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

Submitted:
Munich, 31. 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, 31. November 2025 Kirill Gelvan

Zusammenfassung

Eine kurze Zusammenfassung der Arbeit auf Deutsch.

Abstract

A brief abstract of this thesis in English.

Contents

1 Introduction

note: here I write motivation. From problem to soluiton.

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 requiring 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). SWE agents must perform tasks such as examining files and directories, reading and modifying parts of files, and navigating complex codebases. 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 major 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 can 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. Additionally, irregular sparsity patterns often lead to hardware inefficiencies, as modern accelerators are optimized for dense operations. 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 duality 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+21] 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, leading to high inference latency and cost. 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+23].

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. 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 reduce latency and memory, and avoids external retrieval steps. Figure 1.1 illustrates the core concept.

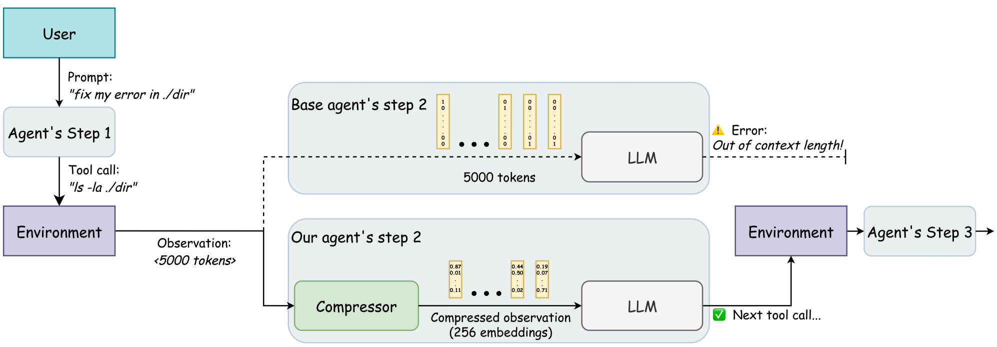


Figure 1.1 Comparison between a base agent and our agent approach for handling large observations exceeding the LLM's context length. The base agent fails when processing long observations directly, while our agent successfully compresses the observation into a fixed set of embeddings before LLM processing,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

The primary goal of context condensation is to leverage this potential density to enable 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 latency and reduced GPU memory cost 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:

RQ1: *Does implicit context condensation allow for the completion of longer agentic trajectories?*

A secondary question explores whether the benefits of this approach on general comprehension tasks transfer to the domain of agentic software engineering:

RQ2: *How does the performance of implicit context condensation on standard NLP benchmarks translate to the distinct demands of agentic software engineering tasks?*

RQ2 VARIANT 1: How does the performance of implicit context condensation for agentic software engineering tasks compare to its performance on standard NLP benchmarks?

RQ2 VARIANT 2: Are implicit compression techniques effective when applied to multi-step agentic software engineering workflows?

2 Background

note: here i write explanations of things. What is needed to understand ICAE? But without e.g. transformer.

2.1 Transformer Architecture and Positional Bias

Self-attention mechanism is well-known and described in detail in many 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+21].

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, but 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+21].

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+25].

TODO: write about it more!!! we would talk about it later in the thesis?

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+21]. LoRA freezes the pre-trained weight matrices and introduces trainable low-rank updates, yielding substantial parameter savings while preserving the base model's knowledge [Hu+21]. 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+21]) 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+21].

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+25]. 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+25].

Agentic toolcall examples (from [Pat+25]):

- 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 toolcall 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. 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.
- **Code Simplification:** Refactoring code to be more readable/efficient or shorter without changing its functionality.
- **Code Translation:** Migrating code from one programming language to another (e.g., Python to JavaScript).
- **Bug Detection and Repair:** Identifying potential bugs in a piece of code and suggesting fixes.
- **Code Editing:** Modifying existing code based on high-level instructions [Gee25].
- **Code Documentation and Summarization:** Generating natural language descriptions from code, such as docstrings or summaries, and answering questions about its functionality.
- **Agentic Code Tasks:** Autonomously planning and executing complex, multi-step software engineering tasks [Lea25].

These capabilities are often evaluated on benchmarks such as HumanEval [Che+21] 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, which are powered by models like OpenAI's Codex. 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.

In-Context Autoencoder (ICAE) As an approach to implicit context compression, the In-Context Autoencoder (ICAE) introduced by Ge et al. [Ge+24] 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 BLEU ≈ 99% on autoencoding across Llama-2 models and small 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 latency 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 finetuning over SWE-bench-like tasks, which Ge et al. [Ge+24] did not target; for later chapters, this section supplies the technical background (slots, compression ratios, fidelity–throughput trade-offs) we build on.

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 like 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 token importance than causal models.

The method demonstrates impressive performance, achieving substantial compression with minimal fidelity loss. 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 with 5× compression, on par with the full uncompressed baseline. This efficiency translates to significant end-to-end latency reductions 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 our implicit, continuous compression. 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 like QA and summarization, whereas our work targets the distinct challenges of multi-turn, agentic software engineering trajectories.

SlimCode Yin et al. [Yin+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, Yin et al. [Yin+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 Yin et al. [Yin+24] offer a powerful, 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, Chatzianastasis et al. [Cha+24] 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 on 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, ARC, and MMLU), soft-token training maintains it. A striking example is a 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 holds for our memory tokens as well.

Infini-Attention As an architectural modification for long contexts, Infini-attention [Las+24] 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. This architectural change yields strong empirical results.

On long-context language modeling benchmarks like PG19 and ArXiv-math, 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 pre-training. Our approach, in contrast, remains compatible with existing LLMs by using ICAE-style memory tokens. We focus on finetuning these representations for agentic coding tasks, whereas Laskin et al. [Las+24] 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+23] 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+23] 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 finetuning with an ICAE-style memory system, specifically targeting the bottlenecks of long code contexts and multi-turn interactions found in software engineering scenarios.

Positioning Summary

Synthesis of Prior Work The literature presents several distinct strategies for managing long contexts. Explicit methods such as LLMLingua-2 [Pan+24] and SlimCode [Yin+24] achieve compression by selectively removing tokens, offering model-agnostic benefits for tasks like question answering and code summarization, but risk information loss for generative tasks. Architectural solutions like Infini-attention [Las+24] fundamentally alter the Transformer to handle virtually infinite sequences, but require extensive pre-training and deviate from standard model structures. In contrast, methods like ICAE [Ge+24] and those for continuous reasoning traces [Cha+24] explore the use of continuous, learned representations—either as compressed context summaries or as reasoning traces—demonstrating that latent vectors can effectively encode complex information without architectural changes. Finally, AgentTuning [Zen+23] highlights the value of post-training on agent trajectories to improve tool-use capabilities, but does not address the input context bottleneck.

Our Contribution. This thesis integrates and extends concepts from these parallel lines of research. Our primary contribution is the novel combination of three key elements: (i) *implicit, embedding-space compression*, following the In-Context Autoencoder (ICAE) paradigm [Ge+24], to create a compact and learnable representation of long code contexts; (ii) *agent-trajectory finetuning*, inspired by Zeng et al. [Zen+23], but specifically tailored to the domain of multi-step software engineering tasks; and (iii) *a specialized evaluation setting* based on SWE-bench, which requires robust handling of long, evolving codebases and complex tool interactions. This synthesis distinguishes our work from prior art, which has not jointly addressed implicit compression and agentic fine-tuning for code.

Rationale for Methodological Choices. Our decision to pursue an ICAE-based approach over alternatives is deliberate. We avoid explicit token deletion [Pan+24; Yin+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 forgo architectural modifications [Las+24] 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 finetuning to create an agent optimized for long-context software engineering challenges.

4 Methods

4.1 ICAE for Agentic Context Management

The In-Context Autoencoder (ICAE) [Ge+24] 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, improving latency and memory footprint while preserving fidelity for downstream tasks. The number of memory tokens controls the compression ratio, and their placement influences how the decoder accesses the stored information [Ge+24].

Figure 4.1 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 like LoRA [Hu+21] are used to adapt the encoder while keeping the base model's capabilities intact.

