VIBESERVE: CAN AI AGENTS BUILD BESPOKE LLM SERVING SYSTEMS?

Anonymous authors
Paper under double-blind review

ABSTRACT

For years, we have built LLM serving systems like any other critical infrastructure, with a single general-purpose stack hand-tuned over many engineer-years to support every model and workload. In this paper, we take the opposite bet and let coding agents automatically synthesize *bespoke* serving systems for different usage scenarios. We propose **VibeServe**, the first agentic loop that generates entire LLM serving stacks end-to-end. VibeServe uses an outer loop to plan and track the search over system designs, and an inner loop to implement candidates, check correctness, and measure performance on the target benchmark. In standard serving settings, VibeServe matches vLLM and SGLang on Llama-3.1-8B, and delivers 2.3× the goodput of a tuned SGLang for a 397B MoE model on four AMD MI300As. In five non-standard scenarios, VibeServe outperforms existing systems by exploiting opportunities that generic systems miss, spanning non-standard model architectures, workload knowledge, and hardware-specific optimizations. Together, these results argue that infrastructure software should favor generation-time specialization over runtime generality.

1 INTRODUCTION

LLM serving systems are critical software infrastructure for an economy increasingly dependent on generative AI. Open-source stacks such as vLLM [36], SGLang [79], and TensorRT-LLM [52] provide efficient abstractions across a broad range of models and hardware. Yet their designs are shaped primarily by mainstream deployments, such as decoder-only Transformers on NVIDIA GPUs serving generic chat workloads. As a result, emerging model families (e.g., multimodal models or hybrid state-space architectures), along with new hardware accelerators and atypical workloads, often suffer from suboptimal performance or even require substantial new implementation effort [31, 76, 63, 28, 17, 44]. As the space of model–hardware–workload combinations continues to expand, a one-size-fits-all serving stack is becoming increasingly difficult to sustain.

In this work, we explore a different point in the design space: *rather than maintaining a single general-purpose runtime, can we generate a bespoke serving system for each combination of model, hardware, and workload?* Per-deployment specialization is a longstanding idea in computer systems [46, 12, 48, 8, 49, 47], but it rarely pays off in practice because per-target engineering cost usually dwarfs the gain. However, coding agents are changing this calculus. Their demonstrated effectiveness on individual components [56, 67, 51, 61, 22] and system policies [80] suggests that per-target specialization could now be feasible at scales where engineering costs were previously prohibitive (Figure 1).

Generating an end-to-end serving system, however, is a long-horizon, multi-component task that existing agentic optimization does not address. Prior systems operate on a much smaller code surface, e.g., a single GPU kernel, an isolated algorithm, or a single policy embedded in an otherwise fixed system [56, 67, 51, 61, 43, 22, 32, 80]. Designing and optimizing an end-to-end system exceeds the context window of any single agent. The standard recourse, compaction [5, 39], can discard details from earlier work and weaken continuity across long tasks [6, 69]. Evolutionary frameworks sidestep this drift via a population of scored programs [51, 61, 43], but a scalar score cannot encode the planning state an end-to-end system needs. Multi-agent loops carry richer state across roles but do not reset agent context windows [22, 32], inheriting limitations from compaction. Long-horizon coding-agent harnesses sustain state across sessions but produce incorrect systems that underperform state-of-the-art baselines [9, 42].

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.

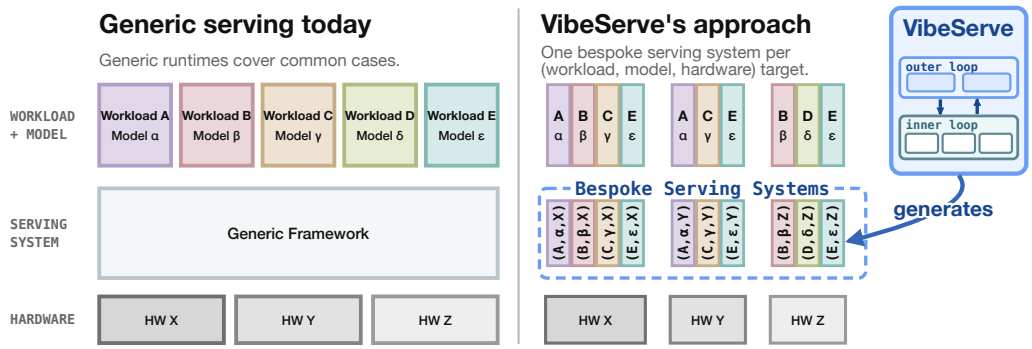


Figure 1: Motivation for VibeServe. General-purpose serving frameworks target common deployments; VibeServe instead generates systems specialized to each model–hardware–workload target.

We present **VibeServe**, a multi-agent system that synthesizes bespoke LLM serving runtimes from scratch. To let agents target the open-ended space of model–hardware–workload deployments, VibeServe exposes two extensible surfaces. The first is a small set of user-provided artifacts (model and reference implementation, accuracy checker, workload benchmark, and target hardware). The second is an Agent Skills library [7] of serving-systems knowledge distilled from existing engines. New model families, hardware platforms, and optimization techniques enter as new skill entries, so coverage extends beyond the combinations hand-engineered into existing runtimes.

For each target, VibeServe factors the work along two axes. An *outer loop* plans across iterations based on git-recorded optimization history, picking the next optimization and dispatching one concrete task to the inner loop. Its planning state is structured and persistent (e.g., issues, a long-term memory file, the commit history), which is richer than a scalar score and not confined to a single agent's context. This lets the outer loop separate design failures from implementation flaws. An *inner loop* executes each task through a coding-agent harness. Implementer, Accuracy Judge, and Performance Evaluator agents take turns in fresh contexts, working over a read-only reference implementation and checker. The outer loop only considers correct implementations. Performance naturally varies as agents explore different design choices, but incorrect candidates cannot derail subsequent rounds. §3 describes the design in detail.

We evaluate VibeServe across seven scenarios (§4). In standard serving settings, VibeServe matches or beats general-purpose systems. On Llama-3.1-8B-Instruct [19] on H100, it reaches near-parity with vLLM [36] and exceeds SGLang [79] by 5% in throughput. For Qwen3.5-397B on four AMD MI300As, it delivers $2.33\times$ higher goodput than an SGLang that we first ported and tuned. More importantly, VibeServe works where generic systems fall short, including non-standard workload patterns (e.g., speculative decoding for code editing with predicted output, workload-aware prompt caching), model architectures (e.g., hybrid attention, complex multimodal models), and hardware backends (e.g., MacBook). These specialized systems achieve a $5.95\times$ speedup for predicted-output code editing, $3.45\times$ higher throughput for hybrid prompt caching, a $1.69\times$ TTFT speedup for streaming speech recognition, a $2.6\times$ speedup for MacBook JSON decoding, a $6.27\times$ speedup for multimodal inference on MacBook, and a 21.4% latency reduction on H100.

In summary, we contribute the following:

1. We argue that per-target bespoke serving is now feasible with long-horizon coding agents.
2. We build VibeServe, a multi-agent loop with an outer planner and an inner Implementer/Judge/Evaluator that synthesizes complete serving runtimes.
3. We demonstrate parity with vLLM and SGLang on standard serving, a $2.33\times$ win over a tuned SGLang on AMD, and concrete wins across five non-standard scenarios spanning workload, model architecture, and hardware.

2 MOTIVATION

Why LLM serving needs bespoke systems. Modern LLM serving stacks [36, 79, 52] achieve strong performance across many models and hardware platforms through various optimization techniques [77, 36, 75, 10]. However, use cases are diversifying rapidly. New model architectures, hardware accelerators, and application interfaces introduce execution structures that challenge runtime abstractions designed for the standard case. This creates persistent *long-tail* scenarios where a general-purpose stack may work suboptimally, miss optimizations that a bespoke system could implement, or be unable to run the workload at all. In other words, generic abstractions impose a *portability tax* on non-standard models, hardware, and applications [12, 49].

Building bespoke systems can solve this problem. For example, knowing workload characteristics at design time can enable optimizations that a workload-agnostic runtime cannot safely assume. RAG-like applications with long shared prefixes can amortize prefill through prompt caching [16, 30], while aggressive speculative decoding based on predicted outputs is possible for some applications like code editing [54, 13, 73, 65]. Similarly, tailoring for a particular model architecture can expose state and execution patterns that fall outside standard decoder-only assumptions. As an example, hybrid state-space/attention models require cache-management strategies different from those used for decoder-only Transformers [41, 53, 57, 64, 78]. Many modern multimodal models also have complex architectures that require significant serving-system effort, such as modality-specific scheduling, memory management, and cross-component execution [31, 76]. Finally, knowing the target hardware can inform the right runtime design. Apple Silicon, for example, exposes a unified memory model that differs from that of CUDA-centric serving stacks [23, 26].

Exploiting these opportunities is fundamentally a system-level problem. It requires coordinated decisions across GPU kernels, memory management, request scheduling, and the external interface, and optimizing one component is insufficient if the rest of the runtime still enforces the generic execution contract. A bespoke system can instead specialize entire layers to the target rather than preserving compatibility with unrelated deployments.

Why bespoke systems are possible now. Specialization to remove abstraction overhead is a longstanding idea in computer systems, from extensible operating systems to unikernels [12, 48, 8, 49, 46, 47]. It has been impractical for LLM serving, since building and maintaining a new runtime for every model–hardware–workload combination would normally cost more than it gains. Recent coding agents suggest a different cost model. They are increasingly effective at writing software, including real-world bug fixes and performance-critical GPU kernels or scheduling policies [29, 70, 11, 56, 67, 51, 61, 22, 80]. Still, end-to-end system generation remains much more challenging because it requires long-horizon reasoning over a large codebase and coordination across components at all layers [35, 38, 11, 66, 6, 42, 9, 32].

We argue that LLM serving can be the first domain in which agents successfully generate useful systems end-to-end, since there is a broad need for specialization, the optimization objective is concrete and numeric (e.g., throughput or time-to-first-token latency), and correctness can be checked against a reference implementation. This motivates VibeServe, which generates a serving system specialized to a given deployment target.

3 DESIGN

Figure 2 shows VibeServe's architecture. An outer planning loop and an inner implementation loop iteratively produce an end-to-end serving system from a small set of user-provided artifacts. The framework is target-agnostic, and specialization enters through three surfaces. First, per-target *inputs* (§3.1) define the model, hardware, and workload. Second, an *agentic pipeline* (§3.2) creates and optimizes a bespoke serving system for the target. Third, an extensible *skills library* (§3.3) lets agents learn about new model families, hardware platforms, and optimization techniques.

