Department of Mathematics
TUM School of Computation, Information and Technology
Technical University of Munich

Implicit Context Condensation for Local Software Engineering Agents

Kirill Gelvan

Thesis for the attainment of the academic degree

Master of Science

at the TUM School of Computation, Information and Technology of the Technical University of Munich

Supervisor:
Prof. Dr. Gjergji Kasneci

Advisors:
Felix Steinbauer
Igor Slinko (JetBrains Research)

Submitted:
Munich, 30. November 2025

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Munich, 30. November 2025 Kirill Gelvan

Abstract

This thesis investigates implicit context condensation for large language model-based software engineering agents. Instead of relying on extended context windows or explicit methods, it compresses long observations into a fixed set of embeddings using an in-context LLM-based encoder attached to a frozen LLM decoder. The compressor is pretrained on general text with a mix of autoencoding and language modeling objectives and then fine-tuned on question answering, code reconstruction, or agentic trajectories derived from SWE-bench style issues. Experiments on SQuAD and RepoQA demonstrate that the condensed representations can match or even surpass the uncompressed baselines on single-shot tasks, preserving enough detail to reconstruct natural language and source code. In the agentic setting, context condensation enables trajectories with more steps within a fixed context budget and reduces per-step inference time, demonstrating clear efficiency gains. However, when the agent is applied to the end-to-end setting of SWE-bench Verified, these benefits do not lead to higher resolve rates and underperform an untrained baseline.

Zusammenfassung

Diese Arbeit untersucht die implizite Kontextkomprimierung für Software-Engineering-Agenten, die auf Large Language Models (LLMs) basieren. Anstatt sich auf erweiterte Kontextfenster oder explizite Methoden zu stützen, werden lange Observationen hierbei in ein festes Set von Embeddings komprimiert. Dies geschieht mithilfe eines In-Context-Encoders auf LLM-Basis, der an einen statischen („frozen") LLM Decoder gekoppelt ist. Der Kompressor wird zunächst auf allgemeinen Textdaten vortrainiert (Pretraining), wobei eine Mischung aus Autoencoding- und Sprachmodellierungs-Zielen zum Einsatz kommt. Anschließend erfolgt ein Fine-Tuning auf Question-Answering, Code-Rekonstruktion oder auf Trajektorien, die von Problemen im Stil des SWE-bench abgeleitet sind. Experimente mit SQuAD und RepoQA zeigen, dass die komprimierten Repräsentationen in Single-Shot-Aufgaben mit unkomprimierten Baselines mithalten oder diese sogar übertreffen können. Dabei bleiben genügend Details erhalten, um sowohl natürliche Sprache als auch Quellcode zu rekonstruieren. Beim Einsatz als Agent ermöglicht die Kontextkomprimierung längere Handlungssequenzen innerhalb eines festen Kontext-Budgets sowie eine reduzierte Inferenzzeit pro Schritt, was deutliche Effizienzgewinne demonstriert. In der End-to-End-Anwendung auf SWE-bench Verified führen diese Vorteile jedoch nicht zu einer höheren Lösungsrate; das System schneidet hierbei schlechter ab als eine untrainierte Baseline.

Contents

1 Introduction **1**
1.1 The Context Length Challenge in Large Language Models 1
1.1.1 Approaches to Extending Context 1
1.1.2 Research Questions 3
1.1.3 Thesis Outline 3

2 Background **5**
2.1 Transformer Architecture and Positional Bias 5
2.2 Parameter-Efficient LLM Fine-Tuning 6
2.3 Agentic Setup and Tool Use 6
2.4 Large Language Models for Code 7

3 Related Work **9**
3.1 Approaches to Context Management and Agentic Training 9
3.2 Synthesis and Positioning 12

4 Methodology **13**
4.1 Overview of the Training Process and Model Variants 13
4.2 ICAE for Agentic Context Management 14
4.3 Training Methodology for Agentic ICAE 16
4.3.1 Pretraining (PT) 17
4.3.2 Fine-Tuning (FT) 17
4.3.3 Disabling "Thinking" for Tool Use 17

5 Evaluation **19**
5.1 Datasets 19
5.1.1 RedPajama 19
5.1.2 SQuAD 19
5.1.3 RepoQA 19
5.1.4 SWE-bench 20
5.1.5 Agentic Trajectories for Fine-Tuning 20
5.2 Quality Metrics 21
5.2.1 Token-Level Metrics 21
5.2.2 Task-Level Metrics 21
5.2.3 Efficiency Metrics 22

6 Experiments **23**
6.1 Feasibility of Training-Free Context Condensation 23
6.2 Learning Projections for Context Condensation 24
6.3 ICAE Pretraining and Results on General Text Reconstruction 25
6.4 ICAE Fine-Tuning and Results on Question Answering 26
6.5 ICAE Fine-Tuning and Results on Code Reconstruction 28
6.6 ICAE Fine-Tuning and Results on SWE-bench Verified 29
6.6.1 Experimental Setup 29
6.6.2 Evaluation and Results 29
6.6.3 Impact on Inference Efficiency 31

6.6.4 Discussion of Performance on Agentic Tasks versus Standard NLP Tasks 32

7 Discussion **35**
7.1 Reconstruction Fidelity and Error Accumulation 35
7.2 Limitations of LoRA for Fine-Tuning Agentic Capabilities 35
7.3 Information Prioritization in Multi-Turn Interactions 36

8 Limitations and Future Work **37**
8.1 Limitations of Fixed-Length Context Condensation 37
8.2 Limitations of KV-caching 37
8.3 Constraints on Computational Resources 37
8.3.1 Model Scale 37
8.3.2 Full Fine-Tuning and the LoRA Bottleneck 37
8.3.3 Disabled Reasoning 38
8.4 Open Source Contributions and Reproducibility 38

9 Conclusion **39**
9.1 Summary of Contributions 39
9.2 Synthesis of Findings 39

A Appendix **41**
A.1 On the Use of AI 41
A.2 SWE-smith First User Message 41
A.3 SWE-smith Prompt 42
A.4 ICAE In-Context Ability Proof-of-Concept Example 43
A.5 Ablation Experiment: Disabling Thinking 44
A.6 Training Details and Hyperparameters 45
A.6.1 Pretraining Configuration 45
A.6.2 Fine-tuning Configuration 45
A.6.3 SWE-bench Inference Configuration 46
A.6.4 Profiling Setup 46

Bibliography **51**

1 Introduction

1.1 The Context Length Challenge in Large Language Models

The ability of Large Language Models (LLMs) to effectively process long sequences of input text is fundamentally constrained by their architecture. Specifically, Transformer-based LLMs face inherent limitations due to the self-attention mechanism, which scales quadratically with the number of tokens [Vas+17]. This quadratic complexity restricts the practical context length, posing a significant challenge for tasks that require extensive history or large documents.

The long context limitation is particularly severe in complex automated scenarios involving agents with many interaction turns. This restriction is worsened in software engineering (SWE) agent applications, where operational trajectories frequently involve tool calls that generate unnecessarily long outputs (environment observations) [Jim+23; Yan+24]. SWE agents must perform tasks such as examining files and directories, reading and modifying parts of files, and navigating complex codebases [Yan+25b]. However, pretrained models literally cannot work with sequences longer than N (e.g., 32,000) tokens, which prevents them from efficiently processing the accumulated history generated by these tools. This is a significant problem, as the history of interactions is crucial for the agent to perform the task correctly.

1.1.1 Approaches to Extending Context

To address this challenge, several strategies have been developed. We categorize them into four main groups: architectural innovations, extended context windows, explicit compression, and implicit compression.

Architectural Innovations. One line of research focuses on modifying the self-attention mechanism to reduce its computational complexity. Sparse and local/windowed attention patterns reduce pairwise interactions to achieve sub-quadratic cost. For instance, Longformer combines sliding windows with global tokens to handle long documents [BPC20], while BigBird employs block- and mixed-sparsity patterns for similar gains [Zah+20]. Linear-time approximations, such as kernelized attention or Linformer's projection method [Wan+20], offer further asymptotic improvements.

While these methods offer benefits such as reduced memory footprint, lower computational cost, and effective modeling of local dependencies, they come with notable trade-offs. Sparse attention patterns can fail to properly route information across distant parts of the sequence when global tokens or connectivity patterns are insufficient [Zhu+23]. Additionally, irregular sparsity patterns often lead to hardware inefficiencies, as modern accelerators are optimized for dense operations [Dao+22]. Most critically, these architectural innovations struggle to overcome a notable decline in performance on long contexts, as they may fail to capture critical long-range dependencies that full attention would preserve.

A more recent line of research explores architectures that, while related to Transformers, offer fundamentally different scaling properties. Recent work has highlighted the connection between Transformers and State Space Models (SSMs) [GGR21], a class of models inspired by control theory that can be viewed as a form of recurrent neural network (RNN) [DG24]. Architectures like Mamba-2, which builds on this duality, have demonstrated performance competitive with state-of-the-art Transformers on language modeling tasks, especially for very long sequences (e.g., beyond 32,000 tokens), while being significantly faster during inference. This direction suggests that the future of long-context modeling may lie in hybrid architectures or even a return to modernized recurrent models that avoid the quadratic bottleneck of self-attention.

Extended Context Windows. A more direct approach is to leverage newer models that are architecturally designed for very long contexts, often extending to one or two million tokens. Many of these models utilize advancements like Rotary Position Embeddings (RoPE) [Su+24] to better handle long-range dependencies. While this seems like a straightforward solution, it comes with its own set of drawbacks. Processing extremely long contexts, even with linear-scaling attention mechanisms, is computationally expensive and memory-intensive, resulting in high inference times and costs. Furthermore, models with large context windows can suffer from the "lost in the middle" problem, where they struggle to effectively utilize information from the middle of a long input sequence [Liu+24b].

Explicit Compression. Another approach is to explicitly compress the context before it is fed to the LLM. This can be done through methods like retrieval-augmented generation (RAG), which selects relevant passages from a larger corpus [Lew+20]. A prominent example is the Retrieval-Enhanced Transformer (RETRO), which conditions on retrieved documents to significantly improve language modeling performance [Bor+22]. The main advantage is a controllable computational cost and the ability to incorporate external knowledge. However, these methods can suffer from selection bias, retrieval errors, and the potential loss of crucial details during the summarization or retrieval process, which can be harmful for downstream tasks that require high fidelity (e.g., code-related tasks).

Implicit Context Condensation. This thesis focuses on a fourth approach: implicit context condensation. This paradigm moves beyond explicit, token-based techniques by utilizing the inherent density of continuous latent spaces. Instead of relying on discrete representations (tokens), implicit compression focuses on mapping information into a compact set of continuous representations (embeddings). The core idea is that a text can be represented in different lengths and densities within an LLM while conveying the same essential information [Che+23]. The goal is to produce task-adapted representations that a model uses during inference, rather than the raw input itself. This approach promises several advantages: it maintains a tight interface with the model, can improve generation speed, and avoids external retrieval steps. An idea of applying this approach to the agentic setting is illustrated in Figure 1.1 below.

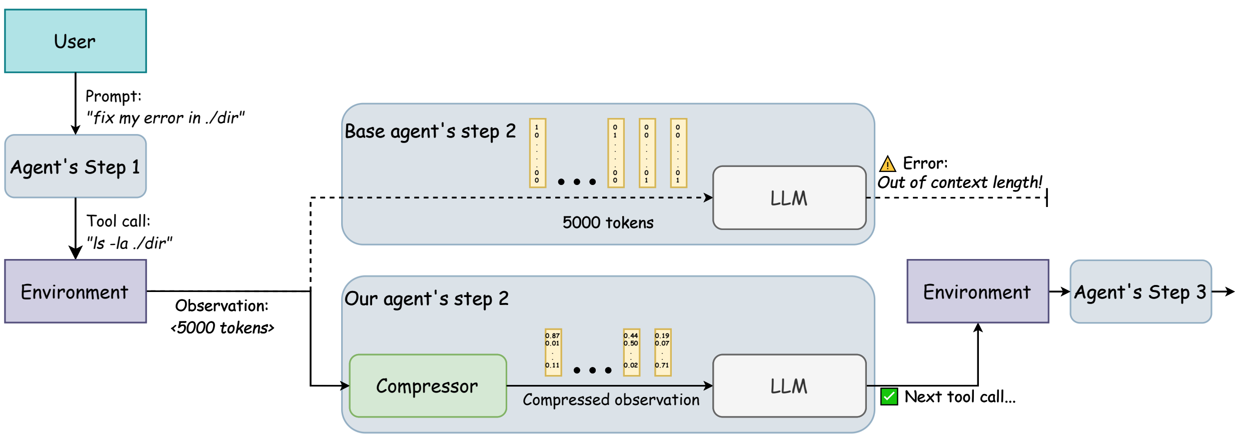


Figure 1.1 Comparison between a base agent and our approach for handling long observations exceeding the LLM's context length. The base agent fails when processing long observations directly, while our agent successfully compresses the observation into a fixed set of embeddings before LLM processing, enabling continued task execution.

The primary goal of context condensation is to leverage this potential density, enabling LLM agents to execute tasks involving long chains of reasoning (Chain-of-Thought, CoT) and more steps by condensing the environment observations. Achieving this condensation improves the model's capability to handle long contexts while offering tangible advantages in improved generation speed during inference.

1.1.2 Research Questions

The exploration of implicit context condensation as a solution to the long-context problem leads to the central research questions of this thesis. The first question addresses the impact on efficiency, here defined by the support for longer operational trajectories and the speed of generation:

RQ1: *How does implicit context condensation influence the efficiency of LLM-based agents when applied to software engineering tasks?*

A secondary question explores whether the results of this approach on general texts transfer to the domains of coding and agentic software engineering:

RQ2: *How does the performance of implicit context condensation on standard NLP benchmarks transfer to software engineering tasks, both single-shot and agentic?*

1.1.3 Thesis Outline

The **Introduction** begins by establishing the context length challenge in LLMs and introducing implicit context condensation as a potential solution, leading to the formulation of the core research questions. Following this, the **Background** provides the necessary foundation, with a focus on Transformer encoding and positional bias, the mechanics of Low Rank Adaptation (LoRA), and the operational principles of agentic workflows. The subsequent **Related Work** situates this thesis within the existing literature, offering an analysis of prior approaches to context management, including both explicit and implicit compression variants. The core method is detailed in the **Methodology**, which explains the adaptation of the In-Context Autoencoder (ICAE) for the agentic setting. Next, the **Evaluation** chapter outlines the datasets used (SWE-bench Verified, SQuAD, and RepoQA) and the metrics for assessment, including task-level success rates and token-level metrics (BLEU, F1 score). The results of this study are presented in the **Experiments** chapter, which presents the outcomes of experiments on text and code reconstruction, QA task, and SWE-bench Verified. A critical analysis of the factors contributing to these outcomes is provided in the **Discussion**. The thesis then addresses its **Limitations and Future Work**, which includes the constraints of fixed-length condensation, the KV-caching and the potential bottleneck of LoRA-based fine-tuning. Finally, the **Conclusion** summarizes the key contributions of the thesis. The results confirm that the method enables longer agentic trajectories and reduces inference time (RQ1). Regarding the transferability of performance (RQ2), the findings show effective transfer to single-shot software engineering tasks, whereas a reduction in quality is observed in the agentic setup.

2 Background

This chapter builds the foundation for the thesis's core methodology. It begins with Transformer architecture, as our compression method directly manipulates positional signals to manage context. Next, it explains Parameter-Efficient Fine-Tuning (LoRA), which is the technique used to train our In-Context Autoencoder. The agentic framework is then detailed, establishing the complex, multi-step problem domain we aim to solve. Finally, we discuss Large Language Models for code, the specific application area for our experiments. These sections collectively explain the prerequisite concepts needed to understand our novel approach to context compression.

2.1 Transformer Architecture and Positional Bias

The self-attention mechanism is well-known and has been described in detail in numerous articles. Thus, we focus on how positional signals influence which tokens are likely to interact, and why the layout of position identifiers (position IDs) matters for context compression.

Transformers are permutation-invariant over their inputs unless augmented with positional information. In the original formulation, absolute position encodings [Vas+17] (either fixed sinusoidal or learned) are added to token embeddings so that attention has access to token order. Subsequent works replace addition with position-dependent biases or transformations that make attention explicitly sensitive to token distances:

- Relative position representations add an index-dependent bias to the attention logits [SUV18].
- Rotary position embeddings (RoPE) apply a rotation to queries and keys so that their inner product becomes a function of relative displacement [Su+24].

Formally, for queries Q, keys K, and values V, attention often takes the form

$$\mathrm{Attn}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top + B}{\sqrt{d_k}}\right)V,$$

where B is a position-dependent bias. In relative schemes, $B_{ij} = b(i - j)$.

In RoPE, each query and key vector is multiplied element-wise by a rotation matrix that depends on its absolute position. However, the resulting dot product $\langle Q_i, K_j\rangle$ depends only on the relative distance $i - j$. Specifically, RoPE applies a rotation to the query and key embeddings:

$$Q_i = R_i q_i, \quad K_j = R_j k_j,$$

where R_m is a rotation matrix parameterized by position m. The key property is that the inner product becomes

$$\langle Q_i, K_j\rangle = \langle R_i q_i, R_j k_j\rangle = \langle q_i, R_{j-i} k_j\rangle,$$

which depends only on the relative offset $j - i$. This is achieved by constructing R_m as a block-diagonal matrix of 2D rotations with frequencies that decrease geometrically across dimensions, allowing the model to capture both short- and long-range dependencies through different frequency components [Su+24].

These mechanisms create a local inductive bias: tokens that are closer in position tend to have higher prior attention affinity. This has direct implications for compression with special tokens ("memory", or compressed tokens): where those tokens are placed in position-ID space controls which parts of the sequence they can most easily interact with. Empirically, assigning position IDs to minimize distance between compressed tokens and the tokens they must interface with (either source content or the downstream prompt) improves effectiveness [Zha+24]. To implement this, we adopt a strategy where the position IDs of the memory tokens are manually assigned to be adjacent to the target context, effectively "teleporting" the compressed history subsequent to the current generation step. This minimizes the relative distance perceived by the attention mechanism (especially with RoPE), ensuring that the model can attend to the compressed memory as strongly as it attends to immediate local context.

