Fuyun: Bridging the Semantic Gap in Serverless Resource Provisioning via LLM Agents

Qingwen Li
ICT, CAS; UCAS
Beijing, China
liqingwen21b@ict.ac.cn

Kai Lv
ICT, CAS; UCAS
Beijing, China
lvkai20z@ict.ac.cn

Mingxuan Yang
ICT, CAS; UCAS
Beijing, China
yangmingxuan20s@ict.ac.cn

Zhengyu Lei
ICT, CAS; UCAS
Beijing, China
leizhengyu20s@ict.ac.cn

Cunchi Lv
ICT, CAS; UCAS
Beijing, China
lvcunchi21s@ict.ac.cn

Xiao Shi*
ICT, CAS; UCAS
Beijing, China
shixiao@ict.ac.cn

Xiaofang Zhao
ICT, CAS; UCAS
Beijing, China
zhaoxf@ict.ac.cn

Abstract

Serverless platforms free developers from infrastructure management, yet resource provisioning for each function remains a major burden. Prior approaches based on Bayesian optimization or reinforcement learning typically treat functions as black boxes without program semantic understanding, which leads to high sample complexity and makes them impractical under tight production budgets.

We present Fuyun, a semantic-aware framework for serverless resource provisioning. Fuyun uses an LLM agent to synthesize a verifiable parametric policy from source code, and a lightweight runtime to instantiate the policy with just-in-time signals for fast, proactive configuration. An asynchronous tuning loop refines policy parameters from execution traces, enabling accurate and stable provisioning with near-zero runtime overhead.

Experiments show that Fuyun reaches full reliability with four times fewer profiling samples than Gaussian process Bayesian optimization. Meanwhile, it reduces the waste ratio by 44 percentage points compared to conservative static provisioning.

Keywords

Serverless Computing, Large Language Models, Resource Provisioning, Symbolic Policy Synthesis, AIOps, Online Tuning

1 Introduction

Serverless computing has emerged as a dominant cloud paradigm, widely adopted in the cloud-native ecosystem [12]. By relieving developers of infrastructure management, it allows them to focus on application logic packaged as serverless functions [10]. Upon deployment, platforms like AWS Lambda [5] automatically scale function instances in response to demand. Furthermore, serverless adopts a pay-as-you-go model, billing users primarily based on resource allocation and execution time rather than pre-provisioned capacity.

Despite these benefits, a pivotal challenge remains delegated to developers: determining the optimal resource configuration. Figure 1 illustrates the fundamental dilemma of static resource configuration. Any fixed memory limit forces an unavoidable trade-off between reliability and efficiency, as small inputs are over-provisioned while large inputs may fail due to insufficient memory.

Accurate configuration is critical for ensuring fast end-to-end execution while minimizing costs. This challenge stems from three primary factors. First, resource consumption is inherently coupled with input data characteristics, leading

*Corresponding author.

AgenticOS '26, Pittsburgh, PA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

to dynamic runtime demands. Second, serverless workloads are highly diverse. Some functions are compute and memory heavy, whereas others are dominated by I/O, so a single configuration rarely works well across functions. Third, the configuration space is large, which makes manual tuning impractical. For example, AWS Lambda allows memory settings from 128,MB to 10,240,MB in 1,MB increments, resulting in over 10,000 choices [4].

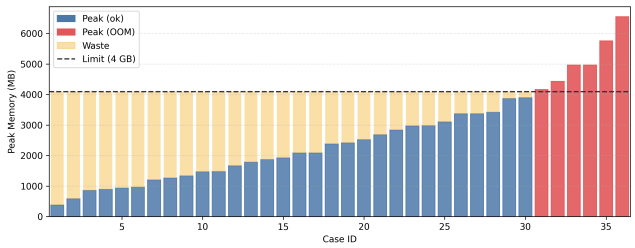

Figure 1: The dilemma of static resource configuration for serverless functions. A fixed resource limit forces a trade-off between reliability and efficiency: it wastes resources for small inputs (yellow) and causes OOM failures for large inputs (red).

