KAP: BRIDGING THE KNOWLEDGE SELECTION–RUNTIME CONSUMPTION GAP IN LLM SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern LLM systems increasingly rely on knowledge-selection pipelines that produce structured signals, such as evidence rankings, graph relations, multimodal alignment, and confidence scores. However, these signals are typically discarded when knowledge is serialized into a flat prompt, leaving the serving system unaware of which knowledge is important during decoding. This creates a gap between knowledge selection and runtime consumption: the model is presented with a complete logical context, while the runtime lacks the information needed to avoid unnecessary KV access. We propose *Knowledge Access Planning* (KAP), an execution abstraction that bridges this gap by carrying structured knowledge priors into LLM serving through a universal intermediate representation (IR)—the runtime access plan. KAP separates logical prompt semantics from physical execution: its compiler translates knowledge-selection priorities into backend-executable plans, while its executor selectively accesses physical KV states according to these plans, without modifying the model, prompt semantics, or training procedure. We instantiate KAP with GRAPHSPEC, a reference realization that executes runtime access plans within VLLM, and derive a conditional cost model characterizing its positive-speedup regime. Across a range of experimental settings, GRAPHSPEC reduces proposal-path KV exposure to 5.1–7.9% of the source state at 128K and translates this reduction into long-context decode speedups while maintaining answer quality comparable to full-context decoding. These results demonstrate KAP's effectiveness in substantially reducing proposal-path KV exposure. More broadly, they show that knowledge-selection signals can be compiled into actionable execution plans, extending their role beyond prompt construction to efficient LLM decoding.

1 INTRODUCTION

The Knowledge Selection–Runtime Consumption Gap. For a fixed serialized prompt, conventional full-context decoding performs the same KV access regardless of which passages the frontend prioritizes. We refer to this disconnect between frontend knowledge priorities and backend KV access as the Knowledge Selection–Runtime Consumption (KSRC) gap. To characterize the gap quantitatively, we define the *evidence-to-context ratio* as $r_{\text{ref}} = K/L$. Here, K counts all tokens in each prompt passage containing dataset-provided reference evidence, with each token position counted once; L counts all tokens in the complete prompt. Table 1 reports median ratios of 1.14–1.29% for native retrieval outputs; at 128K, these passages account for 0.27–0.73% of aggregate prompt tokens. The magnitude of this disparity makes the conversion of knowledge selectivity into execution efficiency a central systems challenge for scaling LLM inference.

The small share of context occupied by reference-matched passages naturally motivates selective processing. Prompt-compression methods introduce selectivity at the prompt-construction level by removing or rewriting parts of the logical input before inference (Jiang et al., 2023; 2024b; Li et al., 2023). This changes the logical input presented to the model and, in turn, the attention computation during decoding. We instead introduce selectivity directly at the execution level while preserving the complete logical prompt and its positional semantics: knowledge-selection decisions are carried into the serving runtime as explicit access plans that govern physical KV access.

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.

Table 1: **Evidence-to-context ratios** ($r_{\text{ref}} = K/L$). K counts all tokens in complete prompt passages matched to official dataset evidence annotations: reference captions, rationales, or figure identifiers for SPIQA, and gold supporting paragraphs for MuSiQue. Passages are identified by complete reference-text matches after Unicode, case, and whitespace normalization. Each prompt-token position is counted once; L counts all prompt tokens. Panel A reports request-level medians [25th–75th percentiles]; Panel B reports token-weighted ratios $\sum_i K_i / \sum_i L_i$.

Panel A: Native retrieval-output distributions

Workload	Frontend	Native tokens (10^3)	r_{ref}
SPIQA	Graph-RAG / BGE-M3	59.7 [45.8, 78.0]	1.29% [0.39%, 1.80%]
MuSiQue	Graph-RAG	26.3 [24.0, 28.9]	1.14% [0.76%, 1.59%]

Panel B: Controlled context-length sweep

Workload	Frontend	4K	8K	16K	32K	64K	128K
SPIQA	Graph-RAG / BGE-M3	14.76%	11.04%	5.79%	2.89%	1.43%	0.73%
MuSiQue	Graph-RAG	7.63%	4.11%	2.08%	1.06%	0.53%	0.27%

Knowledge Access Planning. To realize this execution-level selectivity, we introduce KAP, a compiler–executor architecture built around the runtime access plan. The central design challenges are to define a common plan contract and to compile it efficiently for different serving backends. The runtime access plan serves as a universal IR for expressing selected knowledge, its runtime object mapping, logical positions, and intended access behavior. As Figure 1 illustrates, the compiler constructs the plan from frontend priors, resolving selected knowledge to runtime objects under the target backend's layout and execution constraints. The executor then interprets the plan as runtime access actions. Frontends and backends therefore connect through a common contract rather than depending on each other's implementation (Section 2). This makes knowledge-aware access a reusable serving capability, with concrete execution strategies supplied by backend-specific realizations.

GraphSpec as a concrete instantiation. To realize this contract within an existing serving system, we instantiate KAP as GRAPHSPEC, using Graph-RAG as the knowledge-selection frontend and vLLM as the LLM serving backend. Its compiler maps graph-derived evidence to logical-position-

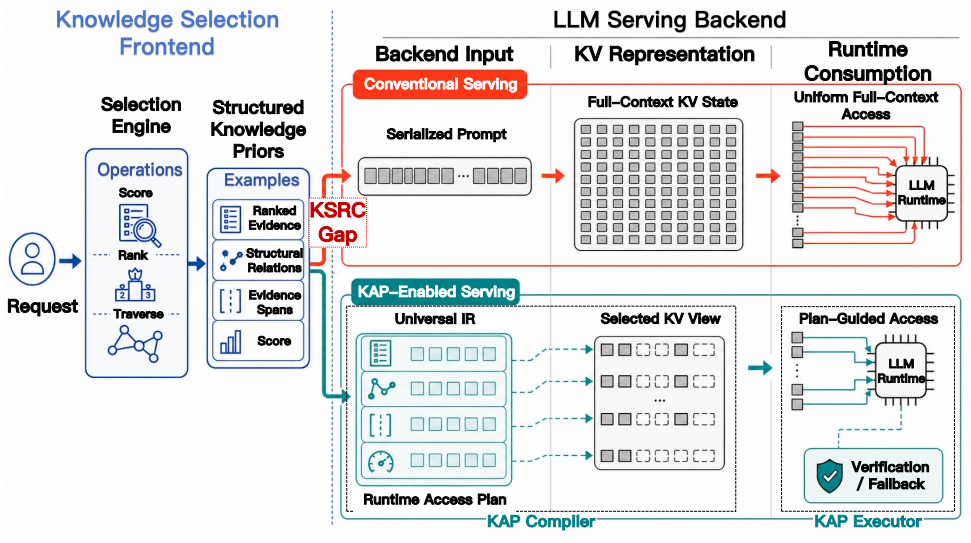


Figure 1: **The KSRC gap and KAP-enabled LLM serving.** Conventional serving collapses structured priors into a serialized prompt and uniformly consumes full-context KV state. KAP instead preserves them in runtime access plans that map selected knowledge to backend objects for plan-guided access with policy-specific verification or fallback.

preserving selected-KV views within runtime access plans; the executor uses these views for same-model proposal with full-context verification (Sections 3 and 4).

Contributions. Our main contributions are:

- **Architectural diagnosis:** We characterize the knowledge selection–runtime consumption gap through architectural and quantitative analyses, revealing a structural mismatch between frontend knowledge priorities and the largely uniform consumption imposed by conventional LLM serving runtimes.
- **Knowledge Access Planning:** We introduce KAP, a plan-driven compiler–executor architecture that establishes a reusable execution contract between knowledge-selection frontends and LLM serving backends. By carrying knowledge priorities into explicit access plans, KAP makes frontend selection decisions actionable in physical execution while decoupling frontend knowledge logic from backend-specific runtime implementations.
- **GraphSpec realization and validation:** We build GRAPHSPEC as a reference implementation of KAP, demonstrating that plan-driven knowledge access substantially reduces proposal-path KV exposure while delivering favorable quality–efficiency trade-offs. Validation spans diverse experimental settings and is complemented by a realization-specific conditional cost model characterizing the conditions for speedup.

2 KNOWLEDGE ACCESS PLANNING

Runtime access plan. The runtime access plan serves as KAP's universal IR, bridging frontend knowledge-selection decisions and backend execution. We represent a plan as

$$\mathcal{A} = (U, M, pos, \rho_{\text{access}}, \rho_{\text{verify}}), \tag{1}$$

where U contains selected knowledge units; M grounds them in the complete logical prompt and maps their token spans to backend runtime objects; pos records their original logical positions; ρ_{access} specifies how the associated runtime state is accessed; and ρ_{verify} specifies the verification or fallback policy governing output commitment. This representation distinguishes knowledge-selection intent from its physical realization: the field semantics and execution requirements define a common contract across backends, while M supplies the backend-specific object references and metadata needed to execute the plan.

