Prelude: Resource-Adaptive Serving for Decision-Style LLM Inference

Anonymous Authors[1]

Abstract

Foundation-model inference is usually served through a single chat-oriented runtime even when requests have very different resource lifetimes. We present **Prelude**, a resource-adaptive serving framework for decision-style LLM inference: judges, reward models, safety classifiers, routers, rerankers, embedding models, and prompt-logprob extractors that read a prompt and return a fixed-size artifact or at most one token. Prelude classifies work into *OneShot*, *Mixed*, and *Decode* execution classes, avoiding per-request paged-decode state for fixed-output work while preserving the standard paged-KV path for open-ended generation. It also performs prefix-aware OneShot planning and uses an inference-only tokenizer, *fasttoken*, to remove CPU-side overhead from the same hot path. On H200, Prelude reaches 16,311 input tok/s on a Qwen3-0.6B prefill-only benchmark (2.08× vLLM, 3.85× SGLang) and 186.7 req/s on Qwen3-4B at concurrency 96; a multi-token decode control closes the gap to 1.03× vLLM, showing that the gains come from execution-class adaptation rather than a uniformly faster forward kernel.

1. Introduction

Foundation-model deployments are increasingly *decision-shaped*. LLM-as-a-judge systems evaluate model outputs (Zheng et al., 2023; Liu et al., 2023; Kim et al., 2024); reward models score candidate responses (Lambert et al., 2024); safety guards, routers, rerankers, embedding endpoints, and prompt-logprob extractors all read prompts and return fixed-size artifacts. These requests differ in API surface, but share one systems shape: one prompt-processing forward pass, bounded output work, and no long autoregressive lifetime.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

This creates a resource-adaptation problem for serving systems. Engines such as vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024) are optimized for chat-style decoding: allocate paged KV blocks, maintain per-sequence decode state, admit new prefills into a running decode batch, and amortize scheduler overhead across many output tokens. Decision-style requests break this amortization. They can still pay decode-shaped fixed costs, reserve future-token KV capacity, and compete with true decode requests for paged resources even though their state is short lived.

Prelude treats this as an execution-class adaptation problem. At admission, Prelude prepares each request once, classifies it into *OneShot*, *Mixed*, or *Decode*, and sends a typed `ForwardBatch` to a device executor. OneShot covers classification, embedding, prompt-logprobs, and generation with `max_tokens` ≤ 1; Decode preserves paged-KV continuous batching; Mixed admits prefill work while decode requests are running without assigning both classes the same memory lifetime.

This framing fits the AdaptFM theme because the adaptation is not a new model architecture but a runtime decision about *which resources a request is allowed to consume*. A decision request should not receive the same KV-cache lifetime, scheduler bookkeeping, and decode-loop machinery as a chat request. Conversely, a chat request should keep the mature paged-KV path where continuous batching and CUDA-graph replay are effective. Prelude makes this distinction explicit and therefore lets one serving stack handle judges, routers, embeddings, prompt-logprobs, and open-ended generation without forcing all of them through the most expensive resource contract.

This paper makes three contributions. First, we identify decision-style LLM inference as a broad resource-adaptive serving class beyond single-token generation. Second, we design execution-class aware dispatch and prefix-aware OneShot planning, which attaches cached prefixes before allocating only the uncached suffix. Third, we implement *fasttoken*, an inference-only tokenizer that drops HuggingFace `tokenizers`' training-mode and stateful-decoder overhead from the request path.

2. Decision-Style Inference

Workload shape. We use *decision-style request* to mean an LLM call that runs one prompt-processing forward pass and returns a fixed-size artifact rather than an open-ended token stream. This includes generation requests whose maximum output length is one token, zero-token prompt-logprob extraction, classification heads, embedding heads, reward-model scores, reranking scores, and routing decisions. The API-level outputs differ, but the engine-level shape is shared: fixed output work and no long-lived autoregressive decode state.

Why existing runtimes overpay. Modern engines are organized around an iteration-level scheduler. Each tick admits new prefills, runs a fused forward step, samples one token per active sequence, emits streaming deltas, and continues. The design is right for chat: the prefill is one tick, the decode is many ticks, and fixed costs such as block-table allocation, sampler state, streaming records, and finish hooks are amortized. For decision-style requests, the amortization condition fails. A one-token judge or classifier may still be registered as a sequence that could continue decoding, reserve future-token KV capacity, and enter scheduling queues built for long-lived autoregressive work.

Three bottlenecks. Prelude targets three fixed costs that become visible when output length is short. First, decode-shaped engines allocate and track per-request state even when there will be no future decode step. Second, repeated rubrics, policies, and retrieval templates create prefix reuse opportunities that should be handled before allocating request-specific suffix work. Third, CPU tokenization is no longer hidden under hundreds of decode steps; it sits directly on every request's critical path.

Design goals. The serving stack should satisfy four requirements. *G1: preserve chat throughput*. Prelude should not replace the mature continuous-batching path for open-ended generation; it should keep paged KV, decode scheduling, and graph replay where those mechanisms are useful. *G2: avoid false lifetimes*. A fixed-output request should not reserve resources for a future that cannot happen. *G3: reuse prefixes before suffix allocation*. Prefix-cache hits should be visible while the scheduler is choosing work, not only after the request has already entered a decode-shaped lifecycle. *G4: remove CPU hot-path overhead*. If a serving path is optimized down to a single GPU forward pass, tokenization and response decoding become first-order costs and must be optimized with the same seriousness as GPU kernels.