3.1 INPUTS

VibeServe takes a small set of user-provided artifacts that define the target deployment. First, the user provides the model weights and a reference implementation, such as a Hugging Face

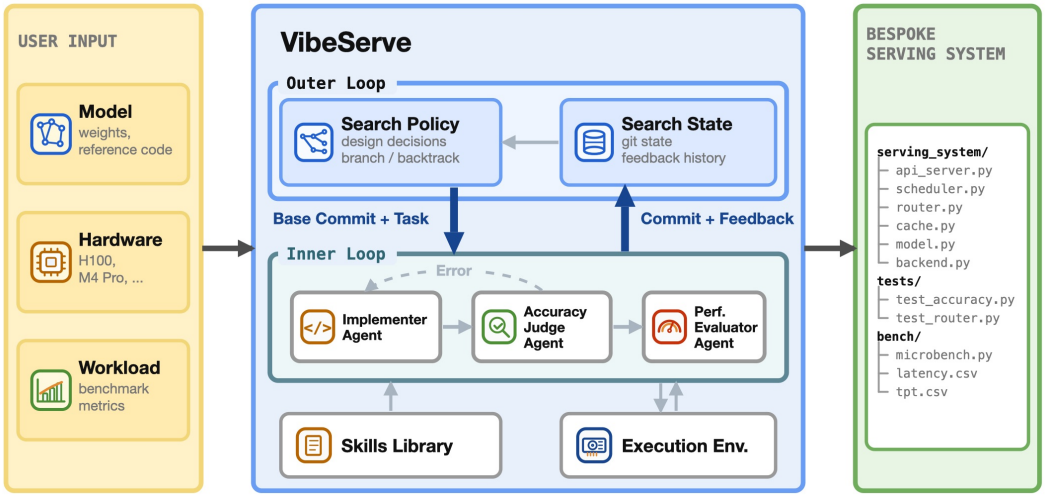


Figure 2: VibeServe architecture. User-provided artifacts define a target deployment. The outer loop plans over validated git checkpoints and dispatches a single-round task to the inner loop, where an Implementer, Accuracy Judge, and Performance Evaluator collaborate on a shared workspace using the execution environment and a skills library of serving-systems knowledge.

Transformers [68] model, which is assumed to be accurate but not efficient. Second, the user provides an accuracy-checking script that compares a candidate serving system against the reference. We treat this checker as the source of truth for correctness, mirroring how continuous integration tests gate human-engineered systems. Fully verifying the semantic accuracy of serving systems is an open problem beyond our scope [24, 18]. Third, the user provides a benchmark script that exercises the target workload and emits the metric to optimize, such as latency, throughput, or time-to-first-token. Finally, the user provides natural-language instructions describing the hardware platform and the expected shape of the deliverable, such as an HTTP API or benchmark harness interface.

3.2 Multi-agent pipeline

Each candidate runs in an isolated workspace that mounts the user-provided artifacts read-only and exposes the target execution environment (a local or cloud GPU) along with platform-specific profilers (Nsight Systems and the PyTorch profiler on NVIDIA). Agents can only edit the serving-system code they generate, and read-only mounts prevent the Implementer from editing the checker or reference implementation. VibeServe wraps existing coding-agent harnesses with three pieces of shared infrastructure: a Model Context Protocol (MCP) [3] server whose schema is defined by the outer-loop policy and through which inner-loop agents return structured information back to the policy, a *skills library* of operational knowledge loaded into the agent context, and an execution environment that issues build, run, and measure calls.

Outer loop. The outer loop's search policy is modular, exposing a single per-round operation. It reads prior state, hands the inner loop a starting commit and a task, and receives the resulting commit with the performance metric. Two shared mechanisms support coordination beyond this contract. First, every accepted build is a git commit, so any policy can revert cheaply when a later round passes correctness but regresses on the headline metric. Second, each policy defines its own MCP server schema, whose tools give inner-loop agents a structured channel to return information to the policy during execution. We implement three policies (evolutionary search [51], the Ralph loop [27], and the issue-tracker policy used in our evaluation).

The *issue-tracker* policy maintains a backlog of structured issues, using the MCP server tool interface to define and enforce the issue schema. Inner-loop agents file issues through the predefined schema. The outer-loop Orchestrator agent picks the next issue to dispatch at each round, optionally requests a revert to an earlier checkpoint, and updates a long-term memory of optimization directions (a

markdown file it reads on entry and edits at the end of each round). The selected issue, including its acceptance criteria, is the contract handed to the inner loop. The long-term memory lets the Orchestrator distinguish implementation failures from evidence that a direction is unsuitable for the workload. A failed attempt may signal that the implementation needs debugging or a narrower scope, not that the technique should be discarded. When issues investigate different hypotheses, the Orchestrator can dispatch several of them to parallel Implementer instances.

Inner loop. In the inner loop, Implementer, Accuracy Judge, and Performance Evaluator agents work in sequence, revising the codebase until it passes correctness checks. This separation keeps implementation, correctness, and performance reasoning in independent contexts. A combined agent can weaken its own correctness criteria to land a hard optimization. The Judge instead inspects diffs and runtime behavior with a fresh context, and the Evaluator runs only after correctness is gated. Each agent is implemented by a coding-agent harness, such as Codex CLI, Claude Code, or DeepAgents, that can read and edit files, run commands, and return structured results [55, 4, 37].

The *Implementer* produces and revises the candidate serving system in the workspace. It receives the task and pass criteria from the outer loop along with pointers to the reference implementation and model weights, and consults the serving-systems skills library (§3.3).

The *Accuracy Judge* gates correctness in three ways. It runs the user-provided accuracy checker against the candidate server, verifies any per-round pass criteria from the outer loop (for the issue-tracker policy, the issue's acceptance criteria), and inspects the candidate's source and runtime behavior for reward-hacking patterns such as prompt-keyed completion caches or fast paths that bypass model inference. On failure, the Judge returns feedback and the inner loop iterates until a build passes or a retry budget is exhausted, at which point the round fails and returns to the Orchestrator.

Once an implementation clears the Judge, the *Performance Evaluator* profiles it and generates performance hints for subsequent rounds. It starts with end-to-end performance on the user-provided benchmark, then drills down with the platform-specific profilers from the workspace when finer measurements are needed, drawing on the skills library (§3.3) for profiler-specific guidance. For targeted investigations, the Evaluator can insert temporary instrumentation around specific code blocks or commit microbenchmarks for repeated measurement. The inner loop then returns the headline metric, trace analysis, hints, and feedback to the outer loop.

3.3 Skills Library

VibeServe provides agents with a serving-systems skills library in the Agent Skills format [7]. The skills are derived from the source code of mature serving engines and the surrounding research literature, organized along the abstraction layers an engineer works through when building a serving engine (model architectures, serving algorithms, programming frameworks, backend libraries, hardware platforms, reference engines, and tooling). This lets an agent retrieve focused guidance for a task, such as how continuous batching changes scheduler state, how to use FlashInfer or FlashAttention without reimplementing kernels [75, 10], how MLX differs from PyTorch on Apple Silicon, or where a mechanism lives in vLLM, SGLang, or TensorRT-LLM.

The library is also an extensibility surface. New model families, hardware platforms, frameworks, backend libraries, or reference engines can be added as new skill entries under the corresponding layer. Because these axes interact, algorithm skills include compatibility notes linking each technique to supported backends, hardware, and engines (e.g., which paged-KV implementations exist on which hardware). The library stops at the serving-system boundary. Custom CUDA, Triton, or CUTLASS kernel authoring is delegated to GPU-kernel skills.

4 Evaluation

Our central question is whether bespoke serving systems generated by VibeServe are competitive with human-engineered systems and address niche yet important use cases where general-purpose systems fall short. We evaluate this on seven scenarios spanning the three axes introduced in §1: workload pattern, model architecture, and hardware.

4.1 SETUP

All scenarios follow the input interface in §3.1. We verify generated systems against the reference implementation and report the workload-relevant metric, such as token throughput, latency, or time-to-first-token (TTFT). The Implementer, Accuracy Judge, and Performance Evaluator are each instantiated with Codex CLI [55] in every scenario except B, which uses Claude Code [4] with Opus 5. The outer loop uses the issue-tracker policy (§3.2) in every scenario.

We evaluate the following scenarios.

- **A: Standard LLM serving.** Llama-3.1-8B-Instruct [19] on H100, a mature setting where existing systems are heavily optimized. We measure generation throughput across arrival rates.
- **B: Distributed MoE serving on AMD.** Qwen3.5-397B-A17B-MXFP4 (512 routed experts, Gated DeltaNet plus attention) on four MI300A accelerators with tensor and expert parallelism, under a multi-turn chat workload that requires preserving KV and DeltaNet state across turns. We measure peak SLO-constrained goodput against an SGLang baseline tuned with VibeServe itself.
- **C: Code editing with predicted outputs.** Qwen3-32B [72] on H100 with a predicted-outputs interface [54]. Code edits overlap heavily with the input [13, 73, 65], so speculative decoding from user-provided predictions can skip most decoding steps, a capability absent from standard serving systems. We measure single-batch latency on CodeEditorBench [20]. #workload
- **D: Hybrid-architecture prompt caching.** Olmo-Hybrid-7B [50] on L4, whose Gated DeltaNet [74] and attention layers make caching hard under limited memory [57]. The workload is RAG-like, with a shared 32k-token prefix. We measure throughput. #model #workload
- **E: Streaming ASR.** Moonshine Streaming medium [34] on L4, whose sliding-window encoder (unlike Whisper [59]) needs per-stream cache support missing from existing systems. We measure TTFT at concurrency 32 against a vLLM plugin. #model #workload
- **F: Local constrained decoding.** Llama-3.1-8B-Instruct on a MacBook, generating JSON under constrained decoding. Fixed schema spans let a specialized decoder skip generic per-step sampling overhead. We measure single-batch latency on JSONSchemaBench [15]. #workload #hardware
- **G: Local image generation.** Show-o2 [71] on a MacBook and H100, a unified model combining a discrete tokenizer, a diffusion head, and an autoregressive LM that neither vLLM nor vLLM-Omni supports. We measure latency. #model #hardware

4.2 RESULTS

We present results in scenario order. Iteration-level details are taken from VibeServe's own logs.

Scenario A: parity on a heavily optimized setting. Figure 3 traces 60 VibeServe iterations on Llama-3.1-8B-Instruct (H100). The generated system reaches vLLM parity on token throughput and TPOT at all four request rates and lands within 5% on TTFT. It exceeds SGLang by 5% on throughput and 3% on TTFT. VibeServe pursued throughput first, reaching parity by iteration 30 with latency roughly flat, then shifted to latency, with TTFT and TPOT improving sharply over iterations 30–60. VibeServe introduced each higher rate after plateauing, escalating to 128 req/s.

Scenario B: distributed MoE serving on AMD. Figure 4 shows peak goodput on Qwen3.5-397B-A17B-MXFP4 across four AMD MI300A accelerators on the multi-turn chat workload detailed in §A. Starting from a simple PyTorch implementation at 12.8 tok/s, VibeServe adds prefix caching that reuses both attention KV and DeltaNet state, decode and prefill graph capture with a fused MoE kernel, a series of communication optimizations (one-shot and sequence-parallel all-reduce), decode graphs up to batch size 128, folding of small turn-opening prefills into decode steps, wide and unpadded prefill packing, better prefix-cache eviction, and finally speculative decoding with the model's multi-token-prediction (MTP) head. The result is 2,242 tok/s, which is 2.33× the tuned SGLang baseline (961 tok/s) and ∼175× the starting point. Stock SGLang does not run this model on MI300A, so the baseline is a VibeServe-modified version that we first made functional and then

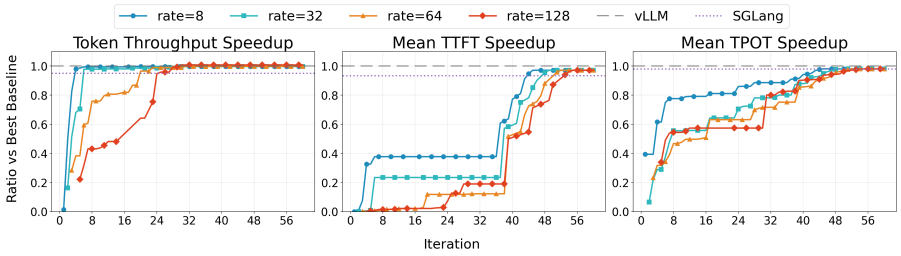


Figure 3: On Llama-3.1-8B-Instruct (H100), VibeServe matches vLLM and exceeds SGLang by 5% (TTFT by 3%) over 60 agentic-loop iterations. Panels show VibeServe's token throughput, mean TTFT, and mean TPOT relative to vLLM (1.0 is parity, higher is better), at four request rates.

