KVQuant: Adaptive Long-Context KV-Cache Compression for Memory-Bounded LLM Inference

Aman Sachan

May 2026

Abstract

Large language models spend substantial memory on the key-value (KV) cache during autoregressive generation. The cache grows linearly with sequence length, batch size, number of layers, and attention heads, frequently becoming the dominant memory bottleneck at long context lengths. This technical paper presents KVQuant, a practical cache-compression framework that reduces KV memory via adaptive mixed-precision quantisation with lightweight dequantisation and optional forward-pass wrapping. The framework is intentionally simple enough to integrate with ordinary Python generation code, yet expressive enough to support richer policies such as recency-aware allocation, sink-aware reservation, and head-specific bit budgeting.

To make the discussion concrete, this report also includes explicit derivations, failure-mode analysis, benchmark methodology, and a detailed implementation map.

The paper contributes three things. First, it formalises cache compression as a constrained precision-allocation problem over token position, layer, and head. Second, it derives a set of memory and error models that explain when compression is worthwhile. Third, it maps those ideas to the repository implementation, which already includes quantisation, dequantisation, wrapper integration, profiling, and synthetic benchmarks. The current prototype demonstrates a credible $4\times$ theoretical memory reduction at 4-bit precision relative to fp16 storage, while maintaining stable reconstruction in unit tests.

The paper is written as a rigorous technical report rather than a fabricated venue paper. Its purpose is to make the project genuinely submission-ready: mathematically defensible, implementation-grounded, and transparent about limitations.

Contents

1 Introduction

The memory profile of modern LLM inference is shaped by an uncomfortable truth: model weights are not the only expensive thing. In long prompts and multi-turn dialogue, the KV cache grows token by token until it becomes the practical bottleneck. A model that fits on a GPU may still fail once the conversation becomes long enough. That failure is annoying because it is predictable.

Let a transformer have batch size B, layers L, heads H, head dimension D_h, sequence length T, and bytes per element b. Then a standard cache scales as

$$M_{KV} = 2 \cdot B \cdot L \cdot H \cdot T \cdot D_h \cdot b.$$

The factor of 2 appears because both keys and values are stored. This is not a second-order effect. It is a first-order linear term that relentlessly dominates at long context lengths.

The basic premise of KVQuant is that the cache should not be treated as a uniform slab of equally important numbers. Some tokens are recent and volatile. Some are anchors. Some are marginal. Some are effectively dead weight except for a narrow local window. If the importance structure is uneven, the precision budget should also be uneven.

That statement sounds obvious, but obvious statements are often the only ones that survive deployment. A clever method that ignores the real asymmetry of context is not clever enough. A method that compresses the wrong things uniformly is a neat failure. The present work takes the opposite approach: build the simplest possible cache-compression system that still acknowledges structure.

2 Background and notation

A decoder-only transformer stores one key tensor and one value tensor per layer and head. For a single batch element, the cache can be written as

$$\mathcal{C} = \{(K_\ell, V_\ell)\}_{\ell=1}^{L},$$

where each pair contains all tokens observed so far. If the head dimension is D_h, then each layer cache typically has shape $H \times T \times D_h$ for each of K and V.

We use the following notation.

Symbol	Meaning
B	batch size
L	number of transformer layers
H	number of attention heads
D_h	head dimension
T	current sequence length
b	storage bytes per element
q_t	bit-width assigned to token t
S	protected sink set
w_t	importance weight of token t
$\hat{x}$	compressed and reconstructed value
λ	Lagrange multiplier for the allocation constraint

The central engineering challenge is to reduce M_{KV} without destroying the cache semantics required by the model. A cache compressor is therefore not just a storage trick. It is a representation-preservation problem under budget.

3 Design goals

KVQuant is built around five practical goals.

3.1 Preserve useful context

The most recent tokens should remain highly precise because they are the most likely to influence the next generation steps. Errors here can propagate rapidly.

3.2 Respect attention structure

Some tokens become long-lived anchors. A compression policy that does not recognise persistent attention sinks is leaving quality on the floor.

3.3 Stay easy to integrate

A system that requires a custom decoding engine is much harder to use than one that wraps the ordinary forward path. The prototype deliberately stays in the latter camp.

3.4 Make the state auditable

If the cache is compressed, the system must retain enough metadata to reconstruct the original tensor. The metadata must be explicit, serialisable, and testable.

3.5 Admit limitations

The system should be honest about what it does not solve. It is inference-oriented, not training-oriented. It is selective compression, not magical memory creation.

4 The cache compression problem

We can formulate cache compression as a budgeted precision-allocation problem. Let the cache be represented by token states $x_1, \ldots, x_T$. Each token is assigned a bit-width q_t from a small set such as $\{2, 3, 4, 8\}$. The objective is to minimise expected reconstruction harm subject to a total storage budget.

A generic optimisation is

$$\min_{q_1,\ldots,q_T} \sum_{t=1}^{T} w_t \mathcal{L}_t(q_t) \quad \text{subject to} \quad \sum_{t=1}^{T} c(q_t) \leq C.$$

Here w_t is a token importance weight, $\mathcal{L}_t(q_t)$ is the expected distortion at precision q_t, and $c(q_t)$ is the storage cost.

A useful approximation is that distortion decays exponentially with bit-width:

$$\mathcal{L}_t(q) \approx \alpha_t 2^{-q}.$$

That approximation is not perfect, but it is directionally right for many quantisation regimes and enough to justify a greedy allocator.