A substantial body of research has explored automatic resource configuration in serverless systems. Prior works [1, 6, 19] apply Bayesian Optimization (BO), while others [16, 17, 20] use Reinforcement Learning (RL) to automate tuning. However, these approaches largely treat resource configuration as a black-box optimization problem: they identify good configurations through profiling outcomes or iterative online exploration, without leveraging source-code semantics to derive structured priors before execution. As a result, they often suffer from high sample complexity and require many costly optimization trials before converging to reliable configurations. For example, FaaSConf [17] uses 31K offline samples to train its multi-agent RL framework, which reportedly require about 43 days of data collection. Such reliance on large profiling budgets limits the practicality of these approaches in production environments with strict time, cost, and reliability constraints.

Recently, Large Language Models (LLMs) [2, 13] and agent-based frameworks [3, 15, 18, 21] have demonstrated strong performance in code generation and task scheduling. These successes highlight their advanced capabilities in semantic understanding and reasoning. We posit that these capabilities can be effectively leveraged to bridge the semantic gap in serverless resource provisioning.

However, natively integrating LLMs into the critical path of serverless systems introduces significant challenges:

First, there is a fundamental tension between the probabilistic nature of LLMs and the deterministic safety requirements of serverless infrastructure. As stochastic generators, LLMs inherently risk hallucinating arbitrary or non-compliant values. Directly mapping such unchecked outputs to infrastructure configuration poses severe operational hazards.

Second, a critical temporal mismatch exists between the high inference latency of LLMs and the strict real-time requirements of serverless execution. While serverless functions demand millisecond-level startup performance, LLM reasoning incurs multi-second delays; consequently, synchronously querying an LLM for every invocation is performance-prohibitive.

Third, a disconnect exists between the static rigidity of code-based predictions and the dynamic volatility of the runtime environment. Since inferences derived solely from source code cannot account for runtime noise or hardware heterogeneity, relying on them without continuous adaptation leads to inevitable accuracy drift.

To address these challenges, we propose Fuyun, a Semantic-Aware Resource Provisioning Framework. Fuyun reframes provisioning as a Symbolic Policy Synthesis task rather than a static numerical prediction task. By employing a rigorous Policy Schema and strict validation mechanisms, it effectively mitigates the risk of hallucination. Structurally, Fuyun operates on a decoupled architecture comprising an offline control plane and a runtime data plane. The former is responsible for synthesizing policies upon deployment and updating them post-execution; the latter is responsible for instantiating the latest policy with real-time signals to execute precise resource configuration. Furthermore, an evolutionary feedback loop continuously refines policy parameters based on execution ground truths.

In summary, this paper makes the following contributions:

- We identify semantic blindness in traditional black-box methods as a critical limitation. To the best of our knowledge, this is the first framework to leverage LLM agents for semantic-aware serverless resource provisioning, shifting the paradigm from statistical guessing to logical reasoning.
- We design a closed-loop, decoupled architecture that reconciles LLM latency with real-time demands. By employing a rigorous Policy Schema, our framework integrates verifiable static synthesis, deterministic just-in-time adjustment, and continuous evolutionary tuning, ensuring the system adapts to workload drifts with near-zero execution overhead.
- We implement a prototype and the experiments demonstrate that Fuyun eliminates OOM failures on volatile workloads, reduces the waste ratio by 44 percentage

points compared to conservative static provisioning, and reaches full reliability with four times fewer profiling samples than Gaussian-process Bayesian optimization.