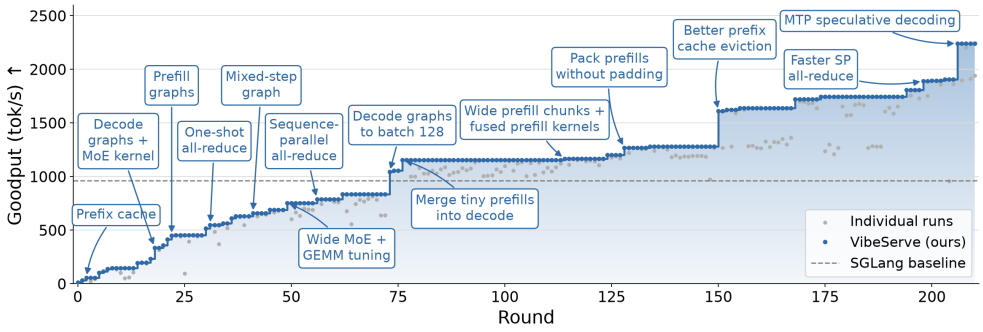


Figure 4: Scenario B. Best goodput across the load ramp at each round, on 4× AMD MI300A. The dashed line is the tuned SGLang baseline (961 tok/s). Gray dots are individual runs.

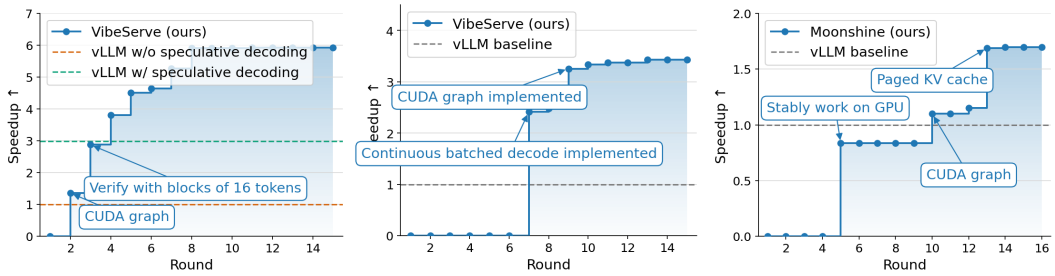
tuned (§A). Both systems use MTP with k=3. We ramp the load for each system and report the best goodput. SGLang peaks at 48 concurrent chat sessions, while our engine peaks at 96.

Scenario C: predicted-output speculative decoding. Figure 5a traces 15 iterations on Qwen3-32B/CodeEditorBench against two vLLM baselines: vanilla autoregressive (1.0×) and draft-model speculative decoding (≈ 3.0×, dashed). After an early CUDA-graph pass, iteration 3 introduces the predicted-output verifier. It proposes tokens from the user-supplied prediction in 16-token blocks and verifies each block in a single target-model forward. This reaches 2.9×, already on par with vLLM's draft-model speculative decoder at zero draft-model compute. Block sizing and acceptance bookkeeping then push this to 5.95× by iteration 14, 2.0× over vLLM-with-spec-dec.

Scenario D: hybrid-architecture prompt caching. Figure 5b shows token-generation throughput vs. vLLM on Olmo-Hybrid-7B (L4) over 15 iterations. The first six rounds fail accuracy gates while VibeServe wires up the dual cache, with attention KV blocks plus per-DeltaNet recurrent-state snapshots at the prefix boundary. Continuous batched decode against the shared state, then CUDA-graph capture, bring the system to a 3.45× plateau. vLLM's GDN prefix caching cannot share DeltaNet state at this workload's shared/unique boundary (§A), so the baseline recomputes the 32k prefix per request.

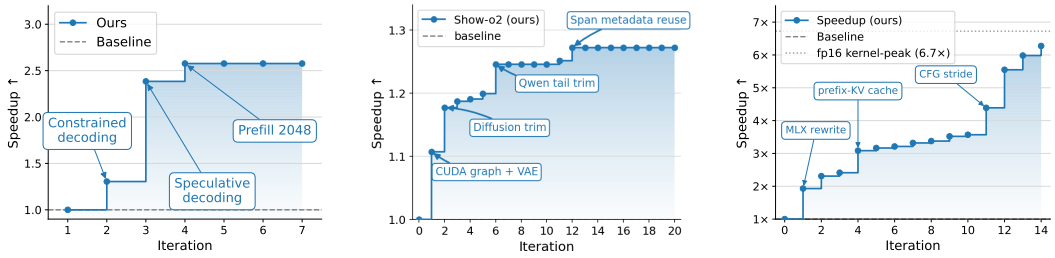
Scenario E: streaming ASR. Figure 5c shows TTFT speedup over a vLLM-Moonshine plugin at concurrency 32 on L4 over 16 iterations. VibeServe first reaches a working but sub-baseline configuration (0.84×) by aligning the per-stream encoder cache with Moonshine's sliding-window attention, then adds CUDA-graph capture and a paged KV cache for per-stream encoder state to reach 1.69×. The improvement comes from giving the encoder layer first-class per-stream cache management, which the plugin path does not expose.

Scenario F: constrained JSON decoding on a MacBook. Figure 6a traces the trajectory from a 22.1 s vanilla autoregressive baseline. VibeServe first adds XGrammar-based constrained decod-



(a) Scenario C. (b) Scenario D. (c) Scenario E.

Figure 5: Workload- and model-specific scenarios: speedup over a baseline across VibeServe iterations (dashed line is parity; higher is better). (a) Qwen3-32B vs. vLLM without/with draft-model speculative decoding. (b) Olmo-Hybrid-7B vs. vLLM. (c) Moonshine TTFT vs. a vLLM plugin.



(a) Scenario F. (b) Scenario G, H100. (c) Scenario G, MacBook.

Figure 6: Hardware- and workload-specific scenarios with speedup over a baseline across VibeServe iterations. The dashed line at 1.0 is parity, and higher is better. (a) Llama-3.1-8B-Instruct JSON decoding on a MacBook (Apple M3 Pro, 36 GB). (b) Show-o2 1.5B-HQ 432×432 text-to-image on H100. (c) Show-o2 on the MacBook, where the dotted line is the fp16 kernel-peak ceiling (6.7×).

ing [40] (16.9 s), then layers speculative decoding with a Llama-3.2-1B-Instruct-4bit draft against the 8B-8bit target at $K{=}4$, reaching 9.3 s. A larger 3B-4bit draft was slower, since the 1B's lower per-step cost outweighed its lower acceptance rate. Raising `mlx_lm`'s `prefill_step_size` from 512 to 2048 lets VibeServe prefill our ∼1300-token prompts in one chunk, yielding 8.6 s (2.6×). K/V quantization, alternative K, and `mx.compile` did not help.

Scenario G: Show-o2 on H100 and MacBook. On H100 (Figure 6b), p50 latency falls from 873 ms to 687 ms (21.4%) over 20 iterations. Gains are front-loaded in the first dozen rounds (CUDA-graph replay/prewarm, trimming inactive diffusion tokens, restricting AdaLN to the active image span, and a Qwen tail trim). Further passes then map the limits, as changes that regress quality or alter outputs are rejected and Qwen prefix reuse yields no gain (§A).

On MacBook (Figure 6c), VibeServe first ports the Qwen2.5-1.5B body and 10-block diffusion head to MLX and elides a redundant SigLIP `und_trans` pass on noisy latents (2.4×). Cross-step redundancy then dominates, and prefix-KV caches on the body and head, plus a prefill trim to `[0, image_end)`, bring warm latency to 3.5×, with the body at ∼92% of the fp16 compute peak. A classifier-free-guidance (CFG) stride at $K{=}16$ skips the unconditional branch on $K{-}1$ of every K steps and reuses the cached `v_uncond`. This reaches 15.54 s (6.27× over PyTorch-MPS), within ∼7% of a 14.5 s fp16 kernel-perfect floor.

4.3 Ablations

We ablate the design choices of §3. The pipeline ablations run on Scenario A (Llama-3.1-8B, H100, Codex GPT-5.5 Medium), which isolates pipeline behavior from target-specific optimizations. None reaches a competitive serving system, whereas the full pipeline matches vLLM (≈10k tok/s) after 60 rounds / 25.0 h (Table 1). §B gives full details.

Starting point: from scratch vs. an existing codebase. VibeServe's inner loop can also start from an existing codebase. We seed Scenarios D and E from vLLM v0.26.0 and Scenario G from vLLM-Omni v0.24.1 and compare against the from-scratch runs (§B). Seeding helps only when the seed's abstractions already fit the target. vLLM already supports Olmo-Hybrid, so the Scenario D agent only adds workload-specific prefix caching and matches from-scratch performance by round 4, roughly twice as fast in wall-clock time. For Moonshine (E) and Show-o2 (G), where the seed lacks support, the agents implement outside the seed's abstractions rather than risk invasive edits.

Agentic pipeline, skills library, and outer loop. *Single-agent.* Collapsing the Implementer, Judge, and Performance Evaluator into one long-running agent reaches only 617 tok/s over 35 hours. Without a fresh-context Judge, it repeatedly skips its own accuracy and benchmark gates and gets stuck in local optimization rather than attempting larger architectural changes. This supports factoring implementation and validation into separate agents (§3.2).

No-skill. Removing the skills library caps throughput at 1,100 tok/s after $\sim$35 rounds. Agents rediscover standard techniques from the codebase alone but, without pointers from skill files, focus on minor parameter tuning rather than the larger architectural changes that close the remaining gap. This shows the skills library (§3.3) supplies directions that experimentation alone does not surface.

Outer-loop $\rightarrow$ OpenEvolve. Replacing the issue-tracker orchestrator with OpenEvolve [61], an AlphaEvolve-style [51] evolutionary search over scored populations, reaches only 537 tok/s after 30 rounds. It mutates a fixed, user-marked code region that does not scale past 1k LoC, and its population-based search lacks the issue-tracker policy's long-term memory (§3.2) for planning across rounds.

5 Related Work

Agentic optimization systems use a few search paradigms, none applied to greenfield end-to-end system synthesis. *Evolutionary search* selects among agent-generated candidates by measured performance [51, 61, 43, 67, 21]. *Multi-agent iteration* has agents hypothesize, experiment, and refine across rounds within a single context window [22, 32]. *Autoresearch* [33] uses one long-running agent that tracks candidates across git branches. All three target a bounded code scope (e.g., a marked region) or rely on a scalar score or a single conversation that cannot encode the bottleneck information driving an end-to-end system's next step. VibeServe is the first agentic system to design a multi-component serving system end-to-end.

VibeServe sits within a broader literature on long-horizon coding agents [35, 38, 11, 6, 27, 69, 1]. Tasks that exceed a context window are typically handled by compaction [5, 39], whose lossy summarization causes drift in performance and correctness. Industrial prototypes from Cursor and Anthropic build end-to-end systems by handing work between fresh agent sessions through task abstractions over shared repository state [42, 9], but stop short of optimizing performance. Building on these ideas, VibeServe targets *performant* code by giving agents direct profiler access, folding performance analysis into every implementation change through role-based agents, and packaging platform, optimization, and profiling knowledge as skills (Appendix D).