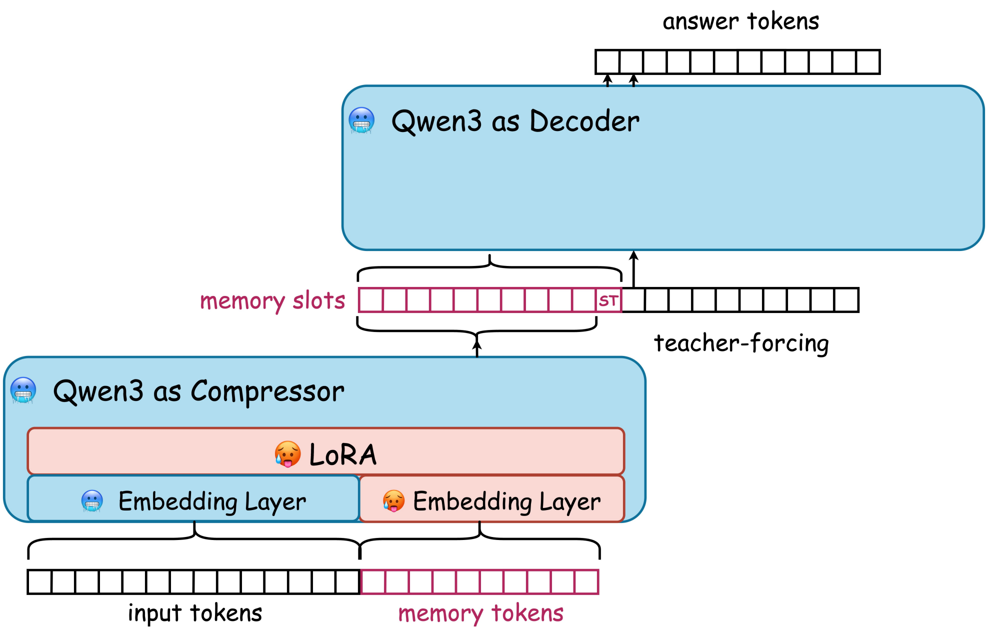


Figure 4.1 In-Context Autoencoder (ICAE) framework architecture.

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 uses ICAE to compress long observations, keeping the overall context manageable without losing critical information.

4.2 Training Methodology for Agentic ICAE

The process for training ICAE in an agentic scenario is depicted in Figure 4.2. The training data consists of pre-recorded expert trajectories, which are sequences of alternating actions and observations.

At the start of the interaction, an uncompressed user prompt is provided. Then, a trajectory unfolds as an agent takes an action (tool call), 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, ensuring that the decoder model never processes lengthy raw text. Instead, it operates on compact embedding representations.

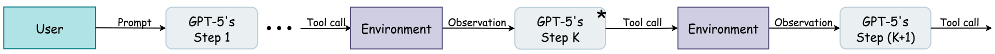


Figure 4.2 Overview of applying ICAE to agentic trajectories during fine-tuning.

Figure 4.3 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.

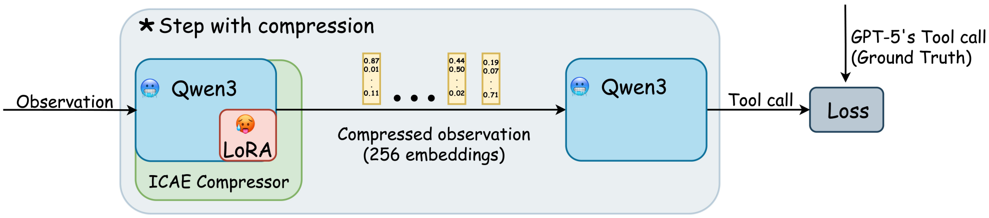


Figure 4.3 A single fine-tuning step for the agentic ICAE model.

4.2.1 Pretraining (PT)

The first stage follows the original ICAE formulation [Ge+24], pretraining the encoder on a large, general-purpose text corpus. Two self-supervised objectives are used:

(i) **Autoencoding (AE)**, where ICAE restores the original input text from its memory slots.

(ii) **Language Modeling (LM)**, which predicts the continuation of a context to improve generalization.

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.

4.2.2 Fine-Tuning (FT)

After pretraining, the ICAE encoder is fine-tuned on a dataset of agentic trajectories. Again, 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 history.

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. 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.

For example, consider the following trajectory:

1. **System prompt** (text): initial instructions and tool descriptions.

2. **Task description** (text): user-provided issue or goal.
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 this example, the encoder is applied twice (to compress observations 2 and 3, highlighted in yellow), while the decoder generates five actions. The model then is able to predict actions from a history where long observations have been replaced by their compact memory representations.

4.2.3 Training Process and Model Variants

Figure 4.4 provides a comprehensive overview of the full 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 the **Baseline**. This is a ready-to-use model, not one with random weights.

The figure illustrates 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 [Tog23], resulting in the **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.

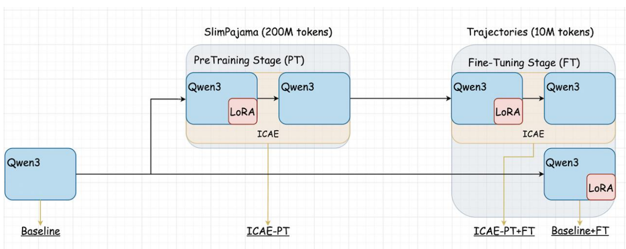


Figure 4.4 Full training process overview, illustrating the derivation of Baseline, ICAE-PT, Baseline+FT, and ICAE-PT+FT models.

5 Evaluation

5.1 Datasets

5.1.1 SWE-bench

We have chosen to use the SWE-bench [Jim+24] dataset for our experiments, as it is a well-known and popular benchmark for evaluating the performance of software engineering agents on real-world tasks. It contains approximately 3000 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 rigorous and 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.

5.1.2 Agentic Trajectories for Fine-Tuning

High-quality training data is crucial for fine-tuning capable software engineering agents. We require expert demonstrations in the form of agentic trajectories. To generate them at scale, we use the data collected in [Yan+25b].

While SWE-smith provides the framework for generating tasks, we use it to generate expert trajectories for fine-tuning our agent. Specifically, we use a powerful teacher model, Claude 3.7 Sonnet, to solve tasks collected by the authors and within the SWE-smith environment. 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. 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+23].

Interaction Protocol and Tools 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.2. 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. We intentionally did not want to complicate our research with more complex scaffolds like SWE-agent [Yan+24] to focus on the core of the compression problem.

5.1.3 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.

For our experiments, we use SQuAD 2.0 [**rajpurkar2018know**], which extends the original dataset by including unanswerable questions. This modification challenges a model not only to extract the correct answer when available but also to correctly identify when no answer exists within the context. This setup makes it a robust benchmark for evaluating how well our compressed representations retain the necessary fidelity for fine-grained semantic understanding.

The choice of SQuAD is motivated by its widespread adoption and focus on extractive answers. In contrast, the original In-Context Autoencoder work [Ge+24] used the PWC dataset, which is less common and focuses more on generating longer, abstractive answers rather than extracting specific spans of text.

5.1.4 RepoQA

To assess how well our compression method preserves high-fidelity information in code, we utilize the RepoQA benchmark [Liu+24]. RepoQA is specifically designed to evaluate a model's ability to understand and reason about code at the repository level. 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 genuine comprehension of both the code's semantics and the description's intent.

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.

5.2 Quality Metrics

We employ a variety of metrics to evaluate our approach, tailored to the specific demands of each task. 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 6.10).

5.2.1 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). This provides a direct, end-to-end measure of the agent's practical software engineering capabilities. We also report the mean tool-call generation time to quantify the efficiency gains from our compression method. We do not use trajectory length as a metric, as it can be misleading; agents can enter repetitive loops that inflate length without making progress.

5.2.2 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. 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.

6 Experiments

note: I guess here I write not only experiments, but also the results of the experiments?
TODO: where to write about enc-lora+dec-lora experiments?

6.1 Initial Prototype Experiments: The Necessity of Training

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 the one-hot vectors, yielding a convex combination over the vocabulary.