2.2 Parameter-Efficient LLM Fine-Tuning

There are many techniques to efficiently fine-tune LLMs, but we focus on Low-Rank Adaptation (LoRA) [Hu+22]. LoRA freezes the pretrained weight matrices and introduces trainable low-rank updates, yielding substantial parameter savings while preserving the base model's knowledge [Hu+22]. Consider a linear projection with base weight $W_0 \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$. LoRA parameterizes an additive update

$$\Delta W = BA, \quad A \in \mathbb{R}^{r \times d_{\text{in}}},\ B \in \mathbb{R}^{d_{\text{out}} \times r},\ r \ll \min(d_{\text{in}}, d_{\text{out}}),$$

so that the adapted layer computes

$$h = (W_0 + \alpha/r \cdot BA)\, x = W_0 x + \alpha/r \cdot B(Ax),$$

with a scalar scaling α controlling the update magnitude. Only A and B are trained; W_0 remains frozen. The trainable parameter count becomes $r(d_{\text{in}} + d_{\text{out}})$, dramatically less than $d_{\text{in}}\, d_{\text{out}}$ for typical ranks (e.g., tens to a few hundreds).

In Transformer blocks, LoRA adapters are commonly inserted in attention projections (query and value projections, see [Hu+22]) and optionally in output or feed-forward projections, trading off capacity and efficiency. The benefits include:

- parameter efficiency and reduced activation memory
- modularity—multiple task-specific adapters can be swapped on top of a single base model
- faster fine-tuning

Practical considerations include choosing the rank r, the scaling α, as well as target layers to balance adaptation capacity and generalization [Hu+22]. It is worth noting that while LoRA is a standard and effective technique for many NLP tasks, its efficacy specifically for long-horizon agentic planning—where models must maintain robust reasoning over many turns—is less explored and potentially more constrained than full-parameter fine-tuning.

2.3 Agentic Setup and Tool Use

We use "agentic" to refer to autonomous decision-making loops in which an LLM plans, invokes tools, and incorporates observations to pursue goals. This paradigm is formalized in the ReAct (Reasoning and Acting) framework [Yao+22], which interleaves reasoning traces with action execution. At time t, the agent conditions on a history $H_t = [(a_1, o_1), \ldots, (a_{t-1}, o_{t-1})]$ and a task-specific system prompt s to select an action a_t. The environment (or tool) returns an observation o_t, which is appended to the history. This action-observation loop continues until termination. Concretely:

1. Prompt assembly: system instructions + task + concise guidelines.
2. Model step: the LLM proposes an action (a tool to invoke and parameters if required) and an optional justification for the action (reasoning).

3. Tool execution: the specified tool is executed non-interactively with provided arguments.

4. Observation: tool output (e.g. stdout/err, JSON, file diffs, retrieval snippets, etc.) is captured.

5. History update: (a_t, o_t) is logged to H_t.

6. Termination or next step: the agent either returns a final answer or continues the loop.

A prominent benchmark for evaluating such agentic capabilities is the Berkeley Function Calling Leaderboard (BFCL) [Pat+]. BFCL provides a standardized suite of tasks to measure an LLM's proficiency in translating natural language requests into precise, executable tool calls. The tasks range from simple, single-function invocations to complex scenarios requiring multi-step reasoning and tool chaining. This benchmark exemplifies the practical challenges in agentic systems, where models must correctly interpret user intent and interact with external APIs or codebases [Pat+].

Agentic tool call examples (from BFCL[Pat+]):

- Vehicle status lookup: `vehicle.getStatus(vin="WVWZZZ...")` — returns battery level, tire pressure, and last seen location.

- Driving route plan: `maps.route(origin="Munich", dest="Berlin", mode="driving")` — computes ETA and step-by-step directions.

- Hotel search: `booking.search(city="Prague", dates="2025-11-03..05", guests=2)` — lists options with price and rating.

- Weather check: `weather.current(city="Warsaw")` — returns temperature, precipitation, and alerts.

Code-oriented tool call examples

- Code/bash execution: `execute(command="pytest -q")` or `execute(command="ls -la")` — runs unit tests using pytest or lists files with details.

- Symbol search: `find(name="parseUser")` — finds the definition and usages of a function in the codebase.

- String replacement in code: `str_replace(file_path="app.py", search="foo", replace="bar")` — replaces all occurrences of "foo" with "bar" in app.py.

2.4 Large Language Models for Code

Large Language Models have demonstrated significant capabilities in understanding, generating, and manipulating source code [Aus+21; Che21]. This proficiency stems from their training on vast corpora of publicly available code, which allows them to learn the syntax, semantics, and common patterns of various programming languages. For an LLM, code is treated as another form of structured text, and the same sequence modeling principles that apply to natural language can be adapted for software.

The core capabilities of LLMs in the context of software engineering include:

- **Code Completion:** Suggesting completions for partially written lines of code, similar to IDE autocompletion but often with more context-awareness [Che21].

- **Code Simplification:** Refactoring code to be more readable/efficient or shorter without changing its functionality [Wan+24].

- **Code Translation:** Migrating code from one programming language to another (e.g., Python to JavaScript) [Wan+21].

- **Bug Detection and Repair:** Identifying potential bugs in a piece of code and suggesting fixes [Jim+23].
- **Code Editing:** Modifying existing code based on high-level instruction [Yan+24].
- **Code Documentation and Summarization:** Generating natural language descriptions from code, such as docstrings or summaries, and answering questions about its functionality [Fen+20].
- **Agentic Code Tasks:** Autonomously planning and executing complex, multi-step software engineering tasks [Zen+24].

These capabilities are often evaluated on benchmarks such as HumanEval [Che21] and MBPP (Mostly Basic Python Programming) [Aus+21], which test a model's ability to generate functionally correct code from specifications.

The application of LLMs to coding tasks has led to the development of powerful developer tools, such as GitHub Copilot [Git21], which are powered by models like OpenAI's Codex [Che21]. These tools act as AI pair programmers, assisting developers and increasing their productivity. In the context of agentic systems, the ability to generate and understand code is fundamental for building autonomous agents that can interact with software environments, execute commands, and solve complex software engineering tasks.

3 Related Work

Here we situate our work within the broader landscape of context management for large language models, with a specific focus on techniques relevant to code-oriented agentic tasks. We review several distinct lines of research to clarify our contribution and justify our methodological choices. The reviewed approaches can be broadly categorized into: 1) implicit, embedding-space context compression; 2) explicit, token-level context reduction; 3) continuous representations for reasoning; 4) architectural modifications for long-context memory; and 5) post-training on agent trajectories.

3.1 Approaches to Context Management and Agentic Training

In-Context Autoencoder (ICAE) As an approach to implicit context compression, the In-Context Autoencoder (ICAE) introduced by Ge et al. [Ge+23] is an encoder–decoder scheme that compresses an input context into a fixed number of "memory slots" (continuous tokens) and conditions the base LLM on these slots instead of the original prompt. The number of these memory slots is a key hyperparameter that must be determined prior to pretraining. To handle inputs that exceed its training context length, ICAE employs a chunking strategy: the long context is segmented, each chunk is independently compressed into a span of memory slots, and these spans are then concatenated. In pretraining, the encoder produces k memory tokens from a longer input, and the decoder is trained to reconstruct the original text; at inference, downstream tasks are solved by attending to the memory tokens rather than the raw prompt. The paper studies both pretrained ICAE (autoencoding + language-modeling objective) and fine-tuned ICAE for instruction-following, showing that memory tokens serve as a compact, trainable context representation. The authors also analyze when and why compression degrades, e.g., over-4× ratios become challenging, and how stronger base models tolerate higher compression.

With 4× compression, the pretrained ICAE achieves a score of $\approx$ 99% on autoencoding, as measured by the Bilingual Evaluation Understudy (BLEU) metric [Pap+02], across Llama-2 [Tou+23] models and slight increases in perplexity at continuation time (e.g., PPL 9.01⟶9.50), indicating near-lossless retention at 4× on natural text. When applied to instruction-following tasks, ICAE memory tokens are competitive with or stronger than baselines that read full ~512-token contexts, e.g., Llama-7B (ICAE) vs Alpaca: 73.1% win+tie. Moreover, this compression leads to significant speed improvements, with measured end-to-end speedups reaching 2.2–3.6× in cacheable regimes where memory slots are pre-computed.

ICAE is the most direct prior for our implicit compression: we also encode contexts into continuous embeddings and condition the model on the learned memory tokens, but extend the setting to newer backbones and coding/agentic workloads (SWE-style trajectories). We also emphasize agent-trajectory fine-tuning over SWE-bench-like tasks, which Ge et al. [Ge+23] did not target. For later chapters, this section providesthe technical background (slots, compression ratios, and fidelity–throughput trade-offs) that we build upon.

LLMLingua-2 Pan et al. [Pan+24] introduce LLMLingua-2, a task-agnostic method for explicit prompt compression. It moves beyond entropy-based pruning by training a dedicated token classifier to identify and discard redundant tokens. This compressor—a small, efficient Transformer encoder, similar to XLM-RoBERTa—learns from a dataset created via data distillation from GPT-4, making the approach model-agnostic and applicable to black-box LLMs. By leveraging bidirectional context, it can more accurately assess the importance of tokens than causal models.

The method demonstrates impressive performance, achieving substantial compression with minimal loss of fidelity. On the in-domain MeetingBank benchmark, Pan et al. [Pan+24] achieve 3.1× compression (from 3,003 to 970 tokens) while scoring 86.9% EM on a QA task, nearly matching the original prompt's 87.8%. For reasoning, they achieve 79.1% EM on GSM8K [Cob+21] with 5× compression, on par with the full uncompressed baseline. This efficiency translates to significant end-to-end speedups of 1.6× to 2.9×. Notably, when paired with Mistral-7B, the compressed prompt outperforms the original on QA (76.2% vs. 67.0%), suggesting it can help models that are less adept at handling long contexts.

The approach of Pan et al. [Pan+24], however, is fundamentally extractive and therefore lossy, which contrasts with the implicit, continuous compression (which theoretically can be lossless). Because it operates by deleting surface tokens, it risks removing subtle syntactic or semantic details that are critical in code generation and repair. Furthermore, its evaluation focuses on single-turn NLP tasks, sucg as QA and summarization, whereas our work targets the distinct challenges of multi-turn, agentic software engineering trajectories.

SlimCode Wang et al. [Wan+24] propose SlimCode, an explicit and model-agnostic method that simplifies code by removing tokens based on their intrinsic properties rather than model-specific attention scores. The approach categorizes code tokens by lexical (e.g., symbols, identifiers), syntactic (e.g., control structures, method signatures), and semantic levels. Through empirical analysis, the authors establish a token importance hierarchy, where method signatures and identifiers are most critical, while symbols are least impactful. Based on this ranking, a 0-1 knapsack-style algorithm is used to discard the lowest-value tokens, aiming to reduce input length while preserving semantic integrity.

The method's effectiveness is demonstrated by its ability to significantly reduce computational load with minimal performance degradation. For instance, removing symbol tokens, which constitute approximately 51% of the code, reduces training time by a similar margin but only lowers code search performance (Mean Reciprocal Rank, MRR) by 2.83% and summarization (BLEU-4) by 0.59%. In contrast, removing identifiers, which make up only 15.5% of tokens, results in a more substantial performance drop of 12.5% in MRR. Overall, Wang et al. [Wan+24] outperform the prior state-of-the-art, DietCode, by 9.46% on MRR and 5.15% on BLEU, while being up to 133 times faster at the pruning process itself. Furthermore, when applied to GPT-4, SlimCode can reduce API costs by 24% and inference time by 27%, and can even yield slight performance improvements at high compression ratios.

While Wang et al. [Wan+24] offer a robust, model-agnostic solution for code understanding tasks and cost reduction, their explicit, token-deleting nature makes it less suitable for our focus on agentic coding. By deleting surface tokens, this approach risks removing subtle syntactic or semantic constraints that are crucial for complex, multi-turn code generation and repair tasks. Our work instead focuses on implicit compression in the embedding space, trained end-to-end on coding trajectories to better align with the demands of generative and tool-use scenarios.

Soft Tokens Another line of research uses continuous representations for reasoning. For instance, Butt et al. [But+25] introduce a scalable method to train models on continuous or "soft" chain-of-thought (CoT) trajectories using reinforcement learning (RL). Their approach avoids costly distillation from discrete CoTs by injecting noise into input embeddings, which enables exploration and allows for learning long continuous thought vectors. The work compares this soft/fuzzy training against traditional hard-token CoT across Llama and Qwen models on mathematical reasoning benchmarks like GSM8K and MATH, studying both performance and out-of-distribution (OOD) robustness.

The results demonstrate that continuous CoT training is highly effective. On benchmarks like GSM8K, models trained with soft tokens achieve parity with discrete training on pass@1 accuracy (e.g., 77.2% for a soft-trained Llama-3B vs. 75.9% for a hard-trained one) while significantly improving pass@32 scores (97.9% vs. 94.1%). This suggests that continuous training encourages a greater diversity of valid reasoning paths. A key operational takeaway is that the best performance is consistently achieved by using standard "hard" (discrete) decoding at inference time, even for models trained with continuous tokens. This allows practitioners to benefit from soft training without altering existing deployment pipelines.

Furthermore, the method provides a "softer touch" to fine-tuning, better preserving the base model's capabilities on OOD tasks. While hard-token training can degrade a model's general knowledge (as measured by NLL on benchmarks like HellaSwag [Zel+19], ARC [Cla+18], and MMLU [Hen+20]), soft-token training maintains it. A striking example is the Llama-8B model trained on GSM8K: when tested on the MATH dataset, the hard-trained model's performance collapses (20.2% pass@1), whereas the soft-trained model generalizes well, recovering to 44.7% pass@1.

While this work focuses on reasoning traces rather than context compression, it is conceptually adjacent to our research as both approaches embed complex information into learned continuous vectors. The authors show that these continuous representations can be scaled to hundreds of tokens and trained stably with RL. Our work applies a similar principle, but at the level of context compression rather than thought-level reasoning. We then fine-tune on agentic coding trajectories, a different domain from the mathematical tasks studied in their work. Nonetheless, we build on the shared finding that continuous latent representations can be effectively trained and then decoded using standard hard inference, a principle that also holds for our memory tokens.

Infini-Attention As an architectural modification for long contexts, Infini-attention [MFG24] equips transformers with a compressive memory pathway that summarizes long-range keys/values while keeping local attention intact. The resulting Infini-Transformer aims to maintain useful information over very long contexts by updating a compact memory state at each segment, achieving a theoretically infinite context window without quadratic growth. The mechanism combines a standard local dot-product attention with a long-term compressive memory that is updated incrementally using a linear attention mechanism. A learned gating scalar then mixes the outputs of the local attention and the compressive memory, allowing the model to balance between short-term and long-term context.

On long-context language modeling benchmarks, such as PG19 [Rae+19], the Infini-Transformer outperforms Transformer-XL and Memorizing Transformers. It achieves this while using 114× fewer memory parameters than a baseline with a 65K-length vector-retrieval memory. Performance further improves when trained on sequences up to 100K tokens, with perplexity on ArXiv-math dropping to approximately 2.20.

The model demonstrates remarkable long-context capabilities, achieving near-perfect passkey retrieval at sequence lengths of 1 million tokens and setting a new state-of-the-art on a 500K-token book summarization task. These gains, however, require modifying the model's architecture and either pretraining from scratch or undergoing extensive continued pretraining. Our approach, in contrast, remains compatible with existing LLMs by using ICAE-style memory tokens. We focus on fine-tuning these representations for agentic coding tasks, whereas Munkhdalai, Faruqui, and Gopal [MFG24] offer a complementary solution for scenarios where processing raw, ultra-long contexts is indispensable.

AgentTuning To improve the agentic capabilities of open-source models via supervision, Zeng et al. [Zen+24] propose a method centered on fine-tuning LLMs with a specialized dataset of agent interaction trajectories. Their core contribution is the creation of AgentInstruct, a high-quality dataset of 1,866 interaction trajectories from six diverse agent tasks, generated and verified using GPT-4. The authors then employ a hybrid instruction-tuning strategy, mixing the agent-specific data with general-domain instructions to create the AgentLM series of models, based on Llama-2.

This approach yields substantial improvements in agent performance without degrading general capabilities. The resulting AgentLM-70B model achieves performance comparable to GPT-3.5 on unseen agent tasks, demonstrating an improvement of up to 176% over the base Llama-2-chat model on held-out tasks. A key finding from their ablation studies is that general-domain data is crucial for generalization; training exclusively on agent trajectories improves performance on seen tasks but fails to generalize to new ones. The work suggests that this fine-tuning process helps to "activate" latent agent capabilities in the base model rather than merely overfitting to specific tasks.

We similarly fine-tune on stepwise trajectories, but our work differs in its focus and domain. Zeng et al. [Zen+24] aim to improve agent behavior but do not address the challenge of long-context compression for complex code-related tasks. Our method integrates this concept of trajectory-based fine-tuning with an ICAE-style memory system, specifically targeting the bottlenecks of long code contexts and multi-turn interactions found in software engineering scenarios.

3.2 Synthesis and Positioning

Synthesis of Prior Work The reviewed literature on long-context management for large language models reveals several distinct methodologies. Explicit, token-level compression methods, such as LLMLingua-2 [Pan+24] and SlimCode [Wan+24], operate by removing tokens from the input, which provides model-agnostic efficiency but risks the loss of critical information for generative tasks. In contrast, architectural modifications, such as Infini-attention [MFG24], redesign the Transformer model itself to process theoretically infinite context streams, although this requires resource-intensive pretraining from scratch. Another line of research investigates implicit compression using continuous representations, where techniques like the In-Context Autoencoder (ICAE) [Ge+23] and soft reasoning traces [But+25] learn to encode information into dense vectors without altering the base model architecture. Finally, methods such as AgentTuning [Zen+24] focus on improving agentic capabilities through fine-tuning on interaction trajectories, but do not directly address the long-context limitation.

