PIXELMEM: PERSISTENT IMAGES FOR CONTEXT EFFICIENT LANGUAGE-MODEL MEMORY

Anonymous authors
Paper under double-blind review

ABSTRACT

For language models to support long-term user interactions, they need to retain and reuse user information, past events, and procedural knowledge. These memories require storage space, and reading them back into the model occupies its limited input capacity. Rendering text as images can shorten model inputs, but may enlarge the storage cost. This motivates a memory format that supports both compact storage and short model inputs. We introduce PixelMem, which encodes information from text memories into the pixel values of 16×16 images. A trained visual reader maps each image to a small number of continuous memory slots, which the language model uses together with the question to generate an answer. The image size remains fixed, while the reader determines how many input positions the memory occupies. We evaluate storage, input length, and answer accuracy on four types of tasks, ranging from simple facts to detailed procedures. Each question is answered from a single memory. With 16 memory input positions, PixelMem uses 74.25% fewer input cost than full text and 93.98% less storage than rendered-text images under the same input budget. Its accuracy is 89.4%, only 3.1 percentage points below full text. In a separate comparison with memories stored as vectors, it reaches 76.2% accuracy with four memory input positions, exceeding the vector baseline's 48.5% by 27.7 percentage points. These results suggest that small pixel images can preserve information useful for answering questions while requiring few model input positions.

1 INTRODUCTION

As language models are used for sustained interactions and cross-session tasks, systems need to accumulate and reuse user information, past events, and procedural knowledge (Park et al., 2023; Zhong et al., 2024). Using these memories involves two connected stages: persistently storing the information and bringing relevant memories back into the model for subsequent tasks. Storing a memory requires space for its representation, while reading it occupies additional positions in the model's limited context window and requires computation during inference. The challenge for efficient memory systems is to store memories compactly and support downstream tasks with few memory input token cost.

DeepSeek-OCR shows that dense document text can be represented using relatively few visual tokens (Wei et al., 2025). This capability suggests using images as memory carriers to reduce input costs. OCR-Memory further applies visual representations to storing and retrieving historical agent trajectories (Li et al., 2026). However, reducing the context passed to a language model does not automatically yield a smaller persistent representation: rendering text as images can substantially increase disk usage. This motivates us to investigate how memory can combine a compact persistent representation with a low-overhead reading interface.

To combine compact storage with low input overhead, we propose PixelMem, a method that encodes memories into small images for storage and uses a learned reader to convert them into language-model inputs. We first use a text encoder to map a memory to a semantic embedding, then apply a fixed projection and quantization scheme to store it as a 16×16 RGB PNG image. The image stores numerical codes derived from the semantic embedding rather than readable text. When the language model needs to use a memory for inference, a trained visual module reads the image using global pixel information and local patch features. This module produces a fixed number of continuous

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

memory slots. Each slot is a continuous vector occupying one language-model input position. The image defines the persistent storage format, while the slot budget controls context occupancy when the memory is used.

We design four types of synthetic single-memory question-answering (QA) tasks ranging from simple to complex: atomic facts, temporal events, complex facts, and procedural steps. Across these tasks, we compare storage bytes, memory input positions, and QA accuracy for text, vector, and image representations. Relative to full text, sixteen-slot PixelMem reduces the average number of memory input positions from 62.13 to 16, a 74.25% reduction, while accuracy decreases from 92.5% to 89.4%, a difference of 3.1 percentage points. Relative to our rendered-text image baseline at the same 16-position memory budget, it reduces the average file size per memory from 14,162.66 to 852 bytes, a reduction of approximately 94%. In a separate comparison with direct FP16 vectors at the same four-slot budget, PixelMem achieves 76.2% accuracy versus 48.5%, a gain of 27.7 percentage points. These results characterize the trade-offs among persistent storage, context occupancy, and QA accuracy under limited input budgets.

Our contributions are threefold. (1) **PixelMem.** To our knowledge, we are the first to use small pixel pictures encoding quantized semantic embeddings to store and read language-model memories. (2) **Visual memory reader.** We design a visual reader that combines global pixel information with local patch features. It converts each stored image into a small number of continuous memory slots for language-model question answering. (3) **Joint evaluation.** We jointly evaluate persistent storage bytes, memory input positions, and task accuracy across four types of memory tasks, comparing our method with text, rendered-text images, and direct vectors.

2 Related Work

Long-term memory systems retain and reuse information from past interactions. Generative agents stores observations and reflections, while MemoryBank, Mem0, and MemGPT support persistent conversational or external memory (Park et al., 2023; Zhong et al., 2024; Chhikara et al., 2025; Packer et al., 2023). Other approaches extend model context through recurrent, compressed, retrieved, or latent memory representations (Dai et al., 2019; Rae et al., 2020; Wu et al., 2022; Bulatov et al., 2022; Wang et al., 2023; Xiao et al., 2024; Wang et al., 2024; Kim et al., 2024; Bertsch et al., 2023; Chen et al., 2025; Cetin et al., 2025). These methods primarily study how information is retained, retrieved, or propagated across long contexts. PixelMem instead focuses on the persistent representation of an already selected memory, and reading it into the language model with few input positions.

The number of model input positions can be reduced either by selecting informative text or by encoding context into compact continuous representations. Token- and sentence-selection methods include Selective Context, LLMLingua, LLMLingua-2, and LongLLMLingua (Li et al., 2023b; Jiang et al., 2023; Pan et al., 2024; Jiang et al., 2024). Other methods compress context into learned latent states or a small number of tokens, including Gist Tokens, AutoCompressors, the In-context Autoencoder, Nugget, Dodo, xRAG, 500xCompressor, Activation Beacon, and COMI (Mu et al., 2023; Chevalier et al., 2023; Ge et al., 2024; Qin & Van Durme, 2023; Qin et al., 2024; Cheng et al., 2024; Li et al., 2025; Zhang et al., 2025; Tang et al., 2026). PixelMem also supplies compact continuous slots to the language model to reduce input token cost.

Vision-language models also compress visual features into a small set of latent representations, often using learned queries (Jaegle et al., 2021; Ryoo et al., 2021; Alayrac et al., 2022; Li et al., 2023a). Text can also be rendered as images and read through a visual encoder. DeepSeek-OCR encodes document images into a small number of visual tokens and recovers the text from them (Wei et al., 2025). OCR-Memory renders past agent trajectories as images and retrieves the text from relevant image regions (Li et al., 2026). Lee et al. (2025) compare optical compression with methods that compress text embeddings directly. In their experiments, the direct encoders perform as well as or better than the visual encoder. We therefore include a direct-vector baseline in our experiments. PixelMem stores quantized numerical codes derived from semantic embeddings in the image, rather than rendered text. When needed, a trained visual reader converts the stored image into continuous memory slots for question answering, without reconstructing the original text.

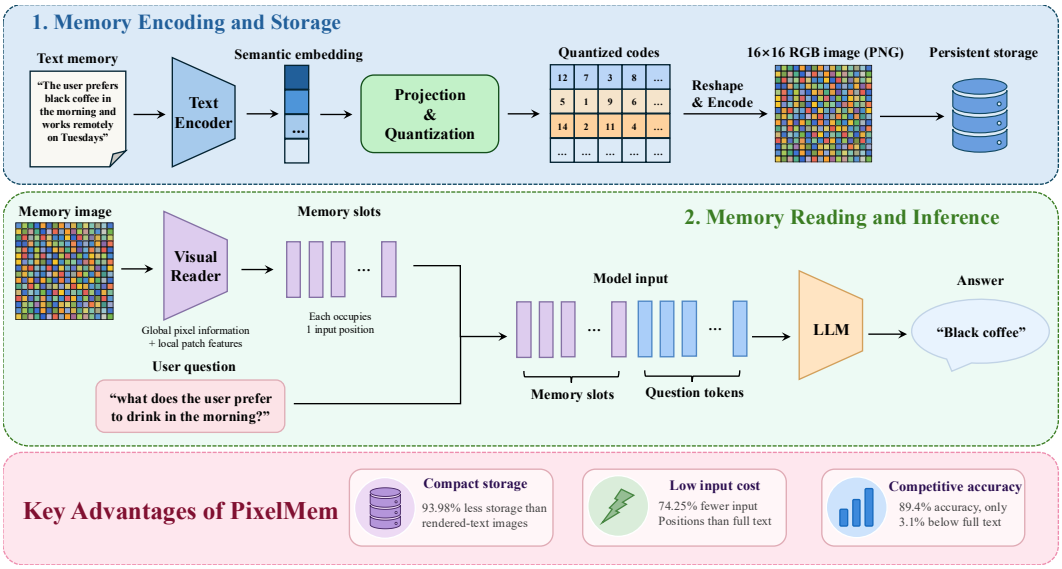

Figure 1: Overview of PixelMem. A fixed writer stores each memory as a 16×16 RGB PNG; a trained visual reader converts it into K slots for question answering. The bottom row summarizes our key advantages.

3 METHOD

PixelMem writes a memory text m to a small pixel image I for storage. When the memory is needed, a trained visual reader R_ϕ converts the image into K continuous memory slots. These slots form $H_K \in \mathbb{R}^{K \times d_{\text{LM}}}$, where d_{LM} is the language model's hidden width. Each slot is a continuous vector supplied directly as an input embedding and occupies one input position. The language model takes these slots together with a question q and generates an answer. Thus, K determines the memory input length, excluding the question and prompt. Figure 1 shows the writing and reading process. The writer and reader process only the memory; the question enters at the language-model stage.