In the soft-embedding setting, we remove the argmax and delete the input embedding layer so the model consumes continuous mixtures rather than token lookups. With Qwen2.5-4B (tied input/output embedding matrices), we feed this vector directly as the next-step input (i.e., into the stack where the removed embedding layer would have produced a token embedding). Figures 6.1a–6.1b explain this online injection point and its relation to the standard token pathway.

Online soft-embedding pathway We implemented the soft pathway by bypassing token sampling and the embedding lookup:

1. run a normal decode step to obtain logits
2. compute probabilities and the corresponding expected embedding
3. insert that continuous vector directly as the next-step input

This is done via KV-cache manipulation to inject the continuous embeddings directly into the generation pipeline. The intent was to test whether context can be compressed into a small number of continuous vectors without any additional training or adapters (see Fig. 6.1a–6.1b).

Regenerate-LLM offline pathway We have thought that in the described case, generating different answers for the first embedding iteratively might be collapsing and the next results, which could decreased score in metrics. So we tried a method that we called regenerate-llm. Given an input sequence, we prompt the model to reproduce that sequence under teacher forcing and, at each step, record all the output embeddings. At inference time, instead of recomputing embeddings online, we reuse the saved embeddings as the context representation.

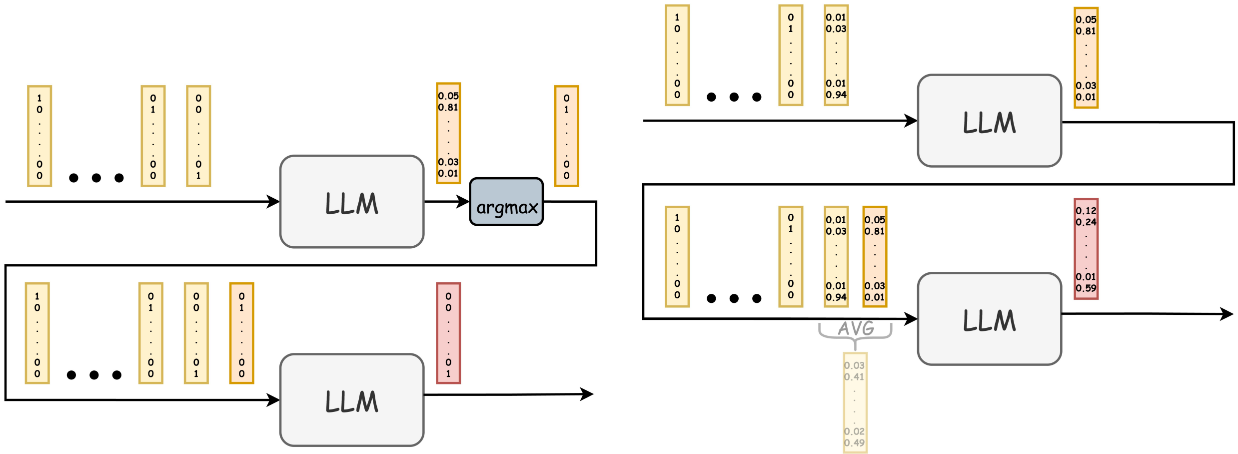
Results and details Table 6.1 reports SQuAD performance under these condensation strategies. These results, together with the implementation schematics in Fig. 6.1a–6.1b, establish that replacing hard tokens with untrained condensed mixtures (whether via online or via offline regenetation) substantially degrades QA accuracy, motivating the trained condensation methods that follow.

6.2 Basic experiments with training

Having established in §5.1 that naively averaging ("avg, ×2") adjacent embeddings sharply degrades QA quality, we next asked whether a learned projection inserted at the embedding interface could recover performance under the same 2× compression ratio. The motivation was that, if the embedding manifold is non-linear, a trained projection might learn a geometry-preserving down-map that simple averaging

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 Baseline against averaging techniques (Prompt–Q–C)



(a) Visualization of the "without training" approach, p1 **(b)** Visualization of the "without training" approach, p2

Figure 6.1 Visualization of the "without training" approach.

cannot provide. The baseline (Step 1) and the "soft/hard mix works" observation (Step 2) are illustrated in 6.2 and frame this question empirically.

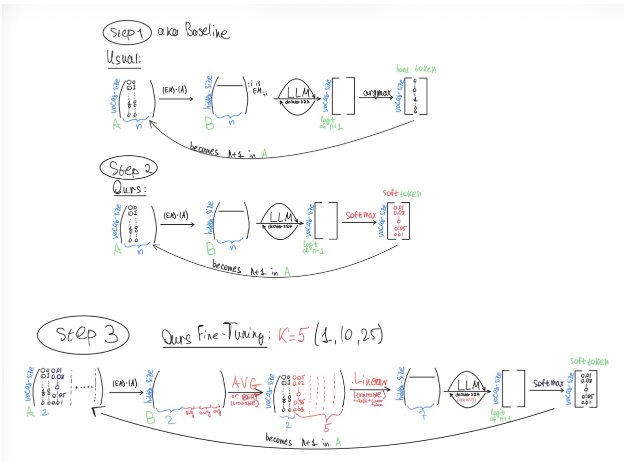


Figure 6.2 Visualization of the baseline approach, step 1-3; should be redrawn

Architectural variants We explored minimal-capacity projections that compress two adjacent hidden vectors into one "soft token" acceptable to the frozen decoder. The first family was a **linear projector**, $g_\theta : \mathbb{R}^{2d} \to \mathbb{R}^d$, applied to $[e_{2t-1}; e_{2t}]$ with optional residual gating on the arithmetic mean to stabilize scale. The second family was a shallow non-linear MLP (one–two layers with GELU), again mapping $2d \to d$. A third variant inserted a full BERT encoder **[devlin2018bert]** (12 layers, 768-dimensional hidden states) to process the concatenated embeddings $[e_{2t-1}; e_{2t}]$ and produce a single compressed representation, which was then projected back to the decoder's dimensionality. This encoder-based approach provided substantially higher capacity than the shallow projections, allowing the model to learn more complex compression patterns through its multi-layer self-attention mechanism. These designs follow the "trainable averaging" schematics shown on 6.3.

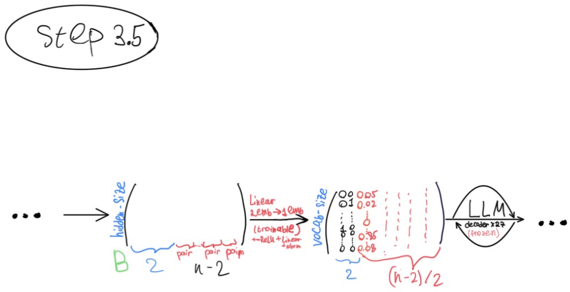
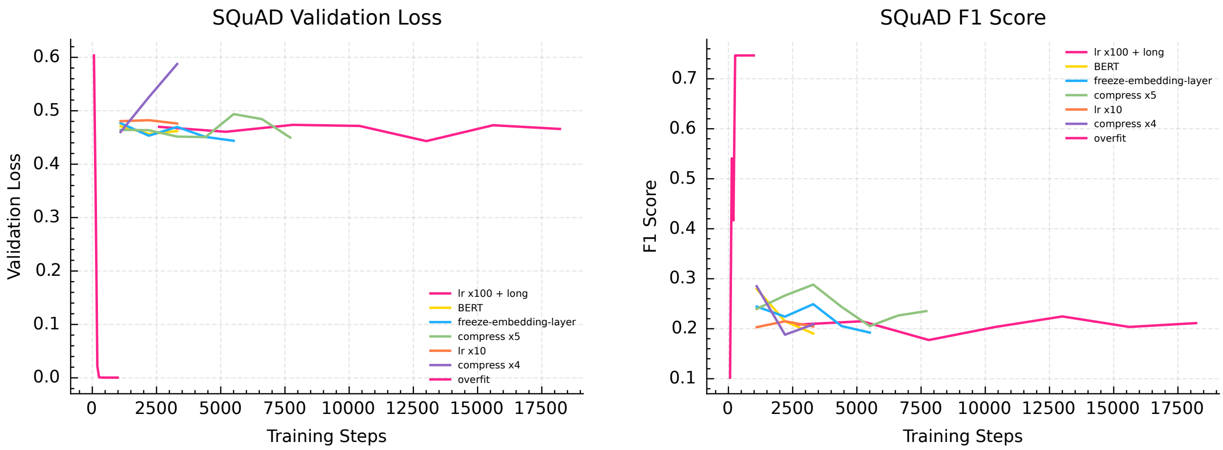


Figure 6.3 Visualization of the experimental setup, step 3.5; should be redrawn

Training protocol and results All experiments used the SQuAD [**squad**] "context-embed" setting from §5.1, keeping Qwen3-8B frozen and training only the projection parameters via token-level cross-entropy on answer continuations. After verifying the pipeline on a single batch, we trained on SQuAD train and evaluated on validation. Across linear, MLP, and BERT projections, models quickly overfit but did not generalize: validation loss flattened after early improvement (see Figure 6.4), and EM/F1 remained well below the hard-token baseline, never closing the large gap to the no-compression control (e.g., the ∼ 50%–80% relative F1 drop visible for averaging).