Execution contract. The contract imposes three cross-field requirements on compilation and execution. Selected units and their grounding must be preserved (*prior preservation*). Their original logical positions must remain unchanged under physical reorganization (*positional fidelity*) (Su et al., 2024; Liu et al., 2024). The executor must apply the declared verification or fallback policy before committing outputs (*policy-explicit validation*). These requirements do not by themselves imply output equivalence, which depends on the policies implemented by the executor.

Compiler–executor architecture. As illustrated in Figure 2, the KAP compiler constructs the runtime access plan $\mathcal{A}$ from frontend knowledge priors and the complete logical prompt, subject to a selection budget and the target backend's resource constraints. It uses these priors to select and ground knowledge, encodes the intended access behavior together with the configured validation policy, and lowers the selected knowledge to backend runtime objects while preserving its original logical positions. In this way, frontend knowledge priorities remain actionable after serialization: they are carried forward as explicit selections and access decisions within an executable plan, rather than remaining frontend-only metadata. The executor therefore need not interpret the frontend's native metadata; it consumes the compiled plan $\mathcal{A}$, resolves the mapped runtime objects, and executes the declared access and validation policies.

3 GRAPHSPEC: INSTANTIATING KAP WITH A GRAPH-RAG FRONTEND AND VLLM BACKEND

Compiler and runtime object mapping. The key role of the GraphSpec compiler is to map frontend-selected evidence to backend-executable KV objects while preserving the original logical positions and prompt semantics. The compiler (Figure 3) resolves selected evidence spans to

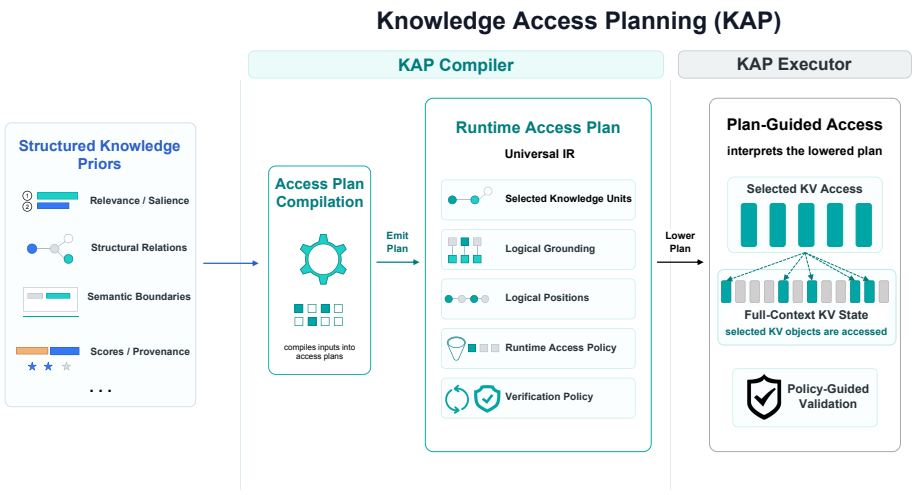


Figure 2: **KAP compiler–executor architecture.** The compiler maps structured priors to runtime objects within an access plan; the executor consumes the plan and applies its access and verification policies.

paged-KV objects and materializes a compact proposal view in M. This view reuses KV states formed by full-context prefill, allowing frontend selection to govern proposal access without re-encoding the selected evidence as a shortened prompt (Section 5 and Appendix D compare related mechanisms and controls). To reconcile compact physical storage with the plan's logical coordinates, block tables map selected logical blocks to compact physical pages, while slot mappings place newly scheduled tokens in the corresponding KV slots. Logical positions remain in original prompt coordinates rather than being replaced by compact storage indices. The SPIQA case in Figure 3 follows three highlighted evidence spans from their original token positions through runtime mapping to plan-guided execution, illustrating how frontend priorities guide physical KV access.

Plan-guided execution and verification. Selective KV access can reduce proposal cost but may omit information available in the full context. GraphSpec therefore combines selected-KV proposal with full-context verification to govern token acceptance under an explicit verification policy. Selected-KV proposal uses single-token decode steps with the same base model as the full-context verifier. After k candidates, verification processes a $(k+1)$-token window containing a verifier-approved carry token and the candidates. It commits the prefix accepted under the declared policy and advances full-context state within that transition, avoiding a separate resynchronization pass. The resulting output guarantees are policy-specific. Top-1 accepts a proposed token only if it matches the full-context verifier's highest-probability token, whereas Top-3 accepts it if it is among the verifier's three highest-probability tokens.

Runtime-efficient plan execution. GraphSpec realizes runtime access plans through a reusable selected-KV execution path integrated with vLLM's optimized uniform-decode machinery. This integration is essential because reduced KV exposure may otherwise be offset by repeated metadata construction, allocation, or disrupted CUDA-graph replay. GraphSpec stabilizes query shapes and buffer addresses, reuses compact-KV workspaces and compatible attention metadata, and updates per-step state in place. Query-length-specific dispatch and separate graph tags support CUDA-graph replay for proposal and verification. The resulting execution path combines knowledge-aware access with the backend's execution regularity, allowing plan-driven selectivity to benefit from existing serving optimizations. Appendix A provides the compilation algorithm, verifier transition, and runtime execution details.

GraphSpec conditional cost boundary. Effective planning must consider not only the reduction in retained KV state, but also the conditions under which selective access translates into improved decoding efficiency. We derive a conditional cost model for GraphSpec that connects compiled-plan selectivity, proposal acceptance, and backend execution costs. Selective access reduces proposal

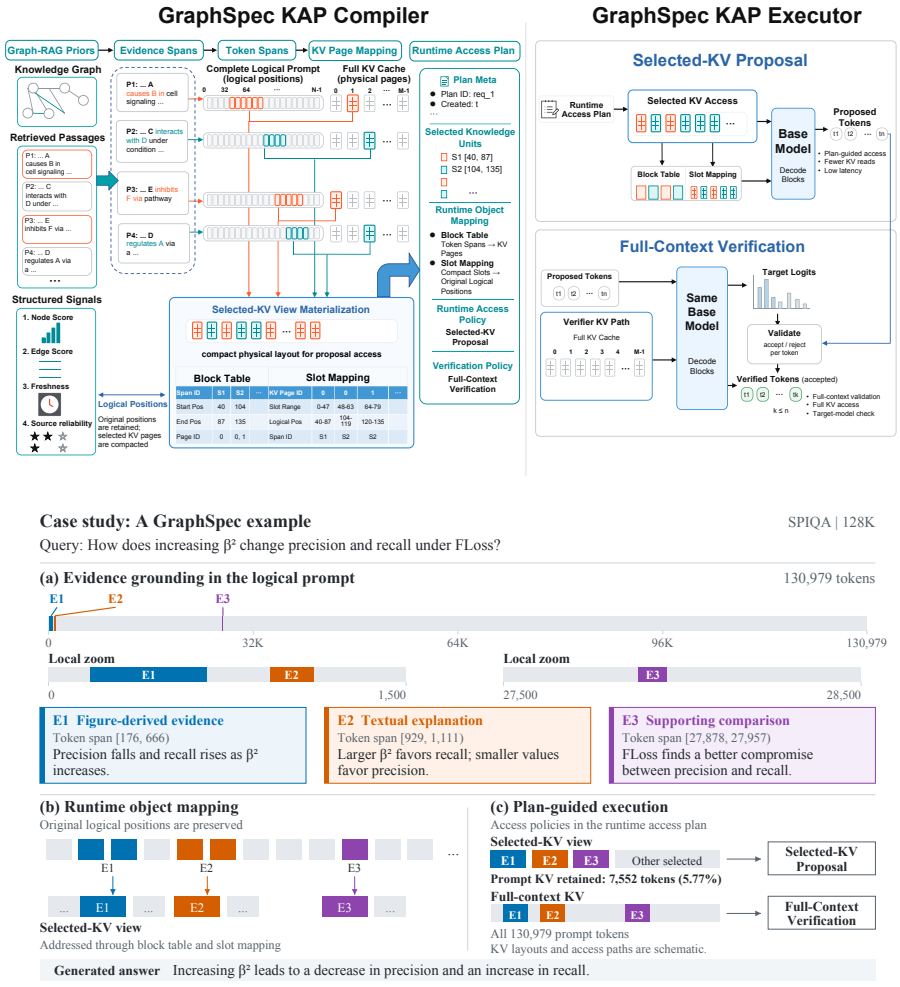


Figure 3: **GRAPHSPEC realization of the KAP compiler–executor architecture.** Top: the compiler maps graph-derived evidence to vLLM paged-KV objects within runtime access plans; the executor performs selected-KV proposal with full-context verification. Bottom: a 128K SPIQA case with three highlighted evidence spans, their runtime mapping, and plan-guided execution. Evidence text is summarized; KV layouts and access paths are schematic.

cost, but lower acceptance limits the amortization of full-context verification. Let s denote the retained-KV ratio, h proposal acceptance, KV the full-state read cost, and W_{main} the per-step model-weight read cost. Appendix B provides the full derivation. Under an idealized memory-bound model, a plan with $h > s$ yields positive speedup when

$$KV > KV_{\text{crit}} \approx \frac{(1-h)W_{\text{main}}}{h-s}. \tag{2}$$

The acceptance–retention margin $h - s$ determines whether selective proposal creates a KV-access advantage, while $(1-h)W_{\text{main}}$ captures the residual model-weight cost that this advantage must offset. Frontend prior quality and the compiler budget determine the attainable (s, h) operating point; backend costs determine the context scale at which that operating point becomes beneficial. The criterion thus connects knowledge selection to execution utility: a plan must preserve sufficient proposal acceptance for reduced KV exposure to translate into decoding efficiency.

