Is Serverless a Misfit for Agentic AI?

Bita Asoodeh
University of Edinburgh
Edinburgh, United Kingdom
bita.asoodeh@ed.ac.uk

Amir Noohi
University of Edinburgh
Edinburgh, United Kingdom
amir.noohi@ed.ac.uk

Antonio Barbalace
University of Edinburgh
Edinburgh, United Kingdom
antonio.barbalace@ed.ac.uk

Abstract

AI agents run model-written commands in the sandboxes that were built for serverless functions. The 60 agent systems that we survey use the same microVMs and containers as 41 serverless systems. It is disputed whether an agent's tool calls should also run as functions. One proposal treats agentic workflows as serverless applications, while agent platforms keep one sandbox per session. We measure it. We state the Function-as-a-Service (FaaS) contract as five properties and test each one on live and published sessions of frontier models on SWE-bench Verified and Terminal-Bench, with 14 functions in the same Firecracker microVM as a reference. The sandbox is not the obstacle, because it boots in about the same time and most tool calls are as short as function calls. The contract is the obstacle, because of state. With a new sandbox for every command, an agent solves 5 of 12 SWE-bench tasks instead of all 12. Keeping the working directory repairs code edits, but the agent loses 12 of the 51 Terminal-Bench tasks that it solves with one sandbox per session. Keeping the whole disk still loses 8 tasks, and 6 of them leave a server running. A session's sandbox, however, is idle for 81–91% of the session while the model replies, and freeing it means saving its memory, which takes 2.07 s. These results explain why agent platforms manage sessions and not calls. Sessions leave open how to size memory per command and how to reclaim idle time without losing state. We propose three mechanisms and measure their costs.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Virtual machines**; *Operating systems*; • **Computing methodologies** → *Intelligent agents*; • **General and reference** → Measurement.

Keywords: serverless computing, Function-as-a-Service, agentic AI, LLM agents, sandboxing, microVMs, agent sessions, workload characterization

ACM Reference Format:
Bita Asoodeh, Amir Noohi, and Antonio Barbalace. 2026. Is Serverless a Misfit for Agentic AI?. In *Workshop on AI and Serverless (WoAIS '26), December 14–18, 2026, Tarragona, Spain.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3861043.3861050



This work is licensed under a Creative Commons Attribution 4.0 International License.

WoAIS '26, Tarragona, Spain
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3108-2/2026/12
https://doi.org/10.1145/3861043.3861050

1 Introduction

AI agents act through tools [25]. In a coding agent such as Claude Code, Codex or OpenHands, a large language model (LLM) writes shell commands. The agent's *harness*, the program that drives the loop around the model, runs each command in a sandbox and returns its output to the model for the next decision. Each such command is a *tool call*. An agent thus has three parts, the model, the harness and the sandbox of its tool calls. The model, not a developer, writes each command, so the sandbox must isolate it. Cloud providers therefore reuse the sandboxes that they built for serverless functions, known as Function-as-a-Service (FaaS). The Firecracker microVM [1] that isolates AWS Lambda functions also isolates the agent sessions of Amazon Bedrock AgentCore [5], AWS Lambda MicroVMs [4] and E2B [9].

Most of the 60 agent systems that we survey run on the same microVMs, user-space kernels and containers as 41 serverless systems (Section 3). If agents run in serverless sandboxes, should their tool calls also run as FaaS functions? Dougherty et al. argue that agentic workflows *are* serverless applications [8]. In their view, the agent logic, the tools and the model work in turn, so an agent-aware serverless platform that loads each stage independently would use resources more efficiently. They call the mismatch between stateful agents and stateless functions "not a fundamental design blocker" and leave the design unevaluated. Practice suggests that the mismatch matters. AgentX runs agent tools as AWS Lambda functions. It replaces its file-system server with object storage, because Lambda functions "lack persistent local storage", and it installs every package in advance, because a function cannot install one at run time [23]. Recent agent platforms go further and keep one sandbox for a whole session [5, 9]. No work has yet measured which one is right.