(a) SQuAD Validation Loss

(b) SQuAD F1 Score

Figure 6.4 Validation loss and F1 score curves for the SQuAD generalization experiment.

Ablation studies We varied:

- projection type (linear vs. 1- or 2-layer MLP),
- normalization (pre/post LayerNorm, scale-preserving residual gates),
- regularization (weight decay, dropout), and
- the decision to re-project via vocabulary space versus staying in hidden space.

We also tried unfreezing the token embedding table while keeping the transformer blocks frozen. None of these changes altered the qualitative outcome: projections still overfit quickly and failed to surpass the baseline (6.2) in EM/F1. These negative findings echo the summary on the checkpoint deck ("all our fine-tuning techniques do not recover quality").

Hypotheses for the failure Two factors appear decisive.

Firstly, we hypothesize that if the embedding manifold is not smooth, merging two embeddings into one may lose critical geometric structure that the frozen decoder relies on, making it impossible for a simple projection to preserve the information needed for downstream tasks.

Secondly, we hypothesize that the fundamental limitation is the expressive power of the overall model architecture. Even though the BERT encoder contains 0.1B parameters (more than the projections), it lacks the capacity to learn effective compression when paired with a frozen 8B decoder.

This observation motivated our transition to the ICAE framework, where training LoRA adapters ($\approx$2% of the 8B model's weights) provides substantially greater expressive power by modulating the decoder's internal representations, despite involving fewer trainable parameters than the full BERT encoder.

Summary The experiments illustrated in 6.3 demonstrate that trainable projections are insufficient to recover QA performance under $\approx 2\times$ compression. These results motivated us to explore larger-scale training approaches and alternative solutions such as the ICAE framework.

6.3 ICAE Pretraining and Results on General Text Reconstruction

We evaluate ICAE autoencoding (AE) pretraining using Qwen3-8B as the base model. During AE, the encoder compresses input contexts at a fixed $\times 4$ 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 `pretrain_2607_1024_4_1B/checkpoint-12000` is selected as the main run and used for fine-tuning.

Run	Checkpoint (# steps)	Compression	BLEU (mean, n=100)
Qwen3-8B/full (no ICAE)	18k	×1	0.867
ICAE PT (pretrain_1207)	9k	×4	0.942
ICAE PT (pretrain_1207)	12k	×4	**0.964**
ICAE PT (pretrain_1207)	27k	×4	0.902
ICAE PT (pretrain_2607_1024_4_1B)	9k	×4	0.909
ICAE PT (pretrain_2607_1024_4_1B)	12k (main)	×4	0.936
ICAE PT (pretrain_2607_1024_4_1B)	18k	×4	0.928

Table 6.2 Autoencoding (AE) reconstruction BLEU on SQuAD contexts (100-sample evaluations only). The `pretrain_2607_1024_4_1B` 12k checkpoint is the main model used for FT.

We have also experimented with compressing 16 tokens into 4 on average instead of 1024 into 256. It which 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 material for code, where near-lossless reconstruction is likely a prerequisite for downstream stability.

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

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 Tasks

While the original ICAE work [Ge+24] demonstrated promising results on their proprietary PWC dataset, we sought to validate these findings on a well-established benchmark to ensure generalizability and facilitate fair comparison with existing approaches. To address the limitation of the authors'-crafter PWC dataset, we conducted fine-tuning experiments on the Stanford Question Answering Dataset (SQuAD) [**squad**], a widely recognized benchmark in the question answering literature that provides standardized evaluation protocols and enables reproducible comparisons.

The fine-tuning procedure follows the encoder-decoder structure established during pretraining, with modifications to accommodate the question answering objective. During fine-tuning, we feed compressed context representations produced by the encoder into the decoder, concatenated with uncompressed question tokens. The decoder then generates answer from this mixed representation, while throught backpropagation we update the LoRA weights of the encoder. This setup encourages the encoder to learn compression strategies that preserve information necessary for the decoder to extract answers from. We employ LoRA fine-tuning [Hu+21] with identical hyperparameters to those used in the original ICAE pretraining to maintain consistency and enable fair comparison.

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 [**mistral7b**] 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+24], 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 Exact Match (EM) and F1 scores on the SQuAD validation set. 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.

Model	**Compression**	**Exact Match**	**F1**
Mistral-7B (no FT)	×1	49	68
LoRA-FT baseline	×1	59	65
ICAE FT (PwC, authors)	×1.7 ± 0.7	41	57
ICAE FT (SQuAD, ours)	×1.7 ± 0.7	**69**	**73**

Table 6.3 ICAE averaging on SQuAD

Several important observations emerge from these results. First, our ICAE fine-tuning on SQuAD achieves the highest performance across both metrics, with an F1 score of 73 and Exact Match of 69, representing substantial improvements over both the untrained baseline and the LoRA fine-tuned control. Notably, this performance gain occurs despite operating under approximately 2× compression, suggesting that the learned compression strategy successfully preserves task-critical information while reducing computational overhead.

Most strikingly, the compressed ICAE model not only outperforms the uncompressed baseline (already valuable given the computational savings) but also surpasses the uncompressed LoRA fine-tuned baseline by 8 F1 points (73 versus 65). This result is quite unexpected: compression typically implies information loss, yet here the compressed model demonstrates superior performance to its uncompressed counterpart trained with identical LoRA fine-tuning procedures. We hypothesize that this advantage stems from the compression mechanism's ability to filter and retain only the most importnat information from the context, effectively introducing a beneficial inductive bias. This might help the model focus on task-relevant features while discarding noise or redundant details.

Moreover, the ICAE model fine-tuned on PWC exhibits notably lower performance than all other configurations, achieving F1 and EM scores of 57 and 41 respectively. This result indicates that the PWC-fine-tuned model fails to generalize effectively to the SQuAD evaluation distribution, despite achieving strong performance on its training domain. This performance gap suggests potential overfitting to the specific characteristics of the PWC dataset, which may have properties that do not transfer well to standard question answering benchmarks.

These results provide encouraging evidence for the viability of applying ICAE-based compression to downstream tasks beyond the original evaluation domain. The strong performance on SQuAD, combined with the observed compression benefits, motivated our subsequent investigation of the ICAE framework in more complex, agentic settings where context length presents significant computational challenges.