2 System Design

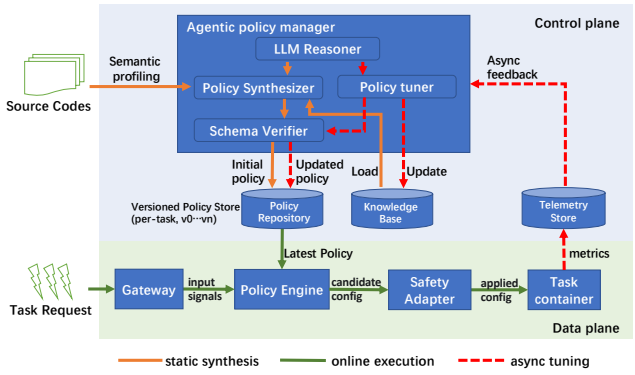

Figure 2: System Overview of Fuyun

2.1 Design Overview

The core philosophy of Fuyun is to elevate resource provisioning from heuristic guessing to Symbolic Resource Modeling. Unlike traditional reactive approaches, we treat resource configuration as a Semantic Policy Synthesis task.

As illustrated in Figure 2, Fuyun provisions resources through a decoupled generate-then-execute architecture with an asynchronous evolution loop. The design cleanly separates heavyweight policy reasoning in the control plane from latency-critical execution in the data plane.

Control plane: Static Policy Synthesis. Given task source code, the *Agentic Policy Manager (APM)* invokes an *LLM Reasoner* to analyze task semantics and synthesize a task-specific v_0 symbolic policy in a constrained DSL. The generated policy is then checked by a *Schema Verifier* to ensure that it is well-formed and schema-compliant for runtime execution. Validated policies are stored in the *Policy Repository*, where multiple policy versions are maintained for each task over time. The APM also maintains a *Knowledge Base* of reusable context distilled from prior executions, which grounds initial policy synthesis. By moving policy synthesis to the control plane, the system keeps LLM inference latency off the critical path of online request handling.

Data plane: Just-in-Time Policy Instantiation. At runtime, each incoming task request first passes through the *Gateway*, which extracts the *input signals* (e.g., request metadata or payload descriptors) required for policy instantiation. The *Policy Engine* then retrieves the *latest policy version* for the target task from the *Policy Repository* and deterministically instantiates it to produce a candidate configuration. Finally, the *Safety Adapter* converts the candidate configuration into the *applied configuration* for the *task container*, while enforcing system guardrails and provider-specific constraints. This lightweight Gateway→Engine→Adapter pipeline adds near-zero overhead to the critical path while enabling proactive, per-invocation resource selection.

Table 1: Policy schema overview (key fields).

Section	Field	Purpose
Metadata	policy_id, task_id, version	Identify the per-task policy and support versioned updates.
Metadata	status, supersedes	Control activation and rollback across versions.
Inputs	task_inputs.source	Specify the runtime context source. In our prototype, this source is the job ticket.
Inputs	variables	Declare symbolic variables, including x, with an explicit type and unit.
Inputs	mapping.x	Define required signals and a deterministic transform that computes x.
Logic	type	Specify the model family. Valid types are linear, piecewise linear, power, and exponential.
Logic	coefficients	Store tunable parameters required by the chosen model type.
Logic	decision_rationale	Record model-choice rationale and supporting evidence for auditability.

Control plane: Asynchronous Evolutionary Tuning. During execution, runtime telemetry are continuously collected from task containers and persists it in the *Telemetry Store*. The *Policy Tuner* asynchronously analyzes this stored telemetry in an off-path feedback loop. When the update condition is met, the tuner refines the policy coefficients, re-validates the updated policy, and stores it in the *Policy Repository* as a new version (v_{n+1}) for the corresponding task. This feedback loop allows Fuyun to converge quickly while maintaining stability under evolving workload characteristics.

2.2 Schema-Constrained Policy Specification

To mitigate LLM hallucinations and ensure executability, we require the agent to output policies in a strict domain-specific language defined by a JSON-based schema.

The schema has three top-level components: Metadata for provenance and versioning, Inputs for declaring observable signals and their deterministic mappings to symbolic variables, and Logic for specifying the model family, coefficients, and a brief decision rationale.