For given s and h, the proposal length k cancels out in the derivation of the ideal boundary but affects verification amortization and runtime overheads; based on implementation experience, we fix $k = 6$ across comparisons. RQ4 assesses whether speedup trends across context lengths are consistent with the predicted break-even behavior.

4 EXPERIMENTS

4.1 SETUP

Settings and controls. The primary setting uses Qwen3-VL-32B-Instruct (Bai et al., 2025) on four NVIDIA A800 80GB GPUs; model transfer uses Llama-3.1-8B-Instruct (Grattafiori et al., 2024) on two A800 GPUs. GraphSpec extends vLLM v0.16.0, uses the same base model for proposal and verification, and fixes $k = 6$. We evaluate SPIQA, a multimodal scientific-paper QA dataset (Pramanick et al., 2024), with Graph-RAG and BGE-M3 (Chen et al., 2024), and MuSiQue-Ans, a multi-hop text QA dataset (Trivedi et al., 2022), with Graph-RAG. These complementary workloads test KAP across evidence sources and reasoning requirements. To compare GraphSpec with the baselines across context lengths, we construct nested 4K–128K prompts that retain the same core evidence while progressively adding hard negatives.

To assess the benefits of translating knowledge priorities into runtime access decisions, we compare GraphSpec Top-1/Top-3 with full-context decoding, three prompt-compression baselines, and two closely related proposal baselines we implement in vLLM: RAPID-core and MagicDec-SnapKV. RQ2 details baseline designs and objectives. Paired methods share model, tokenization, decoding settings, software, and hardware. GraphSpec denotes Top-3 unless noted.

Table 2: **SPIQA quality audit.** Judge–human agreement and evidence coverage.

Dataset	Samples	Judge–human agreement	Answer recall	Rationale recall	Caption recall
SPIQA	100	95%	68.8%	66.8%	76.8%

Table 3: **RQ1 primary-setting results on SPIQA.** Panel A reports throughput and plan statistics; Panel B reports latency and answer quality. GraphSpec uses Top-3 verification.

Panel A: Decode throughput and plan statistics

	Full-context	GraphSpec			
Length	TPS	TPS	Retained-KV (s)	Acceptance (h)	Speedup (95% CI)
4K	51.91	**55.38**	**39.2%**	94.7%	**1.067** [1.057, 1.076]×
8K	51.51	**55.04**	**34.3%**	94.8%	**1.068** [1.059, 1.078]×
16K	50.57	**54.22**	**24.1%**	94.9%	**1.072** [1.062, 1.081]×
32K	48.68	**52.59**	**16.0%**	94.7%	**1.080** [1.069, 1.091]×
64K	45.66	**51.15**	**10.8%**	95.4%	**1.120** [1.110, 1.130]×
128K	41.10	**48.09**	**5.5%**	94.5%	**1.170** [1.161, 1.179]×

Panel B: Mean per-request latency and answer pass rate

	TTFT (s)		TPOT (ms)		E2E latency (s)		Answer pass rate	
Length	Full	GraphSpec	Full	GraphSpec	Full	GraphSpec	Full	GraphSpec
4K	1.51	1.56	19.44	**18.14**	3.77	3.71	80.0%	80.2%
8K	1.69	1.76	19.60	**18.30**	3.99	3.96	82.6%	82.9%
16K	2.06	2.17	19.96	**18.58**	4.38	4.37	87.4%	87.4%
32K	4.87	4.92	20.74	**19.18**	7.26	7.15	86.9%	87.0%
64K	12.56	12.71	22.16	**19.67**	15.04	14.95	86.0%	87.9%
128K	38.31	37.90	24.64	**20.91**	41.00	40.20	84.6%	84.3%

Evaluation protocol. To focus the comparison on serving efficiency, frontend evidence is frozen before timing and retrieval is excluded for all methods. GraphSpec's TTFT and end-to-end latency include plan compilation and selected-KV materialization, accounting for its additional preparation costs. Decode throughput covers the complete executor, and 95% confidence intervals for primary speedups are obtained by bootstrapping paired requests. All LLM-based evaluations use the same judge model across datasets and configurations. SPIQA uses a pre-specified LLM-similarity threshold of 0.5 for pass-rate evaluation; judge decisions agree with three-annotator majority vote on 95/100 audited outputs (Table 2). MuSiQue uses official EM/F1. Appendix C details timing and aggregation; Appendix C.2 specifies the judge and scoring procedures.

RQ1: Does GraphSpec realize KAP's execution benefits on SPIQA? Table 3 shows that, as context length increases from 4K to 128K, the retained-KV ratio s falls from 39.2% to 5.5%, while speedup over full-context decoding rises from 1.067× to 1.170×. GraphSpec's pass rates differ from full-context decoding by at most 0.3 percentage points at five of the six context lengths and are 1.9 points higher at 64K. These results show that KAP can translate frontend knowledge selectivity into effective runtime KV access, improving decoding efficiency without shortening the logical prompt while maintaining comparable answer quality.

RQ2: Mechanism isolation. We compare GraphSpec on SPIQA with prompt-compression baselines and closely related proposal baselines to examine the effects of different mechanisms. Physical concat reserializes planner-selected evidence; Random concat samples passages, and Local window retains context head/tail windows. RAPID-core re-encodes selected evidence into a shortened proposal context (Chen et al., 2025); MagicDec-SnapKV selects proposal KV states using SnapKV's attention-based importance scores (Sadhukhan et al., 2025; Li et al., 2024). Since RAPID and MagicDec use different inference stacks, we implement their proposal mechanisms in vLLM to control for framework differences, matching verification policies within each comparison. Appendix D details these implementations.

Table 4: **RQ2: Mechanism isolation on SPIQA.** s (%): retained KV relative to source-context state; TPS: decode tokens/s; Pass (%): answer pass rate.

	4K			32K			128K		
Method	s	TPS	Pass	s	TPS	Pass	s	TPS	Pass
Full-context	100.0	51.91	80.0	100.0	48.68	86.9	100.0	41.10	84.6
Prompt-compression baselines									
Physical concat	20.1	52.55	75.1	3.9	51.79	75.7	1.0	53.54	76.5
Random concat	39.4	52.13	65.5	16.0	50.51	56.5	7.8	52.44	42.2
Local window	38.2	51.27	66.9	18.2	54.02	56.1	9.8	52.05	47.6
GraphSpec (Top-3)	39.2	55.38	**80.2**	16.0	52.59	**87.0**	5.5	48.09	**84.3**
Closely related proposal baselines									
RAPID-core (Top-1)	39.4	21.08	82.3	16.8	19.74	89.8	8.0	17.48	86.2
GraphSpec (Top-1)	39.4	**31.06**	82.8	16.0	**29.69**	89.6	7.8	**25.39**	86.4
MagicDec-SnapKV (Top-3)	39.5	24.52	81.5	18.0	20.69	88.1	7.8	13.65	85.6
GraphSpec (Top-3)	39.2	**55.38**	80.2	16.0	**52.59**	87.0	5.5	**48.09**	84.3

Table 4 shows that GraphSpec achieves higher pass rates than all three prompt-compression baselines: at 128K, it reaches 84.3%, versus 76.5% for Physical concat, 42.2% for Random concat, and 47.6% for Local window. GraphSpec (Top-1) is included to match RAPID-core's verification policy. Across the three reported lengths, GraphSpec achieves 1.45–1.50× the TPS of RAPID-core under matched Top-1 verification and 2.26–3.52× that of MagicDec-SnapKV under matched Top-3 verification; Appendix Table 13 reports the full sweep. Together, these results demonstrate that GraphSpec combines reduced proposal-side KV exposure and efficient decoding with answer quality comparable to full-context decoding, substantiating KAP's approach of translating frontend knowledge priorities into runtime execution.