3. Design

Execution classes. Figure 1 and Table 1 show Prelude's scheduler contract. **OneShot** requests need one forward pass and no long-lived autoregressive state. **Decode** requests carry one token and a block table per active sequence, use paged KV, and are eligible for CUDA-graph replay. **Mixed** steps combine prefill chunks and running decode requests while accounting for prefill tokens, decode tokens, block tables, and recurrent slots separately. This is stronger than a boolean fast path: it gives the scheduler a resource contract for each request class.

Admission and preparation. Each generation request is normalized once: prompt tokens, effective output budget, greedy-vs-sampling mode, and logits processors are fixed before execution. Classification and embedding requests are normalized into the same scheduler interface with different output heads. The scheduler then constructs one of three typed batches. `ForwardBatch::OneShot` carries flattened prompt slices and output descriptors; `ForwardBatch::Decode` carries one token and a block table per active sequence; `ForwardBatch::Mixed` carries both, but keeps their accounting separate. This removes ambiguity at the executor boundary: device code sees the lifetime contract before launching kernels.

OneShot execution. For an admitted OneShot generation batch, Prelude flattens token slices into a varlen layout, invokes a fused prefill forward, returns logits at the final position of each sequence, samples or extracts prompt logprobs, and frees temporary GPU tensors. Classification and embedding use the same executor idea with different heads. There is no decode-loop sequence record, future-token KV reservation, or streaming-state aggregator. The same path also handles prompt-logprobs: hidden states are retained only for the rows needed to compute requested logprob payloads, then released immediately.

Mixed and Decode execution. Multi-token generation uses an iteration-level scheduler with configurable budgets for running requests, batched tokens, and total KV capacity. Pure decode steps use `ForwardBatch::Decode`; the CUDA executor first tries graph replay and falls back to eager execution. When new prefills arrive while decode is active, the scheduler can build a Mixed step: prefill chunks consume token budget, decode requests consume one token each, and output rows are processed with class-specific logic. This keeps chat performance on a familiar path while allowing fixed-output work to bypass the decode lifecycle.

Resource accounting. The important change is not merely that OneShot skips a loop. Prelude gives each class its own accounting units. OneShot batches are budgeted by

Table 1. Execution classes in Prelude. The scheduler uses the class as a resource contract rather than as a cosmetic API tag.

Class	Representative requests	Long-lived state	Scheduler action
OneShot	classify, embed, prompt-logprobs, $K \leq 1$ generation	none after one forward	batch for throughput; allocate only suffix work
Mixed	prefill chunks while decode is active	prefill state plus active decode state	account prefill tokens and decode slots separately
Decode	open-ended chat / multi-token generation	paged KV, block table, sampler state	use continuous batching and CUDA-graph replay

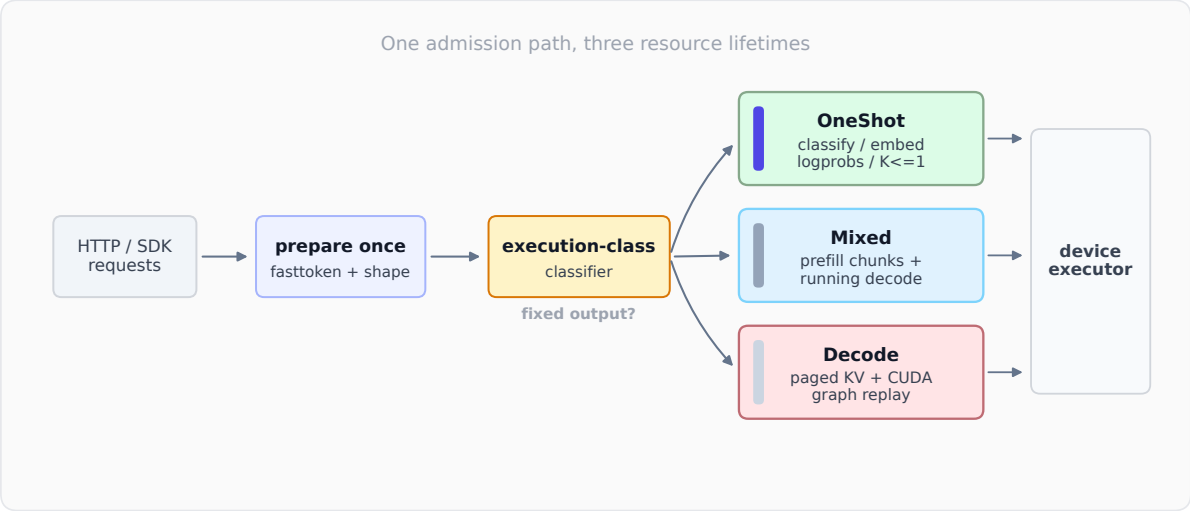


Figure 1. Prelude adapts the serving path to request resource lifetime. OneShot work uses a fixed-output path with no per-request decode state; Decode keeps the paged-KV autoregressive path; Mixed steps admit prefill chunks while decode requests run.

prompt tokens and temporary activation memory; Decode batches are budgeted by active sequence count, block-table entries, one decode token per sequence, and resident KV blocks; Mixed batches reserve both budgets but keep their lifetimes separate. A OneShot row that returns a class label or logprob vector releases its temporary tensors immediately after the forward pass. A Decode row remains resident and is re-enqueued with updated block-table state. This separation prevents high-throughput decision traffic from reducing decode capacity by occupying future-token KV reservations that it never needs.