3.1 WRITING MEMORIES AS PIXEL IMAGES

We first encode the memory text with the frozen `BAAI/bge-small-en-v1.5` text encoder E (BAAI, 2023). We take its CLS output as the memory's semantic representation and apply L2 normalization to obtain a unit-length vector:

$$z = \frac{\text{CLS}(E(m))}{\|\,\text{CLS}(E(m))\|_2}, \qquad z \in \mathbb{R}^{384}. \tag{1}$$

We map the embedding to pixel values using a fixed projection $u = zP$. The matrix $P \in \mathbb{R}^{384 \times 768}$ satisfies $PP^\top = I_{384}$. This projection maps the embedding to 768 coordinates, one for each RGB channel value in a 16×16 image. Each coordinate is clipped and quantized as

$$p_j = 128 + \text{round}\Big(127\ \text{clip}\Big(\frac{u_j}{\alpha}, -1, 1\Big)\Big). \tag{2}$$

Here α is a clipping threshold fitted on the training split and fixed during QA training. The offset of 128 allows each quantized value p_j to be stored as an unsigned 8-bit integer. The projection determines how the embedding is mapped to pixel values, and quantization reduces the precision used to store them.

We arrange the 768 values in a fixed order as a 16×16 RGB image I and save it as a PNG. The image contains numerical codes derived from the semantic embedding, rather than rendered characters. Its raw pixel payload is 768 bytes; the serialized file also includes PNG format overhead. Because writing depends only on the memory text, the same stored image can be reused for different questions. Appendix B gives the projection, quantization, and serialization details.

3.2 Reading Pixel Memories

The visual reader loads the stored pixel values and normalizes them as $x = (p - 128)/127$. It then extracts features through global and local branches. The global branch maps the flattened pixel array to two types of features. An MLP produces a global vector $g \in \mathbb{R}^{d_r}$ shared by all slots. A separate linear map produces an individual feature for each slot, $S \in \mathbb{R}^{K \times d_r}$, where d_r is the reader's hidden width. Both outputs depend on the input memory. In the local branch, a convolution embeds the image's 64 non-overlapping 2×2 RGB patches. We add learned positional vectors to these patch embeddings and pass them through a Transformer to obtain patch features $F \in \mathbb{R}^{64 \times d_r}$. Thus, the reader has access to both the complete pixel array and features computed from local groups of pixel values.

The reader aggregates these features into K slots. It starts from a learned query array $Q_{\text{learned}} \in \mathbb{R}^{K \times d_r}$ and adds the shared global feature and the individual slot features:

$$Q_0 = Q_{\text{learned}} + \text{Broadcast}(g) + S. \tag{3}$$

Broadcast repeats g across the K slots. The same learned query array initializes the slots for every memory. Adding g and S makes the initial slots depend on the current memory. The queries do not depend on the question q. Each reader block uses the current slot representations as queries and F as keys and values for cross-attention. Self-attention among slots and a feed-forward network then further update the slots. The resulting slots are normalized and projected to the language model's hidden width:

$$H_K = \text{Proj}_{d_r \to d_{\text{LM}}} \left(\text{LayerNorm} \left(\text{ReaderBlocks}(Q_0, F) \right) \right). \tag{4}$$

We use the rows of H_K as input embeddings at the K reserved memory positions of Qwen3-VL-8B-Instruct. The language model processes these memory embeddings together with the text embeddings of the question and prompt to generate an answer. Our visual reader supplies these embeddings without reconstructing the source text. The stored image remains 16×16, while K controls how many input positions its readout occupies. Appendix B provides the reader architecture and the embedding interface used to supply these slots to the language model.

3.3 Training for Memory Question Answering

For a training example with memory image I, question q, and target answer y, the reader produces $H_K = R_\phi(I)$. We train the visual reader parameters ϕ and language-model LoRA parameters θ with autoregressive cross-entropy on the answer tokens:

$$\mathcal{L}_{\text{QA}} = -\frac{1}{|y|} \sum_{t=1}^{|y|} \log p_{\theta,\phi}(y_t \mid q, H_K, y_{<t}). \tag{5}$$

Here $|y|$ is the number of answer tokens and $y_{<t}$ denotes the preceding answer tokens. Question, prompt, memory, and padding positions are excluded from the loss. Answer supervision trains the reader to produce slots that support QA; it does not require reconstructing the memory text or explicitly recovering its semantic embedding.

The text encoder, projection, quantization rule, and language-model base weights remain fixed during QA training. Only the visual reader and LoRA are updated. All PixelMem configurations with $K \in \{4, 8, 16\}$ use the same training data and auxiliary text QA examples. For these examples, we provide the memory as text. Answer supervision updates the LoRA parameters shared with the pixel-memory route. The visual reader does not participate in these updates. We use separately adapted readers and LoRA parameters for $K \in \{4, 8, 16\}$, so each input budget corresponds to its own trained configuration. The training schedules, mixture proportions, initialization, and checkpoint selection are described in Appendix C.

4 Experiments

4.1 Experimental Setup

We evaluate QA from a single selected memory using an English template-generated corpus of 9,600 memories and 48,000 semantic QA pairs. We split the corpus by episode into training, development,

Table 1: Dataset and representation comparison. (a) Memory types, evaluation shares, and mean full-text length. (b) Persistent storage, memory input positions, and overall QA accuracy in Batch I.

(a) Memory types and dataset statistics

Memory type	Typical question targets	Corpus (%)	Eval. (%)	Tokens
Atomic facts	Person, project, or location	30	25	10.40
Temporal events	Actor, action, reason, or result	40	25	33.88
Complex facts	Impact, follow-up, or handover	20	25	66.84
Procedural steps	Trigger, action, verification, rollback	10	25	137.40

(b) Representation comparison

	Text			Rendered text		PixelMem
	Full	Sel. 16	Sel. 64	16 pos.	Readable	16 slots
File size (B/memory)	326.66	326.66	326.66	14,162.66	28,563.08	852.00
Memory input positions	62.13	12.97	41.31	16.00	197.52	16.00
Overall QA accuracy (%)	92.5	42.4	85.4	42.8	93.8	89.4

and test sets containing 8,000, 800, and 800 memories, respectively. The templates cover project and credential responsibilities, event records, handovers, and operational procedures.

We use four memory types to assess semantic retention under limited input budgets as contextual and procedural detail increases (Table 1(a)). Atomic facts cover basic relations involving people, projects, locations, and ownership. Temporal events add actors, actions, reasons, and results; complex facts add consequences, follow-up plans, and handover details. Procedural steps combine actions with triggers, verification, rollback, constraints, and follow-up. Questions cover 13 semantic fields. Each memory has five questions about applicable fields, each with four training wordings and one held-out wording. Evaluation uses the original and held-out wordings. Appendix A provides construction details and an example with wording variants.

We conduct two comparisons. The representation comparison (Batch I) evaluates text, rendered images, and pixel memories on 100 test memories balanced across the four categories. Each configuration produces 1,000 answers. The budget comparison (Batch II) evaluates pixel and FP16-vector memories at 4, 8, and 16 slots, alongside budgeted and full-text inputs. Each configuration is evaluated on 1,000 answers, with 250 per category.

Text baselines use the full memory or meet a token budget by keeping its beginning and end or selecting sentences relevant to the question. Rendered-text baselines use Qwen3-VL's native visual encoder to read text images, which can be downscaled to use fewer visual input positions. The direct-vector baseline uses a trained reader to convert BGE embeddings into continuous memory slots. The primary text and rendered-image baselines disable LoRA, while the vector and pixel-memory methods each use their own trained readers and adapters. In the representation comparison, text and rendered-image inputs have exact values masked, while pixel memories are encoded from the original, unmasked text. All methods use the same memory IDs, questions, and reference answers. Appendix D explains cost measurement. Appendix C describes training for the main sixteen-slot Qwen3 system, and Appendix A.4 lists the evaluation subsets and sample counts.

4.2 Storage, Context, and QA Accuracy

Table 1(b) compares persistent storage, memory input positions, and overall QA accuracy in the representation comparison; Figure 2(a) shows accuracy by memory type. With 16 memory slots, PixelMem reduces the mean input from 62.13 full-text positions by 74.25%. Accuracy decreases from 92.5% to 89.4%, a loss of 3.1 percentage points. Each pixel memory occupies 852 B, or 2.61 times the average text file size of 326.66 B. Relative to full text, this configuration trades a larger persistent file and some answer accuracy for a shorter memory input.

The trade-off varies across memory types. Atomic facts average only 10.40 text positions, and both full text and PixelMem reach 98.8% accuracy. Using 16 slots therefore increases the input length by

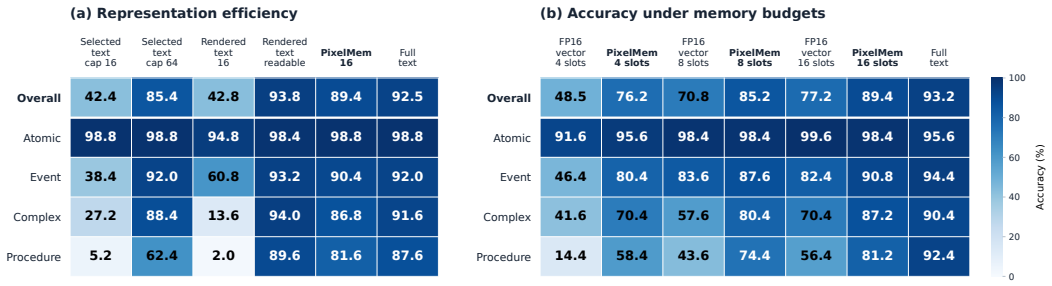

Figure 2: Semantic QA accuracy (%) by representation in Batch I (left) and slot budget in Batch II (right); 1,000 answers per configuration.