We provide this measurement for the sandbox of the tool calls. We leave the other two parts aside, because the model is a shared service and the harness can run as a function (Section 2.3). FaaS is more than a sandbox. The provider runs code under a contract, which we state as five properties. **(P1)** A call is short and independent. It runs for a bounded time, at most 15 minutes on Lambda [3], and relies on no other call. **(P2)** No state is guaranteed to survive a call, so the application stores what it needs later in external storage [10, 12]. **(P3)** Every call of a function gets the same memory and

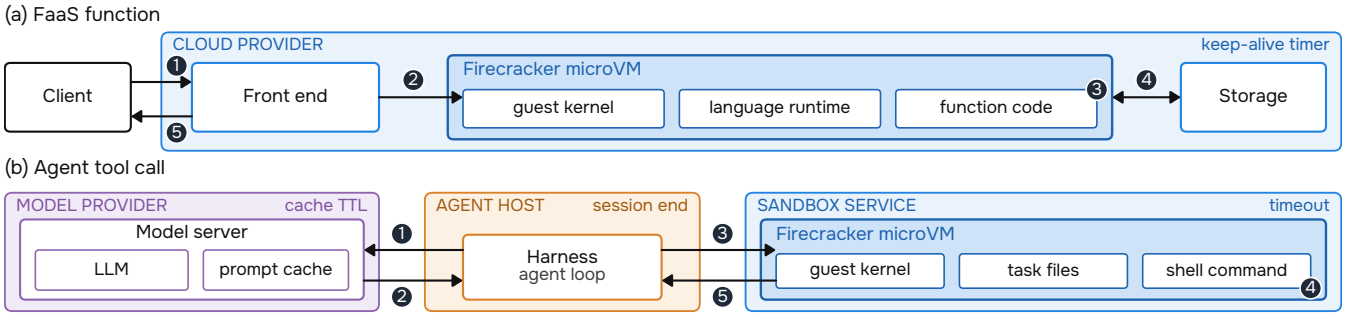

Figure 1. Serving a FaaS function (a) and an agent tool call (b).

CPU, which are set when the function is deployed. **(P4)** A call that finds no warm sandbox pays a cold start, in which the provider boots a sandbox and initializes the function [12]. **(P5)** The provider controls the sandbox's lifecycle. It sees every call, so it alone decides how long an idle sandbox stays warm and when to discard it [21]. This contract makes FaaS cheap to operate, because the provider can discard any idle sandbox and pack sandboxes by their declared sizes.

We test the execution model and take the sandbox's isolation as given. A frontier model, Claude Opus 5.5, drives a coding agent live on SWE-bench Verified and Terminal-Bench tasks [11, 15, 20] in 527 sessions, and we impose the contract on it one property at a time. To test state, we run every command in a new sandbox that keeps nothing, the working directory, or the whole disk, and each benchmark's own tests tell whether the agent still solves its task. To test memory, we replay the agent's commands under one fixed memory size. We also replay published sessions of three other frontier models to check that the effects do not depend on one model. As a reference, 14 benchmark functions run in the same Firecracker microVM.

Our results show that the sandbox is not the obstacle but the contract is. Four of the five properties break, and state (P2) matters most. **P1 breaks.** A tool call is short but relies on the calls before it, and 9 of 117 Terminal-Bench sessions outlast Lambda's 15 minutes. **P2 breaks.** With nothing kept between commands, the agent solves 5 of 12 SWE-bench tasks instead of all 12. With the working directory kept, it still loses 12 of the 51 Terminal-Bench tasks that it solves with one sandbox per session, and with the whole disk kept it loses 8. An agent's state is its whole sandbox, with its running processes, which a durable function does not keep. **P3 breaks.** A session's largest command needs 5–12× the memory of its median one. **P4 holds.** A session boots its sandbox once, in 181 ms. **P5 breaks.** The sandbox is idle for 81–91% of a session while the harness waits for the model. FaaS would discard it, but it holds the session's state, and saving its memory takes 2.07 s. Up to three parties run a session, and only the harness knows whether it goes on.