Scheduler policy. Prelude uses a greedy admission loop with class-aware scoring. Waiting OneShot requests are grouped by model, tokenizer configuration, output head, and compatible sampling/logprob options; the group is admitted while the token budget and temporary-memory estimate permit. Decode requests are admitted according to the continuous-batching budget. Mixed steps are formed when prefill work can be admitted without evicting active decode requests. This policy is deliberately simple: the paper's claim is that making the class boundary explicit already removes a large fixed cost. More sophisticated JCT-aware or deadline-aware policies can be layered on top of the same class contract.

Prefix-aware planning. Decision workloads often reuse long rubrics, policies, or retrieval templates. Prelude matches waiting requests against a block-level prefix trie before allocating suffix work (Figure 2). Cache hits attach ref-counted shared blocks and allocate only the uncached suffix. For a cold shared prefix, the scheduler can let one leader populate the cache and refresh peers before they recompute the same prefix. This shifts prefix reuse from a decode-side optimization into the OneShot admission plan.

Waiting versus batching. Cold shared prefixes create a subtle scheduling problem: batching every waiting request with the same uncached prefix recomputes that prefix many times in one step, while waiting forever sacrifices latency. Prelude uses a lightweight rule: if a waiting request has the same uncached prefix key as a request already scheduled for prefill in the current step, it can be deferred briefly. After the leader populates the cache, waiting peers are refreshed and attached as suffix-only work. The point is not to introduce a new global queueing policy, but to avoid wasting GPU work on a prefix that will become reusable one scheduler tick later.

Fasttoken. For short-output requests, CPU tokenization is no longer hidden under hundreds of decode steps. Fast-

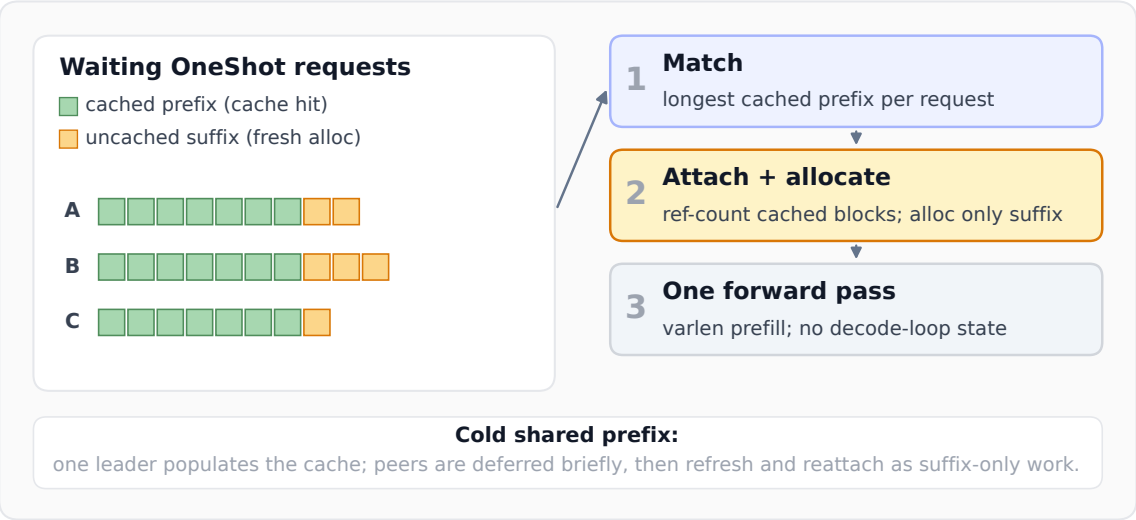


Figure 2. Prefix-aware OneShot planning. Prelude matches waiting requests against a block-level prefix trie before allocating new work. Cache hits attach ref-counted shared prefix blocks and allocate only the uncached suffix. When a cold shared prefix arrives, one leader can populate the cache and peers are refreshed as suffix-only work.

token is a drop-in inference-time tokenizer for the same `tokenizer.json`: vocab and merges are loaded once, immutable tables are read lock-free, and per-token decode is a stateless lookup. The benchmark suite checks encode/decode equality against HuggingFace `tokenizers` before reporting speed. Fasttoken intentionally drops training-mode mutation, vocabulary insertion, and the stateful streaming decoder; what remains is the inference API needed by a serving engine.

Fasttoken implementation. At load time, fasttoken parses the serialized vocabulary, merge list, special tokens, and pre-tokenizer configuration. The id-to-string table is stored as a contiguous vector indexed by token id for allocation-free decode. Encode uses immutable vocab and merge-rank tables plus per-call scratch buffers. The merge loop operates over token ids rather than heap-allocated token fragments, and the per-token streaming decode path is a direct lookup into the vocabulary arena. Since all loaded structures are immutable, the tokenizer is naturally shared across admission worker threads without a lock on each encode or decode operation.

Compatibility boundary. Fasttoken is not a training-time tokenizer. It does not support vocabulary mutation, merge insertion, or dynamic normalizer composition after startup. That boundary is intentional: serving systems need deterministic encode/decode behavior for a fixed model artifact. The benchmark harness therefore treats equality with HuggingFace `tokenizers` as a gate before any performance number is accepted. This keeps the optimization honest: if a model's serialized tokenizer configuration is outside fasttoken's supported inference subset, Prelude falls back rather than silently changing tokens.

4. Evaluation

Questions. The evaluation asks four questions. **Q1:** does execution-class aware OneShot dispatch improve fixed-output generation against vLLM and SGLang? **Q2:** does the win survive at larger model size and higher concurrency? **Q3:** does the gap disappear when the workload becomes true multi-token decode? **Q4:** does the tokenizer specialization matter once Python binding overhead is removed?

