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

1.1 The Context Length Challenge in Large Language Models (LLMs)

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. Much previous research has attempted to tackle this long context issue through architectural innovations, but these efforts often struggle to overcome a notable decline in performance on long contexts despite reducing computation and memory complexity. While alternative attention mechanisms can make scaling closer to linear, they typically cannot achieve true linear scaling without quality drops on longer sequences.

The long context limitation presents a significant practical challenge, particularly 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. 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.

Context compression offers a novel approach to addressing this issue, motivated by the observation that a text can be represented in different lengths in an LLM while conveying the same information. For example, the same information might be represented in various formats and densities without necessarily affecting the accuracy of the model's subsequent response.

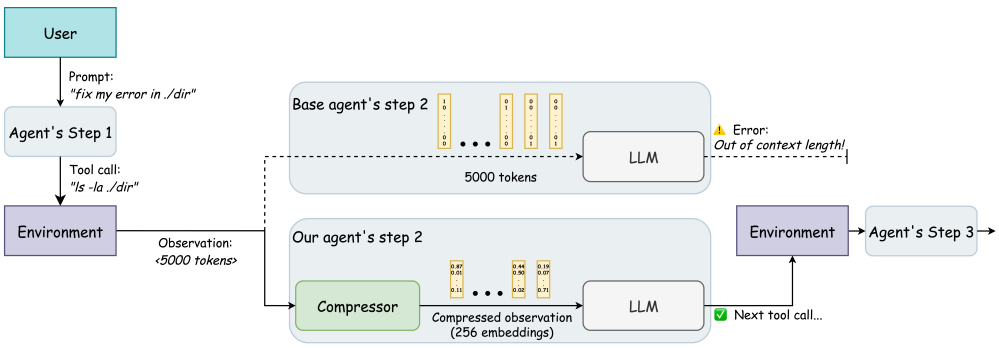

Figure 1.1 Comparison between base agent and our agent approaches for handling large observations exceeding LLM context length. The base agent fails when processing observations that are too long directly, while our agent successfully compresses the observation to 256 embeddings before LLM processing, enabling continued task execution.

The core goal of context condensation is precisely to leverage this potential density to enable LLM agents to execute tasks involving long chains of reasoning (or 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.

1.2 Implicit and Explicit Compression

The core concept of compression, motivated by the observation that information can be represented in different forms and densities, leads directly to the discussion and comparison between implicit and explicit approaches. Implicit compression moves beyond explicit, token-based techniques by utilizing the inherent density of continuous latent spaces.

Instead of relying on discrete representations, implicit compression focuses on mapping information into continuous representations (embeddings). This is possible because continuous latent spaces are inherently much denser than discrete spaces. Our goal is to enable more efficient processing of information by successfully leveraging these dense representations.

2 Background

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, the similarity $\langle Q_i, K_j \rangle$ implicitly depends on $i - j$ via rotations applied to Q_i and K_j.

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]. **Maybe say 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 atop 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. 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 perception–action loop continues until termination. Concretely:

1. Prompt assembly: system instructions + task + concise guidelines + recent trajectory summary.
2. Policy step: the LLM proposes an action (often with brief rationale) and a tool to invoke.
3. Tool execution: the specified tool runs non-interactively with provided arguments.
4. Observation: tool output (stdout/stderr, structured JSON, file diffs, or retrieval snippets) is captured.
5. Memory update: (a_t, o_t) is logged; optional compression/summarization reduces footprint.
6. Termination or next step: the agent either returns a final answer or continues planning.

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.

3 Related Work

3.1 Architectural Approaches for Long Context Modeling

Long-context modeling must address the quadratic cost of vanilla self-attention [Vas+17] while preserving task-relevant dependencies over thousands of tokens.

A useful taxonomy distinguishes:

- attention variants that alter the attention operator itself;
- explicit compression methods that reduce the input via retrieval or summarization;
- implicit compression methods that learn compact, decoder-friendly representations without exposing full inputs during inference.

We briefly review each group and summarize advantages and limitations.

Attention variants. Sparse and local/windowed patterns reduce pairwise interactions to achieve subquadratic cost. Windowed attention restricts each token to a fixed neighborhood and optionally augments a small set of global tokens to propagate long-range information. Longformer combines sliding windows with learnable global tokens for long documents [BPC20]. Block- and mixed-sparsity patterns (e.g., banded + random) as in BigBird provide theoretical expressivity and empirical gains on long sequences [Zah+20]. Sparse Transformers [Chi+19] introduced fixed sparse patterns for scalable generation. Advantages include improved memory/computation and strong local modeling. Disadvantages include potential failures to route cross-window interactions when global tokens or connectivity patterns are insufficient, and hardware inefficiencies for irregular sparsity.

Linear-time approximations further change the attention operator. Kernelized attention (Transformers-as-RNNs) linearizes softmax attention for autoregressive decoding [Kat+20]. Linformer projects keys/values along sequence length to attain linear complexity [Wan+20]. These methods offer asymptotic gains and longer feasible contexts. However, they can underperform full attention on tasks requiring precise long-range interactions or exact softmax geometry. They also introduce approximation/projection hyperparameters that affect quality.