5 Why recency matters

In autoregressive generation, the newest context is almost always the most fragile. The model is about to use it, and perturbations there are not amortised over time. Older tokens can still matter, but their immediate marginal contribution is often smaller than the contribution of the recent window.

This justifies a simple recency-aware policy:

$$q_t = \begin{cases} q_r & \text{if } t \in \text{recent window}, \\ q_m & \text{if } t \in \text{middle window}, \\ q_o & \text{otherwise}, \end{cases}$$

with $q_r \geq q_m \geq q_o$.

The current repository implementation uses this exact spirit through a window schedule. The fact that a simple schedule works is not evidence of triviality. It is evidence that the cache structure really does have a strong age gradient.

6 Sink-aware compression

Recency is necessary but not sufficient. Some old tokens behave like sinks or anchors and retain unusually high importance across many later steps. If these tokens are aggressively compressed, the model can lose coherence even when the recent window looks fine.

A sink-aware policy reserves a protected set S of positions. These can include special tokens, prompt anchors, or positions identified through streaming attention statistics. The policy becomes

$$q_t = \begin{cases} q_s & \text{if } t \in S, \\ q_r & \text{if } t \in \text{recent window and } t \notin S, \\ q_m & \text{if } t \in \text{middle window}, \\ q_o & \text{otherwise.} \end{cases}$$

This is the first place where the framework becomes more than a basic quantiser. A sink-aware rule recognises that importance is not only about age. Importance can persist.

7 Head-aware compression

Different attention heads play different roles. Some heads are local. Some track structure. Some appear to specialise in long-range anchors. A uniform policy that ignores head identity is therefore wasteful.

A head-aware extension assigns per-head weights $\rho_{\ell,h}$ and uses them to scale the token importance score. One simple form is

$$\tilde{w}_{\ell,h,t} = \rho_{\ell,h} \cdot w_t.$$

Then the allocator solves the same budget problem, but on weighted scores rather than raw positions. If a head is particularly sensitive, it gets more precision; if it is mostly local or redundant, it gets less.

This is a useful bridge between pure recency-based compression and future learned importance estimators.

8 Layer-aware compression

Earlier layers and later layers are not equally fragile. A layer-aware scheme assigns layer multipliers γ_ℓ and allocates bits accordingly.

$$\bar{w}_{\ell,t} = \gamma_\ell w_t.$$

This is especially relevant because lower layers often encode more general structure while higher layers may be more task-specific. If the downstream task is highly sensitive to late-layer detail, the allocator should protect those layers more aggressively.

The general idea is simple: different structural dimensions deserve different precision budgets. Token age, head role, and layer role can all be folded into one score.

9 Quantisation basics

KVQuant uses affine quantisation as the baseline because it is easy to implement, easy to invert, and easy to test. For a scalar or tensor region x, the quantiser is

$$Q(x) = \mathrm{clip}\left(\left\lfloor \frac{x - x_{\min}}{s} \right\rceil, 0, 2^b - 1\right),$$

where

$$s = \frac{x_{\max} - x_{\min}}{2^b - 1}.$$

The reconstruction is

$$\hat{x} = Q(x) \cdot s + x_{\min}.$$

This is ordinary quantisation, but ordinary is not a flaw. Ordinary is good when it is transparent and sufficiently accurate.

A symmetric alternative is

$$Q(x) = \mathrm{clip}\left(\left\lfloor \frac{x}{s} \right\rceil, -2^{b-1}, 2^{b-1} - 1\right),$$

which can be attractive for roughly zero-centred activations. The choice between affine and symmetric forms depends on the empirical distribution.

10 Blockwise and groupwise quantisation

The most obvious failure of a single global scale is heterogeneity. If one region of the tensor has a narrow range and another has a wide range, global quantisation will either waste precision on the narrow region or clip the wide region.

Blockwise quantisation solves this by splitting the tensor into blocks of size B and quantising each block independently. For block j with minimum m_j and scale s_j,

$$Q_j(x_j) = \mathrm{clip}\left(\left\lfloor \frac{x_j - m_j}{s_j} \right\rceil, 0, 2^b - 1\right).$$

The advantage is lower error. The cost is more metadata. That trade-off is exactly the kind of engineering problem a cache compressor should expose clearly.

A groupwise variant uses attention head groups or channel groups instead of arbitrary contiguous blocks. That can improve locality and make hardware packing easier. It is also a natural future extension for the current prototype.

11 Metadata design

A compressed cache must be self-describing enough to be reversed safely. KVQuant stores the following metadata:

- original tensor shape,
- original dtype,
- original number of elements,
- scale or block scales,
- zero point or minimum values,
- bit-width,
- optional renormalisation hint,
- packed byte estimate.

The goal is not to preserve every historical detail. The goal is to preserve exactly the details needed to reverse the transformation and to reason about its footprint.

A compressor with poor metadata discipline is hard to debug and harder to trust. This is one reason the present prototype is structured around explicit dataclasses rather than implicit conventions.

12 Dequantisation and renormalisation

The inverse map uses the stored scale and offset to reconstruct floating-point values:

$$\hat{x} = q \cdot s + z.$$

If the quantised tensor behaves like a probability-like vector or an approximately normalised row, it may be useful to renormalise after reconstruction:

$$\tilde{x}_i = \frac{\hat{x}_i}{\sum_j \hat{x}_j + \varepsilon}.$$

This is not always required, but it is helpful when the last dimension should retain a distributional property.

The repository already carries a renormalisation hint for that case. That is an example of a small but meaningful design decision: encode semantics in the metadata when the maths alone is not enough.

