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

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.

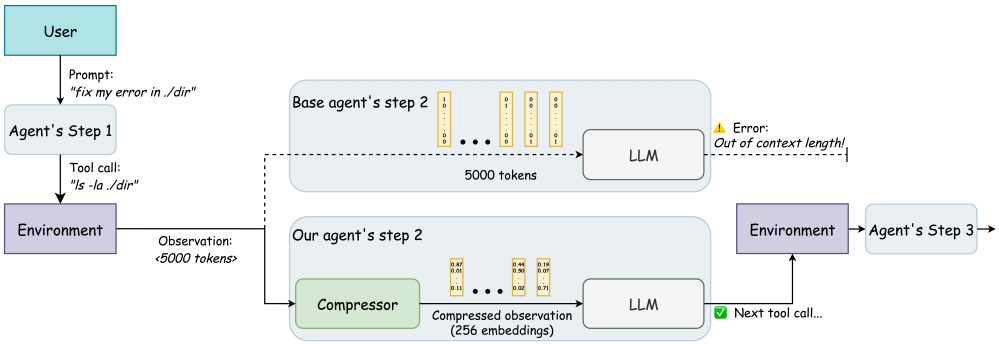

Figure 1.1 Comparison between base agent and our agent approaches for handling large observations exceeding LLM context length. The base agent fails when processing observations that are too long directly, while our agent successfully compresses the observation to 256 embeddings before LLM processing,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

note: here i write explanations of things. not too basic (not transformer), but at the same time not too specific for the thesis (not icae)

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

note: here i write "what was previously done". see 3.4 for that

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 4. 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.

3.3 The In-Context Autoencoder (ICAE) Framework

TODO: I am very lost on where and how to write about ICAE. I guess here should be none, but then I need smth to compare 3 ideas from 3.1? Then I also need ICAE in 4 and 5 somehow. Also 4.2 needs it but 4.3 is the same (expl. of AE vs LM for 50/50)

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.

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.

3.4 Some attempts to do the same thing?

TODO: make a proper literature review..?

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

Should write here about the other solutions for our problem/dataset? but what are those?

4 Methods

note: here I write "what am i doing on top of other methods" I write it conceptually. if i need results maybe reference to the next chapter.

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 **??**.

The ICAE framework 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 4.1 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.

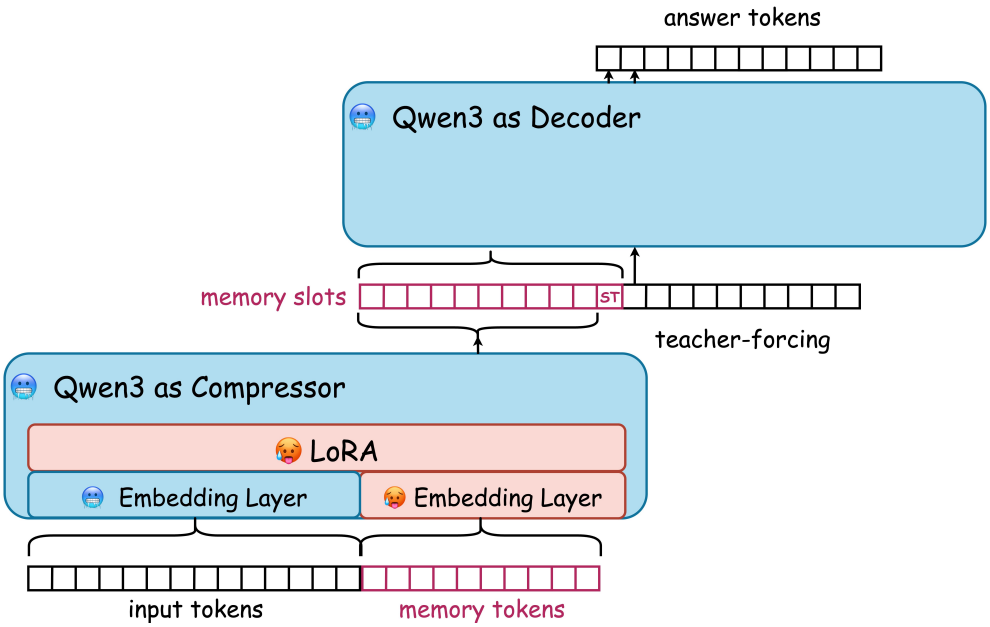

Figure 4.1 In-Context Autoencoder (ICAE) framework architecture

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)**, 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 (a random 1% 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.