Explicit compression. Retrieval-augmented generation (RAG) or summarization-based pipelines reduce the effective input by selecting or rewriting content before decoding. RAG retrieves top-k passages from an external corpus and conditions generation on them. This approach improves knowledge-intensive tasks while decoupling parametric and non-parametric knowledge [Lew+20]. Abstractive summarization pre-compresses long inputs into concise proxies (e.g., PEGASUS pretraining with gap-sentence objectives) [Zha+20]. Benefits include controllable compute and access to external knowledge. Drawbacks include selection bias, retrieval latency, brittleness to retrieval errors, and potential loss of details critical for downstream reasoning.

Implicit compression. Here, the idea is to produce task-adapted representations (often embeddings) that a model uses during inference, rather than the input itself. Examples include learned soft/prefix prompts for steering frozen decoders [LARC21; LL21]. Other examples include tokenized memories trained to preserve answer-relevant information. Implicit methods maintain a tight interface to the model. They can reduce latency and memory without external retrieval. A key challenge of implicit compression is that, by condensing input into a compact intermediate representation, some information may inevitably be lost

and cannot be recovered with perfect fidelity. This may limit the utility of compressed representations, especially when critical details are omitted during the compression process.

Taken together, attention variants trade exactness for structure or approximation. Explicit compression trades completeness for selection. **TODO: this is bullshit but make a change to section 3./ we like implicit so we found ICAE—** — Implicit compression trades human readability for decoder-optimized compact interfaces.

3.2 Soft Prompting, Context Distillation, and Continuous-Thought Representations

Soft prompting is a method for conditioning large language models by learning continuous prompt vectors or prefix tokens rather than discrete text. These learned vectors are typically prepended to the model's input sequence and serve as a compact, trainable interface for adapting frozen decoders. Classic methods in this category include prefix-tuning [LL21] and prompt tuning [LARC21]. Both approaches involve optimizing a small set of continuous embeddings that steer the model toward desired behavior without full-scale model finetuning. This parameter-efficient interface enables flexible adaptation and can be tuned for domain- or task-specific goals. Because the prompt vectors are not constrained to map onto human-readable tokens, they can condense much more information than would be possible with standard textual prompts.

In practical, agentic settings the challenge of long or growing histories becomes acute (???).

CoConut (Chain of Continuous Thought) [**coconut_placeholder**; **arxiv_2412_06769**] generalizes the concept of soft prompting by moving away from natural language tokens altogether and enabling reasoning directly in latent, continuous spaces. Instead of relying on explicit tokenization and sequence rewriting, CoConut leverages the model's own hidden states as a "continuous thought" vector. After processing an input, the final hidden state (or a structured set of latent embeddings) is fed back as a contextual scaffold for further reasoning steps. Experiments show that directly reasoning in the model's own latent space improves downstream performance for tasks with extended, multi-step dependencies, outperforming classic chain-of-thought prompting. By discarding the constraints of discrete tokenization, CoConut demonstrates that agentic LLMs can reason, plan, and retain context in a fundamentally more expressive and compact way.

These three directions form a coherent spectrum of implicit compression strategies. All focus on designing interfaces for decoder-optimized, compact context that can scale to long, complex tasks without being forced back into the limits of surface-level, human-readable summaries.

3.3 The In-Context Autoencoder (ICAE) Framework

ICAE [Ge+24] is closely related to the above implicit compression paradigm. An encoder (often a LoRA-adapted copy of the base LLM) reads a long context and emits a small set of learnable *memory tokens*. A frozen decoder (the base LLM) then conditions on these tokens plus the downstream prompt to generate outputs. Training combines the following objectives:

- an autoencoding objective, prompting the decoder to reconstruct the original text from memory slots
- a language modeling/continuation objective that teaches the decoder to answer queries conditioned on the slots and a prompt

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 (e.g., 4× or higher), and the position-ID placement of these tokens influences how readily the decoder can access stored information (cf. Section 2.1) [Ge+24].

Beyond the high-level description, Figure 3.1 depicts the encoder–decoder split. On the left, the encoder ingests the full context (e.g., a text) and produces a fixed number of memory tokens. On the right, the frozen decoder receives these tokens and a tokens During pretraining, the encoder is optimized so that the decoder can reconstruct the original text and continue language modeling (50/50 chance of being used). During fine-tuning, the objective emphasizes answering prompts correctly given only the memory tokens and the task prompt. In practice, the encoder is frequently adapted with parameter-efficient methods such as LoRA [Hu+21], whereas the decoder remains frozen to preserve the capabilities of the base model.

ICAE differs from heuristic summarization in that the representation is optimized for decoder consumption rather than human readability. It also differs from sparse or windowed attention in that the decoder still operates with dense attention over a small set of memory tokens. ICAE differs from explicit RAG in that no external retrieval is required at inference [BPC20; Lew+20; Zah+20]. Compared to purely recurrent memory, ICAE offers direct, content-dependent access via attention over a small set of tokens. This avoids long chains through recurrent states.