Rationale for Methodological Choices. Our decision to pursue an ICAE-based approach over alternatives is deliberate. We avoid explicit token deletion [Pan+24; Wan+24] because agentic code generation is highly sensitive to subtle syntactic and semantic details that pruning may inadvertently remove. While effective for code understanding, such methods are less suitable for iterative code repair. We also avoid architectural modifications [MFG24] to ensure our solution remains compatible with a wide range of existing, publicly available LLMs, thereby maximizing applicability and leveraging the benefits of KV-caching for the compressed memory tokens. Our method therefore pairs the context-handling efficiency of ICAE with the demonstrated effectiveness of trajectory fine-tuning to create an agent optimized for long-context software engineering challenges.

4 Methodology

This chapter presents the comprehensive methodology for developing and evaluating an In-Context Autoencoder (ICAE) designed for agentic context management. The chapter begins by providing a high-level overview of the entire training pipeline, illustrating how our proposed model and its variants are derived for comparative analysis. Following this, we dive into the core architectural principles of applying the ICAE framework to compress the conversational history of agentic trajectories. The subsequent sections detail the two-stage training process, which begins with a self-supervised pretraining phase on a large-scale text corpus to teach the model effective compression, followed by a fine-tuning phase on a specialized dataset of agentic trajectories to adapt the model for the downstream task.

As an example, we would use the following trajectory (you can also see Figure 4.3 for a basic visual representation of the trajectory):

1. **System prompt** (text): initial instructions and tool descriptions (Appendix A.3).
2. **Task description** (text): user-provided issue or goal (Appendix A.2).
3. **Action 1** (text): e.g., `bash: ls -la`.
4. **Observation 1** (short text, < 256 tokens): directory listing, kept as-is.
5. **Action 2** (text): e.g., `str_replace_editor: view file.py`.
6. **Observation 2** (long text): entire file content, compressed into memory tokens.
7. **Action 3** (text): e.g., `str_replace_editor: str_replace ...`.
8. **Observation 3** (long text): edit confirmation with context, again compressed into memory tokens.
9. **Action 4** (text): e.g., `bash: pytest`.
10. **Observation 4** (short text): test results summary, kept as text.
11. **Action 5** (text): `submit`.
12. **End**.

In the above example, the encoder is applied twice (to compress observations 2 and 3, highlighted in yellow), while the decoder generates five actions. The model is then able to predict actions from a history where long observations have been replaced by their compact memory representations. For more details and rationale on this, see Section 4.3.2.

4.1 Overview of the Training Process and Model Variants

Figure 4.1 provides a comprehensive overview of the complete training and evaluation pipeline, illustrating how each of our model variants is derived.

The starting point for all variants is a standard, pretrained `Qwen3-8B` model [Yan+25a], which we refer to as `Baseline`. This is a ready-to-use model, not one with random weights. Note that the model in `Baseline` is already a pretrained model, but we stick with the naming of the process in Ge et al. [Ge+23] for consistency and name the two stages as Pretraining (PT) and Fine-Tuning (FT).

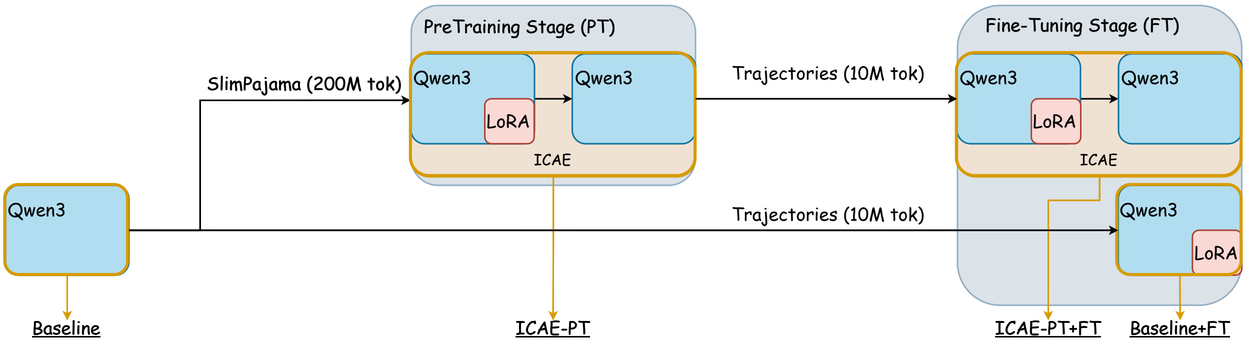


Figure 4.1 Full training process overview, illustrating the derivation of model variants. The diagram shows two parallel training paths starting from the pretrained `Qwen3-8B` baseline. The first path (top) shows the ICAE approach: the baseline initializes both encoder and decoder, then pretraining (PT) on SlimPajama-6B produces `ICAE-PT`, followed by fine-tuning (FT) on agentic trajectories to yield `ICAE-PT+FT`. Only encoder LoRA weights are trained while the decoder remains frozen. The second path (bottom) shows direct LoRA fine-tuning of the baseline on agentic trajectories, producing `Baseline+FT`. Thus, we receive all four main model variants for comparison.

There are two parallel training paths. The first path is for our ICAE model. The `Baseline` model is used to initialize the ICAE encoder and decoder. In the Pretraining (PT) stage, we train the LoRA weights of the encoder on the SlimPajama-6B dataset [Web+24], resulting in `ICAE-PT` model. This model is then fine-tuned on the agentic trajectories dataset, yielding the final `ICAE-PT+FT` model. Crucially, for both ICAE training stages, only the encoder's LoRA weights are updated, while the base Qwen3 model used as the decoder remains frozen.

The second path is for a comparative baseline. The original `Baseline` Qwen3 model is directly fine-tuned with LoRA on the agentic trajectories dataset. This produces the `Baseline+FT` model. This allows us to compare our two-stage ICAE approach against a standard parameter-efficient fine-tuning of a base language model on the target task.

In addition to the process overview in Figure 4.1, Table 4.1 provides a precise mapping of our four main model variant names to their respective encoder and decoder configurations. The encoder is responsible for compressing, while the decoder generates the text. A dash (—) in the 'Encoder' column signifies that no compression module is used, and the decoder processes the uncompressed context directly. For a complete list of all experimental configurations and their results, please refer to Table 6.4 in Chapter 6.

Name in our paper	Encoder	Decoder
Baselines		
`Baseline`	—	Qwen
`Baseline+FT`	—	Qwen (LoRA-finetuned)
ICAE Compression		
`ICAE-PT`	ICAE (LoRA-pretrained)	Qwen
`ICAE-PT+FT`	ICAE (LoRA-pretrained & LoRA-finetuned)	Qwen

Table 4.1 This table maps the names of our four main model variants to their specific encoder and decoder configurations, as depicted in Figure 4.1. The encoder is responsible for compressing the context, while the decoder generates the text. A dash (—) in the "Encoder" column signifies that no compression module is used. Qwen stands for Qwen3-8B model. Pretraining and fine-tuning stages of the training process are described in Section 4.3.1 and Section 4.3.2, respectively. A more comprehensive comparison including other variants is presented in Chapter 6.

4.2 ICAE for Agentic Context Management

The In-Context Autoencoder (ICAE) [Ge+23] consists of two modules: a trainable encoder (typically a LoRA-adapted LLM) and a fixed decoder (the base LLM itself). The encoder processes a long context and generates a fixed number of learnable memory tokens. This design turns a long, potentially unwieldy context into a compact representation that the decoder can efficiently consume. The number of memory tokens controls the compression ratio, and their placement influences how the decoder accesses the stored information [Ge+23].

Figure 4.2 depicts the encoder–decoder split. The encoder ingests the full context and produces memory tokens. The frozen decoder then receives these tokens and a prompt to generate a continuation. During pretraining, the encoder is optimized to enable the decoder to reconstruct the original text. During fine-tuning, the objective shifts to solving a downstream task (e.g., generating a tool call) using the compressed representation. Parameter-efficient methods, such as LoRA [Hu+22], are used to adapt the encoder while preserving the base model's capabilities intact.

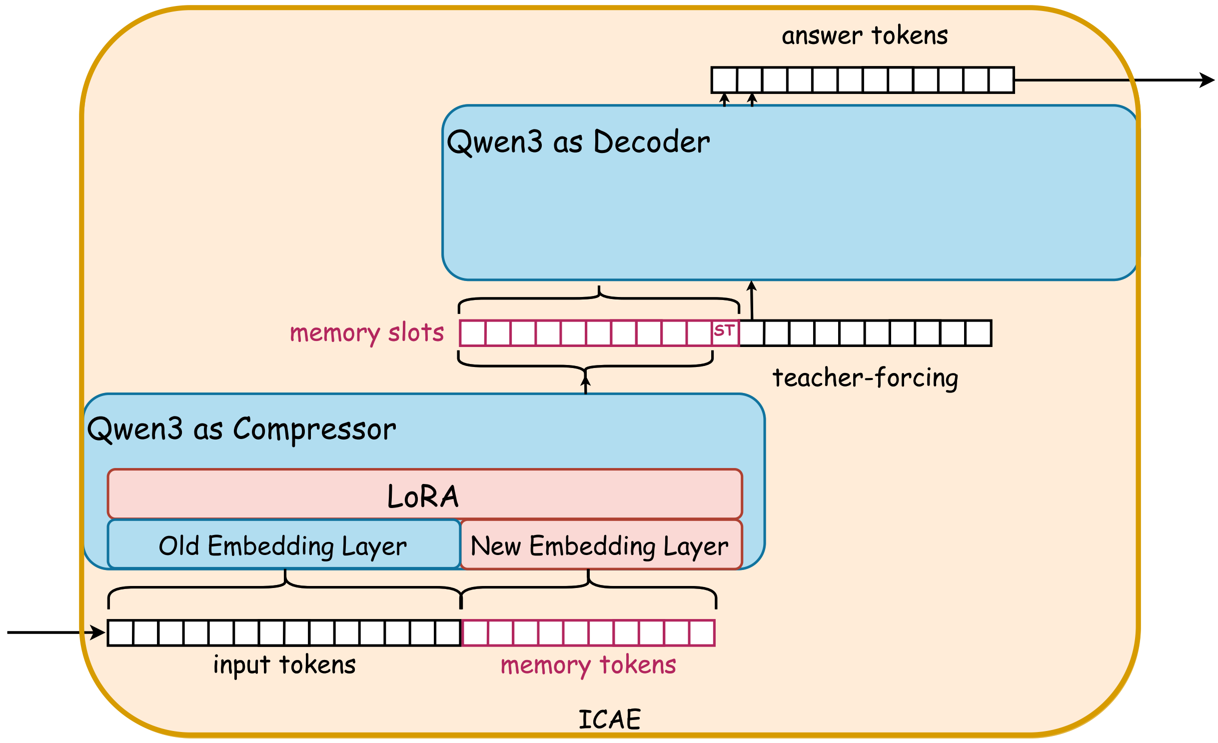


Figure 4.2 The architecture of the In-Context Autoencoder (ICAE). The encoder, a LoRA-adapted LLM, compresses a long context into a few memory slots. The decoder, a frozen base LLM, uses these memory slots (instead of the original context) and a special token (ST) to perform one of two tasks: autoencoding (i.e., text reconstruction) or language modelling (i.e., text continuation). Note that the special token (ST) is just used to signal the type of task to the decoder.

Our main contribution is the adaptation and application of this framework to an agentic setting. In this scenario, an agent interacts with an environment over multiple turns, generating a trajectory of actions and observations. Our method utilizes ICAE to compress long observations, keeping the overall context manageable without losing critical information.

Beyond the basic encoder-decoder structure, the effectiveness of ICAE in our case depends critically on positional information. Further enhancing this process, we incorporate insights on the strategic placement of compressed representations, as detailed by Zhao et al. [Zha+24]. The effectiveness of context compression is highly dependent on the layout of position identifiers (IDs), particularly in models employing relative position embeddings like RoPE, which we discussed in Chapter 2. These mechanisms create a local inductive bias where tokens with smaller positional distances exhibit stronger attention affinity. Consequently, the placement of memory tokens in the position ID space dictates their accessibility to the decoder.

Standard methods often place compressed tokens at the beginning of the context, creating a large positional gap between the compressed history and the current prompt. The work by Zhao et al. [Zha+24] demonstrates that minimizing this distance significantly improves performance. To achieve this, we adopt their strategy of assigning position IDs to the memory tokens that are contiguous with those of the prompt. This effectively "teleports" the compressed context to be adjacent to the current generation step, minimizing the relative distance $j - i$ used in the attention calculation. As a result, the decoder can attend to the compressed information as if it were part of the immediate local context. The adoption of this position ID manipulation strategy yielded significant improvements in our experiments, particularly during the pretraining phase (see Figure 6.3).

4.3 Training Methodology for Agentic ICAE

The process for training ICAE in an agentic scenario is depicted in Figure 4.3. The training data consists of pre-recorded agent trajectories (see an example of a trajectory in Example 4), which are sequences of alternating actions and observations, generated by an expert agent (in our case, Claude 3.7 Sonnet, see Figure 4.3). The collection of these trajectories is described in Section 5.1.5.

At the start of the interaction, an uncompressed user prompt is provided (e.g., system prompt and task description in Example 4). Then, a trajectory unfolds as an agent takes an action (tool call/action, e.g., `bash: ls -la` in Example 4), which is sent to an environment, and receives an observation in return. This observation becomes input for the next step. During the trajectory, compression is applied to every long observation (e.g., observations 2 and 3 in Example 4), ensuring that the decoder model never processes lengthy raw text. Instead, it operates on compact embedding representations.

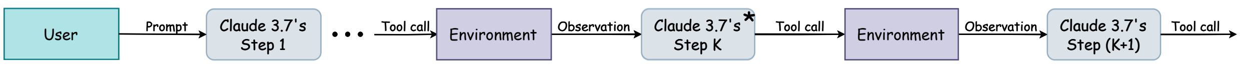


Figure 4.3 Overview of the agentic trajectory generation process used for fine-tuning data. An expert agent (concretely, Claude 3.7 Sonnet) starts with an initial prompt and task description. It then enters a loop, generating a tool call and receiving an observation from the environment. These sequences are collected to form the ground-truth trajectories. Only long observations, marked with an asterisk (*), are selected for compression during the fine-tuning of the ICAE model. The data collection process is further detailed in Section 5.1.5.

Figure 4.4 illustrates a single fine-tuning step. The observation from the environment is passed to the ICAE encoder, which produces a compressed representation. This representation, along with the prior conversation history, is fed to the frozen decoder to generate the next tool call. The training loss is the cross-entropy between the generated tool call and the reference action from the expert trajectory. This loss is backpropagated through both the decoder and encoder to update only the encoder's LoRA weights, while the base model weights remain frozen.

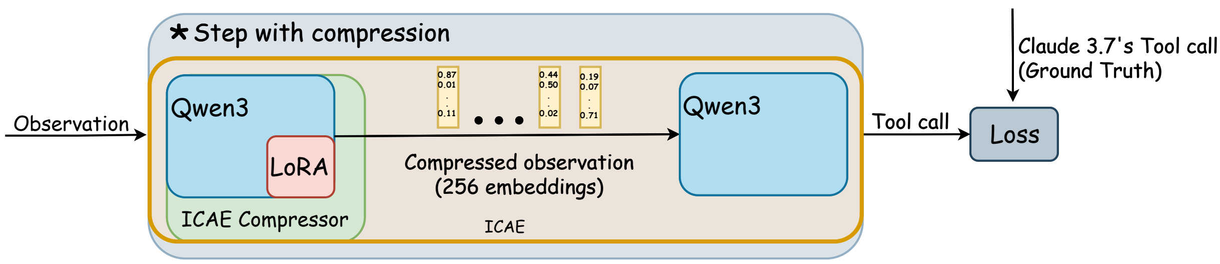


Figure 4.4 A single fine-tuning step for the agentic ICAE. The most recent observation is passed to the encoder to generate memory tokens. These tokens, combined with the history of previous actions and compressed observations, are fed to the frozen decoder. The decoder predicts the next action, and the loss between the predicted action and the reference action from the expert trajectory is backpropagated to update only the encoder's LoRA weights.

4.3.1 Pretraining (PT)

The first stage follows the original ICAE formulation [Ge+23], pretraining the encoder on a large, general-purpose text corpus. Two self-supervised objectives are constructed from text sequences in equal proportion (with a 50% probability for each):

(i) **Autoencoding (AE)**, where ICAE restores the original input text from its memory slots. This task is signaled by a special <AE> token, as conceptually illustrated in Figure 4.2.

(ii) **Language Modeling (LM)**, which predicts the continuation of a context to improve generalization, without the use of any special tokens.

During this stage, only the encoder's LoRA weights are trained. The goal is to teach the encoder to produce embeddings from which the frozen decoder can effectively reconstruct or continue text.

4.3.2 Fine-Tuning (FT)

After pretraining, the ICAE encoder is fine-tuned on a dataset of agentic trajectories (for more details, see Section 5.1.5). It should be noted that only the encoder's LoRA weights are updated. The objective is to maximize the probability of generating the correct agent action (i.e., tool call) conditioned on the memory slots (for compressed observations) and the rest of the previous trajectory history. Each turn in a trajectory is treated as a separate training sample, where the model predicts the next action given the trajectory history up to that point. The last observation is compressed and trained, while all the previous observations are saved during the previous turns and fed in as compressed embeddings.

During training, we optimize over single-step transitions. At a timestep k, the encoder compresses the observation o_{k-1}. The decoder then generates action a_k from the compressed history. The loss from a_k is backpropagated to update the encoder's LoRA weights. Importantly, backpropagation is performed only through a single turn — we do not backpropagate through the entire trajectory history as this would not be feasible due to the memory constraints. Crucially, whenever an observation exceeds a predefined threshold (e.g., 256 tokens), the encoder compresses it into a fixed set of memory tokens. This ensures the model never processes the full raw text of long observations, allowing it to handle arbitrarily long trajectories without exceeding context limits.

4.3.3 Disabling "Thinking" for Tool Use

Finally, a specific configuration choice for our agentic experiments involves the model's reasoning mode. The Qwen3 family includes a "thinking" feature designed for complex chain-of-thought reasoning. However, for our tool-use objectives, we explicitly turn off this feature to prioritize direct action generation and to simplify the context management pipeline. This decision is analyzed further in the ablation study in Appendix A.5.