The FaaS contract is therefore a misfit for the tool calls of agents that act through a shell, although the FaaS sandbox is not. This supports the choice of recent agent platforms, which keep one sandbox per *session* rather than per call. This paper makes three contributions. First, we survey 60 agent systems and 41 serverless systems, which share their isolation technologies but not their lifecycles (Section 3). Second, we present, to our knowledge, the first measurement of agent tool calls and FaaS functions in the same sandbox, and of the tasks that a live agent no longer solves under the FaaS properties (Section 4). Third, we propose three mechanisms that manage the session instead of the call, and we measure what each costs (Section 5). Our code and data are available at https://github.com/bitaasudeh/misfit.

2 Background

2.1 Function-as-a-Service (FaaS)

FaaS runs stateless functions on demand. The path of a call shows where each property applies (Figure 1a). A client sends a request ❶, which is one short and independent call (P1). The provider's front end places the request into a sandbox ❷, such as a microVM. The sandbox runs the function ❸, either warm or after a cold start (P4). The function keeps its state in external storage ❹ (P2), and the client gets the reply ❺. The provider bills the configured memory and CPU for the call's duration (P3). After the call, the provider keeps the sandbox warm for a fixed or a learned time, which is the keep-alive timer, and then discards it (P5) [21].

2.2 Agentic AI

This paper focuses on coding agents that act through a shell (Figure 1b). That is, each tool call is one shell command, which may run any program and change any file in its sandbox. These agents therefore need a full OS sandbox. In mini-SWE-agent [22], the harness sends the conversation to the model server ❶, which returns the next command ❷. The harness sends the command to a sandbox ❸, which runs it against the task's files ❹. The harness then appends the command's output to the conversation ❺. One run of this loop on one task is a *session*. Each pass of the loop is a *step*, and it runs one tool call. The model runs on a separate server, which another provider runs for hosted APIs. The sandbox can belong to a third party, a *sandbox service*.

		serverless		agents	
isolation	boundary	products	research	products	research
Firecracker microVM	hardware VM	Fly.io, Lambda, Lambda container images, Lambda SnapStart, Lambda tenant mode	4	AgentCore, E2B, Lambda MicroVMs, Vercel Sandbox	3
Kata, other light VM	hardware VM	Alibaba Function Compute, Cloud Run gen2	1	Cloudflare Sandbox, Docker Sandboxes, Northflank	1
Hyper-V, other full VM	hardware VM	Azure Functions	–	Azure dynamic sessions, Code Interpreter, Codex Cloud, Copilot agent, Devin, Jules, Manus, Runloop	1
gVisor	user-space kernel	Cloud Run gen1	1	GKE Agent Sandbox, K8s agent-sandbox, Modal	–
container	namespaces, cgroups	Fission, Knative, Lambda Managed Instances, OpenFaaS, OpenWhisk	14	Claude code execution, Claude Managed Agents, Codex Cloud (legacy), Daytona, Gemini CLI	22
OS sandbox	kernel policy	–	–	Claude Code, Codex CLI, Copilot CLI, Cursor	–
process	none	–	–	–	2
V8 isolate, Wasm	runtime	Cloudflare Workers, Fastly Compute, Spin	2	–	1
unikernel, other	varies	–	2	–	3

Table 1. Isolation reported by agent and serverless systems. Bold technologies appear on both sides (sources in our repository).

2.3 FaaS and Agentic AI

An agent has three parts, and each could run as a function. The model already runs as a shared service. The harness computes for only 2.2 ms per step in our SWE-bench sessions (Figure 2a) and keeps only the conversation. It can therefore run as a durable function, which checkpoints each step's result and is not billed while it waits [2]. The open part is the sandbox of the tool calls, which can use FaaS in three ways. First, the harness and its commands can run in one function, which stops at the provider's time limit [3]. Second, each tool call can run as its own function, as AgentX runs its tools [23]. Its outputs and changed files must then pass through external storage. Third, each tool call can be one step of a durable function, which guarantees the data that the step returns but not the files and processes that it leaves. Agent platforms instead keep one sandbox per session [5, 9]. Section 4 tests one function per tool call against this.