To handle heterogeneous task inputs, the schema represents runtime context through a symbolic variable x, computed from observable signals via a deterministic transform. During policy synthesis, the LLM uses code semantics to select the relevant signals and derive the signal-to-x mapping, then chooses the model family in the Logic section to fit the relationship between x and the target resource, recording its rationale for reproducibility.

Table 1 summarizes the key fields. This schema enables efficient runtime instantiation and safe versioned updates through verification and tuning.

2.3 Provider-Compatible Resource Adaptation

The Policy Engine instantiates the latest policy to produce a candidate resource target, but this value is not applied directly. Instead, it is passed through a Safety Adapter before deployment. In practice, the adapter first adds a small safety headroom to tolerate transient runtime spikes, and then projects the result to a valid provider-supported configuration.

Accordingly, we define the adaptation function $f_{\text{safe}}(\cdot)$ as:

$$f_{\text{safe}}(r) = \text{SnapToTier}_{[R_{\min}, R_{\max}]}(r \cdot (1 + \delta_{\text{headroom}})) \quad (1)$$

where δ_{headroom} is a configurable safety margin used to absorb short-lived demand spikes. The operator $\text{SnapToTier}_{[R_{\min}, R_{\max}]}$ projects the inflated continuous target to the nearest supported provider tier while respecting the valid configuration range. Here, $R_{\min}$ denotes the minimum allowed resource setting, and $R_{\max}$ denotes the effective upper bound determined jointly by provider limits and user quotas. This projection is necessary because serverless platforms typically expose discrete configuration options; for example, AWS Lambda requires memory to be specified as an integer value in MB. As a result, the Safety Adapter converts a policy-generated target into a bounded, provider-supported configuration that is safe to apply on the critical path.

2.4 Evolutionary Parameter Tuning

In the static policy synthesis stage, Fuyun selects a symbolic policy structure to model the task's peak resource usage. Rather than fitting coefficients from historical profiles that may be unavailable or outdated, the agent initializes coefficients through code semantics to produce a deployable v0 policy. The first data-driven refit is deferred to a short runtime warm-up period so that subsequent tuning relies on fresh and stable samples collected from the target environment.

During online execution, the *Policy Tuner* treats the policy coefficients as learnable parameters and refines them based on observed ground truth, for example, updating the *base* and *alpha* terms in a linear model. To avoid unnecessary churn, online tuning is triggered only when at least one of the following conditions holds: (i) an OOM occurs, (ii) the prediction error exceeds 2× the actual peak usage, or (iii) the task completes successfully for N consecutive executions.

We update policy parameters using numerical fitting methods consistent with the selected model structure. For a linear policy, we apply ordinary least squares regression. If the fitted parameters hit or exceed predefined bounds (e.g., upper/lower limits on α or *base*), or if the model cannot meet the target error threshold, we treat this as evidence that the current structural assumption is insufficient. In that case, we trigger a re-selection of the policy structure and re-fit the parameters under the new model to maintain feasibility and stability.

3 Implementation

We implemented a fully functional standalone prototype of Fuyun to demonstrate the feasibility of the proposed architecture. The prototype is developed in `Python 3.9` and consists of approximately 2,300 lines of code, organized into modular components for the control plane, data plane, and storage backend.

Control Plane Implementation: The agentic policy manager serves as the system's brain and is powered by `GPT-4o`. We utilized the OpenAI Python SDK to facilitate interaction with the LLM. To ensure the model generates compliant policies, we implemented a template-based prompt construction mechanism. This mechanism dynamically fills predefined slots with the policy schema (constraints and format definitions), real-time system signals, and relevant file contexts before dispatching the request.

Storage Management: We implemented a lightweight, file-system-based repository. Regarding policy storage, each generated policy is serialized into JSON format and stored as a distinct file, named with a monotonic version identifier to support rollback and history tracking. Additionally, the knowledge base utilizes a directory-based hierarchy that physically separates Global Knowledge (shared across all tasks) from Specific Knowledge (isolated per task). This separation minimizes context window consumption by only loading relevant knowledge during inference.