53.85%. For these short relational memories, full text already provides accurate answers with fewer input positions.

Temporal events add information about actions, reasons, and results. Their full-text inputs average 33.88 positions. PixelMem reduces this length by 52.77%, with accuracy decreasing from 92.0% to 90.4%. It thus retains most of the full-text QA accuracy while using less than half the memory input. Rendered text at the same 16-position budget reaches 60.8% accuracy.

Complex facts include consequences, follow-up plans, and handover details. Their mean input falls from 66.84 text positions to 16 slots, a 76.06% reduction, while accuracy decreases from 91.6% to 86.8%. Question-aware text selection under a 64-token cap uses 60.13 positions on average and reaches 88.4%, 1.6 points above PixelMem. Pixel memories provide a shorter input for this task, with a measurable loss in answer accuracy.

Procedural steps have the longest full-text inputs, averaging 137.40 positions. PixelMem reduces this length by 88.36%, with accuracy decreasing from 87.6% to 81.6%. Under a 64-token cap, selected text uses 60.82 positions and reaches 62.4% accuracy. PixelMem uses fewer positions and scores 19.2 points higher. This procedural gain accounts for its overall advantage over the 64-token text baseline: atomic-fact accuracy is equal, while event and complex-fact accuracy are each 1.6 points lower. Across all four types, PixelMem reaches 89.4% with 16 positions, compared with 85.4% and 41.31 positions for selected text.

Compared with rendered text at the same 16-position budget, PixelMem reduces mean file size from 14,162.66 B to 852 B, a 93.98% reduction. The reductions are 83.64%, 93.35%, 95.22%, and 95.91% for atomic facts, temporal events, complex facts, and procedural steps, respectively. Overall accuracy rises from 42.8% to 89.4%. Readable rendered images reach 93.8% accuracy, but require an average of 197.52 memory positions and 28,563.08 B per memory. These comparisons assess the representations together with their readers and adaptation conditions described in Section 4.1. Appendix E provides the remaining configurations.

4.3 Performance under Limited Input Budgets

We next compare pixel and FP16-vector memories at 4, 8, and 16 slots using Batch II (Figure 2(b)). All three slot-budget configurations use the same training data and auxiliary-text training; they differ only in slot count and their separately adapted reader and LoRA parameters. Batch I and Batch II use independently sampled balanced subsets of 100 test memories. PixelMem reaches 76.2%, 85.2%, and 89.4% accuracy, compared with 48.5%, 70.8%, and 77.2% for direct vectors. The corresponding gains are 27.7, 14.4, and 12.2 percentage points. Pixel memories score higher at every tested budget, with the largest overall difference at four slots. The right panel reports overall and per-category accuracy for the six pixel and vector configurations, with uncapped full text as a reference.

Reducing the pixel budget from 16 to 8 slots costs 4.2 percentage points; reducing it to 4 slots costs 13.2 points in total. The vector route loses 28.7 points between 16 and 4 slots. Across budgets, four-slot PixelMem exceeds the eight-slot vector by 5.4 points, and eight-slot PixelMem exceeds

the sixteen-slot vector by 8.0 points. These configurations therefore achieve higher accuracy with fewer memory positions than the corresponding larger-budget vector configurations.

The category results in Figure 2(b) show that atomic facts benefit less from the pixel method. Its advantage is 4.0 points at four slots; the two methods tie at eight slots, and the vector scores 1.2 points higher at sixteen slots. Both reach 98.4% at eight slots. Temporal events show a larger difference at four slots: PixelMem reaches 80.4%, compared with 46.4% for vectors. The gap is smaller at eight and sixteen slots, at 4.0 and 8.4 points, respectively.

Complex facts and procedural steps favor PixelMem at all three budgets. For complex facts, its advantage decreases from 28.8 points at four slots to 16.8 points at sixteen slots. Procedures show the largest category difference at every budget. At four slots, pixel and vector memories score 58.4% and 14.4%, respectively; at sixteen slots, they reach 81.2% and 56.4%. The procedural advantage therefore remains 24.8 points even at the largest tested budget. In these tasks, the pixel representation and reader retain higher QA accuracy across the tested input budgets. The persistent pixel image remains 852 B at all three budgets, compared with 768 B for the FP16 vector. The pixel method therefore uses 84 additional bytes per memory for the observed gains in QA accuracy.

5 Discussion

5.1 Storage and Reading Trade-offs

Persistent storage and model context are used at different stages. A memory file is stored once and can be reused across questions, while its readout occupies input positions whenever it is supplied to the language model. PixelMem separates these choices by retaining a fixed pixel code and producing a specified number of continuous slots. Our 4, 8, and 16 slot configurations use separately adapted readers and LoRA parameters. Using a different budget means selecting its corresponding trained configuration; the memory file itself can be reused. This allows the storage representation to remain fixed as the reading configuration trades QA accuracy against memory input length.

The value of this trade-off depends on the memory being read. In the representation comparison, full text matches sixteen-slot PixelMem on short atomic facts while occupying fewer input positions. These memories already express the required relation concisely, so a fixed slot budget can increase their context use. Event records, complex facts, and procedures contain more contextual or operational detail. Their longer text inputs offer greater scope for reducing memory positions, although the encoded representation incurs a loss in QA accuracy. This favors choosing a representation according to the available context and the accuracy required by the task. Memory type and text length vary together in our corpus, so their individual effects would require further controlled comparisons.

Storage savings also depend on the reference representation. Pixel codes avoid the larger files needed to store rendered characters in our image baselines. The full-text baseline has smaller files still, making text attractive when persistent storage is the main constraint. In the budget comparison, the pixel method uses larger files than the FP16-vector route and achieves higher QA accuracy at the same slot counts. The benefit in this comparison is therefore more effective use of the allocated memory positions. The comparison includes the reader architectures and training histories described in Section 4.1, so the performance difference should not be attributed to the representation alone. Section 5.2 further examines how the global and local branches contribute to memory decoding.

Memory positions account for only part of the cost of using a memory. A complete request includes the prompt, question, and generated answer. The pixel reader processes 64 internal image patches before producing the memory slots. Those patches incur computation but do not become additional language-model input positions. At the same slot count, pixel and vector memories occupy the same number of memory positions and have the same memory-associated KV-cache size. KV-cache compression methods address a complementary source of inference cost by selecting or quantizing cached attention states (Zhang et al., 2023; Liu et al., 2023; Li et al., 2024; Liu et al., 2024b; Nawrot et al., 2024; Liu et al., 2024a; 2025; Yu et al., 2026; Zhang et al., 2024). Their readers can nevertheless differ in computation and latency. Creating either a pixel or a vector memory requires text encoding; the pixel route also constructs an image. Shared encoder, reader, and adapter weights add costs beyond the per-memory files (Appendix D). Assessing deployment efficiency therefore requires measuring these costs together with how often memories are written and reused.

5.2 Branch Ablation and Memory Mismatch

PixelMem combines global pixel information with local patch features. Because removing a branch can disrupt computation learned during training, accuracy drops alone do not establish whether that branch carries memory-specific content. We therefore combine branch ablation with memory mismatch. All reader, LoRA, and base-model weights are frozen, and we evaluate the 4- and 16-slot checkpoints on 200 questions from 20 balanced test memories. Global-only removes patch processing, Patch-only removes the global mappings, and Wrong-patch/Wrong-global replace one branch's input with a same-category donor memory having a different reference answer while retaining the full computation. We report target QA accuracy and donor-field match rate; Appendix F provides the full protocol.

Table 2: Additional analyses of PixelMem. (a) Frozen-reader branch interventions at 4 and 16 slots. A and B denote target and same-category donor memories. (b) QA accuracy across language models using identical pixel memories and 16 memory slots. Scores are percentages.

(a) Branch ablation and memory mismatch

			4 slots		16 slots	
Configuration	Global	Patch	Target	Donor	Target	Donor
Full	A	A	68.5	—	76.5	—
Global-only	A	—	48.5	—	56.0	—
Patch-only	—	A	41.0	—	49.5	—
Wrong-patch	A	B	43.0	31.5	49.0	37.5
Wrong-global	B	A	35.0	44.0	40.5	52.0

(b) Performance across language models

Model	Overall	Atomic	Event	Complex	Proc.
Qwen2.5-VL-7B	63.1	96.4	70.8	50.8	34.4
Qwen3-VL-8B	89.4	98.8	90.4	86.8	81.6
Difference (pp)	+26.3	+2.4	+19.6	+36.0	+47.2

Removing either branch substantially reduces QA accuracy (Table 2(a)). At four slots, Global-only and Patch-only reach 48.5% and 41.0%, compared with 68.5% for the full reader. At sixteen slots, their accuracies are 56.0% and 49.5%, compared with 76.5% for Full. Thus, neither branch alone recovers the performance of the complete reader.

The mismatch interventions further show that both branches carry memory-specific information. Replacing the patch input reduces target accuracy from 68.5% to 43.0% at four slots and from 76.5% to 49.0% at sixteen slots. At the same time, donor-field match rates increase to 31.5% and 37.5%. Replacing the global input produces an even larger shift: target accuracy falls to 35.0% and 40.5%, while donor-field match rates rise to 44.0% and 52.0%. The generated answers therefore respond to the memory content supplied through either branch rather than depending only on one fixed computation path.

These results indicate that the two branches play complementary roles. The global branch carries the stronger memory-specific signal, as changing its input produces the larger donor shift. However, the local patch branch also contributes substantial target-specific information: replacing only its input causes a large loss in target accuracy and redirects a substantial fraction of answers toward the donor memory. This behavior differs from a reader that merely treats the image as a flat collection of quantized values. Instead, the reader uses both global pixel information and local patch features to decode the stored memory. The mismatch results show that local pixel content is not redundant: both global and local information contribute to memory decoding in the trained reader.