5 Evaluation

This chapter details the evaluation framework used to assess the performance of the proposed method. The first section describes the datasets employed in the experiments. These include RedPajama for pretraining, SQuAD and RepoQA benchmarks for evaluating general compression and code reconstruction capabilities, and SWE-bench alongside a dataset of agentic trajectories for the primary agentic task. The subsequent section outlines the quality metrics used for evaluation, distinguishing between token-level metrics for assessing generation quality, task-level metrics for end-to-end performance, and efficiency metrics.

5.1 Datasets

5.1.1 RedPajama

To train the In-Context Autoencoder (ICAE) during the pretraining stage, we utilize a subset of the *RedPajama* dataset [Web+24], called *SlimPajama-6B*[1]. It is a widely used text dataset for pretraining LLMs, consisting of 6 billion tokens (a random 1% subset of the original 627 billion tokens). We opted for this dataset because "The Pile" [Gao+20], which was used by the ICAE authors [Ge+23], is no longer available. It was deleted due to a DMCA takedown regarding copyrighted material [Wik25].

5.1.2 SQuAD

To evaluate the general compression capabilities of our model beyond agentic tasks, we also test it on the *Stanford Question Answering Dataset* (SQuAD) [Raj+16]. SQuAD is a reading comprehension benchmark consisting of over 100,000 question-answer pairs sourced from 536 Wikipedia articles. The dataset was constructed by asking crowdworkers to pose up to five questions on paragraphs from these articles, where the answer to each question is a direct span of text from the passage. For example, given the passage "In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity," a corresponding question is "What causes precipitation to fall?" with the answer being the single word "gravity" extracted from the text.

The choice of SQuAD is motivated by its widespread adoption and focus on extractive answers. In contrast, the original ICAE work [Ge+23] utilized the *PWC dataset* (introduced in the same paper), which was not at all later adopted in the literature and was automatically generated using GPT-4 rather than collected from human annotations. Such synthetic data is often lower in quality compared to human-curated benchmarks. By using SQuAD, we ensure generalizability and facilitate fair comparison with existing approaches.

5.1.3 RepoQA

To assess how well our compression method preserves high-fidelity information in code (e.g., all the special characters, formatting, etc.), we utilize the *RepoQA* benchmark [Liu+24a]. RepoQA is specifically designed to evaluate a model's ability to remember the entire codebase and its context. Unlike traditional code-related tasks that focus on standalone snippets, RepoQA requires a holistic understanding of entire codebases. The benchmark's core task, "Searching Needle Function," challenges models to locate a specific function ("needle") within a long, surrounding code context ("haystack") based solely on a natural-language description of its behavior. This setup moves beyond simple keyword matching and tests for a genuine understanding of both the code's semantics and the description's intent.

[1] `https://huggingface.co/datasets/DKYoon/SlimPajama-6B`

The benchmark's framework is flexible, allowing for the creation of evaluation contexts of arbitrary length. While we conduct experiments with contexts up to 16,000 tokens and observe similar trends, our primary evaluation focuses on a context size of 1024 tokens. This choice reflects our goal: we are less concerned with testing the limits of raw context length and more interested in verifying that our compressed representation retains the nuanced, structural details of the source code required to succeed at this task. In our setting, RepoQA serves as a measure of the model's ability not only to decompress but also to generate code.

5.1.4 SWE-bench

We have chosen to use the *SWE-bench* [Jim+23] dataset for our main set of experiments, as it is the most popular benchmark for evaluating the performance of software engineering agents on real-world tasks. Thus, we can compare our results with the state-of-the-art methods. SWE-bench is an auto-collected dataset and has no correct answers, but rather is a verifiable dataset where it is only possible to verify the final fix of an issue. The benchmark contains approximately 3,000 tasks sourced from real GitHub issues and pull requests from 12 popular Python repositories (e.g., Django, Matplotlib). Each task instance is defined by a "fail-to-pass" scenario: a test suite that fails on the repository state before a fix is applied and passes after. The agent's goal is to generate a patch that resolves the issue, making the test suite pass. This setup provides a realistic measure of an agent's problem-solving capabilities. To ensure reproducibility and isolate the agent's performance, the evaluation for each task is conducted within a dedicated Docker container.

We specifically use *SWE-bench Verified* [Cho+24], a subset of the original benchmark that has been human-validated to address several issues with the original dataset. The verification process filters out tasks with underspecified problem descriptions, overly specific tests that might reject valid solutions, or problematic development environments. This results in a more reliable evaluation of an agent's true software engineering capabilities, with the verified subset containing 500 tasks. Our metric of interest is the percentage of solved instances, which is reported on the verified subset.

5.1.5 Agentic Trajectories for Fine-Tuning

High-quality training data is crucial for fine-tuning capable software engineering agents. Since there are no ground-truth answers in SWE-bench, we require expert demonstrations in the form of agentic trajectories (i.e., from a smarter teacher model). Following the approach of AgentTuning [Zen+24], we used a proprietary, state-of-the-art model to generate solutions, which serve as the basis for our fine-tuning dataset. To generate them at scale, we use the data collected in SWE-smith [Yan+25b]. SWE-smith is a pipeline designed to generate large-scale training data for software engineering tasks. It automates the process by constructing execution environments for Python codebases and synthesizing numerous task instances that intentionally break existing tests. The authors have collected a dataset of tasks, that we will utilize for fine-tuning our agent.

While SWE-smith provides the framework for generating tasks, we use it to generate trajectories for fine-tuning our agent. Specifically, we use a powerful teacher model, Claude 3.7 Sonnet, to solve tasks collected by the authors. Claude 3.7 Sonnet is a proprietary model by Anthropic that is able to solve the tasks of the SWE-Bench format with high accuracy (i.e., on SWE-Bench Verified it has >60% resolve rate), so we can use its trajectories as the ground truth for creating our fine-tuning dataset (following the approach of Zeng et al. [Zen+24]). The teacher model's interactions are recorded as trajectories. We only retain the successful trajectories, resulting in a high-quality dataset of approximately 5000 expert demonstrations (i.e., agentic trajectories). This filtering step is crucial to ensure that the agent learns from effective problem-solving strategies. It is worth noting that there is currently no consensus in the research community on whether including unsuccessful trajectories is beneficial for training [Son+24; Zen+24].

Interaction Protocol and Tools It is crucial to detail the agent's interaction protocol because the choice of tools and system prompts significantly impacts the final resolve rate. Different scaffolding frameworks, such as SWE-Agent [Yan+24], offer varying levels of complexity and abstraction, which can lead to different outcomes. The agent interacts with the environment following a protocol and toolset defined by the SWE-smith setup. This setup is designed to be minimal yet expressive enough to solve complex software engineering tasks. The agent is provided with a system prompt that describes the available tools and how to use them. The exact prompt is detailed in Appendix A.3. The agent generates tool calls as plain text, rather than using a model's specific function-calling format.

The available tools are:

- `bash`: A standard shell interface for running commands, allowing the agent to navigate the file system, inspect files, and run tests.
- `submit`: A tool to submit the final patch for evaluation.
- `str_replace_editor`: A stateful file editor designed for precise, line-exact operations.

The `str_replace_editor` is a critical tool that supports viewing, creating, and editing files. Its state persists across steps, enabling consistent multi-edit workflows. The editor exposes several commands, including `view`, `create`, `str_replace`, `insert`, and `undo_edit`. To ensure deterministic edits, the `str_replace` command requires the `old_str` argument to match one or more consecutive lines exactly, including all whitespace. The matched block is then replaced with the content of `new_str`. Similarly, the `insert` command appends content after a specified line number. These precise controls are essential for making targeted changes to code.

5.2 Quality Metrics

We employ a variety of metrics to evaluate our approach, tailored to the specific demands of each task. The evaluation framework distinguishes between task-level metrics for end-to-end performance, token-level metrics for generation quality, and efficiency metrics to evaluate the system's performance in the context of RQ1. While each dataset has a primary metric suited to its objective, we also report the BLEU score [Pap+02] score across all evaluations to provide a consistent basis for comparison (see Figure 6.8).

5.2.1 Token-Level Metrics

For reconstruction and question-answering tasks, we rely on several standard token-level metrics. Token-wise accuracy measures the fraction of generated tokens that match the ground-truth reference (can be in a teacher-forcing regime or autoregressive mode). Exact Match (EM) is a stricter metric that scores a prediction as correct only if it is an exact character-for-character match with the reference answer. EM is particularly well-suited for shorter answers and is commonly used in datasets like SQuAD [Raj+16], which we also follow in our evaluation. The token-level F1 score computes the harmonic mean of precision and recall between the bags of tokens in the prediction and the reference, providing a measure of lexical overlap. Finally, we use the Bilingual Evaluation Understudy (BLEU) score [Pap+02] to measure the similarity between generated and reference text. Specifically, we report BLEU-1, which considers only unigram overlap. This choice is motivated by the nature of our tasks, particularly with code, where the correctness of individual tokens is more critical than the fluency of longer phrases.

5.2.2 Task-Level Metrics

For our main agentic task on SWE-bench Verified, the primary metric is the number of successfully resolved issues (out of 500). An issue is considered resolved if the agent's generated patch, when applied to the repository, causes all previously failing tests to pass while not breaking any previously passing tests. This binary success criterion provides a direct, end-to-end measure of the agent's practical software engineering capabilities. The resolve rate (percentage of successfully resolved issues) serves as the key indicator for assessing the transferability of implicit context condensation from standard NLP tasks to agentic software engineering scenarios in RQ2.

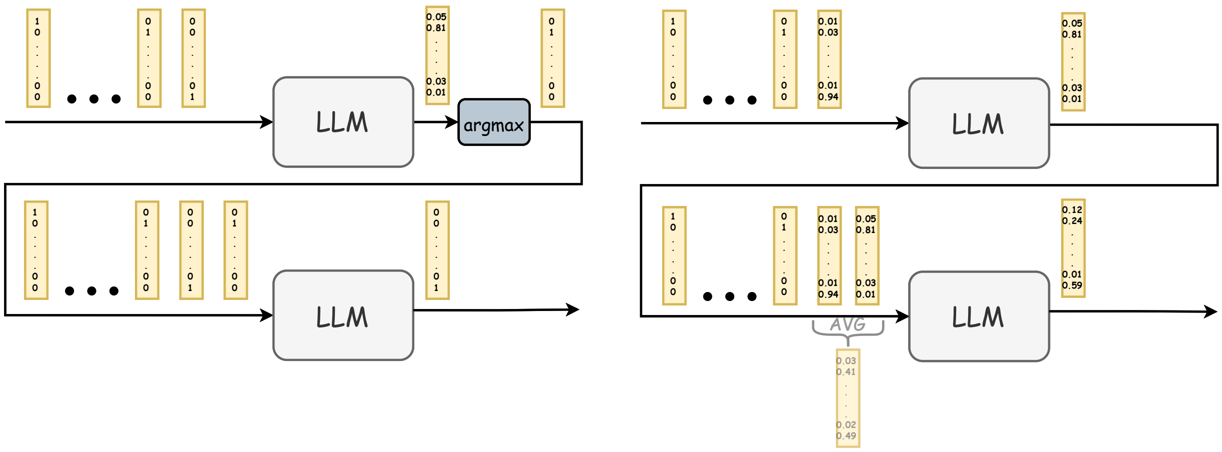
5.2.3 Efficiency Metrics

To evaluate the efficiency of the system in the context of RQ1, two metrics are considered: generation time and the number of steps in the trajectory. Generation time measures the duration required for the model to produce an output (i.e., a tool call/action) and is reported in seconds. The number of steps in the trajectory represents the total amount of tool calls (i.e., actions) in a trajectory. In standard configurations, a hard limit of 75 steps is typically imposed. However, for these specific measurements, this restriction is removed to observe the full trajectory length that the models can achieve. The only remaining constraint is the context window limit of 32,768 tokens. These metrics are used to approximate the agent's efficiency.

6 Experiments

This chapter documents the experimental results. First, training-free and projection-based compression methods are evaluated. Next, the pretraining of the In-Context Autoencoder (ICAE) is described alongside results on general text reconstruction. Subsequently, fine-tuning performance is assessed on SQuAD and later on RepoQA. The chapter concludes with an evaluation on the SWE-bench Verified dataset within an agentic software engineering setting.

6.1 Feasibility of Training-Free Context Condensation



(a) Original approach **(b)** Our modification variant (soft embedding, online)

Figure 6.1 A comparison of token processing pathways. The left image illustrates the standard approach where discrete tokens are converted to embeddings. The right image shows our modified method, which injects continuous soft embeddings directly into the model, bypassing the usual tokenization and embedding lookup stages.

We first investigate whether meaningful context compression can be achieved without any model training. These initial experiments explore methods for replacing discrete tokens with continuous representations, testing the hypothesis that simple embedding aggregation can preserve essential information for downstream tasks.

We explore two primary settings for training-free condensation. In the **hard embedding** setting, discrete tokens are represented as one-hot vectors that index the input embedding matrix. For condensation, we compute the elementwise mean of these vectors. In the **soft-embedding** setting, we bypass the argmax operation and token lookup, feeding a continuous mixture of embeddings directly to the model. Figure 6.1a illustrates the standard token processing pathway, while Figure 6.1b depicts our modification, which injects these continuous embeddings directly.

Online soft-embedding The online pathway is implemented by bypassing token sampling and the embedding lookup. After running a standard decoding step to obtain logits, we compute the corresponding expected embedding and insert this continuous vector directly as the input for the next step using KV-cache manipulation. This approach is illustrated in Figure 6.1b. The goal is to assess whether context can be compressed without trained adapters.

Regenerate-LLM offline To mitigate potential collapse from iterative generation, we also explored an offline method. In this approach, we prompt the model to reproduce a given input sequence under teacher forcing and record all output embeddings at each step. At inference time, these saved embeddings are reused as the context representation, avoiding online recomputation.

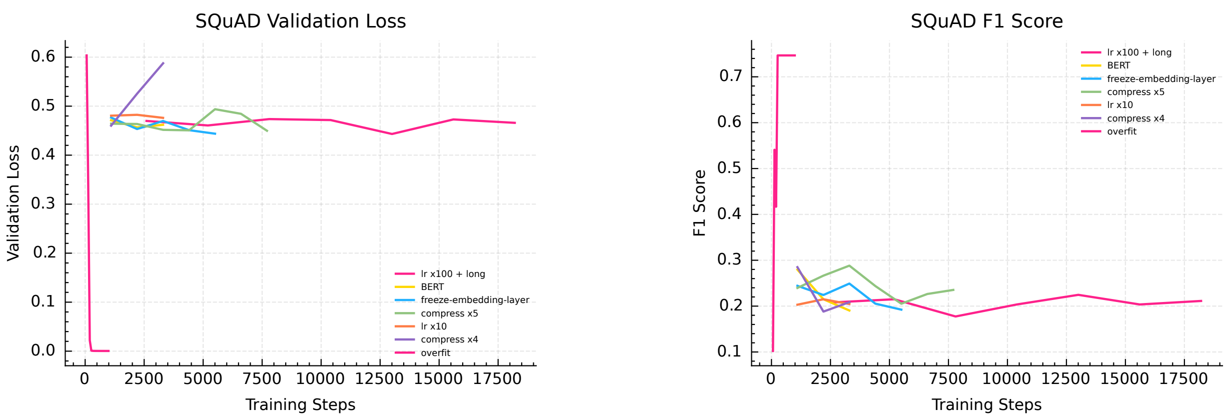
Results Table 6.1 reports SQuAD performance under these condensation strategies. The results establish that replacing discrete tokens with untrained condensed mixtures, whether generated online or offline, substantially degrades performance on the QA task.

Setting (SQuAD), context embed	Exact Match	F1
Baseline — hard tokens	**0.58**	**0.71**
Hard embedded, avg ×2	0.09	0.21
Soft embedded online, avg ×2	0.05	0.11
Soft embedded Regenerate-LLM, avg ×2	0.07	0.16

Table 6.1 Performance of training-free context condensation on SQuAD. This table compares the baseline (hard tokens) against three condensation methods: hard embedding averaging, soft embedding with online generation, and soft embedding with offline regeneration (Regenerate-LLM) averaging.

6.2 Learning Projections for Context Condensation

We also investigate whether a trained, low-capacity projection module could learn to compress adjacent embedding vectors while preserving task-relevant information. We explore several architectural variants to map a pair of concatenated hidden vectors $[e_{2t-1}; e_{2t}]$ from $\mathbb{R}^{2d}$ to a single vector in $\mathbb{R}^{d}$. These variants included a simple linear projector, a shallow non-linear MLP (one to two layers with GELU activation), and a full BERT encoder [Dev+19] to provide a higher-capacity compression mechanism.



(a) Loss on validation set

(b) F1 score on validation set

Figure 6.2 SQuAD validation metrics for projection-based compression. The left plot shows the validation loss, which flattens immediately despite the training loss decreasing. The right plot displays F1 scores that remain far below the baseline. We explored numerous variants, including different learning rates, BERT encoder, varying compression ratios, and extended training durations. While all configurations successfully overfit the training data, none generalized to the validation set, demonstrating that simple projection methods fail to preserve task-relevant information.

All experiments used the SQuAD context-embedding setting, with the base Qwen3-8B model frozen. Only the parameters of the projection module were trained via token-level cross-entropy on the answer generation task. Across all architectural variants, the models failed to generalize. As shown in Figure 6.2, while the models were able to overfit to the training data, the validation loss flattened almost immediately. The resulting F1 scores remained significantly below the uncompressed baseline, demonstrating that these simple projection methods could not recover the performance lost to compression.

We also explored several modifications to the training protocol through systematic ablation studies: We varied the projection type (linear versus 1- or 2-layer MLP), normalization schemes (pre/post LayerNorm, scale-preserving residual gates), regularization techniques (weight decay, dropout), and the decision to re-project via vocabulary space versus staying in hidden space. Additionally, we experimented with unfreezing the token embedding table while keeping the main transformer blocks frozen. None of these changes meaningfully altered the outcome; the models were consistently unable to train and generalize to the SQuAD task.