Data Plane Implementation: The data plane manages the actual execution of tasks. We utilized Docker [8] containers to provide isolated environments. To manage container lifecycles, we implemented a lightweight wrapper around the

Docker CLI [9] using Python's subprocess module [14]. This allows the system to programmatically spawn, monitor, and terminate containers without heavy external dependencies. Specifically, Fuyun parses the synthesized policy and maps its specifications directly to CLI arguments (e.g., `-memory`) during the container initialization phase. This proactive resource provisioning ensures that the execution environment is optimally configured before the workload begins, guaranteeing the smooth execution of the container. Real-time feedback signals are captured through standard output streams and resource monitoring APIs, which are then fed back to the control plane for the tuning loop.

4 Evaluation

Our evaluation is designed to answer the following questions:

(1) **Evolutionary Convergence:** Can Fuyun synthesize a suitable initial resource policy during static analysis, and can it converge rapidly with only a small number of samples while remaining stable during tuning? (§4.1)
(2) **Overall Performance:** Compared to other resource configuration methods, can Fuyun consistently achieve reliable execution with low resource waste across diverse inputs? (§4.2)

Benchmark. We use the *video processing* function as our benchmark, derived from SEBS [7], a widely used serverless benchmark suite. The function loads video frames into memory and applies a watermark using ffmpeg [11].

We evaluate the video-processing function using a corpus of input videos that cover diverse characteristics, including resolution (width/height), frame rate (FPS), and duration. We run the function on every video and record the observed peak memory usage for each invocation. To ensure both sets reflect the overall input distribution, we partition the dataset into a training set and a test set of equal size (36 samples each).

All experiments are conducted on a local server equipped with an Intel Core i7 CPU and 32,GB RAM, running Docker Engine 24.0.

4.1 Evolutionary Convergence

To answer the first research question regarding the quality of the initial policy and the speed of convergence, we evaluate Fuyun using a memory-intensive video processing workload. The experiment consists of two phases: static synthesis and dynamic adjustment.

Static Synthesis and Initial Policy. First, we validate whether Fuyun can synthesize a suitable initial policy without prior execution data. The LLM agent analyzes the source code of the video processing function and identifies that memory usage is highly correlated with video dimensions and frame count. Based on this reasoning, the agent selects a linear model to approximate the memory footprint:

$$y = base + \alpha \cdot (duration \cdot fps \cdot width \cdot height \cdot 3)$$

where the agent initializes the coefficients to $\alpha = 1.0$ and base = 0. This rational derivation provides a strictly better starting point than random initialization, reducing the search space for the subsequent dynamic phase.

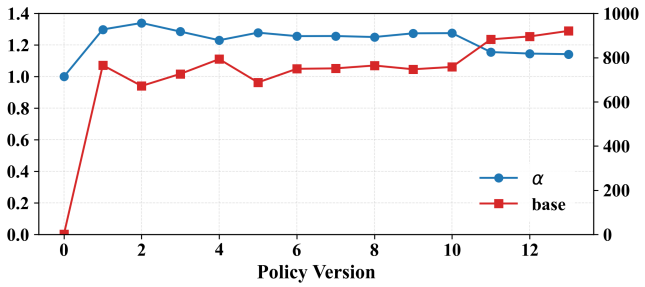

Figure 3: Evolution of linear model parameters. The parameters converge rapidly after the initial warm-up phase (6 samples) and remain stable throughout the remaining invocations.

Dynamic Adjustment and Stability. Next, we evaluate the convergence speed. We emulate a production scenario by replaying a trace of 36 task invocations. Fuyun operates in an online learning mode, where the policy is updated incrementally as new monitoring data arrives. We configured a "warm-up" period of 6 samples before the first numerical fitting occurs to avoid overfitting to outliers.