Setup. We evaluate on one NVIDIA H200 (143 GB) per engine with an AMD EPYC 9575F host. Qwen3-0.6B is used for the prefill-only and decode-control microbenchmarks; Qwen3-4B is used for the concurrency sweep. Baselines are vLLM and SGLang in official Docker images at production-default settings, with prefix caching enabled where it is the default. We do not enable speculative decoding or quantization. The request driver is `genai-bench`; tokenizer measurements use Prelude's `tokenizer_bench`. TTFT is time-to-first-token, TPOT is per-output-token latency excluding TTFT, E2E is total request latency, and RPM is requests per minute.

Measurement discipline. For serving, each engine is run in its native deployment form and receives the same prompt/output configuration. The OneShot microbenchmark uses concurrency 1 to expose per-request fixed cost; the Qwen3-4B sweep raises concurrency to show whether the specialization survives batching; the decode control increases output length to check whether the advantage comes from OneShot-specific work. For tokenization, both fast-

token and HuggingFace `tokenizers` are measured as Rust crates with no Python boundary, so the comparison isolates data structures and algorithms rather than language bindings.

OneShot specialization. Table 2 shows that Prelude reaches 16,311 input tok/s on the prefill-only benchmark, 2.08× vLLM and 3.85× SGLang. TTFT falls from 13.3 ms and 28.2 ms to 5.1 ms. Startup is also lower: 2 s for Prelude versus 44 s for vLLM and 34 s for SGLang, reflecting the native binary and lightweight runtime initialization.

Concurrency sweep. The $c = 1$ microbenchmark isolates fixed cost; the larger Qwen3-4B sweep asks whether the same advantage survives at production-style fan-in. With 512-token inputs and one output token, Prelude leads at every tested concurrency level and peaks at 186.7 req/s ($\approx$ 95,590 input tok/s) at $c = 96$, 1.39× vLLM and 1.23× SGLang. At single-request concurrency, Prelude also reduces P50/P95 latency to $15.4/21.1$ ms versus vLLM's $18.1/27.9$ ms and SGLang's $20.8/26.2$ ms (Figure 3).

Interpreting the sweep. The absolute speedup is smaller on Qwen3-4B than on Qwen3-0.6B because the GPU forward pass consumes a larger fraction of total latency. This is the expected scaling law for systems overhead: as the model becomes larger, fixed CPU and scheduler costs are amortized by heavier matrix multiplications. The fact that Prelude still leads at every concurrency level shows that the OneShot path is not only a tiny-model artifact. The peak at $c = 96$ also shows that batching alone does not erase the class mismatch; vLLM and SGLang can batch many one-token requests, but they still batch them through a decode-oriented lifecycle.

Decode control. The right half of Table 2 asks whether Prelude is simply a uniformly faster engine. On 32-token decode, the throughput gap against vLLM collapses from 2.08× to 1.03×, and TPOT is slightly slower on Prelude. The TTFT advantage remains because admission is lighter, but per-token decode throughput is essentially tied. This supports the core claim: the OneShot improvement comes from adapting the serving path to request resource lifetime.

Tokenizer hot path. Table 3 reports the same-crate comparison with no Python binding overhead. Fasttoken is faster across input lengths, with the gap widest for long BPE inputs and still meaningful for short prompts. Load time is slower (382 ms vs 121 ms) because fasttoken precomputes merge-rank tables at startup; the trade favors steady-state decision workloads. The equality tests cover encode, encode-without-special-tokens, decode, and streaming decode before timing is reported.

Takeaway. Prelude is not a replacement for decode-optimized systems on the workload those systems target. The decode control shows near parity with vLLM once each request emits 32 tokens, and Prelude's TPOT is slightly worse. The value of Prelude is that a single runtime can adapt at request granularity: fixed-output requests avoid decode-shaped state, prefix-reusing requests allocate only fresh suffix work, and open-ended generation remains on the paged-KV path. This is exactly the runtime form of resource adaptation needed by modern LLM deployments where judges, routers, embeddings, and chat completions share the same accelerators.

5. Discussion

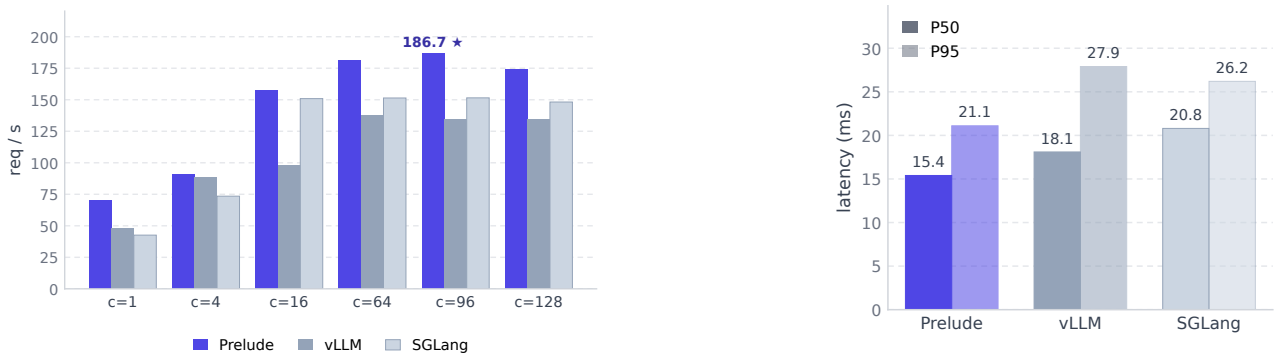
Runtime adaptation as a systems primitive. Most efficient-inference work adapts a model's arithmetic: fewer bits, fewer layers, fewer tokens, smaller experts, or cheaper decoding. Prelude adapts a different axis: the *lifetime* of runtime resources. This matters because modern model services multiplex many APIs over the same weights. A router, judge, embedding request, and chat completion may all target the same checkpoint, but they should not inherit the same sequence-state lifetime. Treating execution class as a first-class scheduler input lets the runtime adapt memory, batching, prefix reuse, and response handling without changing model weights or output semantics.