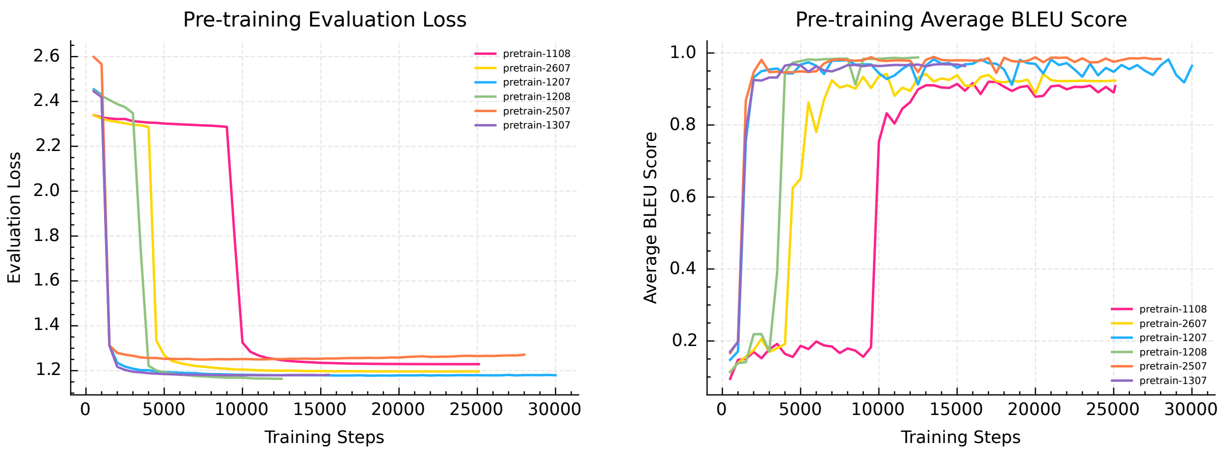
The failure of these methods suggests two potential underlying issues. First, the embedding manifold may possess a non-smooth or complex geometric structure that is disrupted by simple linear or shallow non-linear projections, leading to irreversible information loss that the frozen decoder cannot overcome. Second, even the higher-capacity BERT encoder may lack the expressive power to learn a sufficiently meaning-preserving compression when paired with a much larger, frozen decoder.

6.3 ICAE Pretraining and Results on General Text Reconstruction

This section details the pretraining phase of our In-Context Autoencoder, hereafter referred to as `ICAE-PT`. The methodology for this pretraining stage is conceptually outlined in Chapter 4. The primary objective of this phase is to train the encoder to generate compressed representations from which the original text can be accurately reconstructed.

The pretraining stage was performed on the dataset SlimPajama-6B [Web+24] (detailed in Section 5.1) using a combination of autoencoding and language modeling objectives. Following the original ICAE methodology, the training objective was a 50/50 mix of an autoencoding (AE) task, signaled by a special `<AE>` token, and a language modeling (LM) task, which used no special token. The AE process is conceptually illustrated in Figure 4.2.

In Figure 6.3a you can notice the sudden drop of the loss. This is a phenomenon found in [Zha+24]. It appears due to the modification of positional encodings for the memory tokens. We see the effect being the same as described by the authors. In our experiments, it only appears if we apply the Position ID manipulation (see)



(a) **(b)**

Figure 6.3 Pretraining evaluation metrics across different model configurations. The left plot displays the validation loss, showing a characteristic sharp drop attributed to the positional encoding adjustments for memory tokens. The right plot presents the BLEU scores for text reconstruction, which improve as training progresses. These metrics together illustrate the effectiveness of the pretraining phase.

Run	Checkpoint (# steps)	Compression	BLEU (mean, n=100)
Qwen3-8B (no ICAE)	18k	×1	0.867
10M tokens subset			
ICAE-PT	9k	×4	0.909
ICAE-PT	12k	×4	0.942
ICAE-PT	18k	×4	0.902
1B tokens subset			
ICAE-PT	9k	×4	0.936
ICAE-PT	12k (main)	×4	**0.964**
ICAE-PT	18k	×4	0.928

Table 6.2 Autoencoding reconstruction BLEU scores on SQuAD contexts. This table compares the reconstruction quality for models pretrained (PT) on different data subsets (10M vs. 1B tokens of SlimPajama-6B) at various checkpoints. The checkpoint numbers (9k, 12k, 18k) refer to the number of training steps during pretraining. The run marked (main) is used for all the fine-tunings later. Note that here, pretraining is done via LoRA as well.

We evaluate ICAE-PT autoencoding (AE) pretraining using Qwen3-8B as the base model. During AE, the encoder compresses input contexts at a fixed ×4 ratio (specifically, $1024 \to 256$ tokens on average), and the decoder reconstructs the original text. We report BLEU on SQuAD contexts tested on 100 samples. The checkpoint using 1B tokens for 12k steps is selected as the main run and used for fine-tuning.

We have also experimented with compressing 16 tokens into 4 on average instead of 1024 into 256. This achieved a higher BLEU (≈ 0.982). Notably, none of the 100-sample AE scores reach ≈ 0.99. This may be acceptable for general text, but could be problematic for code, where near-lossless reconstruction is likely a prerequisite for downstream stability.

Below, we show a short example where we attempted to reconstruct the README.md file of the SWE-agent project. The difference is highlighted in yellow.

Autoencoding Reconstruction Failure Example

```
Original:
<p align="center">
<a href="https://swe-agent.com/latest/">
<strong>Documentation</strong></a> 
...

Reconstructed:
<p align="center">
<a href="https://swe-agent.com /agent/ latest/">
<strong>Documentation</strong></a> 
...
```

Even in the very start of the text, the difference is noticeable: the hallucinated `/agent/` path segment in the URL, which could break navigation in a coding task.

In line with internal feedback, these AE findings suggest that the current pretrain/fine-tuning mixes undertrain the model on code: AE BLEU for code should approach text-level (near 1.0) to avoid even small inaccuracies (e.g., link/variable name substitutions).

6.4 ICAE Fine-Tuning and Results on Question Answering

We conduct fine-tuning experiments on the Stanford Question Answering Dataset (SQuAD) [Raj+16]. The rationale for choosing SQuAD over the PWC dataset used in the original ICAE work is detailed in Chapter 5.

During fine-tuning on SQuAD, the encoder compresses the context while the question remains uncompressed ("compressed-text -> uncompressed-question" format). The decoder generates answers from this mixed representation. We use identical LoRA hyperparameters as in [Ge+23].

To establish a comprehensive evaluation baseline, we compare four distinct model configurations that systematically vary the training procedure and compression strategy. First, we evaluate the base Mistral-7B model [Jia+23] without any fine-tuning to establish the zero-shot performance ceiling. Second, we construct a LoRA fine-tuned baseline where we apply LoRA fine-tuning directly to Mistral-7B on the SQuAD dataset without any compression mechanism, thus representing the standard approach without context condensation. This baseline operates without an encoder-decoder structure, functioning as a conventional LLM fine-tuned for question answering at full context length. Third, we evaluate the ICAE model fine-tuned on PWC as provided by the original authors [Ge+23], which represents their reported best configuration. Finally, we train our own ICAE variant by fine-tuning the pretrained encoder-decoder architecture on SQuAD using identical training code and hyperparameters to those employed by the authors for PWC fine-tuning. This parallel setup enables direct comparison while controlling for implementation differences.

Table 6.3 presents the evaluation results across all four configurations, measured using F1 scores on the SQuAD validation set. It is important to note that we apply a hard rule to only compress texts if they are longer than 256 tokens, ensuring that short texts are never inflated by the fixed-length memory tokens. Consequently, the compression ratio for ICAE variants averages approximately 1.7 ± 0.7, meaning contexts are condensed to roughly 60% of their original length while maintaining the compressed representation. The results in Table 6.3, using Mistral-7B, show that our SQuAD-finetuned ICAE model achieves a high

Model	**Compression**	**F1**
`Baseline` (Mistral-7B)	×1	68
`Baseline+FT`	×1	<u>65</u>
`ICAE-PT+FT` (PwC, authors)	×1.7 ± 0.7	57
`ICAE-PT+FT` (SQuAD, ours)	×1.7 ± 0.7	**73**

Table 6.3 This table compares the performance of different models with and without compression. Mistral-7B is used as a base model for these experiments. The model names are explained in Section 4.1. We also compare our Fine-Tuned model on SQuAD to the ICAE authors' Fine-Tuned model on PWC [Ge+23].

F1 score of 73. This surpasses not only the uncompressed baseline but also the uncompressed, LoRA fine-tuned baseline (73 vs. 65). This outcome is notable, as compression typically implies information loss. Here, it might be that the compression provides a beneficial inductive bias, helping the model focus on salient information. However, the ICAE model fine-tuned on the PWC dataset used in the original ICAE work [Ge+23] fails to generalize, underscoring the importance of in-domain fine-tuning.

For consistency across datasets, we also measure BLEU scores, as detailed in Section 5.2. Figure 6.4 presents these results for the Qwen3-8B model. Here, the performance ordering differs: the LoRA fine-tuned baseline surpasses the fine-tuned ICAE model. This is followed by the pretrained-only ICAE model, with the base model performing the lowest. This suggests that while pretraining may temporarily diminish some of the model's capabilities, task-specific fine-tuning helps recover and enhance performance over the baseline on QA tasks.

The discrepancy in relative performance between Table 6.3 and Figure 6.4 can be attributed to the different base models. Despite their close parameter counts, Mistral-7B and Qwen3-8B exhibit distinct characteristics. We hypothesize that Qwen3-8B is more responsive to LoRA fine-tuning, allowing the standard fine-tuned baseline to outperform the compressed variant. This contrasts with the Mistral-7B results, where the ICAE compression provides a more significant relative benefit.

The above results indicate that ICAE compression might be viable for application to downstream tasks. The strong performance on SQuAD motivates our subsequent investigation of the ICAE framework in more complex, agentic settings where context length presents significant computational challenges.

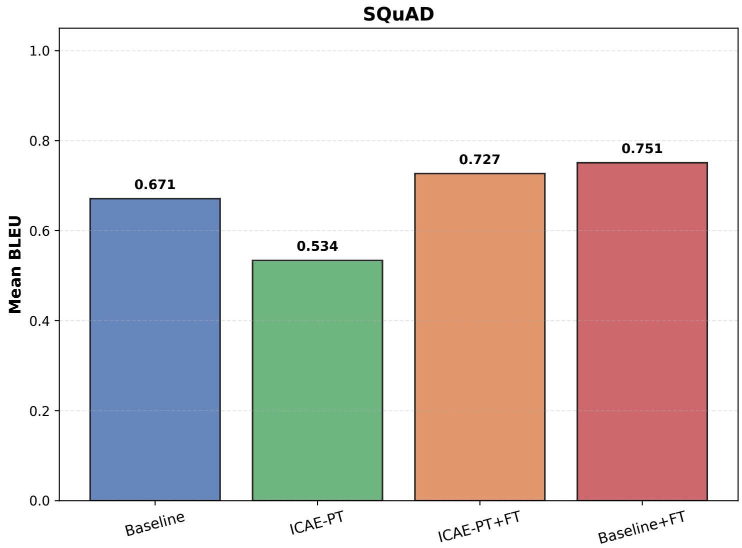
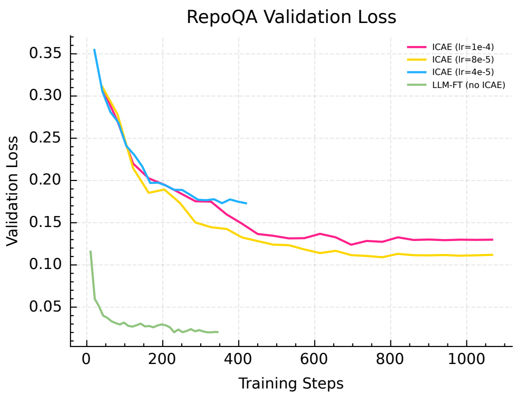


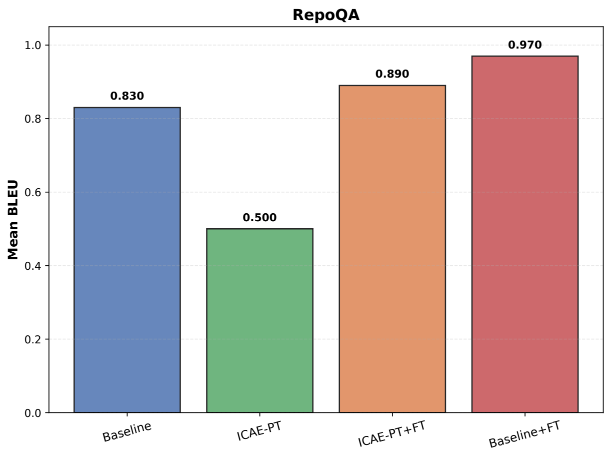
Figure 6.4 BLEU scores on SQuAD using Qwen3-8B as the base model. BLEU is calculated between the generated and the GT answer.

6.5 ICAE Fine-Tuning and Results on Code Reconstruction

To assess the fidelity of our compression mechanism on structured, technical data, we evaluated the ICAE model on the RepoQA benchmark [Liu+24a]. This experiment serves as a test of the encoder's ability to preserve the high-fidelity, granular information inherent to source code, which is a prerequisite for any downstream software engineering task. High-fidelity reconstruction is particularly critical for code, where even minor character-level differences (e.g., incorrect URLs, missing special characters) can lead to functional failures, as illustrated in Example 6.3. As detailed in Section 5.1, we use the "Searching Needle Function" task, which requires the model to reconstruct a specific target function (the "needle") from a long code context (the "haystack") based on a natural language description.



(a) RepoQA Validation Loss



(b) BLEU scores on RepoQA

Figure 6.5 RepoQA evaluation metrics using Qwen3-8B as the base model. The left graph shows the validation loss, which decreases steadily, indicating successful learning, but not even close to an uncompressed LoRA-FT (LLM-FT here refers to the no-compression `Baseline+FT`). The right graph displays BLEU scores for code reconstruction, where the LoRA fine-tuned baseline excels as well.

We fine-tuned the LoRA weights of the encoder on this task, optimizing for token-level cross-entropy loss between the generated and ground-truth functions, exactly like we did for SQuAD. The results of this experiment are summarized in Figure 6.5. Figure 6.5a shows the validation loss during fine-tuning. The loss curve displays a consistent downward trend, confirming that the model effectively learns to reconstruct the target functions from the compressed representations. We also see that Baseline-LoRA instantly achieves better performance in terms of loss, which is expected.

Figure 6.5b presents the BLEU scores, which exhibit a performance ordering consistent with the trends observed in the SQuAD evaluation. The LoRA fine-tuned baseline without compression achieves the highest reconstruction quality, followed by the ICAE model with both pretraining and fine-tuning (`ICAE-PT+FT`). The uncompressed baseline without fine-tuning demonstrates intermediate performance, while the ICAE model with only pretraining (`ICAE-PT`) yields the lowest scores. This hierarchy reinforces the pattern established in our earlier experiments (for all datasets combined, see Figure 6.8) and confirms that task-specific fine-tuning is essential for the compressed representations to approach the performance of uncompressed models.

In summary, the evaluation on RepoQA indicates that the ICAE framework is capable of compressing and reconstructing complex source code with a high degree of fidelity.

6.6 ICAE Fine-Tuning and Results on SWE-bench Verified

This section describes the main experiment of the thesis and discusses its results. The conceptual methodology for applying ICAE to agentic trajectories is detailed in Chapter 4. Figures 4.3 and 4.4 in that chapter illustrate the training process, where the model is fine-tuned on agentic trajectories to predict tool calls from a history of compressed observations. Here, we evaluate the performance of this method on the SWE-bench Verified dataset.

6.6.1 Experimental Setup

We reimplemented the ICAE framework from scratch, building upon the original architecture [Ge+23] with several modifications for improved efficiency and reproducibility. Our implementation uses the Qwen3 model family (specifically Qwen3-8B) as the base LLM, with LoRA adaptation applied to the attention matrices (q_proj and v_proj) using a rank of 128 (see all hyperparameters in Appendix A.6). To reduce complexity, we explicitly turn off the model's "thinking" mode, as detailed in Appendix A.5. The pretraining and fine-tuning processes are described in more detail in Chapter 4.

6.6.2 Evaluation and Results

The results of our experiments are presented in Table 6.4. The configurations for the model variants, including details on the encoder and decoder setups, are described in Chapter 4. The table compares the models on two primary metrics:

- **Resolved (/500)**: The number of issues successfully resolved out of 500.
- **Time (s)**: The mean time in seconds to generate a tool call.

For more details on how we trained the model variants, see Figure 4.1.

We first establish two naive baselines to demonstrate the importance of retaining observation context. The "del long obs-s" approach discards any observation exceeding 256 tokens, while "del all obs-s" removes all observations entirely. As shown in Table 6.4, both methods result in a drastic drop in performance, with almost no issues resolved. While they significantly reduce generation time by shortening the context, their failure highlights that observations are critical for task success, motivating the need for more sophisticated context management techniques like compression.

