Agentic Workflows are Serverless Applications, so deploy them that way!

Ian Dougherty, Natalie Lambert, Joshua Wang, Ethan Xu, Reto Achermann†, Alexandra Fedorova
The University of British Columbia † *Technical University of Munich*

Abstract

Accelerating generative AI adoption has driven the expansion of data centers, which amass GPUs, DRAM, and SSDs to feed emerging, resource-hungry AI workloads. The serverless cloud model offers a path to improve application resource efficiency by loading instances on demand. However, the suitability of emerging AI workloads for serverless remains insufficiently explored.

We survey the state-of-the-art in serverless hosting for LLM applications and find that: (1) Despite advances in serverless LLM hosting, model loading and initialization processes still dominate startup latency. (2) Agentic AI workloads have not yet been characterized under the serverless context.

We propose a deployment scheme for agentic workloads tailored for serverless, accompanied by pre-warming policies that minimize the idle resource footprint and startup latencies. This paper outlines promising research directions for serverless agents.

1 Introduction

The growth of generative AI workloads has triggered a broad shift in computing demand. Generative AI is resource-intensive, requiring large amounts of accelerators, memory, and storage. Additionally, the adoption of AI workflows in industry is accelerating, particularly through the use of text-based agentic applications [1–3]. To meet the large resource requirements of AI demand, cloud providers are building new datacenters and stockpiling hardware [4]. This shock has led to global supply shortages of GPUs, DRAM, and SSDs [5, 6]. It is clear that to meet the growing demand for AI, we need strategies to more efficiently manage the hardware underlying emerging AI workloads.

The workloads behind generative AI are diverse: spanning LLM inference, model training, fine-tuning, agentic workloads, semantic data stores, and more. *We target LLM inference and agentic workloads*, as agentic applications are rapidly emerging as a significant and growing share of AI workloads [1, 2, 7] and also show potential to be adapted to the serverless cloud model.

Serverless computing can reduce an application's resource consumption by only running instances when they receive requests. Serverless clients benefit from flexible billing: they pay only for the time their applications are executing. To fit serverless deployment, a workload should meet two criteria: (1) Requests should arrive intermittently; a constant demand would not benefit from the serverless model. (2) The deployment should be packageable and able to launch new instances on demand. Instances that take too long to spin up cause cold-start delays, where requests must wait on standby before being serviced. Cold-start delays are a primary source of inefficiency in serverless deployments [8–12].

While large closed-weight models such as ChatGPT [13], Gemini [14], and Claude [15] may benefit less from traditional serverless billing models due to sustained demand. We view recent trends in Small Language Models (SLMs) as a natural fit for the serverless paradigm [16, 17]. Small to mid-sized models can deliver similar performance on domain-specific tasks at a fraction of the size, making them far more efficient than large generalist models [16]. While adapting SLM deployments to the serverless model offers opportunities for hardware sharing, inference also poses significant challenges for efficient deployment in a serverless environment. On-demand loading of model weights can cause significant cold-start delays. Additionally, the large memory footprint of each model can lead to GPU contention. We survey recent work that explores the feasibility of serverless LLM inference [18–25]. However, we find that model cold starts remain a challenge.

In contrast to LLM inference, the suitability of agentic workloads to serverless remains largely unexplored. Agents present two primary challenges to serverless deployment: (1) long-running, non-deterministic execution times that complicate instance placement, and (2) stateful actions that do not natively persist on ephemeral serverless backends. We present a holistic characterization of agentic workloads: exploring inference patterns, resource footprints, and execution

overheads across a variety of agent architectures. We propose that agent workflows naturally lend themselves to serverless deployments. By loading each workflow stage independently, we can achieve greater resource efficiency. Additionally, with knowledge of agent workflow stages and their inference patterns, we can pre-load serverless LLM instances precisely when they are needed. We discuss a practical methodology for this modular deployment scheme and propose our vision for future research in serverless agents.

2 State of the Art

We first provide an overview of the state of the art in serverless systems and the anatomy of LLM inference. We then highlight challenges in serverless-based LLM inference and agentic workflows.

2.1 Traditional Serverless

Through revisiting a decade of innovation in the serverless domain, we envision solutions that parallel problems in serverless AI hosting. Serverless computing has become a well-established business model offering flexible pay-as-you-go compute [26–30]: Clients upload custom functions, i.e., programs that are executed when a request arrives, while cloud providers automatically handle resource provisioning and instance scaling. Sharing of compute resources between function instances and tenants enables greater serving efficiency but also introduces challenges for function isolation and request response times. Cloud providers must service requests within the time frame defined by their Service Level Agreement (SLA), while also minimizing the number of machines used to serve them to increase efficiency. Hence, serverless systems crucially rely on instance placement and autoscaling policies to schedule function instances as requests stream in.

A circular dilemma arises in the serverless paradigm: the clients who benefit most from flexible billing due to their infrequent request patterns are ironically the most challenging to serve effectively. Before a request can be serviced, both the serverless runtime and client function must be loaded and initialized. However, infrequently invoked functions will not remain loaded, leading to cold-start delays as requests wait for an instance to spin up.

The cold-start problem has inspired numerous widely accepted policies for handling function loading and scaling. Since requests feature bursty arrival patterns, keeping instances warm in DRAM for a keep-alive timeout can obviate subsequent cold starts [8,9]. Opportunistic pre-warming loads instance data before requests arrive [24, 31]. Additionally, more functions can be kept warm in DRAM through memory deduplication between instances [8, 11, 12].