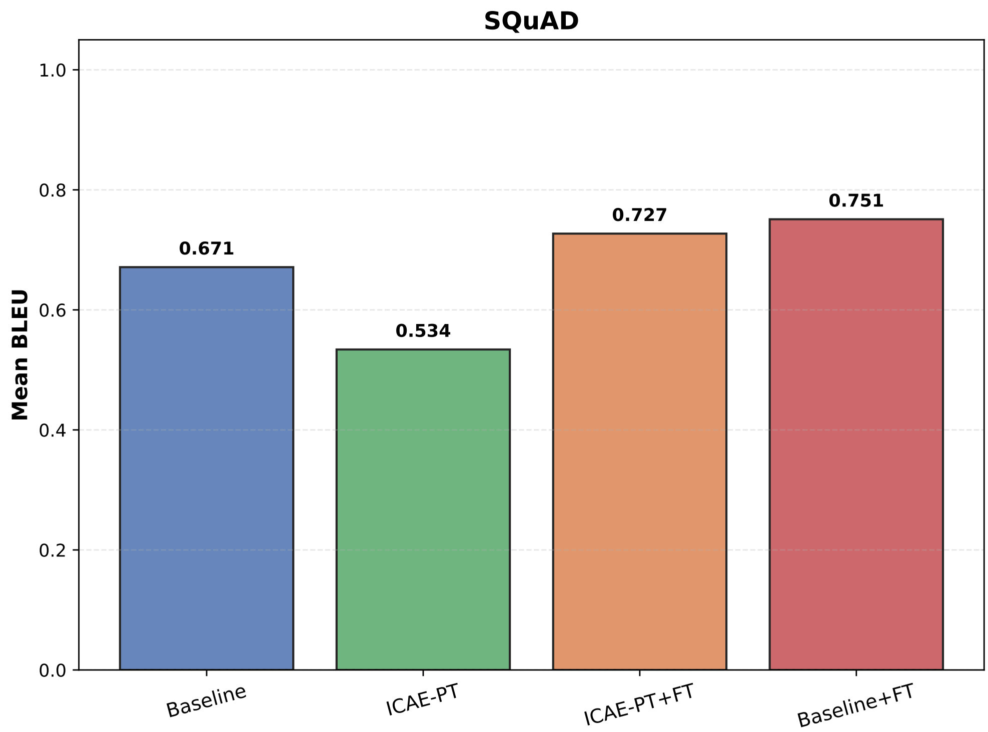
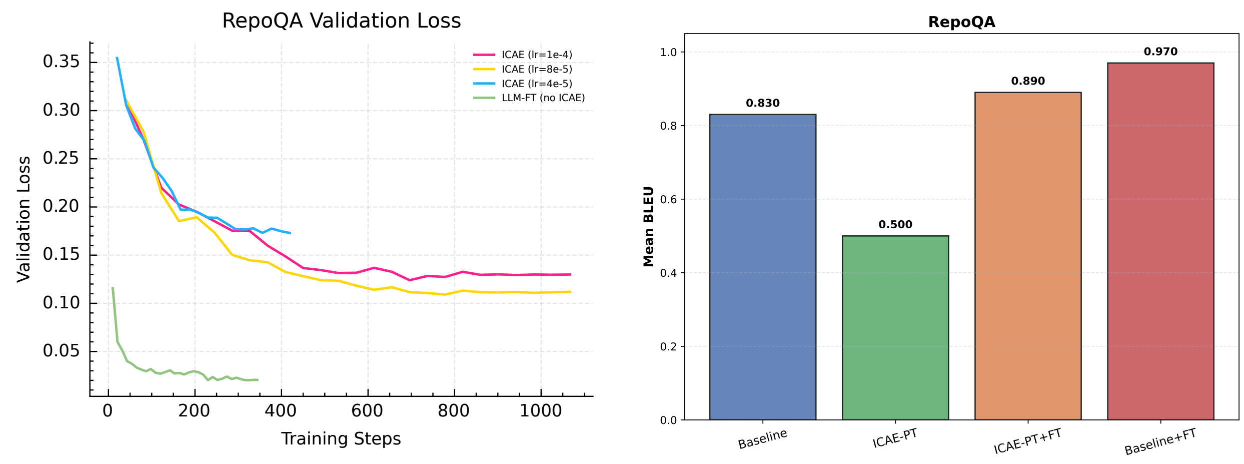


Figure 6.5 BLEU scores on SQuAD

6.5 Checking code data with RepoQA dataset [Liu+24]

To further assess the capabilities of our ICAE model, particularly its ability to handle long, structured, and highly technical contexts, we evaluated it on the RepoQA dataset [Liu+24]. This experiment was designed to test the hypothesis that ICAE can effectively compress entire code repositories while preserving the granular details necessary for complex reasoning and question answering about the code. Given that software development tasks often require a deep understanding of a large codebase, this evaluation serves as a critical stress test for our context compression approach.



(a) RepoQA Validation Loss

(b) BLEU scores on RepoQA

Figure 6.6 RepoQA evaluation metrics.

6.6 [Very main part?] ICAE Fine-Tuning and Evaluation 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 **??**. Figures 4.2 and 4.3 in that chapter illustrate the training process, where the model is fine-tuned on expert 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.

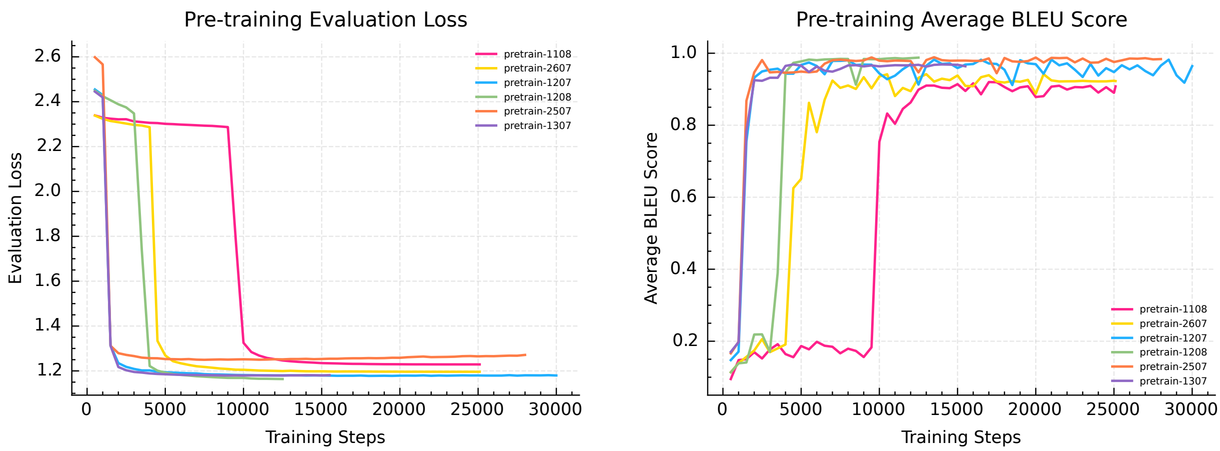
6.6.1 Experimental Setup

We reimplemented the ICAE framework from scratch, building upon the original architecture [Ge+24] with several modifications for improved efficiency and reproducibility. Our implementation uses the Qwen3 model family (specificaly Qwen3-8B) as the base LLM, with LoRA adaptation applied to the attention matrices (q_proj and v_proj) using a rank of 128. To make things simpler, we explicitly disable long-form "thinking" during decoding by inserting the token sequence `\think \think` with a newline marker (`\n`) in between. This is exactly how the authors of [**qwen3**] recommend disabling thinking. This prefix makes the model "believe" it has already produced intermediate thoughts (empty in this case), while in fact no additional content is generated. It should also be noted that the authors of [**ge_context_2024**] have only worked with older models (such as Llama2 and Mistral-v0.1), and only publish the weights of the latter.

Pretraining. This pretraining was performed on the dataset SlimPajama-6B [**pajama6b**][1] using a combination of autoencoding and language modeling objectives. It is a common text dataset, that is used for pretraining LLMs, consisting of 6 billion tokens (a random 1% of the original 627B tokens). The dataset that authors used in their original paper "The Pile" was unavailable.

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

Fine-tuning. After pretraining, ICAE is fine-tuned using trajectories from SWE-bench Verified. Fine-tuning uses a memory size of 256 tokens. If the observation is longer than 1024 tokens, we apply the encoder multiple times, preserving the compression ratio, following the authors of [**ge_context_2024**]. On figure **??** you can notice the sudden drop of the loss. This is phenomenon found in [**PI**]. 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 appers if we apply the positional encodings manipulations.



(a) Evaluation Loss

(b) Average BLEU Score

Figure 6.7 Pre-training evaluation metrics across different model configurations.