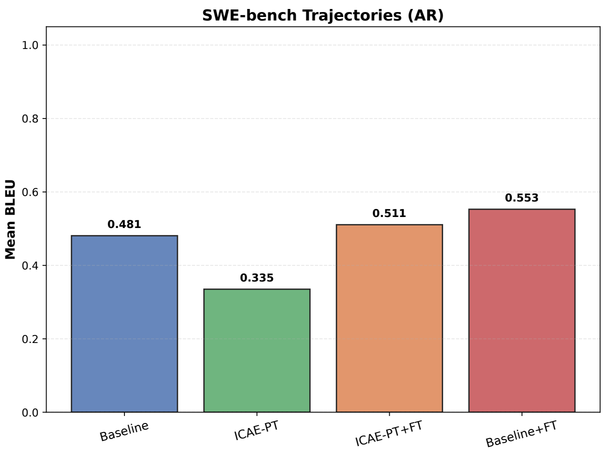
Next, we evaluate three uncompressed baseline models to set performance targets. The fully fine-tuned Qwen3-8B model achieves the highest performance, resolving 86 issues and setting the upper bound for this architecture. The LoRA fine-tuned variant ("`Baseline+FT`") provides a more parameter-efficient alternative, resolving 10 issues. The base Qwen3-8B model without any fine-tuning ("`Baseline`") serves as the most direct point of comparison for our ICAE models, as they use this same frozen model as the

Name in our paper	Encoder	Decoder	Resolved (/500) ↑	Time (s) ↓
Naive Baselines				
—	del long obs-s	Qwen	1	0.44
—	del all obs-s	Qwen	0	0.39
Baselines				
—	—	Qwen (Full-FT)	**86**	1.24
Baseline+FT	—	Qwen (LoRA-FT)	10	1.24
Baseline	—	Qwen	19.4 ± 6.5	1.23
ICAE Compression (ours)				
ICAE-PT	ICAE (LoRA-PT)	Qwen	2	1.23
ICAE-PT+FT	ICAE (LoRA-PT & LoRA-FT)	Qwen	7.8 ± 2.59	**1.12 (0.31+0.81)**
—	ICAE (LoRA-PT & LoRA-FT)	Qwen (LoRA-FT)	10	1.13

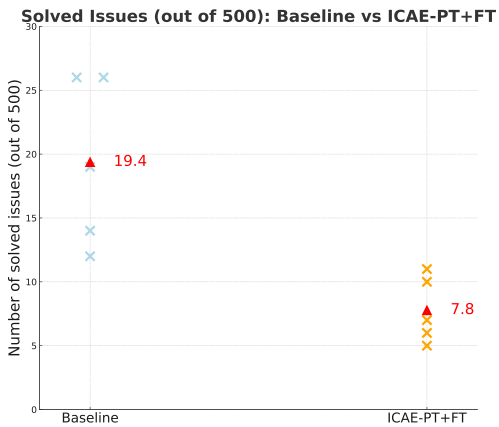
Table 6.4 Performance on SWE-bench Verified. This table compares different model configurations on resolve rate and inference time. "Qwen" refers to the Qwen3-8B model without any fine-tuning. "del long/all obs-s" stands for deleting long (>256 tokens) or all observations during the inference process – these are the naive baselines. PT stands for pretrained, and FT stands for fine-tuned. The fully fine-tuned Qwen model achieves the highest number of resolved issues, setting the upper bound for performance. While ICAE compression reduces inference time, it significantly lowers the task resolution rate compared to the uncompressed baseline.

decoder. It resolves 19.4 ± 6.5 issues, establishing a solid baseline for an off-the-shelf model on this task. The uncertainty (± 6.5) represents the standard deviation across multiple evaluation runs (five runs for each model), quantifying the variability in the model's performance.

The core of our experiment tests ICAE with an encoder fine-tuned on SWE-bench trajectories. When pairing the ICAE-FT encoder with the base Qwen decoder, we observe a modest 10% reduction in generation time. However, this configuration sees a significant drop in task performance, resolving only 7.8 issues compared to the baseline's 19.4. A similar trend holds when using a LoRA-FT decoder, where the resolved rate plummets. This suggests that while compression is time-efficient, it loses critical information necessary for end-to-end task success in this agentic setting.



(a) BLEU scores on SWE-bench



(b) Resolve rate comparison

Figure 6.6 BLEU scores and box plot for SWE-bench Verified. The left chart shows that the fine-tuned ICAE model achieves a higher BLEU score than the baseline. However, the right box plot reveals that the baseline model resolves significantly more issues. The comparison is based on 5 evaluations of the models. Each box represents the interquartile range of the resolve rate, and the triangle stands for the mean value.

The primary negative finding of this study is the substantial decrease in task resolution performance when using ICAE compression. As shown in Table 6.4, the `ICAE-PT+FT` configuration resolved only 7.8 ± 2.59 issues, a marked decline from the 19.4 ± 6.5 issues resolved by the uncompressed baseline. A two-sample Welch t-test on the number of resolved issues across five runs confirms this difference: the baseline achieved a mean of 19.4 resolved issues ($s = 6.54$), while the ICAE model achieved 7.8 ($s = 2.59$), yielding $t(\mathrm{DoF} = 5.22) = 3.69$, $p = 0.013 < \alpha = 0.05$. This establishes that the compressed model on average underperforms the baseline in the agentic setting (also note Figure 6.6b)

This outcome presents a discrepancy when compared to offline evaluation metrics. The BLEU scores for predicting the next tool call, shown in Figure 6.6a, follow a trend consistent with prior experiments on SQuAD and RepoQA, where the `ICAE-PT+FT` model outperforms the uncompressed baseline. This highlights a critical distinction: BLEU is calculated on static, pre-recorded trajectories (by a smarter teacher model, so we take them as ground truth answers), whereas the number of resolved issues is an online metric derived from the agent's autonomous interaction with the environment. We discuss why this might be the case in more detail in Section 7.

Additional experimental configurations. We also investigated additional experimental configurations. We hypothesized that training both the encoder and the decoder simultaneously would improve performance since the number of trained parameters would increase in that case. In one variant, the LoRA-adapted decoder was unfrozen during the fine-tuning stage, allowing for simultaneous training of both the ICAE encoder and the decoder. As reported in Table 6.4, this approach (LoRA-PT & LoRA-FT encoder with LoRA+FT decoder) did not yield a notable performance improvement relative to the uncompressed LoRA-finetuned baseline.

Furthermore, an interesting observation was made regarding the `Baseline+FT` model. Within the online agentic setting of SWE-bench, this configuration substantially underperformed the non-finetuned `Baseline` model. This outcome contrasts with its performance on offline benchmarks such as RepoQA, where LoRA fine-tuning resulted in a significant performance gain (Figure 6.5b). This signals that probably the efficacy of LoRA fine-tuning is task-dependent and its benefits do not necessarily generalize from static benchmarks to dynamic, interactive settings. For more details, see Chapter 7 (Section 7.2).

6.6.3 Impact on Inference Efficiency

In this section, we address RQ1, investigating how implicit context condensation influences the efficiency of LLM-based agents. We measure efficiency in terms of two key metrics: trajectory length (the number of interaction steps within the context window) and inference time (the average time of generating a tool call).

Trajectory Length. Firstly, we analyze the number of interaction steps each agent could perform before termination. For this analysis, the default 75-turn hard cap was disabled, and termination occurred when the context window limit of 32,768 tokens was reached. Our findings provide a clear affirmative answer regarding the benefits of compression. The `ICAE-PT+FT` model was able to execute significantly longer trajectories compared to the uncompressed baseline. On average, the compressed agent performed 113 steps before termination, a 40% increase over the baseline's average of 81 steps. This demonstrates that by condensing lengthy observations into a fixed number of memory tokens, our approach effectively creates more space within the context window, enabling the agent to engage in more extensive problem-solving dialogues.

The difference in trajectory length is further illustrated by the box plot in Figure 6.7. The plot compares the distribution of step counts at termination for the baseline agent and the ICAE-compressed agent. The mean step count for the ICAE agent is visibly higher than that of the baseline, and the entire interquartile range is shifted upwards. Although both models exhibit outliers representing exceptionally long trajectories, the overall distribution for the ICAE agent is skewed towards a higher number of steps, reinforcing the conclusion that compression enables more prolonged interactions.

Inference Time. Secondly, we analyze the computational efficiency of the ICAE compression approach in terms of generation time during inference. The quantitative results, presented in Table 6.4, indicate a measurable reduction in the time required for tool call generation. Specifically, the compressed agent

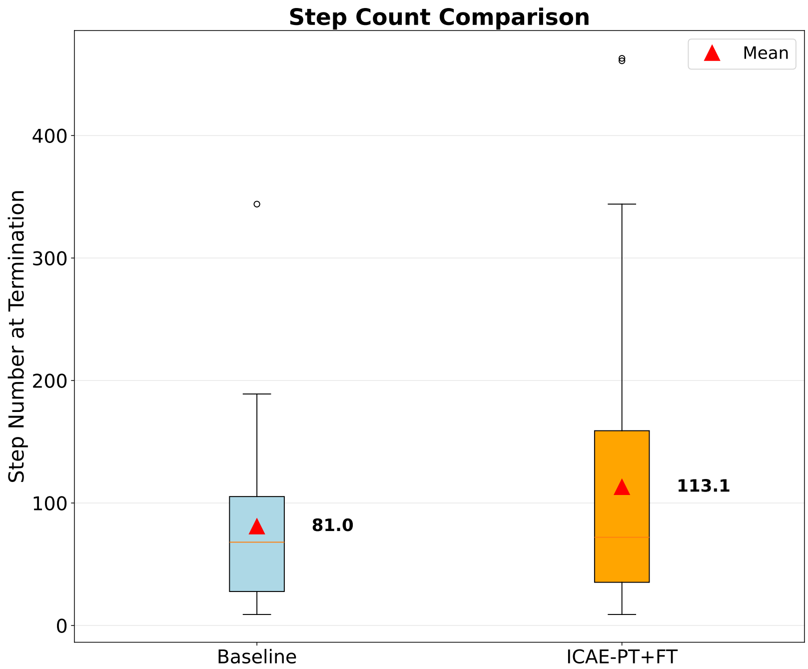


Figure 6.7 Comparison of trajectory lengths at termination. This box plot shows that the ICAE-based agent (right) executes a significantly higher number of steps before reaching the context limit, compared to the uncompressed baseline agent (left). The context window limit is 32,768 tokens for this experiment, while there is no limit on the maximum number of steps the agent can take.

achieved a mean generation time of 1.12 seconds per tool call, representing an approximate 10% reduction compared to the uncompressed baseline's mean of 1.23 seconds. This total time for the compressed agent is composed of two distinct operational phases: the compression of the observation, which averages 0.31 seconds, and the subsequent generation of the next tool call, which averages 0.81 seconds. The observed reduction in generation time is a direct consequence of the decreased context size processed by the decoder following the compression step.

It is relevant to note that these timing measurements were obtained in an experimental setup that did not utilize KV-caching. The absence of KV-caching implies that the reported speedups represent conservative estimates of the potential efficiency gains, as the baseline model processes the full, growing context at every step. A more detailed technical discussion regarding the absence of KV-caching and its implications for real-world deployment scenarios is provided in Section 8.1. These findings confirm that implicit context condensation positively influences inference efficiency, reducing the time per step in the agentic workflow.

6.6.4 Discussion of Performance on Agentic Tasks versus Standard NLP Tasks

In this section, we address RQ2, examining whether the performance of implicit context condensation on standard NLP benchmarks transfers to the more complex domain of agentic software engineering. Our findings indicate that the strong performance of ICAE on standard NLP benchmarks does not directly translate to the dynamic, multi-step environment of agentic software engineering tasks. While offline metrics suggested promise, the practical application revealed a significant performance degradation. As illustrated in Figure 6.6b, the `ICAE-PT+FT` model is statistically significantly outperformed by the uncompressed baseline model in terms of successfully resolved issues. This outcome highlights the gap between single-step, static evaluations and the complexities of online, interactive problem-solving. Interestingly, token-level metrics paint a different picture: Figure 6.8 shows that the fine-tuned ICAE model (`ICAE-PT+FT`) achieves a higher BLEU score than the uncompressed baseline across all datasets. This suggests that while the model learns to mimic the expert's next action well on a static dataset, this capa-

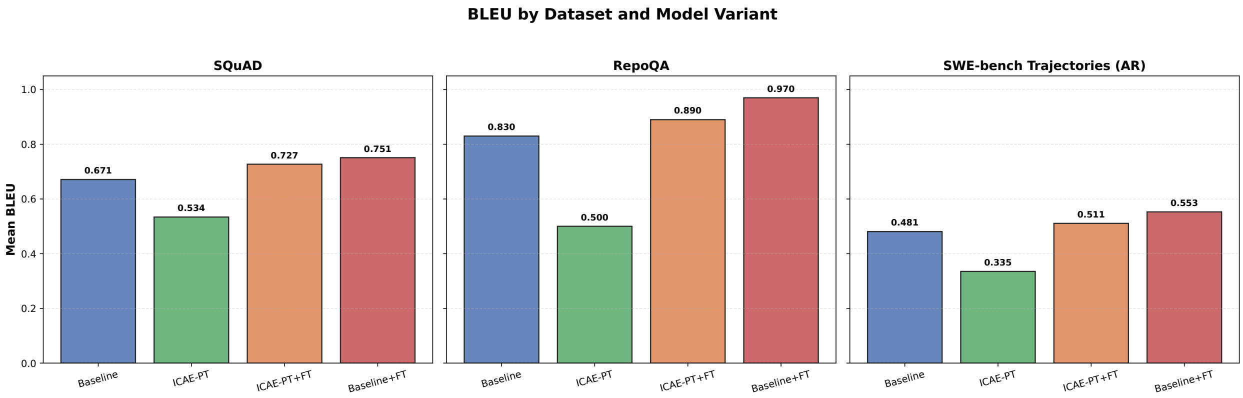


Figure 6.8 BLEU scores comparison across all datasets. This chart consistently shows that the fine-tuned ICAE model (`ICAE-PT+FT`) outperforms the uncompressed baseline in terms of BLEU score. It is also easy to see that the trend is the same for all datasets.

bility does not translate to effective problem-solving when the agent must navigate the consequences of its own actions in a live environment.

7 Discussion

This chapter provides an in-depth analysis of the experimental results presented in Chapter 6 and explores potential explanations for the observed performance patterns. We examine the factors that may contribute to the degradation in agentic task performance when using compression, despite strong results on other benchmarks. The discussion is structured around several hypotheses that collectively might shed some light on the challenges of applying compression methods to multi-turn, interactive SWE tasks.

7.1 Reconstruction Fidelity and Error Accumulation

As established in Section 6.3, the autoencoding pretraining phase does not achieve perfect reconstruction, with BLEU scores peaking at 0.964 and not approaching the near-lossless threshold of ≥ 0.99. While this level of fidelity may be sufficient for general text, it poses a significant risk in the context of software engineering, where precision is critical.

In multi-step agentic trajectories, even minor inaccuracies introduced during the compression of an observation can have cascading effects. An illustrative example is provided in Section 6.3, where a URL was incorrectly reconstructed. In a real-world scenario, such an error could lead to a failed network request, sending the agent down a non-productive path and consuming valuable steps in its trajectory. The same might happen when writing code: a missing or incorrect import statement could lead to a runtime error, causing the agent to waste time and resources on a failed attempt. Over a sequence of dozens of interactions, the accumulation of such minor errors can lead to a significant deviation from the optimal path, often resulting in task failure. The dynamic nature of agentic tasks means that the agent must rely on the outputs of its previous actions. If the compressed representation of those outputs is flawed, the foundation for subsequent decisions becomes unstable.

7.2 Limitations of LoRA for Fine-Tuning Agentic Capabilities

One of the key findings of our experiments is the poor performance of LoRA-based fine-tuning for agentic tasks. As shown in Table 6.4, the `Baseline+FT` model, which uses parameter-efficient fine-tuning without any compression, performs worse than the non-fine-tuned `Baseline` model (10 vs. 19.4 resolved issues). This suggests that the performance degradation seen in our ICAE models may not be solely a consequence of compression, but also a result of the limitations of LoRA in this specific context. We hypothesize that there are two primary reasons for this phenomenon.

First, it is possible that LoRA, due to its design, may be insufficient to capture the complex reasoning and planning capabilities required for multi-step agentic tasks. While full-parameter fine-tuning demonstrates strong performance (86 resolved issues), LoRA may not provide enough expressive power to modify the model's behavior in a meaningful way for these tasks. There is an evident absence of literature on the application of LoRA to agentic trajectories; the standard methodology appears to be full-parameter fine-tuning. We speculate that other researchers may have attempted to use LoRA for agentic fine-tuning, encountered similarly discouraging results, and consequently chose not to publish their findings.

Second, the fine-tuning process may be biased towards stylistic imitation or format learning rather than genuine task learning. Our agentic trajectory dataset was generated using a "teacher" model from a different family (Claude 3.7 Sonnet) than the "student" model being trained (Qwen3-8B). It is plausible that the LoRA fine-tuning process primarily trains the Qwen model to replicate the token distribution and stylistic patterns of the Claude model, rather than learning the underlying problem-solving logic. The model may become proficient at predicting the next tool call in a way that "looks like" the teacher's output (as suggested by the high BLEU scores in Figure 6.6a), but fails to generalize this to effective, autonomous task execution in a live environment. This limitation could be addressed in future work through reinforcement learning (RL) approaches that optimize directly for task success on the benchmark itself, using reward signals rather than supervised learning on pre-collected trajectories.

7.3 Information Prioritization in Multi-Turn Interactions

Agentic software engineering tasks are inherently dynamic and unpredictable. The information that is critical for success at step $k + i$ may seem a minor detail in the observation at step k. This presents a fundamental challenge for any compression method that operates on a per-step basis.

In single-shot NLP tasks such as SQuAD or RepoQA, the context is static and complete. The model is given all the necessary information at once and can learn to identify the salient details required to produce the correct output. In an agentic setting, however, the model must compress an observation without full knowledge of its future relevance. The compression process, which is optimized for immediate reconstruction during pretraining, may inadvertently discard details that appear unimportant at the moment of compression but are essential for long-term planning.

Furthermore, our training methodology relies on single-step transitions. The loss is calculated based on the model's ability to predict the next action, and gradients flow back through only one compression step. We are, in effect, training the model on single-step prediction but evaluating it on multi-step task resolution. This mismatch between the training objective and the evaluation task may explain why the model fails to learn how to prioritize information with long-term consequences. It might be learning to compress for immediate reconstruction, but not for future utility.

A related challenge is the potential degradation of the base model's inherent agentic capabilities during pretraining. The pretraining of the ICAE model is performed on a general text corpus with the objectives of autoencoding and language modeling. While this teaches the encoder to create effective compressed representations, it may simultaneously degrade the base model's inherent agentic capabilities, such as planning, reasoning, and tool-use. Fine-tuning on agentic data is intended to recover these capabilities. Our results on the RepoQA benchmark (Section 6.5) demonstrate that fine-tuning indeed recovers task-specific abilities, such as code generation. However, this specialized fine-tuning may not be sufficient to restore the full spectrum of general agentic competence that is required for complex, multi-step tasks. It is plausible that restoring general agentic capabilities through fine-tuning is inherently more challenging than restoring reconstruction abilities, as observed in other datasets, and may even be impossible to fully achieve once degraded. This could explain why the non-fine-tuned `Baseline` model, despite its lack of specific training, outperforms the more specialized, fine-tuned models in the complex agentic environment of SWE-bench.