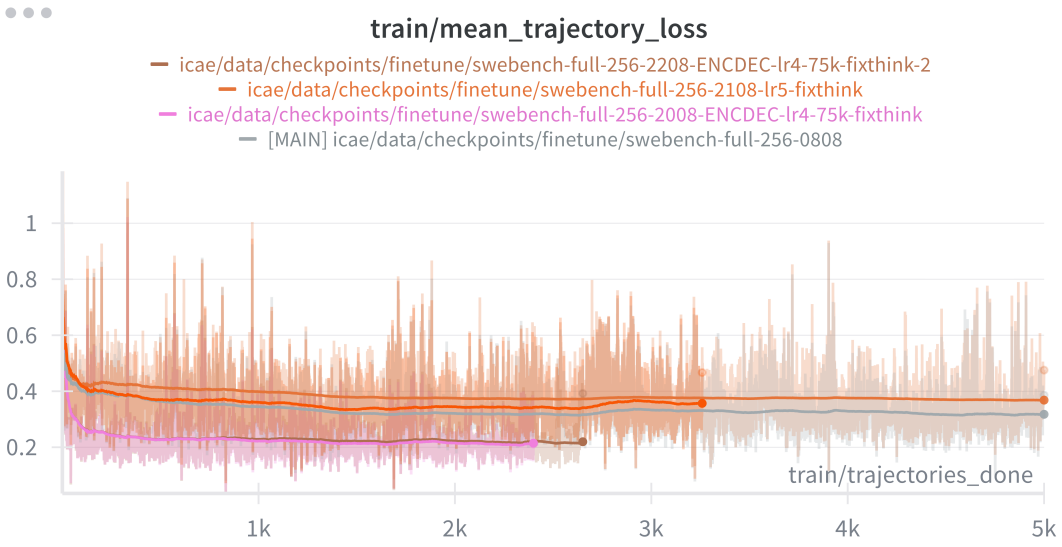

Figure 4.2 Pretraining loss curves averaged by trajectory

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

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 applied 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 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 4.3 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.

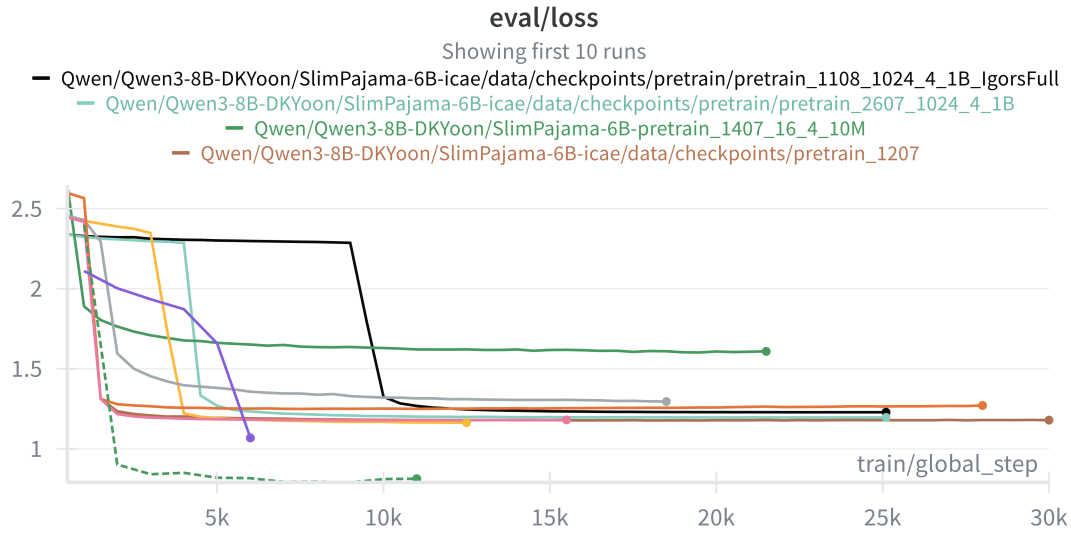

Figure 4.3 Fine-tuning loss curves

4.4 Code Reproduction and Testing Methodology

4.4.1 Key Metrics

We report several metrics to evaluate our approach. As a simple proxy metric, we measure token-wise accuracy —- averaged fraction of the guessed tokens that match the reference trajectory (note: this is measured with teacher forcing). 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.

4.4.2 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

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?

5.1 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 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. Fine-tuning on SWE-bench trajectories uses a 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

TODO: for this I need o explain SQuAD???