3 One Sandbox, Two Lifecycles

Before we measure, we check what FaaS and agent platforms run on today. Table 1 lists the isolation that 60 agent systems and 41 serverless systems report in their papers and documentation. The technologies are the same on both sides, in products and in research. This is no surprise, since virtual machines serve many workloads, including network functions [13]. Only the extremes differ. In-process isolation, such as V8 isolates, serves almost only functions. The operating system's own sandboxing on a developer's machine, such as Seatbelt, serves only agents.

What differs is the lifecycle around the sandbox. A Lambda function runs for at most 15 minutes with one memory size between 128 MB and 10 GB [3]. Agent platforms add the session that FaaS lacks. AgentCore by default stops a session after 15 idle minutes or after 8 hours [5]. Lambda MicroVMs suspend a session with its memory and disk [4], and E2B pauses and resumes it [9]. Practice has thus left the FaaS contract and kept only its sandbox. Section 4 measures why.

4 Testing the Contract

We describe our setup and then test the five properties.

4.1 Methodology

Host. One host with two AMD EPYC 9255 processors, 512 GB of DRAM and Linux 6.8 runs every workload. Each workload runs in a Firecracker v1.16.1 microVM with two vCPUs, 4.3 GB of memory, a Linux 6.1.141 guest kernel and a disk built from the workload's Docker image. Both workloads thus share its virtualization overhead, such as nested address translation [19]. A guest daemon runs each command in a fresh control group (cgroup) and reports its run time and its memory peak, which counts the page cache. The harness runs on the host, outside the microVM (Figure 1b).

Agent workload. *Live sessions* run mini-SWE-agent 2.4.6 [22] with Claude Opus 5.5 through its hosted API on two benchmarks. Each benchmark's own tests decide whether a session solved its task. SWE-bench Verified [20] asks for a patch to a repository. We use 12 tasks from 10 repositories in two environments, with the task's Python environment active or, as in the published traces, not active. Terminal-Bench 2.0 [15] asks for work beyond editing code, such as building software, configuring servers and training models. We use the 80 of its 89 tasks that fit our microVM. Each task has the benchmark's time limit of 10 to 60 minutes, and we stop a command after 15 minutes. Under each policy below, a task runs up to three times per environment on SWE-bench and twice on Terminal-Bench. We set aside the sessions that end for reasons outside the sandbox, nearly all because the model provider's filter against misuse cut a reply. This leaves 527 sessions. *Replayed sessions* re-execute commands in order and without a model. We replay the live sessions and the published traces of Claude Sonnet 4.5, GPT-5 mini and GLM-5 on the same SWE-bench tasks (36 sessions, 1,794 commands).

Policies. A policy decides what state a command inherits from the commands before it. The baseline policy, **session**, runs all commands of a session in one microVM, as agent platforms do [5, 9]. The other policies start a new microVM for every command, as if each tool call were its own FaaS function (P2), and they differ in what they keep. **Nothing** keeps no state. **Directory** keeps the task's working directory and `/tmp`, as serverless agent systems keep files in external

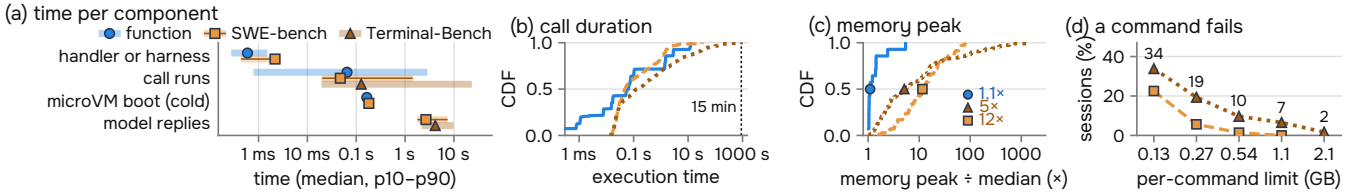

Figure 2. Time and memory of functions and of the tool calls of live agent sessions. Bars in (a) span the 10th to 90th percentiles.