RQ3: Transferability across configurations. To evaluate KAP's transferability beyond the primary experimental setting, we test runtime-contract reuse across knowledge-selection frontends, base models, QA workloads, and verification policies. All four transfers reuse KAP's runtime contract and GraphSpec's compiler–executor architecture within the same vLLM serving framework. Appendix Table 14 provides the complete SPIQA sweeps.

Frontend transfer. We replace Graph-RAG with BGE-M3, retaining Qwen3-VL-32B, SPIQA, and Top-3 verification. The compiler maps BGE-M3's scored passages into runtime access plans for the existing backend executor. Table 5 shows stable decoding throughput and high answer pass rates across the reported context lengths, alongside reduced KV retention and high acceptance (Appendix Table 10). These results support KAP's frontend transferability: separating knowledge selection from runtime execution enables different frontends to reuse the same backend while maintaining decoding efficiency and answer quality.

Table 5: **RQ3 frontend transfer on SPIQA.** Graph-RAG versus BGE-M3 priors.

	Graph-RAG		BGE-M3	
Length	TPS	Pass rate	TPS	Pass rate
4K	55.38	80.2%	47.25	81.6%
32K	52.59	87.0%	47.47	88.9%
128K	48.09	84.3%	43.50	86.8%

Table 6: **RQ3 model transfer on SPIQA.** Llama-3.1-8B efficiency and answer quality.

	Full-context	GraphSpec				Pass rate	
Length	TPS	TPS	s	h	Speedup	Full-context	GraphSpec
4K	132.03	123.03	**38.5%**	94.6%	0.93×	83.6%	83.3%
32K	115.42	**116.84**	**15.8%**	95.5%	**1.01**×	85.7%	86.4%
128K	78.66	**105.41**	**7.7%**	95.6%	**1.34**×	75.9%	76.8%

Model transfer. We replace Qwen3-VL-32B with Llama-3.1-8B for both proposal and verification, retaining Graph-RAG, SPIQA, and Top-3 verification. Table 6 shows comparable pass rates and 1.34× speedup at 128K. GraphSpec's throughput is less sensitive to context length than full-context decoding, with greater speedups at longer contexts. These results show that KAP's execution benefits extend to another model family, supporting reuse of the runtime contract across models.

Table 7: **RQ3 workload transfer to MuSiQue.** Decode TPS, plan statistics, and official EM/F1. The complete sweep appears in Appendix Table 9.

Panel A: Decode throughput and plan statistics

	4K			32K			128K		
Method	TPS	s (%)	h (%)	TPS	s (%)	h (%)	TPS	s (%)	h (%)
Full-context	48.54	100.0	–	45.30	100.0	–	36.82	100.0	–
RAPID-core	22.90	28.6	69.0	22.15	4.6	65.8	18.96	1.5	53.2
MagicDec-SnapKV	25.21	30.6	89.1	21.00	9.7	86.6	13.75	6.3	83.3
GraphSpec	**51.65**	29.4	**93.5**	**50.63**	9.0	**93.8**	**42.11**	5.1	**89.5**

Panel B: Official answer EM / F1 (%)

Method	4K	32K	128K
Full-context	45.0/52.6	40.9/50.5	26.5/38.4
RAPID-core	45.0/51.2	41.4/50.2	27.0/38.5
MagicDec-SnapKV	42.0/50.8	42.4/50.7	27.5/38.8
GraphSpec	42.5/51.5	39.4/49.0	27.5/38.6

Workload transfer. We replace SPIQA with MuSiQue-Ans, retaining Graph-RAG, Qwen3-VL-32B, and Top-3 verification. GraphSpec achieves higher TPS than all baselines with F1 close to full-context decoding (Table 7). Speedup reaches 1.14× at 128K, slightly below SPIQA despite lower KV retention. MuSiQue's connected multi-hop reasoning may make selective proposals more sensitive to evidence coverage, consistent with the observed lower acceptance and weaker verification amortization. These results support reuse of KAP's runtime contract across QA workloads, with gains shaped by workload-specific evidence requirements.

Verification-policy transfer. We replace Top-3 with Top-1 verification, retaining Graph-RAG, Qwen3-VL-32B, and SPIQA. Top-3 achieves 1.77–1.89× Top-1's TPS with pass rates at most 2.6 points lower (Table 8). These results support verification-policy transfer within KAP and motivate Top-3 as our default quality–efficiency choice.

Together, these transfers support KAP's separation of frontend knowledge logic from backend execution: a common runtime contract makes knowledge priorities actionable in physical KV access across configurations while accommodating distinct quality–efficiency trade-offs within vLLM.

Table 8: **RQ3 verification-policy transfer on SPIQA.** Graph-RAG priors; s, h, and Pass are percentages.

	Top-1				Top-3			
Length	TPS	s	h	Pass	TPS	s	h	Pass
4K	31.06	39.4	52.1	82.8	**55.38**	**39.2**	**94.7**	80.2
32K	29.69	16.0	52.1	89.6	**52.59**	16.0	**94.7**	87.0
128K	25.39	7.8	48.4	86.4	**48.09**	**5.5**	**94.5**	84.3

RQ4: Cost-model consistency. The conditional cost model predicts that a larger acceptance–retention margin $h - s$ strengthens selective proposal's KV-access advantage.

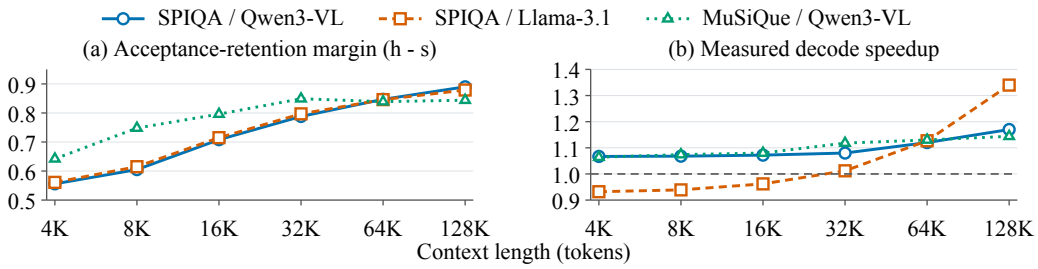


Figure 4: **RQ4 cost-model consistency.** Acceptance–retention margins (left) and measured speedups (right). The horizontal dashed line marks throughput parity.

Figure 4 shows generally widening $h - s$ margins alongside increasing speedups across all three sweeps. This agrees with the cost model's directional prediction that larger acceptance–retention margins favor selective execution at longer contexts. These trends support the model's explanation of KAP's conditional decoding gains in GRAPHSPEC. Complete values appear in Appendix Table 15.

5 RELATED WORK

Knowledge-selection methods improve which evidence reaches an LLM (Lewis et al., 2020; Karpukhin et al., 2020; Xiong et al., 2021; Edge et al., 2024; Jiménez Gutiérrez et al., 2024), while serving systems optimize scheduling and cache reuse (Yu et al., 2022; Kwon et al., 2023; Zheng et al., 2024; Jin et al., 2025; Yao et al., 2025). Prompt compression changes serialized input (Jiang et al., 2023; 2024b; Li et al., 2023); sparse attention and KV methods instead infer model-internal access (Zhang et al., 2023; Li et al., 2024; Cai et al., 2025; Tang et al., 2024; Jiang et al., 2024a). KAP contributes the missing cross-layer contract: structured frontend decisions become an explicit backend-consumed IR. GraphSpec realizes this contract within speculative long-context decoding, whose alternatives use sparse-KV, shortened-context, quantized, or learned proposal paths (Leviathan et al., 2023; Sadhukhan et al., 2025; Sun et al., 2024; Chen et al., 2025; Tiwari et al., 2025; Yang et al., 2026). Appendix D provides a mechanism taxonomy and positions the VLLM-aligned controls used here.

6 CONCLUSION

We introduced KAP to translate frontend knowledge priorities into runtime access plans. GRAPHSPEC demonstrates proposal-side KV savings and conditional decoding gains within VLLM. KAP represents a first step toward LLM systems that jointly optimize knowledge selection and runtime execution. More broadly, its runtime access contract opens a design space for new selection methods, planning strategies, and serving architectures that jointly advance quality and efficiency.

AI USE STATEMENT

We used generative AI tools to assist with writing and language polishing, literature search, and code implementation. The authors take full responsibility for the final manuscript, implementation, experimental results, and conclusions.