6 Conclusion

We argue that LLM serving should generate a bespoke system for each deployment target rather than maintain a single general-purpose runtime. VibeServe matches vLLM and beats a tuned SGLang in standard settings, and yields concrete wins across five non-standard scenarios, two of which cannot run on any generic stack.

VibeServe relies on a user-supplied correctness checker and a non-trivial per-target compute budget (§A), was evaluated only on a single node, and leaves the maintainability of generated systems unmeasured. Looking ahead, as coding agents improve, we expect generation-time specialization to beat runtime generality [46, 12, 48] in more domains where generic abstractions cost performance.

AI USE STATEMENT

We used Codex CLI and Claude Code to aid and polish writing, retrieve and discover related work, and support research ideation and execution. In the reported experiments, these coding agents generated and revised serving-system implementations, ran accuracy and performance checks, and analyzed measurements to guide subsequent optimization. We evaluated the generated systems using the accuracy checkers and benchmarks described in Sections 3 and 4. The authors take responsibility for the final text, citations, experiments, and claims.

REFERENCES

[1] Thomas Anderson, Ratul Mahajan, Simon Peter, and Luke Zettlemoyer. Self-defining systems, 2025. URL https://foci.uw.edu/papers/whitepaper2025-sds.pdf.

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, 2024. doi: 10.1145/3620665.3640366.

[3] Anthropic. Introducing the model context protocol. https://www.anthropic.com/news/model-context-protocol, 11 2024.

[4] Anthropic. Claude Code. https://docs.anthropic.com/en/docs/claude-code/overview, 2025. Accessed: 2026-05-06.

[5] Anthropic. Effective context engineering for ai agents. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents, 9 2025. Anthropic Engineering blog. Accessed: 2026-05-06.

[6] Anthropic. Effective harnesses for long-running agents. https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents, 11 2025. Anthropic Engineering blog. Accessed: 2026-05-06.

[7] Anthropic and Contributors. Agent skills. https://agentskills.io, 2025. Open standard; https://github.com/agentskills/agentskills.

[8] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pp. 267–283, 1995.

[9] Nicholas Carlini. Building a C compiler with a team of parallel Claudes. https://www.anthropic.com/engineering/building-c-compiler, 2 2026. Anthropic Engineering blog. Accessed: 2026-05-06.

[10] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems*, volume 35, pp. 16344–16359, 2022.

[11] Xiang Deng, Jeff Da, Edwin Pan, Yannis Y. He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa R Kundurthy, Sean M. Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-bench pro: Can AI agents solve long-horizon software engineering tasks?, 2026. URL https://openreview.net/forum?id=9R2iUHhVfr.

[12] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, 1995.

[13] Fireworks AI. How cursor built fast apply using the speculative decoding api. https://fireworks.ai/blog/cursor, 2024. Accessed: 2026-05-05.

[14] Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. Perfbench: Can agents resolve real-world performance bugs?, 2025. URL https://arxiv.org/abs/2509.24091.

[15] Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. Jsonschemabench: A rigorous benchmark of structured outputs for language models, 2025. URL https://arxiv.org/abs/2501.10868.

[16] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6: 325–338, 2024.

[17] In Gim, Zhiyao Ma, Seung-seob Lee, and Lin Zhong. Pie: A programmable serving system for emerging llm applications. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pp. 415–430, 2025.

[18] Raja Gond, Aditya K Kamath, Ramachandran Ramjee, and Ashish Panwar. Llm-42: Enabling determinism in llm inference with verified speculation. *arXiv preprint arXiv:2601.17768*, 2026.

[19] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[20] Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma, Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi Li, Ruibo Liu, Yue Wang, et al. Codeeditorbench: Evaluating code editing capability of large language models. *arXiv preprint arXiv:2404.03543*, 2024.

[21] Ping Guo, Chenyu Zhu, Siyuan Chen, Fei Liu, Xi Lin, Zhichao Lu, and Qingfu Zhang. EvoEngineer: Mastering automated CUDA kernel code evolution with large language models. *ArXiv*, abs/2510.03760, 2025. URL https://arxiv.org/abs/2510.03760.

[22] Pouya Hamadanian, Pantea Karimi, Arash Nasr-Esfahany, Kimia Noorbakhsh, Joseph Chandler, Ali ParandehGheibi, Mohammad Alizadeh, and Hari Balakrishnan. Glia: A human-inspired ai for automated systems design and optimization, 2026. URL https://arxiv.org/abs/2510.27176.

[23] Awni Hannun, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert. MLX: Efficient and flexible machine learning on Apple silicon, 2023. URL https://github.com/ml-explore/mlx.

[24] Horace He and Thinking Machines Lab. Defeating nondeterminism in llm inference. *Thinking Machines Lab: Connectionism*, 2025. doi: 10.64434/tml.20250910. https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

[25] Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Yin Zheng, Zejian Yuan, and Zejun Ma. Swe-perf: Can language models optimize code performance on real-world repositories?, 2025. URL https://arxiv.org/abs/2507.12415.

[26] Paul Hübner, Andong Hu, Ivy Peng, and Stefano Markidis. Apple vs. oranges: Evaluating the apple silicon m-series SoCs for HPC performance and efficiency, 2025. URL https://arxiv.org/abs/2502.05317.

[27] Geoffrey Huntley. Everything is a ralph loop. https://ghuntley.com/loop/, 1 2026. Blog post. Accessed: 2026-05-06.

[28] Shashwat Jaiswal, Kunal Jain, Yogesh Simmhan, Anjaly Parayil, Ankur Mallick, Rujia Wang, Renee St Amant, Chetan Bansal, Victor Ruhle, Anoop Kulkarni, et al. Sageserve: Optimizing llm serving on cloud data centers with forecast aware auto-scaling. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 9(3):1–24, 2025.

[29] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? *ArXiv*, abs/2310.06770, 2023. URL https://arxiv.org/abs/2310.06770.

[30] 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):1–27, 2025.

[31] Keisuke Kamahori, Wei-Tzu Lee, Atindra Jha, Rohan Kadekodi, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Voxserve: Streaming-centric serving system for speech language models. *arXiv preprint arXiv:2602.00269*, 2026.

[32] Pantea Karimi, Kimia Noorbakhsh, Mohammad Alizadeh, and Hari Balakrishnan. Improving coherence and persistence in agentic AI for system optimization, 2026. URL https://arxiv.org/abs/2603.21321.

[33] Andrej Karpathy. autoresearch: An autonomous LLM research loop. https://github.com/karpathy/autoresearch, 2026. Accessed: 2026-05-06.

[34] Manjunath Kudlur, Evan King, James Wang, and Pete Warden. Moonshine v2: Ergodic streaming encoder asr for latency-critical speech applications. *arXiv preprint arXiv:2602.12241*, 2026.

[35] Thomas Kwa, Ben West, Joel Becker, et al. Measuring AI ability to complete long software tasks. https://metr.org/blog/2025-03-19-measuring-ai-ability-to-complete-long-tasks/, 3 2025.

[36] 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 ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[37] LangChain. DeepAgents. https://github.com/langchain-ai/deepagents, 2025. Accessed: 2026-05-06.

[38] Tue Le, Minh Vu Thai Pham, Dung Nguyen Manh, Huy Nhat Phan, and Nghi D. Q. Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios, 2026. URL https://arxiv.org/abs/2512.18470.

[39] Kangwook Lee. Investigating how Codex context compaction works. https://kangwooklee.com/blogs/codex_context_compaction.html, 2026. Accessed: 2026-05-07.

[40] Linzhang Li, Yixin Dong, Guanjie Wang, Ziyi Xu, Alexander Jiang, and Tianqi Chen. Xgrammar-2: Dynamic and efficient structured generation engine for agentic llms, 2026. URL https://arxiv.org/abs/2601.04426.

[41] Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Haim Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, Omri Abend, Raz Alon, Tomer Asida, Amir Bergman, Roman Glozman, Michael Gokhman, Avshalom Manevich, Nir Ratner, Noam Rozen, Erez Shwartz, Mor Zusman, and Yoav Shoham. Jamba: A hybrid transformer-mamba language model. *ArXiv*, abs/2403.19887, 2024. URL https://arxiv.org/abs/2403.19887.

[42] Wilson Lin. Scaling long-running autonomous coding. https://cursor.com/blog/scaling-agents, 1 2026. Cursor blog. Accessed: 2026-05-06.

[43] Shu Liu, Mert Cemri, Shubham Agarwal, Alexander Krentsel, Ashwin Naren, Qiuyang Mang, Zhifei Li, Akshat Gupta, Monishwaran Maheswaran, Audrey Cheng, Melissa Pan, Ethan Boneh, Kannan Ramchandran, Koushik Sen, Alexandros G. Dimakis, Matei Zaharia, and Ion Stoica. SkyDiscover: A flexible framework for AI-driven scientific and algorithmic discovery, 2026. URL https://skydiscover-ai.github.io/blog.html.

[44] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E Gonzalez, et al. Autellix: An efficient serving engine for llm agents as general programs. *arXiv preprint arXiv:2502.13965*, 2025.

[45] Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. Swe-fficiency: Can language models optimize real-world repositories on real workloads?, 2025. URL https://arxiv.org/abs/2511.06090.

[46] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News*, 41(1):461–472, 2013.

[47] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. Jitsu: Just-In-Time summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 559–573, 2015.

[48] Henry Massalin and Calton Pu. Threads and input/output in the synthesis kernel. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, pp. 191–201, 1989.

[49] Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems (TOCS)*, 19(2):217–251, 2001.

[50] William Merrill, Yanhong Li, Tyler Romero, Anej Svete, Caia Costello, Pradeep Dasigi, Dirk Groeneveld, David Heineman, Bailey Kuehl, Nathan Lambert, et al. Olmo hybrid: From theory to practice and back. *arXiv preprint arXiv:2604.03444*, 2026.

[51] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

[52] NVIDIA. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM, 2023.