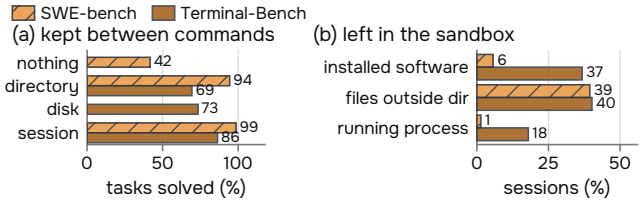

Figure 3. Tasks solved under each policy (a) and what sessions leave in their sandbox (b).

storage [23]. On SWE-bench, it leaves out the files that `git` ignores. **Disk** keeps the whole disk, as a function with a persistent volume would. The agent runs live under each policy and is not told which one. We run the nothing policy only on SWE-bench with the environment active, and the disk policy only on Terminal-Bench. We also replay sessions under a policy. A replayed command is *broken* when its exit code differs from the live run's, or when fewer than 90% of its output characters match the live output after we mask digits. We count only commands that a replay under the session policy reproduces. For P3, the **memory limit** policy replays a session in one microVM with every command under one limit between 134 and 2,147 MB, Lambda's sizes of 128 to 2,048 MiB. A command fails when the out-of-memory (OOM) killer kills one of its processes, or when it stalls without the memory that it used live.

Reference workload. Functions only give the agent numbers a reference from the same microVM. We run 14 Python functions from FunctionBench and vSwarm [14, 24], unchanged except that storage calls go to local files and each sits behind an HTTP handler. Each function boots cold 5 times, serves 100 warm calls and resumes once from a snapshot.

4.2 A tool call is one step of a dependent loop (P1)

P1 asks for calls that are short and independent. Tool calls are short. The median command runs for 47 ms on SWE-bench and 127 ms on Terminal-Bench, against 65 ms for a warm function (Figure 2a and b). However, tool calls are not independent. A tool call is one of 6 to 7 steps at the median, and each step builds on what the earlier steps left in the sandbox. Only the whole session relies on no earlier call, and a session is not short. SWE-bench sessions end within 3.7 minutes. In contrast, 9 of the 117 Terminal-Bench sessions, and even 2 single commands, outlast the 15 minutes that Lambda allows a function. The longest lasts 40 minutes.

Insight 1 (P1). *Each step is short but relies on the steps before it. Only the whole session relies on no earlier call, and it can outlast a function's time limit.*

4.3 Agent state is the whole sandbox (P2)

P2 lets the provider run each call in a new sandbox. For agents, this is the property that matters most (Figure 3a). Under the nothing policy, the agent solves 5 of 12 SWE-bench tasks. It must redo its edits inside every command, and 7 sessions use up the 80 steps that we allow. With one sandbox per session, it solves the task in all 35 sessions of the same environment. Replays show the same for single commands. With a new microVM for every command, 37% of the commands of the 71 SWE-bench sessions with one sandbox per session break, as do 19–42% for each other model.

Keeping the directory is enough when the work stays in it. The agent then solves 68 of 72 SWE-bench sessions, against 70 of 71 with one sandbox per session. Keeping the directory is not enough once the agent changes the system. Of the published sessions, 29 of 36 install a package. After the install, 35% of their commands break although the directory is kept. On Terminal-Bench, 37% of the sessions install software (Figure 3b). We could grade 59 tasks under all three policies, and under the session policy the agent solves 51 of them in every session. With only the directory kept, it never solves 12 of these 51 tasks. Where the agent still succeeds, it installs its tools again in the commands that need them, with 4× the install commands per session. Telling the agent what its sandbox keeps removes most of the repeated installs. However, 10 of the 11 lost tasks that this run covers stay lost.

