2001.

[20] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. "SWE-bench: Can Language Models Resolve Real-world Github Issues?" In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=VTF8yNQM66.

[21] Tom Kilburn, David BG Edwards, Michael J Lanigan, and Frank H Sumner. "One-level storage system". In: *IRE Transactions on Electronic Computers*. Vol. 2. 1962, pp. 223–235.

[22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. Koblenz, Germany: Association for Computing Machinery, 2023, pp. 611–626. ISBN: 9798400702297. DOI: 10.1145/3600006.3613165. URL: https://doi.org/10.1145/3600006.3613165.

[23] Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, Shuyi Wang, Qunhong Zeng, Di Wang, Xuandong Zhao, Yuanli Wang, Roey Ben Chaim, Zonglin Di, Yipeng Gao, Junwei He, Yizhuo He, Liqiang Jing, Luyang Kong, Xin Lan, Jiachen Li, Songlin Li, Yijiang Li, Yueqian Lin, Xinyi Liu, Xuanqing Liu, Haoran Lyu, Ze Ma, Bowei Wang, Runhui Wang, Tianyu Wang, Wengao Ye, Yue Zhang, Hanwen Xing, Yiqi Xue, Steven Dillmann, and Han-chung Lee. *SkillsBench: Benchmarking How Well Agent Skills Work Across Diverse Tasks*. 2026. arXiv: 2602.12670 [cs.AI]. URL: https://arxiv.org/abs/2602.12670.

[24] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D Manning, Christopher Re, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue WANG, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri S. Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Andrew Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. "Holistic Evaluation of Language Models". In: *Transactions on Machine Learning Research* (2023). Featured Certification, Expert Certification, Outstanding Certification. ISSN: 2835-8856. URL: https://openreview.net/forum?id=iO4LZibEqW.

[25] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. “Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation”. In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023. URL: https://openreview.net/forum?id=1qvx610Cu7.

[26] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. *Self-Refine: Iterative Refinement with Self-Feedback*. 2023. arXiv: 2303.17651 [cs.CL].

[27] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. “Distributed Representations of Words and Phrases and their Compositionality”. In: *Advances in Neural Information Processing Systems*. Ed. by C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger. Vol. 26. Curran Associates, Inc., 2013. URL: https://proceedings.neurips.cc/paper_files/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf.

[28] Misty AI. *IT Training*. https://mistyai.com/it-training/. IT training resources.

[29] Piotr Nawrot, Adrian Łańcucki, Marcin Chochowski, David Tarjan, and Edoardo Ponti. “Dynamic Memory Compression: Retrofitting LLMs for Accelerated Inference”. In: *Forty-first International Conference on Machine Learning*. 2024. URL: https://openreview.net/forum?id=tDRYrAkOB7.

[30] OpenAI. *ChatGPT*. https://chatgpt.com/. Large language model. 2022.

[31] Sebastiano Panichella. “Vulnerabilities Introduced by LLMs Through Code Suggestions”. In: *Large Language Models in Cybersecurity: Threats, Exposure and Mitigation*. Ed. by Andrei Kucharavy, Octave Plancherel, Valentin Mulder, Alain Mermoud, and Vincent Lenders. Cham: Springer Nature Switzerland, 2024, pp. 87–97. ISBN: 978-3-031-54827-7. DOI: 10.1007/978-3-031-54827-7_9. URL: https://doi.org/10.1007/978-3-031-54827-7_9.

[32] *Qwen/Qwen3-Coder-30B-A3B-Instruct · Hugging Face*. https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct. [Accessed 08-03-2026].

[33] Rico Sennrich, Barry Haddow, and Alexandra Birch. “Neural Machine Translation of Rare Words with Subword Units”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. DOI: 10.18653/v1/P16-1162. URL: https://www.aclweb.org/anthology/P16-1162.

[34] Adam Smith. *The wealth of nations [1776]*. Vol. 11937.

[35] *Software Agent SDK - OpenHands Docs*. https://docs.openhands.dev/sdk. [Accessed 07-03-2026].