In the context of local software engineering agents, ICAE-style compression is particularly attractive: developer tooling yields verbose logs, test outputs, diffs, and compiler errors. Compact, learned memories can preserve salient facts (e.g., failing tests, stack traces, file paths, and prior fixes) while improving latency and GPU memory use. In later chapters we evaluate compression ratios around ~ 1.5–$2\times$ and analyze the accuracy–latency trade-offs compared to uncompressed baselines, as well as ablations that drop long or all observations to quantify the contribution of learned memory tokens.

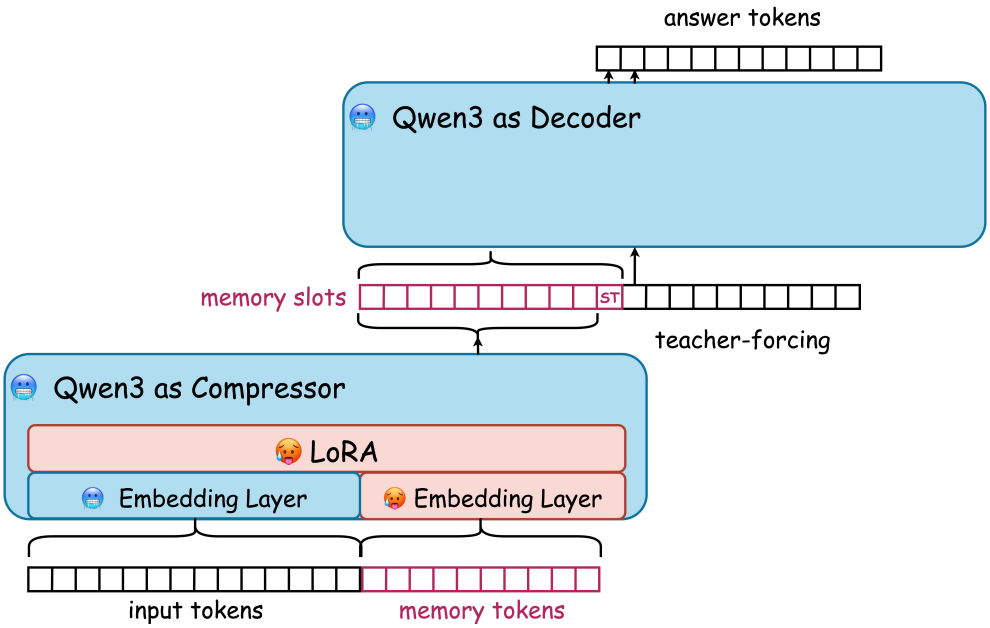

Figure 3.1 In-Context Autoencoder (ICAE) framework architecture

3.4 Some attempts to do the same thing?

Tobias' blogpost from openhands is not implicit, but explicit. so i have no knowledge of the same solutions using implicit compression?

4 Methods

4.1 Data Acquisition and Setup

4.1.1 SWE-bench

We have chosen to use the SWE-bench [Jim+24] dataset for our experiments. It is a well known dataset for evaluating the performance of SWE agents. It contains a large number of SWE tasks, each with a set of instructions and a set of expected outputs. They were collected from real life GitHub issues. We have chosen to work with a subset – SWE-bench Verified [**swebench-verified**]. It is a subset of the SWE-bench dataset that contains only the tasks that have been verified to be correct.

4.1.2 Tools and Interaction Protocol (setup?)

We follow the SWE-smith setup for SWE-bench Verified [Jim+24], where the assistant agent interacts with the environment through a minimal toolset and a fixed protocol. Concretely, the available tools are a shell interface (`bash`), a submission tool (`submit`), and a custom editing utility (`str_replace_editor`). These tools generate the observations that accumulate within the trajectory context. The exact SWE-smith prompt (and the tools description) is provided in Appendix A.2. It should be noted that this method does not use function calling functionality of the model. While some models support a special format for function calling (e.g. additional fields for the tools descriptions etc.), in this case we just use the tools as described in the prompt.

The `str_replace_editor`. This is a stateful file editor that supports viewing, creating, and editing files with precise, line-exact operations. Its state persists across steps, enabling consistent multi-edit workflows. The interface exposes the following commands: `view`, `create`, `str_replace`, `insert`, and `undo_edit`.

For deterministic edits, `str_replace` requires `old_str` to match exactly one or more consecutive lines in the target file, including whitespace. The `new_str` content replaces the matched block. The `insert` command appends `new_str` after a specified line number.

By this, there is a lot of tools combined into one, but it works well as described in [**swe-smith**], so we adopt it. And also to be comparable to other open sourced scores.

4.1.3 Agentic Trajectories as Data

We treat sequential action–observation interactions (trajectories) as training data. These trajectories were obtained using a strong teacher model (e.g., Claude Sonnet 3.7) on the SWE-bench Verified dataset to produce high-quality inputs suitable for futher training. The setup for generating these trajectories follows the SWE-smith setup closely[**swe-smith**].