Training Infrastructure. Training was performed on a single NVIDIA H200 GPU, requiring approximately 1 day and 15 hours for pretraining and 3 days for fine-tuning due to the computational complexity of the autoencoding objective and resulting lack of effective batching opportunities. Detailed hyperparameters and training configurations are provided in Appendix A.5.

6.6.2 Compression Strategy

The encoder is only applied if all of the next are true:

- The text is an observation (i.e. response from the environment, not the action)
- The text is longer or equal than 256 tokens

The actions are short and do not require compression, while the observations can be long and complex. Due to the nature of the ICAE framework, the compression produces a fixed number of memory tokens, which we fix at 256. Applying the encoder to the shorter texts would be a waste of resources and would create a mismatch between the training and the validation settings.

6.6.3 Results

The results of our experiments are presented in Table 6.4. The table compares several model configurations across five oder is only applied if akey metrics:

- **Encoder**: The model used to compress observations. "—" indicates no encoder and thus no compression.
- **Decoder**: The model used to generate the next tool call. We test the base Qwen3-8B ("Qwen"), a LoRA fine-tuned version ("Qwen-LoRA-FT"), and a fully fine-tuned version ("Qwen-Full-FT").

- **Accuracy**: The token-wise accuracy on the SWE-bench Verified test set.
- **Resolved**: The number of issues successfully resolved out of 500.
- **Time**: The mean time in seconds to generate a tool call.

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 ("Full-FT") achieves the highest performance, resolving 86 issues and setting the upper bound for this architecture. The LoRA fine-tuned variant ("LoRA-FT") provides a more parameter-efficient alternative, resolving a respectable number of issues. The base Qwen3-8B model without any fine-tuning ("Qwen") serves as the most direct point of comparison for our ICAE models, as they use this same frozen model as the decoder. It resolves 26 issues, establishing a solid baseline for an off-the-shelf model on this task.

Encoder	Decoder	Acc. ↑	Resolved (/500) ↑	Time (s) ↓
Naive Baselines				
del long obs-s	Qwen	0.8873	1	0.44
del all obs-s	Qwen	0.8802	0	0.39
Baselines				
—	Full-FT	**0.9484**	**86**	1.24
—	LoRA-FT	0.9118	10 (or 25 with more layers)	1.24
—	Qwen	0.8967	<u>26</u>	1.23
ICAE-FT Compression				
ICAE (LoRA-FT)	Qwen	0.9020	11 - is it broken or not?	**1.12 (0.31+0.81)**
ICAE (LoRA-FT)	LoRA-FT	<u>0.9263</u>	3 (lora is broken? 10)	1.13
ICAE-PT Compression (Ablation)				
ICAE (LoRA-PT w/ Full-FT)	Full-FT	0.9219	—	—
ICAE (LoRA-PT w/)	Qwen	0.8808	—	—

Table 6.4 No think bug table. ICAE variants. All encoders and decoders are Qwen3-8B. FT=FineTuning, PT=PreTraining

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 slight improvement in token-wise accuracy over the uncompressed Qwen baseline (0.9020 vs. 0.8967) and a modest 10% reduction in generation time. However, this configuration sees a significant drop in task performance, resolving only 11 issues compared to the baseline's 26. A similar trend holds when using a LoRA-FT decoder, where accuracy increases but the resolved rate plummets. This suggests that while compression is efficient, it loses critical information, and that token-wise accuracy is a poor proxy for end-to-end task success in this agentic setting.

We also experimented with an ICAE encoder that was only pretrained on general text (ICAE-PT) and not fine-tuned on SWE-bench data. Our hypothesis was that a general-purpose compressor could be a shareable artifact, applicable to new domains without task-specific fine-tuning. However, this ablation proved unsuccessful. The models using the ICAE-PT encoder performed worse than even the naive baselines in terms of token accuracy and failed to resolve any issues. This demonstrates that for a complex domain like software engineering, the encoder must be fine-tuned on in-domain data to learn what information is important to preserve during compression.

The key negative result is the sharp decline in resolved issues when using ICAE-FT compression (11) compared to the uncompressed baseline (26). We hypothesize that this performance degradation is a direct

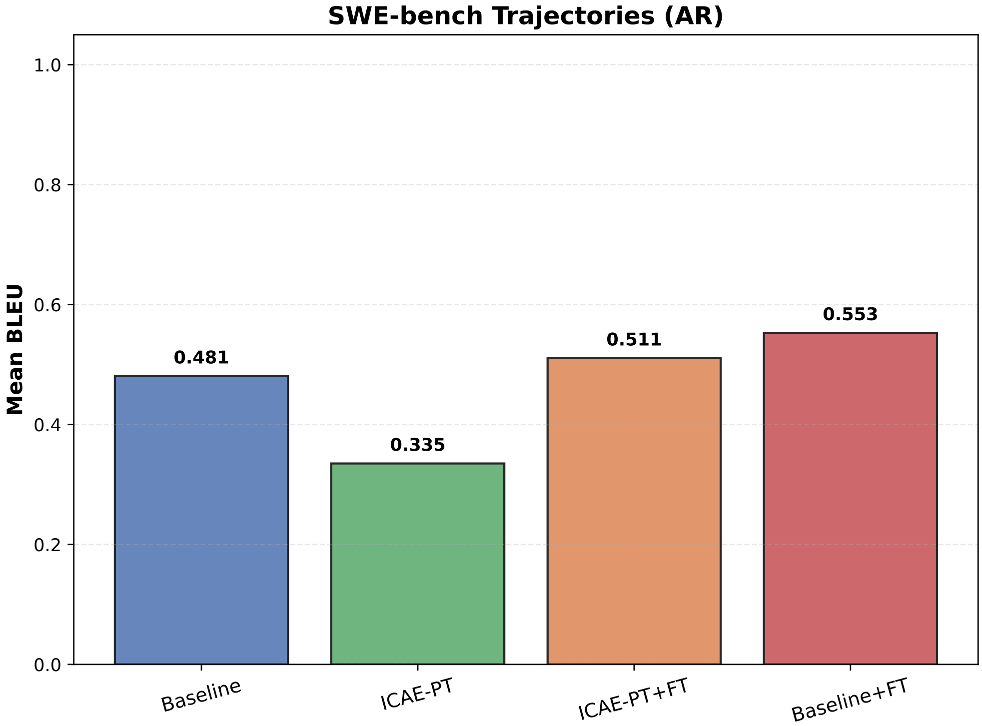


Figure 6.8 BLEU scores on SWE-bench

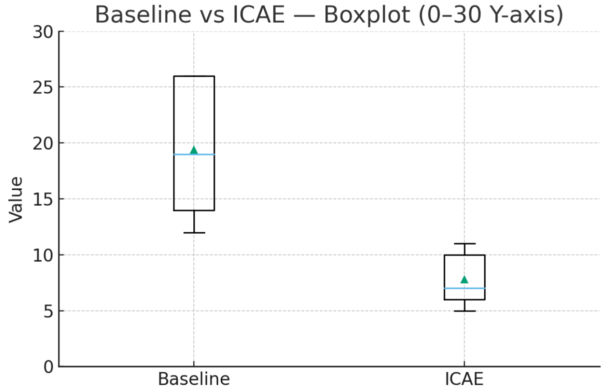
result of information loss during the compression stage. As demonstrated in our pretraining evaluation, the autoencoding reconstruction is not lossless. For SWE-bench tasks, subtle details in observations—such as exact file paths, variable names, or specific error messages—are often critical for making correct decisions. The ICAE encoder, even after fine-tuning, likely discards or corrupts some of this vital information. While the compression may be sufficient for preserving the general semantics of the text, it fails to retain the high-fidelity details required for complex, multi-step coding tasks, ultimately leading to agent failure.

Efficiency Gains. Beyond trajectory length, we also examined the computational efficiency of the ICAE compression approach. The compressed agent demonstrated measurable improvements in generation speed, achieving approximately 10% faster mean tool-call generation time compared to the uncompressed baseline (1.12s versus 1.23s per tool call). This speedup is composed of two phases: the compression of the observation (averaging 0.31s) and the subsequent generation of the next tool call (averaging 0.81s). The reduction in generation time is a direct consequence of the smaller context size that the decoder must process after compression.