[53] NVIDIA, Aaron Blakeman, Aarti Basant, Abhinav Khattar, Adithya Renduchintala, Akhiad Bercovich, Aleksander Ficek, Alexis Bjorlin, Ali Taghibakhsh, Amala Sanjay Deshmukh, Ameya Sunil Mahabaleshwarkar, Andrew Tao, Anna Shors, Ashwath Aithal, Ashwin Poojary, Ayush Dattagupta, Balaram Buddharaju, Bobby Chen, Boris Ginsburg, Boxin Wang, Brandon Norick, Brian Butterfield, Bryan Catanzaro, Carlo del Mundo, Chengyu Dong, Christine Harvey, Christopher Parisien, Dan Su, Daniel Korzekwa, Danny Yin, Daria Gitman, David Mosallanezhad, Deepak Narayanan, Denys Fridman, Dima Rekesh, Ding Ma, Dmytro Pykhtar, Dong Ahn, Duncan Riach, Dusan Stosic, Eileen Long, Elad Segal, Ellie Evans, Eric Chung, Erick Galinkin, Evelina Bakhturina, Ewa Dobrowolska, Fei Jia, Fuxiao Liu, Gargi Prasad, Gerald Shen, Guilin Liu, Guo Chen, Haifeng Qian, Helen Ngo, Hongbin Liu, Hui Li, Igor Gitman, Ilia Karmanov, Ivan Moshkov, Izik Golan, Jan Kautz, Jane Polak Scowcroft, Jared Casper, Jarno Seppanen, Jason Lu, Jason Sewall, Jiaqi Zeng, Jiaxuan You, Jimmy Zhang, Jing Zhang, Jining Huang, Jinze Xue, Jocelyn Huang, Joey Conway, John Kamalu, Jon Barker, Jonathan Cohen, Joseph Jennings, Jupinder Parmar, Karan Sapra, Kari Briski, Kateryna Chumachenko, Katherine Luna, Keshav Santhanam, Kezhi Kong, Kirthi Sivamani, Krzysztof Pawelec, Kumar Anik, Kunlun Li, Lawrence McAfee, Leon Derczynski, Lindsey Pavao, Luis Vega, Lukas Voegtle, Maciej Bala, Maer Rodrigues de Melo, Makesh Narsimhan Sreedhar, Marcin Chochowski, Markus Kliegl, Marta Stepniewska-Dziubinska, Matthieu Le, Matvei Novikov, Mehrzad Samadi, Michael Andersch, Michael Evans, Miguel Martinez, Mike Chrzanowski, Mike Ranzinger, Mikolaj Blaz, Misha Smelyanskiy, Mohamed Fawzy, Mohammad Shoeybi, Mostofa Patwary, Nayeon Lee, Nima Tajbakhsh, Ning Xu, Oleg Rybakov, Oleksii Kuchaiev, Olivier Delalleau, Osvald Nitski, Parth Chadha, Pasha Shamis, Paulius Micikevicius, Pavlo Molchanov, Peter Dykas, Philipp Fischer, Pierre-Yves Aquilanti, Piotr Bialecki, Prasoon Varshney, Pritam Gundecha, Przemek Tredak, Rabeeh Karimi, Rahul Kandu, Ran El-Yaniv, Raviraj Joshi, Roger Waleffe, Ruoxi Zhang, Sabrina Kavanaugh, Sahil Jain, Samuel Kriman, Sangkug Lym, Sanjeev Satheesh, Saurav Muralidharan, Sean Narenthiran, Selvaraj Anandaraj, Seonmyeong Bak, Sergey Kashirsky, Seungju Han, Shantanu Acharya, Shaona Ghosh, Sharath Turuvekere Sreenivas, Sharon Clay, Shelby Thomas, Shrimai Prabhumoye, Shubham Pachori, Shubham Toshniwal, Shyamala Prayaga, Siddhartha Jain, Sirshak Das, Slawek Kierat, Somshubra Majumdar, Song Han, Soumye Singhal, Sriharsha Niverty, Stefania Alborghetti, Suseella Panguluri, Swetha Bhendigeri, Syeda Nahida Akter, Szymon Migacz, Tal Shiri, Terry Kong, Timo Roman, Tomer Ronen, Trisha Saar, Tugrul Konuk, Tuomas Rintamaki, Tyler Poon, Ushnish De, Vahid Noroozi, Varun Singh, Vijay Korthikanti, Vitaly Kurin, Wasi Uddin Ahmad, Wei Du, Wei Ping, Wenliang Dai, Wonmin Byeon, Xiaowei Ren, Yao Xu, Yejin Choi, Yian Zhang, Ying Lin, Yoshi Suhara, Zhiding Yu, Zhiqi Li, Zhiyu Li, Zhongbo Zhu, Zhuolin Yang, and Zijia Chen. Nemotron-H: A family of accurate and efficient hybrid mamba-transformer models, 2025. URL https://arxiv.org/abs/2504.03624.

[54] OpenAI. Predicted outputs. https://platform.openai.com/docs/guides/predicted-outputs, 2024. OpenAI API documentation. Accessed: 2026-05-05.

[55] OpenAI. OpenAI Codex CLI. https://github.com/openai/codex, 2025. Accessed: 2026-05-06.

[56] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *ArXiv*, abs/2502.10517, 2025. URL https://arxiv.org/abs/2502.10517.

[57] Rui Pan, Zhuang Wang, Zhen Jia, Can Karakus, Luca Zancato, Tri Dao, Yida Wang, and Ravi Netravali. Marconi: Prefix caching for the era of hybrid LLMs. *ArXiv*, abs/2411.19379, 2024. URL https://arxiv.org/abs/2411.19379.

[58] Ori Press, Brandon Amos, Haoyu Zhao, Yikai Wu, Samuel K. Ainsworth, Dominik Krupke, Patrick Kidger, Touqir Sajed, Bartolomeo Stellato, Jisun Park, Nathanael Bosch, Eli Meril, Albert Steppi, Arman Zharmagambetov, Fangzhao Zhang, David Perez-Pineiro, Alberto Mercurio, Ni Zhan, Talor Abramovich,

Kilian Lieret, Hanlin Zhang, Shirley Huang, Matthias Bethge, and Ofir Press. Algotune: Can language models speed up general-purpose numerical programs?, 2025. URL https://arxiv.org/abs/2507.15887.

[59] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *International Conference on Machine Learning (ICML)*, 2023. URL https://arxiv.org/abs/2212.04356.

[60] Atharva Sehgal, James Hou, Swarat Chaudhuri, Jennifer J. Sun, and Yisong Yue. Formulacode: Evaluating agentic superoptimization on large codebases. In *ICML 2025 Workshop on Programmatic Representations for Agent Learning*, 2025. URL https://openreview.net/forum?id=CMdtl83aZF.

[61] Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025. URL https://github.com/algorithmicsuperintelligence/openevolve.

[62] Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. Gso: Challenging software optimization tasks for evaluating swe-agents, 2025. URL https://arxiv.org/abs/2505.23671.

[63] Prabhu Vellaisamy, Thomas Labonte, Sourav Chakraborty, Matt Turner, Samantika Sury, and John Paul Shen. Characterizing and optimizing llm inference workloads on cpu-gpu coupled architectures. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 49–61. IEEE, 2025.

[64] vLLM Team. Hybrid KV cache manager — vLLM documentation. https://docs.vllm.ai/en/stable/design/hybrid_kv_cache_manager/, 2025.

[65] Peiding Wang, Li Zhang, Fang Liu, Yinghao Zhu, Wang Xu, Lin Shi, Xiaoli Lian, Minxiao Li, Bo Shen, and An Fu. Efficientedit: Accelerating code editing via edit-oriented speculative decoding. *arXiv preprint arXiv:2506.02780*, 2025.

[66] Xinyu Jessica Wang, Haoyue Bai, Yiyou Sun, Haorui Wang, Shuibai Zhang, Wenjie Hu, Mya Schroder, Bilge Mutlu, Dawn Song, and Robert D Nowak. The long-horizon task mirage? diagnosing where and why agentic systems break, 2026. URL https://arxiv.org/abs/2604.11978.

[67] Nina Wiedemann, Quentin Leboutet, Michael Paulitsch, Diana Wofk, and Benjamin Ummenhofer. KernelFoundry: Hardware-aware evolutionary GPU kernel optimization, 2026. URL https://arxiv.org/abs/2603.12440.

[68] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[69] Junde Wu, Minhao Hu, Jiayuan Zhu, Jiazhen Pan, Yuyuan Liu, Min Xu, and Yueming Jin. Git context controller: Manage the context of agents by agentic git, 2026. URL https://arxiv.org/abs/2508.00031.

[70] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

[71] Jinheng Xie, Zhenheng Yang, and Mike Zheng Shou. Show-o2: Improved native unified multimodal models, 2025. URL https://arxiv.org/abs/2506.15564.

[72] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[73] Nan Yang, Tao Ge, Liang Wang, Binxing Jiao, Daxin Jiang, Linjun Yang, Rangan Majumder, and Furu Wei. Inference with reference: Lossless acceleration of large language models. *arXiv preprint arXiv:2304.04487*, 2023.

[74] Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. *arXiv preprint arXiv:2412.06464*, 2024.

[75] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving, 2025. URL https://arxiv.org/abs/2501.01005.

[76] Peiqi Yin, Jiangyun Zhu, Han Gao, Chenguang Zheng, Yongxiang Huang, Taichang Zhou, Ruirui Yang, Weizhi Liu, Weiqing Chen, Canlin Guo, et al. vllm-omni: Fully disaggregated serving for any-to-any multimodal models. *arXiv preprint arXiv:2602.02204*, 2026.

[77] 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 (OSDI 22)*, pp. 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

[78] Chen Zhang, Kuntai Du, Shu Liu, Woosuk Kwon, Xiangxi Mo, Yufeng Wang, Xiaoxuan Liu, Kaichao You, Zhuohan Li, Mingsheng Long, et al. Jenga: Effective memory management for serving llm with heterogeneity. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pp. 446–461, 2025.

[79] 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, 2024. URL https://arxiv.org/abs/2312.07104.

[80] Yusheng Zheng, Yanpeng Hu, Wei Zhang, and Andi Quinn. Towards agentic OS: An LLM agent framework for linux schedulers, 2025. URL https://arxiv.org/abs/2509.01245.

A Detailed Evaluation Scenarios

This appendix provides additional details for each evaluation scenario. Across scenarios, the agentic loop receives model weights, a HuggingFace Transformers reference implementation [68], accuracy-checking scripts, performance-evaluation scripts, and natural-language instructions. The generated system is evaluated against the reference implementation for correctness and against one or more baseline serving systems for performance.

Scenario A: Standard LLM serving on H100. *Architecture.* Llama-3.1-8B-Instruct [19] is a dense decoder-only Transformer with grouped-query attention (32 query heads sharing 8 key/value heads), RoPE positional encodings, SwiGLU MLPs, and a 128k-token context window. This configuration is the design center of every modern serving stack: dense decoder-only inference on data-center GPUs is precisely what vLLM, SGLang, and TensorRT-LLM are tuned for, and the standard optimization stack is by now well-known — paged KV cache [36], continuous batching [77], CUDA graphs, FlashAttention/FlashInfer kernels [10, 75], and operator fusion.

Workload and metric. An open-loop synthetic load generator drives the system at four request rates (8, 32, 64, 128 req/s). Request arrivals follow a Poisson process (exponential inter-arrival times), and requests are launched independently of completion. Each rate is run for 60 seconds with seed 42. Prompts are sampled uniformly from a predefined prompt pool, and output length is capped at `max_tokens=128`. Most requests emit exactly 128 chunks, and generation uses temperature 0. We report token-generation throughput, mean TTFT, and mean TPOT relative to vLLM. Greedy decoding outputs are checked against the Hugging Face Transformers reference.

Scenario B: Distributed MoE serving on AMD MI300A. *Architecture.* Qwen3.5-397B-A17B-MXFP4 is a sparse mixture-of-experts model with approximately 397B total parameters, approximately 17B active parameters per token, and 512 routed experts in MXFP4 precision. It interleaves Gated DeltaNet linear-attention layers with recurrent state, full-attention layers with paged KV state, and shared/routed MoE computation, so prefix reuse must preserve both conventional attention KV and DeltaNet recurrent state at once. The engine runs on one node of 4× AMD MI300A (ROCm, gfx942, unified host/accelerator memory): dense projections, attention, and DeltaNet are tensor-sharded across the four ranks, and MoE experts are partitioned across the same ranks with partial outputs combined collectively.