5.3 Performance Across Language Models

We examine whether the same stored pixel memory can be used effectively with different language models. We compare PixelMem on Qwen2.5-VL-7B-Instruct and Qwen3-VL-8B-Instruct using

byte-identical pixel memories, 16 memory slots, and the same 1,000 test questions. The two systems use closely matched reader architectures, LoRA configurations, training data, and adaptation procedures, while their parameters are optimized separately for each model.

Qwen3-VL reaches 89.4% overall accuracy, compared with 63.1% for Qwen2.5-VL (Table 2(b)). The difference is small for atomic facts (+2.4 percentage points) but substantially larger for temporal events (+19.6), complex facts (+36.0), and procedures (+47.2). The same stored pixel code can therefore support markedly different QA accuracy across language models, especially for memories containing richer contextual and procedural information.

Because the stored files, slot budget, and adaptation setup are closely matched, the difference reflects how each adapted model reads and uses the same stored pixel code. The results suggest that stronger language models, paired with suitably trained readers, may better extract and use information from pixel memories. Thus, PixelMem performance depends on how well the language model and its memory reader work together.

5.4 Limitations and Future Directions

Available experimental resources limited the complexity of the datasets we could construct and the range of models we could evaluate. Our four task types cover factual, temporal, and procedural information, but evaluation remains focused on QA from individual memories. More demanding datasets could include longer event sequences, dependencies across memories, and questions requiring several reasoning steps. These settings would help examine how the trade-offs among storage, input length, and accuracy change as memory content becomes more complex. The current corpus is also limited to English. Extending it to other languages and domain-specific content would broaden the evaluation.

Our main experiments use Qwen3-VL-8B-Instruct, an 8B-parameter model, without systematic comparisons with larger models or other model families. Such comparisons would help determine how the effectiveness of pixel memories depends on model scale and architecture. They could also examine how much adaptation different models need to use the same stored pixel codes.

We use a fixed 16×16 pixel representation and separately adapted readers for 4, 8, and 16 memory slots. Other combinations of storage capacity and reading budgets remain to be explored. Future work could vary image resolution and quantization precision, and train readers that support variable slot counts. These experiments would help determine how to choose storage and input budgets for memories of different lengths and tasks with different accuracy requirements.

6 Conclusion

We introduced PixelMem, a memory method that stores quantized semantic embeddings as 16×16 RGB PNG images and uses a learned visual reader to produce continuous language-model input slots. The stored image can be reused across questions and reading configurations. Separately trained readers with different slot counts allow the input budget to change while the persistent code remains fixed.

Across four types of synthetic single-memory QA tasks, we jointly evaluated storage, memory input positions, and answer accuracy. With 16 slots, PixelMem reduces memory input positions by 74.25% relative to full text with only a 3.1-point accuracy loss, and reduces file size by 93.98% relative to rendered text at the same input budget. Compared with direct FP16 vectors, PixelMem achieves higher overall accuracy at all 4-, 8-, and 16-slot budgets, with the largest gain of 27.7 percentage points at four slots. These results highlight favorable trade-offs among context use, storage, and QA accuracy.

Frozen-reader interventions show that both global pixel values and local patch features contribute memory-specific information, with the two branches playing complementary roles in memory decoding. The same pixel files can also be reused across the two tested Qwen models after adaptation, with higher QA accuracy in the Qwen3-based system. Together, these findings support compact pixel storage coupled with a learned reading interface for QA under limited memory input budgets. Future work will examine more complex tasks involving multiple memories and a wider range of model sizes and families.

AI USE STATEMENT

We used generative AI tools to assist with language editing and parts of the experimental design. We take responsibility for the final content of this work.

ETHICS STATEMENT

We do not identify ethical concerns arising from the experiments and datasets in this work.

REPRODUCIBILITY STATEMENT

Appendix A describes dataset construction, data splits, question variants, and evaluation sample counts. Appendix B specifies the pixel encoding procedure and reader architecture, while Appendix C documents the training procedure for the main sixteen-slot Qwen3 model. Appendix D defines storage and context accounting, Appendix E reports the complete comparison results, and Appendix F describes the branch intervention protocol. The accompanying supplementary package contains the dataset, encoded PNG memories, encoding parameters, reader implementations and configurations, and the trained reader and corresponding LoRA weights for the main sixteen-slot Qwen3 model.

REFERENCES

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob L. Menick, Sebastian Borgeaud, Andy Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikołaj Bińkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karén Simonyan. Flamingo: a visual language model for few-shot learning. In *Advances in Neural Information Processing Systems*, volume 35, 2022. doi: 10.52202/068431-1723. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/960a172bc7fbf0177ccccbb411a7d800-Abstract-Conference.html.

BAAI. bge-small-en-v1.5 model card. Hugging Face, 2023. URL https://huggingface.co/BAAI/bge-small-en-v1.5. Accessed September 18, 2026.

Amanda Bertsch, Uri Alon, Graham Neubig, and Matthew R. Gormley. Unlimiformer: Long-range transformers with unlimited length input. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-1543. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/6f9806a5adc72b5b834b27e4c7c0df9b-Abstract-Conference.html.

Aydar Bulatov, Yury Kuratov, and Mikhail Burtsev. Recurrent memory transformer. In *Advances in Neural Information Processing Systems*, volume 35, pp. 11079–11091, 2022. doi: 10.52202/068431-0805. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/47e288629a6996a17ce50b90a056a0e1-Abstract.html.

Edoardo Cetin, Qi Sun, Tianyu Zhao, and Yujin Tang. An evolved universal transformer memory. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/da85790fb1cb4f11f431648455c561b5-Abstract-Conference.html.

Yinpeng Chen, DeLesley Hutchins, Aren Jansen, Andrey Zhmoginov, David Racz, and Jesper Andersen. MELODI: Exploring memory compression for long contexts. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a8e3197f627b9c88f86c0d3eb1ade5d7-Abstract-Conference.html.

Xin Cheng, Xun Wang, Xingxing Zhang, Tao Ge, Si-Qing Chen, Furu Wei, Huishuai Zhang, and Dongyan Zhao. xRAG: Extreme context compression for retrieval-augmented generation with one token. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi:

10.52202/079017-3476. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/c5cf13bfd3762821ef7607e63ee90075-Abstract.html.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 3829–3846. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.232. URL https://aclanthology.org/2023.emnlp-main.232/.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready AI agents with scalable long-term memory. In *ECAI 2025*, Frontiers in Artificial Intelligence and Applications. IOS Press, 2025. doi: 10.3233/FAIA251160. URL https://doi.org/10.3233/FAIA251160.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 2978–2988. Association for Computational Linguistics, 2019. doi: 10.18653/v1/P19-1285. URL https://aclanthology.org/P19-1285/.

Tao Ge, Hu Jing, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. In-context autoencoder for context compression in a large language model. In *International Conference on Learning Representations*, pp. 2591–2607, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/0b276510ec2d3f6613a8b60c41ff0438-Abstract-Conference.html.

Andrew Jaegle, Felix Gimeno, Andy Brock, Oriol Vinyals, Andrew Zisserman, and Joao Carreira. Perceiver: General perception with iterative attention. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 4651–4664. PMLR, 2021. URL https://proceedings.mlr.press/v139/jaegle21a.html.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LLMLingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 13358–13376. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.825. URL https://aclanthology.org/2023.emnlp-main.825/.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LongLLMLingua: Accelerating and enhancing LLMs in long context scenarios via prompt compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1658–1677. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.91. URL https://aclanthology.org/2024.acl-long.91/.

Jang-Hyun Kim, Junyoung Yeom, Sangdoo Yun, and Hyun Oh Song. Compressed context memory for online language model interaction. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/97dc07f1253ab33ee514f395a82fa7cc-Abstract-Conference.html.

Ivan Yee Lee, Cheng Yang, and Taylor Berg-Kirkpatrick. Optical context compression is just (bad) autoencoding. *arXiv preprint arXiv:2512.03643*, 2025. doi: 10.48550/arXiv.2512.03643. URL https://arxiv.org/abs/2512.03643v2. Version 2, revised April 4, 2026.

Jinze Li, Yang Zhang, Xin Yang, Jiayi Qu, Jinfeng Xu, Shuo Yang, Junhua Ding, and Edith Cheuk-Han Ngai. OCR-Memory: Optical context retrieval for long-horizon agent memory. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10409–10420. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.474. URL https://aclanthology.org/2026.acl-long.474/.

Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. BLIP-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 19730–19742. PMLR, 2023a. URL https://proceedings.mlr.press/v202/li23q.html.

Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. Compressing context to enhance inference efficiency of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 6342–6353. Association for Computational Linguistics, 2023b. doi: 10.18653/v1/2023.emnlp-main.391. URL https://aclanthology.org/2023.emnlp-main.391/.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. SnapKV: LLM knows what you are looking for before generation. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-0722. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/28ab418242603e0f7323e54185d19bde-Abstract-Conference.html.

Zongqian Li, Yixuan Su, and Nigel Collier. 500xCompressor: Generalized prompt compression for large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 25081–25091. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.1219. URL https://aclanthology.org/2025.acl-long.1219/.