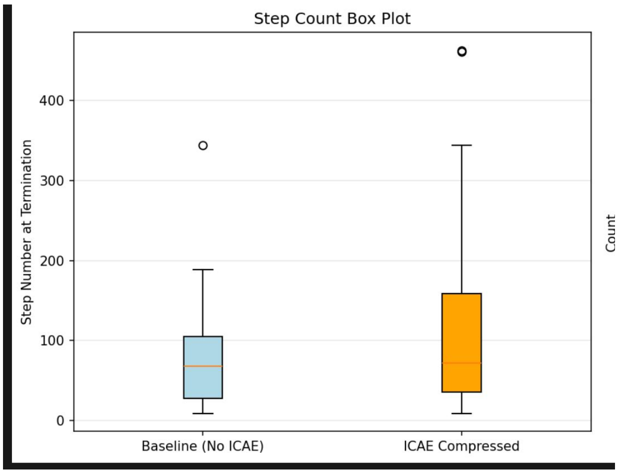
It is important to note that these timing measurements were obtained without the use of KV-caching, which would typically accelerate generation in multi-turn scenarios. The absence of KV-caching in our experimental setup means that the reported speedups are conservative estimates of the potential efficiency gains. A more detailed discussion of this limitation and its implications for real-world deployment is provided in Section **??**.

6.6.4 Impact on Agentic Trajectory Length

To address our first research question—whether implicit context condensation allows for the completion of longer agentic trajectories—we analyzed the number of interaction steps each agent could perform before reaching the context window limit of 32,000 tokens. Our findings provide a clear affirmative answer. The `ICAE-PT+FT` model, which employs compression, 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



(a) Box plot comparison of the baseline against the ICAE-PT+FT model



(b) Comparison of the number of steps at termination for the baseline model and the ICAE-compressed model

Figure 6.9 Box plot comparisons of baseline and ICAE-compressed models.

lengthy observations, 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 6.9b. The plot compares the distribution of step counts at termination for the baseline agent and the ICAE-compressed agent. The median 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.

This result is a direct consequence of the ICAE mechanism. The baseline model must store all the observations in its context, which quickly consumes the available token budget. In contrast, the ICAE model compresses these observations into a fixed number of memory tokens, substantially reducing their footprint. This efficiency gain allows the agent to accumulate a much larger history of interactions before exhausting its context, thereby theoretically allowing it to successfully complete the task.

6.6.5 Performance on Agentic Versus Standard NLP Tasks

Our second research question investigates how the performance of implicit context condensation on standard NLP benchmarks translates to the distinct demands of agentic software engineering tasks. The comparison of BLEU scores across all evaluated datasets, presented in Figure 6.10, reveals a clear performance disparity between these two domains. While the ICAE-based models achieve high BLEU scores on question-answering and reconstruction tasks (SQuAD and RepoQA), indicating effective preservation of semantic and syntactic information for these benchmarks, there is a noticeable degradation in performance on the SWE-bench agentic task.

This disparity suggests that while our compression method is effective for tasks requiring general comprehension and information retrieval from text and code, it struggles with the high-fidelity requirements of multi-step software engineering workflows.

The underlying reasons for this performance gap are multifaceted and point to several methodological limitations of the current approach. These factors, including the constraints of fixed-length compression at high, lossy ratios and the trade-offs associated with parameter-efficient fine-tuning, are discussed in detail in Chapter 7. This analysis underscores that transferring context condensation techniques to complex agentic domains is not straightforward and requires further research to bridge the performance gap.

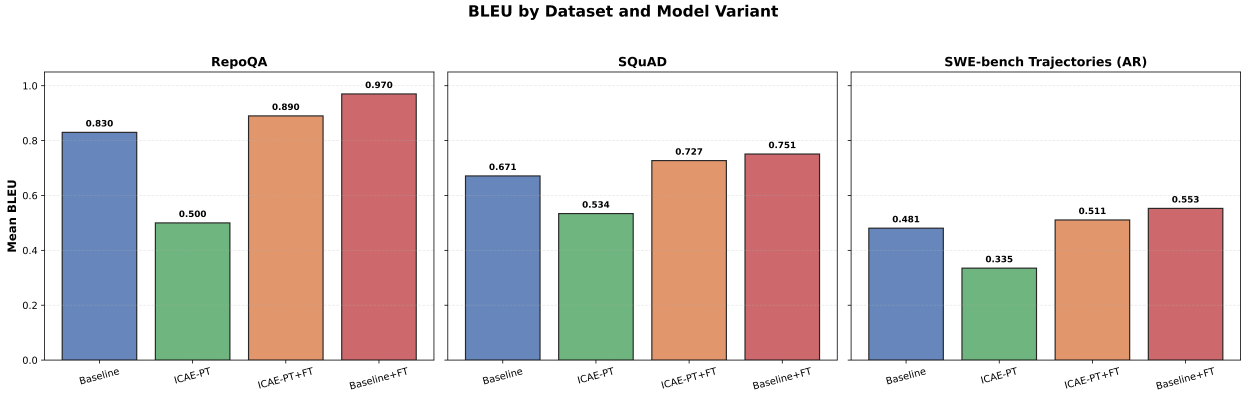


Figure 6.10 BLEU scores comparison across all datasets

Ablation: 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 6.5 compares the performance of key model configurations under both the official "Clean" prompting protocol and our "Cumulative" prompting experiment.

Table 6.5 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 (Qwen)	Cumulative	0.9000
Baseline (Qwen)	Clean	0.8967
ICAE-FT + Qwen	Cumulative	0.9089
ICAE-FT + Qwen	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-FT compressed model (0.9089 vs. 0.9020). Note, that we, of cource, 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.

Anyways, 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.

7 Limitations and Future Work

note: here I write "what can we not cover and how would you try to improve it in the future. e.g. only 256 tokens etc"

7.1 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).

7.2 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.

7.3 Constraints on Computational Resources

7.3.1 Model Scale

Due to computational and time constraints, our experiments were confined to the Qwen3-8B model. While evaluating on 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.

7.3.2 Full Fine-Tuning

Our experiments with ICAE exclusively utilized LoRA for parameter-efficient fine-tuning, consistent with the original work. However, our baseline experiments revealed that a fully fine-tuned Qwen3-8B model significantly outperforms its LoRA-tuned counterpart, establishing a high-performance upper bound (resolving 86 vs. 10 issues on SWE-bench). Although the original ICAE authors did not explore full fine-tuning, this result suggests that applying full fine-tuning to the ICAE encoder could yield substantial benefits in a complex agentic setting. Our open-sourced codebase is designed to facilitate such experiments, and we encourage future work to explore this promising direction, time and resources permitting.

7.3.3 Reasoning Enabled

For methodological simplicity, we disabled the Qwen3 model's "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.

7.4 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. Our comprehensive release includes all training configurations, hyperparameters, and experiment logs, enabling future researchers to reproduce our results and build upon this work.

All code, model checkpoints, and experiment logs are available at the project repository[1], with full Weights & Biases experiment tracking for both pretraining and fine-tuning phases. 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`

8 Conclusion

note: here I write "very high level overview of the whole thesis"
TODO: I'm not sure where to write this part so let it be here for a while:

8.1 Why our approach did not work?

Several factors may contribute to this performance degradation. One hypothesis is a representation-behavior mismatch, where the compression process perturbs the latent representations in a way that hinders the decoder's ability to perform tasks like tool use. Other contributing factors include a drop in reconstruction quality for specialized content, such as code, and potential overfitting to training labels, as suggested by high local accuracy but low overall task resolution rates.

8.2 Summary of Achievements

Describe main results of main experiment (ICAE on SWE-bench Verified dataset).

8.3 Synthesis of Findings

Describe the conclusions of all the findings (additional experiments)

8.4 Positioning the Work

I plan to make a big analysis of this work and it's position in the field and add it here + probably a picture.

A Appendix

A.1 On the Use of AI

I **have** used generative AI to help me write the text for this work.
I **have** used generative AI to help me write the code for the experiments.
I **have not** used AI in order to create new experiments, nor for any goals, research questions, hypotheses, etc.