Workload and metric. A deterministic multi-turn chat benchmark. Concurrent sessions are ramped from 1 to 192 in a load-ramp benchmark, with 3–6 turns per session, 80–300 generated tokens per turn, 1–8 s of think time between turns, the full accumulated conversation resent on every turn, and greedy decoding at fixed completion lengths. A response fails the run if it is dropped, errored, empty, or truncated, or if it misses the SLO of p95 TPOT ≤ 250 ms and p95 TTFT ≤ 10 s. We report peak goodput in tokens/s, the token throughput of SLO-compliant responses, taking the best value across the load ramp for each system. The comparison is against an optimized SGLang v0.5.18 baseline using MTP speculative decoding (k=3). SGLang peaks at 48 concurrent sessions, while our engine

peaks at 96, where we report $2{,}242$ tok/s. Every accepted change must also pass a numerics gate, with at least 1,074 of 1,088 teacher-forced top-1 tokens matching the reference and 5 of 5 greedy reference completions. Because node-to-node variation is roughly $\pm 10\%$, gains are measured as same-node paired runs. Performance evaluation uses a 20% held-out set.

Optimization opportunity. Most of the gain from 12.8 to $2{,}242$ tok/s comes from a long campaign of general serving-engine engineering, including HIP-level decode and prefill graph capture, fused MoE and attention kernels, chunked DeltaNet recurrence, split-K attention, per-shape GEMM tuning, and an overlap scheduler. Because the model is tensor- and expert-sharded across four ranks, cross-GPU communication is a first-order cost here in a way it is not in the other scenarios, and a substantial share of the campaign's gains are communication-focused, including a custom all-reduce collective, fusing the decode-path all-reduce with the residual add and RMSNorm into a single collective, wide HIP MoE kernels, rank-agreed validation, and a vocab-parallel LM head. Later, workload-specific refinements build on this tuned base. Because every turn resends the entire conversation, the engine reuses both the attention KV cache and the DeltaNet recurrent state from prior turns rather than recomputing them, and folds the turn-opening chat-template suffix (roughly five tokens) into decode steps already scheduled for the active batch instead of issuing it as a separate small prefill call (9.6% on top of an already-tuned system). A DeltaNet state snapshot costs about 47 MiB per conversation point, so the prefix cache holds far fewer conversations than it would for a pure-attention model, and improving its eviction policy gives a further large jump. The final step is speculative decoding with the model's built-in MTP head. Making it correct required exact DeltaNet recurrence and causal convolution over the drafted tokens, plus rollback of recurrent state on rejected drafts that stays safe under graph capture.

Why stock SGLang does not run. SGLang v0.5.18 does not serve this model on MI300A as released. The main blocker is the expert path. Its MXFP4 MoE relies on AITER kernels, and on gfx942 these abort with an unsupported fp4x2 output type. The CK a4w4 kernels are built for gfx950 only, the FlyDSL a4w4 path fails, and CK's MXFP4 grouped GEMM needs the gfx950 scaled MFMA instruction. No shipped kernel can consume MXFP4 weights on this part. Several smaller problems also block a usable launch. The shipped AITER GEMM tuning table is keyed to gfx950 with 256 CUs while MI300A reports gfx942 with 228, so no row matches and dense projections fall back to untuned kernels. Checkpoint loading takes about an hour because per-tensor expert materialization is serialized, far above the 22 s storage floor. Rank initialization crashes when setting GPU process affinity under a Slurm-restricted cpuset. The hybrid-model cache sizing fails with a negative Mamba cache size when free memory is uneven across ranks. AITER also builds some attention kernel variants lazily, which stalls the first real request by about 105 s. VibeServe first replaced the expert path with a Triton w4a16 fused-MoE kernel that dequantizes MXFP4 in-kernel, which brought the server up at about 28 tok/s on a single stream. It also threaded the expert weight loader, which cut load time to about 4 minutes, and fixed the remaining launch failures. Profiling showed the Triton kernel at roughly $26\times$ above its byte bound and limited by dequantization and latency rather than bandwidth, and retuning its tiling did not help.

Tuning baseline SGLang. We ran VibeServe on SGLang itself before comparing. It replaced the gfx942 Triton MoE fallback with a custom fused HIP MXFP4 MoE kernel, added a skinny BF16 GEMM path for small-batch decode, enabled the model's NEXTN/MTP speculative-decoding head, generated PyTorch TunableOp GEMM tables for every captured draft/verify/decode shape, padded and re-bucketed prefill GEMMs, and made other changes. Once SGLang ran, this tuning improved its goodput by a further 42%. The reported $2.33\times$ gain is measured against this VibeServe-tuned version.

Why the tuned SGLang still falls short. The remaining gap comes from changes that SGLang's serving abstractions cannot express as configuration or kernel substitution. Jointly placing DeltaNet recurrent state and MoE expert shards across the same four ranks under MI300A's unified host/accelerator memory — where snapshot pools, KV pools, captured-graph allocations, and process RSS all compete for the same physical capacity — requires a memory layout that SGLang's parallelism and cache-allocation interfaces do not expose as tuning knobs. Folding the turn-opening chat-template suffix into an already-scheduled decode step similarly requires restructuring the request lifecycle itself, which means invasive SGLang changes. The system generated by VibeServe implements both directly.

Scenario C: Code editing with predicted outputs. *Architecture and workload.* Qwen3-32B [72] is a dense decoder-only Transformer (with Q/K-norm and grouped-query attention) served on an NVIDIA H100. The workload is code editing under OpenAI's predicted-outputs interface [54]: each request carries both an instruction and a string of *predicted output tokens* representing the most likely answer. This prediction is naturally available for code editing tasks, since the pre-edit file is typically a near-prediction of the post-edit file, and prior work and deployed systems show that the overlap is large in practice [13, 73, 65]. We report single-batch latency on CodeEditorBench [20].

Optimization opportunity. The predicted-outputs interface is a degenerate case of speculative decoding in which the draft is the user-supplied prediction at zero draft-model cost. The serving system feeds a window of K predicted tokens through the target model in a single forward pass and commits the longest prefix whose argmax matches the prediction. On a mismatch, it falls back to ordinary autoregressive decoding for one token and then resumes from the prediction. With high overlap, latency drops by nearly a factor of K with no additional compute.

Why generic systems cannot exploit this. Standard systems like vLLM or SGLang support speculative decoding, but do not support the predicted-outputs interface. Predicted outputs are a different request type. The engine needs a per-request token stream, a verifier loop that consumes from that stream until divergence, and well-defined fallback semantics. Adding this to vLLM means non-trivial changes to the scheduler, sequence-group state, and sampler. A bespoke system can build the request lifecycle directly around the predicted-outputs API.

Scenario D: Prompt caching for a hybrid architecture. *Architecture.* Olmo-Hybrid-7B [50] interleaves Gated DeltaNet layers [74] with standard self-attention layers. Gated DeltaNet is a linear-attention/SSM-style layer, and its per-sequence state is a fixed-size matrix that is updated recurrently as tokens arrive, in contrast to attention's KV cache, which grows linearly with sequence length. The two layer types therefore carry different kinds of state, and the cache layout, eviction policy, and sharing semantics differ between them.

Workload and metric. A RAG-like workload in which every request shares a 32k-token system prefix, appends a 128-token request-specific suffix, and produces 128 output tokens. We report token-generation throughput at concurrency 20 on an NVIDIA L4 (24 GB), where memory pressure rules out keeping uncompressed per-request state copies.

Optimization opportunity. Prefix sharing across requests is a standard technique, but a hybrid model needs two cache mechanisms in parallel: KV blocks for attention layers and a snapshot of the recurrent state at the prefix boundary for each DeltaNet layer. Knowing the workload at design time, the agent can specialize the system to this case and avoid the overhead of supporting prompt caching for generic workloads.

Why generic systems cannot exploit this. vLLM and SGLang were architected around the attention KV cache. First-class hybrid-KV support is recent and limited [64, 57]. Sharing the recurrent state across requests requires snapshotting at prefix boundaries, which incurs significant memory overhead to support generic cases, especially on hardware with limited memory capacity like L4. vLLM does expose a GDN-layer prefix-caching setting, `mamba_cache_mode`, which we enable in the baseline, but its two policies are both a poor fit for this workload's 32k-token shared prefix followed by a 128-token request-specific suffix: `"all"` snapshots the GDN state every 80 tokens (an interval sized from the relative sizes of the attention KV cache and the GDN state), producing a large number of snapshots that are never reused and overwhelm GPU memory capacity, while `"align"` snapshots only at the end of each prefill, which is also not reusable since the final 128 tokens differ across requests and the complete prefill is therefore never shared. This generic two-policy design cannot express the boundary this workload actually needs (a snapshot exactly at the 32k shared/unique split), which is what the bespoke system implements directly.

Scenario E: Streaming ASR with sliding-window encoder attention. *Architecture.* Moonshine Streaming medium [34] is an encoder-decoder ASR model designed for low-latency streaming. The encoder uses *sliding-window* attention over audio frames, so previously encoded frames remain valid as new audio arrives; only the new tail needs to be encoded. The decoder is a small autoregressive Transformer that emits text tokens conditioned on encoder outputs. In contrast, Whisper [59], the standard ASR baseline, encodes a full clip in a single pass and is not designed for incremental encoding.

Workload and metric. 32 concurrent streaming clients, each sending a 2-second audio chunk every 2 seconds. We report time-to-first-token (TTFT) per chunk, which captures responsiveness for interactive transcription. We compare against a vLLM-plugin Moonshine baseline.

Optimization opportunity. Sliding-window attention permits *encoder-output caching*: each chunk encodes only the new tail and reuses the previous encoder outputs to feed the decoder. The system needs (i) a per-stream encoder cache aligned with the sliding window, (ii) eviction synchronized with the window's stride, and (iii) a scheduler that batches per-chunk encoder work alongside per-token decoder work across many concurrent streams.

Why generic systems cannot exploit this. vLLM can support the Moonshine Streaming model via a plugin, but it cannot support encoder prompt caching without significant modifications to its code, which leads to redundant computation for streaming applications. In contrast, the bespoke system can optimize around Moonshine's specific sliding-window attention and expose the encoder layer to per-stream cache management.

Scenario F: Local constrained decoding for JSON generation. *Architecture and target.* Llama-3.1-8B-Instruct [19] (8-bit MLX quantization) on a MacBook Pro (Apple M3 Pro, 36 GB unified memory) running macOS 26.5 (build 25F5042g). We optimize single-stream end-to-end latency at $T{=}0$ on JSONSchemaBench [15], a corpus of $\sim$9,558 real-world JSON schemas drawn from json-schema-corpus, GlaiveAI function-call schemas, and Kubernetes schemas, which is used to measure both the speed and the schema-feature coverage of constrained-decoding engines. The schemas are partitioned into 10 splits along two axes (domain and complexity): function-calling (GlaiveAI-2K, 1,707), operational/resource-access APIs (Snowplow 403, Washington Post 125), Kubernetes API configurations (1,064), a curated JSONSchemaStore set (492), and five GitHub-sourced *Misc* tiers graded by constraint complexity (Trivial 444, Easy 1,943, Medium 1,976, Hard 1,240, Ultra 164). The distribution is skewed toward GitHub Easy/Medium and function-call schemas, with progressively rarer Hard/Ultra tails that stress less common JSON Schema features. The workload is held fixed across seven VibeServe iterations and we report p50 latency.

Optimization opportunity. Three techniques compose. First, JSON-schema-constrained decoding with XGrammar [40] masks tokens that would violate the schema and applies *jump-forward* to skip over deterministic tokens implied by the schema. Second, speculative decoding uses Llama-3.2-1B-Instruct (4-bit, MLX) as the draft against the 8B-8bit target with $K{=}4$ draft tokens per step; the smaller draft is preferred because its lower per-step cost outweighs its lower acceptance rate. Third, raising mlx_lm's prefill chunk size from the default 512 to 2048 lets a $\sim$1300-token prompt prefill in a single chunk.