Alternatively, cold-start latency can be addressed head-on by optimizing function loading. Traditionally, containers provide isolation of functions between tenants, but using lightweight virtualized language runtimes eliminates the heavy initialization costs of containers [10, 32].

We see variations of these techniques applied for serverless LLMs in §2.3, and extend these concepts to serverless agents in §4.

2.2 Anatomy of an LLM Inference

LLM inference is a heavyweight, multipart operation that plays a central role in generative AI serving. We divide an inference into 3 stages: initialization, prefill, and decode — the latter two occur on every inference, while the former occurs only once at model load time. Understanding the internal structure of an inference is foundational to many systems optimizations for serving LLMs. Initialization is especially important in serverless systems, as it lies on the hot path for requests to cold instances. Additionally, the KV cache reduces duplicate inference computations, thereby speeding up inference across agent prompts that share context. An example depicting inference breakdown across these stages is shown in Figure 1. Out of the total 37.34 seconds runtime, only 4.3 seconds (11.5%) are actually running the inference.

Initialization. Before running inference, the runtime must load model weights into the GPU's VRAM (model load in Figure 1). This process is data-transfer-bound because model weights are large, ranging from tens to several hundreds of gigabytes [33]. In addition, the inference engine performs initialization (init engine in Figure 1), which includes compiling the model (e.g., in the case of PyTorch) and two optimizations: First, the engine determines the sequence of kernels and data transfers to reduce the CPU-GPU communication bottleneck [34, 35]. Secondly, the runtime pre-allocates VRAM for the KV cache, which is computed during the prefill stage and used to accelerate generation [34].

Prefill. The prefill stage ingests the prompt, builds the KV cache, and predicts the first token of the response. This stage is compute-bound, as the entire prompt is processed in parallel. The KV cache is an essential data structure for LLM inference, storing intermediate states that would otherwise require recomputation during each forward pass of the decode stage. The length of the prefill stage is captured by the Time to First Token (TTFT) metric, which is also the first responsive point for the user.

Decode. The decode stage is the autoregressive process that generates each token. Each step takes the last token, queries the KV cache, predicts the next token, and updates the KV cache until a termination token is generated. This stage is memory bandwidth-bound and determines the inference throughput (measured as tokens per second or TPS).

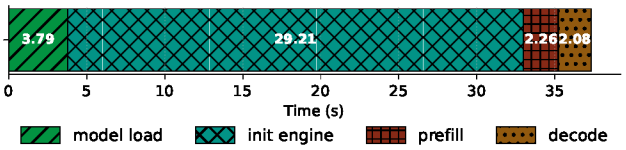


Figure 1: Initialization and inference time breakdown for gpt-oss-20b of a simple summarization task with ∼25k token context and ∼300 tokens response. Model loading is measured from DRAM to VRAM. The first two green and teal phases show the initialization process, while the latter two, prefill and decode stages, show the inference operation.

2.3 Serverless LLMs

As the community around open-weight models has matured, demand for custom cloud inference solutions has emerged. In 2025 alone, the number of models hosted on Hugging Face doubled [17]. This trend reveals a broad interest in open-weight LLMs. Yet, hosting a large catalog of LLMs remains challenging. A serverless inference system serving custom LLMs does not have the luxury of fitting warm model instances in GPU memory. At production scale, there are thousands of custom models to handle [19], many of which observe sparse request patterns [36–38]. The challenge of building serverless LLM infrastructure that mitigates cold starts and efficiently utilizes compute has attracted significant recent attention. We provide an overview of recent work in serverless LLMs.

Model Initialization. Before a cold instance can service requests, the LLM must load into GPU memory, and the inference engine must be initialized. Model initialization is the single largest contributor to cold start latency for serverless LLMs, often exceeding inference time [18, 19, 35]. We cover the state of the art for mitigating both model transfer times and engine initialization costs.

Model loading times can be partially mitigated by effectively managing the storage hierarchy. Prior to inference, a model checkpoint may live in VRAM, DRAM, on disk, or in remote storage; each storage tier adds orders of magnitude in model loading latency [19]. ServerlessLLM [18] routes requests to nodes where model checkpoints are local, avoiding costly model loading from remote storage. When loading from remote storage is necessary, HydraServe [22] provides an optimization through aggregating multiple peers' network bandwidth. Model checkpoints are distributed among peer nodes, and new instances are placed on nodes that minimize network contention.

High-bandwidth GPU-GPU links are a powerful mechanism for moving model weights that are already present within one accelerator's memory. These links can be used for rapid horizontal scaling in response to request bursts [19, 39], achieving over 10× speedup for model loading at 72 billion parameters. Alternatively, task migration can leverage these high-bandwidth links to free up space for new instances loaded from local DRAM [18].

In addition to transferring weights, inference engines run lengthy initialization processes to optimize inference serving. Medusa [35] targets the initialization overhead with materialization, delegating a portion of the initialization to offline preprocessing. Medusa introduces optimizations to remove dynamic memory management during KV cache creation, and further parallelizes this stage alongside model loading. Similarly, CUDA graphs are consistent in structure across consecutive model loads but still depend on dynamically determined data pointer values at runtime. Medusa addresses non-determinism by inserting a shim layer to abstract runtime-dependent kernel parameters.

While there is significant progress in reducing cold-start costs for serverless LLMs, hardware limits still set a lower bound on achievable cold-start times. Loading a mid-sized 34 billion parameter model takes on the order of seconds under typical conditions (e.g., 2s from DRAM, 11s from disk in prior work [19]).