Xiang Liu, Zhenheng Tang, Peijie Dong, Zeyu Li, Yue Liu, Bo Li, Xuming Hu, and Xiaowen Chu. ChunkKV: Semantic-preserving KV cache compression for efficient long-context LLM inference. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-0966. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/2987f911151b39cd3a1761e212319e8e-Abstract-Conference.html.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. CacheGen: KV cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pp. 38–56. Association for Computing Machinery, 2024a. doi: 10.1145/3651890.3672274. URL https://doi.org/10.1145/3651890.3672274.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-2279. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/a452a7c6c463e4ae8fbdc614c6e983e6-Abstract.html.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32332–32344. PMLR, 2024b. URL https://proceedings.mlr.press/v235/liu24bz.html.

Jesse Mu, Xiang Lisa Li, and Noah Goodman. Learning to compress prompts with Gist Tokens. In *Advances in Neural Information Processing Systems*, volume 36, pp. 19327–19352. Curran Associates, Inc., 2023. doi: 10.52202/075280-0848. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/3d77c6dcc7f143aa2154e7f4d5e22d68-Abstract.html.

Piotr Nawrot, Adrian Łańcucki, Marcin Chochowski, David Tarjan, and Edoardo Ponti. Dynamic memory compression: Retrofitting LLMs for accelerated inference. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning*

Research, pp. 37396–37412. PMLR, 2024. URL https://proceedings.mlr.press/v235/nawrot24a.html.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023. doi: 10.48550/arXiv.2310.08560. URL https://arxiv.org/abs/2310.08560.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, H. Vicky Zhao, Lili Qiu, and Dongmei Zhang. LLMLingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 963–981. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-acl.57. URL https://aclanthology.org/2024.findings-acl.57/.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative Agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pp. 1–22. Association for Computing Machinery, 2023. doi: 10.1145/3586183.3606763. URL https://doi.org/10.1145/3586183.3606763.

Guanghui Qin and Benjamin Van Durme. Nugget: Neural agglomerative embeddings of text. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 28337–28350. PMLR, 2023. URL https://proceedings.mlr.press/v202/qin23a.html.

Guanghui Qin, Corby Rosset, Ethan C. Chau, Nikhil Rao, and Benjamin Van Durme. Dodo: Dynamic contextual compression for decoder-only LMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9961–9975. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.536. URL https://aclanthology.org/2024.acl-long.536/.

Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SylKikSYDH.

Michael S. Ryoo, A. J. Piergiovanni, Anurag Arnab, Mostafa Dehghani, and Anelia Angelova. TokenLearner: What can 8 learned tokens do for images and videos? In *Advances in Neural Information Processing Systems*, volume 34, 2021. URL https://arxiv.org/abs/2106.11297.

Jiwei Tang, Shilei Liu, Zhicheng Zhang, Yujin Yuan, Libin Zheng, Wenbo Su, and Bo Zheng. COMI: Coarse-to-fine context compression via marginal information gain. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/4d893f766ab60e5337659b9e71883af4-Abstract-Conference.html.

Weizhi Wang, Li Dong, Hao Cheng, Xiaodong Liu, Xifeng Yan, Jianfeng Gao, and Furu Wei. Augmenting language models with long-term memory. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-3259. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/ebd82705f44793b6f9ade5a669d0f0bf-Abstract-Conference.html.

Yu Wang, Yifan Gao, Xiusi Chen, Haoming Jiang, Shiyang Li, Jingfeng Yang, Qingyu Yin, Zheng Li, Xian Li, Bing Yin, Jingbo Shang, and Julian McAuley. MEMORYLLM: Towards self-updatable large language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50453–50466. PMLR, 2024. URL https://proceedings.mlr.press/v235/wang24s.html.

Haoran Wei, Yaofeng Sun, and Yukun Li. DeepSeek-OCR: Contexts optical compression. *arXiv preprint arXiv:2510.18234*, 2025. doi: 10.48550/arXiv.2510.18234. URL https://arxiv.org/abs/2510.18234.

Yuhuai Wu, Markus N. Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=TrjbxzRcnf-.

Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. InfLLM: Training-free long-context extrapolation for LLMs with an efficient context memory. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-3801. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html.

Zhiyuan Yu, Shijian Xiao, Zhangyue Yin, Xiaoran Liu, Lekai Xing, Wenzhong Li, Cam-Tu Nguyen, and Sanglu Lu. ProtoKV: Long-context knowledges are already well-organized before your query. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/e73ad1f690542144ce354637bb913c35-Abstract-Conference.html.

Peitian Zhang, Zheng Liu, Shitao Xiao, Ninglu Shao, Qiwei Ye, and Zhicheng Dou. Long context compression with activation beacon. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/fc797c61eb18f7b3ce9a74af0ef0d876-Abstract-Conference.html.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H2O: Heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/6ceefa7b15572587b78ecfcebb2827f8-Abstract.html.

Zhenyu Zhang, Shiwei Liu, Runjin Chen, Bhavya Kailkhura, Beidi Chen, and Zhangyang Wang. Q-Hitter: A better token oracle for efficient LLM inference via sparse-quantized KV cache. In *Proceedings of Machine Learning and Systems*, volume 6, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/hash/bbb7506579431a85861a05fff048d3e1-Abstract-Conference.html.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19724–19731, 2024. doi: 10.1609/aaai.v38i17.29946. URL https://ojs.aaai.org/index.php/AAAI/article/view/29946.

A Dataset Construction and Evaluation Splits

A.1 Dataset Construction and Splits

We build an English dataset from templates for project responsibilities, event records, handovers, and operational procedures. It contains 2,400 episodes, each with four memories, giving 9,600 memories in total. Each memory has five semantic question–answer (QA) pairs. Each pair asks about one fact or action and provides a reference answer. The same question can then be written in different ways. We count the pair once, regardless of its wording. This gives 48,000 semantic QA pairs across the dataset.

The dataset construction seed is 20260722. We assign 2,000 episodes to training, 200 to development, and 200 to testing (Table 3). All memories from one episode stay in the same split. The dataset report states that episode IDs, memory IDs, memory texts, and semantic signatures are kept separate across splits. The loading checks also look for duplicate memory IDs and text shared between training and testing. These checks cover overlap between records. The splits still use the same types of templates, vocabulary, and tasks. The test therefore evaluates new records drawn from this shared setting.

Table 3: Dataset splits. Each episode has four memories, and each memory has five semantic QA pairs. Different wordings of the same question are not counted separately here.

Split	Episodes	Memories	Semantic QA pairs
Training	2,000	8,000	40,000
Development	200	800	4,000
Test	200	800	4,000
Total	2,400	9,600	48,000

A.2 Memory Categories and Question Targets

The dataset has four memory categories. Atomic facts describe short relations between people, projects, locations, and responsibilities. Temporal events describe who did what, along with details such as the reason or result. Complex facts add effects, follow-up plans, and handover details. Procedural memories describe when to act, what to do, how to check the result, and how to recover from failure. They also include rules and follow-up steps. Table 4 gives the number and length of memories in each category. Length is measured in words, not language-model tokens. More detailed categories also have longer texts. Comparing categories therefore does not show the effect of text length alone.

Table 4: Category counts and text lengths for all 9,600 memories. In each evaluation subset, the four categories each account for 25% of the memories.

Category	Dataset share (%)	Memories	Word range	Mean words
Atomic facts	30	2,880	6–12	7.61
Temporal events	40	3,840	20–38	26.11
Complex facts	20	1,920	46–60	51.65
Procedural steps	10	960	105–121	111.51

Questions cover 13 field types: `actor`, `followup`, `impact`, `location`, `operation`, `owner`, `project`, `reason`, `region`, `result`, `rollback`, `trigger`, and `verification`. Each memory has questions about five relevant fields. A question asks for a specific fact or action. For example, a follow-up question asks what is planned next. A rollback question asks what to do if a check fails. We do not score timestamps as a separate field, so temporal-event accuracy is not a direct measure of reasoning about time. Procedural accuracy measures answers about individual fields, not whether the model can carry out the whole procedure.

A.3 Question Wordings and Input Content

Each semantic QA pair has four training wordings and one held-out wording. The original question is one of the four training wordings. Training uses these four forms. Evaluation uses two forms: the original question and the held-out wording. Both have the same reference answer. Each evaluated memory therefore produces $5 \times 2 = 10$ answers per configuration. The 40,000 training QA pairs provide 160,000 question–answer examples when all four training wordings are included. This is the number of available examples, not the number of times they are used during training.

For a test memory, the original question uses a training-style wording, but the memory belongs to the test split. The held-out wording is not used for training. We do use it in development evaluation and have examined its results during this work. Here, *held-out wording* means a question form excluded from training. It does not mean that the wording was never seen during development or base-model pretraining.

Some source memories contain exact values, such as change-ticket IDs and commands. Our questions do not test exact recall of these values. In the representation comparison, text and rendered-text baselines replace them with `[exact value omitted]`. Pixel and vector memories use the original source text. All methods share the memory IDs, questions, and reference answers. Their source texts can still differ because of this replacement. The examples below show the original memories, including their exact values.

Text baselines. In Batch I, head-and-tail selection takes text from both ends at character boundaries and checks its length with the model tokenizer. Question-aware selection scores sentences or clauses using a BM25-style rule with $k_1 = 1.2$ and $b = 0.75$. It selects segments that fit the budget and restores their original order. If the highest-ranked segment exceeds the budget, it uses head-and-tail truncation on that segment.

Rendered-text baselines. Batch I renders black text on a white background with DejaVuSans.ttf at 24 pixels. The canvas is 768 pixels wide, with a text width of 720 pixels, a left margin of 24 pixels, and a starting vertical position of 20 pixels. The line spacing is 34 pixels. For n lines, the canvas height is $48 + 34n$, rounded up to a multiple of 32. Budgeted images use LANCZOS resizing to target 16, 32, 64, or 128 merged visual positions while preserving the aspect ratio as closely as possible. The native processor uses `min_pixels`=32^2 and `max_pixels`=2048^2. The implementation checks visual-position counts against the image token IDs and `image_grid_thw`.