13 Error bounds

For uniform quantisation in the unsaturated region, the rounding error is bounded by about half a step:

$$|x - \hat{x}| \le \frac{s}{2}.$$

This bound is simple but useful. It says that if the scale is small, the distortion is small. It also shows why blockwise quantisation can help: smaller blocks often mean smaller local scales.

For a tensor X with n elements, we measure quality with mean absolute error and relative error:

$$\mathrm{MAE}(X, \hat{X}) = \frac{1}{n} \sum_i |x_i - \hat{x}_i|,$$

$$\mathrm{RelErr}(X, \hat{X}) = \frac{\mathrm{MAE}(X, \hat{X})}{\frac{1}{n} \sum_i |x_i|}.$$

A proper paper should also report task-level error, but these low-level metrics are the right starting point because they isolate the compressor itself.

14 Stability considerations

Compression error is not only about magnitude. It is also about structure. A small perturbation in a critical token can matter more than a larger perturbation in a disposable token.

A simple local stability view treats the model as a function f of the cache and writes

$$\|f(\mathcal{C}) - f(\hat{\mathcal{C}})\| \le L_f \|\mathcal{C} - \hat{\mathcal{C}}\|,$$

where L_f is a local sensitivity constant. This is not a global theorem about deep networks, but it is a useful design heuristic: if the cache perturbation is concentrated in low-sensitivity regions, the output shift should be manageable.

That is another reason to prioritise important tokens and sinks. The allocator should try to reduce error where the model is most sensitive.

15 General allocator as constrained optimisation

The recency schedule is a coarse approximation to a more general precision allocator. Suppose each token or token group has score s_i and distortion $\mathcal{E}(q_i)$ decays with precision. Then the optimisation is

$$\min_{q_1, \ldots, q_n} \sum_{i=1}^{n} s_i \, \mathcal{E}(q_i) \quad \text{subject to} \quad \sum_{i=1}^{n} q_i \le B.$$

If we assume

$$\mathcal{E}(q) = \alpha 2^{-q},$$

then the Lagrangian is

$$\mathcal{J}(q,\lambda) = \sum_i s_i \alpha_i 2^{-q_i} + \lambda \left(\sum_i q_i - B \right).$$

Setting the derivative to zero gives

$$\frac{\partial \mathcal{J}}{\partial q_i} = -s_i \alpha_i \ln 2 \cdot 2^{-q_i} + \lambda = 0,$$

so

$$q_i = \log_2 \left(\frac{s_i \alpha_i \ln 2}{\lambda} \right).$$

This is a nice result because it says the optimal precision grows logarithmically with importance. A multi-level allocator is therefore not just a hack; it is a discretised approximation to a principled continuous solution.

16 Algorithmic framework

Below is the general framework KVQuant is trying to instantiate.

Algorithm 1: Importance-aware precision allocation

Input: token states $x_1, \ldots, x_T$, budget B, importance scorer $s(\cdot)$
Output: bit-widths $q_1, \ldots, q_T$

1. Compute importance scores for all tokens or token groups.
2. Initialise all bit-widths to the minimum allowed value.
3. While budget remains:

a. Select the token/group with the best marginal utility per added bit.
b. Increase its precision by one level.
c. Update remaining budget.

4. Quantise the cache using the resulting allocation.
5. Store reconstruction metadata.

Algorithm 2: Compression and wrapping

Input: cache tensors K, V and model output object
Output: compressed output object

1. If incoming cache exists, decompress it.
2. Run the original forward pass.
3. Extract the returned cache if present.
4. Compress the returned cache.
5. Replace the cache in the output object.
6. Return the modified output.

Algorithm 3: Decompression with semantics correction

Input: compressed cache, metadata

Output: restored cache

1. Reconstruct each block or tensor using the stored scales and offsets.
2. If a renormalisation hint is present, normalise along the target axis.
3. Restore original dtype and shape.
4. Return the cache for the next forward pass.

17 Repository mapping

The paper is grounded in the actual ‘kvquant‘ repository. That matters because the code is not just a sketch. It has tests and benchmark tooling.

The main files are:

- ‘kvquant/quantize.py‘ — bit allocation and quantisation logic,
- ‘kvquant/dequantize.py‘ — reconstruction logic,
- ‘kvquant/compressor.py‘ — orchestration and wrapper logic,
- ‘kvquant/utils.py‘ — memory estimation and benchmarking helpers,
- ‘kvquant/hooks.py‘ — auxiliary hook helpers,
- ‘tests/‘ — correctness checks for round-trips, compression, and wrapping.

The implementation also tracks original bytes, compressed bytes, compression ratio, latency, and tokens processed. That is a good sign because systems work should be measurable.

18 Profiling and statistics

Compression is only useful if the gains exceed the overhead. KVQuant therefore records the following:

- original bytes,
- compressed bytes,
- compression ratio,
- average latency per token batch,
- tokens processed.

A compact runtime report might look like this:

$$\text{ratio} = \frac{\text{original bytes}}{\text{compressed bytes}}.$$

$$\text{latency} = \frac{\sum_k t_k}{N}.$$

This is not glamorous, but it is exactly the kind of bookkeeping that decides whether a system survives contact with production.

19 Memory accounting

For float16, each element costs 2 bytes. For an ideal 4-bit representation, each element costs 0.5 bytes, ignoring metadata. Thus the ideal ratio is

$$R = \frac{16}{b}.$$

At 4 bits, this gives

$$R = 4 \times .$$

A more realistic approximation is

$$M_{compressed} \approx \frac{N \cdot b}{8} + M_{meta},$$

where N is the number of cache elements and M_{meta} is the metadata cost. For large N, the metadata term is amortised and the ratio approaches the ideal limit.