A.2 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.3 SWE-smith First User Message

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

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|>
```

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. *<system_prompt> is from Section ??.

A.5 Training Details and Hyperparameters

A.5.1 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.1 Pretraining hyperparameters and configuration

A.5.2 Fine-tuning Configuration

A.5.3 SWE-bench Inference Configuration

A.5.4 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.

A.5.5 Reproducibility Resources

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][5].

[1] `https://github.com/JetBrains-Research/icae`
[2] `https://wandb.ai/kirili4ik/icae-pretraining`
[3] `https://wandb.ai/kirili4ik/icae-swebench-finetune`
[4] **TODO**
[5] **TODO**

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.2 Fine-tuning hyperparameters and configuration

Parameter	Value
Dataset	SWE-bench Verified
Temperature	0.0
Per-instance Call Limit	75
Max Input Tokens	32768

Table A.3 SWE-bench inference configuration

TODO: 4 is Pretrained model checkpoints achieving 95% reconstruction BLEU:
TODO: 5 is Fine-tuned models that outperform uncompressed baselines on SQuAD:

List of Figures

List of Tables

Bibliography

[Ano25] Anonymous. *AttentionRAG: Attention-Guided Context Pruning for RAG.* 2025. arXiv: 2503.10720 [cs.CL].

[Ano24] Anonymous. *Block-Attention for Efficient RAG.* 2024. arXiv: 2409.15355 [cs.CL].

[Aus+21] J. Austin et al. *Program Synthesis with Large Language Models.* 2021. arXiv: 2108.07732 [cs.LG].

[BPC20] I. Beltagy, M. E. Peters, and A. Cohan. *Longformer: The Long-Document Transformer.* 2020. arXiv: 2004.05150 [cs.CL].

[Bor+22] S. Borgeaud et al. *Improving language models by retrieving from trillions of tokens.* ICML 2022. 2022. arXiv: 2112.04426 [cs.CL].

[Cha+24] M. Chatzianastasis et al. *Soft Tokens, Hard Truths: Continuous CoT for Robust Reasoning.* 2024. arXiv: 2409.19170 [cs.LG].

[Che+21] M. Chen et al. *Evaluating Large Language Models Trained on Code.* 2021. arXiv: 2107.03374 [cs.LG].

[Chi+19] R. Child et al. *Generating Long Sequences with Sparse Transformers.* 2019. arXiv: 1904.10509 [cs.LG].

[Cho+21] K. Choromanski et al. *Rethinking Attention with Performers.* 2021. arXiv: 2009.14794 [cs.LG].

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

[DG24] T. Dao and A. Gu. *Transformers are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality.* ICML 2024. 2024. arXiv: 2405.21060 [cs.LG].

[Ge+24] T. Ge et al. *In-context Autoencoder for Context Compression in a Large Language Model.* arXiv:2307.06945 [cs]. May 2024.

[Gee25] GeeksforGeeks. *How to Use Cursor AI with Examples.* https://www.geeksforgeeks.org/blogs/how-to-use-cursor-ai-with-examples/. 2025.

[GGR21] A. Gu, K. Goel, and C. Ré. *Efficiently Modeling Long Sequences with Structured State Spaces.* ICLR 2022. 2021. arXiv: 2111.00396 [cs.LG].

[Hu+21] E. J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models.* 2021. arXiv: 2106.09685 [cs.LG].

[Jim+24] C. E. Jimenez et al. "SWE-bench: Can Language Models Resolve Real-world GitHub Issues?" In: *The Twelfth International Conference on Learning Representations.* 2024.

[Kat+20] A. Katharopoulos et al. *Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention.* 2020. arXiv: 2006.16236 [cs.LG].

[Las+24] M. Laskin et al. *Leave No Context Behind: Efficient Infinite Context Transformers with Infini-attention.* 2024. arXiv: 2404.07143 [cs.CL].

[Lea25] Learn Cursor. *Agent AI Assistant | Learn Cursor.* https://learn-cursor.com/en/wiki/user-guide/agent. 2025.

[LARC21] B. Lester, R. Al-Rfou, and N. Constant. *The Power of Scale for Parameter-Efficient Prompt Tuning.* 2021. arXiv: 2104.08691 [cs.CL].

[Lew+20] P. Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP.* 2020. arXiv: 2005.11401 [cs.CL].

[LL21] X. L. Li and P. Liang. *Prefix-Tuning: Optimizing Continuous Prompts for Generation.* 2021. arXiv: 2101.00190 [cs.CL].

[Liu+24] J. Liu et al. *RepoQA: Evaluating Long Context Code Understanding*. 2024. arXiv: 2406.06025 [cs.SE].

[Liu+23] N. F. Liu et al. "Lost in the Middle: How Language Models Use Long Contexts". In: *Transactions of the Association for Computational Linguistics* (2023). arXiv: 2307.03172 [cs.CL].

[Nak+21] R. Nakano et al. *WebGPT: Browser-assisted question-answering with human feedback.* 2021. arXiv: 2112.09332 [cs.CL].

[Pan+24] Z. Pan et al. "LLMLingua-2: Data Distillation for Efficient and Faithful Task-Agnostic Prompt Compression". In: *Annual Meeting of the Association for Computational Linguistics.* 2024.

[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.* 2002, pp. 311–318.

[Pat+25] 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*. 2025.

[Raj+16] P. Rajpurkar et al. *SQuAD: 100,000+ Questions for Machine Comprehension of Text.* 2016. arXiv: 1606.05250 [cs.CL].

[Sch+23] T. Schick et al. *Toolformer: Language Models Can Teach Themselves to Use Tools.* 2023. arXiv: 2302.04761 [cs.CL].

[SUV18] P. Shaw, J. Uszkoreit, and A. Vaswani. "Self-Attention with Relative Position Representations". In: *NAACL-HLT.* 2018.

[Son+24] Y. Song et al. *AgentBank: Towards Generalized LLM Agents via Fine-Tuning on 50000+ Interaction Trajectories.* 2024. arXiv: 2410.07706 [cs.CL].

[Su+21] J. Su et al. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 2021. arXiv: 2104.09864 [cs.CL].

[Tog23] Together. *RedPajama-Data-v2: An open dataset with 30 trillion tokens for training large language models.*
urlhttps://www.together.ai/blog/redpajama-data-v2. Accessed: 2025-11-14. 2023.

[Vas+17] A. Vaswani et al. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems.* 2017.

[Wan+20] S. Wang et al. *Linformer: Self-Attention with Linear Complexity.* 2020. arXiv: 2006.04768 [cs.LG].

[Yan+25a] A. Yang et al. *Qwen3 Technical Report.* 2025. arXiv: 2505.09388 [cs.CL].

[Yan+24] J. Yang et al. "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering". In: *The Thirty-eighth Annual Conference on Neural Information Processing Systems.* 2024.

[Yan+25b] J. Yang et al. *SWE-smith: Scaling Data for Software Engineering Agents.* 2025. arXiv: 2504.21798 [cs.SE].

[Yao+22] S. Yao et al. *ReAct: Synergizing Reasoning and Acting in Language Models.* 2022. arXiv: 2210.03629 [cs.CL].

[Yin+24] P. cheng Yin et al. *SlimCode: A Model-Agnostic Simplifier for Code.* 2024. arXiv: 2405.11196 [cs.LG].

[Zah+20] M. Zaheer et al. *Big Bird: Transformers for Longer Sequences.* 2020. arXiv: 2007.14062 [cs.LG].

[Zen+23] A. Zeng et al. *AgentTuning: Enabling Generalized Agents with Training on Agent-Specific Trajectories.* 2023. arXiv: 2310.12823 [cs.CL].

[Zha+20] J. Zhang et al. *PEGASUS: Pre-training with Extracted Gaps Sentences for Abstractive Summarization*. 2020. arXiv: 1912.08777 [cs.CL].

[Zha+25] R. Zhao et al. *Position IDs Matter: An Enhanced Position Layout for Efficient Context Compression in Large Language Models*. arXiv:2409.14364 [cs]. Sept. 2025.