Figure 3 illustrates the convergence behavior of the linear model parameters (α and *base*) over the sequence of invocations. As shown in the figure, the parameters undergo a rapid adjustment phase immediately after the warm-up period (policy v0->v1), effectively correcting the initial static estimation to match the runtime reality. Crucially, after this brief tuning phase, both parameters stabilize and remain steady with minimal oscillation. This result indicates that Fuyun reaches a well-calibrated policy quickly and keeps the parameter updates stable over time, with no observable drift once the model has converged.

4.2 Performance overview

We evaluate Fuyun on the test set and report three policy versions obtained during training in Section 4.1, namely v1, v7, and v13. These versions correspond to the first, an intermediate, and the final policy update, respectively. All three versions achieve a success rate of 1.00 on the test set while keeping the waste ratio low (0.24, 0.22, and 0.23).

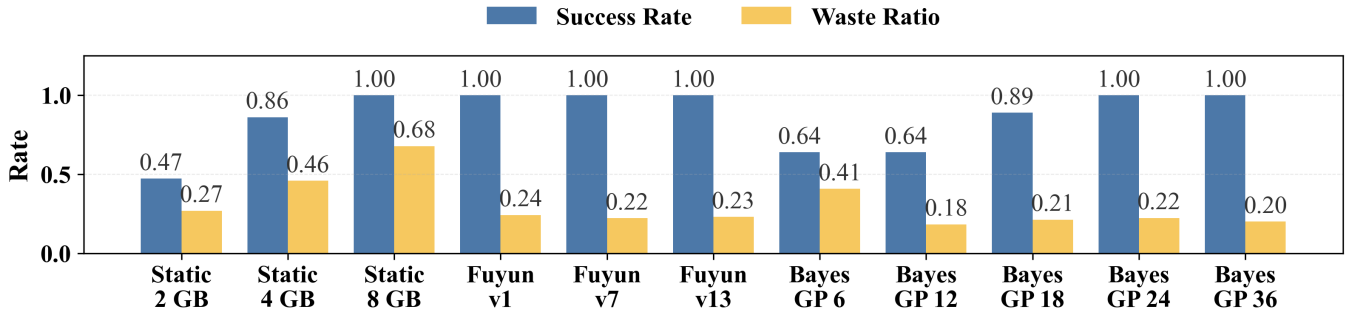

Figure 4: Reliability and efficiency on the test set. The left panel reports success rate and the right panel reports waste ratio. We compare Fuyun policy versions (v1, v7, v13), static provisioning with fixed memory limits (2 GB, 4 GB, 8 GB), and a Gaussian-process Bayesian optimization baseline trained with the first N profiling samples (GP 6/12/18/24/36).

We first compare against static provisioning with three fixed memory limits (2 GB, 4 GB, and 8 GB). With lower limits, reliability degrades substantially, with success rates of 0.47 (2 GB) and 0.86 (4 GB). Achieving full reliability requires provisioning 8 GB, but this incurs high waste (0.68). In contrast, Fuyun reaches the same success rate of 1.00 with much lower waste (0.22 to 0.24), reducing waste by 0.44 compared to the 8 GB static baseline.

We then compare against a representative black-box baseline based on Gaussian-process Bayesian Optimization (GP-BO). We run GP-BO using the first N samples from the training stream as profiling observations, where $N \in \{6, 12, 18, 24, 36\}$ (denoted as GP 6/12/18/24/36). As N increases, GP-BO improves steadily in both reliability and efficiency. However, matching Fuyun's full reliability requires substantially more data. GP-BO reaches a success rate of 1.00 only when $N = 24$, which is four times the data used by Fuyun v1, while achieving a similar waste ratio (0.22).

5 Conclusion

The inherent rigidity of static resource configuration has long forced serverless developers to choose between reliability and cost efficiency. Black-box optimization methods such as Bayesian optimization and reinforcement learning offer an alternative, but they suffer from high sample complexity and fail to capture the input-dependent dynamics of volatile workloads. In this work, we argue that robust provisioning requires a paradigm shift from numerical regression to semantic reasoning.