Why generic systems cannot exploit this. vLLM and SGLang implement constrained decoding well, but only on CUDA backends. mlx_lm runs on Apple Silicon but lacks both XGrammar integration and a speculative-decoding pipeline. Beyond the missing backend, the wins here demand integration deeper than a generic structured-output API allows. The schema must be enforced inside the decoder via a per-token XGrammar bitmask with grammar-aware termination, and that bitmask must coexist with speculative decoding's rollback semantics: draft tokens may be partially accepted, so a constrained decoder that assumes monotonic token consumption silently drifts on rejection (VibeServe hit two such MLX correctness bugs during the study). The performance levers are similarly non-generic: `prefill_step_size=2048` is an MLX-specific fix for the interaction between $\sim$1300-token prompts and cache chunking; XGrammar's `any_whitespace=False` and compact separators change token counts and thus latency; and several plausible generic optimizations (larger draft, different K, KV quantization, forced-token jump-forward) regressed in this configuration. Residual failures (unbounded `patternProperties`, nested arrays, approximate `oneOf`/`anyOf` unions) need schema-aware decoding behavior that off-the-shelf APIs do not expose.

Scenario G: Local image generation with a unified vision-language model. *Architecture.* Show-o2 [71] is a unified vision-language model whose forward pass interleaves autoregressive (AR) text-token generation (a Qwen2.5-1.5B body) with diffusion-style image-token refinement (a 10-block diffusion head with SigLIP-based image conditioning). Each generation step is partly an AR decode (text body, with prefix-KV cache) and partly a denoising step (head, with classifier-free guidance over conditional and unconditional branches). The control flow does not match either a pure decoder-only LLM or a pure diffusion image model.

Workload and targets. We evaluate text-to-image generation at 432×432 resolution with 20 sampler steps in two deployments. (i) MacBook Pro (Apple M3 Pro, 36 GB unified memory) running macOS 26.5 (build 25F5042g), where we measure single-stream warm-min latency against the Show-o2 PyTorch-MPS reference implementation. (ii) NVIDIA H100, where we measure single-request latency at a fixed prompt and seed against the baseline's PyTorch implementation wrapped as a server. The benchmark client and server share the same container over the loopback interface and exchange raw PPM frames, so the measured target is serving latency rather than network or encoding overhead.

Accuracy gate. Bitwise reproduction of the baseline is too restrictive when quantization, kernel substitution, or step-skipping is on the table. We provide VibeServe with a custom checker that compares each generated image against the baseline at a fixed prompt, seed, step count, guidance scale, and device profile; the checker accepts an image if it matches the baseline's 432×432 dimensions and meets a quality bar of MAE ≤ 2, PSNR ≥ 35 dB, and local luminance SSIM ≥ 0.98. The same checker gates both the H100 and MacBook variants.

Optimization opportunity. VibeServe ports the body and head to the target backend (MLX on MacBook), elides a redundant SigLIP encode of noisy latents on every step, adds prefix-KV caches on body and head, trims prefill to the active image span, and applies a CFG *stride* that skips the unconditional branch on $K-1$ of every K denoising steps and reuses the cached `v_uncond`. Quantization is restricted to the bandwidth-bound head; weight quantization on the compute-bound body regresses latency. On H100, aggressive trimming and naive batching regress quality. FlashAttention-2, GQA, `torch.compile`, and fp16 alter outputs or produce NaNs. Qwen prefix reuse yields no gain because the text prefix is tiny next to the 730-token image span. The 14.5 s MacBook floor replaces each per-step component with its fp16 kernel-perfect time. See §4.2 (H100) and §4.2 (MacBook) for iteration-level breakdowns.

Why generic systems cannot exploit this. There is no generic serving stack for Show-o2: vLLM does not implement diffusion paths, vLLM-Omni does not include this model, and the reference is a research-grade PyTorch implementation. The AR/diffusion interleaving and the body/head/sampler co-design needed for the wins above do not generalize across models, so adding Show-o2 to a generic stack would be model-specific work that competes for engineering attention with every other model the stack supports. A bespoke system can wire the loop around exactly this control flow.

Baseline configurations. For reproducibility, we list the exact non-default flags used for each scenario's serving-system baseline.

- **Scenario A:** vLLM with `-gpu-memory-utilization 0.95 -max-model-len 8192 -max-num-seqs 256 -max-num-batched-tokens 16384 -performance-mode throughput -optimization-level 2 -async-scheduling`.
- **Scenario B:** SGLang v0.5.18 with environment `SGLANG_USE_AITER=1 SGLANG_USE_AITER_UNIFIED_ATTN=1 SGLANG_MXFP4_MOE_HIP=1 SGLANG_SKINNY_GEMM=1 SGLANG_AITER_GEMM_PAD_M=64 SGLANG_MAMBA_SSM_DTYPE=bfloat16 PYTORCH_TUNABLEOP_ENABLED=1 PYTORCH_TUNABLEOP_TUNING=0` and server flags `-tp 4 -attention-backend aiter -page-size 16 -mem-fraction-static 0.72 -max-total-tokens 787936 -speculative-algorithm NEXTN -speculative-eagle-topk 1 -speculative-num-steps 3 -speculative-num-draft-tokens 4`. Unlike the other scenarios, this baseline also required source-level changes to SGLang itself, not just flags. We describe those changes next.
- **Scenario C:** vLLM vanilla, plus `-gpu-memory-utilization 0.95 -max-model-len 4096`. The speculative-decoding baseline additionally sets `{"method": "eagle3", "model": "AngelSlim/Qwen3-32B_eagle3", "num_speculative_tokens": 7}`.
- **Scenario D:** vLLM vanilla, plus `-dtype bfloat16 -max-model-len 36864 -gpu-memory-utilization 0.92 -enable-prefix-caching`.
- **Scenario E:** vLLM vanilla, with a port of Moonshine into vLLM as an out-of-tree model plugin.

- **Scenario F:** `mlx_lm` with `-temp 0 -max-tokens 512 -decode-concurrency 1 -prompt-concurrency 1 -prompt-cache-size 0 -prefill-step-size 512`.
- **Scenario G:** the vanilla PyTorch reference implementation.

SGLang source modifications for Scenario B. Scenario B's baseline is not stock SGLang plus flags, but SGLang with VibeServe-authored source changes. We summarize those changes at a high level for reproducibility. VibeServe replaced SGLang's gfx942 Triton MoE fallback with a custom fused HIP MXFP4 MoE kernel, added a skinny BF16 GEMM path for small-batch decode, and iteratively refined the MoE kernel with register byte permutation, wider packed-weight loads, and a lower stage-1 threshold. It also changed the serving path to reuse cached-prefix tokenization for the known turn-opening suffix instead of retokenizing it, and to use constant-time expert-weight mapping with threaded, sharded weight loading to reduce startup time (this last change does not affect steady-state throughput). These source changes sit on top of SGLang's existing NEXTN/MTP speculative-decoding support, which we enable through the flags above rather than modify.

Per-role agentic-loop breakdown. Table 1 reports per-role LLM-call counts and active time across evaluation scenarios. Across all scenarios except B, the Implementer dominates active time in every run (47–60%), reflecting that producing and revising candidate code is the most compute-intensive step. The Accuracy Judge is the next-largest contributor (20–30%) and is especially heavy on Scenario D, where the dual KV/recurrent-state cache machinery makes correctness review more involved. The Performance Evaluator runs less often because performance work is gated on a passing accuracy round, and the Orchestrator is consistently a small share (3–7%) at low average agent execution time (34–46 s) since it only selects the next issue and updates long-term memory.

Scenario B departs from this pattern. This is a hard model and hardware combination to optimize, so the Orchestrator reasons through multiple candidate optimization directions each round. That raises its average agent execution time (574 s, 12–17× the other scenarios) and its share of active time (8.7%, above the range of the other scenarios). To save wall-clock time, the Orchestrator also dispatches multiple inner-loop rounds in parallel (§3.2), so independent optimizations proceed at their own pace and trigger correctness and performance measurements independently. These durations are therefore cumulative per-role active time that overlaps across parallel rounds. The 392.8 h of total active time corresponds to a 105 h end-to-end run, an average overlap of about 3.7×. The Performance Evaluator is the largest share of active time (49.7%) and its average agent execution time (2164 s) is well above the other scenarios (175–373 s). Booting a distributed serving engine for a model this large takes longer, and we run on a shared GPU cluster with unpredictable contention, so evaluation jobs can also wait in the queue. Judge and Implementer average agent execution times (1080 s and 1205 s) are also above the other scenarios (139–411 s and 393–648 s).

B Ablation Details

This appendix gives the full comparison tables and per-round detail for the two ablations summarized in §4.3.

Starting-point ablation. VibeServe is seeded from vLLM v0.26.0 (Scenarios D, E) or vLLM-Omni v0.24.1 (Scenario G), instead of an empty repository, and compared against the from-scratch runs of §4.2.

Per-round detail follows for each seeded scenario.

Scenario D: Olmo-Hybrid prompt caching. The agent patches vLLM's OLMo-Hybrid fused conv1d weight loader (a shard-id-recovery bug after `WeightsMapper.apply()`), configures the V1 engine (`AsyncLLM`, chunked prefill, `FULL_AND_PIECEWISE` CUDA graphs), and replaces a per-request global lock with vLLM's native continuous batching. Table 3 traces throughput on the same 20-concurrent-request, 32k-shared-prefix workload as §4.2 across 8 rounds. A controlled A/B at round 5 isolates the effect of prefix caching: enabling it raises throughput from 6.04 to 74.98 tok/s (12.4×) at matched request count, confirming that the seeded run's gains come from prefix reuse rather than incidental engine tuning.

Table 1: Per-role LLM-call breakdown across evaluation scenarios. "Calls" counts agent invocations, "Duration" is cumulative active time, "Share" is the fraction of the scenario's total active time, and "Avg/call" is the mean agent execution time per invocation, from receiving a task to delivering its result. The parenthesis after each scenario is its end-to-end wall-clock time. Roles can run in parallel, so per-role durations overlap and their sum can exceed it. Roles correspond to the inner-loop Implementer, Accuracy Judge, and Performance Evaluator (§3) plus the outer-loop Orchestrator.

Scenario	Role	Calls	Duration (h)	Share	Avg/call (s)
A: Llama-3.1-8B standard (25.0 h)	Orchestrator	120	1.40	5.6%	42
	Implementer	90	13.35	53.4%	534
	Judge	90	6.08	24.3%	243
	Perf. Evaluator	60	4.18	16.7%	251
B: AMD distributed MoE (105 h)	Orchestrator	215	34.3	8.7%	574
	Implementer	396	132.5	33.7%	1205
	Judge	102	30.6	7.8%	1080
	Perf. Evaluator	325	195.4	49.7%	2164
C: Qwen3-32B code edit (8.66 h)	Orchestrator	47	0.61	7.0%	46
	Implementer	36	4.55	52.5%	455
	Judge	36	1.98	22.8%	198
	Perf. Evaluator	23	1.53	17.6%	239
D: Olmo-Hybrid prefix caching (12.54 h)	Orchestrator	47	0.44	3.5%	34
	Implementer	33	5.94	47.4%	648
	Judge	33	3.77	30.1%	411
	Perf. Evaluator	23	2.39	19.1%	373
E: Moonshine streaming (8.71 h)	Orchestrator	49	0.55	6.3%	41
	Implementer	46	5.25	60.3%	411
	Judge	45	1.74	20.0%	139
	Perf. Evaluator	24	1.17	13.4%	175
F: JSON constrained decoding (3.0 h)	Orchestrator	11	0.13	4.3%	43
	Implementer	13	1.42	47.3%	393
	Judge	13	0.85	28.3%	235
	Perf. Evaluator	7	0.60	20.0%	309
G: Show-o2 H100+MBP (14.0 h)	Orchestrator	70	0.85	6.1%	44
	Implementer	53	6.85	48.9%	465
	Judge	51	3.05	21.8%	215
	Perf. Evaluator	34	3.25	23.2%	344

Table 2: Ablating the starting point: seeding VibeServe with an existing codebase instead of building from scratch. Outcomes are mixed: convergence accelerates when the seed's abstractions already fit the target (D), and the agent works around the seed's abstractions otherwise (E, G).