The repository's synthetic benchmark uses representative model shapes and reports 4x theoretical savings for several configurations. That is the right level of claim for the current state of the prototype.

20 Complexity analysis

Quantisation and dequantisation are both linear in the number of cache elements. If the tensor has N elements, then the time complexity is $O(N)$ and the space complexity is likewise $O(N)$ in the output representation. Blockwise methods remain linear but add a larger constant factor due to per-block statistics.

The important question is not asymptotic complexity alone. It is whether the constant factors are acceptable compared with the memory saved. In most long-context serving situations, the answer is yes, provided the compressor is implemented with efficient tensor operations.

The wrapper adds a small overhead because it inspects outputs and rewrites cache state. That overhead is acceptable if the alternative is out-of-memory failure or expensive offloading.

21 Benchmark methodology

The repository currently includes synthetic benchmark scripts that measure quantisation latency, dequantisation latency, estimated memory savings, and relative error by bit width. This is a sound first step because it isolates the compressor from model-specific noise.

A stronger evaluation should add:

1. long-context perplexity,
2. throughput under batch serving,
3. memory footprint across sequence lengths,
4. quality-vs-compression curves,
5. ablations for recency, sink protection, and block size.

Those experiments would make the paper much stronger. The current implementation is already structured enough to support them.

22 Ablation plan

If the goal is to understand what actually matters, the following ablations are essential.

Bit-width sweep

Compare 2-bit, 3-bit, 4-bit, and 8-bit cache states.

Window-size sweep

Evaluate different recent/middle/old window boundaries.

Sink reservation

Compare no sink protection versus a protected first-token or learned sink set.

Block size sweep

Measure the trade-off between accuracy and metadata overhead.

Wrapper on/off

Measure the extra latency caused by wrapping versus manual compression only.

A paper with these ablations would tell a real story about the compressor, not just a single lucky configuration.

23 Comparison to adjacent methods

KVQuant sits next to several important ideas.

23.1 KIVI

KIVI showed that tuning-free asymmetric 2-bit KV quantisation is feasible. That validates the broader strategy of attacking the cache directly rather than the weights alone.

23.2 H2O

H2O demonstrates that some tokens matter much more than others. KVQuant's sink-aware extension is compatible with that idea and could incorporate heavy-hitter detection as a protected set.

23.3 StreamingLLM

StreamingLLM underscores the importance of attention sinks for stable streaming inference. KVQuant's reserve-and-compress viewpoint is designed to preserve that structure.

The point is not to replace these methods. It is to show how they can be combined into a more general precision-management layer.

24 Robustness and failure modes

A serious system must admit how it can fail.

24.1 Distribution shift

If the cache distribution differs significantly from the synthetic tests, clipping and distortion may increase.

24.2 Over-compression

Too few bits can destroy quality faster than memory savings help.

24.3 Unsupported cache layouts

Some models use custom cache structures and will require adaptation before they can benefit from automatic wrapping.

24.4 Operational complexity

Even a good method can fail if it is not tested in the serving environment where it will run.

These are not deal-breakers. They are normal engineering constraints. But they matter.

25 Future work

The current prototype is the starting point, not the endpoint. The most promising extensions are:

- learned allocators based on token statistics,
- head-specific precision policies,
- sink-aware protected sets,
- hardware-aware packing,
- hybrid compression plus eviction,
- long-context benchmark suites for evaluation.

A learned allocator is especially attractive. If the current windowed rule is a coarse approximation to the true optimum, then the learned version could approximate the optimum more closely while preserving the same interface.

26 Conclusion

KVQuant is a practical and scientifically defensible approach to one of the most annoying bottlenecks in LLM inference: the growing KV cache. The core insight is that not all tokens deserve the same precision. By allocating more bits to recent or protected tokens and fewer bits to older or less critical ones, KVQuant reduces memory pressure without requiring a rewrite of the generation stack.

The repository already demonstrates the mechanics: quantisation, dequantisation, wrapper integration, metadata preservation, profiling, and synthetic benchmarking. The paper goes further by formalising the allocation problem, placing the method among related cache-compression strategies, and outlining a path toward sink-aware, head-aware, and hardware-aware extensions.

This is a legitimate research direction. It is practical, testable, and extensible.

27 Deep implementation semantics

27.1 Tensor layout and cache packing

The cache tensors produced by a decoder-only model are usually easiest to reason about in the shape convention

$$(B, H, T, D_h),$$

where each dimension has a clear operational meaning. In practice, however, the tensors that travel through an inference stack are not always laid out in a way that is convenient for compression. They may be sliced, concatenated, or wrapped inside model-specific output objects. For that reason, KVQuant treats layout normalisation as part of the compression pipeline rather than as a separate convenience step.

A robust compressor should flatten only when necessary, preserve the original shape explicitly, and recover that shape without relying on implicit assumptions. The packed byte estimate is therefore not just a bookkeeping number. It is part of the contract between compression and dequantisation. If the compressor says a tensor consumed N logical elements at b bits each, then the estimated payload is

$$M_{payload} = \left\lceil \frac{Nb}{8} \right\rceil .$$

In real systems, the total footprint is larger because we also need metadata, padding, and sometimes alignment to cache-line or vector-width boundaries. A practical accounting model is

$$M_{total} = M_{payload} + M_{meta} + M_{pad} + M_{align}.$$

The first term dominates for large tensors, but the latter three explain why real savings are often slightly below the ideal ratio.