Reproducibility statement. Appendices A–E document the implementation, cost derivation, evaluation protocols and complete results, baseline controls, and judge prompt.

REFERENCES

Shuai Bai et al. Qwen3-VL technical report. *arXiv preprint arXiv:2511.21631*, 2025. URL https://arxiv.org/abs/2511.21631.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, and Wen Xiao. PyramidKV: Dynamic KV cache compression based on pyramidal information funneling. In *Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=ayi7qezU87.

Guanzheng Chen, Qilong Feng, Jinjie Ni, Xin Li, and Michael Qizhe Shieh. RAPID: Long-context inference with retrieval-augmented speculative decoding. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 8093–8107. PMLR, 2025. URL https://proceedings.mlr.press/v267/chen25s.html.

Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. M3-Embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 2318–2335, 2024. doi: 10.18653/v1/2024.findings-acl.137. URL https://aclanthology.org/2024.findings-acl.137/.

Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. From local to global: A graph RAG approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024. URL https://arxiv.org/abs/2404.16130.

Aaron Grattafiori et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. URL https://arxiv.org/abs/2407.21783.

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, 2023. doi: 10.18653/v1/2023.emnlp-main.825. URL https://aclanthology.org/2023.emnlp-main.825/.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention. In *Advances in Neural Information Processing Systems*, 2024a. URL https://arxiv.org/abs/2407.02490.

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, 2024b. doi: 10.18653/v1/2024.acl-long.91. URL https://aclanthology.org/2024.acl-long.91/.

Bernal Jiménez Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. HippoRAG: Neurobiologically inspired long-term memory for large language models. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.14831.

Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Shufan Liu, Xuanzhe Liu, and Xin Jin. RAGCache: Efficient knowledge caching for retrieval-augmented generation. *ACM Transactions on Computer Systems*, 44(1):2:1–2:27, 2025. doi: 10.1145/3768628. URL https://doi.org/10.1145/3768628.

Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pp. 6769–6781, 2020. doi: 10.18653/v1/2020.emnlp-main.550. URL https://aclanthology.org/2020.emnlp-main.550/.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023. URL https://doi.org/10.1145/3600006.3613165.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 19274–19286, 2023. URL https://proceedings.mlr.press/v202/leviathan23a.html.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, 2020. URL https://arxiv.org/abs/2005.11401.

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, 2023. 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*, 2024. URL https://arxiv.org/abs/2404.14469.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638. URL https://aclanthology.org/2024.tacl-1.9/.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems*, 2023. URL https://arxiv.org/abs/2211.05102.

Shraman Pramanick, Rama Chellappa, and Subhashini Venugopalan. SPIQA: A dataset for multimodal question answering on scientific papers. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-3773. URL https://doi.org/10.52202/079017-3773.

Ranajoy Sadhukhan, Jian Chen, Zhuoming Chen, Vashisth Tiwari, Ruihang Lai, Jinyuan Shi, Ian En-Hsu Yen, Avner May, Tianqi Chen, and Beidi Chen. MagicDec: Breaking the latency-throughput tradeoff for long context generation with speculative decoding. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=CS2JWaziYr.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024. doi: 10.1016/j.neucom.2023.127063. URL https://doi.org/10.1016/j.neucom.2023.127063.

Hanshi Sun, Zhuoming Chen, Xinyu Yang, Yuandong Tian, and Beidi Chen. TriForce: Lossless acceleration of long sequence generation with hierarchical speculative decoding. In *Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=HVK6nl3i97.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context LLM inference. In *International Conference on Machine Learning*, 2024. URL https://arxiv.org/abs/2406.10774.

Rishabh Tiwari, Haocheng Xi, Aditya Tomar, Coleman Richard Charles Hooper, Sehoon Kim, Maxwell Horton, Mahyar Najibi, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. QuantSpec: Self-speculative decoding with hierarchical quantized KV cache. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 59668–59686. PMLR, 2025. URL https://proceedings.mlr.press/v267/tiwari25b.html.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. MuSiQue: Multihop questions via single-hop question composition. *Transactions of the Association for Computational Linguistics*, 10:539–554, 2022. doi: 10.1162/tacl_a_00475. URL https://aclanthology.org/2022.tacl-1.31/.

Wenhan Xiong, Xiang Lorraine Li, Srinivasan Iyer, Jingfei Du, Patrick Lewis, William Wang, Yashar Mehdad, Wen-tau Yih, Sebastian Riedel, Douwe Kiela, and Barlas Oğuz. Answering complex open-domain questions with multi-hop dense retrieval. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=EMHoBG0avc1.

Penghui Yang, Cunxiao Du, Fengzhuo Zhang, Haonan Wang, Tianyu Pang, Chao Du, and Bo An. LongSpec: Long-context lossless speculative decoding with efficient drafting and verification. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1826–1844. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.83. URL https://aclanthology.org/2026.acl-long.83/.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, 2025. doi: 10.1145/3689031.3696098. URL https://doi.org/10.1145/3689031.3696098.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation*, 2022. URL https://www.usenix.org/conference/osdi22/presentation/yu.

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. H_2O: Heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2306.14048.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2312.07104.

A GRAPHSPEC IMPLEMENTATION DETAILS

A.1 FRONTEND PRIORS AND PLAN COMPILATION

GraphSpec treats the Graph-RAG frontend as a source of graph-derived knowledge priors rather than as part of the serving runtime. Our hybrid frontend combines vector retrieval, reranking, graph-node scoring, and PPR-based expansion. Retrieved units may carry node type, passage provenance, graph neighborhood, anchor location, and other annotations. Graph-RAG is one point in the broader KAP frontend space: application-specific structures enter serving as compilable priors rather than backend-specific execution logic.

Algorithm 1 constructs one runtime access plan from Graph-RAG priors and vLLM cache metadata. Compilation assumes that full-context prefill of X has populated the KV cache referenced by $\mathcal{P}_{\mathrm{KV}}$. The compiler grounds evidence in the complete prompt, selects semantic units under budget, and preserves their logical positions. It then constructs the selected-KV view and runtime object mapping M, emitting $\mathcal{A}$ with the access and verification policies.

Selection and frontend adapters. Budgets are expressed in prompt tokens before cache-page mapping. The Graph-RAG adapter grounds graph anchors and query-related evidence spans in the serialized prompt. Overlapping spans contribute each token once; the instruction prefix and question tail remain visible. The BGE-M3 adapter instead ranks passages by retrieval score, skips duplicates, and selects whole passages that fit the token budget. If no passage fits, it retains the highest-ranked passage. Selected passages are restored to source order and mapped to their original token intervals; non-passage prompt tokens remain visible. Both adapters produce a full-length token-visibility mask combining selected spans with protected prompt tokens. The algorithm uses this mask to resolve cache objects, preserve logical positions, and construct slot mappings. Thus, prior-specific selection changes U, while the construction of M and the executor interface remain shared.

Algorithm 1 GraphSpec KAP compiler. Runtime access-plan compilation for vLLM.

procedure GRAPHSPECCOMPILE($\pi_{\mathrm{graph}}, X, \mathcal{P}_{\mathrm{KV}}, B, \rho_{\mathrm{verify}}$)
Input: Graph-RAG priors π_{graph}, logical prompt X, paged-KV metadata $\mathcal{P}_{\mathrm{KV}}$, budget B, verification policy ρ_{verify}
Output: runtime access plan $\mathcal{A}$
  spans $\leftarrow$ GROUNDEVIDENCE(π_{graph}, X)
  tokenSpans $\leftarrow$ ALIGNTOKENSPANS(spans, X)
  $U \leftarrow$ SELECTEVIDENCEUNDERBUDGET(tokenSpans, π_{graph}, B)
  mask $\leftarrow$ BUILDVISIBILITYMASK(U, tokenSpans, X)
  $pos \leftarrow$ PRESERVELOGICALPOSITIONS(mask, X)
  $\rho_{\mathrm{access}} \leftarrow$ BUILDPROPOSALINTENT(U, mask, B)
  $V \leftarrow$ RESOLVESELECTEDKVOBJECTS(mask, $\mathcal{P}_{\mathrm{KV}}$)
  slots $\leftarrow$ ASSIGNCOMPACTSLOTS(V)
  blockTable $\leftarrow$ BUILDBLOCKTABLE(V, slots, $\mathcal{P}_{\mathrm{KV}}$)
  slotMap $\leftarrow$ BUILDSLOTMAPPING(mask, slots)
  view $\leftarrow$ BUILDSELECTEDKVVIEW(V, mask, blockTable, slotMap, pos)
  $M \leftarrow$ {tokenSpans, mask, view, blockTable, slotMap}
  $\mathcal{A} \leftarrow$ EMITACCESSPLAN($U, M, pos, \rho_{\mathrm{access}}, \rho_{\mathrm{verify}}$)
  return $\mathcal{A}$
end procedure

A.2 SELECTED-KV VIEW MATERIALIZATION

The compiler materializes ρ_{access} as a selected-KV view over vLLM's paged cache. Semantic selections are resolved into token-span and page mappings, and selected pages are compacted in access order. Only physical placement changes: visibility masks, prompt-length metadata, block tables, and slot mappings retain original logical positions. Block tables map plan-visible logical blocks to compact pages; slot mappings place newly scheduled tokens in the corresponding physical slots.