What unification buys. The framework point is important for deployment. Operators do not want a separate bespoke server for judges, another for embeddings, another for reranking, and another for chat if all of them share model weights and accelerator pools. A unified server avoids duplicated model residency and gives the scheduler a global view of contention. The key is that unification must not mean homogenization: Prelude uses one admission path, but it does not force all requests to pay the same paged-KV cost. This is why the execution-class boundary is the central abstraction rather than a collection of isolated fast paths.

Where the design is conservative. Prelude deliberately keeps the Decode path conventional. It does not claim a new attention kernel, a better CUDA graph policy, or a replacement for mature continuous batching. The decode control is therefore a sanity check: when the workload becomes true multi-token generation, Prelude should converge toward existing engines rather than win by accident. The large gain on OneShot and the near tie on Decode make the result easier to interpret: the improvement comes from removing inappropriate resource lifetimes from fixed-output work.

Table 2. End-to-end serving results. The OneShot benchmark is Qwen3-0.6B, 128-in/1-out, $c = 1$, 100 requests. The decode control is Qwen3-0.6B, 128-in/32-out, $c = 4$, 400 requests. Speedup rows are latency baseline/Prelude and throughput Prelude/baseline.

	OneShot prefill-only				Multi-token decode control			
Engine	TTFT (ms)	E2E (ms)	In tok/s	RPM	TTFT (ms)	TPOT (ms)	Out tok/s	RPM
Prelude	5.1	5.2	16,311	7,316	14.6	2.1	1,474	2,764
vLLM	13.3	13.5	7,843	3,515	30.9	1.7	1,430	2,682
SGLang	28.2	28.3	4,236	1,898	48.3	1.8	1,177	2,206
vs vLLM	2.6×	2.6×	2.08×	2.08×	2.12×	0.81×	1.03×	1.03×
vs SGLang	5.5×	5.4×	3.85×	3.85×	3.31×	0.86×	1.25×	1.25×



(a) Throughput across concurrency.

(b) P50/P95 latency.

Figure 3. Qwen3-4B / 512-in / 1-out concurrency sweep on H200. Prelude peaks at 186.7 req/s while reducing P50/P95 latency relative to the decode-oriented baselines.

Table 3. Fasttoken versus HuggingFace `tokenizers` Rust crate on Qwen3-0.6B. Speedup is HF/fasttoken.

Operation	HF (μs)	fasttoken (μs)	Speedup
Encode, 50 chars	5.7	0.4	12.9×
Encode, 500 chars	50.7	14.5	3.5×
Encode, 8K chars	601.4	27.3	22.0×
Encode, 200K chars	21,320	309.3	68.9×
Decode, 1,245 toks	70.2	29.7	2.4×
Streaming decode	0.13	0.06	2.0×
Batch encode, $b = 64$	675.7	194.0	3.5×
Concurrent encode	8.3	3.1	2.7×

6. Related Work and Scope

LLMs as decision engines. LLM-as-a-judge systems use language models to evaluate, score, or rank other model outputs, including MT-Bench/Chatbot Arena (Zheng et al., 2023), G-Eval (Liu et al., 2023), and Prometheus (Kim et al., 2024). Reward-model benchmarks study the same scalar-feedback role in alignment pipelines (Lambert et al., 2024). LLMs are also used as rerankers, routers, and embedding or classification backends (Sun et al., 2023; Ong et al., 2025; Reimers & Gurevych, 2019). These works motivate decision-style traffic; Prelude targets the serving substrate for such traffic.

Prefill-only and adaptive serving. PrefillOnly (Du et al., 2025) specializes single-token generation through hybrid prefilling and JCT-aware scheduling. LinkedIn reports large latency and throughput gains from a prefill-only path for reranking (LinkedIn Engineering, 2026). Prelude is complementary: it treats single-token generation as one member of a broader OneShot execution class that also includes classification, embedding, and prompt-logprob extraction.

LLM serving and KV reuse. Continuous-batching engines build on iteration-level scheduling (Yu et al., 2022), paged KV management (Kwon et al., 2023), cache-aware serving (Zheng et al., 2024), chunked prefill (Agrawal et al., 2024), and prefill/decode disaggregation (Zhong et al., 2024; Patel et al., 2024). Prefix-cache systems such as Prompt Cache, CacheBlend, and LMCache reuse KV state across requests (Gim et al., 2024; Yao et al., 2025; Liu et al., 2025). Prelude differs in when reuse is used: prefix hits are discovered before OneShot allocation, avoiding unnecessary decode-shaped state for fixed-output requests.

Tokenizers. HuggingFace `tokenizers` (Hugging Face, 2019) is the standard for open-weight LLMs; `tiktoken` (OpenAI, 2022) targets GPT-style BPE vocabularies. Fasttoken specializes immutable inference.

References

Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., Tumanov, A., and Ramjee, R. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 117–134, 2024. arXiv:2403.02310.