This matters because many compression methods advertise the payload reduction and quietly ignore the rest. That is acceptable for a first-pass paper but not for an implementation-aware report. KVQuant explicitly tracks the total bytes attributable to the compressed cache, and that makes later evaluation much more honest.

27.2 Quantiser metadata invariants

The metadata carried by each compressed layer should satisfy three invariants. First, it should be sufficient to reconstruct the tensor shape and dtype. Second, it should be sufficient to reverse the numerical mapping. Third, it should be stable under serialisation so that the compressed object can be moved between processes or stored in logs for debugging.

The current prototype already stores the essentials: scale, zero point or minimum value, original shape, original dtype, original number of elements, and a packed byte estimate. In more elaborate variants, one might also store block indices, sink masks, head-group identifiers, or calibration statistics.

A useful way to think about metadata is as a minimal witness for invertibility. If the quantiser is a map $Q : \mathcal{X} \to \mathcal{Y}$, then the metadata supplies enough context for a dequantiser D such that $D(Q(x), m(x)) \approx x$. The goal is not exact identity for arbitrary tensors. The goal is controlled and testable reconstruction error.

27.3 Invertibility in the no-clipping regime

The affine quantiser is exactly invertible on the quantisation grid when no clipping occurs. Let the representable levels be $\{0, \ldots, 2^b - 1\}$. If a value x is encoded as

$$q = \left\lfloor \frac{x - x_{\min}}{s} \right\rceil,$$

and then reconstructed by

$$\hat{x} = qs + x_{\min},$$

then $\hat{x}$ matches the nearest representable grid point. If the original x already sits on that grid and is not saturated, then reconstruction is exact.

This observation is useful for two reasons. First, it gives a clean correctness argument for synthetic tests. Second, it tells us that compression error is not mysterious noise; it is structured rounding error. Once that is understood, the rest of the design can focus on making the rounding happen where it hurts least.

27.4 Numerical stability in mixed precision

The compressor works with quantised cache tensors, but the intermediate arithmetic should remain in float32 where practical. That is because min, max, scale, and zero-point calculations can become noisy in lower precision, especially for wide dynamic ranges or small blocks. The additional cost is negligible compared with the cache itself, and the stability gain is worth it.

This design choice is one of the easiest to defend in a paper. Use the inexpensive precision where it saves memory, and use the more stable precision where it protects the transformation itself. That rule is conceptually simple and practically strong.

28 A rate-distortion view

The cleanest theoretical framing of KVQuant is rate-distortion optimisation. Each token or token group has a distortion curve that falls as the precision budget rises. The system then solves a constrained allocation problem: how should we spend bits so that the expected utility loss is as small as possible?

Let the distortion for item i at precision q_i be approximated as

$$\mathcal{E}_i(q_i) = \alpha_i 2^{-q_i}.$$

The total objective is

$$\min_{q_1,\ldots,q_n} \sum_{i=1}^{n} s_i \alpha_i 2^{-q_i} \quad \text{subject to} \quad \sum_{i=1}^{n} q_i \leq B.$$

Here s_i is a sensitivity score and B is the total bit budget.

The Lagrangian is

$$\mathcal{J}(q,\lambda) = \sum_i s_i \alpha_i 2^{-q_i} + \lambda \left(\sum_i q_i - B \right).$$

The KKT stationarity condition gives

$$\frac{\partial \mathcal{J}}{\partial q_i} = -s_i \alpha_i \ln 2 \cdot 2^{-q_i} + \lambda = 0.$$

Solving for q_i yields

$$q_i = \log_2 \left(\frac{s_i \alpha_i \ln 2}{\lambda} \right).$$

The precise constant does not matter much. The shape does. Precision should grow logarithmically with importance. That is the key insight.

This result explains why a discrete schedule with only a few bit levels can still be sensible. The optimum of the relaxed problem is continuous, but the nearest practical implementation is a small number of tiers. KVQuant is effectively a discretised rate-distortion system.

28.1 Monotonicity of the optimal allocator

Suppose the sensitivity scores are ordered so that $s_1 \geq s_2 \geq \cdots \geq s_n$ and the scale factors α_i do not increase too sharply. Then the optimal precision schedule is nonincreasing in the reverse order of importance. In other words, more important items should not receive fewer bits than less important items unless a strong structural constraint says otherwise.

This monotonicity argument is one of the simplest ways to justify the recency-aware rule. Recent tokens typically have higher effective sensitivity than old tokens, so a decreasing bit schedule across age is consistent with the relaxed optimum.

29 Sink discovery and protected sets

Attention sinks are important because a cache compressor that blindly compresses everything equally will sometimes destroy the few positions that the model keeps returning to. A practical compressor therefore benefits from a protected set S of positions that remain at high precision.

There are several ways to define S. One can use fixed positions, such as the first few tokens or delimiter tokens. One can use moving average attention mass. Or one can use a hybrid rule that protects tokens whose aggregate attention score stays above a threshold for many steps.

A simple sink score at time t is

$$u_t = \beta a_t + (1-\beta) r_t,$$

where a_t is accumulated attention mass and r_t is a recency bonus. Tokens with the highest u_t values enter the protected set. A hysteresis margin can be added so that the protected set does not oscillate wildly between steps.

This matters because stream behaviour is not static. The tokens that matter early in generation are not always the same tokens that matter later. A practical sink detector should therefore be incremental, not batch-only.

29.1 A conservative sink policy

A conservative sink policy can be summarised as follows:

Input: attention mass estimates, recency values, protection threshold
Output: sink set S

1. Compute sink scores for all candidate positions.
2. Add positions above the threshold to S.
3. Keep special prompt anchors in S by default.
4. Remove positions only if they remain below the threshold for several steps.
5. Return the stable protected set.