During proposal, $\mathcal{A}$ exposes the objects identified by M under ρ_{access}; during verification, ρ_{verify} selects full-context metadata. Both paths retain the original logical positions, preserving positional semantics across proposal and verification. The plan thus carries evidence grounding, runtime object mappings, positions, and policies together, with the selected-KV view providing the mapped state for proposal execution.

A.3 Executor transition and verification policies

Each proposal step is a decode-like forward with $q_{\text{prop}} = 1$. The executor supplies the proposed token, its full-sequence position, compact block table, slot mapping, and attention metadata. Model parameters match the full-context verifier; only prompt-side KV visibility differs. After k candidates, verification uses a contiguous $q_{\text{verify}} = k + 1$ window: a verifier-approved carry token c followed by $\hat{\mathbf{x}}_{1:k}$. The state transition is

$$(m, c^{+}, \mathrm{KV}^{\text{full},+}) = \mathcal{V}_{\theta}^{\text{full}}\left(\mathrm{KV}^{\text{full}}, c, \hat{\mathbf{x}}_{1:k}; \rho_{\text{verify}}\right), \tag{3}$$

where $m \in \{0, \ldots, k\}$ is the length of the consecutively accepted prefix. The transition returns this length, the next approved carry token, and committed full-context state, avoiding a separate resynchronization pass.

Top-1 verification accepts a proposed token only if it matches the full-context verifier's highest-probability token. Top-3 tolerant verification accepts candidates within the verifier's top three; it is an empirical quality–efficiency policy, not an exact or lossless guarantee. Full-context guidance is therefore applied before commitment under both policies, while output equivalence remains policy-specific.

A.4 Runtime-efficient plan execution

GraphSpec integrates proposal and verification with vLLM's query-length-specific uniform-decode dispatch. For a fixed proposal length k, proposal uses single-token queries, while verification uses $(k+1)$-token windows. Separate graph tags distinguish compact proposal execution from dense verification, allowing both paths to use the CUDA-graph infrastructure without conflating their runtime bindings.

The compact proposal path uses capacity-bucketed KV workspaces. Each reusable workspace retains its allocated KV buffers and block table, providing stable memory addresses for graph replay. Within a proposal cycle, compatible attention metadata is constructed once and refreshed in place, while sequence-length and slot-mapping buffers are updated as decoding advances. Workspace capacity and scalar sequence-length bounds remain stable; actual sequence lengths, token positions, and buffer contents continue to reflect the evolving execution state. This separation between stable execution structures and changing state reduces repeated allocation and metadata construction without freezing the logical progress of decoding.

When compact execution is unavailable, proposal can use the token-exact sparse-mask attention path. Attention metadata that does not support in-place reuse is rebuilt through the backend's metadata construction path.

B GraphSpec Cost-Model Derivation

This analysis applies to GraphSpec's selected-KV proposal and full-context verification executor, rather than to KAP executors in general. We adopt a memory-bound view of autoregressive decoding (Pope et al., 2023; Kwon et al., 2023), treat context length as locally constant within a proposal–verification cycle, and approximate execution cost by model-weight and KV reads in a common effective read-cost unit. Scheduler, metadata, synchronization, and launch overheads affect realized performance but are abstracted from the idealized break-even condition.

Notation and per-step costs. Let $W_{\text{main}} > 0$ denote the model-weight read cost of one full-context decode step, and let $KV \geq 0$ denote its historical full-context KV read cost. Here, KV is a scalar cost, not a cache tensor.

One full-context decode step has cost

$$\mathrm{Cost}_{\mathrm{full}} \approx W_{\mathrm{main}} + KV. \tag{4}$$

Let $\mathcal{A}_{\mathrm{GS}}$ be a GraphSpec runtime access plan in the format of Equation 1. Its retained-state ratio s and proposal acceptance h are

$$\begin{aligned} s = s(\mathcal{A}_{\mathrm{GS}}) &= \frac{\text{prompt-KV state retained for proposal}}{\text{full prompt-KV state}}, \\ h = h(\mathcal{A}_{\mathrm{GS}}) &= \frac{\text{accepted proposed tokens}}{\text{total proposed tokens}}. \end{aligned} \tag{5}$$

Both ratios lie in $[0, 1]$. We approximate the proposal-side historical KV read cost by sKV; this relates retained-state size to the model's read-cost estimate rather than treating s as measured memory traffic. Because proposal and verification use the same model weights, one proposal step costs approximately $W_{\mathrm{main}} + sKV$.

Equal-output cycle costs. Let $k \geq 1$ be the fixed integer number of candidates generated per proposal–verification cycle. With h counting tokens in accepted prefixes, a complete cycle accepts kh proposed tokens on average and commits one additional verifier-approved carry token. It therefore produces $1 + kh$ tokens in expectation, abstracting from end-of-generation truncation. We compare the baseline cost of producing this same expected number of tokens, $\mathrm{Cost}_{\mathrm{base}}$, with the cost of one GraphSpec cycle, $\mathrm{Cost}_{\mathrm{GraphSpec}}$. Under the read-cost model,

$$\mathrm{Cost}_{\mathrm{base}} = (1 + kh)(W_{\mathrm{main}} + KV), \tag{6}$$

$$\mathrm{Cost}_{\mathrm{GraphSpec}} = k(W_{\mathrm{main}} + sKV) + (W_{\mathrm{main}} + KV). \tag{7}$$

The verification term approximates the $(k+1)$-token window as one effective weight and historical-KV read, shared across the query window rather than paid independently for each candidate. This is a memory-traffic approximation for a short verification window, not an assumption that its arithmetic work equals single-token decoding.

Deriving the break-even condition. Positive speedup under this model requires a smaller cost for the same expected output:

$$\mathrm{Cost}_{\mathrm{GraphSpec}} < \mathrm{Cost}_{\mathrm{base}}.$$

Substituting the two cycle costs and expanding the baseline gives

$$\begin{aligned} k(W_{\mathrm{main}} + sKV) + (W_{\mathrm{main}} + KV) &< (1 + kh)(W_{\mathrm{main}} + KV) \\ &= (W_{\mathrm{main}} + KV) + kh(W_{\mathrm{main}} + KV). \end{aligned}$$

Subtracting the shared term $W_{\mathrm{main}} + KV$ from both sides yields

$$k(W_{\mathrm{main}} + sKV) < kh(W_{\mathrm{main}} + KV).$$

Since $k > 0$, division by k preserves the inequality:

$$W_{\mathrm{main}} + sKV < h(W_{\mathrm{main}} + KV).$$

Expanding the right-hand side and collecting weight and KV terms gives

$$\begin{aligned} W_{\mathrm{main}} + sKV &< hW_{\mathrm{main}} + hKV, \\ hKV - sKV &> W_{\mathrm{main}} - hW_{\mathrm{main}}, \\ (h - s)KV &> (1 - h)W_{\mathrm{main}}. \end{aligned} \tag{8}$$

For $h > s$, division by the positive margin $h - s$ gives

$$KV > \frac{(1 - h)W_{\mathrm{main}}}{h - s}.$$

Thus the critical full-context KV read-cost threshold, denoted by KV_{crit}, is

$$KV_{\mathrm{crit}} \approx \frac{(1 - h)W_{\mathrm{main}}}{h - s}. \tag{9}$$

This is Equation 2 in the main text and explains GraphSpec's long-context orientation: larger historical KV cost can amortize same-model proposal and full-context verification when acceptance remains high and retention falls.

Interpreting the boundary. The quantities s and h are coupled: retaining more state can improve acceptance, while stronger selectivity may reduce it. Thus, decoding gains depend on both selectivity and acceptance, rather than sparsity alone.

Proposal length k cancels algebraically from the ideal break-even condition for given s and h. Changing k can change acceptance and verification efficiency, so the cancellation does not imply invariant realized speedup. The same boundary applies to each verification policy using its measured acceptance h; it is a conditional cost criterion, not an output-equivalence guarantee.

C Experimental Protocol Details

C.1 Serving measurements

Context construction and paired execution. For each query, the answer and core supporting evidence remain fixed while a fixed, query-specific distractor list contributes progressively longer prefixes to the 4K–128K sweep. Full-context and GraphSpec receive the same complete prompt; prompt-compression baselines receive the shortened prompt produced by their selection strategy. Paired methods otherwise share model weights, tokenization, decoding settings, software, and hardware.