The additional inference engine initialization is also on the order of seconds, even with offline preprocessing [35]. To further mitigate cold starts for LLMs, we need methods to predict which models are accessed before a request arrives.

Pre-warming. Serverless orchestrators often proactively load instance data in anticipation of an incoming request, a process called *pre-warming*. With serverless LLMs, the orchestrator can preload the model into DRAM or even VRAM, thereby significantly reducing time to first token.

DeepServe [19] implements pre-warming by predictively loading up to 1.5TB worth of models to DRAM. Unfortunately, even if the model is present in DRAM, loading to GPU memory can still take significant time. Many prior works assume DRAM pre-warming as an implicit starting point for cold start optimizations [20, 21, 25, 35].

InstaInfer [24] surpasses pre-warming and proposes preloading model data directly to the GPU. InstaInfer does not evaluate against LLMs, but the technique shows promise for other ML inference workloads.

Another optimization angle lies in parameter duplication across models. In particular, fine-tuned models derived from the same base model share over 99% of parameters [20]. ServerlessLoRA [20] leverages this to separate the loading stages of base models from their fine-tuned parameters. By keeping base models in GPU memory, subsequent inference requests require loading only the small, fine-tuned layers. This mechanism effectively prewarms multiple models for the price of one. PipeBoost [25] takes this idea further, and allows for inference to start on the GPU in parallel with the loading of the fine-tuned layers.

Serving Density. By maximizing the utilization of all available compute, serverless LLM systems can serve more requests with the same amount of hardware. Serverless LLM systems that rely solely on GPUs for inference leave CPUs underutilized. SLINFER [21] demonstrates that CPUs equipped with matrix accelerators (Intel AMX) can serve small to mid-sized LLMs without violating SLAs.

CPU compute is not the only underutilized resource; the memory bandwidth-heavy nature of LLM inference leaves plenty of GPU compute headroom [40]. WAGES explores GPU sharing through NVIDIA Multi-Process Service (MPS), partitioning GPUs into multiple instances for simultaneous inference of multiple models [23]. Additionally, GPUs expose fine-grained frequency controls, which WAGES leverages to minimize energy consumption during LLM execution.

2.4 Agent Serving

While agent serving has not been covered under the serverless context, many works [41–44] have explored how agent execution context can aid LLM serving. These serving optimizations happen behind the LLM API, within the inference provider's infrastructure. We discuss how we can leverage the structure of agent workflows and management of KV caches to accelerate LLM inference. These same concepts also extend to serving agents themselves, which we explore in §4.

Control Flow Graphs For our purposes, we define *agents as any logic that encapsulates and extends LLM inference.* Some agents follow predefined static paths, while others have dynamic, recursive workflows that depend on the input query. Existing LLM serving systems optimize for single-shot inference, overlooking the broader application context available in agentic workflows.

Decomposing agentic workloads into their component stages reveals scheduling opportunities to better utilize underlying hardware. The stages of agentic workflows can be modeled as a Control Flow Graph[1] (CFG), where nodes perform inference or invoke tools. Several works leverage an agent's CFG to schedule inference invocations, optimizing end-to-end response times through batching parallel phases and boosting priority of late inference stages [41, 42, 44].

KV Cache Management Agentic workloads reuse significant portions of prompts across LLM invocations, allowing for sharing of intermediate inference results [41–45]. There are two kinds of context reuse: *intra-request* reuse of query-specific context that overlaps between workflow stages, and *inter-request* reuse of static system prompts reused across all queries. CORTEX leverages inter-request reuse through stage separation, routing all inference requests of each stage to a dedicated inference instance [43]. This improves the KV cache hit rate as requests on the same engine share stage-specific system prompts. Alternatively, HELIUM treats KV cache reuse as a first-class citizen by analyzing workflow stages for prompt similarity and proactively collocating similar prompts [42]. Additionally, HELIUM implements a proactive caching strategy that precomputes KV cache data for the static portion of agent prompts. These results are cached globally and preloaded into GPU memory as the inference engine fields similar requests.

[1]Workflows are commonly referred to as DAGs in the literature; many agent workflows are cyclic. We opt for CFG to clarify the term.

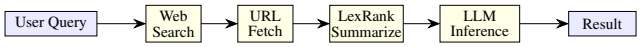


Figure 2: Langchain RAG workflow graph.

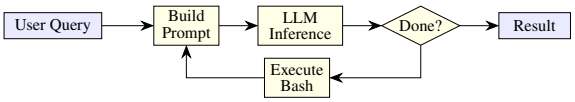


Figure 3: Mini SWE agent workflow graph.