4.2 ICAE Model in Agentic Setup

4.2.1 ICAE Components

As described previously, the ICAE [**ge_context_2024**] consists of two modules: a lightweight encoder (implemented via a LoRA-adapted LLM) and a fixed decoder (the LLM itself). The encoder processes the long trajectory context and generates a fixed number of learnable memory token.

We use the encoder for every environment observation (**TODO: picture here?**) So, more in detail, 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. You can see the example of this in figure 5.1 and 5.2 from Chapter **??**.

4.2.2 Base Model Configuration (Qwen3)

We use the Qwen3 model family (specificaly Qwen3-8B) as the base LLM. 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. We found that deleting or leaving this prefix in the history does have an inretesting affect the quality of the model, we describe this in more detail in Section **??**.

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.

4.3 Training Procedures and Evaluation

4.3.1 Pretraining (PT)

Our pretraining procedure follows the original ICAE formulation [Ge+24]. We use two primary self-supervised objectives:

(i) **Autoencoding (AE)**, where ICAE restores the original input text from its memory slots (prompted by a special token `[AE]`).

(ii) **Language Modeling (LM) / Text Continuation**, which predicts the continuation of the context (prompted without any special tokens) to improve generalization and prevent overfitting to the AE task.

This pretraining was performed on the dataset SlimPajama-6B [**pajama6b**][1]. It is a common text dataset, that is used for pretraining LLMs, consisting of 6 billion tokens (the first 10% of the first 10% of the original 627B tokens). The dataset that authors used in their original paper "The Pile" was unavailable. It should be noted that only LoRA weights of the encoder are trained here (specifically only form Q and K matrices of the attention layers). So, in theory, we are training the encoder to encode the context into such embeddings, that the decoder will be able to reconstruct the original context from them or to continue the text from the context.

4.3.2 Fine-Tuning (FT)

After pretraining, ICAE is fine-tuned using trajectories from SWE-bench Verified (the proccess of obtaining trajectories is described in Section 4.1.3). Here as well only the LoRA weights are trained. The fine-tuning objective maximizes the probability of generating the correct agent action (i.e. tool call) conditioned on the memory slots (if the encoder is pplied at the current step) and the previous history???

It should be noted that the training is only applied to the encoder part, while the decoder is frozen.

During training, the backpropagation process constrains us to optimize over single-step transitions: So at timestep k, the encoder compresses the observation o_{k-1} into embeddings. Then, the decoder generates

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

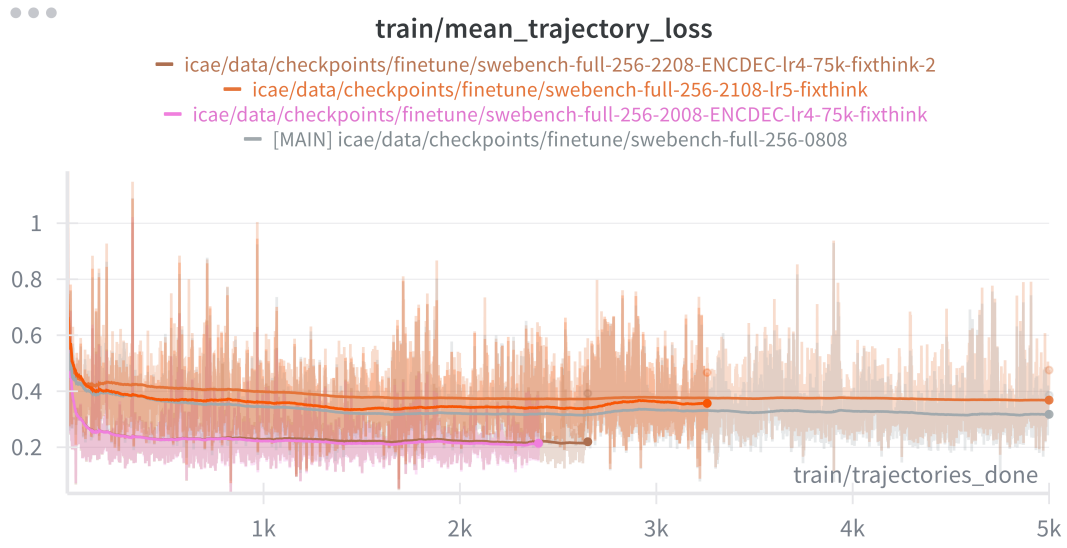

Figure 4.1 Pretraining loss curves averaged by trajectory

the action a_k given the memory slots and the previous history. Then, token-by-token of a_k we backpropagate the loss through the decoder and the encoder, making an update of the LoRA weights of the encoder. Crucially, whenever an observation exceeds 256 tokens, the encoder compresses it into a fixed set of 256 memory tokens (continuous embeddings), and these memory tokens replace the original text in the accumulated history. 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**]. Consequently, the model never processes the full raw text of long observations during subsequent steps—it only conditions on the compact memory representations.

For example, consider a 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, ≥ 256 tokens): entire file content, compressed into 256 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**.