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 5.1–5.2 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. 5.1–5.2).

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 5.1 reports SQuAD performance under these condensation strategies. These results, together with the implementation schematics in Fig. 5.1–5.2, 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.

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)

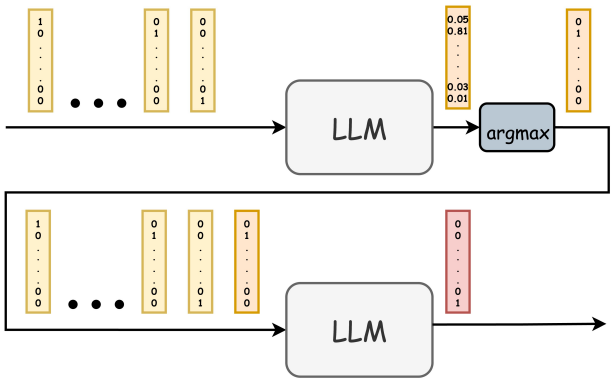

Figure 5.1 Visualization of the "without training" approach, p1

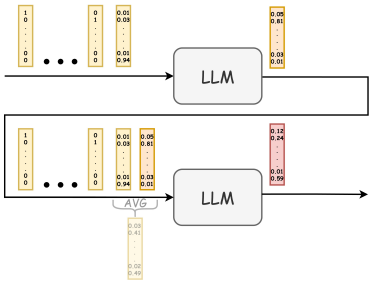

Figure 5.2 Visualization of the "without training" approach, p2

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.

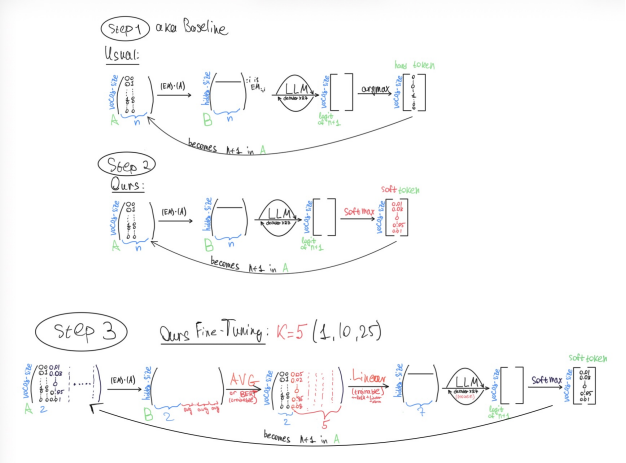

Figure 5.3 Visualization of the baseline approach, step 1-3; should be redrawn

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 5.4.

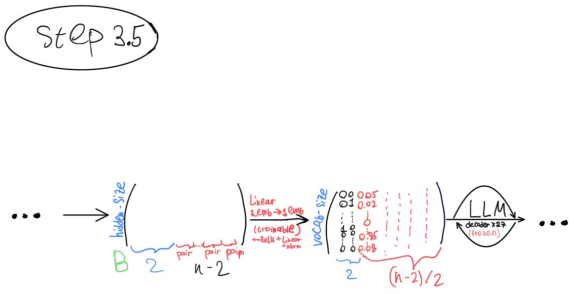

Figure 5.4 Visualization of the experimental setup, step 3.5; should be redrawn

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 (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 $\sim$ 50%–80% relative F1 drop visible for averaging).

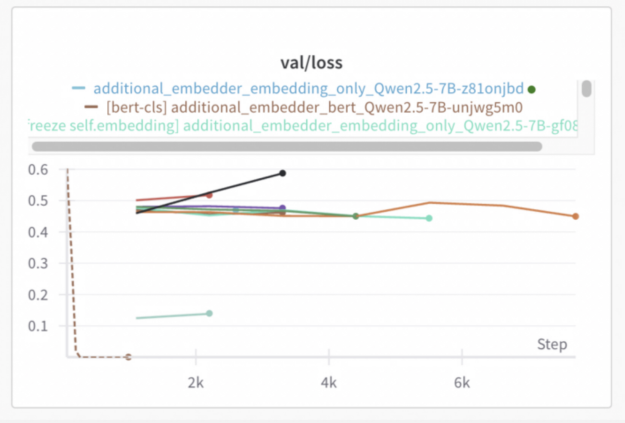

Figure 5.5 Validation loss curves for the SQuAD generalization experiment using linear, MLP and BERT

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.