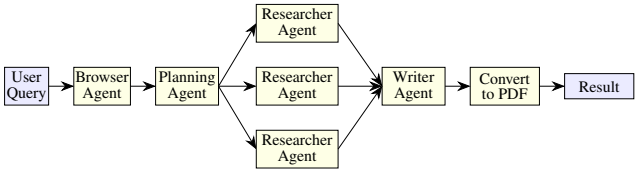


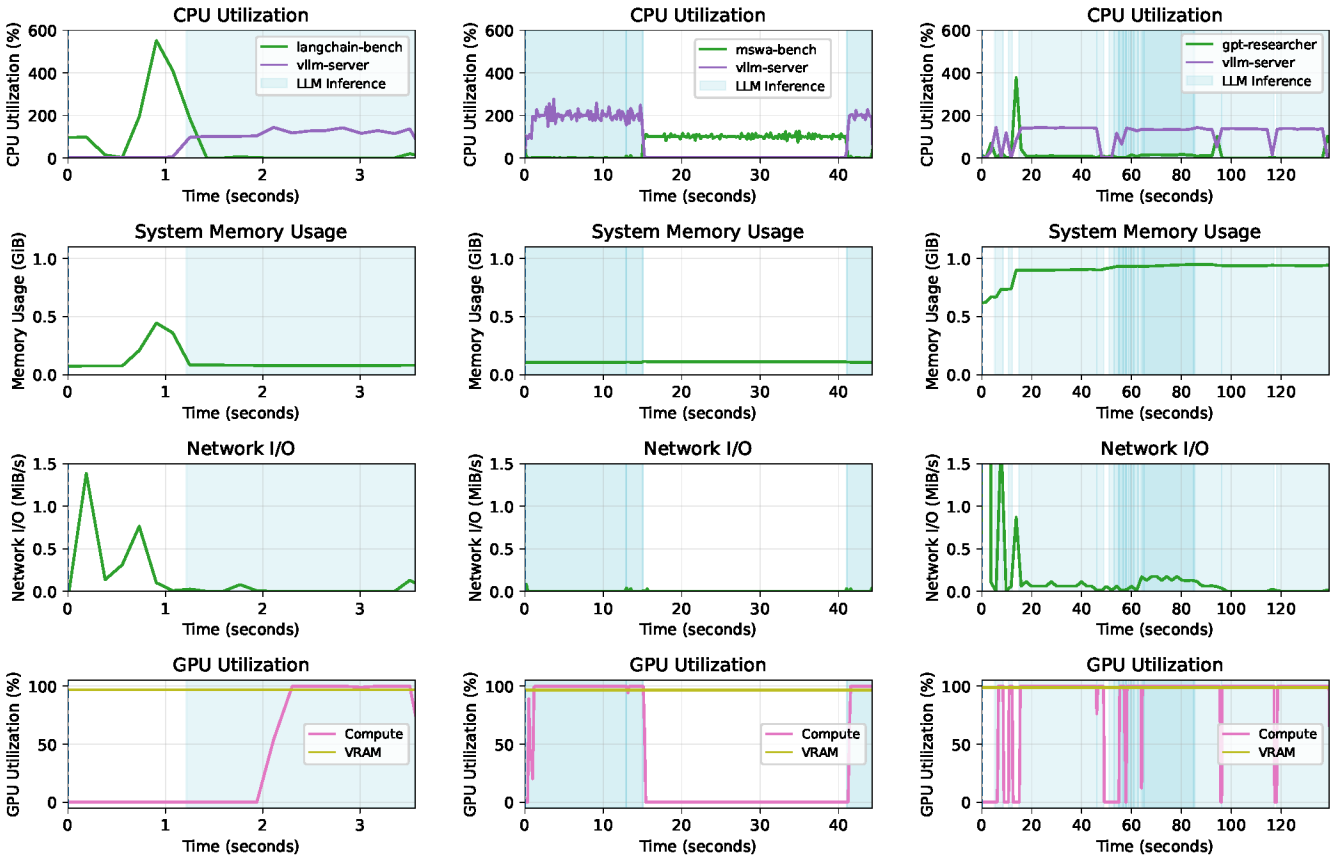
Figure 4: GPT Researcher workflow graph.

3 Characterizing Agentic Resource Footprints

To design effective systems to support agentic workloads, we first need to understand their workload characteristics A central challenge in characterizing agents is the diversity in their size and complexity. Static Retrieval Augmented Generation (RAG) workflows typically issue only a single inference invocation and use minimal CPU orchestration, whereas coding agents can exhibit long execution times due to iterative control flows and heavyweight tool calls. We present a case study over three representative agentic workloads, chosen to cover a breadth of workflow archetypes:

Static RAG Agent (Figure 2) A static question answering RAG workflow [46]. Based on the user query, the web search API returns a list of relevant pages. The agent then fetches the content of each page and uses LexRank [47] to extract key sentences. Finally, a single LLM inference invocation generates the answer to the user query.

Mini SWE agent (Figure 3) A linear iterative workflow for solving coding tasks. The execution loop takes a query and prompts an LLM to either generate a bash script or exit at each step. The agent executes the bash script and prompts the LLM again with the execution results. This is a simple example of a ReAct-style agent architecture, where agents



(a) Static RAG Agent. (b) Mini SWE Agent. (c) GPT Researcher.

Figure 5: Resource usage over agent execution. Shaded regions denote active LLM inference, with darker regions representing concurrent inference.

take successive reason, act, and observe steps to achieve a task [48].

GPT-Researcher (Figure 4) A multi-agent workflow for generating research reports. This is an example of both the scatter-gather architecture and multi-agent execution. The browser agent receives a user query and performs a preliminary web search to gather the background. The planning agent then processes the background to determine subtopics for multiple researcher agents. Each researcher agent gathers data through additional web search and summarization steps. Finally, a writing agent compiles each researcher's results into a final report.

We deploy each agent system on a single node equipped with an A100 GPU (80GB VRAM), two AMD 7343 16-core processors, and 1 TB DDR4 DRAM. The static RAG agent and GPT-Researcher use OpenAI's gpt-oss-20b. Mini SWE Agent uses Qwen3-Coder-30B, which is tailored for programming tasks. We capture CPU, memory, network, and GPU utilization over the course of execution for each workload, split between the vLLM inference engine and the agent.

Varying Execution Time. The execution time between each agent varies substantially. The static RAG pipeline (Figure 5a) executes in over 3 seconds, with 65% of time spent solely on the LLM inference. Mini SWE Agent (Figure 5b) spends 3 turns completing its task in over 44 seconds, roughly equally splitting execution time between inference and tool execution. GPT researcher (Figure 5c) fans out to 3 researcher agents and invokes 14 inference requests over a 128-second time period.