A.4 Evaluation Subsets and Sample Counts

Table 5 lists the evaluation subsets. We sample 100 test memories for Batch I and another 100 for Batch II. Each sample has 25 memories per category. All configurations within a batch use the same questions. Each produces 1,000 answers: 250 per category and 500 per wording. The two samples are drawn independently, but they may share some memories. We report them separately and do not combine their answers into a larger test set.

Table 5: Evaluation counts. Each memory has five semantic QA pairs, tested with two wordings. Counts are per configuration: per model in the cross-model comparison, and per slot budget and intervention in the branch analysis.

Evaluation	Memories	Semantic QA	Answers	Answers/category
Batch I	100	500	1,000	250
Batch II	100	500	1,000	250
Branch interventions	20	100	200	50
Cross-model comparison	100	500	1,000	250

The branch analysis uses 20 test memories, with equal numbers from each category. We test the frozen four- and sixteen-slot readers under five interventions. Each configuration produces 200 answers, compared with 1,000 in each main comparison. The cross-model analysis gives both language models the same 1,000 questions and exactly the same pixel files. Both systems use 16 memory slots. For development, we use a fixed subset of 100 memories with equal numbers from each category. This gives 1,000 answers per evaluated configuration.

Each question is answered from one selected memory. We do not test how to find that memory in a collection or combine information from several memories. Different wordings and model configurations reuse the same records. More generated answers therefore do not mean more independent memories. Each category has the same number of answers in an evaluation. Overall accuracy is therefore the simple average of the four category accuracies. It does not use the category proportions in the full dataset.

A.5 Memory and Question Examples

The examples below show the four memory categories. Each includes a source memory, two question wordings, and a reference answer for one of its five fields. We use them to show the task format. They contain no model predictions and were not selected to show successful answers.

Atomic fact: `memory_009529` / `owner`

Memory. Grace maintains Cedar's recommendation policy in Singapore.
Original question. Who is the primary operator for Cedar?
Held-out wording. Who is in charge of this project?
Reference answer. Grace

Temporal event: `memory_009218` / `reason`

Memory. After a customer maintenance window, Eve updated the deployment configuration for Ion in Tokyo on Thursday; the result was lower latency. Change ticket: CHG-009218.
Original question. Why was Ion's deployment configuration changed?
Held-out wording. What caused the change described in the memory?
Reference answer. a customer maintenance window

Complex fact: `memory_009212` / `followup`

Memory. On Sunday, Relay was affected by dependency instability in Seoul. Heidi responded by enabling the audit logging, and the result was partial success. During the response, core requests queued briefly. The team decided to observe the service for thirty minutes and recorded the conclusion in the handoff notes. Change ticket: CHG-009212.
Original question. What follow-up was planned for Relay?
Held-out wording. Name the follow-up step in the memory.
Reference answer. observe the service for thirty minutes

Procedural steps: `memory_009488` / `rollback`

Memory. The standard response for Vertex's access policy begins when capacity usage approaches its upper limit. Rupert should verify the alert in London, capture a snapshot, notify the duty engineer, and then change the access policy. Audit logging must remain active, and no operator may bypass the approval policy. When the action finishes, execute `opsctl verify --job 009488`, then compare error rate and latency before and after the change. If that check does not pass, restore the latest snapshot and repeat the health check. Document the result as failure, include the observation that false monitoring alerts decreased, and schedule the team to ask the on-call lead to review the change. Change ticket: CHG-009488.
Original question. What should happen if validation for Vertex fails?
Held-out wording. State the rollback procedure from the memory.
Reference answer. restore the latest snapshot and repeat the health check

B Pixel Encoding and Memory Slots

This appendix describes the stored pixel code, the reader, and the interface that supplies memory slots to the language model. It also specifies the input and output of the direct-vector baseline and the scope of the comparison.

B.1 Writing a Memory as Pixels

We encode each memory with the frozen `BAAI/bge-small-en-v1.5` text encoder. We take its CLS output, `last_hidden_state[:, 0]`, and apply L2 normalization to obtain a 384-dimensional vector z. The encoder receives the memory text without a retrieval-query instruction.

A fixed matrix $P \in \mathbb{R}^{384\times768}$ maps z to 768 values. Its rows are orthonormal, so $PP^\top = I_{384}$. The projection seed is 20260715. The clipping threshold $\alpha = 0.135$ is fitted on the training split and then fixed. We scale and clip the projected values, quantize them to integers from -127 to 127, and add 128 for unsigned 8-bit storage. The writing steps are:

1. $e \leftarrow \mathrm{CLS}(E(m)), \quad z \leftarrow e/\|e\|_2$.
2. $u \leftarrow zP$.
3. $v \leftarrow \mathrm{clip}(u/0.135, -1, 1)$.
4. $p \leftarrow \mathrm{uint8}(128 + \mathrm{round}(127v))$.
5. Reshape p to $3 \times 16 \times 16$ in channel–height–width order.
6. Reorder to height–width–channel format and save as an RGB PNG.

The resulting PNG stores one memory, with no added text, border, title, index, or resizing. The raw pixel payload occupies $3 \times 16 \times 16 = 768$ bytes. Each PNG in our evaluation occupies 852 bytes, including format overhead. This file size depends on the serialization settings. PNG preserves the quantized pixel values exactly. Appendix D.1 distinguishes payload size from serialized file size.

B.2 READING THE PIXEL IMAGE

The reader loads the RGB values in channel–height–width order and normalizes them as $x = (p - 128)/127$. It maps these values directly to memory slots, without multiplying by α or recovering the BGE vector. Table 6 gives the main sixteen-slot architecture. The reader width is $d_r = 384$, and the Qwen3-VL output width is $d_{\mathrm{LM}} = 4096$. We omit batch dimensions below.

Table 6: Main sixteen-slot pixel reader. Linear(a, b) maps a input values to b output values. FFN denotes a feed-forward network. The image contains 64 internal patches; the reader outputs 16 language-model memory slots.

Component	Setting
Input	$3 \times 16 \times 16$ pixels, normalized as $(p - 128)/127$
Shared global feature	LayerNorm(768), Linear(768,384), GELU, Linear(384,384)
Global slot features	Linear$(768, 16 \times 384)$ without bias; reshape to 16×384
Patch embedding	Conv2d with kernel size 2 and stride 2; 64 patches of width 384
Patch positions	Learned 64×384 position vectors
Patch Transformer	4 layers, 8 heads, FFN width 1,536, GELU, pre-norm, dropout 0
Learned queries	16×384 parameters
Reader blocks	2 blocks; cross-attention, self-attention, and FFN; 8 attention heads
Output	LayerNorm(384), Linear(384,4096)

From these normalized pixels, the global branch maps the flattened 768-value array to a shared vector $g \in \mathbb{R}^{384}$ and slot-specific features $S \in \mathbb{R}^{K \times 384}$. These mappings are named `raw_global` and `raw_slots` in the implementation. The local branch embeds 64 non-overlapping 2×2 patches, adds learned position vectors, and applies a Transformer to obtain $F \in \mathbb{R}^{64 \times 384}$.

To combine the global and local features into slots, the reader starts from learned queries. We add g to every query and add the corresponding row of S:

$$Q_0 = Q_{\mathrm{learned}} + \mathrm{Broadcast}(g) + S. \tag{6}$$

The learned queries are shared across memories; g and S depend on the current image. Each reader block applies cross-attention with the slots as queries and F as keys and values, followed by slot self-attention and an FFN. The blocks use residual connections. A final LayerNorm and linear layer produce $H_K \in \mathbb{R}^{K \times 4096}$. The reader receives no question, so a fixed image and reader can supply the same slots for different questions.

The slot count K also determines the number of learned queries and the output size of the global slot mapping. Each budget uses a reader with matching query and slot-map sizes and its own adapted parameters. Smaller-budget configurations are separately adapted readers, rather than outputs truncated from the sixteen-slot reader at inference. The stored 16×16 image stays the same. Appendix C describes the recorded training procedure for the main sixteen-slot Qwen3 system.

B.3 SUPPLYING SLOTS TO THE LANGUAGE MODEL

The main system uses `Qwen/Qwen3-VL-8B-Instruct`. Our pixel reader supplies continuous memory embeddings directly. The encoded memory image is processed by this reader; rendered-text image baselines use the model's native visual encoder.

In the main sixteen-slot implementation, the processor receives a gray 128×128 placeholder image to reserve 16 image positions. The placeholder carries no memory content and is separate from the stored 16×16 pixel image. The stored image is passed directly to the pixel reader without being resized to the placeholder dimensions.

During prefill, an embedding hook replaces the embeddings at the reserved positions with the rows of H_K. The replacement is applied only in this initial pass over the prompt. The implementation retains the input token IDs, position handling, and generation cache. The language model processes the injected slots together with the prompt and question, then generates the answer. It receives no source memory text on this route.

Each row of H_K occupies one language-model input position. The 768 stored pixel-channel values describe the persistent payload, the 64 patches are internal reader features, and K counts the memory positions supplied to the language model. The prompt and question occupy additional positions. Internal patches contribute to reader computation, and the placeholder dimensions do not change the stored memory size.

B.4 Direct-Vector Baseline

The direct-vector baseline stores the 384-dimensional BGE embedding of each memory in FP16. Its numerical payload occupies $384 \times 2 = 768$ bytes, before any container header or metadata. The same stored vector is reused at each evaluated slot budget. Appendix D.1 gives the storage accounting for each batch.