Du, K., Wang, B., Zhang, C., Cheng, Y., Lan, Q., Sang, H., Cheng, Y., Yao, J., Liu, X., Qiao, Y., Stoica, I., and Jiang, J. PrefillOnly: An inference engine for prefill-only workloads in large language model applications. *arXiv preprint arXiv:2505.07203*, 2025.

Gim, I., Chen, G., seob Lee, S., Sarda, N., Khandelwal, A., and Zhong, L. Prompt cache: Modular attention reuse for low-latency inference. In *Proceedings of Machine Learning and Systems (MLSys)*, 2024.

Hugging Face. Tokenizers: Fast state-of-the-art tokenizers optimized for research and production. https://github.com/huggingface/tokenizers, 2019. Accessed 2026-04-28.

Kim, S., Suk, J., Longpre, S., Lin, B. Y., Shin, J., Welleck, S., Neubig, G., Lee, M., Lee, K., and Seo, M. Prometheus 2: An open source language model specialized in evaluating other language models. *arXiv preprint arXiv:2405.01535*, 2024.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pp. 611–626, 2023. arXiv:2309.06180.

Lambert, N., Pyatkin, V., Morrison, J., Miranda, L., Lin, B. Y., Chandu, K., Dziri, N., Kumar, S., Zick, T., Choi, Y., Smith, N. A., and Hajishirzi, H. RewardBench: Evaluating reward models for language modeling. *arXiv preprint arXiv:2403.13787*, 2024.

LinkedIn Engineering. Scaling LLM-based ranking systems with SGLang at LinkedIn. LinkedIn Engineering Blog, 2026. Accessed 2026-04-29.

Liu, Y., Iter, D., Xu, Y., Wang, S., Xu, R., and Zhu, C. G-Eval: NLG evaluation using GPT-4 with better human alignment. *arXiv preprint arXiv:2303.16634*, 2023.

Liu, Y., Cheng, Y., Yao, J., An, Y., Chen, X., Feng, S., Huang, Y., Shen, S., Zhang, R., Du, K., and Jiang, J. LMCache: An efficient KV cache layer for enterprise-scale LLM inference. *arXiv preprint arXiv:2510.09665*, 2025.

Ong, I., Almahairi, A., Wu, V., Chiang, W.-L., Wu, T., Gonzalez, J. E., Kadous, M. W., and Stoica, I. RouteLLM: Learning to route LLMs with preference data. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025.

OpenAI. tiktoken: A fast BPE tokeniser for use with OpenAI's models. https://github.com/openai/tiktoken, 2022. Accessed 2026-04-28.

Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S., and Bianchini, R. Splitwise: Efficient generative LLM inference using phase splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132, 2024. arXiv:2311.18677.

Reimers, N. and Gurevych, I. Sentence-BERT: Sentence embeddings using siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 3982–3992, 2019. doi: 10.18653/v1/D19-1410.

Sun, W., Yan, L., Ma, X., Wang, S., Ren, P., Chen, Z., Yin, D., and Ren, Z. Is ChatGPT good at search? investigating large language models as re-ranking agents. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 14918–14937, 2023. doi: 10.18653/v1/2023.emnlp-main.923.

Yao, J., Li, H., Liu, Y., Ray, S., Cheng, Y., Zhang, Q., Du, K., Lu, S., and Jiang, J. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys)*, pp. 94–109, 2025. doi: 10.1145/3689031.3696098.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 521–538, 2022.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., and Stoica, I. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. *arXiv preprint arXiv:2306.05685*, 2023.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*, 2024. arXiv:2312.07104.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. DistServe: Disaggregating prefill and

decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 193–210, 2024. arXiv:2401.09670.

A. Additional Design Details

Serving stack. Prelude is a Rust LLM serving framework with an OpenAI-compatible HTTP server above a `ScheduledEngine`. The engine owns model weights, prepares requests, schedules work, and submits device-specific `ForwardBatch` objects to a single executor interface. The central implementation point is that execution class is decided before device launch: the scheduler does not pass a generic sequence object and ask the executor to infer its lifetime later.

Admission predicate. For generation requests, the direct-call path and the scheduled path apply the same shape rule. A request with an effective output budget of at most one token is eligible for OneShot execution; requests with longer output budgets enter Decode, and scheduler iterations that mix new prefill chunks with active decode rows enter Mixed. Non-generation APIs such as classification, embeddings, and prompt-logprobs are normalized into the same admission interface with a fixed-output descriptor instead of a streaming generation descriptor.

OneShot executor. For an admitted OneShot generation batch, Prelude collects token slices into a flat varlen layout, runs one fused prefill forward, reads logits at the final position of each sequence, samples or extracts the requested logprob vector, and releases temporary tensors immediately. If prompt-logprobs are requested, the executor keeps hidden states only for the rows needed to compute the payload. Classification and embedding reuse the same idea with different output heads. The missing operations are as important as the operations that remain: no per-request decode block table lifecycle, no future-token KV reservation, no decode-loop sequence record, and no streaming-state aggregator.

Prefix-cache metadata. Prelude maintains a block-level prefix index over token blocks. Each cached entry stores parent-child structure, a cache key, optional paged-block identifiers, reference counts, and LRU metadata. On a hit, the scheduler obtains the matched token count and attaches the corresponding shared blocks before allocating suffix work. On a cold shared prefix, the scheduler may briefly defer peers with the same prefix key so that one leader populates the cache; peers are then refreshed and scheduled as suffix-only work. This is the appendix version of the tradeoff in Figure 2: waiting one scheduler tick can be cheaper than recomputing the same cold prefix many times in the same batch.