We note that, despite similar model sizes and architectures, LLM inference times vary with task complexity. Simple, scoped tasks (e.g., write a short summary, issue a single bash command) correlate with short inference requests (3-5 seconds). On the other hand, we see inference for reasoning-heavy tasks (report writing and code generation) takes substantially longer (10-30 seconds).

Different Resource Consumption. Just as execution times vary across agent workflows, so does the resource consumption. Memory usage peaks at ∼500MiB for the RAG agent, ∼100MiB for the Mini SWE Agent, and 1GiB for GPT-

Researcher. We attribute this to intermediate document stores built from web-scraped content. The CPU usage of the RAG agent and GPT-Researcher is relatively low, while the Mini SWE Agent exhibits a long tool-calling period that fully utilizes a single core.

Significant Idle Periods. The plots clearly show the phases of each workflow as agents switch between LLM and tool invocations. We observe that the different system components experience significant idle periods. While the overall agent spans many components, not all of them execute work simultaneously. For example, the allocated GPU resources (VRAM) are not freed, even though there is no interference running on the GPU. This suggests that loading and unloading components independently can reduce the agent's resource $\times$ time footprint. For example, long GPT-Researcher prompts cause LLM inference to last up to 30 seconds. Consequently, the agent logic could be unloaded during this time. Similarly, Mini SWE Agent experiences a 26-second tool call, where the LLM could instead be scaled down. We extend this idea into a practical serverless agent deployment scheme in §4.

Summary. Our analysis shows that agentic workloads exhibit distinct, predefined execution patterns during which components become idle. This raises the question: *How can we build an efficient, serverless system for agentic workloads?*

4 A Vision For Serverless Agents

Agents are emerging as an increasingly important driver of AI demand. Deploying agents in a serverless architecture can improve the resource efficiency of AI workloads. However, agents pose several challenges distinct from those of traditional serverless workloads. For one, agents can be long-running, with execution times on the order of minutes that amplify total resource consumption compared to high-turnaround tasks. Additionally, agents are stateful: context can persist over subsequent requests, and agent functionality may rely on stateful actions within tool environments.

We propose the following concepts to amend agents towards practical serverless deployment:

1. **Modular Deployment**: Agent workloads have a widely recognized substructure of LLM inference and tool invocations. Deconstructing agent workflow stages into isolated function instances gives the serverless orchestrator greater flexibility for instance scaling.
2. **Small Language Models**: Small fine-tuned models show promise for narrowly scoped tasks and have a much smaller resource footprint than LLMs. We show how small language models can benefit from serverless deployment through co-design with serverless agents.
3. **Workflow Graph Framework**: Deploying agents on serverless systems requires application-level changes. We introduce a programming model that minimizes porting effort from popular agent frameworks while enabling efficient serverless scaling across agent components.
4. **Tool Sandboxes**: To manage the high degree of session-level state between agent requests, we propose a solution for checkpointing and restoring stateful tool sandboxes within the serverless context.

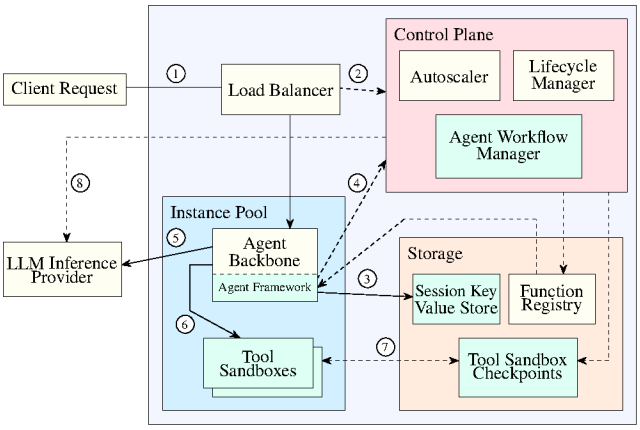


Figure 6: Modular Serverless Agent System Design. The light green components highlight additions to the existing serverless infrastructure. Solid arrows denote the client request path, and dashed arrows represent infrastructural interactions.

4.1 Use Modular Deployments

We can deploy an agent in a serverless system today, just as any other function, by bundling its orchestration logic, LLM invocation, and tooling in a single package. The serverless agent may make external API calls for subtasks that require heterogeneous compute or persistence: LLM inference through paid token APIs [13–15], custom models hosted on a serverless LLM platform [19, 49–51], vector databases, document stores, or any number of external resources. We refer to this single-package approach as the *monolithic deployment*. All serverless scheduling decisions, for instance, loading, keep-alive, and horizontal scaling must operate on the entire agent's container.

Alternatively, we can deconstruct serverless agent deployments into their individual workflow stages for greater resource efficiency. We represent an agentic workload as a control flow graph (CFG) [41, 42, 44], where each node represents an LLM invocation or tool execution. LLM invocations are already inherently decoupled from monolithic deployments via API calls to LLM-serving systems. Additionally, we decouple the agent orchestration logic from invoked tool sandboxes, resulting in a fully *modular deployment*.

Figure 6 shows the modular design of the resulting system. The system handles the request as follows: (1) Client request enters the serverless system and is routed by the load balancer.