A trained vector reader maps the stored embedding to K continuous memory slots:

$$H_K^{\mathrm{vec}} = R_K^{\mathrm{vec}}(z_{\mathrm{FP16}}), \qquad H_K^{\mathrm{vec}} \in \mathbb{R}^{K \times d_{\mathrm{LM}}}. \tag{7}$$

Here, R_K^{vec} denotes the reader for budget K, and $d_{\mathrm{LM}} = 4096$ for Qwen3-VL-8B-Instruct. Each row of H_K^{vec} supplies one continuous embedding at a reserved memory position. The language model uses these embeddings together with the prompt and question to generate an answer.

Table 7: Pixel and direct-vector memory interfaces in Batch II. Payload sizes exclude file-format overhead. The output dimensions specify language-model inputs, not the capacity of the readers.

Property	Pixel memory	Direct FP16 vector
Stored values	768 quantized pixel-channel values	384 embedding coordinates
Stored precision	Unsigned 8-bit integers	FP16
Numerical payload	768 bytes	768 bytes
Reader input	$3 \times 16 \times 16$ normalized pixels	384-dimensional stored embedding
Slot budgets	$K \in \{4, 8, 16\}$	$K \in \{4, 8, 16\}$
Output shape	$K \times 4096$	$K \times 4096$
Adaptation	Reader and LoRA for each budget	Reader and LoRA for each budget

Batch II uses these interfaces to compare pixel and vector memories at $K \in \{4, 8, 16\}$. Each configuration uses its corresponding reader and LoRA adapter. The vector route reads the stored BGE embedding directly; the pixel route reads the projected and quantized image described in Appendix B.1. Table 7 summarizes their interfaces. At a given K, the two routes occupy the same number of memory input positions. This matches their language-model memory budgets, but does not establish matched reader capacity, initialization, or training histories. The comparison therefore evaluates each stored representation together with its adapted reader and language model; it does not isolate the effect of pixel encoding alone.

C Training Procedure

This appendix describes the recorded training procedure for the main sixteen-slot Qwen3 system. It covers the model settings, training stages, sampling, and checkpoint selection. Data construction and pixel processing are described in Appendices A and B.

C.1 Model and Trainable Parameters

The text encoder, projection, quantization rule, and base language-model weights remain fixed during QA training. We train the reader and LoRA parameters. Table 8 lists the main Qwen3 sixteen-slot settings. The reader and LoRA parameters are stored in FP32. The base model uses BF16, and training uses BF16 automatic mixed precision.

The main sixteen-slot reader has 18,610,048 parameters, and its LoRA adapter has 2,424,832 parameters. The four-, eight-, and sixteen-slot pixel systems use the same training data and auxiliary text QA, with separately adapted readers and LoRA parameters.

Table 8: Main Qwen3 sixteen-slot training configuration.

Item	Setting
Base model	`Qwen/Qwen3-VL-8B-Instruct`
Model revision	`0c351dd01ed87e9c1b53cbc748cba10e6187ff3b`
Base-model attention	Scaled dot-product attention (SDPA)
Trainable modules	Pixel reader and LoRA adapter
Reader parameters	18,610,048
LoRA location	Last 4 language-model layers
LoRA modules	Query, key, value, and output projections; gate, up, and down projections
LoRA settings	Rank 8, alpha 16, dropout 0, no bias
LoRA parameters	2,424,832
Training memory settings	Gradient checkpointing with `use_reentrant=False`; generation cache disabled during training

C.2 Initialization and Training Stages

We initialize the Qwen3 reader from a previously trained Qwen2.5-VL reader. We keep the compatible reader weights and replace the final output layer, changing its width from 3,584 to 4,096. We initialize a new Qwen3 LoRA adapter. Table 9 summarizes the four stages that follow. The final model inherits weights through the selected checkpoints. Some stages also restore earlier best weights and reset the optimizer.

Table 9: Training stages for the main sixteen-slot model. Update counts are local to each run. Selected update identifies the checkpoint retained from that run.

Stage	Run updates	Selected update	Main purpose
Transfer	519	519	Adapt the reader to Qwen3
Coverage	3,288	3,200	Train on the full training set
Extended training	9,723	8,875	Emphasize difficult fields, then reduce learning rates
Weighted training	8,966	8,966	Sample complex facts and procedures more often

Transfer. This stage uses 1,000 training memories. For the first 100 updates, we train only the new output layer. We then add the LoRA parameters. After 300 updates, we also train the rest of the reader. The output-layer, reader, and LoRA learning rates are 2×10^{-4}, 10^{-5}, and 10^{-5}, respectively. The run completes 519 updates within its training-time budget.

Coverage. We continue from the transfer checkpoint using all 8,000 training memories and their 40,000 semantic QA pairs. Pixel QA is mixed with auxiliary text QA. The first 200 updates focus on output-layer adaptation. The output-layer learning rate is initially 5×10^{-5} and later 2×10^{-5}. After the reader is unfrozen, its learning rate is 2×10^{-6}; the LoRA learning rate is 5×10^{-6}. We use AdamW with weight decay 0.01. Development evaluation occurs every 400 updates. The run completes 3,288 updates and retains update 3,200.

Extended training. We continue from the selected coverage checkpoint. This stage first covers the training data, then gives extra attention to difficult fields, and finally lowers the learning rates. The difficult fields are result, rollback, follow-up, verification, operation, and trigger. During the middle phase, at most one in four pixel QA examples is an extra difficult-field sample. The first two phases use peak learning rates of 10^{-5}, 10^{-6}, and 3×10^{-6} for the output layer, remaining reader, and LoRA. The last phase uses 5×10^{-6}, 5×10^{-7}, and 10^{-6}. Each phase uses 50 warm-up updates followed by cosine decay. At phase changes, we restore the current best weights and reset the optimizer. Development evaluation occurs every 30 minutes. The run completes 9,723 updates and retains update 8,875.

Weighted training. We start from that selected checkpoint, keep its reader and LoRA weights, and create a new AdamW optimizer. This final stage increases the sampling rates of complex facts

and procedural steps. It completes 8,966 additional updates. The sampling and optimization settings are given below. The transfer, coverage, and extended runs use seed 20260908; the final weighted run uses seed 20260909. GRPO and 32-slot runs form separate experiment branches.

C.3 Sampling and Optimization

Training uses the answer-token cross-entropy defined in the main text. Label -100 masks prompt, memory, question, and padding positions in the loss. Pixel QA updates both the reader and LoRA. Auxiliary text QA supplies the memory as text and updates only the shared LoRA. Text QA forms a separate training route.

From the coverage stage onward, the recorded runs use microbatches of two examples and accumulate gradients over eight microbatches. Every fourth microbatch uses text QA; the other microbatches use pixel QA. Each optimizer update therefore uses 12 pixel examples and four text examples. Each microbatch loss is divided by eight before gradient accumulation. The 3:1 ratio is measured by example count. The final weighted stage uses this QA objective alone.

In the final stage, the four categories are sampled at 15%, 15%, 35%, and 35%, in the order used in Appendix A. Every group of 20 samples contains 3 atomic facts, 3 temporal events, 7 complex facts, and 7 procedures. Pixel and text routes use separate queues with the same proportions. Within each category, we shuffle the QA list and traverse it before reshuffling. Gradient updates use training-set QA pairs with the four training wordings in rotation.

Table 10: Final weighted training stage. Learning rates are peak values. The schedule follows active training time.

Setting	Value
Optimizer	AdamW; weight decay 0.01
Microbatch / accumulation	2 examples / 8 microbatches
Examples per update	12 pixel QA + 4 text QA
Output-layer learning rate	10^{-5}
Remaining-reader learning rate	10^{-6}
LoRA learning rate	3×10^{-6}
Warm-up and decay	100 updates; cosine decay to 20% of peak
Gradient norm limit	1.0
Random seed	20260909
Checkpoint saving	Every 100 updates and after full validation
Development evaluation	Every 45 minutes
Run budget	480 active minutes, including 60 reserved for final evaluation
Update limit	20,000; actual updates: 8,966

The final stage visits 107,592 pixel QA examples and 35,864 text QA examples. These counts include repeated visits to existing examples. The pixel route covers all 8,000 training memories. The run stops at its time budget.

C.4 Development Evaluation and Checkpoint Selection

We use the fixed development subset described in Appendix A. The recorded development runs use a generation batch size of 4 and a maximum of 48 new tokens. Extended training selects checkpoints using held-out-wording exact match and token F1, subject to a limit on the drop in atomic-fact performance.

For the final stage, each candidate requires completed development judgments. Its overall semantic accuracy must be at least as high as that of the starting checkpoint. The allowed drop in atomic-fact accuracy is at most one percentage point. Among eligible candidates, we first rank held-out-wording accuracy and then overall accuracy. The starting checkpoint is also a candidate. This procedure selects update 8,966 from the final stage.

Held-out wordings support development selection. The later answer recheck updates only the scoring records.

C.5 Configurations Across Slot Budgets

Batch II evaluates pixel and FP16-vector memories at 4, 8, and 16 slots. Each representation and slot budget uses separately adapted reader and LoRA parameters. Across slot budgets, the training data and auxiliary text QA examples remain the same.

Table 11 summarizes the six configurations. The output shape specifies the memory embeddings supplied to Qwen3-VL-8B-Instruct.

Table 11: Reader configurations in Batch II. Each reader has a corresponding LoRA adapter. All outputs have language-model width 4,096.