Scenario	Seed	Outcome
D: Olmo-Hybrid prompt caching	vLLM v0.26.0	Matches from-scratch parity by round 4 ($\approx 2\times$ faster)
E: Moonshine streaming ASR	vLLM v0.26.0	Agent implements outside vLLM's abstractions
G: Show-o2 (H100)	vLLM-Omni v0.24.1	Agent implements outside vLLM-Omni's abstractions

Scenario E: Moonshine streaming ASR. vLLM v0.26.0 is present in the environment but unused for inference (vLLM has no Moonshine model), so the agent implements a custom encoder-decoder path outside it, matching the outcome in Table 2. Table 4 traces TTFT p50 over 2-second streaming audio chunks. The jump from round 5 to rounds 6–7 (5902 ms $\rightarrow$ 430 ms) comes from moving to concurrent streaming with bounded per-session admission and a queue/drain microbatcher, not from model-level changes. Per-chunk cost is dominated by the decoder ($\sim$464 ms) rather than the encoder ($\sim$19 ms).

Table 3: Scenario D (Olmo-Hybrid, seeded from vLLM v0.26.0): per-round throughput on the 20-request, 32k-shared-prefix workload.

Round	What changed	Throughput (tok/s)
R2 (cold)	Serving-contract baseline	8.66
R2 (hot)	Same, prefix-warm	10.69
R3	Profiler/tokenizer fix	10.33
R4	Endpoint fix, 20/20 requests stable	76.01
R5 (cache enabled)	Prefix-cache A/B, enabled	74.98
R5 (cache bypassed)	Prefix-cache A/B, bypassed	6.04
Hot final	Best hot measurement	85.98
R7	Profiler contract repair	75.58
R8	Worker-side CUDA profiling	75.86

Table 4: Scenario E (Moonshine, seeded from vLLM v0.26.0): per-round streaming TTFT p50.

Round	What changed	TTFT p50 (ms)	Concurrency
R2	Baseline serving contract	7076.6	1
R4	Streaming encoder partial fast path	6079.5	1
R5	Decoder KV/cross-attention cache	5902.0	1
R6–7	Bounded admission, disconnect-safe streaming	430.2	4
R8	Queue/drain microbatcher	367.0	4
R9	Encoder CUDA-graph buckets (warm)	293.8	4
R11	Recovery to R9 path	295.4	4
R13	Concurrent-function cohort wait	306.2	4
Final	Best stable warm	268.4	4

Scenario G: Show-o2 (H100). The agent builds a serving layer around vLLM-Omni (FastAPI endpoints, VAE integration) but implements Show-o2's active-span forward pass — projecting only the 4,380 active image-latent tokens instead of the full 6,144-token sequence — outside vLLM-Omni's model-serving abstractions, again matching Table 2. Table 5 traces single-request p50 latency at 4 diffusion steps over 13 rounds. The largest single gain (509.6 ms $\rightarrow$ 428.6 ms) comes from the active-span path, and response-serialization tuning contributes a further $\sim$104 ms.

Table 5: Scenario G (Show-o2, seeded from vLLM-Omni v0.24.1): per-round T2I p50 latency, 4 diffusion steps.

Round	What changed	p50 latency (ms)
Baseline	Server contract established	509.6
R6	Active-span fast T2I path (4380/6144 tokens)	428.6
R11	ASGI benchmark forwarder	396.1
R13	Pre-rendered JSON, lower-copy base64	324.8

Agentic pipeline ablation. Table 6 gives the full comparison across all four pipeline configurations on Scenario A (Llama-3.1-8B, H100, Codex GPT-5.5 Medium); §4.3 discusses each ablation's failure mode.

C Existing Assets and Licenses

Table 7 lists the third-party models, datasets, frameworks, and coding-agent harnesses used in this paper, together with their licenses. All assets are used in accordance with their published terms.

D Extended Related Work

This appendix expands the discussion in §5.

Table 6: Ablating the agentic pipeline on Scenario A (Llama-3.1-8B, H100), Codex GPT-5.5 Medium. Each ablation removes one design choice from the full VibeServe pipeline; none converges to a competitive serving system.

Configuration		Peak tok/s	Budget	Failure mode
Single-agent		617	35 h, 1 context	Stuck in local optimization; skips accuracy/benchmark gates
No-skill		1,100	∼35 rounds, 2 days	Exhausts incrementally discoverable optimizations
Outer-loop OpenEvolve	→	537	30 rounds, 1 day	Combines early wins, misses full optimization set
Full VibeServe		≈10k (parity)	60 rounds, 25.0 h	—

Table 7: Existing assets used in this paper, with versions, licenses, and source URLs. Citations point to the paper or release we used; see §4 and §A for how each asset enters the evaluation.

Asset	Type	Version	License	URL
Llama-3.1-8B-Instruct [19]	Model	n/a	Llama 3.1 Community License	https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct
Llama-3.2-1B-Instruct (4-bit, MLX) [19]	Model	n/a	Llama 3.2 Community License	https://huggingface.co/mlx-community/Llama-3.2-1B-Instruct-4bit
Qwen3-32B [72]	Model	n/a	Apache 2.0	https://huggingface.co/Qwen/Qwen3-32B
Olmo-Hybrid-7B [50]	Model	n/a	Apache 2.0	https://huggingface.co/allenai/Olmo-Hybrid-7B
Moonshine Streaming medium [34]	Model	n/a	MIT	https://huggingface.co/UsefulSensors/moonshine-streaming-medium
Show-o2 1.5B-HQ [71]	Model	n/a	Apache 2.0	https://github.com/showlab/show-o
Qwen3.5-397B-A17B-MXFP4	Model	n/a	Apache 2.0	https://huggingface.co/amd/Qwen3.5-397B-A17B-MXFP4
CodeEditorBench [20]	Dataset	n/a	Apache 2.0	https://github.com/CodeEditorBench/CodeEditorBench
JSONSchemaBench [15]	Dataset	n/a	No license specified	https://github.com/guidance-ai/jsonschemabench
vLLM [36]	Framework	v0.19.1	Apache 2.0	https://github.com/vllm-project/vllm
SGLang [79]	Framework	v0.5.18	Apache 2.0	https://github.com/sgl-project/sglang
TensorRT-LLM [52]	Framework	v1.2.1	Apache 2.0	https://github.com/NVIDIA/TensorRT-LLM
HuggingFace Transformers [68]	Framework	v5.5.2	Apache 2.0	https://github.com/huggingface/transformers
MLX / mlx_lm [23]	Framework	v0.31.2	MIT	https://github.com/ml-explore/mlx
PyTorch [2]	Framework	v2.10	BSD-3-Clause	https://github.com/pytorch/pytorch
FlashAttention [10]	Library	fa4-v4.0.0.beta4	BSD-3-Clause	https://github.com/Dao-AILab/flash-attention
FlashInfer [75]	Library	v0.6.6	Apache 2.0	https://github.com/flashinfer-ai/flashinfer
XGrammar [40]	Library	v0.2.0	Apache 2.0	https://github.com/mlc-ai/xgrammar
Codex CLI [55]	Coding-agent harness	v0.125	Apache 2.0	https://github.com/openai/codex
Claude Code [4]	Coding-agent harness	v2.1.122	Anthropic ToS (proprietary)	https://www.anthropic.com/claude-code
DeepAgents [37]	Coding-agent harness	v0.4.11	MIT	https://github.com/langchain-ai/deepagents

Using agents to optimize performance has attracted substantial attention, with recent benchmarks measuring this ability across kernels, numerical routines, repositories, and performance bugs [56, 58, 25, 45, 60, 14, 62]. Agentic optimization systems organize around a few search paradigms, none of which have been applied to greenfield end-to-end system implementation and optimization. *Evolutionary search* maintains a population of agent-generated candidates and selects among them by measured performance. The score numerically encodes the optimization goal, and selection within the population carries that goal forward without summarization, sidestepping the drift that compaction-based handoffs incur. AlphaEvolve [51], OpenEvolve [61], and SkyDiscover [43] provide general outer-loop frameworks, and KernelFoundry [67] and EvoEngineer [21] apply this style to GPU kernel generation under paired correctness/performance gates. As implemented today, these frameworks evolve only small components, e.g., user-marked code regions inside an otherwise-fixed file. A scalar score is sufficient at that scope but cannot encode much of what an end-to-end system needs, e.g., prerequisite dependencies between optimizations (one technique often requires another to be in place first) or internal-bottleneck information that drives the next step, since which component bottlenecks is itself shaped by the agent's prior design choices. *Multi-agent iteration* replaces the population with agents that hypothesize, experiment, and refine designs across rounds, carrying richer reasoning forward to drive the next decision: Glia [22] and Engram [32] use this approach to tune systems policies and heuristics, enabling gains on LLM-serving routing and autoscaling, among other tasks. This reasoning, however, lives within a single context window. Glia's multi-context variant runs independent instances in parallel rather than passing strategic state forward. A third approach simplifies the loop further: *autoresearch* [33] puts one long-running agent in charge of the entire search, tracking candidate ideas across git branches, but is prone to drift within the agent's single context. Adjacent agentic-synthesis work targets similarly bounded scopes, e.g., SchedCP [80] generates Linux scheduling policies without modifying the kernel via LLM-driven techniques. Across these approaches, the agent's output is a bounded policy, heuristic, or module within a larger system. VibeServe is, to our knowledge, the first agentic system under any of these paradigms to do the multi-file coding work needed to design a system itself, creating bespoke LLM serving systems with multiple interconnected internal components.

VibeServe sits within a broader literature on agents performing long-horizon tasks [35, 38, 11, 6, 27, 1]. The standard recourse when a task exceeds a context window is compaction [5, 39], where a session distills its state into a handoff to a fresh successor. Lossy summarization causes drift in both performance and correctness over many rounds, and across optimization sessions an agent must additionally remember which bottleneck to target next, which directions have been tried and discarded, and which platform quirks have surfaced. VibeServe is inspired by industrial prototypes for long-horizon coding agents from Cursor and Anthropic, which introduce explicit task abstractions, shared repository state, and custom agent loops so fresh coding-agent sessions can execute small units of work within multi-week autonomous projects [42, 9]. These prototypes show that agent harnesses can build end-to-end systems from scratch, but they stop short of optimizing them. Git primitives such as commits and branches have also been used to manage agent context and explore distinct strategies across sessions [69]. VibeServe exposes a version-controlled repository interface that allows flexible outer-loop strategies, including the issue-driven loop used in our evaluation. VibeServe couples each task with domain-specific correctness and performance gates, so progress is tracked over validated system designs rather than unconstrained repository edits.