Keeping the whole disk is not enough either. The agent never solves 8 of the 51 tasks. In 6 of them, the session ends with a server running in its sandbox, such as a web server, a mail system or an emulator. The task's tests then contact this server, but with a new microVM for every command the server is gone. A session's sandbox thus accumulates state of every kind. Of the Terminal-Bench sessions, 40% change files outside their directory, up to 38,293 files, and 18% end with a process still running (Figure 3b).

Insight 2 (P2). *Agent state is the whole sandbox, with its disk and its running processes. A session must therefore keep its sandbox between commands.*

4.4 One memory size fits no session (P3)

P3 gives every call of a function one memory size. This fits a function, which runs one program. Its handler peaks at only

1.1× its median memory (Figure 2c). A session's commands are different programs, and each runs in its own cgroup. A session's largest command peaks at 12× the memory of its median command on SWE-bench and at 5× on Terminal-Bench, at the median over sessions. The published sessions give 46×. The largest commands use up to 3.3 GB. With the smallest limit of 134 MB for every command, a command fails in 16 of the 71 SWE-bench sessions and in 21 of 62 replayed Terminal-Bench sessions (Figure 2d). A limit of 1,074 MB fails no SWE-bench session, but it still fails 4 of 61 Terminal-Bench sessions. A size that fits is mostly unused. The median SWE-bench guest holds 162 MB between commands.

Insight 3 (P3). *A session's largest command needs 5–12× the memory of its median command, and sometimes gigabytes. A fixed size therefore fails commands or wastes memory.*

4.5 One cold start per session (P4)

P4 charges a cold start to a call that finds no warm sandbox. The same microVM boots in 164 ms for a function and in 181 ms for an agent (Figure 2a). Inside a session, the keep-alive timer (Section 2.1) never needs to expire, because 99.5% of the model's replies arrive within a minute.

Insight 4 (P4). *With a few minutes of keep-alive, only a session's first command starts cold, so P4 holds.*

4.6 An idle sandbox and no single owner (P5)

P5 lets the provider alone decide how long an idle sandbox stays warm and when to discard it. An agent's sandbox is idle most of the time, because the harness waits for the model's next command. At the median, one reply takes 2.7 s on SWE-bench and 4.1 s on Terminal-Bench (Figure 2a). These waits add up to 91% of a SWE-bench session and 81% of a Terminal-Bench session at the median. In 29 of the 117 Terminal-Bench sessions, however, the sandbox computes for more than half of the time. An idle agent sandbox holds the session (Section 4.3), so it cannot be discarded. Freeing it means saving its memory, which took 2.07 s at the median for our 4.3 GB microVM. A sandbox that is left running holds its memory through every wait. After a SWE-bench session ends, its microVM still holds 416 MB, its peak (median).

There is also no single owner. Up to three parties run a session, and each has its own timer (Figure 1b). The model provider caches the conversation and drops it after a time-to-live (TTL) of 5 minutes by default on Anthropic's API [6]. The harness ends the session. The sandbox service stops a sandbox after a timeout [5, 9]. The harness is kept separate because it holds the user's credentials, while the sandbox runs untrusted commands.

Insight 5 (P5). *The sandbox idles while the model thinks, but freeing it means saving its memory. Only the harness knows whether the model is thinking or the user has left, and it owns neither the cache nor the sandbox.*

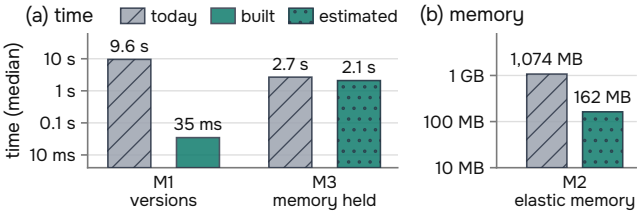

Figure 4. Measured cost today and with each mechanism.

5 Proposed Solution

Agent platforms, like our session policy, give each session one sandbox. This lifts P1 and P2, because a session may run long and its state stays. It gives up what P3 and P5 gave the provider. The sandbox must fit its session's largest command, and it cannot be discarded while the model thinks. No mechanism below is new. We add the measured cost of each gap and the missing signal, the session's phase.