This kind of policy is attractive because it is robust and simple to explain.

30 Head-and-layer coupling

A head-aware allocator extends the score with per-head and per-layer multipliers. Let $\rho_{\ell,h}$ denote head sensitivity and γ_ℓ denote layer sensitivity. Then the combined score is

$$\tilde{s}_{\ell,h,t} = \gamma_\ell \rho_{\ell,h} u_t.$$

This is a compact way to express the idea that the same token does not mean the same thing to every head or every layer.

If one head acts mostly as a local pattern matcher, then aggressive compression may be harmless. If another head appears to carry long-range dependencies, then it deserves more precision. The same logic applies to layers: some layers are more fragile than others with respect to cache perturbation.

The important point is that head-and-layer coupling is not a separate model. It is the same allocator viewed through a richer set of weights. That makes it easy to extend without rewriting the rest of the pipeline.

31 Runtime wrapper contract

The wrapper around the forward pass should satisfy a few simple but non-negotiable properties.

31.1 No accidental mutation

If the model output does not contain cache state, the wrapper must return the output unchanged. It should not invent or infer structure where none exists.

31.2 Type preservation

If the model returns a dictionary, the wrapper should preserve the dictionary shape. If it returns an object with a `past_key_values` attribute, the wrapper should replace only that attribute. If it returns a tuple, the wrapper should rewrite only the element that actually contains cache state.

31.3 Idempotence over compressed inputs

If an already-compressed cache is passed in, the wrapper should not compress it again. That is why the implementation uses a structured wrapper object and checks for it before recompression.

31.4 Decompression before reuse

Incoming cache state should be decompressed before the model consumes it. Compression should occur only after the new forward pass has produced a fresh cache.

These rules are mundane, but they are exactly what separate a trustworthy systems layer from a brittle demo.

32 Benchmark design for a publication-quality evaluation

A publishable version of the paper should evaluate not only memory savings but also quality, latency, and robustness. The core metrics are:

Metric	Why it matters
Peak memory	Shows whether the method solves the deployment bottleneck.
Latency per token	Measures whether compression overhead is acceptable.
Throughput	Captures serving efficiency in batch settings.
Task accuracy or perplexity	Measures the quality cost of compression.
Stability under long context	Tests whether errors accumulate over time.
Metadata overhead	Quantifies the hidden cost of reversibility.

A strong benchmark should sweep sequence length, batch size, and bit-width. It should also report how the method behaves under long prompts with repeating anchors, because that is where sink behaviour becomes most visible.

33 Ablation matrix

The following ablation matrix describes the experiments that would make the paper far more convincing. It is intentionally explicit so that the evaluation plan can be executed without rethinking the paper structure.

Factor	Values	Hypothesis	Failure mode
Bit-width	2, 3, 4, 8	More bits reduce error but lower compression	Diminishing returns
Window sizes	short / medium / long	Wider recent windows preserve quality	Smaller savings
Sink protection	off / on	Protection helps repeated anchors	Extra metadata
Block size	32, 64, 128, 256	Smaller blocks reduce distortion	Larger overhead
Wrapper mode	on / off	Wrapper adds small overhead	Unsupported outputs
Head weighting	uniform / weighted	Weighted policy better matches sensitivity	Calibration cost
Layer weighting	uniform / weighted	Layer-aware policy protects fragile layers	More complexity

The point of such a matrix is not to create many numbers for their own sake. It is to show which design choice actually drives the gain.

34 Baselines and comparison protocol

A fair comparison should include at least four baselines: no compression, uniform int8 quantisation, simple blockwise quantisation, and recency-only allocation without sink protection. If possible, a token-eviction baseline should also be included, since some systems trade memory for preserved tokens rather than compressed tokens.

The comparison should be done under the same decoding settings, with the same model and prompt set, and the same measurement procedure. Otherwise, the numbers become a disagreement about engineering details rather than a comparison of methods.

A good paper also separates static and dynamic baselines. Static baselines use fixed bit-widths across the entire cache. Dynamic baselines change precision with sequence age or importance. KVQuant belongs to the dynamic family, and that distinction should be clear in the experimental section.

35 Failure analysis and guardrails

A good paper earns trust by discussing how things go wrong.

35.1 Distribution shift

If the cache distribution on real prompts is wider than on synthetic tests, the quantiser may clip more often. This can inflate error unexpectedly. The fix is simple: calibrate on representative prompts and monitor saturation rates.

35.2 Unsupported output types

Some model wrappers return unusual structures. The compressor should detect those cases and fall back safely rather than trying to guess the object layout. Silence is better than corruption.

35.3 Mixed precision interactions

If the base model already uses low-precision weights or activation formats, the cache compressor must avoid assumptions about dtype uniformity. In practice, the metadata should record enough information to reconstruct the intended compute type.

35.4 Streaming drift

In long generations, small errors can accumulate. That is one reason sink protection and recency protection are both useful: they reduce the chance that the compressor keeps damaging the same critical state over and over.

36 Ethics and release policy

The paper should remain honest about what is measured and what is not. That means no fabricated acceptance claims, no made-up benchmark numbers, and no pretending that a technical report is already a conference paper.

The responsible release strategy is straightforward:

- publish the source code and paper draft,
- state the model, dataset, and benchmark scope exactly,
- describe the limitations in plain language,
- avoid overclaiming novelty where the method is an extension of existing ideas,
- keep the implementation reproducible.

That is not a constraint. It is a credibility multiplier.