5.4 ICAE Pretraining and Evaluation 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, $1024 \rightarrow 256$ 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 5.2 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 ($\approx$ 0.982). Notably, none of the 100-sample AE scores reach $\approx$ 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).

5.5 ICAE Fine-Tuning and Evaluation 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 5.3 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	<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 5.3 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.

5.6 [Very main part?] ICAE Fine-Tuning and Evaluation on SWE-bench Verified

Here we describe the main experiment of the thesis and discuss its results. We should start by describing the pictures 5.6 and 5.7. Picture 5.6 discribes our contribution - training procedure of ICAE model to an agentic scenario (more specifically SWE-bench Verified dataset). Firstly, we have trajectories to train on (generated by Claude and described in Section **??**). Then, we explain how to train useing them. One reajectory is basically a loop of LLM making Step K (this generates tool call), then we provide it to the environment and recieve an observation. Observation is then passed to LLM's step K+1, and so on. In the strart there is an uncompressed user prompt. Compression happens for every observation, so that the decoder Qwen3 never sees any observation longer than 256 tokens in plain text, only in compressed form (embeddings).

More details of the LLM's step are in the picture 5.7. It is one LLM's step in detail. The observation is passed to encoder, which produces compressed representation of the observation. Then this compressed representation (with all the previous history obviously) is passed to the decoder, which generates the next tool call for the next step. We then compute the loss as a cross-entropy on the tool call token-by-token, since we have the reference tool call from smarter model; and backpropagate through the decoder and the encoder (only LoRA weights of the encoder are updated). Note that the encoder and decoder models are both qwen3 and same. The only difference is that the encoder has a LoRA adapter, which is trained to produce compressed representations of the observations.

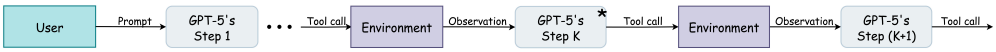

Figure 5.6 ICAE application to SWE-bench

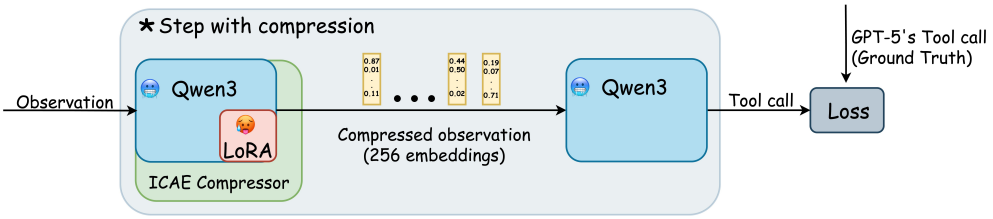

Figure 5.7 ICAE application to SWE-bench

Then we have to describe the results in the table 5.4. First of all we need to describe the table structure: we have 5 columns: Encoder, Decoder, Accuracy, Time and Resolved.

Encoder is the model that is used to compress the observations ("—" means no encoder).

Decoder is the model that is used to generate the next tool call ("Qwen" means uncompressed Qwen3-8B, "Qwen-LoRA-FT" means LoRA fine-tuned Qwen3-8B, "Qwen-Full-FT" means full fine-tuned Qwen3-8B).

Accuracy is the token-wise accuracy of the model on the SWE-bench Verified dataset (described in more details in Section **??**)

Resolved is the number of issues resolved by the model (resolved issues / total issues) out of 500 issues.

Time is the time taken by the model to generate the next tool call (mean time per tool call) in seconds.

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 5.4, 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	26	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	0.9263	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 5.4 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 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 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., 1.12s vs 1.23s). It took on average 0.31s to compress the observation and 0.81s to generate the next tool call. It should be noted that all the experiments in the table do not use KV-caching. More on that is in Chapter 6 limitations section.

5.7 Disabling Thinking Experiments

TODO: maybe we push it to appendix?

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

Table 5.5 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 5.4, were conducted using the official "Clean" prompting methodology to ensure the validity and reproducibility of our findings.

6 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"

6.1 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).

6.2 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.

6.3 Constraints on Computational Resources

6.3.1 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.

6.3.2 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.

6.3.3 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.

6.4 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.

7 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:

7.1 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.

7.2 Summary of Achievements

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

7.3 Synthesis of Findings

Describe the conclusions of all the findings (additional experiments)

7.4 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

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 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.3.4 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].

TODO: 4 is Pretrained model checkpoints achieving 95% reconstruction BLEU:

TODO: 5 is Fine-tuned models that outperform uncompressed baselines on SQuAD:

[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**

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.