(2) The load balancer queries the serverless control plane to inform instance startup and scaling. (3) Agent Framework retrieves session history from the storage layer. (4) As the agent executes, it updates the Agent Workflow Manager with the current stage progress. (5) An external LLM provider handles inference requests. (6) Stateful tool invocations are routed to independent tool sandboxes. (7) The Agent Workflow Manager policy decides the tool sandbox checkpoint and restore. (8) The Agent Workflow Manager policy decides when to load serverless LLM instances.

We discuss tool sandboxes and the handling of tool side effects in 4.4. The agent's orchestrator remains responsible for assembling prompts, LLM invocations, parsing LLM responses, and invoking tool calls. The modularized deployment enables execution with the minimal set of resources active at once, aligning agent deployments with the objectives of the serverless cloud model.

Separating agent workflow components enables us to leverage serverless instance scaling more efficiently. Consider a keep-alive policy that selectively keeps the agent orchestrator warm in DRAM: when a new request arrives, the orchestrator can execute immediately, while latter tool environments load in parallel. This approach will reduce the agentic deployment's resource footprint while achieving performance comparable to that of the monolithic deployment. A similar argument holds for cold starts: when a request arrives, only the orchestrator must be loaded before function execution can begin. Subsequent tool sandboxes can be loaded in parallel, mitigating cold-start latency compared to coarse-grained loading in monolithic agent deployments.

The primary challenge of a modular deployment is in which stages to load and when. If workflow stages are loaded only when accessed, the system maintains maximum resource efficiency but incurs the cost of additional internal cold starts. Conversely, if all workflow stages remain loaded throughout the agent's execution, the deployment offers no efficiency gain over the monolithic deployment. We propose adding an *agent workflow manager* to the serverless system's control plane to manage the lifetime of each component relative to the agent's execution progress. With the agent's CFG and a history of component execution times, the agent workflow manager can selectively unload and restore components as needed to balance application performance and resource efficiency.

4.2 Leverage Small Language Models

Despite the industry dominance of large generalist models, small custom models show strong performance on agentic workloads at a fraction of the resource requirements [16]. Fine-tuned small language models (SLMs) are an efficient alternative for narrowly scoped agentic subtasks that do not require the generality of LLMs. We argue that serverless infrastructure for agents should explicitly support custom SLMs as first-class citizens.

SLMs are a natural fit for serverless deployment; they are lightweight and application-specific. Specifically, the size of SLMs provides practical benefits for agent deployments: Many models can be pre-warmed into DRAM simultaneously, and deduplication of shared base model weights further leads to substantial memory savings [20]. Moreover, SLMs enable practical CPU-based inference, which improves model deployment density by harnessing underutilized CPU compute [21].

Co-designing serverless agents with SLM deployments can effectively mitigate cold-starts. When using GPUs to run SLMs, the fundamental cold-start overhead from the heavy data movement required to load model weights into GPU memory remains. However, the application context from agent workflow execution removes the guesswork from SLM scaling. The agent workflow manager treats SLM inference as another modular component of an agent's deployment, proactively loading and unloading the model as the agent executes. We further discuss how co-designing serverless agents and SLM deployment can improve inference performance through KV cache management (see §4.4).

4.3 Promote Workflows to First Class Citizens

Migrating an agent to serverless infrastructure requires application-level changes. This is generally the case for any workload, as developers must contend with the limitations of ephemeral function instances. However, we note that the event-driven nature of agentic workflows naturally suits the serverless model. We outline a programming model that complements our modular deployment strategy, enabling stage-wise loading and execution of agent workflows.

Similar to popular agent frameworks, we model each workflow stage as a function execution. We arrange the stages into a control flow graph (CFG) and register it with the serverless agent framework. Each stage takes in the output of the previous stage, executes some work, and returns an intermediate state for the succeeding stages, similar to Dandelion [52].

The framework controls stage execution by coordinating with the agent workflow manager on component loading decisions, for instance. This programming model is flexible and comfortably handles the breadth of agent diversity, from small static workflows to complex multi-agent architectures.

Agent workflow components can fall into one of the following categories: agent orchestration, LLM/SLM API calls, or tool invocations. When registering with the serverless agent framework, we annotate each component with one of these types, along with its state-management context (§4.4). The serverless agent framework uses these annotations to execute the deployment with the minimal required resource footprint. For example, we find that the agent orchestration logic experiences significant inactivity, e.g., while waiting for downstream execution results. This inactivity is especially prevalent in scatter-gather operations, where many LLM inference invoca-

tions happen in parallel and agent execution cannot continue until the last inference finishes. The serverless agent framework can instead scale down the orchestrator during these operations and pass the results into the next workflow stage when they're ready.

4.4 Persist Agent State

There is a critical design tension when deploying agents on serverless infrastructure: agents are stateful, while serverless functions are stateless. This mismatch is not a fundamental design blocker, but it does need to be addressed to prevent considerable inefficiencies in stateful function serving. From the agent's perspective, the state exists at the session level — exposing a turn-based interface to users for iteration on a target task. We can break down the agent's state into the following: orchestration state, used for the prompt generator, tool execution state, which captures the results of actions within the tool execution environment, and KV Cache state for inference.

Orchestration State. By isolating the tool execution environment from orchestration through modular deployment, we remove nearly all session-state overhead. The orchestrator still requires some session-level state for agent execution: specifically, the LLM prompt history, which captures context from prior user requests. A key-value store can easily back the agent's context based on a session ID. The history can be retrieved when a request arrives and appended to after each LLM inference. To reduce latency, we can embed local key value stores alongside function instances [53].