37 Expanded appendix: proof sketch for monotone precision

Let the items be ordered by nonincreasing sensitivity $s_1 \geq s_2 \geq \cdots \geq s_n$. Suppose the distortion function is decreasing and convex in the precision variable. If an allocator assigns a lower precision to a more sensitive item while a less sensitive item receives a higher precision, then swapping one unit of precision from the less sensitive item to the more sensitive item strictly improves the objective while preserving the budget, provided the derivative gap is large enough. Therefore, any optimum must be monotone up to ties induced by structural constraints.

This is only a sketch, but it is exactly the kind of sketch that motivates the discrete window schedule used by KVQuant. If the relaxed optimum is monotone, a quantised schedule should roughly respect that monotonicity.

38 Expanded appendix: artefact checklist

A final submission bundle should include:

- the PDF,
- the LaTeX source,
- the bibliography file,
- the code commit hash,
- benchmark scripts,
- a brief environment description,
- a note describing what is synthetic versus what is end-to-end.

The point of the checklist is to make the paper easy to evaluate by a stranger who has not seen the project before.

A Deriving the budgeted allocator

Assume the per-token distortion behaves like

$$\mathcal{L}_t(q) = \alpha_t 2^{-q}.$$

We want to solve

$$\min_{q_1,\ldots,q_T} \sum_{t=1}^{T} w_t \mathcal{L}_t(q_t) \quad \text{subject to} \quad \sum_{t=1}^{T} q_t \leq B.$$

The Lagrangian is

$$\mathcal{J}(q,\lambda) = \sum_t w_t \alpha_t 2^{-q_t} + \lambda \left(\sum_t q_t - B \right).$$

The stationarity condition gives

$$\frac{\partial \mathcal{J}}{\partial q_t} = -w_t\alpha_t \ln 2 \cdot 2^{-q_t} + \lambda = 0,$$

so

$$q_t = \log_2\left(\frac{w_t\alpha_t \ln 2}{\lambda}\right).$$

Thus, optimal precision is logarithmic in importance. A discrete multi-level schedule is therefore a reasonable approximation.

B Expanded pseudocode

Input: cache tensors K, V, budget B, importance scorer $s(\cdot)$

Output: compressed cache

1. Compute importance scores.
2. Initialise all bit-widths to the minimum allowed value.
3. While budget remains, give one more bit to the token or group with the largest marginal gain per bit.
4. Quantise keys and values using the resulting allocation.
5. Store metadata for reconstruction.
6. Return compressed cache.

C Symbol table

Symbol	Description
M_{KV}	Total KV cache memory
C	Total storage budget
q_t	Precision assigned to token t
S	Protected sink set
α_t	Distortion scale for token t
λ	Lagrange multiplier
$\rho_{\ell,h}$	Head-specific sensitivity weight
γ_ℓ	Layer-specific sensitivity weight

D Reproducibility notes

The local repository currently includes unit tests for bit allocation, uniform quantisation, blockwise quantisation, dequantisation dtype, cache compression round-trip, wrapper behaviour, and memory reduction accounting. The test suite passes, which is a strong signal that the implementation is internally coherent.

For a submission bundle, the following should be preserved:

- the exact commit hash,
- the benchmark script,
- the environment description,
- the public PDF and source archive,
- any external evaluation code used for real-model experiments.

E Worked numerical examples

To make the memory story less abstract, it helps to walk through a few concrete cases.

E.1 Single-layer intuition

Suppose a single attention layer stores $H = 32$ heads with head dimension $D_h = 128$ over a sequence of $T = 4096$ tokens. For one batch element in fp16, the raw KV footprint is

$$M_{KV} = 2 \cdot 1 \cdot 1 \cdot 32 \cdot 4096 \cdot 128 \cdot 2$$

bytes, which is about 64 MiB for the layer alone. Scale that to 32 layers and the number becomes large enough to matter very quickly.

The 4-bit theoretical footprint reduces the element cost by a factor of four. Even after metadata, the drop is substantial. If a blockwise scheme stores one scale and one offset per block, the extra bytes are usually tiny compared with the cache once the sequence length is large.

E.2 Long-context dialogue

Now consider a multi-turn dialogue with a long persistent prefix. The oldest tokens include system instructions, user preferences, and a few anchor terms. The newest tokens are the current thought process. A compressor that spends the same precision on both is overspending on the stable region and underspending on the volatile region.

A recency-aware rule therefore behaves sensibly even before we add more elaborate importance signals. The oldest tokens can often be shrunk more aggressively, while the recent window remains protected. If a sink detector identifies a long-lived anchor, that anchor gets pulled back into the high-precision bucket.

E.3 Budget pressure scenario

Assume the serving system targets a fixed memory budget that is 40 percent lower than the uncompressed KV footprint. Under that constraint, a uniform 8-bit cache may not be enough, but a tiered 4-bit or mixed 4/3/2-bit policy might still fit comfortably. This is exactly the regime where selective cache compression becomes operationally useful rather than merely elegant.

F Detailed pseudocode for the full system

The following pseudocode describes a more complete system than the minimal wrapper shown earlier. It is not a literal code listing. It is a readable summary of the intended control flow.

Algorithm 4: Full KVQuant serving loop

Input: model f, prompt tokens $x_{1:T_0}$, budget B, sink policy π, precision levels Q
Output: generated tokens $y_{1:U}$

1. Encode the prompt and run the model on the initial context.
2. Extract the first cache state.
3. Estimate token importance using recency, sinks, and head weights.
4. Solve or approximate the precision allocation problem under budget B.
5. Compress the cache and store metadata.
6. For each new generation step:

a. Decompress the cache.
b. Run the model for one step.
c. Recompute importance scores if needed.
d. Recompress the updated cache.
e. Append the new token to the output.