So, in this example, the encoder is applied 2 times (steps 7 and 9), while the decoder is applied 5 times (steps 3, 5, 7, 9, 11). At training time, each step's loss is computed independently: the model learns to predict a_k

from the prefix ending at o_{k-1}, where any long observation has already been replaced by its memory tokens. At inference time, the same replacement occurs dynamically, ensuring consistency between training and deployment. This design allows the model to handle arbitrarily long trajectories without exceeding context limits, as the effective history remains compact.

On figure 4.2 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.

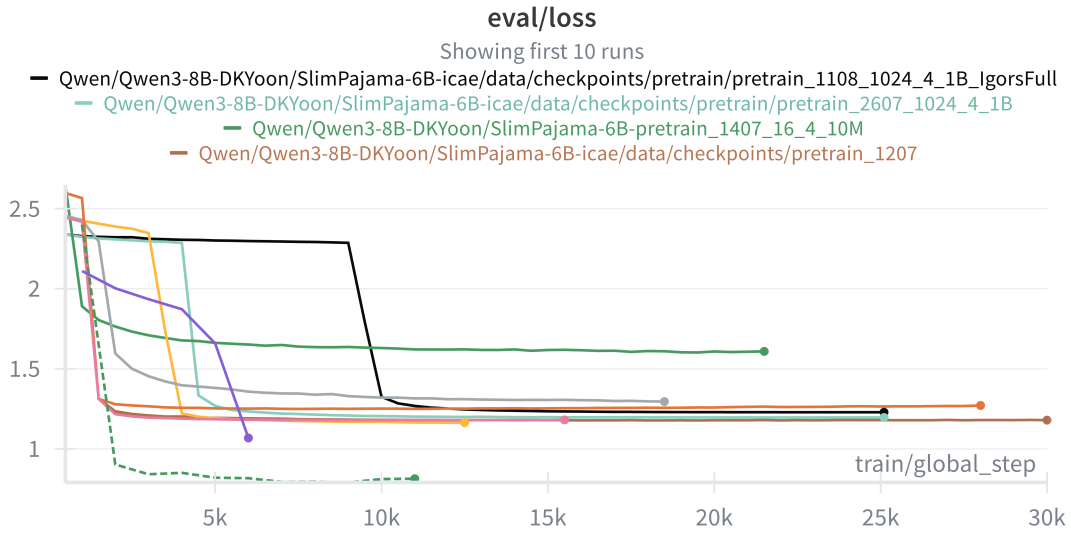

Figure 4.2 Fine-tuning loss curves

4.4 Key Metrics

We report several metrics to evaluate our approach. As a simple proxy metric, we measure token-wise accuracy—the fraction of generated tokens that match the reference trajectory. However, the most important metric is the number of successfully resolved issues on SWE-bench Verified, which directly reflects the model's ability to complete real-world software engineering tasks. We also measure mean tool-call generation time to assess computational performance.

We note that measuring trajectory length is not particularly meaningful in our setting, as 8B-scale models frequently enter loops where they repeatedly call the same tool, artificially inflating trajectory length without making meaningful progress toward task completion.

4.5 Code Reproduction and Testing Methodology

4.5.1 Code 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 more modular and provides separate training pipelines for both pretraining (PT) and fine-tuning (FT). We support pretraining on general text datasets (such as SlimPajama) and fine-tuning on both question-answering tasks (SQuAD) and agentic trajectories (SWE-bench). We provide a link to our code repository[2] for transparency and reproducibility.

[2] `https://github.com/JetBrains-Research/icae`

5 Experiments and Evaluation

5.1 Experimental Setup

We implement our ICAE framework from scratch, building upon the original architecture [Ge+24] with several modifications for improved efficiency and reproducibility. Our implementation uses Qwen3-8B as the base model, with LoRA adaptation applied to the attention matrices (q_proj and v_proj) using a rank of 128.

Pretraining is conducted on the SlimPajama-6B dataset using a combination of autoencoding and language modeling objectives, achieving 95% reconstruction BLEU score on general text. Fine-tuning on SWE-bench trajectories uses a larger memory size of 256 tokens and explicitly disables thinking mechanisms for simplicity, focusing on direct tool-call generation.

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.3.

5.2 Initial Prototype Experiments: The Necessity of Training

The initial approach tested replacing hard tokens with soft/averaged continuous embeddings without fine-tuning. These prototype experiments, using methods like KV-cache hacks or direct embedding inputs in vLLM, demonstrated that scores decreased by more than 50% on QA tasks (e.g., SQuAD context embed F1 dropped from 0.71 to 0.17 or 0.11). This negative result confirmed the hypothesis that training is necessary to effectively condense context into the latent space.

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 5.1 Baseline against averaging techniques (Prompt–Q–C)

5.3 Initial Prototype Experiments: First Attempts at 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 cannot provide. The baseline (Step 1) and the "soft/hard mix works" observation (Step 2) are illustrated in 5.3 and frame this question empirically.