Prompt-compression controls. All three controls retain the question and task instructions, but reserialize only their selected context. They use standard vLLM decoding on the shortened prompts rather than GraphSpec's plan-guided proposal and verification.

Aggregation and timing. Decode TPS is aggregated as total generated tokens divided by total decode time, not as the mean of per-request TPS. Empty generations are removed from systems aggregates. BGE-M3 TPS additionally excludes five length-truncated generations under a uniform rule across context lengths; these outputs remain in the quality evaluation. For requests with multiple output tokens, TPOT divides post-first-token decode time by output tokens after the first. E2E latency is TTFT plus decode time. The reported s and h are token-weighted within method–length buckets. Bootstrap intervals resample paired requests and therefore capture workload-sampling uncertainty rather than run-to-run variance.

Frontend evidence is frozen before timing. TTFT begins at frozen-request submission and includes runtime plan compilation and selected-KV materialization for GraphSpec; retrieval is excluded for every method. Profiling is disabled during timed runs, and throughput covers the complete executor rather than isolated forward passes.

C.2 Answer-quality evaluation

LLM judge inputs and decoding. All LLM-based evaluations on SPIQA and MuSiQue use the same Qwen3-VL-32B-Instruct judge across methods and configurations. It evaluates saved generations using the question, generated answer (Answer A), and reference answer (Answer B). The evaluation prompt contains these texts but no method identifier or runtime statistics. The judge is queried through the chat-completions interface with temperature 0.1 and a 512-token output limit. Appendix E gives the full system message and scoring rubric.

Scoring and aggregation. The prompt assigns weights to fact consistency (40%), information completeness (30%), logical structure (20%), and expression quality (10%), and requests a JSON response containing an overall `score`, an explanation, dimension scores, and key findings. The evaluator reads the overall `score` in $[0, 1]$; dimension scores and explanations are retained for inspection. Pass rate is the fraction of successfully scored answers with `score`≥ 0.5 within each method–length bucket. Each answer has one successful judgment; the evaluator retains completed scores and retries failed requests or invalid responses for up to three attempts, recording evaluation status separately from the score.

Independent human audit. The 0.5 threshold was fixed before the human audit. Three annotators independently assessed correctness on 100 randomly sampled SPIQA outputs without access to

judge scores. Their majority vote defined the human reference label; thresholded judge decisions agreed on 95 samples (Table 2). Evidence-recall diagnostics separately measure coverage of answer, rationale, and caption evidence.

MuSiQue official answer metrics. The reported MuSiQue EM/F1 is computed independently of the LLM judge from short-answer generations using the official `AnswerMetric` implementation. Normalization lowercases the text, removes punctuation and the articles *a*, *an*, and *the*, and collapses whitespace. EM compares normalized strings; F1 uses multiset token overlap. Each metric takes the maximum over the reference answer and its provided aliases, then averages request-level scores within a context-length bucket. All methods use the same retained question–length keys after excluding empty full-context generations.

C.3 Supplementary results for RQ2–RQ4

Table 9: **Complete RQ3 workload-transfer results on MuSiQue.** Decode TPS, plan statistics, and official EM/F1. Table 7 reports the 4K, 32K, and 128K settings.

Panel A: Decode TPS

Method	4K	8K	16K	32K	64K	128K
Full-context	48.54	48.36	47.28	45.30	42.27	36.82
RAPID-core	22.90	22.77	22.57	22.15	21.26	18.96
MagicDec-SnapKV	25.21	24.34	23.18	21.00	18.09	13.75
GraphSpec	51.65	51.95	51.08	50.63	47.78	42.11

Panel B: Retained-KV ratio / acceptance (s/h, %)

Method	4K	8K	16K	32K	64K	128K
RAPID-core	28.6/69.0	16.7/68.2	8.9/67.6	4.6/65.8	2.9/62.3	1.5/53.2
MagicDec-SnapKV	30.6/89.1	20.2/88.4	13.9/87.9	9.7/86.6	7.8/86.8	6.3/83.3
GraphSpec	29.4/93.5	19.1/93.9	13.3/92.9	9.0/93.8	7.7/91.6	5.1/89.5

Panel C: Official answer EM / F1 (%)

Method	4K	8K	16K	32K	64K	128K
Full-context	45.0/52.6	43.5/51.0	40.7/49.9	40.9/50.5	35.0/43.8	26.5/38.4
RAPID-core	45.0/51.2	43.5/50.6	40.2/48.7	41.4/50.2	34.0/42.4	27.0/38.5
MagicDec-SnapKV	42.0/50.8	41.5/49.7	40.7/49.1	42.4/50.7	33.5/42.8	27.5/38.8
GraphSpec	42.5/51.5	41.0/49.3	38.7/48.1	39.4/49.0	33.0/42.5	27.5/38.6

Table 10: **Frontend-transfer plan statistics on SPIQA.**

	Graph-RAG		BGE-M3	
Length	s	h	s	h
4K	39.2%	94.7%	37.5%	89.7%
8K	34.3%	94.8%	33.8%	90.4%
16K	24.1%	94.9%	24.4%	92.7%
32K	16.0%	94.7%	16.3%	91.9%
64K	10.8%	95.4%	12.2%	94.6%
128K	5.5%	94.5%	7.9%	92.7%

D Mechanism Taxonomy and Evaluation Controls

Table 16 distinguishes KAP from the nearest long-context speculative systems by the source of selection, the representation that enters runtime execution, and the contract between logical context and physical access. The table is a mechanism taxonomy rather than a performance ranking.

RAPID-core: shortened-context proposal. RAPID's RAG drafter materializes retrieved evidence as a shortened proposal prompt (Chen et al., 2025). We implement this mechanism within

Table 11: **Top-1 comparison of RAPID-core and GraphSpec on SPIQA.** s, h, and Pass are percentages; speedup is GraphSpec TPS / RAPID-core TPS.

	RAPID-core				GraphSpec				
Length	TPS	s	h	Pass	TPS	s	h	Pass	Speedup
4K	21.08	39.4%	61.3%	82.3%	31.06	39.4%	52.1%	82.8%	1.47×
8K	20.97	33.4%	60.6%	83.8%	31.29	34.4%	53.1%	84.1%	1.49×
16K	20.30	23.6%	57.3%	88.9%	30.65	24.2%	52.7%	88.2%	1.51×
32K	19.74	16.8%	54.9%	89.8%	29.69	16.0%	52.1%	89.6%	1.50×
64K	18.99	12.3%	52.7%	88.4%	28.67	12.0%	52.3%	88.4%	1.51×
128K	17.48	8.0%	48.1%	86.2%	25.39	7.8%	48.4%	86.4%	1.45×

Table 12: **Top-3 comparison of MagicDec-SnapKV and GraphSpec on SPIQA.** s, h, and Pass are percentages; speedup is GraphSpec TPS / MagicDec-SnapKV TPS.

	MagicDec-SnapKV				GraphSpec				
Length	TPS	s	h	Pass	TPS	s	h	Pass	Speedup
4K	24.52	39.5%	90.3%	81.5%	55.38	39.2%	94.7%	80.2%	2.26×
8K	23.32	34.5%	90.4%	85.0%	55.04	34.3%	94.8%	82.9%	2.36×
16K	22.81	23.2%	89.7%	88.5%	54.22	24.1%	94.9%	87.4%	2.38×
32K	20.69	18.0%	88.6%	88.1%	52.59	16.0%	94.7%	87.0%	2.54×
64K	17.69	11.2%	88.6%	89.4%	51.15	10.8%	95.4%	87.9%	2.89×
128K	13.65	7.8%	87.8%	85.6%	48.09	5.5%	94.5%	84.3%	3.52×

Table 13: **Complete RQ2 context-length sweep on SPIQA.** Panels A–C report retained-KV ratio, decode TPS, and answer pass rate. s measures served-prompt state for compression and proposal-state exposure for speculative methods, relative to source-context KV. Verification policies are shown explicitly; the policy-matched GraphSpec Top-1 comparison appears in Table 11.