[36] *Tokenizer - OpenAI API*. https://platform.openai.com/tokenizer. [Accessed 10-03-2026].

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[38] vLLM Team. *vLLM: Easy, Fast, and Cheap LLM Serving for Everyone*. https://vllm.ai/. 2023.

[39] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. "Chain-of-thought prompting elicits reasoning in large language models". In: *Proceedings of the 36th International Conference on Neural Information Processing Systems*. NIPS '22. New Orleans, LA, USA: Curran Associates Inc., 2022. ISBN: 9781713871088.

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

[41] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering". In: *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. 2024. URL: https://arxiv.org/abs/2405.15793.

[42] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. "ReAct: Synergizing Reasoning and Acting in Language Models". In: *arXiv preprint arXiv:2210.03629* (2022).

[43] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. "A Survey of Large Language Models". In: *arXiv preprint arXiv:2303.18223* (2023). URL: http://arxiv.org/abs/2303.18223.

Appendix A

The Project Repository

A.1 Navigating the Project

The project is either available [on the lab Github] or [on my Github].

Here is a quick guide to the main folders in the repository:

- *agent_runner/*: Holds the core agent code, prompts, and the security analyzer.
- *l2norm/*: Contains the files for the L_2 norm service.
- *benchmarking/*: Stores the code used for tracking performance.
- *assets/*: Contains test tasks and problems for the agent (like the sorting tasks).
- *scripts/*: Holds extra helper scripts to run the project.
- *test/*: Contains the testing files.
- *docs/* and *appendix/*: Hold project documentation and extra thesis materials.

A.1.1 Threads and Background Tasks

The client code runs alongside a vLLM server and uses multiple threads to handle different tasks:

- Core Agent: The *main.py* file in the root folder is the starting point. It launches the *agent_runner*.

- Benchmarking: A separate background thread tracks performance by checking the vLLM metrics endpoint every second.
- L_2 Norm Service: This service uses async queues for its input and output. During each turn, it gets a chat completion ID from its input queue and collects the L_2 norms. It then checks if anything needs to be removed from the cache. If so, it adds that information to an output queue. The *agent_runner* condenser (which optimizes the KV cache, as discussed earlier) then reads from this queue.

A.1.2 vLLM Versions and Flags

The L_2 norm service is optional and only runs if you start the client with the *–l2evict* flag.

- Running with *–l2evict*: You must use a specific, custom version of vLLM. The container image is hosted on Docker Hub (https://hub.docker.com/r/broccolin/vllm-l2) by a PhD student from the HyScale Lab at NTU. You will need to use Singularity to pull and convert this Docker image to run it.
- Running normally (no flag): Without the flag, the project works perfectly fine with any standard vLLM version 0.10.2 or newer.

A.2 How to use the framework

This framework is designed to be flexible. You can easily modify or add to the code to fit the needs of your current experiments.

To use the framework, the dataset must be organized into a specific folder structure. The code expects two main folders: a problems folder for the algorithms, and a testing folder for the test cases.

Because all algorithms of the same type (like different sorting methods) use the exact same test files, your folders should look like this:

```
|-- problems
|   |-- sorting
|   |   |-- heap_sort
|   |   |   \-- heap_sort.py
|   |   \-- bubble_sort
|   |       \-- bubble_sort.py
|   \-- graph_traversal
|       \-- bfs
|           \-- bfs.py
\-- testing
    |-- sorting
    |   |-- input1.txt
    |   |-- output1.txt
    |   |-- input2.txt
    |   \-- output2.txt
    \-- graph_traversal
        |-- input1.txt
        \-- output1.txt
```

Figure A.1: Accepted Dataset Hierarchy

In this setup, the problems folder is broken down by the type of task, then by the specific algorithm, and finally contains the Python file. The testing folder mirrors the main task types and holds the shared input and output text files needed to check them.

Once you have crafted your dataset and your linked in with the framework. (using Github for instance), you simply have to start the vLLM server with a strong enough coding model. The recommended model for this framework is Qwen/Qwen3-Coder-30B-A3B-Instruct (available on Hugging Face). Then, once the server has started, you just have to run the main script with the flags that you want.