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**,

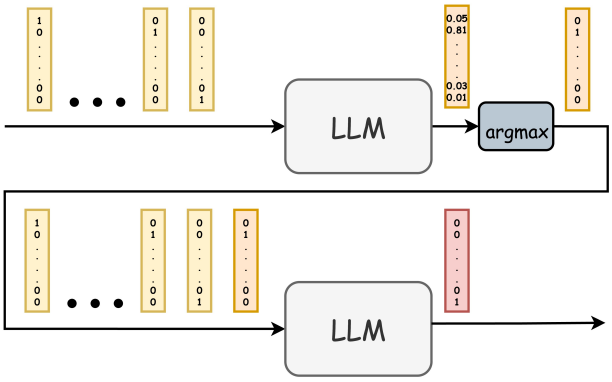

Figure 5.1 Visualization of the "without training" approach, p1

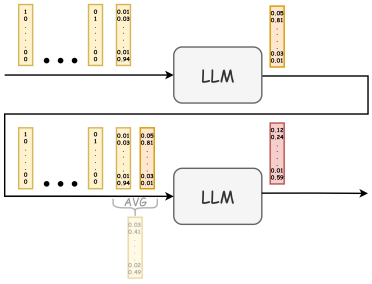

Figure 5.2 Visualization of the "without training" approach, p2

$g_\theta : \mathbb{R}^{2d} \to \mathbb{R}^d$, 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 5.4.

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

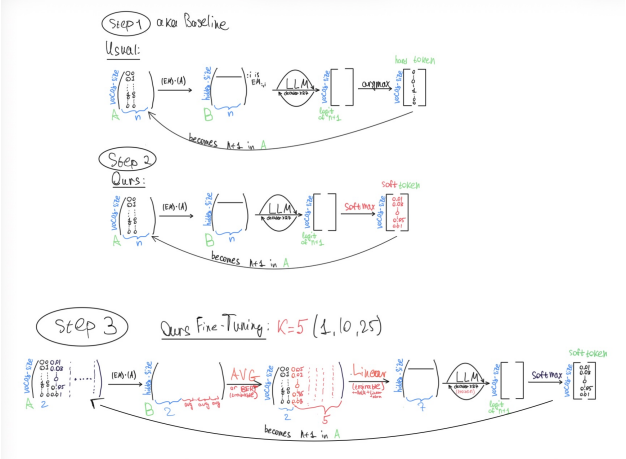

Figure 5.3 Visualization of the baseline approach, step 1-3

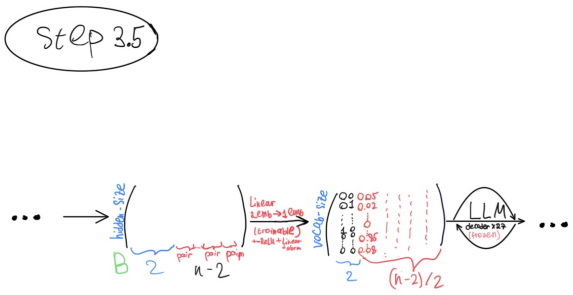

Figure 5.4 Visualization of the experimental setup, step 3-5

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 (5.5), 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).

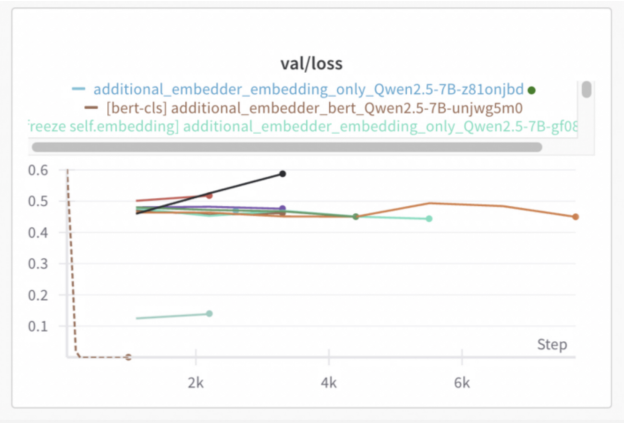

Figure 5.5 Validation loss curves for the SQuAD generalization experiment using linear, MLP and BERT

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 (5.3) 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 (≈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 5.4 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.

TODO: I STOPPED HERE

5.4 ICAE Pretraining and Evaluation on General Text Reconstruction

Pretrained ICAE [Ge+24] demonstrated the ability to decompress general texts almost perfectly. High BLEU scores were achieved on datasets like PWC (99.1 for Mistral-7B, 99.5 for Llama-2-7B) and SQuAD (98.1 for Qwen3-8B), indicating that memory slots retained almost all context information for contexts up to 400 tokens. Analysis of reconstruction errors showed patterns similar to human memorization mistakes (e.g., restoring "large pretrained language model" as "large pretrained model"), suggesting the model selectively emphasizes or neglects information based on its understanding.

5.5 ICAE Fine-Tuning and Evaluation on Question Answering Tasks

When fine-tuned on QA tasks (SQuAD), ICAE-FT [Ge+24] achieved high F1 (73) and Exact Match (69%) scores, performing well compared to LoRA-FT baselines. The quality of the compressed representation was shown to significantly outperform summaries generated by GPT-4 under the same length constraint (128 tokens).