We proposed Fuyun, a framework that leverages Large Language Models to perform Symbolic Policy Synthesis. By decoupling heavy-weight reasoning from critical-path execution, Fuyun reconciles the high latency of generative agents with the strict real-time demands of serverless functions. Our architecture ensures operational safety through a rigorous policy schema and continuously adapts to runtime drifts via an evolutionary feedback loop.

Evaluation on input-sensitive video processing benchmarks shows that Fuyun eliminates OOM failures, reduces the waste ratio by 44 percentage points compared to conservative static provisioning, and reaches full reliability with four times fewer profiling samples than Gaussian-process Bayesian optimization. These findings suggest that deriving resource scaling patterns from code semantics is a promising path toward next-generation intelligent cloud infrastructure.

References

[1] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. 2020. COSE: Configuring serverless functions using statistical learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 129–138.

[2] Anthropic. 2024. The Claude 3 Model Family: Opus, Sonnet, Haiku. *Anthropic Technical Report* (2024).

[3] Anthropic. 2026. *Introducing Claude Opus 4.6.* Retrieved Feb 7, 2026 from https://www.anthropic.com/news/claude-opus-4-6

[4] Aws. 2026. *AWS Lambda.* Retrieved Feb 3, 2026 from https://aws.amazon.com/pm/lambda/

[5] Aws. 2026. *Lambda Configuration-memory guidance.* Retrieved Feb 7, 2026 from https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html

[6] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. 2023. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 381–397.

[7] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*. 64–78.

[8] Docker, Inc. 2026. Docker. https://www.docker.com/. Accessed: 2026-02-07.

[9] Docker, Inc. 2026. Docker CLI Reference. https://docs.docker.com/engine/reference/commandline/cli/. Accessed: 2026-02-07.

[10] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl

Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[11] Karl Kroening. 2019. ffmpeg-python. Python Package Index (PyPI). https://pypi.org/project/ffmpeg-python/ Version 0.2.0, released July 6, 2019.

[12] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. Faasflow: Enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th acm international conference on architectural support for programming languages and operating systems*. 782–796.

[13] OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).

[14] Python Software Foundation. 2026. subprocess — Subprocess management. https://docs.python.org/3/library/subprocess.html. Accessed: 2026-02-09.

[15] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 15174–15186.

[16] Haoran Qiu, Weichao Mao, Archit Patke, Chen Wang, Hubertus Franke, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. 2022. SIMPPO: A scalable and incremental online learning framework for serverless resource management. In *Proceedings of the 13th Symposium on Cloud Computing*. 306–322.

[17] Yilun Wang, Pengfei Chen, Hui Dou, Yiwen Zhang, Guangba Yu, Zilong He, and Haiyu Huang. 2024. FaaSConf: QoS-aware Hybrid Resources Configuration for Serverless Workflows. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 957–969.

[18] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. AutoGen: Enable Next-Gen Large Language Model Applications. https://github.com/microsoft/autogen

[19] Guangba Yu, Pengfei Chen, Zibin Zheng, Jingrun Zhang, Xiaoyun Li, and Zilong He. 2023. Faasdeliver: Cost-efficient and qos-aware function delivery in computing continuum. *IEEE Transactions on Services Computing* 16, 5 (2023), 3332–3347.

[20] Hanfei Yu, Athirai A Irissappane, Hao Wang, and Wes J Lloyd. 2021. Faasrank: Learning to schedule functions in serverless platforms. In *2021 IEEE international conference on autonomic computing and self-organizing systems (ACSOS)*. IEEE, 31–40.

[21] YUSHENG ZHENG, YanPeng Hu, Wei Zhang, and Andi Quinn. [n. d.]. Towards Agentic OS: An LLM Agent Framework for Linux Schedulers. In *Machine Learning for Systems 2025*.