7. Return the generated sequence.

This loop makes the design goal clear: compression is not a one-off transform but a repeated part of generation.

G Precision scheduling heuristics

The current prototype uses a small set of bit levels. That is reasonable because bit-width itself is discrete in deployment. The scheduler can therefore be thought of as a classifier over importance bands.

A simple importance-to-bit mapping can be written as

$$q_i = g(s_i),$$

where g is monotone decreasing in the sensitivity score if higher sensitivity should mean more bits.

One practical form is a thresholded step function:

$$g(s) = \begin{cases} 8 & s \geq \tau_3, \\ 4 & \tau_2 \leq s < \tau_3, \\ 3 & \tau_1 \leq s < \tau_2, \\ 2 & s < \tau_1. \end{cases}$$

The thresholds can be tuned by calibration. In a mature system, they might be estimated from validation prompts using a trade-off objective that balances error and memory.

G.1 Why thresholding is useful

Thresholding is crude, but it is explainable. It makes it obvious which tokens are protected and which are compressed. That transparency is valuable when debugging quality regressions, especially for a first public release.

H Calibration protocol

A robust compressor should be calibrated before it is deployed. Calibration does not need to be expensive. A small representative set of prompts can reveal the typical activation range, the frequency of sink-like tokens, and the sensitivity of different heads.

A compact calibration workflow is:

1. collect representative prompts,
2. run the model with full precision,
3. measure token-wise activation ranges and attention mass,
4. estimate per-head sensitivity weights,
5. choose window sizes and thresholds,
6. validate on held-out prompts.

Calibration is what turns a general algorithm into a model-specific system. It is also the point where a paper can honestly claim that it has been adapted to a real workload.

I Experiment templates

The paper should include a section like this in a submission-ready version. The goal is to make the evaluation easy to reproduce.

Template 1: Memory stress test

Prompt lengths: 1k, 2k, 4k, 8k, 16k, 32k

Batch sizes: 1, 2, 4

Metrics: peak memory, successful decode rate, compression ratio

Expected result: KVQuant should delay or prevent OOM at long lengths.

Template 2: Quality preservation test

Tasks: long-context QA, summarisation, dialogue continuation

Metrics: exact-match, token F1, perplexity, human preference

Expected result: 4-bit and mixed precision should preserve utility better than 2-bit.

Template 3: Sensitivity ablation

Settings: with and without sinks, with and without head weights, different block sizes

Metrics: error, memory, latency

Expected result: sink-aware and head-aware variants should outperform pure recency on prompts with repeated anchors.

J Worked error decomposition

Let the error between the original and compressed cache be decomposed as

$$\Delta = \Delta_{round} + \Delta_{clip} + \Delta_{scale} + \Delta_{layout}.$$

Here Δ_{round} is due to integer rounding, Δ_{clip} due to saturation, Δ_{scale} due to finite-scale approximation, and Δ_{layout} due to any packing or reshaping artefacts.

This decomposition is practical because each term points to a different fix. If clipping dominates, widen the dynamic range. If rounding dominates, spend more bits. If layout error dominates, improve the packing logic. A good systems paper should isolate these terms rather than bundle them into one vague "quantisation error" bucket.

K Deployment checklist

Before turning the prototype into a real serving component, the following should be validated:

- the wrapped forward pass preserves model outputs when compression is disabled,
- compressed caches can be serialised and restored,
- runtime overhead is lower than the memory savings justify,
- all supported model families return cache layouts the wrapper understands,

- the chosen bit policy does not catastrophically degrade quality on long prompts,
- the benchmark and evaluation scripts are fixed to a commit hash,
- the paper clearly marks synthetic results versus end-to-end results.

This checklist is useful because a surprising number of apparently strong ideas fail one of these boring but essential checks.

L Additional proof sketch: metadata sufficiency

Suppose the compressor stores a tuple (q, m) where q is the quantised data and m is the metadata. If the metadata determines the scale, zero point, original shape, and any necessary padding, then a deterministic dequantiser can recover a unique tensor in the reconstruction space. Thus the metadata is sufficient for invertibility within the chosen numerical model.

The statement is not that all error disappears. The statement is that the compressed object is well-defined and reversible up to the expected quantisation distortion.

M Suggested presentation figures

A final paper would benefit from the following figures:

1. a diagram of the cache lifecycle from prompt to compressed cache to restored cache,
2. a plot of memory usage versus sequence length for different bit-widths,
3. a quality-vs-compression curve,
4. a breakdown of error contributions,
5. a diagram showing how sinks remain protected while older non-sink positions shrink.

These figures are not cosmetic. They will make the paper much easier to understand at a glance.

N Submission positioning

The strongest honest positioning for the paper is as a technical report or preprint on efficient long-context inference. That gives the work room to be useful without pretending it already has venue acceptance. If the future evaluation is strong enough, it could become a workshop paper, a main-conference submission, or a systems-oriented preprint that the community can cite.

That is a much better story than forcing the work into a venue where it does not belong.

O References

References

[1] KVQuant repository codebase.

[2] C. Lou et al. KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache. arXiv:2402.02750.

[3] Z. Zhang et al. H_2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. arXiv:2306.14048.

[4] Y. Xiao et al. Efficient Streaming Language Models with Attention Sinks. arXiv:2309.17453.

[5] Survey literature on efficient LLM inference and long-context serving.