Slots	Pixel reader	Vector reader	Output shape
4	R_4^{pix}	R_4^{vec}	4×4096
8	R_8^{pix}	R_8^{vec}	8×4096
16	R_{16}^{pix}	R_{16}^{vec}	16×4096

For the pixel reader, changing K changes the number of learned queries and the size of the global slot mapping. With reader width $d_r = 384$, the learned query matrix has shape $K \times 384$. The global slot mapping produces $K \times 384$ values before the reader blocks and final output projection. Each configuration therefore has a reader sized for its slot budget.

Auxiliary text QA updates the LoRA parameters shared with the pixel route. Pixel QA updates both the pixel reader and its LoRA adapter. The main sixteen-slot Qwen3 training schedule is described in Appendices C.2 and C.3.

The budget comparison evaluates each adapted configuration at its assigned slot count. Its results reflect the slot budget together with the reader and LoRA parameters learned for that configuration. Appendix E reports the complete Batch II results.

D Storage and Context Accounting

We measure storage per unique memory and input positions per QA request. Shared model resources are recorded at the deployment level. This appendix defines these units and the reduction ratios used in the main text.

D.1 Storage per Memory

File size is the number of bytes in the saved memory file, including its format overhead. Raw payload size counts the stored values at their chosen precision. A pixel memory has a payload of $3 \times 16 \times 16 = 768$ bytes, and each PNG in our evaluation occupies 852 bytes. A 384-dimensional FP16 vector has a payload of $384 \times 2 = 768$ bytes. Appendix B describes the pixel encoding.

Table 12 summarizes the reported costs by batch. Text selection stores the full memory and chooses a shorter input at reading time. Its file size therefore matches the full-text route. The Batch II summary records 768 bytes per FP16 vector. The earlier Batch I record uses an FP16 NPY file of 896 bytes, including its header.

D.2 Memory Positions and Reduction Ratios

Memory input positions count the memory content supplied to the language model for one QA request. For text, we tokenize the actual input segment with `add_special_tokens=False`. For rendered text, we count the visual positions supplied by the native encoder. For pixel and vector memories, we count the K injected slots. Questions, prompts, boundary markers, padding, and generated answers are accounted for separately. The reader's 64 internal patches contribute to reader computation.

Table 12: Reported storage and memory input positions. Values are batch means, except for the listed slot budgets. Text caps give the maximum allowed input length.

Batch	Representation	Bytes/memory	Positions/QA
I	Full text	326.66	62.13
I	Selected text, cap 16	326.66	12.97
I	Selected text, cap 64	326.66	41.31
I	Rendered text, 16 positions	14,162.66	16.00
I	Readable rendered text	28,563.08	197.52
I	Pixel memory	852.00	16
II	Pixel memory	852.00	4 / 8 / 16
II	FP16 vector	768.00	4 / 8 / 16

For N aligned QA requests, let K_i be the pixel-memory position count and T_i the full-text memory position count for request i. The position reduction is

$$r_{\text{positions}} = 1 - \frac{\sum_{i=1}^{N} K_i}{\sum_{i=1}^{N} T_i}. \tag{8}$$

For M unique memories in the same batch, let B_m^{pixel} and $B_m^{\text{reference}}$ be the file sizes of memory m under the pixel and reference representations. The storage reduction is

$$r_{\text{storage}} = 1 - \frac{\sum_{m=1}^{M} B_m^{\text{pixel}}}{\sum_{m=1}^{M} B_m^{\text{reference}}}. \tag{9}$$

We aggregate costs across samples before computing each ratio. Multiplying by 100% gives the reported percentage; a negative value indicates an increase in cost.

Using the rounded Batch I means, the position reduction relative to full text is $1 - 16/62.13 \approx 74.25\%$. The file-size reduction relative to rendered text at 16 positions is $1 - 852/14{,}162.66 \approx 93.98\%$. Each comparison uses its stated reference representation.

E Complete Results for Batch I and Batch II

E.1 Batch I: Representation Comparison

Batch I compares full text, selected text, rendered text images, and pixel memories. Table 13 reports overall and category accuracy for the main representations. Table 14 adds head-and-tail selection and further text and image budgets.

Table 13: Batch I: main representations. Accuracy is reported in percent, with 1,000 answers per configuration and 250 per category. Positions are the mean memory input length.

Representation	Positions	Atomic	Event	Complex	Procedure	Overall
Full text	62.13	98.8	92.0	91.6	87.6	92.5
Selected text, cap 16	12.97	98.8	38.4	27.2	5.2	42.4
Selected text, cap 64	41.31	98.8	92.0	88.4	62.4	85.4
Rendered text, 16 positions	16.00	94.8	60.8	13.6	2.0	42.8
Readable rendered text	197.52	98.4	93.2	94.0	89.6	93.8
PixelMem, 16 slots	16.00	98.8	90.4	86.8	81.6	89.4

E.2 Batch II: Memory Budget Comparison

Batch II compares PixelMem and FP16 vector memories at 4, 8, and 16 slots. Table 15 includes budgeted text and full text as references, and reports accuracy across the four memory categories.

Table 14: Batch I: additional baselines, with 1,000 answers per configuration. Budget is a text-token cap or a fixed visual-position count. Positions are the actual mean memory input length.

Configuration	Budget	Positions	Accuracy (%)
Head-and-tail text	16	14.60	29.3
Head-and-tail text	32	26.55	55.0
Head-and-tail text	64	43.00	83.9
Question-aware selected text	32	24.52	70.1
Rendered text	32	32.00	62.6
Rendered text	64	64.00	82.9
Rendered text	128	128.00	91.2

Table 15: Batch II: all ten configurations. Accuracy is reported in percent, with 1,000 answers per configuration and 250 per category. Budget gives the slot count for learned routes and the token cap for budgeted text. Full text uses the complete memory.

Representation	Budget	Atomic	Event	Complex	Procedure	Overall
FP16 vector	4	91.6	46.4	41.6	14.4	48.5
PixelMem	4	95.6	80.4	70.4	58.4	76.2
FP16 vector	8	98.4	83.6	57.6	43.6	70.8
PixelMem	8	98.4	87.6	80.4	74.4	85.2
FP16 vector	16	99.6	82.4	70.4	56.4	77.2
PixelMem	16	98.4	90.8	87.2	81.2	89.4
Budgeted text	4	7.6	2.4	0.4	0.4	2.7
Budgeted text	8	67.6	18.4	14.4	0.4	25.2
Budgeted text	16	96.4	41.6	21.6	7.6	41.8
Full text	Full	95.6	94.4	90.4	92.4	93.2

F Branch Ablation and Memory Mismatch

Section 5.2 examines how the trained reader uses its global and local branches. Removing a branch changes both the available memory information and the computation performed by the reader. We therefore also replace one branch's input with another memory while retaining the full computation. These two interventions test the reader's dependence on a processing path and its sensitivity to the memory content supplied through that path.

F.1 Evaluation Setup and Branch Interventions

We evaluate the four- and sixteen-slot checkpoints with all reader, LoRA, and base-model weights frozen. The evaluation uses 20 test memories, with five from each of the four categories. Each memory has five semantic question–answer pairs, evaluated with the original and held-out wordings. This gives 200 answers per configuration. Within each slot budget, the five intervention conditions use the same target questions and checkpoint. No parameters are updated during these evaluations.

Let A denote the target memory and B a donor memory. The question always concerns A. As described in Appendix B.2, the global branch produces a shared feature g and slot-specific features S, while the local branch produces patch features F. Table 16 specifies which memory supplies each branch and which operations are retained.

In Full, each initial slot combines its learned query with g and the corresponding row of S. Patch-only retains the learned query but removes both memory-dependent global contributions. Wrong-global obtains both g and S from B while retaining A's patch features. Wrong-patch obtains F from B while retaining A's global features. Thus, the mismatch conditions change the source of memory content without removing reader operations or changing the output slot count.

Table 16: Frozen-reader interventions. A denotes the target memory and B a same-category donor with a different reference answer for the evaluated field. A dash indicates a removed branch. All conditions supply K memory slots to the language model.

Condition	Global	Patch	Reader operation
Full	A	A	Use both branches with target memory content.
Global-only	A	—	Remove patch processing and patch cross-attention; retain slot self-attention, FFN, and output layers.
Patch-only	—	A	Remove both global mappings; retain learned queries and target patch features.
Wrong-patch	A	B	Retain the full computation and replace only the patch-branch input.
Wrong-global	B	A	Retain the full computation and replace the input to both global mappings.

F.2 Donor Memories and Evaluation Measures

For each evaluated field, the donor belongs to the same memory category as the target and has a different reference answer for that field. The donor supplies an alternative memory input to one branch; the other branch continues to receive the target memory. For example, when the question asks for the responsible person, the target and donor references identify different people. The model is still asked the target question, and donor matching is evaluated against the corresponding field in B.

The frozen model generates an answer from the supplied slots and the target question. We evaluate the answer against the target memory and reference answer. For each mismatch condition, we also evaluate that same output against the donor's reference answer for the corresponding field. Let $T_i = 1$ when output i correctly answers the target question, and let $D_i = 1$ when it matches the donor field answer; otherwise, the respective indicator is zero. With $N = 200$ answers per configuration, we report

$$\text{Target} = \frac{100}{N}\sum_{i=1}^{N} T_i, \qquad \text{Donor} = \frac{100}{N}\sum_{i=1}^{N} D_i. \tag{10}$$

Target accuracy is evaluated for all five conditions. Donor-field match rate is reported for Wrong-patch and Wrong-global, using all evaluated answers as the denominator. It is not conditioned on the target answer being incorrect. The two measures evaluate different properties of the same output and should not be treated as complementary percentages. A donor match measures agreement with B's field answer while answering A's question; it is not a separate QA evaluation with a question about B.