Tool Sandbox State. A tool sandbox provides agents with an environment to enact actions and observe their results. The persistence requirements are inherently diverse: Some tool invocations are entirely stateless, like the web search in a single-turn RAG agent. Whereas modern coding agents depend heavily on stateful environments, requiring persistent filesystem modifications and shell sessions between requests.

To support these capabilities in a serverless deployment, we require a mechanism that can snapshot a running instance and restore it on demand. CRIU (Checkpoint/Restore In Userspace) [54] provides precisely this functionality through freezing a container's execution and checkpointing its state to disk. This technique is used in AWS Firecracker [30] to checkpoint micro VM state immediately after initialization, minimizing cold-start delays. We can use the same concept to pause and resume an agent's tool environment in accordance with the workflow manager's policy.

Freezing and checkpointing a container are heavyweight operations. Ideally, we want to incur full checkpointing costs only when necessary for the application. We propose annotating tool executions with their required level of statefulness, ranging from stateless to file-system only to file-system and active processes.

KV Cache Management. Despite being hidden behind the LLM inference API, KV cache state remains essential to agent performance. The KV cache is an implicit piece of session state that stores information about processed tokens. Agent prompts share a large amount of context across successive inferences, and prompt caching reduces redundant work by precomputing the prefill steps for shared tokens. Some closed-weight LLM APIs provide limited control over prompt caching [55, 56], enabling short-term storage, but with limited flexibility.

In the context of serverless SLM deployments, KV cache management can be aided by workflow progress. If an SLM instance is scaled down during agent execution, we want to persist the KV cache for future inference invocations. We can further improve KV cache management through precomputing the static portion of system prompts at compile time and maintaining a global view over KV caches [42]. Since KV caches can grow to several gigabytes, the tradeoff between storing to disk and recomputing remains nuanced.

5 Conclusion

We survey the state of the art for serverless hosting of generative AI inference workloads. Our study shows that agentic workloads have an inherent structure that parallels that of serverless deployments. Agents should be treated as workflows whose stages can be scheduled, scaled, and optimized independently. This requires the serverless architecture and the scheduler to become agent-aware. We present promising research directions for agent deployment to reduce resource consumption, efficiently serve small language models, and manage agent state.

References

[1] Flexera. 2026 state of the cloud report. Technical report, Flexera, 2026. Accessed: April 25, 2026.

[2] Cloudera. 96% of enterprises are expanding use of AI agents. Cloudera Press Release, 2025.

[3] Mike Vizard. Survey surfaces rapid adoption of agentic AI. Techstrong.ai, 2025.

[4] Grand View Research. U.S. data center market size, share & trends analysis report by component, by type, by server rack density, by redundancy, by PUE, by design, by tier level, by enterprise size, by end-use, and segment forecasts, 2023–2030. Technical Report GVR-4-68040-170-7, Grand View Research, 2023. Accessed: 2026-04-25.

[5] Kif Leswing. Ai memory is sold out, causing an unprecedented surge in prices. CNBC, January 2026. Accessed: 2026-04-25.

[6] Francisco Jeronimo, Tom Mainelli, Bryan Ma, Ryan Reith, and Jeff Janukowicz. Global memory shortage crisis: Market analysis and the potential impact on the smartphone and PC markets in 2026. IDC Blog, December 2025. Accessed: 2026-04-25.

[7] Tim Tully, Joff Redfern, Deedy Das, and Derek Xiao. 2025 Mid-Year LLM market update: Foundation model landscape and economics. Industry report, Menlo Ventures, Menlo Park, CA, July 2025. Survey of 150 technical decision-makers, June 30–July 10, 2025.

[8] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in AWS lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 315–328, Boston, MA, July 2023. USENIX Association.

[9] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[10] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. Memory deduplication for serverless computing with medes. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 714–729, New York, NY, USA, 2022. Association for Computing Machinery.

[12] Wei Qiu, Marcin Copik, Yun Wang, Alexandru Calotoiu, and Torsten Hoefler. User-guided page merging for memory deduplication in serverless systems, 2023.

[13] OpenAI. ChatGPT. https://chatgpt.com, 2025. Accessed: 2026-04-25.

[14] Google DeepMind. Gemini. https://gemini.google.com, 2025. Accessed: 2026-04-25.

[15] Anthropic. Claude. https://claude.ai, 2025. Accessed: 2026-04-25.

[16] Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small language models are the future of agentic ai, 2025.

[17] Francisco Ríos. Hugging face's two million models and counting. AI World, https://aiworld.eu/story/hugging-faces-two-million-models-and-counting, December 2025. Accessed: 2026-04-25.

[18] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. Serverlessllm: Low-latency serverless inference for large language models, 2024.

[19] Junhao Hu, Jiang Xu, Zhixia Liu, Yulong He, Yuetao Chen, Hao Xu, Jiang Liu, Jie Meng, Baoquan Zhang, Shining Wan, Gengyuan Dan, Zhiyu Dong, Zhihao Ren, Changhong Liu, Tao Xie, Dayun Lin, Qin Zhang, Yue Yu, Hao Feng, Xusheng Chen, and Yizhou Shan. Deepserve: Serverless large language model serving at scale, 2025.

[20] Yifan Sui, Hao Wang, Hanfei Yu, Yitao Hu, Jianxun Li, and Hao Wang. Serverlesslora: Minimizing latency and cost in serverless inference for lora-based llms, 2025.

[21] Chuhao Xu, Zijun Li, Quan Chen, Han Zhao, Xueyan Tang, and Minyi Guo. Towards resource-efficient serverless llm inference with slinfer, 2025.