8 Limitations and Future Work

8.1 Limitations of Fixed-Length Context Condensation

A core methodological limitation of the investigated approach is its reliance on a fixed number of memory tokens (e.g., 256) for context condensation. This design choice imposes a hardcoded, fixed compression ratio. For instance, a model configured with 256 memory tokens and a 4x compression ratio can only process a maximum of 1024 tokens of context at once. For longer inputs, such as an observation of 10,000 tokens, the condensation process would need to be applied iteratively in a loop. This would likely be slow and undermine the efficiency gains of the approach.

This fixed-length strategy is based on the assumption that all tokens in the context are equally important, which aligns with lossless autoencoding. However, this assumption becomes problematic at the high, lossy compression ratios required for significant context reduction. Experimental results confirm that performance attenuates or fails at high compression ratios (e.g., beyond 15x or 31x).

8.2 Limitations of KV-caching

A notable limitation of our experimental setup is the exclusion of Key-Value (KV) caching, a standard optimization for autoregressive inference in Transformer models. For methodological simplicity and to isolate the effects of context compression, in our experiments, we recomputed the full attention state at each decoding step.

However, the ICAE framework is fully compatible with KV-caching. The continuous embeddings produced by the encoder can be treated as a fixed prefix, and their corresponding key-value states can be pre-computed and cached. Subsequent token generation would then reuse these cached states, significantly improving the absolute speed of the decoding process. While KV-caching is essential for production deployment to achieve practical inference speeds, it was not necessary for our comparative evaluation.

8.3 Constraints on Computational Resources

8.3.1 Model Scale

Due to computational and time constraints, our experiments were confined to the Qwen3-8B model. While evaluating the full 500-issue SWE-bench Verified dataset provides sufficient statistical power for robust comparisons at this scale, an important direction for future research is to investigate ICAE's effectiveness on larger models like Qwen3-32B. It remains an open question whether larger models, with their increased capacity, can better leverage compressed representations or if they are more sensitive to information loss during compression.

8.3.2 Full Fine-Tuning and the LoRA Bottleneck

Our experiments with ICAE exclusively utilized LoRA for parameter-efficient fine-tuning, consistent with the original work. However, our `baseline` experiments revealed a critical limitation: the LoRA-tuned baseline significantly underperformed the frozen base model in the agentic setting, whereas the fully fine-tuned model established the high-performance upper bound. This "LoRA bottleneck" suggests that low-rank adaptation may be insufficient for learning the complex, multi-step reasoning required for autonomous software engineering, regardless of context compression. Consequently, applying full fine-tuning to the ICAE encoder is not merely an optimization but likely a necessity for this domain. Our open-sourced codebase facilitates this, and we identify it as the most important next step for future research.

8.3.3 Disabled Reasoning

For methodological simplicity, we disabled the Qwen3 decoder model's reasoning (i.e., "thinking") mechanism, which is designed to improve performance on complex reasoning tasks. Enabling this feature could prove highly beneficial, as it might allow the model to iteratively reason over the compressed knowledge stored in memory slots, potentially leading to better decision-making. Given that chain-of-thought reasoning has been shown to dramatically improve performance on various benchmarks, exploring the interaction between compressed context and explicit reasoning steps is a critical avenue for future research.

8.4 Open Source Contributions and Reproducibility

We reimplemented the ICAE framework from scratch, as the original authors' code was outdated and difficult to adapt to our experimental needs. Our implementation is modular and provides separate training pipelines for both pretraining (PT) and fine-tuning (FT). We support pretraining on general text datasets and fine-tuning on question-answering, repo-qa tasks, and agentic trajectories.

To advance the field of context compression for software engineering agents, we release our complete implementation, including pretrained models achieving 95% reconstruction BLEU. To ensure full reproducibility, we provide our complete implementation[1], full Weights & Biases experiment logs for pretraining[2] and fine-tuning[3], and model checkpoints[4]. Our comprehensive release includes all training configurations, hyperparameters, and experiment logs, enabling future researchers to reproduce our results and build upon this work. This open-source release provides the tools and transparency necessary for scientific progress in context management for LLMs.

[1] https://github.com/JetBrains-Research/icae
[2] https://wandb.ai/kirili4ik/icae-pretraining
[3] https://wandb.ai/kirili4ik/icae-swebench-finetune
[4] https://huggingface.co/Kirili4ik/icae

9 Conclusion

This thesis examined implicit context condensation for software engineering agents, instantiating the approach with an In-Context Autoencoder (ICAE) built on Qwen3-8B and evaluating it on question answering (SQuAD), code reconstruction (RepoQA), and end-to-end agentic SWE tasks (SWE-bench Verified). The work also assessed training-free and lightweight learned compression attempts, and documented efficiency effects and limits.

9.1 Summary of Contributions

1. **An applied methodology for agentic context condensation.** This work details the implementation of a compressor–decoder pipeline based on the In-Context Autoencoder (ICAE) framework (Figure 4.2), specifically adapted for managing the long observational contexts in agentic workflows (Figure 4.3). This provides a reproducible template for researchers investigating implicit compression as an alternative to retrieval-based or extended-context models.

2. **A multi-domain evaluation of implicit compression.** The method was benchmarked across three distinct domains: open-domain question answering (SQuAD, see Table 6.3), code reconstruction (RepoQA, see Figure 6.5), and agentic software engineering (SWE-bench Verified, see Table 6.4). This comprehensive evaluation offers a granular understanding of the trade-offs involved, suggesting that performance on standard NLP tasks does not necessarily transfer to multi-turn, interactive agentic settings.

3. **Quantification of performance trade-offs.** The empirical results establish a trade-off. ICAE improves question-answering accuracy (Table 6.3) and allows for longer agent trajectories (Figure 6.7) with a modest generational time reduction (Table 6.4). However, under the tested configuration, it degrades end-to-end task resolution on SWE-bench Verified (Table 6.4). This suggests that for other researchers, the value of this compression approach is task-dependent, highlighting a critical challenge: maintaining high-fidelity information in compressed state representations for multi-step tasks.

4. **Exclusion of simpler compression alternatives.** The research also documents the failure of simpler, training-free condensation methods (e.g., averaging, as shown in Table 6.1) and lightweight projectors (Figure 6.2). This finding steers future work away from trivial baselines and reinforces the need for expressive compressors to create useful context representations.

5. **Publicly available implementation and experimental artifacts.** A modular reimplementation of the framework, along with training configurations and logs (see Appendix A.6), has been made available. This facilitates direct replication of the presented results and provides a foundation for extensions, such as exploring different backbone models, training procedures, or more advanced compression architectures.

9.2 Synthesis of Findings

RQ1 — Efficiency: Longer trajectories and generation speed. Implicit context condensation increased the number of agent steps before context overflow (113 vs. 81 on average, a 40% increase) and reduced per-call generation time by approximately 10% (1.12 seconds vs. 1.23 seconds for the baseline). The compressed agent's total time comprises 0.31 seconds for observation compression and 0.81 seconds for tool call generation. Thus, the method enables the completion of longer trajectories under a fixed context window, thereby improving efficiency.

RQ2 — Transferability: NLP to Agentic SWE. Performance gains observed on SQuAD and preserved fidelity on RepoQA indicate successful transfer to single-shot software engineering tasks. However, these benefits did not extend to the agentic setup in end-to-end SWE-bench Verified (Table 6.4; Figure 6.7). The model learned to predict next actions well on static trajectories (higher BLEU) yet under-resolved tasks when its own actions shaped subsequent observations (Figure 6.5 vs. Figure 6.6). This gap highlights the difference between offline imitation or single-shot tasks and online, multi-turn control in software engineering environments (Section 6.6.4).

Overall interpretation. ICAE-style implicit condensation is suitable when the goal is to read more context or run longer with moderate accuracy demands, and it is effective for extractive QA and single-shot SWE tasks with task-specific fine-tuning. In contrast, for agentic SWE tasks where agents must plan, act, and recover from their own intermediate outputs, the present configuration (4× compression; LoRA-only encoder; disabled reasoning) shows a decrease in the ability to solve the tasks end-to-end. The limitations chapter lists concrete avenues for future research (full-parameter fine-tuning of the encoder, higher-fidelity AE for code, adaptive memory sizing, enabling reasoning) that follow directly from the observed failure modes and should likely address the current shortcomings (Section 8.1, Section 8.2, Section 8.3).

In summary, the approach enables agents to run longer and faster, is applicable for QA and single-shot coding tasks, but does not improve (and, in this setup, actually harms) agentic end-to-end SWE-bench resolve rates.

A Appendix

A.1 On the Use of AI

I **have** used generative AI to help me write the text for this work. [Com+25]
I **have** used generative AI to help me write the code for the experiments. [Ant25]
I **have not** used AI in order to create new experiments, nor for any goals, research questions, hypotheses and other types of research-related work.

A.2 SWE-smith First User Message

This is the initial user message template that provides the agent with the repository context and task instructions for solving the issue.

SWE-smith First User Message

```
<|im_start|>user
<uploaded_files>
<repo_path>*
</uploaded_files>
I've uploaded a python code repository in the directory /mnt/shared-fs/gelvan/swe-agent-distillation. Consider the following PR
description:

<pr_description>
<issue_description>*
</pr_description>

Can you help me implement the necessary changes to the repository so that the requirements specified in the <pr_description> are met?
I've already taken care of all changes to any of the test files described in the <pr_description>. This means you DON'T have to
modify the testing logic or any of the tests in any way!
Your task is to make the minimal changes to non-tests files in the /mnt/shared-fs/gelvan/swe-agent-distillation directory to ensure
the <pr_description> is satisfied.
Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to find and read code relevant to the <pr_description>
2. Create a script to reproduce the error and execute it with `python <filename.py>` using the bash tool, to confirm the error
3. Edit the source code of the repo to resolve the issue
4. Rerun your reproduce script and confirm that the error is fixed!
5. Think about edgecases and make sure your fix handles them as well
Your thinking should be thorough and so it's fine if it's very long.<|im_end|>
```

*`<issue_description>` is replaced with the specific problem description for each SWE-bench task.
*`<repo_path>` is replaced with the path to the repository.

A.3 SWE-smith Prompt

The following prompt is from the SWE-smith paper [Yan+25b].

SWE-smith Prompt

```
You are a helpful assistant that can interact with a computer to solve tasks.
<IMPORTANT>
* If user provides a path, you should NOT assume it's relative to the current working directory. Instead, you should explore the
file system to find the file before working on it.
</IMPORTANT>

You have access to the following functions:

---- BEGIN FUNCTION #1: bash ----
Description: Execute a bash command in the terminal.

Parameters:
  (1) command (string, required): The bash command to execute. Can be empty to view additional logs when previous exit code is
  `-1`. Can be `ctrl+c` to interrupt the currently running process.
---- END FUNCTION #1 ----

---- BEGIN FUNCTION #2: submit ----
Description: Finish the interaction when the task is complete OR if the assistant cannot proceed further with the task.
No parameters are required for this function.
---- END FUNCTION #2 ----

---- BEGIN FUNCTION #3: str_replace_editor ----
Description: Custom editing tool for viewing, creating and editing files
* State is persistent across command calls and discussions with the user
* If `path` is a file, `view` displays the result of applying `cat -n`. If `path` is a directory, `view` lists non-hidden files
and directories up to 2 levels deep
* The `create` command cannot be used if the specified `path` already exists as a file
* If a `command` generates a long output, it will be truncated and marked with `<response clipped>`
* The `undo_edit` command will revert the last edit made to the file at `path`

Notes for using the `str_replace` command:
* The `old_str` parameter should match EXACTLY one or more consecutive lines from the original file. Be mindful of whitespaces!
* If the `old_str` parameter is not unique in the file, the replacement will not be performed. Make sure to include enough
context in `old_str` to make it unique
* The `new_str` parameter should contain the edited lines that should replace the `old_str`

Parameters:
  (1) command (string, required): The commands to run. Allowed options are: `view`, `create`, `str_replace`, `insert`,
  `undo_edit`.
Allowed values: [`view`, `create`, `str_replace`, `insert`, `undo_edit`]
  (2) path (string, required): Absolute path to file or directory, e.g., `/repo/file.py` or `/repo`.
  (3) file_text (string, optional): Required parameter of `create` command, with the content of the file to be created.
  (4) old_str (string, optional): Required parameter of `str_replace` command containing the string in `path` to replace.
  (5) new_str (string, optional): Optional parameter of `str_replace` command containing the new string (if not given, no string
  will be added). Required parameter of `insert` command containing the string to insert.
  (6) insert_line (integer, optional): Required parameter of `insert` command. The `new_str` will be inserted AFTER the line
  `insert_line` of `path`.
  (7) view_range (array, optional): Optional parameter of `view` command when `path` points to a file. If none is given, the full
  file is shown. If provided, the file will be shown in the indicated line number range, e.g., [11, 12] will show lines 11 and
  12. Indexing at 1 to start. Setting [start_line, -1] shows all lines from start_line to the end of the file.
---- END FUNCTION #3 ----


If you choose to call a function ONLY reply in the following format with NO suffix:

Provide any reasoning for the function call here.
<function=example_function_name>
<parameter=example_parameter_1>value_1</parameter>
<parameter=example_parameter_2>
This is the value for the second parameter
that can span
multiple lines
</parameter>
</function>

<IMPORTANT>
Reminder:
- Function calls MUST follow the specified format, start with <function= and end with </function>
- Required parameters MUST be specified
- Only call one function at a time
- Always provide reasoning for your function call in natural language BEFORE the function call (not after)
</IMPORTANT>
```

A.4 ICAE In-Context Ability Proof-of-Concept Example

This example demonstrates a hand-crafted scenario, closely related to SWE-bench, designed to test the model's ability to comprehend and utilize decompressed data from its memory. The highlighted part shows the model successfully identifying a hidden requirements file by using the context provided in its compressed memory.

ICAE Proof-of-Concept Example

```
<|im_start|>system
<system_prompt>*
<|im_end|>

<|im_start|>user
<uploaded_files>
/mnt/shared-fs/gelvan/swe-agent-distillation
</uploaded_files>
I've uploaded a python code repository in the directory /mnt/shared-fs/gelvan/swe-agent-distillation. Consider the following PR
description:
<pr_description>
Your task is to read file README.md and run the commands that are presented in that file for the setup. when i run them i get errors.
</pr_description>
<fixed_part_of_the_first_user_message>*<|im_end|>

<|im_start|>assistant
$ str_replace_editor --command view --path /mnt/shared-fs/gelvan/swe-agent-distillation/README.md<|im_end|>

<|im_start|>user
<MEMORY_TOKEN> x 512 <|im_end|>

<|im_start|>assistant
$ cat /mnt/shared-fs/gelvan/swe-agent-distillation/requirements.txt<|im_end|>

<|im_start|>user
There is no file named requirements.txt<|im_end|>

<|im_start|>assistant
$ str_replace_editor --command view --path /mnt/shared-fs/gelvan/swe-agent-distillation<|im_end|>

<|im_start|>user
You have no access to this directory.<|im_end|>

<|im_start|>assistant
$ ls -la /mnt/shared-fs/gelvan/swe-agent-distillation<|im_end|>

<|im_start|>user
You have no access to this directory. You have to call a command from README.md to install.<|im_end|>

<|im_start|>assistant
$ str_replace_editor --command view --path /mnt/shared-fs/gelvan/swe-agent-distillation/README.md<|im_end|>

<|im_start|>user
<MEMORY_TOKEN> x 512 <|im_end|>

<|im_start|>assistant
$ cat /mnt/shared-fs/gelvan/swe-agent-distillation/.my_secret_requirements.txt <|im_end|>
```

*`<system_prompt>` is from Appendix A.3.

This proof-of-concept example demonstrates that the ICAE framework can successfully retrieve and utilize information from compressed memory in principle. However, it is important to note that this success in a controlled, hand-crafted scenario does not translate to robust performance in practice. As shown in Table 6.4, the `ICAE-PT+FT` model resolved only 7.8 ± 2.59 issues on SWE-bench Verified, compared to the baseline's 19.4 ± 6.5 issues, indicating that while the model can access compressed information in isolated cases, it struggles to maintain task performance in the full complexity of multi-turn agentic interactions.

A.5 Ablation Experiment: Disabling Thinking

The Qwen3 model family provides a mechanism to toggle its "thinking" mode, which is designed for complex reasoning. For our agentic experiments, where a tool call generation is prioritized, we decided to disable this feature not to overcomplicate the pipeline. The official recommendation for disabling thinking involves prefixing the generation prompt with a specific token sequence, `<think>\n\n</think>`, which signals to the model that the "thinking" step has already occurred and is empty. Crucially, for multi-turn interactions, this prefix should be ephemeral: it is added for generation and then omitted from the conversation history to prevent it from influencing subsequent turns.

In an early stage of our experiments, we explored the model's sensitivity to this prompting convention by deviating from the recommended protocol. Instead of removing the thinking prefix from the history, we allowed it to accumulate with each agent step. This resulted in a setup where the context for generating action a_k contained not only the history of actions and observations but also $k-1$ instances of the thinking-disabling prefix. While unintentional, this created a distinct experimental condition that we analyzed for its impact on model performance.

Table A.1 compares the performance of key model configurations under both the official "Clean" prompting protocol and our "Cumulative" prompting experiment.

Table A.1 Comparison of prompting strategies for disabling thinking in Qwen3-8B. "Cumulative" refers to accumulating the ‘`<think>...`‘ prefix in the history, while "Clean" follows the official recommendation of removing it after each step.

Configuration	Prompting Strategy	Accuracy ↑
`Baseline`	Cumulative	0.9000
`Baseline`	Clean	0.8967
`ICAE-PT+FT`	Cumulative	0.9089
`ICAE-PT+FT`	Clean	0.9020

The results present an unexpected finding. The "Cumulative" prompting strategy, despite polluting the context with repetitive, non-semantic tokens, yielded slightly higher token-wise accuracy compared to the "Clean" approach for both the `baseline` Qwen model (0.9000 vs. 0.8967) and the `ICAE-PT+FT` compressed model (0.9089 vs. 0.9020). Note that we, of course, do not calculate the accuracy on the thinking tokens. We hypothesize that the repeated, structured nature of the accumulated prefixes might enable the model's attention mechanism to process the context more efficiently, or that the model learns to largely ignore these predictable tokens, leading to a marginal improvement in next-token prediction.