Why today's fixes fall short. Agent platforms can suspend a sandbox [4, 9]. However, a sandbox service that is separate from the harness cannot tell a thinking model from a user who has left, so it must use a timeout. A short one pays 2.1 s to suspend during a reply of 2.7 s, and a long one holds 416 MB through every wait.

The session as the unit. We propose that the sandbox service manage an *agent session* as an OS manages a process. A session is one object that bundles the sandbox and its place in the harness's loop. The harness reports the session's phase to the sandbox service, as agents do to the workflow manager that Dougherty et al. sketch [8]. The service then schedules, sizes, versions and tears down the session, and not the call, through three mechanisms. They rest on primitives of the host kernel and the virtual machine monitor (VMM), like those that our earlier work builds for the memory of model training and serving [16–18]. We built M1 and measured the costs that motivate M2 and M3 (Figure 4).

M1, versioned state (P2). A session's sandbox keeps the state only while it runs. M1 takes a copy-on-write snapshot of the session's memory and disk after each command. The whole environment, with its running processes (Section 4.3), then outlives the sandbox. The session can also return to an earlier command to retry a step or to try another path, as DeltaBox does [7]. We built M1 with Firecracker diff snapshots and a reflink copy of the disk, and we replayed the 36 published sessions with it. At the median, a version took 98 ms after each command and wrote 5.9 MB. Returning to the middle command of a session took 35 ms from a full snapshot, against 9.6 s to re-run its commands in a fresh microVM (Figure 4a). The next command reproduced in all 36 sessions.

M2, elastic memory (P3). M2 sizes the sandbox's memory per command. Between commands, the sandbox keeps a small base size. It grows for a command that needs more, through memory ballooning, hot-plug or a cgroup limit, and it shrinks afterwards. On SWE-bench, a fixed size that fails no command is 1,074 MB, which is 6.6× the memory that

the guest holds between commands. On Terminal-Bench, 4 sessions fail even with this size (Section 4.4, Figure 4b).

M3, suspension while the model thinks (P5). M3 replaces the sandbox's timeout with a decision that knows the session's phase. The phase says whether the session waits for the model, runs a command or waits for the user. M3 snapshots the sandbox once a wait has lasted an idle period that the sandbox service sets, and it restores the sandbox when the next command arrives. A full snapshot and resume of our 4.3 GB microVM takes 2.1 s and grows with the guest's memory. This is about as long as a median reply of 2.7 s (Section 4.6, Figure 4a). Suspension therefore pays in waits that the phase marks as long, such as a wait for the user.

6 Limitations

Our results hold for agents that act through a shell. They do not extend to agents that only call web services, because those change no files and start no processes. We did not test the isolation of the sandbox, and we did not run the harness as a function. The live sessions use one model at one provider, whose network adds to their waits. The published sessions of three other models support only the command-level results. The provider's filter against misuse (Section 4.1) ended 46 of our 166 Terminal-Bench sessions with one sandbox per session. It left 16 tasks ungraded, including all cryptanalysis, DNA and protein tasks. The tasks that we compare are therefore not a random sample, and some of them rest on one session per policy. Finally, a replay shows where a command first breaks, but not what the agent would do next.

7 Conclusion

Is serverless a misfit for agentic AI? Its sandbox is not, because agents boot it about as fast as functions do and most tool calls are as short as function calls. Its contract is a misfit for an agent's tool calls. An agent's state is its whole sandbox, with its disk and its running processes, so a new sandbox for every call loses tasks even when the working directory or the whole disk is kept. Agents need a sandbox that lives as long as the session, and agent platforms now provide one [5, 9]. In return, the session gives up one memory size that fits every call and an idle sandbox that can be discarded. The sandbox service should therefore manage the agent session instead of the call. With the primitives of the OS and the VMM, it should size memory per command and free the sandbox while the model thinks, without losing state.

References

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, et al. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI*. 419–434.