Fasttoken supported surface. Fasttoken is deliberately an inference-only tokenizer. It supports loading a fixed `tokenizer.json`, encoding with or without special tokens, batch encode, decode, per-token decode for streaming/logprob emission, vocabulary size queries, and BOS/EOS/pad-token accessors. It intentionally omits training-time mutation such as BPE merge insertion, vocabulary extension, dynamic serialization, and runtime normalizer composition. This makes the loaded vocabulary, merge-rank table, and token-id-to-string table immutable and shareable across admission worker threads without a lock on each call.

Fasttoken concurrency. The id-to-string table is stored as a contiguous vector indexed by token id, and encode uses immutable vocab and merge-rank tables plus per-call scratch buffers. The merge loop operates over token ids rather than heap-allocated token fragments; per-token decode is a direct lookup into the vocabulary arena. Since the loaded structures are immutable, the tokenizer is naturally `Sync`. Prelude can tokenize waiting requests across CPU worker threads during an admission tick without tokenizer-level synchronization.

B. Additional Serving Results

B.1. $K{=}1$ Microbenchmark with Startup

Table 4 restores the full prefill-only microbenchmark table, including startup time. Figure 4 shows the corresponding speedup view that was kept out of the six-page main body to preserve readability.

Table 4. Full prefill-only microbenchmark. Qwen3-0.6B, H200, 128-in / 1-out, $c = 1$, 100 requests. Latency speedups are baseline/Prelude; throughput speedups are Prelude/baseline.

Engine	Startup (s)	TTFT (ms)	E2E (ms)	In tok/s	RPM
Prelude	2	5.1	5.2	16,311.1	7,316.4
vLLM	44	13.3	13.5	7,843.1	3,514.9
SGLang	34	28.2	28.3	4,236.1	1,898.1
vs vLLM	22×	2.6×	2.6×	2.08×	2.08×
vs SGLang	17×	5.5×	5.4×	3.85×	3.85×

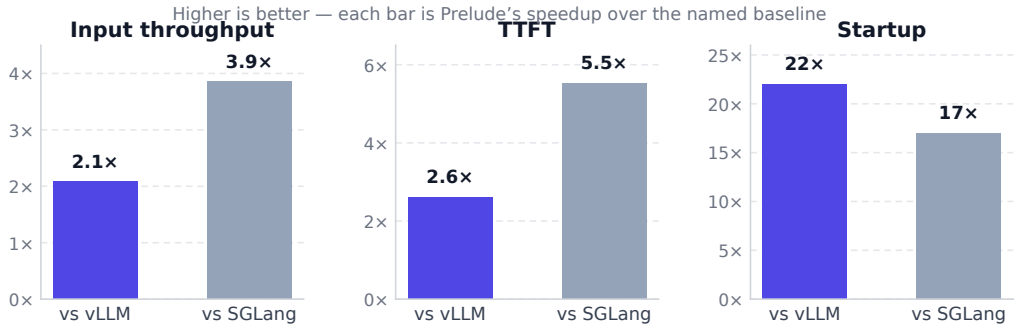


Figure 4. Prelude speedup on the Qwen3-0.6B prefill-only microbenchmark. Throughput and TTFT gains come from OneShot dispatch; startup reflects the native binary and lightweight runtime initialization.

B.2. Full Concurrency Sweep

Table 5 reports the per-concurrency values behind Figure 3. The workload is Qwen3-4B on H200 with 512-token inputs, one output token, and 200 requests per level. Prelude leads at every tested concurrency and peaks at 186.7 req/s at $c = 96$.

Table 5. Qwen3-4B / 512-in / 1-out throughput sweep. Values are requests per second.

Concurrency	Prelude	vLLM	SGLang	vs vLLM	vs SGLang
1	70.5	47.8	42.6	1.47×	1.65×
4	90.8	88.7	73.5	1.02×	1.24×
16	157.6	98.0	150.9	1.61×	1.04×
64	181.3	137.4	151.4	1.32×	1.20×
96	186.7	134.5	151.5	1.39×	1.23×
128	173.9	134.3	148.2	1.29×	1.17×

B.3. Full Multi-Token Decode Control

Table 6 restores the full decode-control table. The key interpretation is unchanged from the main text: once each request emits 32 tokens, the throughput gap against vLLM collapses to 1.03×, showing that the large OneShot gains are not explained by a uniformly faster forward kernel.

Table 6. Multi-token decode control. Qwen3-0.6B, H200, 128-in / 32-out, $c = 4$, 400 requests. Latency speedups are baseline/Prelude; throughput speedups are Prelude/baseline.

Engine	TTFT (ms)	TPOT (ms)	E2E (ms)	In tok/s	Out tok/s	RPM
Prelude	14.6	2.1	80.2	6,167.8	1,474.0	2,763.8
vLLM	30.9	1.7	83.3	5,979.2	1,430.4	2,682.0
SGLang	48.3	1.8	103.3	4,917.3	1,176.7	2,206.2
vs vLLM	2.12×	0.81×	1.04×	1.03×	1.03×	1.03×
vs SGLang	3.31×	0.86×	1.29×	1.25×	1.25×	1.25×

C. Full Tokenizer Microbenchmarks

The tokenizer benchmark uses Qwen/Qwen3-0.6B's BPE tokenizer with a 151,936-token vocabulary loaded from the HuggingFace Hub `tokenizer.json`. The harness first asserts encode equality, encode-without-special-tokens equality, decode equality, and streaming decode equality against HuggingFace `tokenizers`; only then does it report timing. Both fasttoken and HuggingFace `tokenizers` are measured as Rust crates with no Python binding overhead.