However, it is important to note that these differences are small and might be attributable to randomness in the evaluation process. Due to resource constraints, we did not conduct multiple runs to calculate standard deviations for these measurements, which would be necessary to establish statistical significance.

Consequently, all other experiments reported in this work, including the main results in Table 6.4, were conducted using the official "Clean" prompting methodology to ensure the validity and reproducibility of our findings. This ablation study may be of interest to future work involving the Qwen3 model family.

A.6 Training Details and Hyperparameters

A.6.1 Pretraining Configuration

Parameter	**Value**
Base Model	Qwen3-8B
Dataset	SlimPajama-6B
Learning Rate	1×10^{-4}
Batch Size	1
Gradient Accumulation	8
Training Steps	≈100,000
Memory Size	256 tokens (4× compression)
LoRA r	128
LoRA α	32
LoRA Target Modules	q_proj, v_proj
Optimizer	AdamW
Warmup Steps	300
Hardware	1× NVIDIA H200 GPU
Training Time	≈1 day 15 hours

Table A.2 Pretraining hyperparameters and configuration

A.6.2 Fine-tuning Configuration

Parameter	**Value**
Base Model	Qwen3-8B
Dataset	SWE-bench trajectories
Learning Rate	5×10^{-5}
Batch Size	1
Gradient Accumulation	1
Training Steps	≈150,000
Memory Size	256 tokens (4× compression)
LoRA r	128
LoRA α	32
LoRA Target Modules	q_proj, v_proj
Optimizer	AdamW
Warmup Steps	250
Hardware	1× NVIDIA H200 GPU
Training Time	≈3 days

Table A.3 Fine-tuning hyperparameters and configuration

A.6.3 SWE-bench Inference Configuration

Parameter	Value
Dataset	SWE-bench Verified
Temperature	1.0
Per-instance Call Limit	75
Max Input Tokens	32768

Table A.4 SWE-bench inference configuration

A.6.4 Profiling Setup

All experiments were conducted on a single server equipped with an Intel processor and 1× NVIDIA H200 GPU. We utilized only one GPU for all training and evaluation tasks to ensure consistency across experiments. We thank JetBrains Research[1] for providing the computational resources necessary for this work.

[1] `https://www.jetbrains.com/research/`

List of Figures

1.1 Comparison between a base agent and our approach for handling long observations exceeding the LLM's context length. The base agent fails when processing long observations directly, while our agent successfully compresses the observation into a fixed set of embeddings before LLM processing, enabling continued task execution. 2

4.1 Full training process overview, illustrating the derivation of model variants. The diagram shows two parallel training paths starting from the pretrained `Qwen3-8B` baseline. The first path (top) shows the ICAE approach: the baseline initializes both encoder and decoder, then pretraining (PT) on SlimPajama-6B produces `ICAE-PT`, followed by fine-tuning (FT) on agentic trajectories to yield `ICAE-PT+FT`. Only encoder LoRA weights are trained while the decoder remains frozen. The second path (bottom) shows direct LoRA fine-tuning of the baseline on agentic trajectories, producing `Baseline+FT`. Thus, we receive all four main model variants for comparison. 14

4.2 The architecture of the In-Context Autoencoder (ICAE). The encoder, a LoRA-adapted LLM, compresses a long context into a few memory slots. The decoder, a frozen base LLM, uses these memory slots (instead of the original context) and a special token (ST) to perform one of two tasks: autoencoding (i.e., text reconstruction) or language modelling (i.e., text continuation). Note that the special token (ST) is just used to signal the type of task to the decoder. 15

4.3 Overview of the agentic trajectory generation process used for fine-tuning data. An expert agent (concretely, Claude 3.7 Sonnet) starts with an initial prompt and task description. It then enters a loop, generating a tool call and receiving an observation from the environment. These sequences are collected to form the ground-truth trajectories. Only long observations, marked with an asterisk (*), are selected for compression during the fine-tuning of the ICAE model. The data collection process is further detailed in Section 5.1.5. 16

4.4 A single fine-tuning step for the agentic ICAE. The most recent observation is passed to the encoder to generate memory tokens. These tokens, combined with the history of previous actions and compressed observations, are fed to the frozen decoder. The decoder predicts the next action, and the loss between the predicted action and the reference action from the expert trajectory is backpropagated to update only the encoder's LoRA weights. . . . 16

6.1 A comparison of token processing pathways. The left image illustrates the standard approach where discrete tokens are converted to embeddings. The right image shows our modified method, which injects continuous soft embeddings directly into the model, bypassing the usual tokenization and embedding lookup stages. 23

6.2 SQuAD validation metrics for projection-based compression. The left plot shows the validation loss, which flattens immediately despite the training loss decreasing. The right plot displays F1 scores that remain far below the baseline. We explored numerous variants, including different learning rates, BERT encoder, varying compression ratios, and extended training durations. While all configurations successfully overfit the training data, none generalized to the validation set, demonstrating that simple projection methods fail to preserve task-relevant information. 24

6.3 Pretraining evaluation metrics across different model configurations. The left plot displays the validation loss, showing a characteristic sharp drop attributed to the positional encoding adjustments for memory tokens. The right plot presents the BLEU scores for text reconstruction, which improve as training progresses. These metrics together illustrate the effectiveness of the pretraining phase. 25
6.4 BLEU scores on SQuAD using Qwen3-8B as the base model. BLEU is calculated between the generated and the GT answer. 28
6.5 RepoQA evaluation metrics using Qwen3-8B as the base model. The left graph shows the validation loss, which decreases steadily, indicating successful learning, but not even close to an uncompressed LoRA-FT (LLM-FT here refers to the no-compression `Baseline+FT`). The right graph displays BLEU scores for code reconstruction, where the LoRA fine-tuned baseline excels as well. 28
6.6 BLEU scores and box plot for SWE-bench Verified. The left chart shows that the fine-tuned ICAE model achieves a higher BLEU score than the baseline. However, the right box plot reveals that the baseline model resolves significantly more issues. The comparison is based on 5 evaluations of the models. Each box represents the interquartile range of the resolve rate, and the triangle stands for the mean value. 30
6.7 Comparison of trajectory lengths at termination. This box plot shows that the ICAE-based agent (right) executes a significantly higher number of steps before reaching the context limit, compared to the uncompressed baseline agent (left). The context window limit is 32,768 tokens for this experiment, while there is no limit on the maximum number of steps the agent can take. 32
6.8 BLEU scores comparison across all datasets. This chart consistently shows that the fine-tuned ICAE model (`ICAE-PT+FT`) outperforms the uncompressed baseline in terms of BLEU score. It is also easy to see that the trend is the same for all datasets. 33

List of Tables

4.1 This table maps the names of our four main model variants to their specific encoder and decoder configurations, as depicted in Figure 4.1. The encoder is responsible for compressing the context, while the decoder generates the text. A dash (—) in the "Encoder" column signifies that no compression module is used. Qwen stands for Qwen3-8B model. Pretraining and fine-tuning stages of the training process are described in Section 4.3.1 and Section 4.3.2, respectively. A more comprehensive comparison including other variants is presented in Chapter 6. 14

6.1 Performance of training-free context condensation on SQuAD. This table compares the baseline (hard tokens) against three condensation methods: hard embedding averaging, soft embedding with online generation, and soft embedding with offline regeneration (Regenerate-LLM) averaging. 24

6.2 Autoencoding reconstruction BLEU scores on SQuAD contexts. This table compares the reconstruction quality for models pretrained (PT) on different data subsets (10M vs. 1B tokens of SlimPajama-6B) at various checkpoints. The checkpoint numbers (9k, 12k, 18k) refer to the number of training steps during pretraining. The run marked (main) is used for all the fine-tunings later. Note that here, pretraining is done via LoRA as well. 26

6.3 This table compares the performance of different models with and without compression. Mistral-7B is used as a base model for these experiments. The model names are explained in Section 4.1. We also compare our Fine-Tuned model on SQuAD to the ICAE authors' Fine-Tuned model on PWC [Ge+23]. 27

6.4 Performance on SWE-bench Verified. This table compares different model configurations on resolve rate and inference time. "Qwen" refers to the Qwen3-8B model without any fine-tuning. "del long/all obs-s" stands for deleting long (>256 tokens) or all observations during the inference process – these are the naive baselines. PT stands for pretrained, and FT stands for fine-tuned. The fully fine-tuned Qwen model achieves the highest number of resolved issues, setting the upper bound for performance. While ICAE compression reduces inference time, it significantly lowers the task resolution rate compared to the uncompressed baseline. 30

A.1 Comparison of prompting strategies for disabling thinking in Qwen3-8B. "Cumulative" refers to accumulating the '`<think>...`' prefix in the history, while "Clean" follows the official recommendation of removing it after each step. 44

A.2 Pretraining hyperparameters and configuration 45

A.3 Fine-tuning hyperparameters and configuration 45

A.4 SWE-bench inference configuration 46

Bibliography

[Ant25] Anthropic. *Introducing Claude Opus 4.5.* Anthropic Blog. Accessed: 2025-11-30. 2025.

[Aus+21] J. Austin et al. "Program synthesis with large language models". In: *arXiv preprint arXiv:2108.07732* (2021). https://arxiv.org/abs/2108.07732.

[BPC20] I. Beltagy, M. E. Peters, and A. Cohan. "Longformer: The long-document transformer". In: *arXiv preprint arXiv:2004.05150* (2020). https://arxiv.org/abs/2004.05150.

[Bor+22] S. Borgeaud et al. "Improving language models by retrieving from trillions of tokens". In: *International conference on machine learning.* https://arxiv.org/abs/2112.04426. PMLR. 2022, pp. 2206–2240.

[But+25] N. Butt et al. "Soft tokens, hard truths". In: *arXiv preprint arXiv:2509.19170* (2025). https://arxiv.org/abs/2509.19170.

[Che21] M. Chen. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021). https://arxiv.org/abs/2107.03374.

[Che+23] A. Chevalier et al. "Adapting language models to compress contexts". In: *arXiv preprint arXiv:2305.14788* (2023).

[Cho+24] N. Chowdhury et al. *Introducing SWE-bench Verified.* OpenAI Blog. Accessed: 2025-11-13. 2024.

[Cla+18] P. Clark et al. "Think you have solved question answering? try arc, the ai2 reasoning challenge". In: *arXiv preprint arXiv:1803.05457* (2018).

[Cob+21] K. Cobbe et al. "Training verifiers to solve math word problems". In: *arXiv preprint arXiv:2110.14168* (2021).

[Com+25] G. Comanici et al. "Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities". In: *arXiv preprint arXiv:2507.06261* (2025).

[DG24] T. Dao and A. Gu. "Transformers are ssms: Generalized models and efficient algorithms through structured state space duality". In: *arXiv preprint arXiv:2405.21060* (2024). https://arxiv.org/abs/2405.21060.

[Dao+22] T. Dao et al. "Flashattention: Fast and memory-efficient exact attention with io-awareness". In: *Advances in neural information processing systems* 35 (2022), pp. 16344–16359.

[Dev+19] J. Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers).* https://arxiv.org/abs/1810.04805. 2019, pp. 4171–4186.

[Fen+20] Z. Feng et al. "Codebert: A pre-trained model for programming and natural languages". In: *arXiv preprint arXiv:2002.08155* (2020).

[Gao+20] L. Gao et al. "The pile: An 800gb dataset of diverse text for language modeling". In: *arXiv preprint arXiv:2101.00027* (2020).

[Ge+23] T. Ge et al. "In-context autoencoder for context compression in a large language model". In: *arXiv preprint arXiv:2307.06945* (2023). https://arxiv.org/abs/2307.06945.

[Git21] GitHub. *GitHub Copilot.* https://github.com/features/copilot. 2021.

[GGR21] A. Gu, K. Goel, and C. Ré. "Efficiently modeling long sequences with structured state spaces". In: *arXiv preprint arXiv:2111.00396* (2021). https://arxiv.org/abs/2111.00396.

[Hen+20] D. Hendrycks et al. "Measuring massive multitask language understanding". In: *arXiv preprint arXiv:2009.03300* (2020).

[Hu+22] E. J. Hu et al. "Lora: Low-rank adaptation of large language models." In: *ICLR* 1.2 (2022). https://arxiv.org/abs/2106.09685, p. 3.

[Jia+23] A. Q. Jiang et al. *Announcing Mistral 7B*. https://mistral.ai/news/announcing-mistral-7b. 2023.

[Jim+23] C. E. Jimenez et al. "Swe-bench: Can language models resolve real-world github issues?" In: *arXiv preprint arXiv:2310.06770* (2023). https://arxiv.org/abs/2310.06770.

[Lew+20] P. Lewis et al. "Retrieval-augmented generation for knowledge-intensive nlp tasks". In: *Advances in neural information processing systems* 33 (2020), pp. 9459–9474.

[Liu+24a] J. Liu et al. "Repoqa: Evaluating long context code understanding". In: *arXiv preprint arXiv:2406.06025* (2024). https://arxiv.org/abs/2406.06025.

[Liu+24b] N. F. Liu et al. "Lost in the middle: How language models use long contexts". In: *Transactions of the Association for Computational Linguistics* 12 (2024). https://arxiv.org/abs/2307.03172, pp. 157–173.

[MFG24] T. Munkhdalai, M. Faruqui, and S. Gopal. "Leave no context behind: Efficient infinite context transformers with infini-attention". In: *arXiv preprint arXiv:2404.07143* 101 (2024). https://arxiv.org/abs/2404.07143.

[Pan+24] Z. Pan et al. "Llmlingua-2: Data distillation for efficient and faithful task-agnostic prompt compression". In: *arXiv preprint arXiv:2403.12968* (2024). https://arxiv.org/abs/2403.12968.

[Pap+02] K. Papineni et al. "Bleu: a method for automatic evaluation of machine translation". In: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. https://aclanthology.org/P02-1040.pdf. 2002, pp. 311–318.

[Pat+] S. G. Patil et al. "The Berkeley Function Calling Leaderboard (BFCL): From Tool Use to Agentic Evaluation of Large Language Models". In: *Forty-second International Conference on Machine Learning*. https://openreview.net/forum?id=2GmDdhBdDk.

[Rae+19] J. W. Rae et al. "Compressive transformers for long-range sequence modelling". In: *arXiv preprint arXiv:1911.05507* (2019).

[Raj+16] P. Rajpurkar et al. "Squad: 100,000+ questions for machine comprehension of text". In: *arXiv preprint arXiv:1606.05250* (2016). https://arxiv.org/abs/1606.05250.

[SUV18] P. Shaw, J. Uszkoreit, and A. Vaswani. "Self-attention with relative position representations". In: *arXiv preprint arXiv:1803.02155* (2018). https://arxiv.org/abs/1803.02155.

[Son+24] Y. Song et al. "Agentbank: Towards generalized llm agents via fine-tuning on 50000+ interaction trajectories". In: *arXiv preprint arXiv:2410.07706* (2024). https://arxiv.org/abs/2410.07706.

[Su+24] J. Su et al. "Roformer: Enhanced transformer with rotary position embedding". In: *Neurocomputing* 568 (2024). https://arxiv.org/abs/2104.09864, p. 127063.

[Tou+23] H. Touvron et al. "Llama 2: Open foundation and fine-tuned chat models". In: *arXiv preprint arXiv:2307.09288* (2023).

[Vas+17] A. Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017). https://arxiv.org/abs/1706.03762.

[Wan+20] S. Wang et al. "Linformer: Self-attention with linear complexity". In: *arXiv preprint arXiv:2006.04768* (2020). https://arxiv.org/abs/2006.04768.

[Wan+24] Y. Wang et al. “Natural is the best: Model-agnostic code simplification for pre-trained large language models”. In: *Proceedings of the ACM on Software Engineering* 1.FSE (2024). https://arxiv.org/abs/2405.11196, pp. 586–608.

[Wan+21] Y. Wang et al. “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation”. In: *arXiv preprint arXiv:2109.00859* (2021).

[Web+24] M. Weber et al. “Redpajama: an open dataset for training large language models”. In: *Advances in neural information processing systems* 37 (2024). https://arxiv.org/abs/2411.12372, pp. 116462–116492.

[Wik25] Wikipedia. *The Pile (dataset)*. https://en.wikipedia.org/wiki/The_Pile_(dataset)#DMCA_takedown. Accessed: 2025-11-26. 2025.

[Yan+25a] A. Yang et al. “Qwen3 technical report”. In: *arXiv preprint arXiv:2505.09388* (2025). https://arxiv.org/abs/2505.09388.

[Yan+24] J. Yang et al. “Swe-agent: Agent-computer interfaces enable automated software engineering”. In: *Advances in Neural Information Processing Systems* 37 (2024). https://arxiv.org/abs/2405.15793, pp. 50528–50652.

[Yan+25b] J. Yang et al. “Swe-smith: Scaling data for software engineering agents”. In: *arXiv preprint arXiv:2504.21798* (2025). https://arxiv.org/abs/2504.21798.

[Yao+22] S. Yao et al. “React: Synergizing reasoning and acting in language models”. In: *The eleventh international conference on learning representations*. https://arxiv.org/abs/2210.03629. 2022.

[Zah+20] M. Zaheer et al. “Big bird: Transformers for longer sequences”. In: *Advances in neural information processing systems* 33 (2020). https://arxiv.org/abs/2007.14062, pp. 17283–17297.

[Zel+19] R. Zellers et al. “Hellaswag: Can a machine really finish your sentence?” In: *arXiv preprint arXiv:1905.07830* (2019).

[Zen+24] A. Zeng et al. “Agenttuning: Enabling generalized agent abilities for llms”. In: *Findings of the Association for Computational Linguistics: ACL 2024*. https://arxiv.org/abs/2310.12823. 2024, pp. 3053–3077.

[Zha+24] R. Zhao et al. “Position IDs Matter: An Enhanced Position Layout for Efficient Context Compression in Large Language Models”. In: *arXiv preprint arXiv:2409.14364* (2024). https://arxiv.org/abs/2409.14364.

[Zhu+23] B. Zhuang et al. “A survey on efficient training of transformers”. In: *arXiv preprint arXiv:2302.01107* (2023).