[22] Chiheng Lou, Sheng Qi, Chao Jin, Dapeng Nie, Haoran Yang, Yu Ding, Xuanzhe Liu, and Xin Jin. Hydraserve: Minimizing cold start latency for serverless llm serving in public clouds, 2025.

[23] Tianyu Wang, Gourav Rattihalli, Aditya Dhakal, Xulong Tang, and Dejan Milojicic. Wages: Workload-aware gpu sharing system for energy-efficient serverless llm serving. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC Workshops '25, page 508–515, New York, NY, USA, 2025. Association for Computing Machinery.

[24] Yifan Sui, Hanfei Yu, Yitao Hu, Jianxun Li, and Hao Wang. Pre-warming is not enough: Accelerating serverless inference with opportunistic pre-loading. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 178–195, New York, NY, USA, 2024. Association for Computing Machinery.

[25] Chongpeng Liu, Xiaojian Liao, Hancheng Liu, Limin Xiao, and Jianxin Li. Pipeboost: Resilient pipelined architecture for fast serverless llm scaling, 2025.

[26] Samuel Kounev, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup, Ian Foster, Prashant Shenoy, Omer Rana, and Andrew A. Chien. Serverless computing: What it is, and what it is not? *Communications of the ACM*, 66(9):80–92, September 2023.

[27] Google. Google cloud functions. https://cloud.google.com/functions/. Accessed: 2026-04-25.

[28] Microsoft. Azure functions serverless architecture. https://azure.microsoft.com/en-us/services/functions/. Accessed: 2026-04-25.

[29] Apache Software Foundation. Apache OpenWhisk is a serverless, open source cloud platform. http://openwhisk.apache.org/. Accessed: 2026-04-25.

[30] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[31] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84, Carlsbad, CA, July 2022. USENIX Association.

[32] Nicholas C. Wanninger, Joshua J. Bowden, Kirtankumar Shetty, Ayush Garg, and Kyle C. Hale. Isolating functions at the hardware limit with virtines. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 644–662, New York, NY, USA, 2022. Association for Computing Machinery.

[33] Hugging Face. Models, 2026. Accessed, May 21, 2026.

[34] 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, 2023.

[35] Shaoxun Zeng, Minhui Xie, Shiwei Gao, Youmin Chen, and Youyou Lu. Medusa: Accelerating serverless llm inference with materialization. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 653–668, New York, NY, USA, 2025. Association for Computing Machinery.

[36] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. Lmsys-chat-1m: A large-scale real-world llm conversation dataset, 2024.

[37] Yuxing Xiang, Xue Li, Kun Qian, Wenyuan Yu, Ennan Zhai, and Xin Jin. Servegen: Workload characterization and generation of large language model serving in production, 2025.

[38] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Burstgpt: A real-world workload dataset to optimize llm serving systems, 2025.

[39] Minchen Yu, Rui Yang, Chaobo Jia, Zhaoyuan Su, Sheng Yao, Tingfeng Lan, Yuchen Yang, Zirui Wang, Yue Cheng, Wei Wang, Ao Wang, and Ruichuan Chen. λscale: Enabling fast scaling for serverless large language model inference, 2026.

[40] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, et al. The llama 3 herd of models, 2024.

[41] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of llm-based applications with semantic variable, 2024.

[42] Noppanat Wadlom, Junyi Shen, and Yao Lu. Efficient llm serving for agentic workflows: A data systems perspective, 2026.

[43] Nikos Pagonas, Yeounoh Chung, Kostis Kaffes, and Arvind Krishnamurthy. Cortex: Workflow-aware resource pooling and scheduling for agentic serving, 2025.

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

[45] You Peng, Youhe Jiang, Wenqi Jiang, Chen Wang, and Binhang Yuan. Hexgen-flow: Optimizing llm inference request scheduling for agentic text-to-sql, 2026.

[46] Ritik Raj, Souvik Kundu, Ishita Vohra, Hong Wang, and Tushar Krishna. Towards understanding, analyzing, and optimizing agentic ai execution: A cpu-centric perspective, 2026.

[47] Günes Erkan and Dragomir R. Radev. Lexrank: Graph-based lexical centrality as salience in text summarization. *CoRR*, abs/1109.2128, 2011.

[48] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.

[49] Amazon Web Services. Amazon bedrock. https://aws.amazon.com/bedrock/, September 2023. Generally available September 28, 2023. Accessed: 2026-04-25.

[50] Oracle. OCI generative AI. https://www.oracle.com/artificial-intelligence/generative-ai/generative-ai-service/, January 2024. Generally available January 23, 2024. Accessed: 2026-04-25.

[51] Anyscale. Anyscale: A unified AI compute platform. https://www.anyscale.com, 2019. Founded 2019. Accessed: 2026-04-25.

[52] Tom Kuchler, Pinghe Li, Yazhuo Zhang, Lazar Cvetković, Boris Goranov, Tobias Stocker, Leon Thomm, Simone Kalbermatter, Tim Notter, Andrea Lattuada, and Ana Klimovic. Unlocking true elasticity for the cloud-native era with dandelion, 2025.

[53] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, July 2020.

[54] Checkpoint-Restore Project. CRIU: Checkpoint/restore tool. https://github.com/checkpoint-restore/criu. Accessed: 2026-05-19.

[55] OpenAI. Prompt caching. https://developers.openai.com/api/docs/guides/prompt-caching. OpenAI API Documentation. Accessed: 2026-05-15.

[56] Anthropic. Prompt caching. https://platform.claude.com/docs/en/build-with-claude/prompt-caching. Claude API Docs. Accessed: 2026-05-15.