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

Table 5.2 ICAE averaging on SQuAD

5.6 ICAE Fine-Tuning and Evaluation on SWE-bench Verified

5.7 Disabling Thinking Experiments

We evaluated the impact of disabling thinking by inserting the token sequence `\think \think` with a newline marker during decoding. We tested two variants: adding the sequence after every step and adding it only once at the end. Surprisingly, the approach commonly suggested by authors (frequent insertion) yielded worse output quality than the minimal, end-only variant in our setting. We note that related observations of atypical Qwen3 behavior under prompting heuristics have been reported anecdotally; a systematic investigation is out of scope for this work. (as a result e.g. coder-30bA3 has no thinking mode at all) Further details and ablations are left for future work.

Efficiency Results: ICAE [Ge+24] compression led to measurable efficiency improvements, achieving a theoretically 10% faster mean tool-call generation time than the vanilla baseline (e.g., 0.4880s vs 0.5437s).

Furthermore, latency tests showed speedups of 2.2× to 3.6× in total time for inference. Token-wise Accuracy vs. Resolved Rate: Although token-wise accuracy performed on par with (or slightly better than) the vanilla Qwen baseline (e.g., 0.9089 vs 0.9000), this metric was noted to be problematic ("token-wise accuracy is bullshit") and decoupled from true task success. End-to-End Task Success: The primary negative finding was that the model with compression resolved significantly fewer than 50% as many issues as the original Qwen model on the SWE-bench Verified dataset.

Encoder	Decoder	Accuracy	Mean tool-call time (s)
—	Full-FT	0.9484	1.24
—	LoRA-FT	0.9118	1.24
—	Qwen	0.8967	1.23
del long obs-s	Qwen	0.8873	0.44
del all obs-s	Qwen	0.8802	0.39
ICAE (LoRA-PT w/ Full-FT)	Full-FT	0.9219	—
ICAE (LoRA-PT w/ Qwen)	Qwen	0.8808	**1.12 (0.31+0.81)**
ICAE (LoRA-FT)	Full-FT	?	—
ICAE (LoRA-FT)	LoRA-FT	0.9263	—
ICAE (LoRA-FT)	Qwen	0.9020	—

Table 5.3 No think bug table. Qwen and ICAE future variants. FT=FineTuning, PT=PreTraining

Encoder	Decoder	Acc.	Time (s)	Resolved (/500)
—	Qwen-Full-FT	0.9484	1.24	—
—	Qwen-LoRA-FT	0.9118	1.24	10
—	Qwen	0.8967	1.23	26
ICAE (Qwen-LoRA-FT)	Qwen	0.9020	—	11
ICAE (Qwen-LoRA-FT)	Qwen-LoRA-FT	0.9263	—	3 (overfit?)
ICAE (Qwen-LoRA-FT)	Qwen-Full-FT	?	—	—
del long obs-s	Qwen	0.8873	0.44	1
del all obs-s	Qwen	0.8802	0.39	0
ICAE (Qwen-LoRA-PT w/ Q-Full-FT)	Qwen-Full-FT	0.9219	—	—
ICAE (Qwen-LoRA-PT w/ Qwen)	Qwen	0.8808	**1.12 (0.31+0.81)**	—

Table 5.4 No think bug table. Qwen and ICAE future variants. FT=FineTuning, PT=PreTraining

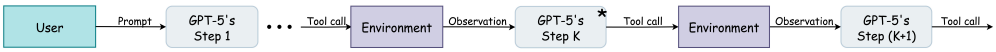

Figure 5.6 ICAE application to SWE-bench - Results 1

5.8 Discussion of Agentic Failure Hypotheses

Hypotheses for the end-to-end performance degradation include Representation–behavior mismatch, where the compression perturbs the decoder's behavior necessary for tool use. Other factors include reconstruction quality falloff for specialized content like code files, and potential overfitting to labels demonstrated by high local accuracy but low resolved rates.

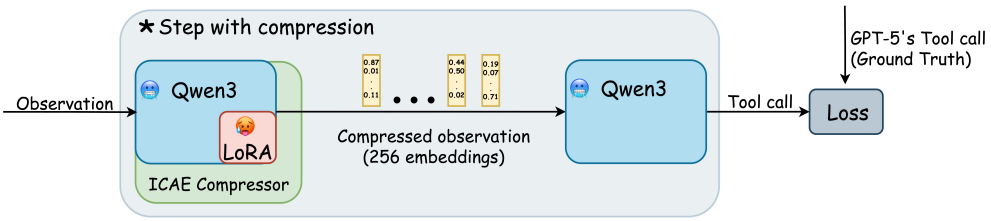

Figure 5.7 ICAE application to SWE-bench - Results 2

6 Limitations and Future Work

6.1 Reframing the Goal: Feature Extraction vs. Compression

When developing a context condensation strategy, the definition of success must be carefully framed. The approach investigated utilizes the In-context Autoencoder (ICAE) [Ge+24], which leverages the power of an LLM to compress a long context into short compact memory slots that can be directly conditioned upon by the decoder LLM.

However, the approach should be redefined not merely as "compression," which often implies a robust, high-ratio data reduction, but rather as "summarization" or "fixed length feature extraction," due to a core methodological constraint. This constraint arises from the hardcoded, non-robust nature of the intended output length (e.g., aiming for 256 tokens in general discussion).