Table 7. Encode speed: HuggingFace `tokenizers` Rust crate vs fasttoken. Times are wall-clock per call.

Input	Chars	Tokens	HF (µs)	fasttoken (µs)	Speedup
tiny	5	1	1.4	0.1	11.9×
short_english	51	11	5.7	0.4	12.9×
short_chinese	90	19	4.9	0.4	11.0×
medium_prose	674	111	50.7	14.5	3.5×
code_snippet	470	153	58.4	16.7	3.5×
mixed_multilingual	641	141	40.4	16.5	2.4×
long_repeat	2,025	451	182.0	27.2	6.7×
long_unique	4,000	625	300.3	36.2	8.3×
very_long	8,000	1,245	601.4	27.3	22.0×
chat_template	212	37	17.1	11.9	1.4×
long_32K	32,000	4,981	2,452.9	75.3	32.6×
long_64K	64,000	9,961	4,842.8	129.7	37.3×
long_200K	200,000	31,124	21,320.0	309.3	68.9×
long_code_16K	16,000	4,403	1,862.2	55.9	33.3×
multi_turn_chat_8K	21,370	4,220	1,935.9	204.8	9.5×
multi_turn_chat_32K	85,480	16,880	8,169.1	1,075.1	7.6×
long_chinese_32K	31,998	5,659	1,045.0	66.1	15.8×

Table 8. Decode speed: HuggingFace `tokenizers` Rust crate vs fasttoken. Times are wall-clock per call.

Input	Tokens	HF (µs)	fasttoken (µs)	Speedup
tiny	1	0.1	0.1	1.6×
short_english	11	0.5	0.3	1.8×
short_chinese	19	1.0	0.5	2.0×
medium_prose	111	5.8	2.6	2.2×
code_snippet	153	5.7	3.3	1.8×
mixed_multilingual	141	6.5	3.4	1.9×
long_repeat	451	19.2	9.7	2.0×
long_unique	625	34.9	14.9	2.3×
very_long	1,245	70.2	29.7	2.4×
chat_template	37	2.0	0.9	2.2×
long_32K	4,981	278.3	115.4	2.4×
long_64K	9,961	594.0	370.7	1.6×
long_200K	31,124	1,782.2	728.2	2.4×
long_code_16K	4,403	189.2	96.5	2.0×
multi_turn_chat_8K	4,220	196.5	89.4	2.2×
multi_turn_chat_32K	16,880	795.1	361.3	2.2×
long_chinese_32K	5,659	292.1	138.6	2.1×

Batch encode. At a fixed input size of 674 characters, batch encode wall-clock per batch is HF 50.1/109.8/236.7/675.7 µs and fasttoken 15.6/54.6/82.6/194.0 µs for batch sizes 1/4/16/64, a 3.2/2.0/2.9/3.5× speedup respectively.

Per-token streaming decode. The logprob and response hot path decodes one token-id to its text fragment. HF takes 0.13 µs per call and fasttoken takes 0.06 µs, a 2.0× speedup.

Concurrent encode. With 8 threads and 100 encodes per thread, HF takes 8.3 µs per encode and fasttoken takes 3.1 µs per encode, a 2.7× speedup. Correctness is asserted across all 800 encodes.

Load time. Tokenizer load time is HF 120.9 ms versus fasttoken 382.0 ms. HF is faster at load because fasttoken precomputes merge-rank tables at startup. This one-time startup cost is the deliberate tradeoff that makes steady-state encode and decode calls cheaper.

D. Hardware, Software, and Reproducibility

Hardware. All serving benchmarks use a single NVIDIA H200 with 143 GB HBM3e per engine, isolated on its own GPU, and an AMD EPYC 9575F 64-core host.

Models and precision. The prefill-only and decode-control microbenchmarks use Qwen3-0.6B in BF16. The concurrency sweep uses Qwen3-4B in BF16. No model training is performed.

Serving software. Prelude is built as a native Rust release binary. vLLM and SGLang run in their official Docker images at production-default settings: BF16 weights, default block size, default scheduling budgets, and prefix caching enabled where it is the default. Speculative decoding, quantization, and non-default kernels are disabled for all engines.

Benchmark harness. The request driver is `genai-bench`. The OneShot microbenchmark uses 100 requests of 128 input tokens and one output token at concurrency 1. The decode control uses 400 requests of 128 input tokens and 32 output tokens at concurrency 4. The Qwen3-4B sweep uses 200 requests per concurrency level, 512 input tokens, and one output token. Tokenizer microbenchmarks use Prelude's `tokenizer_bench` binary against Qwen/Qwen3-0.6B's `tokenizer.json`.

Artifacts. The benchmark driver records the exact command lines and raw JSON logs for all reported serving runs. The source release includes the paper figure scripts, tokenizer benchmark harness, and documentation needed to reproduce the reported tables.

E. Scope and Limitations

Prelude targets execution-class dispatch, prefix-aware OneShot planning, and tokenizer overhead. It is not presented as a new attention kernel, speculative decoder, quantization method, tensor-parallel runtime, or multi-node serving system. The evaluation runs on one H200; tensor parallelism, pipeline parallelism, non-Hopper GPUs, and multi-node deployments can shift the balance between CPU admission overhead and GPU forward cost. Each configuration is reported as a timed benchmark run rather than a statistical study with confidence intervals. Finally, prefix-aware OneShot planning is evaluated structurally in this workshop version; a future trace-driven study should quantify latency and throughput under measured prefix-reuse distributions.