[2] Amazon Web Services. 2026. Lambda Durable Functions. https://docs.aws.amazon.com/lambda/latest/dg/durable-functions.html.

[3] Amazon Web Services. 2026. Lambda Quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html.

[4] Amazon Web Services. 2026. Run Isolated Sandboxes with Full Lifecycle Control: AWS Lambda Introduces MicroVMs. https://aws.amazon.com/blogs/aws/run-isolated-sandboxes-with-full-lifecycle-control-aws-lambda-introduces-microvms/.

[5] Amazon Web Services. 2026. Use Isolated Sessions for Agents (Amazon Bedrock AgentCore). https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/runtime-sessions.html.

[6] Anthropic. 2026. Prompt caching. https://platform.claude.com/docs/en/build-with-claude/prompt-caching.

[7] Yunpeng Dong, Jingkai He, Shiqi Liu, et al. 2026. DeltaBox: Scaling Stateful AI Agents with Millisecond-Level Sandbox Checkpoint/Rollback. arXiv:2605.22781

[8] Ian Dougherty, Natalie Lambert, Joshua Wang, et al. 2026. Agentic Workflows are Serverless Applications, so deploy them that way! *ACM SIGOPS Oper. Syst. Rev.* 60, 1 (2026), 30–40. doi:10.1145/3830422.3830426

[9] E2B. 2026. E2B Runtime. https://github.com/e2b-dev/runtime.

[10] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, et al. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR*.

[11] Carlos E. Jimenez, John Yang, Alexander Wettig, et al. 2024. SWE-bench: Can Language Models Resolve Real-world GitHub Issues?. In *ICLR*. arXiv:2310.06770

[12] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, et al. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. UC Berkeley.

[13] Abbas Khater, Seyed Amirmasoud Noohi, Massoud Reza Hashemi, and Zeinab Zali. 2024. An NFV-Based Framework for Autonomous Deployment of New Protocols in SDN Networks. *IEEE Access* 12 (2024), 148727–148740. doi:10.1109/ACCESS.2024.3476244

[14] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *IEEE CLOUD*. 502–504. doi:10.1109/CLOUD.2019.00091

[15] Mike A. Merrill, Alexander Glenn Shaw, Nicholas Carlini, et al. 2026. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. In *ICLR*. arXiv:2601.11868

[16] Amir Noohi, Bita Asoodeh, and Antonio Barbalace. 2026. Towards a Solution to the Management Scaling Paradox in Distributed LLM Inference. In *EuroMLSys*. 183–191. doi:10.1145/3805621.3807658

[17] Amir Noohi, Mostafa Derispour, and Antonio Barbalace. 2025. RMAI: Rethinking Memory for AI (Inference): In-Kernel Remote Shared Memory as a Software Alternative to CXL. In *EuroMLSys*. 122–131. doi:10.1145/3721146.3721954

[18] Amir Noohi, Mostafa Derispour, Saba Ebrahimi, and Antonio Barbalace. 2026. NTSM: OS-Level Memory for Scaling AI Without Boundaries. In *Middleware*.

[19] Amir Noohi, Mostafa Derispour, Rodrigo C. O. Rocha, et al. 2026. SagePTE: Co-Locating Guest and Host PTEs to Accelerate Virtual Memory Translation. In *MICRO*.

[20] OpenAI. 2024. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/.

[21] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, et al. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *USENIX ATC*. 205–218.

[22] SWE-agent team. 2026. mini-SWE-agent Documentation. https://mini-swe-agent.com/latest/.

[23] Shiva Sai Krishna Anand Tokal, Vaibhav Jha, Anand Eswaran, et al. 2025. AgentX: Towards Orchestrating Robust Agentic Workflow Patterns with FaaS-hosted MCP Services. arXiv:2509.07595

[24] vHive Ecosystem. 2026. vSwarm: Serverless Benchmarking Suite. https://github.com/vhive-serverless/vSwarm.

[25] Shunyu Yao, Jeffrey Zhao, Dian Yu, et al. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR*. arXiv:2210.03629