This reframing is critical because lossless compression typically struggles to achieve ratios exceeding 10×. Under high compression ratios (lossy compression), the assumption that "all tokens in the context are equally important"—an assumption intrinsically aligned with lossless autoencoding training—is violated. When the information carrier capacity is limited, the compression mechanism should ideally focus only on the most important tokens, which conflicts with the uniform coverage implied by fixed-length encoding.

The fundamental mechanism supporting this goal is the relative density of different representation spaces. The latent space of embeddings is "much denser than the discrete space of tokens," which is the underlying justification for learning context condensation. Therefore, the thesis investigates how condensing environment observations (which contain irrelevant or redundant information) into continuous representations (embeddings/memory slots) affects agent performance and efficiency when addressing the context length challenge.

6.2 Limitations of Fixed-Length Compression

The methodology assumes that "all tokens in the context are equally important," aligning intrinsically with lossless compression (autoencoding), which becomes problematic under high compression ratios (lossy compression). The non-robustness of the approach is constrained by the hardcoded number of memory tokens (e.g., 256 tokens in discussion), defining the limitation of the approach as fixed-length feature extraction. Experimental results confirm that improvement attenuates or fails at high compression ratios (e.g., beyond 15x or 31x).

6.3 Constraints on Model Scale

Due to computational limitations, experiments were mainly conducted on Llama models up to 13 billion parameters.

6.4 Outlook for Future Research

Future work should explore validating the ICAE [Ge+24] effectiveness on larger and stronger LLMs, as performance is expected to benefit more from more powerful target models. Potential extension to multimodal LLMs (images, video, audio) is suggested, as these modalities have greater compression potential.

6.4.1 Open Source Contributions and Reproducibility

To advance the field of context compression for software engineering agents, we release our complete implementation, including pretrained models achieving 95% reconstruction BLEU and fine-tuned models that outperform uncompressed baselines on SQuAD. Our comprehensive release includes all training configurations, hyperparameters, and experiment logs, enabling future researchers to reproduce our results and build upon this work.

The open-source nature of this contribution addresses the reproducibility crisis in machine learning research, providing both the tools and transparency necessary for scientific progress in context management for LLM agents. All code, model checkpoints, and experiment logs are available at the project repository, with full Weights & Biases experiment tracking for both pretraining and fine-tuning phases.

6.5 Caching would not ever work with our approach? Or would it?

7 Conclusion and Outlook

7.1 Summary of Achievements

The ICAE [Ge+24] framework successfully achieves context condensation/feature extraction for general text, demonstrating high reconstruction quality (BLEU $\approx$ 99%) and measurable efficiency gains (speedup 2× to 3.6×). The work provided insights into LLM memorization patterns, suggesting similarities to human memory encoding.

7.2 Synthesis of Findings

Despite achieving efficiency and local accuracy on agent trajectories, the performance degradation in end-to-end task completion (resolved issues) highlights the critical gap between local context compression quality and robust decision-making in complex agentic settings.

7.3 Positioning the Work

The findings position this work within the broader research efforts on LLM context management, emphasizing the experimental results concerning condensation effectiveness across different data types, and suggesting caution when applying general compression methods to fine-grained, critical agent behaviors (code and tool use).

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

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 Training Details and Hyperparameters

A.3.1 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 Rank	128
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.3.2 Fine-tuning Configuration

Parameter	Value
Base Model	Qwen3-8B
Dataset	SWE-bench trajectories
Learning Rate	5×10^{-5}
Batch Size	1
Gradient Accumulation	1
Training Steps	≈150,000
Memory Size	256 tokens (4× compression)
LoRA Rank	128
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

A.3.3 Reproducibility Resources

To ensure full reproducibility, we publish our complete implementation including:

- Complete ICAE framework for both pretraining and fine-tuning phases (`https://github.com/JetBrains-Research/icae`)

- Full Weights & Biases experiment logs for pretraining: `https://wandb.ai/kirili4ik/icae-pretraining`
- Full Weights & Biases experiment logs for fine-tuning: `https://wandb.ai/kirili4ik/icae-swebench-fi`
- Pretrained model checkpoints achieving 95% reconstruction BLEU: TODO
- Fine-tuned models that outperform uncompressed baselines on SQuAD: TODO

A.4 Profiling Setup and Latency Measurement

Technical details of the test machine and runtime configuration used for latency measurements.

A.5 Detailed Evaluation Tables

Comprehensive tables of model performance, including token-wise accuracy, mean tool-call time, and resolved issues for various ICAE [Ge+24] variants (e.g., Qwen-LoRA-FT, ICAE (Qwen-LoRA-FT) Qwen).

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].

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

[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].

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

[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].

[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].

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

[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.

[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.

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

[Swe] *SWE-smith: Agent Setup and Prompt for SWE-bench Verified*. Preprint. Describes the SWE-smith tool protocol and prompt for SWE-bench Verified. Prompt text included in Appendix A.2. 2024.

[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].

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

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

[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.