Panel A: Retained-KV ratio (s)

Method	4K	8K	16K	32K	64K	128K
Full-context	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
GraphSpec (Top-3)	39.2%	34.3%	24.1%	16.0%	10.8%	5.5%
RAPID-core (Top-1)	39.4%	33.4%	23.6%	16.8%	12.3%	8.0%
MagicDec-SnapKV (Top-3)	39.5%	34.5%	23.2%	18.0%	11.2%	7.8%
Physical concat	20.1%	15.1%	8.0%	3.9%	1.5%	1.0%
Random concat	39.4%	34.4%	24.2%	16.0%	12.0%	7.8%
Local window	38.2%	31.7%	23.8%	18.2%	13.4%	9.8%

Panel B: Decode TPS

Method	4K	8K	16K	32K	64K	128K
Full-context	51.91	51.51	50.57	48.68	45.66	41.10
GraphSpec (Top-3)	55.38	55.04	54.22	52.59	51.15	48.09
RAPID-core (Top-1)	21.08	20.97	20.30	19.74	18.99	17.48
MagicDec-SnapKV (Top-3)	24.52	23.32	22.81	20.69	17.69	13.65
Physical concat	52.55	52.78	52.54	51.79	52.81	53.54
Random concat	52.13	51.66	51.73	50.51	52.20	52.44
Local window	51.27	52.07	50.88	54.02	52.23	52.05

Panel C: Answer pass rate

Method	4K	8K	16K	32K	64K	128K
Full-context	80.0%	82.6%	87.4%	86.9%	86.0%	84.6%
GraphSpec (Top-3)	80.2%	82.9%	87.4%	87.0%	87.9%	84.3%
RAPID-core (Top-1)	82.3%	83.8%	88.9%	89.8%	88.4%	86.2%
MagicDec-SnapKV (Top-3)	81.5%	85.0%	88.5%	88.1%	89.4%	85.6%
Physical concat	75.1%	74.9%	75.9%	75.7%	74.1%	76.5%
Random concat	65.5%	63.8%	63.1%	56.5%	50.1%	42.2%
Local window	66.9%	66.9%	61.8%	56.1%	47.1%	47.6%

Table 14: **Complete RQ3 sweeps on SPIQA.** Panel C entries report Top-1 / Top-3; s, h, and Pass are percentages.

Panel A: Frontend transfer

	Graph-RAG		BGE-M3	
Length	TPS	Pass rate	TPS	Pass rate
4K	55.38	80.2%	47.25	81.6%
8K	55.04	82.9%	47.99	84.8%
16K	54.22	87.4%	48.68	89.1%
32K	52.59	87.0%	47.47	88.9%
64K	51.15	87.9%	46.91	89.7%
128K	48.09	84.3%	43.50	86.8%

Panel B: Model transfer to Llama-3.1-8B

	Full-context	GraphSpec				Pass rate	
Length	TPS	TPS	s	h	Speedup	Full-context	GraphSpec
4K	132.03	123.03	38.5%	94.6%	0.93×	83.6%	83.3%
8K	130.32	122.36	33.6%	95.1%	0.94×	84.8%	85.7%
16K	125.30	120.60	23.8%	95.3%	0.96×	85.8%	85.5%
32K	115.42	116.84	15.8%	95.5%	1.01×	85.7%	86.4%
64K	100.00	112.70	11.9%	96.5%	1.13×	82.5%	81.9%
128K	78.66	105.41	7.7%	95.6%	1.34×	75.9%	76.8%

Panel C: Verification-policy transfer

Metric	4K	8K	16K	32K	64K	128K
TPS	31.06/55.38	31.29/55.04	30.65/54.22	29.69/52.59	28.67/51.15	25.39/48.09
s	39.4/39.2	34.4/34.3	24.2/24.1	16.0/16.0	12.0/10.8	7.8/5.5
h	52.1/94.7	53.1/94.8	52.7/94.9	52.1/94.7	52.3/95.4	48.4/94.5
Pass	82.8/80.2	84.1/82.9	88.2/87.4	89.6/87.0	88.4/87.9	86.4/84.3

Table 15: **RQ4 cost-model consistency: complete values.** Acceptance–retention margins and measured decode speedups plotted in Figure 4.

Setting	Metric	4K	8K	16K	32K	64K	128K
SPIQA / Qwen3-VL	$h - s$	0.556	0.605	0.708	0.788	0.847	0.890
	Speedup	1.067×	1.068×	1.072×	1.080×	1.120×	1.170×
SPIQA / Llama-3.1	$h - s$	0.561	0.615	0.715	0.797	0.846	0.879
	Speedup	0.932×	0.939×	0.962×	1.012×	1.127×	1.340×
MuSiQue / Qwen3-VL	$h - s$	0.642	0.748	0.796	0.849	0.839	0.844
	Speedup	1.064×	1.074×	1.080×	1.118×	1.130×	1.144×

vLLM using the same frozen evidence as GraphSpec and an independent short-context draft KV cache. We use Top-1 in both RAPID-core and GraphSpec to match the greedy target-decision rule, with the base model and proposal length fixed. RAPID's target-distribution transfer is omitted because it changes the target distribution; retaining the original full-context target in both vLLM paths isolates runtime representation from knowledge-transfer effects. Table 11 compares re-encoded short-context state with GraphSpec's selected state from full-context prefill, testing the execution value of runtime representation under comparable evidence and answer quality.

MagicDec-SnapKV: attention-selected proposal state. MagicDec supports SnapKV-based self-speculation (Sadhukhan et al., 2025; Li et al., 2024), providing a model-internal selection counterpart to frontend-guided access. Our control implemented in vLLM constructs proposal state from attention-derived KV salience and uses the same base model, proposal length, and Top-3 verification policy as GraphSpec. This matched Top-3 policy holds validation fixed while comparing the source and physical realization of proposal-state selection (Table 12).

These controls bring the proposal mechanisms into the common serving stack: RAPID's released code wraps Transformers generation, whereas MagicDec uses its own inference engine. The comparisons concern these implementations in vLLM, rather than reproducing the original engines or their reported performance.

Table 16: **Mechanism taxonomy for long-context proposal systems.**

Method	Selection signal	Runtime representation	Position treatment	Physical realization	Verification / contract
RAPID / RAPID-core	Retriever-ranked chunks	Shortened RAG draft prompt	Re-serialized draft positions	Short-context draft KV	Target verification; RAPID adds distribution transfer; no plan IR
MagicDec / MagicDec-SnapKV	Model-internal KV salience	Compressed draft KV	Positions inherited from cached states	Sparse-KV self-speculation	Full-KV verification; no frontend–runtime contract
TriForce	Query-to-KV chunk scores	Dynamic retrieval cache	Positions inherited from cached states	Hierarchical drafting with two models and three caches	Final full-KV verification; no frontend–runtime contract
GRAPHSPEC (KAP)	Structured frontend knowledge priors	Runtime access plan with object mappings	Logical positions are explicit plan fields	Plan-mapped selected-KV view	Policy-explicit verification/fallback under the KAP contract

E LLM JUDGE PROMPT

The following reproduces the prompt content, with line wrapping adjusted for readability. Braced fields are substituted with the corresponding request text.

System message
You are a professional text semantic similarity evaluation expert who always responds with accurate and objective assessments.
User message
You are a professional text semantic similarity evaluation expert. Please evaluate the semantic consistency between two answers based on the following criteria.
Evaluation Task
Compare the semantic consistency between two answers and provide a score between 0-1.
Input Information
Question: {`question`}
Answer A (RAG System Answer): {`prediction`}
Answer B (Reference Answer): {`reference`}
Evaluation Dimensions and Weights
Please conduct a comprehensive evaluation based on the following dimensions (100-point scale, then converted to 0-1):
1. **Core Fact Consistency (Weight: 40%)**
   - Consistency of key entities, data, and conclusions
   - Alignment of main viewpoints and arguments
   - Presence of factual conflicts or contradictions
2. **Information Completeness (Weight: 30%)**
   - Coverage of identical key information points
   - Comparability of detail description completeness
   - Presence of important information gaps or redundancies

Continued on next page

LLM judge prompt (continued)

3. **Logical Structure Consistency (Weight: 20%)**
   - Similarity in argumentation logic
   - Consistency in information organization
   - Matching of causal relationships
4. **Semantic Expression Quality (Weight: 10%)**
   - Clarity and accuracy of expression
   - Consistency in professional terminology usage

Scoring Criteria

- **0.9-1.0**: Almost identical, highly matched in core facts and logical structure
- **0.7-0.89**: Highly similar, main information consistent with minor differences
- **0.5-0.69**: Moderately similar, core viewpoints consistent but noticeable differences
- **0.3-0.49**: Partially similar, some common points but significant differences
- **0.1-0.29**: Slightly similar, only limited common information
- **0.0-0.09**: Almost unrelated or fundamental conflicts

Output Requirements

You must **strictly output in the following JSON format**:

```
{
  "score": 0.85,
  "explanation": "Brief explanation of scoring rationale",
  "dimension_scores": {
    "fact_consistency": 0.9, "information_completeness": 0.8,
    "logical_structure": 0.7, "expression_quality": 0.8
  },
  "key_findings": ["Common point 1", "Common point 2",
                   "Difference 1", "Difference 2"]
}
```

Please begin the evaluation: