Serverless: Making Your Computing Quantifiable

Changyue Li, 224040358
The Chinese University of Hong Kong, Shenzhen

***Abstract*—Serverless computing enables developers to focus on application logic by abstracting away infrastructure management. By leveraging Function-as-a-Service and Backend-as-a-Service models, serverless platforms provide cost-effective, scalable, and event-driven solutions for modern applications. Despite its advantages, serverless computing faces challenges such as cold start latency, programming complexity, resource allocation inefficiencies, and security vulnerabilities.**

This paper addresses these challenges through a comprehensive exploration of four critical aspects: cold start performance, programming frameworks, resource management, and security. It proposes strategies to mitigate cold start delays via techniques like container prewarming and snapshot restoration, evaluates programming frameworks such as OpenFaaS for enhancing developer productivity, and introduces resource management approaches to optimize dynamic provisioning and workload balancing. Additionally, it identifies key security risks and presents innovative solutions, including machine learning for anomaly detection and fine-grained access control.

By tackling these issues, this work provides actionable insights to optimize serverless architectures, paving the way for their broader adoption in complex applications.

***Index Terms*—serverless computing, cold start optimization, resource management, serverless security**

I. Introduction

Serverless computing allows a cloud customer to run their code in production without configuring and allocating the software and the infrastructure stack [1]. A cloud customer can thus focus on their application, rather than on managing the production environment. Serverless computing supports two primary service models: Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS) [2]. FaaS allows developers to build applications as small, event-driven functions (i.e., serverless functions), while BaaS provides ready-to-use cloud services such as storage (e.g., AWS S3 [3]), database, and API gateway management. This collaboration between FaaS and BaaS enables developers to efficiently create serverless applications.

The serverless computing model has been successfully adopted in a wide range of application domains, such as for ingesting IoT data for flood warnings [138] and for air quality monitoring and alerting in smart ports [139]. There are open-source serverless deployments such as OpenFaas, Apache OpenWhisk and KNative[1], and commercial ones such as AWS Lambda [2], Google Cloud Functions [3] and Azure Functions [4]. Its many advantages include: (a) lower deployment costs where many of the management tasks are taken care by the platform provider and as a result users do not explicitly provision or configure virtual machines (VMs), (b) resource elasticity where applications can scale up to tens of thousands of cloud functions on demand, in seconds, with no advance notice, and (c) lower operational costs based on a pay-as-you-use policy where users only get charged based on the number of resources consumed by the application functions during execution [140].

Despite its advantages, Serverless computing also presents unique challenges and areas for optimization. In this paper, we focus on four critical aspects of Serverless: Cold Start Performance, Programming Frameworks, Resource Management, and Security. Each of these topics addresses fundamental limitations and opportunities within the Serverless ecosystem:

- Cold Start Performance: Serverless functions often face latency due to initialization delays [4], particularly in high-demand or large-scale scenarios. We analyze the impact of cold starts on user experience and system throughput, explore strategies like container prewarming, snapshot restoration, and optimized scheduling, and propose methods to mitigate these delays without compromising resource efficiency.
- Programming Frameworks: Building serverless applications demands robust frameworks that abstract complexities while supporting diverse use cases [5]. We evaluate existing frameworks such as OpenFaaS and Knative, focusing on their ability to streamline function orchestration, enable multi-cloud compatibility, and support emerging workloads like AI/ML inference [6]–[8]. Additionally, we propose enhancements to improve developer productivity and support for advanced workflows [9].
- Resource Management: The elasticity of serverless computing relies on intelligent resource allocation to handle varying workloads effectively [145]. We investigate techniques for dynamic provisioning, task scheduling, and workload prediction, with an emphasis on reducing underutilization and addressing bottlenecks in compute and memory [147], [148]. We also analyze trade-offs between cost efficiency and performance, proposing solutions tailored to heterogeneous environments.
- Security: The serverless model introduces new security challenges, including misconfigurations in function deployment, vulnerabilities in third-party dependencies, and risks of data breaches during function invocation [10]. We study strategies for safeguarding serverless environments,

[1]https://knative.dev/
[2]https://aws.amazon.com/lambda/
[3]https://cloud.google.com/functions
[4]https://azure.microsoft.com/en-gb/products/functions/

including static and dynamic analysis of configurations, runtime monitoring [11], and isolation mechanisms [12]. Furthermore, we examine innovative methods, such as leveraging machine learning for anomaly detection and employing fine-grained access control, to enhance security without undermining the agility of serverless architectures.

Through detailed exploration of these topics, this paper aims to provide actionable insights into optimizing serverless computing. By addressing the inherent challenges in cold starts, programming frameworks, resource management, and security, we contribute to advancing the adoption and efficiency of serverless systems, paving the way for their integration into more complex and demanding applications.

II. Related Work

A. Cold Start Performance

1) Bin packing with setup times: Weng et al. [13] study similar problem of minimizing mean weighted completion time in case of tasks with sequence dependent setup times. [14] presents dynamic algorithms addressing scheduling with setup times with objective of minimal weighted flow time.

2) Quadratic programming: Existing study [15] was unable to find an optimal schedule in 15 minutes (on a reasonable desktop machine) even for a small instance with $N = 20$ jobs each of $n_l = 20$ tasks. Schedulers in production systems need to respond in seconds, thus an approach based on a generic solver is probably not sufficient.

3) Workflow scheduling: [16] measures how inaccurate runtime estimates influence the schedules — which complements our study (as we assumed that estimates are known). [17] analyzes possible performance benefits of resource interleaving across the parallel stages. [18] proposes Balanced Minimum Completion Time, an algorithm for scheduling tasks with dependencies (and without setup times) on heterogeneous systems. [19] schedules workflows with setup times using branch-and-bound. While they considered small instances (up to $N * n_l = 100$ task and $m = 4$ machines); their method required 100s time limit for execution. Such long running times makes this method unusable in data-center schedulers. A comprehensive survey on workflow scheduling in the cloud is presented in [20]. [21] analyzes scheduling tasks with sequence-dependent setup times, precedence constraints, release dates on unrelated machines with resource constraints and machine eligibility. The authors present two solutions: based on genetic algorithm and based on an artificial immune system. Their largest instances had 60 tasks and 8 machines and needed 25 minutes (on the average) to solve, again rendering these methods unusable for FaaS.

B. Programming Framework

1) Serverless inference systems: Extensive research has focused on optimizing ML model serving in serverless architectures, targeting batching [22]–[24], scheduling [25], [26], and resource efficiency [27], [28]. Industry solutions like AWS SageMaker and Azure ML [29], along with the open-source KServe [30], demonstrate practical implementations. Despite these advancements, serverless inference systems still perform suboptimally with LLMs, as our paper demonstrates.

2) Exploiting locality in serverless systems: Locality plays a crucial role in various optimization strategies for serverless systems. This includes leveraging host memory and local storage for data cache [46]–[48], optimizing the reading of shared logs [49], and enhancing communication efficiency in serverless Directed Acyclic Graphs (DAGs) [50], [51]. ServerlessLLM, distinct from existing methods, introduces a high-performance checkpoint cache for GPUs, markedly improving checkpoint loading from multi-tier local storage to GPU memory. Recent studies [52], [53] have also recognized the need for leveraging locality in orchestrating serverless functions. Beyond these studies, ServerlessLLM leverages LLM-specific characteristics in improving the locality-based server's selection and launching locality-driven inference.

3) LLM serving systems: Recent advancements in LLM serving have improved inference latency and throughput. Orca [54] uses continuous batching for better GPU utilization during inference. AlpaServe [55] shows that model parallelism can enhance throughput while meeting SLO constraints, though it has yet to be tested on generative models. vLLM [56] introduces PagedAttention for efficient KV cache management. SplitWise [57] improves throughput by distributing prompt and token generation phases across different machines. Some approaches [58], [59] also use storage devices to offload parameters from GPUs to manage large LLM sizes. However, these systems often overlook model loading challenges, leading to increased first token latencies when multiple models share GPUs. ServerlessLLM addresses this by focusing on minimizing loading latency to complement these throughput and latency optimizations.

C. Resource Management

1) Container Optimization: The authors of the paper [145] focus on how to optimize the container creation by using shortcuts based on checkpoint-and-restore procedures, without the need of recreating the docker container image from the very first step. The authors of [146] improve the container boot process to achieve cold starts in the low hundreds of milliseconds. The work of [177] proposes a new lightweight isolation mechanism which, in order to reduce initialisation times, restores Faaslets from already-initialised snapshots.

2) Prediction methods: The authors of the paper [147] focus on reducing the number of cold starts in a serverless environment. They propose an adaptive resource management policy called hybrid histogram for prewarming and keep-alive time windows. They also use ARIMA modeling for applications that have infrequent invocations. This is considered as a state-of-the-art technique. However, their work does not consider and exploit the similarity of serverless applications as we do in our work. The authors of this work [148] propose a keep-alive policy based on a Greedy Dual Size Frequency caching scheme, which relies in the container pool of the OpenWhisk serverless platform. The work actually replaces

the default TTL scheme which is the typical case of the industry standards. The authors use a function hit-ratio curve for determining the percentage of warm-starts at different server memory sizes. The work of [178], is implemented on top of Apache Storm and incorporates a congestion-aware scheduler and a fixed-size worker pool into an edge friendly Streaming process environment, but this could not be applied to our setting, in which we focus on batches of requests.

D. Security

1) Traditional Misconfiguration Detection: Existing misconfiguration detection methods can be categorized into two types: white-box and black-box approaches. White-box approaches [60]–[64] generally focus on source code or program analysis to identify misconfigurations within the codebase, relying on manually defined domain-specific rules. For example, Rex [60] detected dependency violations between source code and configurations that must be updated together. Ctest [61] identified configuration-induced failures in code affected by configuration changes. SPEX [62] employed static program analysis to infer configuration constraints, designing predefined rules from variables in the source code to uncover misconfiguration vulnerabilities.

However, these methods are not well-suited for detecting misconfigurations in serverless applications, which rely on YAML-based configuration files rather than source code structures. Serverless-specific misconfigurations, embedded in configuration files, require new approaches that extend beyond traditional white-box techniques.

Black-box approaches [65]–[68] are generally data-driven and rely on learning configuration patterns from a dataset of example configurations. For instance, EnCore [65] used numerous configurations to learn and customize rule templates, inferring correlations and detecting misconfigurations in server applications. ConfigC [66] analyzed a dataset of correct configurations to build a language model that could detect errors in new configurations. DRIVE [67] created a Dockerfiles dataset and applied sequential pattern mining to extract frequent patterns, identifying rule violations through heuristic-based reduction and human intervention. However, these data-driven methods have inherent limitations: (i) They require a well-curated dataset, but ensuring the completeness and correctness of such datasets is challenging. As a result, configurations not represented in the training data may be missed, while normal configurations might be incorrectly flagged as anomalies due to dataset gaps. (ii) To compensate for dataset issues, these methods incorporate domain-specific knowledge (e.g., customized rule templates), requiring significant manual effort and continuous checking. These limitations hinder the practical application of data-driven approaches. Our results on RQ1 show that such approaches are less effective in our scenario.

2) LLM-based Misconfiguration Detection: LLM-based approaches offer a promising alternative. A recent paper presented Ciri [69], an LLM-based configuration validator. It demonstrated the potential of LLMs for detecting misconfigurations in systems such as Alluxio, Django, Etcd, and HDFS. However, Ciri depends on an external database containing valid configurations, misconfigurations, related questions, and ground-truth responses. Constructing this database is costly and challenging for various scenarios. In contrast, *SlsDetector* employs zero-shot learning that does not require external datasets, eliminating the need for predefined data. On the other hand, Ciri used a prompt without any constraint, limiting its ability to detect dependencies [69]. Serverless applications have complex configuration structures and stronger interdependencies, making simple prompt-based methods less effective. Our results on RQ2 and RQ3 show that such a method (i.e., BL method) is less effective in our scenario. Instead, *SlsDetector* incorporates carefully designed multi-dimensional constraints without predefined data, providing a more effective detection for serverless application configurations.

III. Cold Start Performance

A. Modeling FaaS Resource Management

1) Resource Management in OpenWhisk: In this section, we describe from the resource management perspective a representative implementation of a serverless cloud platform, the open-source Apache OpenWhisk [70]. OpenWhisk is mature, actively-developed software also offered commercially (IBM Cloud Functions, Adobe I/O Runtime). OpenWhisk alternatives include OpenLambda [150] and Fission [72]. OpenLambda uses containers to provide runtime environment for functions. Fission is designed for Kubernetes [73]; it can be deployed on existing cluster among other applications, which makes its adoption significantly easier. This section forms a background for our scheduling model that follows in Section III-A2.

OpenWhisk allows a cloud *customer* to upload *functions* (essentially, code snippets). A function is executed when *end-users* issue requests. A function executes in an *environment* — an initialized Docker container. Different Docker images are used for each of supported languages; a customer can also provide a custom image (with, e.g., additional libraries). Before the first execution of a function, the container must be initialized (e.g., setting up the container or compiling a Go function). This initialization can take a considerable amount of time (called later the *setup time*) — [74] reports at least 500ms. An environment is specific to a function (it is not reused between different functions). Subsequent invocations may reuse the same environment (no further setup times are necessary). By default, in OpenWhisk an environment executes at most a single invocation at any given moment.

OpenWhisk also allows to compose several functions into a chain (a sequence). After one function finishes, its result are passed to the next function; the last function responds to the end-user. While sequences are natively supported, in order to spawn two or more functions in parallel (resulting in a DAG), the developer may use an additional OpenWhisk Composer module or call the OpenWhisk API from the function code.

Architecture of OpenWhisk is complex. However, from our perspective the key components are the *controller* and the

invoker. The controller communicates with the invokers by message passing (via Apache Kafka).

The invoker is an agent program running on a worker node. The invoker is responsible for executing actions scheduled on a particular node. Each invoker has a unique identifier; it announces itself to the controller while starting.

The controller acts as a scheduler handling incoming events and routing function invocations to invokers. The controller monitors the status of workers and the currently executing invocations.

The controller attempts to balance load across nodes. The algorithm selects the initial worker node for each function based on a hash of the workspace name and the function name. Similarly, the algorithm picks for each function another number, called the *step size* (a number co-prime with the count of worker nodes). Each time a function is invoked, the controller attempts to schedule the invocation on its initial worker. If a worker doesn't have sufficient resources immediately available, the controller tries to schedule the invocation on the next node (increased by the *step size*). If the invocation cannot be immediately scheduled on any node, it is queued on a randomly chosen node.

2) A Scheduling Model for FaaS: In this section we define the optimization model for the FaaS resource management problem. The aim of this model is to have the simplest possible (yet still realistic) approximation of a FaaS system that enables us to show that considering FaaS compositions allow optimizations. We thus deliberately do not take into account some factors that we argue are orthogonal for this work.

We use the standard notation from [75]. A single end-user request corresponds to a *job* J_i. A job is composed of one or more *tasks* $O_{i,k}$, each corresponding to a single FaaS invocation. The request is responded to (the job completes) at time C_i when the last task completes, $C_i = \max_j C_{i,j}$. Tasks have dependencies resulting from, e.g., before-after relationships in the code. While in general such dependencies can be modeled by a DAG, in this work we concentrate on chains of tasks, i.e., task $O_{i,k+1}$ starts (at time $\sigma_{i,k+1}$) only after $O_{i,k}$ completes, $\sigma_{i,k+1} \geq C_{i,k}$ (we show additional results for DAGs in [76]).

We assume that individual functions are repeatedly executed (modeling similar requests from many end-users but also shared modules like authorization). We model such grouping by mapping each task $O_{i,k}$ to exactly one family $f(O_{i,k})$ (obviously, two tasks $O_{i,k}$ and $O_{i,l}$ from a job J_i might belong to different families). All tasks from a family f require the same environment E_f, have the same execution time (duration) p_f and require the same amount of resources q_f.

A task $O_{i,k}$ from a family $f(O_{i,k})$ is executed on exactly one machine in an *environment* (OpenWhisk container) E_f. E_f requires set-up time s_f (initialization of the environment) before executing the first task (subsequent tasks do not require set-up times). Typically, s_f is non-negligible and longer than the task's duration, $s_f > p_f$ (but we don't assume this).

A machine commonly hosts many environments (thus supporting parallel execution of tasks). Since the moment the environment's preparation starts – and until it is removed – each environment e_f uses q_f of the machine's resources (e.g., bytes of memory) whether a task executes or not. The number of hosted environments is limited by the capacity of the machine Q ($\sum q_f \leq Q$). We consider only a single dimension of the resource requests as OpenWhisk assumes a linear relation between memory and CPU limits of the underlying containers. Similarly, Google Cloud Functions allow customers to specify only a single dimension (memory requirement). However, it should be relatively easy to extend our model to vector packing [77].

We do not consider the additional latency caused by communication between tasks because we assume that a high-throughput, low-latency network of a modern datacenter is less of a limit than the link between the datacenter and the Internet. We assume that the machines are homogeneous (machine resources Q and execution times p_f are the same). If a FaaS system is deployed on VMs rented from an IaaS cloud, it is natural to use a Managed Instance Group (MIG) that requires all VMs to have the same instance type. If FaaS is deployed on a bare-metal data-center, the amount of machines having the same hardware configuration should be higher that other scalability limits (e.g. at a Google data-center, 98% of machines from a 10,000-machine cluster belong to one of just 4 hardware configurations [78]).

We assume that *jobs* have no release times, i.e., the first tasks of all the jobs are ready to be scheduled at time 0. This assumption approximates a system under peak load — there is a queue of requests to be scheduled now. Note that in contrast to *jobs*, individual tasks (in particular, the tasks that follow the first task of a job) do have non-zero release times, resulting from inter-task dependencies.

Our model is *clairvoyant*. A FaaS system repeatedly (thousands of times) executes individual functions. Thus, once a particular family is known for some time, q_f, p_f and the function structure should be easy to estimate using standard statistical methods — and before that, the system can use conservative upper bounds (e.g., defaults used by OpenWhisk). [79] shows that even simple methods estimate precisely memory and CPU for long-running containers (which, in principle, is harder than estimating FaaS, as FaaS are shorter, thus repeated much more frequently than a container).

The system optimizes the average response latency. As all N jobs are ready at time 0, this metric corresponds to $\frac{1}{N}\sum_{i=1}^{N} C_i$.

To summarize, the scheduling problem consists of finding for each task $O_{i,k}$ a machine and a start time $\sigma_{i,k}$ so that:

1) at $\sigma_{i,k}$, there is a prepared environment for $f(O_{i,k})$ on that machine that does not execute any other task during $[\sigma_{i,k}, \sigma_{i,k} + p_f]$ (a scheduling constraint);
2) dependencies are fulfilled: if $k > 1$, $\sigma_{i,k} \geq C_{i,k-1}$ (a dependency constraint);
3) at any time, for each machine, the sum of requirements of the installed environments is smaller than the machine

capacity (a multiple knapsack constraint).

This problem is NP-hard, as generalizing several NP-hard problems (knapsack [80], $P2|chains|\sum C_i$ [75]).

B. Method Description

In this section we describe heuristics to schedule FaaS invocations. We decompose the FaaS scheduling problem into three aspects: *sequencing* of invocations; *deployment* of execution environments on machines; and *allocation* of invocations to deployed environments. We describe specific *Sequencing* corresponds to the ordering policy (Section III-B1) and the awareness of task dependencies (Section III-B4). *Deployment* corresponds to the removal policy (Section III-B2). *Allocation* corresponds to the waiting/non-waiting variants (Section III-B3).

The framework algorithm is a standard scheduling loop executing *schedulingStep* at time t when at least one task completes. The algorithm maintains a queue of tasks $[O_{i,k}]$ to schedule.

1) Queue the successors $O_{i,k+1}$ of tasks completed at t ($\{O_{i,k} : \sigma_{i,k} + p_f = t\}$) (*queueDependentTasks*).
2) Apply a scheduling policy to the queued tasks (*Order*).
3) Try to find an environment e for each queued task:
   a) Try to claim an initialized environment of the required type (*FindUnusedEnvironment*, and – if $wait$ – *FindEnvironmentToWait*). In this step we iterate over all machines and take the first matching environment. (Section III-B3 describes the $wait$ variant).
   b) If (a) fails, try to create a new environment without removing any existing one (*PlaceNewEnvironment*). As above, we use the first fitting machine.
   c) If (b) fails, try to find a machine with sufficient capacity for e that is currently claimed by environments that do not execute any task; remove these environments, and install e (*RemoveAndPlaceEnvironment*).
   d) If (c) fails, the task remains in the queue.
4) If an environment e is found, assign the task (*AssignTask*); otherwise (3.a-c all fail) the task remains in the queue.

AssignTask starts a task on an environment as follows. Each environment has a queue of assigned task. Immediately after creating an environment, it is initialized (which takes time s_f). Then, the environment starts to execute tasks sequentially from its queue. If the head task is not ready (waiting for dependencies), the environment waits (no backfilling). This may happen in the *start* policy (see Section III-B4).

In the following, we propose concrete variants for these functions. We denote the full scheduling policy by a tuple (A, B, C, D), e.g., , $(FIFO, LRU, wait, start)$, where A denotes the ordering policy, B denotes the removal policy, C indicates if variant is *waiting* and D describes whether the variant is dependency-aware.

1) Ordering policy (Order): We compare the standard FIFO and SJF with three orderings taking into account the dependencies:

- *FIFO (First Come First Served)* – use the order in which the tasks were added.
- *EF (Existing First)* – partition the tasks into two groups: (1) there is at least one idle, initialized environment e of matching type $E_{f(O_{i,k})}$; (2) the rest. Schedule the first group before the second group. The relative order of the tasks in both groups remains stable (FIFO). For example, if queue contains five tasks $[O_{i_1,k_1}, O_{i_2,k_2}, O_{i_3,k_3}, O_{i_4,k_4}, O_{i_5,k_5}]$, there is only one environment e that is idle and only tasks O_{i_1,k_1}, O_{i_3,k_3}, O_{i_4,k_4} require environment with type matching e, the resulting order is $[O_{i_1,k_1}, O_{i_3,k_3}, O_{i_4,k_4}, O_{i_2,k_2}, O_{i_5,k_5}]$.
- *SJF (Shortest Jobs First)* – order by increasing durations p_f;
- *SW (Smallest Work)* – order by increasing remaining work *in a job*, i.e. for a task $O_{i,k}$, order by $\sum_{k' \geq k} p_{f(O_{i,k'})}$.
- *RT (Release Time)* – ordered by the time the task's predecessors are completed.

2) Removal policy: RemoveAndPlaceEnvironment removes environments according to either a standard LRU, or one of policies considering either initialization time s_f or environment popularity:

- LRU – remove the LRU (Least Recently Used) environment(s) from first fitting machine (i.e. having enough space to be freed).
- min time removal – remove the environment(s) with the smallest setup time s_f (if more than one, select a single machine having environments with the smallest total s_f).
- min family removal – remove the environment(s) from the family with the highest number of currently initialized environments. As it may be needed to remove more than one environment, choose a machine to minimize resulting number of families without any environment.

3) Greedy environment creation: If there is no unused environment of the required type E_f, a greedy algorithm (i.e. when *wait* is false) just attempts to create a new one. However, when setup times s_f are longer than task's duration p_f, it might be faster just to wait until one of currently initialized environments completes its assigned task. When no idle environment is available, function *FindEnvironmentToWait* computes for each initialized environment e of type E_f the time C_e the last task currently assigned to this environment completes. If an environment e^* is available sooner than the time needed to set up a new environment ($\min C_e \leq t + s_f$), the task is assigned to e^*. This variant use the (limited) clairvoyance of the scheduler by taking into account the knowledge of tasks' durations and setup times of their execution environments.

The *waiting* variant is analogous to scheduling tasks in Heterogeneous Earliest Finish Time (HEFT [81], [82]) that places a task on a processor that will finish the task as the earliest.

4) Awareness of task dependencies: A myopic (*default*) scheduler queues just the tasks that are currently ready to execute: $O_{i,0}$ (the first tasks in the jobs), or the tasks for which the predecessors completed $\{O_{i,k} : C_{i,k-1} \leq t\}$. However,

when a task's $O_{i,k}$ predecessors complete, it might happen that there is no idle environment $e_{f(O_{i,k})}$, and thus $O_{i,k}$ must still wait s_f until a new environment is initialized.

We propose two policies, *start* and *start with break (stbr)*, that use the structure of the job to prepare environments in advance. Both policies put the successor $O_{i,k+1}$ to the end of the queue when scheduling $O_{i,k}$; the successor has the release time $t+p_{f(O_{i,k})}$ (the time when $O_{i,k}$ completes). The notion of the release time allows us to block $O_{i,k+1}$'s execution until it is ready (as described in *AssignTask*). Note that *start* and *stbr* may result in an environment that is (temporarily) blocked: e.g., if an empty system schedules a chain of two tasks, the second task from the chain is added to the queue immediately after scheduling the first task; this second task will be assigned to its environment, but cannot be started until the first task is completed. In *start* variant, after *schedulingStep* completes and new tasks were added to queue, scheduler tries placing them following the same procedure. Compared with *start*, *stbr* immediately after adding $O_{i,k+1}$ reorders tasks in the queue according to the scheduling policy and restarts the placement.

C. Evaluation

We evaluate our algorithms with a calibrated simulator. We use a simulator rather than modify the OpenWhisk scheduler for the following reasons. First, a discrete-time simulator enables us to execute much more test scenarios and on a considerably larger scale (we perform tests on $1440 \cdot 15$ problem instances). Second, as our results will show, to schedule tasks more efficiently, the OpenWhisk controller (the central scheduler) should take over some of the decisions currently made by the invokers (agents residing on machines). For example, *min family removal* needs to know which family has the highest number of installed environments in the whole cluster — thus, the state of the whole cluster (note that this policy can be implemented in a distributed way: the cluster state can be broadcasted to the invokers). To ensure that our simulator's results can be generalized to an OpenWhisk installation, we compare the performance of an actual OpenWhisk system with its simulation; the Pearson correlation between these results is very high (Section III-C2).

1) Method: To test the performance of our algorithms, we generated synthetic instances with a wide range of parameter values. We are not aware of any publicly-available workloads for FaaS or related systems (having dependencies, function families and setup times). Nevertheless, we also attempted to create instances resembling real scenarios by using Google Cluster Trace [83] and generating only missing data. We present results of this approach in the Appendix [76].

Many parameters of instances have a relative, rather than absolute, effect on the result. For example, multiplying by a constant both Q, the machine capacity, and q_f, the size of the task, results in an instance that has very similar scheduling properties. There is a similar relationship between setup times s_f and durations p_f; and between the total number of tasks n and the number of tasks in a chain l. We thus fix one parameter from each pair to a constant (or a small range); and vary the other. We have $n = 1000$ tasks; p_f is generated by the uniform distribution over integers $p_f \sim U[1, 10]$; similarly $q_f \sim U[1, 10]$. The remaining parameters have ranges:

- family count n_f: 10, 20, 50, 100, 200, 500;
- setup times s_f: $[0, 0]$, $[10, 20]$, $[100, 200]$, $[1000, 2000]$;
- chain lengths l: $[2, 10]$, $[10, 20]$, $[50, 100]$;
- machine count m: 2, 5, 10, 20, 50;
- machine sizes Q: 10, 20, 50.

For each combination of the parameters (or ranges) n_f, s_f, l, we generate 20 random instances, resulting in 1440 instances. We evaluate each instance on each of the 15 machine environments.

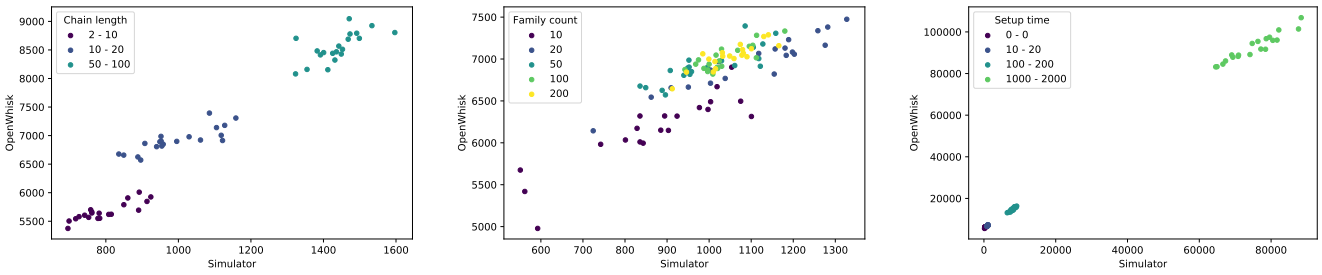
These ranges of parameters are wide. As we experiment on synthetic data, one of our goals is to explore trends – characterize instances for which our proposed method works better (or worse) than the current baseline. In particular, chains longer than 10 ($l > 10$) are longer than what we suspect is the current FaaS usage. On the other hand, it is not a lot compared with a call graph depth on any non-trivial software. At this point of FaaS evolution it is difficult to foresee the degree of compartmentalization future FaaS software will have – and chains longer than 10 invocations represent fine-grained decomposition (similar to modern non-FaaS software).

Given n_f, $[s_{\min}, s_{\max}]$, $[l_{\min}, l_{\max}]$ we generate an instance as follows. For each of n_f, we set $s_f \sim U[s_{\min}, s_{\max}]$ and $p_f \sim U[1, 10]$. For each of $n = 1000$ tasks, we set its family f to $U[1, n_f]$. We then chain tasks to jobs. Until all tasks are assigned, we are creating jobs by, first, setting the number of tasks in a job to $l \sim U[l_{\min}, l_{\max}]$ (the last created job could be smaller, taking the remaining tasks); and then choosing l unassigned tasks and putting them in a random sequence.

For each experiment, our simulator computes the average response latency, $(1/n)\sum C_i$. Due to space constraints, we omit results on tail, 95%-ile latency – the 95%-ile results also support our conclusions (unsurprisingly, the ranges are larger than for the averages).

We simulate the current, round-robin behavior of the OpenWhisk scheduler (Section III-A1) with an algorithm *OW*. *OW* randomly selects for each family f the initial machine m_f and the *step size* k_f, an integer co-prime with the number of machines m. When scheduling a task $O_{i,k}$ in family f, *OW* checks machines m_f, $m_f + k_f$, $m_f + 2k_f$, ...(all additions modulo m), stopping at the first machine that has either the environment E_f ready to process, or q_f free resources (including unused environments that could be removed) to install a new environment E_f. If there is no such machine, $O_{i,k}$ is queued on a randomly-chosen machine.

2) Validation of the simulator against OpenWhisk: To compare the results of our simulator with OpenWhisk, we developed a customized OpenWhisk execution environment that emulates a function with a certain setup time s_f, execution time p_f and resource requirement q_f. This environment emulates initialization by sleeping for $s_f * 10ms$; and it emulates execution by sleeping for $p_f * 10ms$. While sleeping does not use the requested memory ($q_f * 128MB$), the memory is blocked (through Linux cgroup limits) and therefore cannot be



(a) by chain length ($n_f = 50$, $s_f \in [10, 20]$) (b) by family count ($l \in [10, 20]$, $s_f \in [10, 20]$) (c) by setup time ($n_f = 50$, $l \in [10, 20]$)

Fig. 1: Average latency on OpenWhisk system (Y axis) and simulation (X axis). 1 unit is 10ms. Each point corresponds to a single instance executed on both OpenWhisk and simulator.

simultaneously used by other environments. We chose 10ms as the time unit to reduce impact of possible fluctuations of VM or network parameters in the datacenter (we performed some early experiments with 1ms and this noise was significant; and with a longer time unit tests take unreasonable time). We emulate a single instance from our simulator by creating, for each job J_i, an equivalent sequence of invocations in OpenWhisk. To avoid caching of results in OpenWhisk, we ensure that each invocation is executed with a distinct set of parameters. We deployed an OpenWhisk cluster (1 controller and $m = 10$ invokers) on 11 VMs in GCE. All machines have 2 vCPU and 16GB RAM. We further restrict the memory OpenWhisk can use on machines to 1280MB (equivalent to $Q = 10$). In order to reduce impact of cloud storage on system performance, we used a ramdisk to store OpenWhisk accounting database. We also extended limits (maximum duration and sequence length) and changed the default log level to WARN. To reduce the impact of brief performance changes, we executed each test instance thrice and reported the median.

In Figure 1 we compare the average response latency in OpenWhisk and in our simulator varying chain lengths, the number of families and the ranges of setup times. For consistency, OpenWhisk results are rescaled to the simulator time unit (divided by 10) The Pearson correlation between OpenWhisk and simulator is very high (between 0.86 when varying family count, Fig. 1.b, and 0.999 when varying the setup time, Fig. 1.c). There is, however, an additive factor in OpenWhisk noticeable especially in smaller instances in Fig., 1.(a) and Fig. 1.(b): the range of OpenWhisk results in $[5000, 9000]$, while the range of simulated results is in $[550, 1600]$; on larger instances, as in Fig. 1.(c), this constant factor is less noticeable. This additive factor is caused by an additional system overhead added to every function execution: each invocation stores data in a database and requires internal communication. We conclude that the high correlation between the simulator and the OpenWhisk results validates our simulator – that the differences between algorithms observed in the simulator are transferable to the results in OpenWhisk.

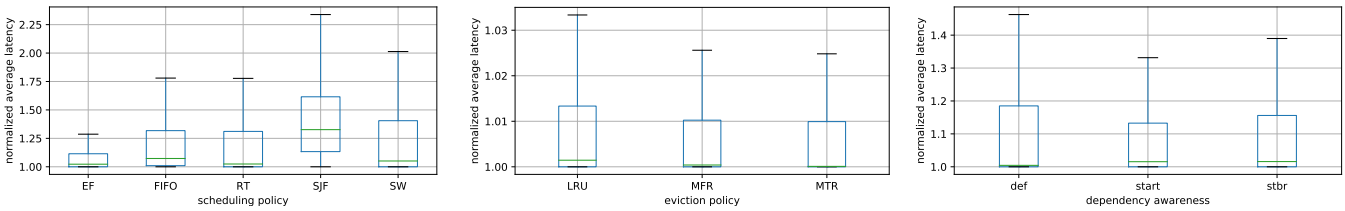
3) Relative Performance of Policies: We first analyze the impact of each policy by analyzing their relative performance. For each variant (A, B, C, D), on each instance, we compute the relative performance of the policy we measure by finding the minimal average latency across all variants of the measured policy while keeping the rest of the variants the same. For example, when measuring the effect of the scheduling policy (A), on an instance, we find the minimum average latency from the 5 variants of the scheduling policy: (EF, b, c, d), (FIFO, b, c, d), (RT, b, c, d), (SFJ, b, c, d), (SW, b, c, d) (keeping b, c, d the same); and then we divide all 5 by this value. The goal of this analysis is to narrow down our focus to the aspects of the problem that are crucial for the performance. Using this method, we show that, e.g., all removal policies result in very similar outcomes. Figure 2 shows the results.

Ordering: *EF* policy dominates other ordering policies, confirming that it is better to avoid environment setup by reusing existing environments. Its median is similar to RT (and lower than other algorithms), and the range of values (including the third quartile) is the lowest. *Removal*: Unlike scheduling policies, all the removal policies result in virtually the same schedule length: the range of Y axis is 1.035; thus outliers are only 3.5% worse than the minimal schedule found in the alternative methods.

Dependency awareness: Both *start* and *stbr* result in similar performance. We confirmed this result by looking at individual instances: the performance of *start* and *stbr* were similar.

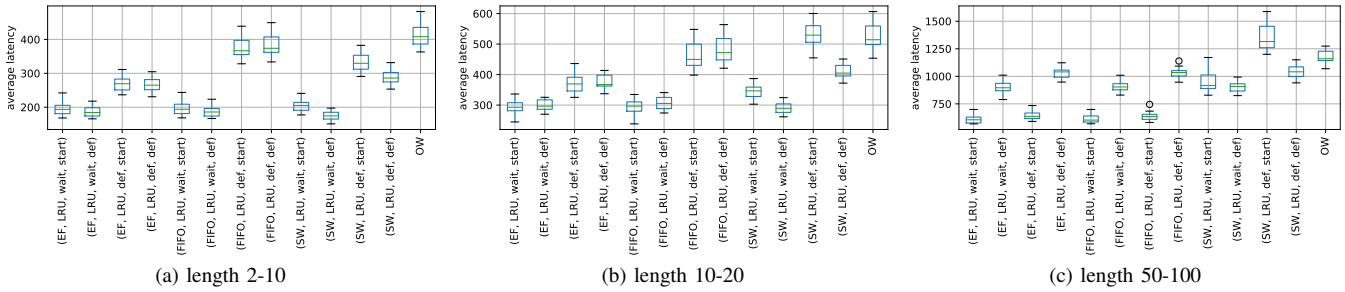
To improve the readability in the remainder, given that the removal policies have little effect on the schedule length (Figure 2), we show only the results for LRU. Similarly, we skip results for SJF and RT orderings: RT is close to FIFO and SJF is clearly dominated by other variants. Finally, as the difference between *start* and *stbr* variants is small, we show results only for *start*.

4) Impact of the length of the chain: In the rest of the experimental section, we analyze the sensitivity of the policies to various parameters of the instance, starting with the average length of the chain. In Figure 3, in all instances $n_f = 50$, $s_f \in [10, 20]$, $m = 20$, $Q = 10$ (results for larger n_f, s_f m and Q are similar; we omit them due to space constraints). All scheduling algorithms using EF as the ordering policy significantly reduce latency compared to the baseline OW (1.3-2.4x), with larger reductions for shorter chains. The *start* dependency-aware variant further reduces latency, especially



(a) (b) (c)

Fig. 2: Comparison of resulting average latency under: different scheduling policies (a), removal policies (b) and variants of dependency–awareness (c). In (a), for each instance (the same tasks, machine capacities and machine count), and having other variants of the algorithm set (removal policy, *waiting* and dependency-awareness), we find the minimal average latency among the 5 scheduling policies; we then normalize the results from all 5 scheduling policies by this minimal average latency. Each box corresponds to a statistics over experiments with all the removal policies (both in *waiting* non-*waiting* variant) and all dependency-awareness variants (def, start, stbr), performed on all instances and all possible machine environments (over 300k individual data points). For (b) and (c) results are normalized as in a), but for different removal policies (b) and for different dependency-aware variants (c), rather than scheduling policies. Here and in all following box plots, the box height indicates the first and the third quartile, the line inside the box indicates the median, and the whiskers extend to the most extreme data point within 1.5 $\times$ IQR.



(a) length 2-10 (b) length 10-20 (c) length 50-100

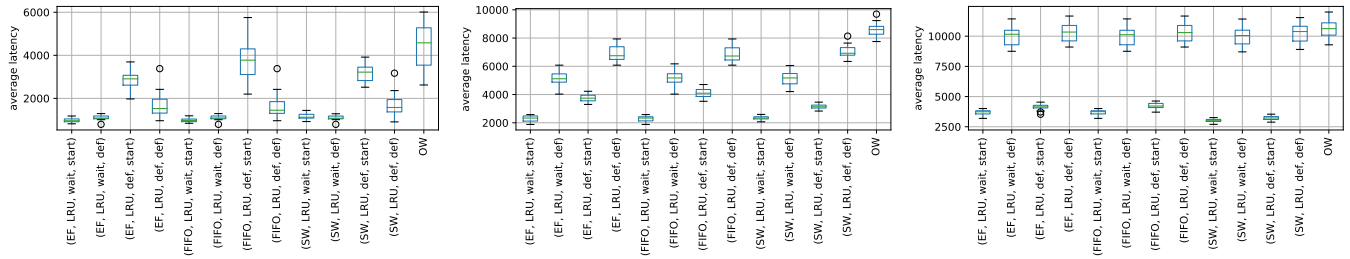
Fig. 3: Influence of the length of the chain. For all instances $n_f = 50$, $m = 20$, $Q = 10$ with setup times 10-20.

for longer chains ($[50 - 100]$), and also for other scheduling methods (FIFO). Therefore, for deployments with long (50 tasks and above) chains, at least 100 families, setup times 100 (and larger) with at least 20 machines of size 10 (or more), implementing dependency-aware scheduler can provide measurable benefits.

5) Impact of the number of families: Figure 4 compares results as a function of the number of task families in the system. When the number of task families is small (up to 20), variants without dependency awareness (*def*) and with *wait* can give better results than dependency-aware variants. In such cases, variants using EF method are slightly better than their equivalents using FIFO. The same applies to the removal method: *wait* variants give better results than their equivalents using plain LRU. The higher the number of families, the higher the probability that the required type of environment is missing. With at least $n_f = 100$ families (Fig. 4.c, similar results for $s_f \geq 100$, $l \geq 50$, $m \geq 20$, $Q \geq 10$ omitted due to space constraints), dependency awareness plays a crucial role – variants using *start* outperforms *def* regardless of the used scheduling algorithm and removal policy. Thus, in case of high variability of functions (i.e. requiring different environments), taking into account tasks' dependencies can significantly reduce the serving latency.

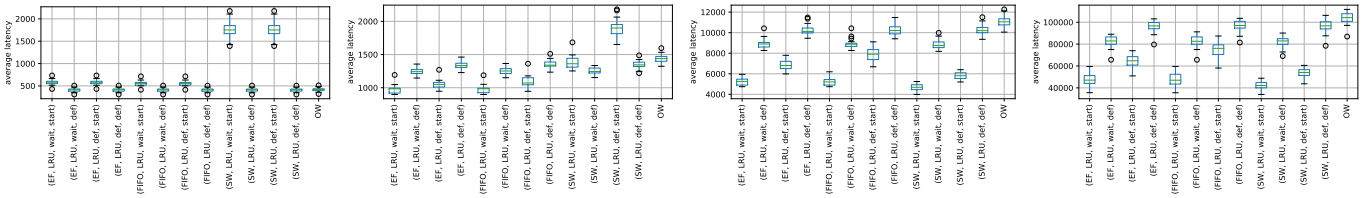
6) Impact of the setup time: Figure 5 compares results as a function of different setup time ranges. In the edge case with no setup times, $s_f = 0$, we see no difference between the *waiting* and the non-*waiting* variants, as there is no additional penalty for inefficient environment re-creation. Similarly, there are no differences between EF and FIFO. For non-zero setup times, dependency awareness (*start*) reduces the latency. However, with no setup time, *start* latencies are *longer*. This behavior is caused by adding tasks with future release time to the queue (see Section III-B4). Consider two jobs each of two tasks: 1) a *long* job with task *A* (duration 10) followed by task *B* (duration 1); 2) a *short* job with task *C* (duration 1), followed by task *B* (same as in *"long"*). *EF* and *FIFO* using *start* variants may assign the second task from the *long* job to the environment of type B immediately after assigning the first task. This might block the second task from the *short* job until $t = 11$; while the optimal schedule starts this task at $t = 1$. For the same reason, *start* has worse results when there are more jobs (i.e. shorter chains) and the systems are smaller (less machines, smaller capacities).

We further investigate for which instance parameters the dependency-aware *start* dominates the myopic *def*, assuming



(a) 10 families (b) 50 families (c) 200 families

Fig. 4: Influence of the different number of families. To show general trend, we present results for 10, 50 and 200 families. For all instances $m = 20$, $Q = 10$, $s_f \in [100, 200]$, $l \in [50, 100]$



(a) setup time 0 (b) setup time 10-20 (c) setup time 100-200 (d) setup time 1000-2000

Fig. 5: Influence of the setup time. For all instances $n_f = 50$, $m = 10$, $Q = 10$, $l \in [50, 100]$.

non-negligible setup times $s_f \geq 100$. We aggregate results by all simulation parameters (count of families n_f, machines m, machine sizes Q, range of chain lengths l, range of setup times s_f and used algorithm variant) and compute the median average latency among 20 instances. Then we analyze in how many of resulting cases changing *def* to *start* improves performance. For long chains ($l \geq 50$), many task families ($n_F > 100$), and many machines ($m \geq 10$), changing the default (*def*) variant to dependency-aware one improves performance in all cases.

7) Impact of machine capacity: Figure 6 compares results as a function of the number of machines and their size. For all instances $n_f = 50$, $l \in [10, 20]$, $s_f \in [10, 20]$. To show general trend and ensure clarity, out of 15 considered machine configurations we present results only for instances with $(m, Q) \in \{(5, 20), (20, 20), (50, 10), (50, 50)\}$. For cases up to $(m, Q) = (5, 20)$, the only observable differences between the plain and dependency-aware variants are for SW scheduling policy. Due to large number of jobs (chain lengths are in range 10-20), when dependent tasks are added to the queue earlier, environments may get blocked as described in Section III-C6, therefore there is no additional benefit of dependency-awareness. For capacities up to $(m, Q) = (50, 10)$, using *wait* variants outperform the default (def) variants using the same scheduling algorithm and with the same setting of dependency-awareness. In all presented cases, for *FIFO* and *EF* scheduling policies, variants using *wait* with *start* have one of the lowest average latency. The improvement on overall system performance is most visible in the case of highly-overloaded machines. Therefore, our methods could be used to improve handling of situation when datacenter has to handle rapid increase (peak) of requests.

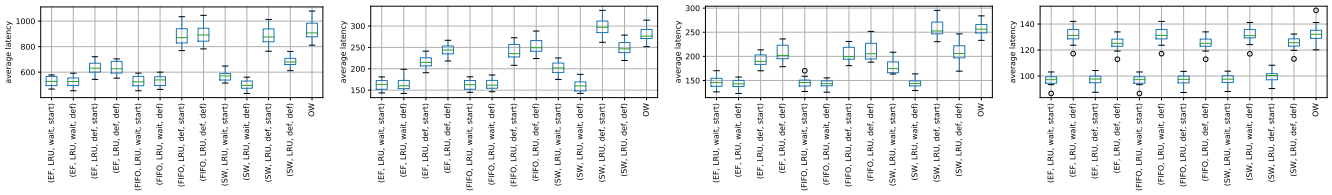
IV. Resource Management

A. Method Description

In this section we present our approach that uses neural networks for predicting the most appropriate resource configuration for the mobile data processing pipelines in order to satisfy certain SLO deadlines and ensure their timely responses. Then we discuss our methodology for resource provisioning in cases that no a priori information exists for these pipelines.

1) Resource estimation: To estimate the most appropriate resource configuration, one simple way is to use grid search, i.e., enumerate all possible combinations of memory, CPU and number of instances, which would provide the optimal solution. However, the cost of enumeration is incredibly high since there is a very large number of combinations that need to be explored, and thus this approach is rather infeasible. In the bibliography, there exist approaches like multi-armed bandits [153] or Bayesian Optimization approaches [154], which are still limited for the setting we consider. Multi-armed bandit approaches require the setup of a set of functions for reward and regret, where, through exploration and exploitation, they will likely derive a sub-optimal configuration. On the same path, Bayesian Optimization approaches, through sampling and minimizing the number of trials to identify a near-optimal solution [155], require also a trial-and-error approach in order to find the solution.

Our intuition comes from a different perspective: Can we exploit pre-existing knowledge from other pipeline runs in order to derive the best configuration for the serverless functions



(a) 5 machines, size 20 (b) 20 machines, size 20 (c) 50 machines, size 10 (d) 50 machines, size 50

Fig. 6: Influence of the machine environment. For all instances $f_n = 50$, $l \in [10, 20]$, $s_f \in [10, 20]$

of a specific pipeline? *Transfer learning* refers to a technique for predictive modeling on a different but somehow similar problem that can then be reused partly or on its entirety to accelerate the training or improve the performance of a model on the problem of interest. This characteristic can be quite beneficial if exploited appropriately.

In TIMBER, we opt for a prediction model approach. More specifically, several works in the literature have utilized neural networks as the appropriate prediction model for different types of inference tasks. Typically, in a neural network setting, an agent learns how to benefit most from making sequential decisions by iteratively propagating information back and forth during the training phase and accumulating knowledge from previous experience. This fundamental characteristic is inline with our intuition that existing knowledge can serve as the appropriate means for enhancing a prediction model.

To this end, in TIMBER, we build a Sequential Neural network model for predicting the appropriate configuration for each one of the serverless functions comprising the mobile data processing pipeline. A Sequential model [156] by its design consists of a plain stack of layers where each layer has exactly one input tensor and one output tensor i.e. it allows us to build a model by stacking layers of nodes (neurons) on top of each other. The Sequential model is the simplest neural network base model in Keras (https://keras.io/). Each argument of the Sequential constructor is a layer of neurons; in this case Dense layers. In dense layers (or densely-connected or fully-connected) all the neurons receive an input from all the neurons present in the previous layer.

Input Layer. Each neuron has an activation function which computes the value that is passed on to the neurons in the next layer. In terms of the layers in TIMBER we choose the ReLU function as an activation function in all layers apart from the last one, which has shown faster convergence times as shown in various works in the bibliography [157], as it requires the estimation of a max value in each neuron rather than the estimation of exponential formulas compared to the sigmoid activation function. More formally, the activation function is described as: $g(h) = h^{+} = max(0, h)$ where h is the input to a neuron. The input layer takes into consideration the container allocation, *i.e.,* the CPUs and memory allocated for each of the functions of the pipeline as well as the request rate of the pipeline. We use the request rate of the pipeline to estimate the pipeline completion time (PCT) to satisfy a certain deadline, as requested from the service provider.

Output Layer. The goal of the neural network is to predict the appropriate number of function replicas for each function of the mobile data processing pipeline required in order to satisfy the specific request rate, as defined by the service provider. In TIMBER, the number of predicted necessary instances that will satisfy the user imposed constraints is translated to a set of labels, where each label refers to one of the serverless functions that consist the pipeline. At the very last step of our prediction algorithm using the neural network, it is required to normalize its output to a probability distribution over predicted output classes. For this purpose, in the last layer of our neural network, we utilize the softmax activation function, based on Luce's choice axiom [158]. The softmax activation function implies that we have different probabilities among the different labels. More formally, this implies that $\sigma(\overrightarrow{\mathrm{q_n}})_n = \frac{e^{q'_n}}{\sum_{b=1}^{K} e^{q'_b}}$, where $\sigma(\overrightarrow{\mathrm{q_n}})_n$ represents the prediction probabilities for each one of the possible labels, K is the number of classes in the multi-class classifier and q_n are the elements of the input vector to the softmax function, and they can take any real value, positive, zero or negative. For example a neural network could have output a vector such as (-0.62, 8.12, 2.53), which is not a valid probability distribution, hence why the softmax would be necessary.

Loss Function. A prerequisite of the model training process is to monitor how well the model's prediction fits the training data. This is denoted by the loss (or cost) function of a model, where the target is to minimize the loss value by adjusting the weights accordingly. In TIMBER, we use cross-entropy [159], a widely used loss function when optimizing classification model. In order to speed up the training process, we opt for using the cross-entropy error instead of the sum-of-squares error function, as well as, to improve the generalization of the model [160]. Another important aspect in our prediction problem setting is that we aim to solve a multiple class classification problem. For this reason, we opt for categorical cross-entropy rather than binary cross-entropy (since we consider each number of instances as a different label). In our experimental evaluation, we show extensively that other metrics such as the KullbackLeibler divergence [161] (which indicates that the two data distributions in question have identical quantities of information) and the Poisson distribution [162] (which is a generalized linear model form of regression analysis used to model count data), are outperformed by the categorical cross-entropy metric.

2) Pipeline Similarity: Reinforcement learning models have shown to be a good fit [163] for learning policies for computer systems, because the model agents are capable of learning from real-world workloads and operating conditions without human-designed inaccurate assumptions and interference. More specifically, the model learns how to benefit most from making sequential decisions by iteratively interacting with the environment and accumulating knowledge from previous experience. In addition to that, regardless the training process overhead required, neural networks feature an important characteristic: neural networks representations may exhibit significant similarities and correspondences between representations in networks trained from different initializations and they can be identified reliably [164]. Similar to earlier works [165], [166], TIMBER assumes that mobile data processing pipelines with similar codebase will have the same behavior for the same size of input, making it an exploitable characteristic to improve the estimation of resource provisioning configurations based on accumulated knowledge of the model from previous experiences, and provide likely similar timely responses.

Estimating the required resources for mobile data processing pipelines with zero *a priori* knowledge. Our intuition is that mobile data processing pipelines for different but somehow similar problems can be partially or entirely reused, as the means to estimate the number of instances for each serverless function consisting a new and probably agnostic mobile data processing pipeline. The main goal is to avoid retraining the neural network model for each different pipeline and overcome any limitations regarding the size of the trained model. Recent works in the literature [165] have exploited the notion of execution plans of distributed VM-based applications, rather than mobile data processing pipelines consisting of serverless functions. On the other hand, we opt for the finer-graind notion of call graphs [167] for estimating the similarity between mobile data processing pipelines. Each mobile data processing pipeline comprises a sequence of serverless function calls. Our goal is to exploit that different pipelines may have similar serverless functions call graphs.

To compute the similarity between the call graphs comprising the serverless applications we need a graph similarity measure. There have been multiple graph similarity metrics but with the most popular to be: *i) the graph edit distance (GED)* [151], [165], *ii) the maximum common subgraph (MCS)* [168] and *iii) the Prefix Preference* [169]. We decided to opt for the Graph Edit Distance metric as it has been accepted as the most appropriate measure for representing the distance between graphs. More specifically, GED defines the similarity between two graphs by the minimum amount of required distortions to transform one graph into the other. Moreover, GED is error-tolerant and can identify similar graphs even in the presence of noise and errors. *Prefix preference* on the other hand, though it has been adopted in Deep Neural Networks training, like in [169], does not evaluate the whole mobile data processing pipeline, and mostly, it focuses on the prefixes of the graphs rather than also the suffixes, as the other two metrics. For this purpose, in our experimental evaluation we report the results for the GED and the MCS metrics. For a new mobile data processing pipeline, for which there is no a priori knowledge, we identify which of the existing pipelines have similar serverless function call graphs. Then, we use the trained neural network to estimate the number of instances of each serverless function consisting the pipeline that will satisfy certain SLO deadlines.

GED Computation. Let us assume two call graphs, CG_1 and CG_2 of the mobile data processing pipelines consisting of serverless functions f_k respectively and that ged_{CG_1,CG_2} is the GED distance between call graph CG_1 and call graph CG_2. The main idea is to match the call graph CG_1 with exactly the call graph of CG_2, compute their GEDs (i.e., the number of necessary distortions to make two call graphs identical). The lower the values of GED, the more similar two call graphs are. The main drawback of computing the GED metric is the fact that its computation has exponential complexity in terms of the graph vertices as the problem of measuring the graph edit distance is NP-hard. For this reason we decided to use a well-known approximation technique [151] that is able to effectively and in polynomial time approximate the GED between two call graphs. The main idea is to transform a graph structure to a multiset of a special data-structure, called star structure, and then compute the distance between these multisets instead of the actual graphs. Comparing multisets reduces the search space of the problem as they do not consider the complete structure of the original graph.

Detecting the most similar pipeline. It is necessary to compute the GED of two call graphs in order to detect whether a mobile data processing pipeline exists for which we have already built a prediction model. The idea is to compute the GED between the newly submitted pipeline and all the pipelines that comprise the set of mobile data processing pipelines HQ with which the neural network has been built with. We find the pipeline that leads to the maximum GED value and then examine whether this value is greater than a pre-defined threshold $\mathcal{T}$ (T is an administrator-defined parameter to control the similarity among two graphs; this can be tuned dynamically based on the degree of similarity we target). If this condition is true we simply return the already built prediction model. In the case that the GED is lower than T then we proceed with the next most similar serverless application in the HQ set. By performing this step, we can estimate the number of instances for each serverless function comprising the pipeline to *prewarm* and set them ready for execution in order to meet the developer SLO deadline, even if there is no prior knowledge about the pipeline's resource needs or performance.

B. Implementation

In this section, we discuss the implementation of our TIMBER framework (the TIMBER architecture is shown in Figure 50). TIMBER comprises the following main components: (a) **Configurator,** utilized by the developers to define their parameters, i.e., service level objectives for the pipelines and upload

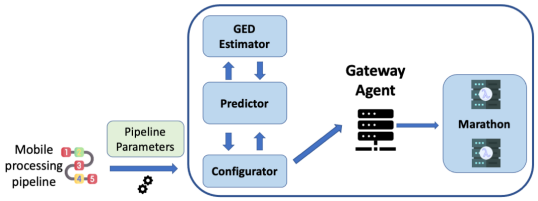


Fig. 7: TIMBER architecture

the source code for the functions comprising the pipeline. These parameters will be utilized by the Predictor Component to specify the number of instances required for each one of the pipeline functions to execute. The functions will be deployed through our container orchestrator. (b) **Graph-Edit Distance Estimator,** computes the GED value between two pipelines, that captures the degree of similarity between the developer's submitted pipeline for execution and previously executed pipelines. (c) **Predictor:** uses the GED value that was determined by the Graph-Edit Distance Estimator Component in order to estimate the number of instances for each one of the pipeline functions to satisfy the SLO constraint. (d) **Gateway Agent.** We developed a Gateway Agent similar to [170] utilized for the deployment of new pipelines as well as to scale up and down deployed serverlesss functions consisting the pipelines. The Agent also acts as a Proxy for function invocations, which are propagated to the deployed function containers. (e) **Mesosphere Marathon:** This is our orchestrator (https://mesosphere.github.io/marathon/) that we utilized to start serverless function containers. The deployed containers run in our Apache Mesos cluster. Apache Mesos abstracts the compute resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to easily be built and run effectively. Each container listens to port 8080 and Mesos maps that port to a random port of the host Agent. Functions are invoked via sending an HTTP POST request to $http://<agentIp>:<mappedPort>$. All functions are generated with the OpenFaas Function build tools and services that provide the ability to the user to deploy and invoke its functions using the underlying container orchestrator. We adapted OpenFaas to be compatible with Apache Mesos and we used Mesosphere Marathon for easier container deployment on Mesos. We also build our functions using the OpenFaas Watchdog Docker containers, which work as follows: when it receives a request for a pipeline invocation it starts another process within the container that runs the function handler and dispatches the received request to the standard input of the forked process. Then it receives the function result from the function standard output and returns it as the response to the received HTTP request that carried the original function invocation call. Since we do not focus solely on serverless functions but on pipelines consisting of a sequence of serverless function invocations, then the sequence of serverless function call invocations is routed through the gateway agent, since existing production serverless infrastructures do not support *direct* communication between serverless functions explicitly (only through a gateway [171]).

Workload	Pipeline Type	Serverless Functions
W1	Processing	Jaes, Linpack, Matmul
W2	Sensor Correlation	PyPearsons, KNN
W3	Clustering	Kmeans

TABLE I: Different types of pipeline workloads

C. Experimental Evaluation

1) Experimental Setup: We conducted our experiments in our local cluster comprising 7 nodes (Intel i7-7700 3.6GHz processors), with a total number of 56 CPUs and 112GB of RAM. This setup allows us to illustrate the benefits of our approach and balance the trade-off among finding the configuration that satisfies the deadline constraints that is also cost-effective. Similar experimental size setups have been utilized in related works [141], [165]. All the nodes are interconnected with 1Gbps Ethernet. All nodes run on Ubuntu 20.04 LTS. We run Apache Mesos 1.9 as our serverless platform and we use Marathon 1.5 in order to deploy Docker containers on top of Mesos. Marathon is used only to start/stop the containers and no other features are utilized. In this cluster, we deployed our TIMBER system. Similar to related works [141], we varied the number of serverless function instances from 5 up to 30. We used OpenFaas Python and Java templates in order to create our functions and pipelines and PyCG [172] to extract the call graphs for the pipelines consisting of serverless functions written in Python.

2) Functions and Workloads: **Functions:** In order to evaluate the performance of our approach, we conducted our experiments using real world pipeline scenarios from state-of-the-art performance benchmarks [173]. The FunctionBench is composed of micro benchmark and pipeline workload; the micro-benchmark uses simple system calls to measure the performance of resources exclusively, and the pipeline-benchmark represents realistic data-oriented pipeline that generally utilize various resources together. In our experimental evaluation we use six realistic serverless functions that consist mobile processing pipelines chosen from both micro-benchmark libraries [173], as well as, developed by us in the context of real-world pipelines as in the works of [165], [174]. Below, we give a brief description for each one of those pipeline functions (**PF**):

PF1 - *Jaes:* Jaes benchmark that performs private key-based encryption and decryption. It is a Java implementation of the AES block-cipher algorithm in CTR mode.

PF2 - *Linpack:* Linpack solves linear equations (Ax = b). We identified that the Linpack serverless function is used by many state of the art works [175] to benchmark CPU performance.

PF3 - *Matmul:* Matmul performs square matrix multiplications. Along with Linpack, these are considered as CPU benchmarks, which are mainly used to measure the CPU-bound performance.

PF4 - *PyPearsons:* PyPearsons is a Python implementation of Pearsons correlation over a smart-city sensor network and

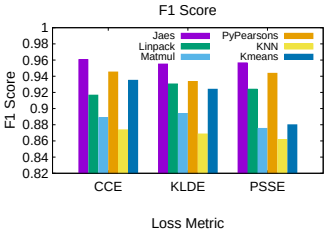


Fig. 8: F1 Score

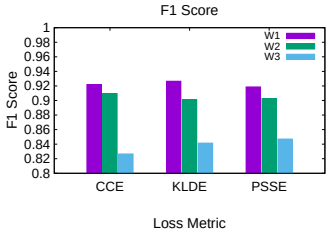


Fig. 9: Average F1 Score

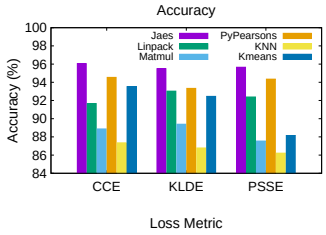


Fig. 10: Accuracy

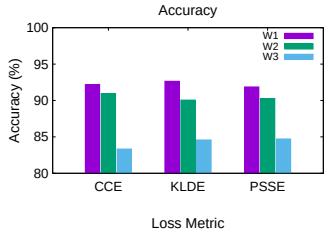


Fig. 11: Average Accuracy

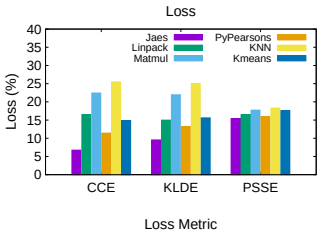


Fig. 12: Loss

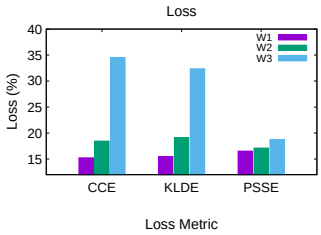


Fig. 13: Average Loss

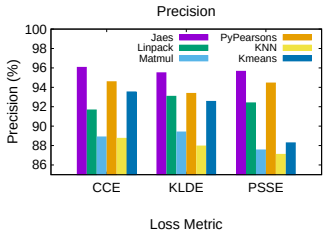


Fig. 14: Precision

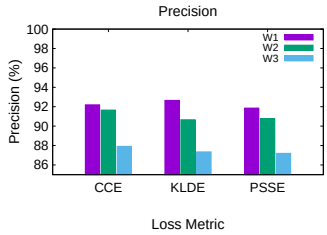


Fig. 15: Average Precision

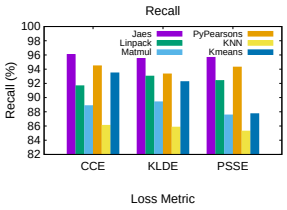


Fig. 16: Recall

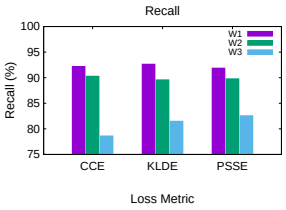


Fig. 17: Average Recall

for a given set of geospatial coordinates it returns a list of the most correlated sensors.

PF5 - *KNN:* KNN is the well known k-nearest neighbor algorithm implemented in Java and for a given set of geospatial coordinates it returns a list of the K nearest sensors from a smart-city sensor network.

PF6 - *Kmeans:* Kmeans function refers to the Kmeans algorithm as implemented in the scikit-learn python library. Kmeans runs on a small batch of data equal to 2Mb and fits the data to the model.

Workloads & Pipelines: Furthermore, we have grouped these functions in the context of different real-world pipeline types and evaluate their performance, as shown in Table I. This allows us to monitor, identify and understand the behaviour of similar pipelines in the context of a serverless environment. Specifically, these workloads, namely **W1**, **W2** and **W3**, refer to three different types of real-world pipelines: Processing, Sensor Correlation and Clustering respectively. Our initial intuition was to show that similar pipelines with those in the workloads examined will also have similar demands in terms of computational resources as well as the same expected throughput for similar input. Once we have trained the neural model with the pipelines from each one of these different workloads, we can then use it to estimate the number of instances for each other function in similar pipelines and their resource needs even for pipelines for which we have possibly zero knowledge on their performance.

3) Prediction Performance: **Dataset** We have evaluated the prediction performance of our neural network utilizing real data from the workloads described in the previous subsection. More specifically, we followed similar guidelines with state-of-the-art works [141], [165] and utilized four different types of memory and CPU configurations, and also varied the number of replica instances from 5 up to 30. We performed 1000 runs using Hey[5], for each one of the possible configurations, and aggregated these results in order to construct the training dataset for the neural network deployed in TIMBER.

Metrics

F1-Score: The F1-Score metric is computed from the precision and recall of the data. The highest possible value of an F-score is 1.0, indicating perfect precision and recall.

Accuracy: The accuracy metric illustrates the frequency with which predicted labels match the true labels.

Precision: The precision metric is the fraction of relevant labels among the retrieved labels, that is the true positive labels over the sum of true positive labels with false positive labels.

Recall: The recall metric illustrates the fraction of relevant labels that were retrieved, that is the true positive labels over the sum of true positive labels and those that were falsely estimated as negative.

Loss: The loss metric refers to the cross-entropy loss (also known as log loss) and captures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label. In TIMBER, we evaluated different types of losses in order to find the appropriate for our problem. These are namely Categorical Cross-Entropy loss [159], KullbackLeibler Divergence Entropy loss [161] and the

[5]https://github.com/rakyll/hey

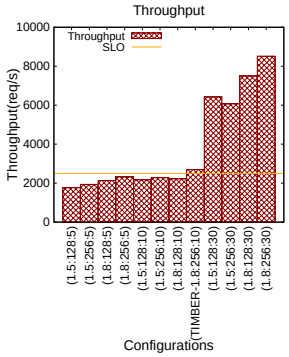


Fig. 18: Matmul throughput

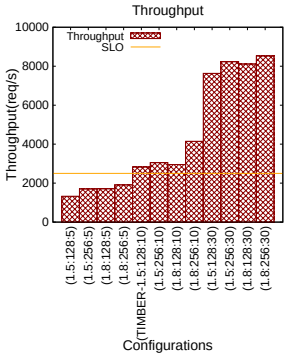


Fig. 19: Linpack throughput

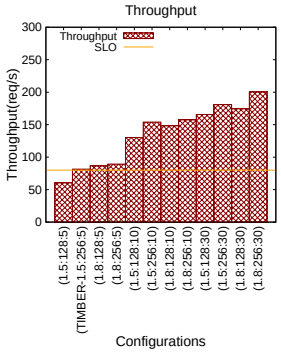


Fig. 20: PyPearsons throughput

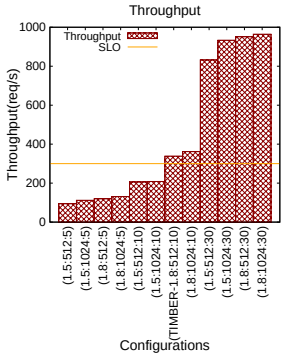


Fig. 21: Kmeans throughput

Poisson Entropy loss [162]. We describe our findings regarding those metrics in the following section.

Findings In Figures 8, 10, 12, 14 and 16, we illustrate the behaviour of each different serverless function consisting the pipelines, where in the x-axis we vary the loss metric utilized for the training of the neural model (CCE stands for Categorical cross-entropy loss, KLDE is the KullbackLeibler Divergence Entropy loss and PSSE is the Poisson Entropy loss). The results show that each different function can achieve very good performance in terms of F1-Score, accuracy, precision and recall, when using the categorical cross entropy as the loss metric required for training. We reason these results due to the fact that KLDE calculates the relative entropy between two probability distributions, whereas cross-entropy can be used to calculate the total entropy between two distributions. That is, CCE can estimate how similar two label distribution are, whereas KLDE how relevant they are. Moreover, we looked into the prediction performance across the different workloads. The results illustrated in Figures 9, 11, 13, 15 and 17 validate our initial intuition of choosing the CCE as the appropriate loss metric for our neural network model training. Despite the fact that the loss value of CCE in the case of workload W3 (Clustering) is higher, the overall prediction performance is best when choosing the CCE as the loss metric required for the neural network model training for these three different kind of real-world workloads.

4) Serverless performance: Please note that due to lack of space, we present the results for the most computationally heavy Python functions (Matmul, Linpack, PyPearsons and Kmeans). **Metrics**

Throughput: Throughput is typically defined as the number of requests/second that are successfully served from the serverless system. That is, in TIMBER, our goal is to identify the appropriate container configuration (in terms of cpu, memory allocation) that will achieve a certain level of throughput, as requested from the pipeline developer.

Cost: We evaluated TIMBER's efficiency in terms of monetary units over the duration of one month. Our goal is to illustrate the ability of TIMBER to identify the best configuration in order to meet the developer's defined SLO deadline, while keeping the cost low and flourish the benefits of using a *pay-as-you-use* model.

Findings In Figures 18, 19, 20 and 21, we draw the throughput achieved by each one of the configurations for a given SLO (equal to 2500 requests / second for matmul and linpack, 80 requests / second for PyPearsons (which is computationally intensive) and 300 requests / second for Kmeans). In the x-axis, we draw all the examined configurations, in y-axis we draw the number of throughput achieved by each one of the examined configuration, and we also annotate the one predicted by TIMBER. We may observe that in every function of each of the three workloads examined (W1, W2 and W3), TIMBER succeeds in estimating the best configuration in order to meet the developer's service level objective. Despite the fact that, there also exist configurations that also succeed in meeting the developer imposed deadline, this may lead to additional provisioning costs, as we discuss next.

In Figures 22, 23, 24 and 25, we draw the corresponding costs, using the pricing scheme from (https://cloud.ibm.com/functions/learn/pricing), for selecting a specific configuration for a pipeline. In the x-axis, we draw all the examined configurations, in the y-axis we draw the cost in monetary units for each one of the examined configurations and we also annotate the cost of the predicted configuration by TIMBER. We may observe that in every function of the three pipelines examined, TIMBER succeeds in predicting and selecting the configuration that not only meets the developer's requirements but is also beneficial in terms of cost (compared to configurations that also succeed in meeting the deadline but require greater number of replicas, thus invoking additional provisioning costs). Compared to choosing naively the greatest number of replicas (i.e. 30) and the largest configuration available, TIMBER can save up to 68.32% for the Matmul serverless app in terms of operations costs *without overprovisioning* (66.84% for linpack, 64.86% for PyPearsons and 64.96% for Kmeans).

GED vs Estimated Performance: In the last set of experiments we evaluated TIMBER's performance with respect to the estimated configuration for pipelines with zero existing knowledge. We developed 5 different agnostic pipelines (**L1** corresponds to a combination of Matmul and Linpack, **L2** corresponds to a slightly modified version of PyPearsons,

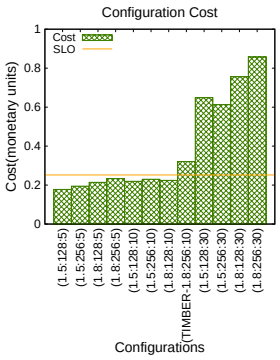


Fig. 22: Matmul Cost over a month

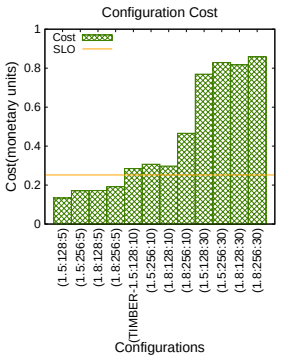


Fig. 23: Linpack Cost over a month

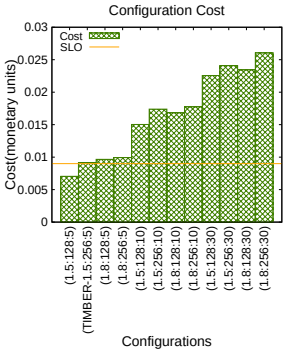


Fig. 24: PyPearsons Cost over a month

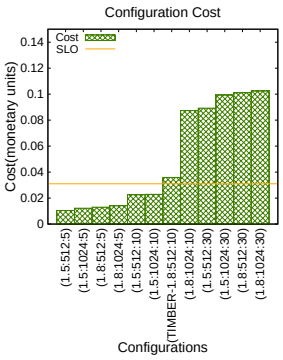


Fig. 25: Kmeans Cost over a month

App	GED	MCS	ps (%)
L1	**GED (PF3,L1)** = 5	**MCS (PF3,L1)** = 1.5	77.81%
L1	**GED (PF2,L1)** = 2	**MCS (PF2,L1)** = 2.0	**94.1%**
L2	**GED (PF4,L2)** = 2	**MCS (PF4,L2)** = 1.25	**96.4%**
L3	**GED (PF4,L3)** = 6	**MCS (PF4,L3)** = 1.0	90%
L4	**GED (PF6,L4)** = 2	**MCS (PF6,L4)** = 1.5	**91.9%**
L5	**GED (PF6,L5)** = 15	**MCS (PF6,L5)** = 1.0	68.1%

TABLE II: Performance vs GED vs MCS

L3 is a moderately modified version of PyPearsons, **L4** is a slightly modified version of Kmeans and **L5** corresponds to an extensively modified version of Kmeans) and computed their throughput using the configuration estimated by TIMBER based on the pipeline they are most similar to. In Table II, we summarize for each one of the agnostic pipelines, the corresponding graph similarities with existing pipelines and the performance similarity ps [176], which is defined as follows: $ps = |1 - \frac{Thr_{Agnostic} - Thr_{SimilarApp}}{Thr_{SimilarApp}}| * 100\%$, where $Thr_{Agnostic}$ is the throughput achieved by the agnostic function and $Thr_{SimilarApp}$ is the estimated throughput by TIMBER for the existing known pipelines. We may safely conclude that as the GED gets closer to 1, the performance similarity increases. Similarly, as the MCS metric increases (that is, the two pipelines share multiple same call graph nodes), the performance similarity also increases, which is inline with our findings using the GED metric. Therefore, TIMBER provides good estimations even for agnostic pipelines.

V. Programming Framework

A. Preliminary and Motivation

1) Why Serverless Inference for LLMs: Numerous companies, including Amazon, Azure, Google, HuggingFace, Together AI [84], Deepinfra [85], Replicate [86], Databricks [87], Fireworks-AI [88], and Cohere [89], have introduced serverless inference services (also known as serverless model entry-points). These services enable users to deploy standard open-source LLMs either in their original form or by modifying them through fine-tuning or by running custom-built models.

Serverless inference can significantly reduce costs for LLM users by charging only for the duration of inference and the volume of processed data. These serverless platforms also offer functionalities such as auto-scaling and auto-failure-recovery to keep instances in an "always-on" state. For the providing companies, serverless inference allows effective multiplexing of models within a GPU cluster, improving resource utilization, and generating a software premium for managing infrastructure on behalf of users.

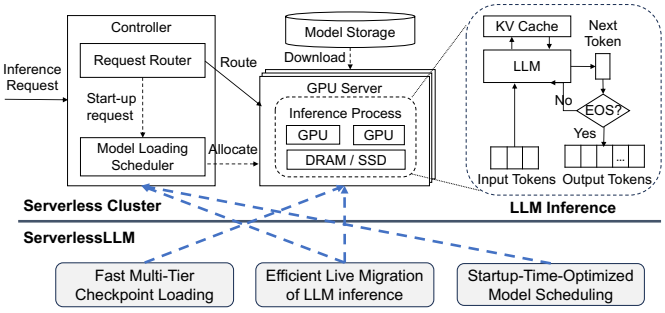


Fig. 26: Overview of GPU serverless clusters, LLM inference and new designs introduced by ServerlessLLM.

Serverless inference systems are especially advantageous for LLM applications with dynamic and unpredictable workloads. These may include newly launched products without clear predictions of user engagement (e.g., the launch of the ChatGPT service) or those facing spontaneous and unpredictable demands, which are typical in sectors such as healthcare, education, legal, and sales. Unlike global-scale LLM services, these applications are activated only when users access the LLM service.

2) Serverless Cluster and LLM Inference: We introduce the key components in GPU serverless clusters in Figure 26. Upon receiving a new inference request, the controller dispatches it to GPU-equipped nodes in a cluster running LLM inference service instances, and to cloud storage hosting model checkpoints. The controller typically consists of two main components: the *request router* and the *model loading scheduler*. The request router directs incoming requests to nodes already running LLM inference processes, or instructs the model loading scheduler to activate LLM inference processes on unallocated GPUs. The selected GPU node initiates a

GPU process/container, setting up an inference library (e.g., HuggingFace Accelerate [90] and vLLM [56]). This inference process involves downloading the requested models checkpoint from a remote model storage and loading it into the GPU, passing through SSD and DRAM.

The LLM inference process often handle requests that include user-specified input prompts, i.e., a list of tokens, as shown in Figure 26. This process iteratively generates tokens based on the prompt and all previously generated tokens, continuing until an end-of-sentence token (denoted as EoS) is produced, resulting in non-deterministic total inference time [57]. During each iteration, the LLM caches intermediate computations in a KV-cache to accelerate subsequent token generation [56], [91]. The tokens generated by each iteration are continuously streamed back to the requesting client, making LLM applications interactive by nature. Their performance is thus measured by both first-token latency (i.e., the time to return the first token) and per-token latency (i.e., the average time to generate a token).

3) Challenges with Serverless LLM Inference: The deployment of LLMs on serverless systems, although promising, often incurs significant latency overheads. This is largely due to the substantial proportions of cold-start in serverless clusters, as demonstrated by public data: the Azure Trace [39] shows that over 40% of functions exhibit a cold-start rate exceeding 25%, and approximately 25% of functions experience a cold-start rate greater than 60%, within a 5-minute keep-alive interval. These figures align with the findings from our experiments, underscoring the impact of cold-starts in real-world settings. Consequently, many serverless providers, including Bloomberg, have publicly acknowledged experiencing extremely high latencies, often reaching tens of seconds, when initializing state-of-the-art LLMs for inference on their platforms.

We observe several primary reasons for the prolonged LLM cold-start latency:

(1) LLM checkpoints are large, prolonging downloads. LLM checkpoints are significantly larger than conventional DNN checkpoints, which leads to longer download times. For instance, Grok-1 [92] checkpoints are over 600 GB, DBRX [93] are 250GB, and Mixtral-8x22B [94] are about 280GB [6] Downloading such large checkpoints from remote storage becomes costly. For example, acquiring an LLM checkpoint with a size of 130GB (e.g., LLaMA-2-70B [7]) from S3 or blob storage takes a minimum of 26 seconds using a fast commodity network capable of 5GB/s [95].

(2) Loading LLM checkpoints incurs a lengthy process. Even when model checkpoints are stored locally on NVMe SSDs, loading these checkpoints into GPUs remains a complex process including model initialization, GPU memory allocation, tensor creation, and tensor data copy, typically taking tens of seconds (as detailed in §V-F2). For instance, loading the OPT-30B model into 4 GPUs requires 34 seconds using PyTorch, and loading LLaMA-2-70B into 8 GPUs takes 84 seconds. This loading latency far exceeds the time required for generating a token during the inference process, which is usually less than 100ms [57]. Consequently, the prolonged first-token latency can significantly disrupt user experience.

4) Existing Solutions and Associated Issues: To improve the latency performance when supporting LLMs, existing solutions show a variety of issues:

(1) Over-subscribing GPUs. The prevalent solutions [24], [96], aimed at circumventing model download and loading times in serverless inference clusters, frequently involve oversubscribing GPUs to accommodate peak demand scenarios. For instance, AWS Serverless Inference [96] maintains a certain number of GPU instances in a warmed state to alleviate the impacts of slow cold starts. While this strategy is effective for managing conventional smaller models, such as ResNet and BERT, it proves challenging for LLMs, which require substantially greater resources from costly GPUs.

(2) Caching checkpoints in host memory. Several solutions [45], [97] have been developed that cache model checkpoints in the host memory of GPU servers to eliminate the need for model downloads. This approach is typically effective for smaller conventional models (e.g., up to a few GBs [45]). However, solely relying on host-memory-based caching proves inadequate for LLMs. LLMs can easily exceed hundreds of GBs in size, challenging the capacity of host memory to store a sufficient number of their checkpoints adequately. The limited size of host memory leads to significant cache misses, resulting in frequent model downloads, as further discussed in §V-F4.

(3) Deploying additional storage servers. Various strategies [95] recommend the deployment of additional storage servers within a local cluster to cache model checkpoints. Despite these enhancements, recent trace studies [95] indicate that model downloads can exceed 20 seconds through an optimized pipeline, even when connected to local commodity storage servers equipped with a 100 Gbps NIC. Although the integration of faster networks (e.g., 200 Gbps Ethernet or InfiniBand) could reduce this latency, the associated costs of implementing additional storage servers and high-bandwidth networks are substantial [98], [99]. For instance, utilizing network-optimized AWS ElasticCache servers [100] to support a 70B model can lead to a 100% increase in costs. Specifically, cache.c7gn.16xlarge servers, which provide 210GB of memory and 200 Gbps of network performance, are priced at $16.3/h, equivalent to the cost of an 8-GPU g5.48xlarge server.

B. Exploiting In-Server Multi-Tier Storage

ServerlessLLM addresses the challenges highlighted in the previous sectionsnamely, high model download times and lengthy model loadingusing a design approach that is cost-effective, scalable, and long-term viable.

1) Design Intuitions: Our design is inspired by the simple observation that GPU servers used for inference feature a multi-tier storage hierarchy with substantial capacity and bandwidth. From a capacity standpoint, these servers are equipped with extensive memory capabilities. For example, a

[6]Model size calculated in float16 precision.

contemporary 8-GPU server can support up to 4 TBs of main memory, 64 TBs on NVMe SSDs, and 192 TBs on SATA SSDs [101]. Additionally, we observe that in the serverless inference context, a significant portion of the host memory and storage devices in GPU servers remains underutilized.

Regarding bandwidth, GPU servers typically house multiple GPUs, each connected to the host memory via a dedicated PCIe connection, providing significant aggregated bandwidth between the memory and GPU. NVMe and SATA SSDs also connect through their respective links and can be configured in RAID to enhance throughput. For instance, an 8-GPU server utilizing PCIe 5.0 technology can achieve an aggregated bandwidth of 512 GB/s between the host memory and GPUs, and around 60 GB/s from NVMe SSDs (RAID 0) to host memory.

Building on these observations, we propose a design approach that leverages the unused in-server multi-tier storage capacity to store models locally and load them more rapidly, thus reducing latency. This approach is (i) *cost-effective*, as it reutilizes existing, underutilized storage resources in GPU servers; (ii) *scalable*, given that the available local storage capacities and bandwidth can naturally increase with the addition of more inference servers; and (iii) *long-term viable*, as upcoming GPU servers will include even greater capacities and bandwidth (*e.g.*, each Grace-Hopper GPU features 1 TB on-chip DRAM and a 900GB/s C2C link between on-chip DRAM and HBM).

2) Design Concerns and Overview: In implementing our design, we identify three crucial concerns that must be addressed.

(1) Support complex multi-tiered storage hierarchy. Current checkpoint and model loading tools such as PyTorch [102], TensorFlow [103], and ONNX Runtime [104] are primarily designed to enhance the training and debugging phases of model development. However, these tools are not optimized for read performance, which becomes critically important in a serverless inference environment. In these settings, model checkpoints are stored once but need to be frequently loaded and accessed across multiple GPUs. This insufficient optimization for read operations results in significant loading delays. While solutions like Safetensors [5] can enhance loading performance, as demonstrated in Section V-F, they still fail to fully leverage the capabilities of a multi-tiered storage hierarchy.

(2) Strong locality-driven inference. Supporting efficient model loading alone is insufficient; we also need approaches that can effectively schedule requests onto GPU servers with locally stored checkpoints. Implementing locality-driven LLM inference, however, presents challenges. Current ML model serving systems such as ClockWork [97] and Shepherd [105] take checkpoint locality into account. Yet, they either depend on accurate predictions of model inference time, which is problematic with LLMs, or they preempt ongoing model inferences, causing significant downtime and redundant computations. Therefore, ServerlessLLM must adopt a new approach that is tailored to the unique characteristics of LLM inference (*i.e.*, this workload is interactive and features long, unpredictable durations), necessitating the support for inference live migration, which is further detailed in Section V-D.

(3) Scheduling models for optimized startup time. ServerlessLLM is designed to minimize the model startup latency. The cluster scheduler (or controller) plays a crucial role in scheduling models onto GPU resources to answer incoming inference requests. However, the scheduler needs to carefully consider the checkpoint's locality in the entire cluster. Many factors may influence the overall startup latency, such as the difference in the bandwidth offered by each layer in the memory hierarchy. There may be instances where it is beneficial to move the current inference execution to a new GPU than to allocate the request to a GPU where the model may have to be loaded from the storage media. Hence, ServerlessLLM needs to accurately estimate the startup times considering the cluster's checkpoint locality status and accordingly allocate resources to minimize startup time.

Overview. ServerlessLLM addresses these concerns with three novel designs, as depicted in Figure 26. Firstly, it facilitates fast multi-tier checkpoint loading (Section V-C) to fully utilize the storage capacity and bandwidth of each GPU server. It also coordinates GPU servers and the cluster controller for efficient live migration of LLM inference (Section V-D), ensuring locality-driven inference with minimal resource overhead and user disruption. Lastly, ServerlessLLM features a startup-time-optimized model scheduling policy (Section V-E) implemented in its controller, effectively analyzing the checkpoint storage status of each server within a cluster, and it chooses a server for initiating a model, minimizing its startup time.

C. Fast Multi-Tier Checkpoint Loading

In this section, we introduce the design of fast multi-tier checkpoint loading in ServerlessLLM, with several key objectives: (i) to fully utilize the bandwidth and capacity of multi-tier local storage on GPU servers, (ii) to ensure predictable loading performance, critical for ServerlessLLM's readiness in low-latency inference clusters, and (iii) to maintain a generic design that supports checkpoints from various deep learning frameworks.

1) Loading-Optimized Checkpoints: Our design is motivated by the observation that LLM checkpoints are often written frequently during training and debugging but loaded infrequently. Conversely, in serverless inference environments, checkpoints are uploaded once and loaded multiple times. This discrepancy has inspired us to convert these checkpoints into a loading-optimized format.

To ensure our design is generic for different frameworks, we operate under a set of assumptions that are common in checkpoints. The checkpoints have: (i) *Model execution files* which define the model architecture. Depending on the framework, the format varies; TensorFlow typically uses protobuf files [106], while PyTorch employs Python scripts [107]. Beyond architecture, these files detail the size and shape of each tensor and include a model parallelism plan. This plan

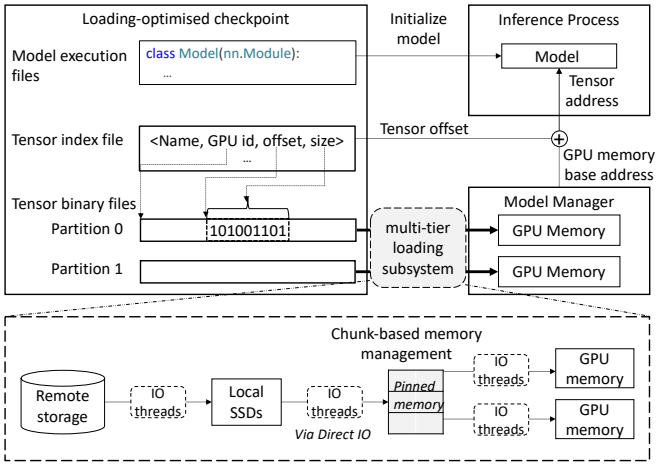


Fig. 27: Components in fast multi-tier checkpoint loading.

specifies the target GPU for each tensor during checkpoint loading. (ii) *Model parameter files* which stores the binary data of parameters in an LLM. Tensors within these files can be arranged in any sequence. Runtimes such as PyTorch may also store tensor shapes as indices to calculate the offset and size for each tensor.

To ensure fast loading performance, we implement two main features for the converted checkpoints: (i) *Sequential chunk-based reading*: To ensure efficient sequential reading, tensors for each GPU are grouped in partitions (shown in Figure 27). These files contain only the binary data of model parameters and exclude metadata such as tensor shapes, facilitating large chunk reading. (ii) *Direct tensor addressing*: We create a tensor index file (shown in Figure 27) that maps tensor names to a tuple of GPU id, offset, and size, facilitating the efficient restoration of tensors. The tensors are aligned with memory word sizes, facilitating direct computation of memory address.

We observe that decoupling the loading and inference processes can further enhance loading performance. This separation allows checkpoint loading to be pre-scheduled and overlapped with the initialization of the inference process. For this, ServerlessLLM uses a model manager to load tensor data, while allowing the inference process to focus on initializing the model by setting the data pointers for each tensor. More specifically, the model manager allocates memory on GPUs and loads the binary data of the checkpoint via a fast multi-tier loading subsystem (see details in V-C2). The inference process initializes the model object and sets the GPU memory address for each tensor. It acquires the base addresses for each GPU (i.e., CUDA IPC handles) from the model manager and reads the tensor offset from the tensor index file, facilitating the computation of the tensor GPU memory address (i.e., *base + offset*). To ensure the model is fully initialized before inference, the inference process and the model manager perform a synchronization.

2) Multi-Tier Loading Subsystem: To achieve fast and predictable checkpoint loading performance, we design a multi-tier loading subsystem, integrated within the model manager. This subsystem incorporates several techniques:

Chunk-based data management. For fast loading performance, we have implemented chunk-based data management with three main features: (i) *Utilizing parallel PCIe links*. To mitigate the bottleneck caused by a single PCIe link from storage when loading multiple models into GPUs, we employ parallel DRAM-to-GPU PCIe links to facilitate concurrent checkpoint loading across GPUs. (ii) *Supporting application-specific controls*. Our memory pool surpasses simple caching by providing APIs for the allocation and deallocation of memory. This enables fine-grained management of cached or evicted data chunks, based on specific requirements of the application. (iii) *Mitigating memory fragmentation*. We address latency and space inefficiencies caused by memory fragmentation by using fixed-size memory chunks.

Predictable data path. We have created an efficient data path in our model manager with two main strategies: (i) *Exploiting direct file access*. We use direct file access (e.g., 'O_DIRECT' in Linux) to avoid excessive data copying by directly reading data into user space. This method outperforms memory-mapped files (mmap), currently adopted in high-speed loaders such as Safetensors [5], which rely on system cache and lack consistent performance guarantees (critical for predictable performance). (ii) *Exploiting pinned memory*. We utilize pinned memory to eliminate redundant data copying between DRAM and GPU. This approach allows direct copying to the GPU with minimal CPU involvement, ensuring efficient use of PCIe bandwidth with a single thread.

Multi-tier loading pipeline. We have developed a multi-tier loading pipeline to support various storage interfaces and improve loading throughput. This pipeline has three features: (i) *Support for multiple storage interfaces*. ServerlessLLM offers dedicated function calls for various storage interfaces, including local storage (e.g., NVMe, SATA), remote storage (e.g., S3 object store [108]), and in-memory storage (pinned memory). It utilizes appropriate methods for efficient data access in each case. (ii) *Support for intra-tier concurrency*. To leverage modern storage devices' high concurrency, ServerlessLLM employs multiple I/O threads for reading data within each storage tier, improving bandwidth utilization. (iii) *Flexible pipeline structure*. We use a flexible task queue-based pipeline design, supporting new storage tiers to be efficiently integrated. I/O threads read storage chunks and enqueue their indices (offset and size) for the I/O threads in the next tier.

D. Efficient Live Migration of LLM Inference

In this section, we describe why live migration is the key to effective locality-driven LLM inference, and how to make such a live migration process particularly efficient.

1) Need for Live Migration: We consider a simple example to analyze the performance of different current approaches in supporting the checkpoint locality. In this example, we have two servers (named Server 1 and Server 2) and two models (named Model A and Model B), as illustrated in Figure 28. Server 1 currently has Model A in DRAM and Model B in

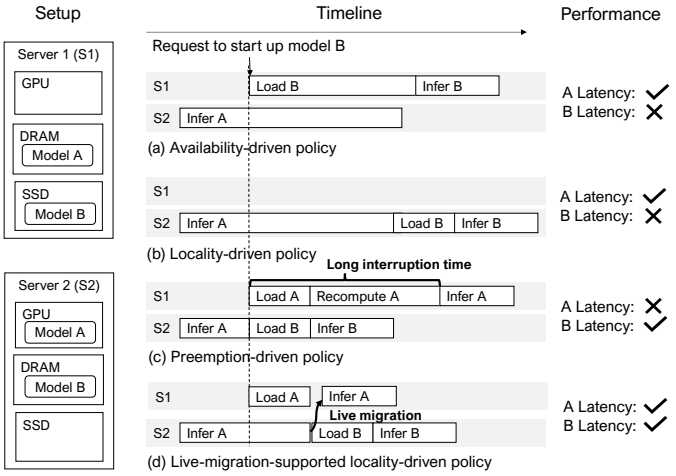


Fig. 28: Analysis of different locality-driven policies

SSD and its GPU is idle, while Server 2 currently has Model B in DRAM, and its GPU is running the inference of Model A.

In Figure 28, we analyze the performance of potential policies for starting up Model B. Our analysis is based on their impact on the latency performance of both Model A and B:

- *Availability-driven policy* chooses Server 1 currently with an available GPU, and it is agnostic to the location of Model B. As a result, the Model B's startup latency suffers while the Model A remains unaffected.
- *Locality-driven policy* opts for the locality in choosing the server and thus launching Model B on Server 2. However, it waits for Model A to complete, making Model B suffer from a long queuing delay. Furthermore, the locality policy leaves Server 1 under-utilized, preventing all servers from being fully utilized.
- *Preemption-driven policy* preempts Model A on Server 2 and startups Model B. It identifies that Server 1 is free and reinitiates Model A there. This policy reduces Model B's latency but results in significant downtime for Model A when it performs reloading and recomputation.
- *Live-migration-supported locality-driven policy* prioritizes locality without disrupting Model A. It initially preloads Model A on Server 1, maintaining inference operations. When Model A is set on Server 1, its intermediate state is transferred there, continuing the inference seamlessly. Following this, Model B commences on Server 2, taking advantage of locality. This policy optimizes latency for both Models A and B.

According to the examples above, live migration stands out in improving latency for both Model A and Model B among all locality-driven policies.

2) Making Live Migration Efficient: We aim to achieve efficient live migration of LLM inference, incurring minimal resource overhead and minimal user interruption. We initially considered using the snapshot method from Singularity [109], which involves snapshotting the LLM inference. However, this method is slow due to lengthy snapshot creation and transfer times (e.g., typically 10s seconds or even minutes). Dirty-page-based migration might be considered to accelerate virtual machine migration, but this approach is currently not supported in GPU-enabled containers and virtual machines. Hence, we decided to explore live migration methods that can be easily implemented in applications.

To make the live migration method effective for LLM inference, we aim to achieve two objectives: (i) the migrated inference state must be minimal to reduce network traffic, and (ii) the destination server must quickly synchronize with the source server's progress to minimize migration times.

For (i), we propose to migrate tokens (typically 10-100s KB) instead of the large KV-Cache (typically 1-10s GB), as recomputing the KV-Cache based on the migrated tokens on the destination GPU is generally much faster than transferring the dirty state over the network. In certain conditions (e.g., given high-bandwidth network and short input sequences), migrating KV-Cache might also be fast yet it still increases cluster network traffic compared to migrating tokens.

For (ii), we leverage an insight from LLM inference: recomputing the KV-Cache for current tokens on the destination GPU is significantly faster (usually an order of magnitude shorter) than generating an equivalent number of new tokens on the source GPU. This approach facilitates efficient convergence of multi-round token-based migration, with the quantity of tokens generated on the source diminishing with each round. For example, time to recompute the KV-Cache for 1000 tokens equals to the time to generate about 100 new tokens according to [110].

3) Multi-Round Live Migration Process: We implement the above proposal as a multi-round live migration process. In each migration round (step ❸, ❹ and ❺), the destination server (referred to as the *dest* server) recomputes the KV cache using the intermediate tokens sent by the source server (referred to as the *src* server). When the gap (i.e., the tokens generated after the last round) between the source server and the destination server is close enough, the *src* server stops generating and sends all tokens to the *dest* via the request router, ensuring minimal interruption on ongoing inference during migration. This migration process is depicted in Figure 29 with its steps defined below:

1) The model loading scheduler sends a model loading request to *dest* server to load model A into GPUs. If there is an idle instance of model A on *dest* server, the scheduler skips this step.
2) After loading, the scheduler sends a migration request carrying the address of *dest* server to *src* server.
3) Upon receiving a migrate request, *src* server sets itself as "migrating", sends a resume request with intermediate tokens (i.e., input tokens and the output tokens produced before step 3) to *dest* server if the inference is not completed. Otherwise, it immediately returns to the scheduler.
4) *dest* server recomputes KV cache given the tokens in the

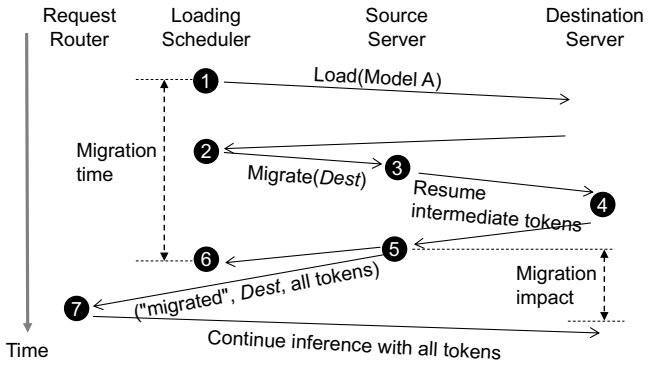


Fig. 29: Live migration process for LLM inference

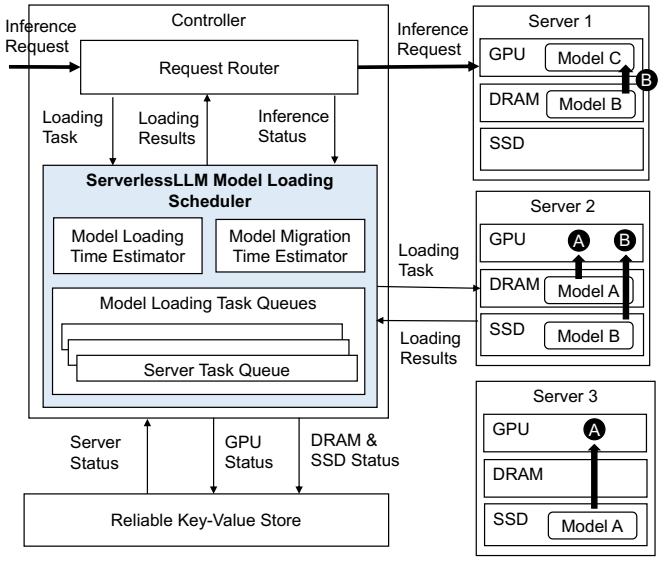


Fig. 30: Overview of the model loading scheduler design

resume request.

5) Once *resume* request is done, *src* server stops inference, returns to the scheduler, and replies to the request router with all tokens (i.e., the intermediate tokens together with the remaining tokens produced between step 3 and step 5) and a flag "migrated".
6) The scheduler finishes the migration, unloads model A at *src* server and starts loading model B.
7) The request router checks the flag in the inference response. If it is "migrated", the request router replaces *src* server with *dest* server in its route table and sends all tokens to *dest* server to continue inference.

4) Practical Concerns: **Handling inference completion.** The autoregressive nature of LLM inference may lead to task completion at *src* server between steps ❸ and ❺. In such cases, *src* server informs the request router of the inference completion as usual. Additionally, it notifies the loading scheduler, which then instructs *dest* server to cease resuming, terminating the migration.

Handling server failures. ServerlessLLM can manage server failures during LLM inference migration. In scenarios where *src* server fails, if the failure happens during loading (i.e., before step ❷ in Figure 29), the scheduler aborts the migration and unloads the model from the destination. If the failure occurs during migration (i.e., between steps ❷ and ❸), the scheduler directs the destination to clear any resumed KV cache and unload the model.

In cases where *dest* server fails, if the failure takes place during loading, the migration is canceled by the scheduler. Should the failure occur while resuming, the source notifies the scheduler of the failure and continues with the inference.

E. Startup-Time-Optimized Model Scheduling

In this section, we describe the design of the startup-time-optimized model scheduling implemented in ServerlessLLM's cluster scheduler (denoted as controller), as shown in Figure 30. This scheduler processes loading tasks from the request router and employs two key components: a model loading time estimator and a model migration time estimator. The former assesses loading times from various storage media, while the latter estimates times for necessary model migrations. For example, as shown in Figure 30, the scheduler calculates the time to load Model A (indicated by Ⓐ) from different servers' DRAM and SSD, aiding in server selection. Similarly, for Model B (Ⓑ), it assesses whether to migrate Model C to another server or load Model B from Server 2's SSD.

To ensure robust time estimation, the ServerlessLLM scheduler employs distinct loading task queues for each server, effectively mitigating the impact of contentions caused by concurrent loading activities. Upon assigning a task, it promptly updates the server statusincluding GPU and DRAM/SSD statesin a reliable key-value store (e.g., , etcd [111] and ZooKeeper [112]). This mechanism enables ServerlessLLM to maintain continuity and recover efficiently from failures.

1) Estimating Model Loading Time: To estimate the time needed to load models from different storage tiers, we consider three primary factors: (i) *queuing time* (q), which is the wait time for a model in the server's loading task queue. This occurs when other models are pending load on the same server; (ii) *model size* (n), the size of the model in bytes, or its model partition in multi-GPU inference scenarios; (iii) *bandwidth* (b), the available speed for transferring the model from storage to GPUs. ServerlessLLM tracks bandwidth for network, SSD, and DRAM, allowing us to calculate loading time as $q + n/b$. Here, q accumulates from previous estimations for the models already in the queue.

For precise estimations, we have implemented: (i) Sequential model loading per server, with single I/O queues for both Remote-SSD and SSD-DRAM paths (since these paths are shared by multiple GPUs on a server), reducing bandwidth contention which complicates estimation; (ii) In multi-tier storage, ServerlessLLM uses the slowest bandwidth for estimation because of ServerlessLLM's pipeline loading design. For example, when SSD and DRAM are both involved,

SSD bandwidth is the critical bottleneck since it is orders of magnitude slower than DRAM; (iii) The scheduler monitors the loading latency returned by the servers. It leverages the monitoring metrics to continuously improve its estimation of the bandwidth through different storage media.

2) Estimating Model Migration Time: For live migration time estimation, our focus is on model resuming time (as shown in step ❹ in Figure 29), as this is significantly slower (seconds) than token transfer over the network (milliseconds). We calculate model resuming time considering: (i) *input tokens* (t_{in}), the number of tokens in the LLM's input prompt; (ii) *output tokens* (t_{out}), the tokens generated so far; and (iii) *model-specific parameters* (a and b), which vary with each LLM's batch sizes and other factors, based on LLM system studies like vLLM [56]. With all the above factors, we can compute the model resuming time as $a \times (t_{in} + t_{out}) + b$.

However, obtaining real-time output tokens from servers for the scheduler can lead to bottlenecks due to excessive server interactions. To circumvent this, we developed a method where the scheduler queries the local request router for the inference status of a model, as illustrated in Figure 30. With the inference duration (d) and the average time to produce a token (t), we calculate $t_{out} = d/t$.

For selecting the optimal server for model migration, ServerlessLLM employs a dynamic programming approach to minimize migration time.

3) Practical Concerns: **Selecting best servers.** Utilizing our time estimations, ServerlessLLM evaluates all servers for loading the forthcoming model, selecting the one offering the lowest estimated startup time. The selection includes the server ID and GPU slots to assign. If no GPUs are available, even after considering migration, the loading task is held pending and retried once the request router informs the scheduler to release GPUs.

Handling scheduler failures. ServerlessLLM is built to withstand failures, utilizing a reliable key-value store to track server statuses. On receiving a server loading task, its GPU status is promptly updated in this store. Post server's confirmation of task completion, the scheduler updates the server's storage status in the store. Once recorded, the scheduler notifies the request router of the completion, enabling request routing to the server. In the event of a scheduler failure, recovery involves retrieving the latest server status from the key-value store and synchronizing it across all servers.

Scaling schedulers. The performance of the loading scheduler has been significantly enhanced by implementing asynchronous operations for server status reads, writes, and estimations. Current benchmarks demonstrate its capability to handle thousands of loading tasks per second on a standard server. Plans for its distributed scaling are earmarked for future development.

Resource fairness. ServerlessLLM treats all models with equal importance and it ensures migrations do not impact latency. While we currently adopt sequential model loading on the I/O path, exploring concurrent loading on servers with a fairness guarantee is planned for future work.

Estimator accuracy. Our estimator can continuously improve their estimation based on the monitored loading metrics returned by the servers. They offer sufficient accuracy for server selection, as shown in Section V-F.

F. Evaluation

This section offers a comprehensive evaluation of ServerlessLLM, covering three key aspects: (i) assessing the performance of our loading-optimized checkpoints and model manager, (ii) examining the efficiency and overheads associated with live migration for LLM inference, and (iii) evaluating ServerlessLLM against a large-scale serverless workload, modelled on real-world serverless trace data.

1) Evaluation Setup: **Setup.** We have two test beds: (i) a GPU server has 8 NVIDIA A5000 GPUs, 1TB DDR4 memory and 2 AMD EPYC 7453 CPUs, two PCIe 4.0-capable NVMe 4TB SSDs (in RAID 0) and two SATA 3.0 4TB SSDs (in RAID 0). This server is connected to a storage server via 1 Gbps networks on which we have deployed MinIO, an S3 compatible object store; (ii) a GPU cluster with 4 servers connected with 10 Gbps Ethernet connections. Each server has 4 A40 GPUs, 512 GB DDR4 memory, 2 Intel Xeon Silver 4314 CPUs and one PCIe 4.0 NVMe 2TB SSD.

Models. We use state-of-the-art LLMs, including OPT [6], LLaMA-2 [7] and Falcon [8] in different sizes. For cluster evaluation (§V-F3 and §V-F4) on test bed (ii), following prior work [55], we replicate OPT-6.7B/OPT-13B/OPT-30B models for 32/16/8 instances respectively (unless otherwise indicated) that are treated as different models during evaluation.

Datasets. We use real-world LLM datasets as the input to models. This includes GSM8K [113] that contains problems created by human problem writers, and ShareGPT [114] that contains multilanguage chat from GPT4. Since the models we used can handle at most 2048 context lengths, we truncate the input number of tokens to the max length. We also randomly sample 4K samples from each dataset to create a mixed workload, emulating real-world inference workloads.

Workloads. Since there are no publicly available LLM serverless inference workloads, we use Azure Serverless Trace [39] which is a representative serverless workload used in recent serverless studies [115] and model-serving studies [55], [105]. We designate functions to models and creates bursty request traces (CV=8 using Gamma distribution), following the workload generation method used in AlpaServe [55]. We then scale this trace to the desired requests per second (RPS). For cluster evaluation, we replicate each model based on its popularity and distribute them across nodes' SSDs using round-robin placement until the total cluster-wide storage limit is reached. Optimization of checkpoint placement is considered a separate issue and is not addressed in this paper. For all experiments (unless we indicate otherwise), we report the model startup latency, a critical metric for serverless inference scenarios.

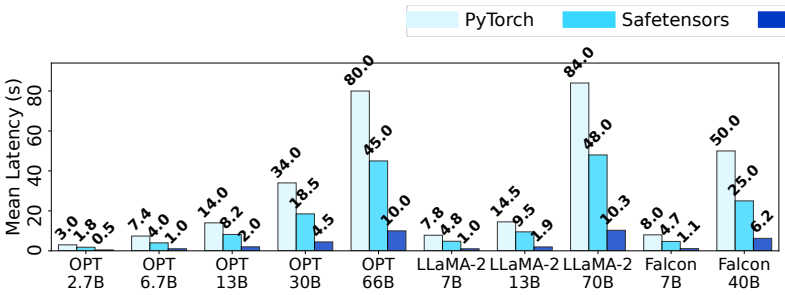


Fig. 31: Checkpoint loading speed.

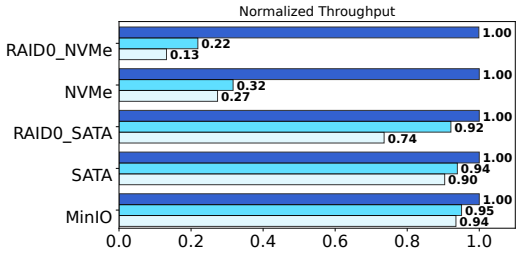


Fig. 32: Normalized bandwidth utilization.

When migration or preemption is enabled, this latency is added with pause latency, accounting for the impacts of delays.

2) ServerlessLLM Checkpoint Loading: We now evaluate the model manager's effectiveness in reducing the model loading latency. For our experiments, we test the checkpoint read on test bed (i). We record reads from 20 copies of each model checkpoint to get a statistically significant performance report. We clear the page and inode caches after checkpoint copies are made to ensure a cold start. For each type of model, we randomly access the 20 copies to simulate real-world access patterns.

Loading performance. We aim to quantify the performance gains achieved by the ServerlessLLM checkpoint manager. We compare PyTorch [102] and Safetensors [5], representing the read-by-tensor checkpoint loading and mmap-based checkpoint loading, respectively. We use all types of models with all checkpoints in FP16 and run the test on RAID0-NVMe SSD having a throughput of 12 GB/s.

Figure 31 shows the performance comparison in terms of mean latency for all the models[7]. We observe that ServerlessLLM is 6X and 3.6X faster than PyTorch and Safetensors, respectively, for our smallest model (OPT-2.7B). We observe similar results with the largest model (LLaMA-2-70B) where ServerlessLLM is faster than PyTorch and Safetensors by 8.2X and 4.7X respectively. Safetensors is slower than ServerlessLLM due to a lot of page faults (112K for LLaMA-2-7B) on cold start. In contrast, ServerlessLLM's checkpoint manager leverages direct I/O and realizes chunk-based parallel loading, all contributing to the significant improvement in loading throughput. PyTorch is about 2X slower than Safetensors in our evaluation, consistent with the results in a public benchmark reported by Safetensors. The primary reason is that PyTorch first copies data into host memory and then into GPU memory.

Furthermore, we observe that the loading performance of ServerlessLLM is agnostic to the type of the model. For example, the performance of both OPT-13B and LLaMA-2-13B is similar signifying the fact that the performance is only dependent on the checkpoint size.

Loading performance with LoRA adapters. ServerlessLLM also supports loading LoRA adapters [9] in PEFT format. For an adapter (rank=32, size=1GB) of LLaMA-70B model, ServerlessLLM achieves 83.5ms loading latency which is 4.4X faster than Safetensors whose loading latency is 370ms. This demonstrates ServerlessLLM's loader design efficiency in small checkpoint loading.

Harness full bandwidth of the storage devices. We now move to understand if ServerlessLLM can utilize the entire bandwidth that a storage medium offers to achieve low latency. We use the same setup as described above. We choose LLaMA-2-7B to represent the SOTA LLM model. We use FIO with the configuration of asynchronous 4M direct sequential read with the depth of 32 as the optimal baseline and optimized throughput using the result in all storage media. We test various settings of FIO to make sure the configuration chosen has the highest bandwidth on each storage media. For object storage over the network, we use the official MinIO benchmark to get the maximum throughput.

Figure 32 shows the bandwidth utilization across different storage devices, normalized relative to the measurements obtained using FIO and MinIO. The storage device from bottom to top is ascending in maximum bandwidth. We observe that ServerlessLLM's model manager is capable of harnessing different storage mediums and saturating their entire bandwidth to get maximum performance. Interestingly, we observe that ServerlessLLM is well suited for faster storage devices such as RAID0-NVMe compared to Pytorch and Safetensors. It shows that existing mechanisms are not adaptive to newer and faster storage technology. Despite the loading process passing through the entire memory hierarchy, ServerlessLLM is capable of saturating the bandwidth highlighting the effectiveness of pipelining the loading process.

Performance breakdown. We now move to highlight how each optimization within the model manager contributes towards the overall performance. We run an experiment using RAID0-NVMe with various OPT models. We start from the basic implementation (ReadByTensor) and incrementally add optimizations until the Pipeline implementation. Figure 33 shows the performance breakdown for each model. We observe similar contributions by different optimizations for all the models despite having different checkpoint size.

Bulk reading improves 1.2x throughput, mitigating the

[7]The number after the model name represents the number of parameters in the figure and B stands for Billion.

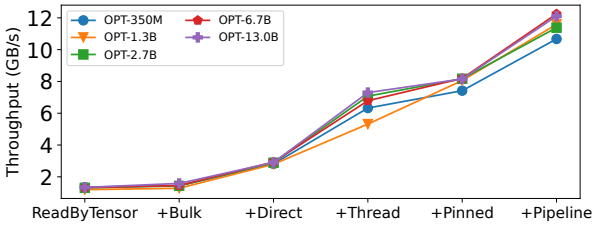


Fig. 33: Performance breakdown of checkpoint loaders.

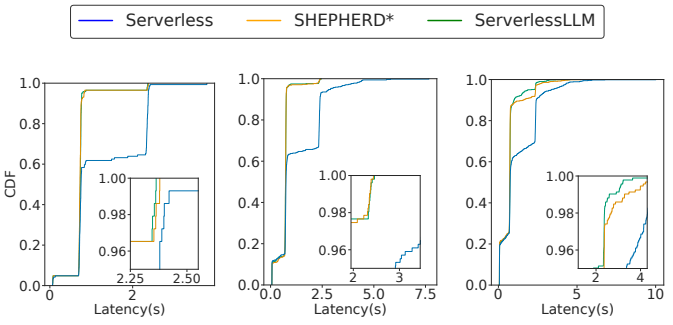


Fig. 34: GSM8K, RPS=0.2

Fig. 35: GSM8K, RPS=0.8

Fig. 36: GSM8K, RPS=1.4

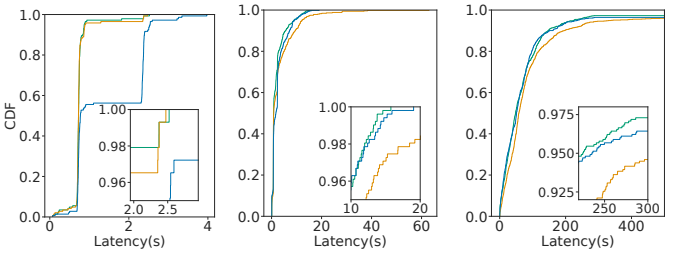


Fig. 37: ShareGPT, RPS=0.2

Fig. 38: ShareGPT, RPS=0.8

Fig. 39: ShareGPT, RPS=1.4

throughput degradation from reading small tensors one after another (on average one-third of the tensors in the model are less than 1MB). Direct IO improves 2.1x throughput, bypassing cache and data copy in the kernel. Multi-thread improves 2.3x throughput, as multiple channels within the SSD can be concurrently accessed. Pinned memory provides a further 1.4x throughput, bypassing the CPU with GPU DMA. Pipeline provides a final 1.5x improvement in throughput, helping to avoid synchronization for all data on each storage tier.

We run ServerlessLLM in a container to limit the CPU cores it can use. We find that with 4 CPU cores, ServerlessLLM can achieve maximum bandwidth utilization. We set a sufficiently large chunk size in bulk reading (16MB) to involve less number of reads and also pinned memory-based chunk pool does not need extra CPU cycles for data copy.

3) ServerlessLLM Model Scheduler: In this section, we evaluate the performance of the ServerlessLLM's cluster scheduler on test bed (ii). We compare ServerlessLLM against two schedulers – the de-facto serverless scheduler and Shepherd [105] scheduler. The serverless scheduler randomly chooses any GPU available and does not comprise any optimization for loading time. We implement Shepherd scheduler and use ServerlessLLM's loading time estimation strategy to identify the correct GPU. We call the modified scheduler as Shepherd*. Therefore, in principle, Shepherd* and ServerlessLLM will choose the same GPU. However, Shepherd* will continue to rely on preemption, while ServerlessLLM will rely on live migration to ensure lower latency times.

Figure 34 shows the result of a scenario where we run all three schedulers against OPT-6.7B model and GSM8K and ShareGPT dataset while increasing the requests per second. ShareGPT dataset's average inference time is 3.7X longer than GSM8K. Figure 34 and Figure 37 show the case where there is no locality contention for both datasets. The serverless scheduler cannot take advantage of locality-aware scheduling unlike ServerlessLLM and Shepherd* leading to longer latency. For 40% of the time, the model is loaded from SSD due to random allocation of the GPUs. As there is no migration or preemption, the performance of Shepherd and ServerlessLLM is similar.

When the schedulers are subjected to medium requests per second, for GSM8K (Figure 35, without locality-aware scheduling, the loading times start causing queueing latency leading with Serverless scheduler resulting in increasing the P99 latency by 1.86X. As there is no migration or preemption, the performance of Shepherd and ServerlessLLM is similar. With a longer inference time with ShareGPT (Figure 38, we even observe 2X higher P99 latency with Shepherd* compared to ServerlessLLM due to preemption. As ServerlessLLM relies on live migration in case of locality contention, ServerlessLLM performs better than the other schedulers despite the number of migrations is higher (114 out of 513 total requests) than the number of preemptions (40 out of 513 total requests).

On further stressing the system by increasing the requests per second to 1.4, for GSM8K, one can clearly observe the impact of live migration and preemption. ServerlessLLM outperforms Shepherd* and Serverless schedulers by 1.27X and 1.95X on P99 latency respectively. There are 9 preemptions and 53 migrations respectively for a total of 925 requests. As discussed in Section V-D1, preemptions lead to longer latency compared to migrations. We also observe that with Shepherd*, model checkpoints are read from SSD 2X times more than with ServerlessLLM. With ShareGPT (figure 39, we observe that the GPU occupancy reaches 100% leading to requests timeouts with all the three schedulers[8]. Shepherd behaves the worst compared to Serverless and ServerlessLLM schedulers, i.e., 1.43X and 1.5X higher P95 latency respectively. ServerlessLLM and Shepherd* issue 64 migrations and 166 preemptions, respectively for a total of 925 requests. In this scenario, ServerlessLLM's effectiveness is constrained by resource limitations.

We further stress the system by running even larger mod-

[8]Based on the average inference time of OPT-6.7B on ShareGPT dataset, the maximum theoretically RPS is 1.79.

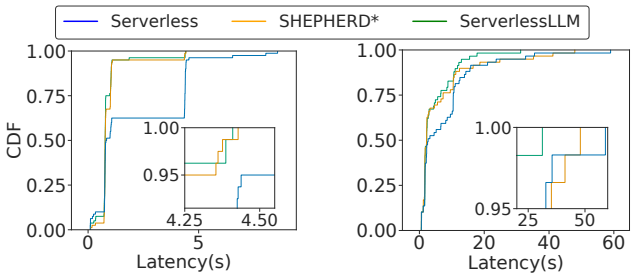


Fig. 40: OPT-13B GSM8K

Fig. 41: OPT-30B GSM8K

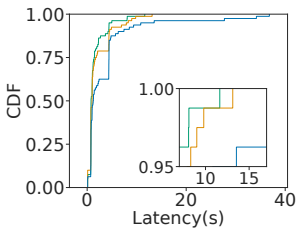


Fig. 42: OPT-13B ShareGPT

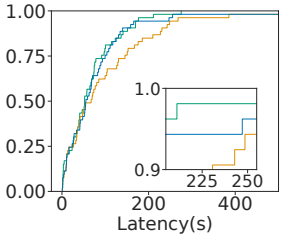


Fig. 43: OPT-30B ShareGPT

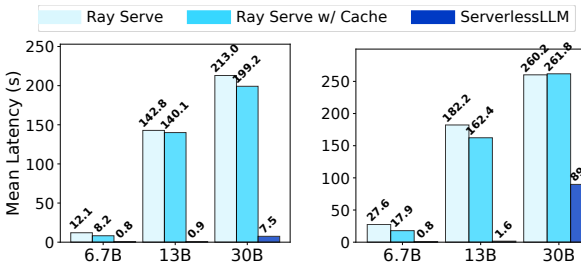


Fig. 44: GSM8K

Fig. 45: ShareGPT

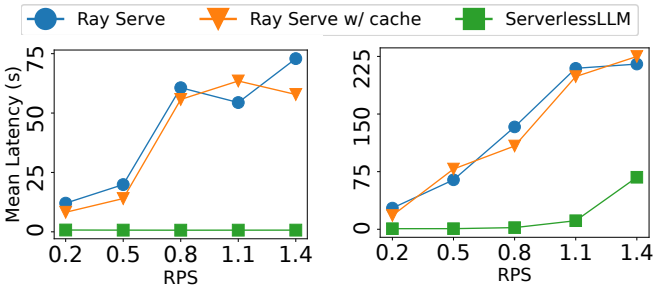


Fig. 46: GSM8k

Fig. 47: ShareGPT

els (OPT-13B and OPT-30B) with GSM8K and ShareGPT datasets. Figure 40 shows the results for those experiments. locality-aware scheduling is more important for larger models as caching them in the main memory can reap better performance. As ServerlessLLM and Shepherd* are both locality-aware, they can make better decisions while scheduling the requests leading to better performance. As Serverless scheduler makes decisions randomly, for GSM8K, we observe that for 35-40% times, the model is loaded from SSD leading to poor performance. We see similar behavior for ShareGPT, OPT-13B experiment too. For the OPT-30B ShareGPT case, the model size is 66 GB. Hence, only two models can be stored in the main memory at any given time reducing the impact of locality-aware scheduling. Even in this extreme case, ServerlessLLM still achieves 35% and 45% lower P99 latency compared to Serverless and Shepherd* respectively.

Time Estimation. The GPU time estimation error is bounded at 5ms, while the SSD loading error is bounded at 40ms. However, we do observe instability in CUDA driver calls. For instance, when migrating a model, we noted that cleaning up GPU states (e.g., KV cache) using `torch.cuda.empty_cache()` can lead to inaccurate estimations, resulting in an average underestimation of 25.78 ms. While infrequent, we observed a maximum underestimation of 623 ms during GPU state cleanup in one out of 119 migrations (as depicted in Figure 38).

4) Entire ServerlessLLM in Action: We aimed to deploy the entire ServerlessLLM with a serverless workload on test bed (ii). Here, we compare ServerlessLLM against state-of-the-art distributed model serving systems: (i) Ray Serve (Version 2.7.0), a version we have extended to support serverless inference scenarios with performance that can match SOTA serverless solutions such as KServe; (ii) Ray Serve with Cache, a version we improved to adopt a local SSD cache on each server (utilizing the LRU policy as in ServerlessLLM) to avoid costly model downloads; and (iii) KServe (Version 0.10.2), the SOTA serverless inference system designed for Kubernetes clusters.

For best performance, Ray Serve and its cache variant are both enhanced by storing model checkpoints on local SSDs and estimating download latency by assuming an exclusively occupied 10 Gbps network. For each system, we set the maximum concurrency to one and set the keep-alive period equal to its loading latency, following prior work [26]. We launch parallel LLM inference clients to generate various workloads, where each request has a timeout threshold of 300 seconds.

Effectiveness of loading-optimized checkpoints. We aimed to assess the effectiveness of loading-optimized checkpoints within a complete serverless workload, employing various model sizes and datasets to diversely test the checkpoint loaders.

In this experiment, as depicted in Figure 44, Ray Serve and Ray Serve with Cache utilize Safetensors. Owing to the large sizes of the models, the SSD cache cannot accommodate all models, necessitating some to be downloaded from the storage server. With OPT-6.7B and GSM 8K, ServerlessLLM starts models in an average of 0.8 seconds, whereas Ray Serve takes 12.1 seconds and Ray Serve with Cache 8.2 seconds, demonstrating an improvement of over 10X. Even with a faster network (i.e., 100 Gbps), the average latency of Ray Serve could drop to 3.8 seconds, making it still 4.7 times slower than ServerlessLLM. The significance of the model loader becomes more pronounced with larger models, as ServerlessLLM can utilize parallel PCIe links when loading large models partitioned on multiple GPUs from pinned memory pool. For instance, with OPT-30B, ServerlessLLM still initiates the

model in 7.5 seconds, while Ray Serve's time escalates to 213 seconds and Ray Serve with Cache to 199.2 seconds, marking a 28X improvement.

This considerable difference in latency substantially affects the user experience in LLM services. Our observations indicate that ServerlessLLM can fulfill 89% of requests within a 300-second timeout with OPT-30B, whereas Ray Serve with Cache manages only 26%.

With the ShareGPT dataset (Figure 45), which incurs a 3.7X longer inference time than GSM 8K, the challenge for model loaders becomes even more intense. For models like 6.7B and 13B, ServerlessLLM achieves latencies of 0.8 and 1.6 seconds on average, respectively, compared to Ray Serve and Ray Serve with Cache, which soar to 182.2 and 162.4 seconds. When utilizing OPT-30B, ServerlessLLM begins to confront GPU limitations (with all GPUs occupied and migration unable to free up more resources), leading to an increased latency of 89.9 seconds. However, this is still a significant improvement over Ray Serve with Cache, which reaches a latency of 261.8 seconds

Effectiveness of live migration and loading scheduler. In evaluating the effectiveness of LLM live migration and the loading scheduler, we created workloads with varying RPS levels. Scenarios with higher RPS highlight the importance of achieving load balancing and locality-aware scheduling since simply speeding up model loading is insufficient to address the resource contention common at large RPS levels.

From Figure 46, it is evident that ServerlessLLM, equipped with GSM8K, consistently maintains low latency, approximately 1 second, even as RPS increases. In contrast, both Ray Serve and Ray Serve with Cache experience rising latency once the RPS exceeds 0.5, which can be attributed to GPU resource shortages. Their inability to migrate LLM inference for locality release or to achieve load balancing, unlike ServerlessLLM, results in performance degradation.

With the more demanding ShareGPT workload, as shown in Figure 47, ServerlessLLM maintains significant performance improvements up to 212 times better over Ray Serve and Ray Serve with Cache across RPS ranging from 0.2 to 1.1. However, at an RPS of 1.4, ServerlessLLM's latency begins to rise, indicating that despite live migration and optimized server scheduling, the limited GPU resources eventually impact ServerlessLLM's performance.

Resource efficiency. A major advantage of the low model startup latency in ServerlessLLM is its contribution to resource savings when serving LLMs. We vary the number of GPUs available on each server to represent different levels of resource provisioning. As shown in Figure 48, ServerlessLLM scales well with elastic resources. With just one GPU per server, ServerlessLLM already achieves a 4-second latency by efficient migrations and swaps. In contrast, Ray Serve with Cache requires at least four GPUs per server to attain a 12-second latency, which is still higher than ServerlessLLM's performance with only one GPU per node. With larger clusters, the resource-saving efficiency of ServerlessLLM is expected

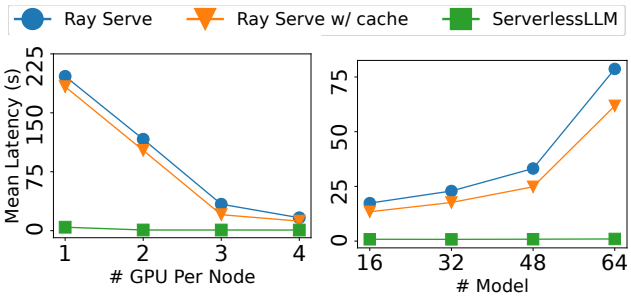


Fig. 48: Impacts of # GPUs per node

Fig. 49: Impacts of # models

to become even more pronounced, as larger clusters offer more options for live migration and server scheduling.

The resource efficiency of ServerlessLLM is further evident when maintaining a fixed number of GPUs while increasing the number of LLMs in the cluster. In Figure 49, with a limited number of models, Ray Serve with Cache can match ServerlessLLM in latency performance. However, as the number of models grows, the performance gap widens, showcasing ServerlessLLM's potential suitability for large-scale serverless platforms.

KServe comparison. In our study, we assess KServe and ServerlessLLM within a Kubernetes cluster. Given that our four-server cluster is unsuitable for a Kubernetes deployment, we instead utilize an eight-GPU server, simulating four nodes with two GPUs each. Since KServe performs slower than the other baselines considered in our evaluation, we only briefly mention KServe's results without delving into details.

With KServe, the GPU nodes initially exhibited a first token latency of 128 seconds. This latency was primarily due to KServe taking 114 seconds to download an OPT-6.7B model checkpoint from the local S3 storage over a 1 Gbps network. However, after applying the same enhancement as those for Ray Serve, we reduced the first token latency to 28 seconds. Despite this improvement, KServe's best latency was significantly higher than those achieved by ServerlessLLM. Notably, ServerlessLLM was the only system able to reduce the latency to within one second.

VI. Security

A. Our approach: SlsDetector

We present *SlsDetector*, an LLM-based framework designed to detect misconfigurations in serverless applications. *SlsDetector* takes a configuration file of the serverless application to be detected as input and outputs structured results, providing a list of detected misconfigurations along with detailed explanations for each issue. The framework is designed to be adaptable, supporting various LLMs. In the following sections, we provide an overview of *SlsDetector* and outline its component.

1) Overview: Fig. 50 shows an overview of *SlsDetector*. It converts a misconfiguration detection request into a meticulously constructed prompt for LLMs. We employ zero-shot

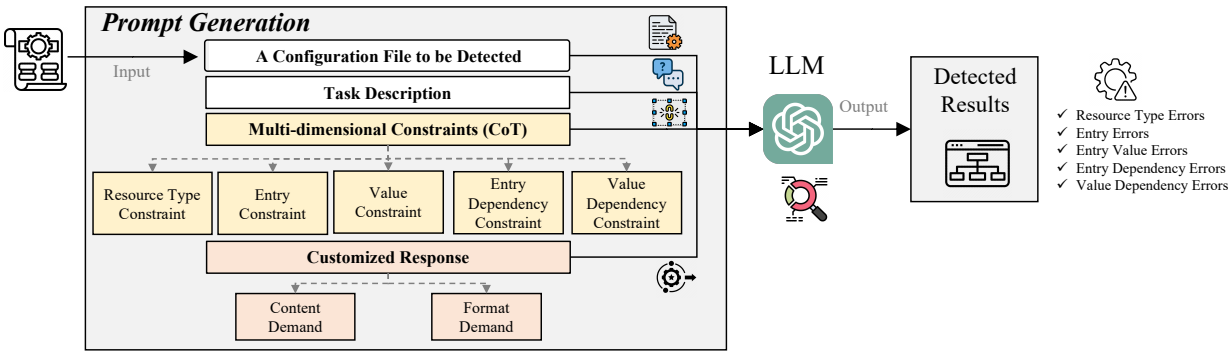


Fig. 50: The overview of our approach *SlsDetector*.

learning to minimize reliance on external sample configurations. This technique, which requires no prior examples, is a popular optimization technique [12], [116], [117]. While many studies [11], [69], [118] have utilized few-shot learning to improve effectiveness by learning from examples during inference, it relies heavily on the quality and selection of labeled samples. In contrast, zero-shot learning avoids the cost and effort associated with sample collection and curation, making it the preferred technique for our framework.

In *SlsDetector*, we design a prompt generation component to construct a tailored prompt focused on the objective of detecting misconfiguration in serverless applications. This prompt is structured into four parts, where multi-dimensional constraints are core of *SlsDetector* and highly context-aware, shown in Fig. 50. Once the prompt is constructed, it is sent to the LLM, which generates the final output. Next, we introduce the prompt generation component in detail.

2) Prompt Generation: We present the prompt content generated by the prompt generation component, which includes: (i) the configuration file to be analyzed, (ii) a task description for the LLMs, (iii) detailed multi-dimensional constraints, and (iv) a customized response. Fig. 51 illustrates our prompt structure.

Task Description The task description includes the following elements: (i) a role-playing instruction designed to enhance the LLMs ability to detect misconfigurations, which is a common prompt optimization technique [12], [119]; and (ii) a task description instruction. In our scenario, the role is designed as "You are an expert at writing AWS SAM configurations for serverless applications", while the task description asks, "Are there any misconfigurations in the above configuration file?". These elements are carefully crafted to clearly outline the tasks the LLM needs to complete within the assigned role.

Multi-dimensional Constraints Multi-dimensional constraints are designed based on the configuration characteristics of serverless applications. The constituent elements of a configuration file are diverse and encompass the following aspects:

• *Resource Types*: Serverless application configurations are primarily centered around defining resource type. Resource types are core to establishing application execution settings. For instance, custom names such as "BucketEventConsumer" (line 16) are assigned to objects tied to specific resource types, such as "AWS::Serverless::Function". Moreover, resource type names are case-sensitive.

• *Configuration Entries*: Each resource type specifies diverse execution parameters, including language runtime and required resources for predefined events. These parameters are represented by configuration entries.

• *Values of Configuration Entries*: Each configuration entry is assigned specific values, often governed by varied constraints. For example, the `Runtime` entry (line 20) has a set of allowed languages, e.g., "python3.6" and "nodejs16.x", while the `Bucket` entry (line 27) accepts only referenced objects.

• *Entry Dependencies*: Certain configuration entries depend on others. These relationships are implicit and generally discovered by consulting documentation.

• *Value Dependencies*: Some values of configuration entries are interdependent across different resource types. For instance, the `RestApiId` entry for API event triggers depends on the object name value corresponding to the "AWS::Serverless::Api" resource. This shows how values can be linked across different resource types, showing extensive value dependencies. Such dependencies are common in configurations due to the collaboration between FaaS and BaaS.

Based on these configuration characteristics, we design five dimensions of constraints (i.e., multi-dimensional constraints) to enhance the LLM's ability to identify serverless application misconfigurations: *resource type constraint*, *entry constraint*, *value constraint*, *entry dependency constraint*, and *value dependency constraint*. Fig. 51 shows their details.

Before explaining constraints, we introduce the Chain of Thought (CoT) technique [120]–[122]. CoT is a reasoning strategy to guide the problem-solving process toward more accurate and logical conclusions. This technique breaks down complex tasks into smaller, manageable steps. A CoT-based prompt includes several intermediate natural language reasoning steps that describe how to solve the task step by step. Based on the principle of this technique, we design our CoT strategy for detecting misconfigurations of serverless applications by guiding LLMs to consider constraints in a "category-by-category" manner.

For *resource type constraint*, we describe it as follows:

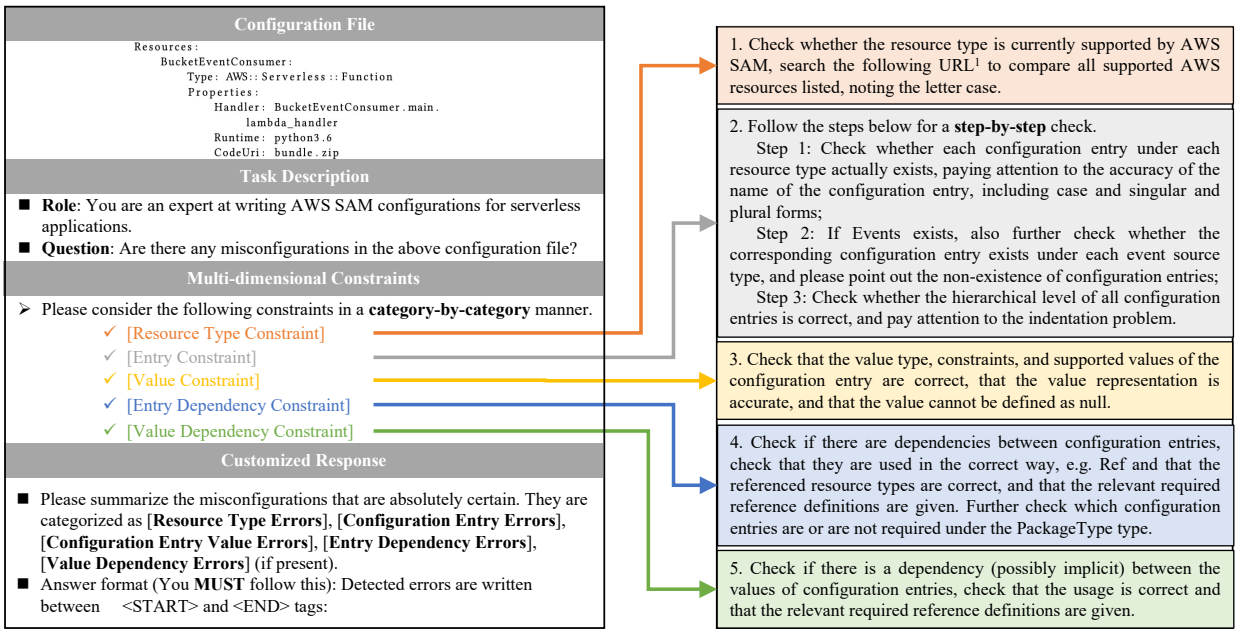


Fig. 51: The prompt structure of *SlsDetector*.

"Check whether the resource type is currently supported by AWS SAM, search the following URL[9] to compare all supported AWS resources listed, noting the letter case." By providing a direct link to the official documentation, we enable *SlsDetector* to effectively identify and compare resource type names, with a particular focus on case sensitivity, a critical aspect in AWS SAM configurations.

For *entry constraint*, we design a three-step validation process to ensure the correctness of configuration entries. The first step checks the correctness of each entry in relation to its corresponding resource type. The second step checks the correctness of event-related entries. The third step ensures that all configuration entries follow the correct hierarchical structure. *SlsDetector* applies these checks using the CoT technique, following a "step-by-step" process. The three steps are as follows. *Step 1: SlsDetector* checks that each configuration entry exists under its respective resource type. This includes checking the entry's name for accuracy, paying particular attention to case sensitivity, and the use of singular or plural forms. *Step 2:* For event-related entries, *SlsDetector* checks that configuration entries corresponding to each event source type are present. If any non-existent entries are given, *SlsDetector* flags them for review. *Step 3: SlsDetector* checks the correct hierarchical structure of all configuration entries, with special attention to indentation. Misplaced or improperly indented entries may lead to errors, as they will not be recognized under the expected resource type. This three-step validation process allows *SlsDetector* to systematically detect errors, ensuring comprehensive and accurate checks for configuration entries.

[9]Supported resource types: https://docs.aws.amazon.com/ serverlessrepo/ latest/devguide/ list-supported-resources.html

For *value constraint*, we describe it as follows: "Check that the value type, constraints, and supported values of the configuration entry are correct, that the value representation is accurate, and that the value cannot be defined as null". These constraints consider various aspects such as the correct data type, valid value ranges, and proper value formatting, ensuring that all values adhere to the required specifications.

For *entry dependency constraint*, we describe it as: "Check if there are dependencies between configuration entries, check that they are used in the correct way". We also provide specific guidelines for validating dependencies, such as checking the accuracy of referenced resource types, ensuring required reference definitions are present, and confirming that required function entries are properly configured.

For *value dependency constraint*, we specify it as: "Check if there is a dependency (possibly implicit) between the values of configuration entries, check that the usage is correct and that the relevant required reference definitions are given". This constraint ensures that value dependency checks are comprehensive across the configuration, helping to maintain consistency and correctness in how values interact and depend on each other within the configurations.

Customized Response We customize the LLMs' output by specifying both the content and format requirements for the responses, ensuring their effectiveness and relevance. For the content demand, we aim to avoid receiving vague or uncertain answers that fail to explicitly identify configuration errors. To achieve it, we instruct the model with the directive: "Please summarize the misconfigurations that are absolutely certain". This ensures that only clear, deterministic errors are returned. Additionally, when applicable, we categorize the detected misconfigurations into specific groups, including "Resource Type Errors," "Configuration Entry Errors," "Configuration

Entry Value Errors," "Entry Dependency Errors," and "Value Dependency Errors".

For the format demand, to eliminate redundant content that does not reveal specific misconfigurations from the raw output, we use delimiters: "¡START¿" and "¡END¿", to mark the required portion of the response. In *SlsDetector*, the desired output is enclosed within these markers, for example: "¡START¿ Resource Type Errors: ..., Value Dependency Errors: ... ¡END¿". This structured way ensures that only the relevant content is captured. During post-processing, *SlsDetector* employs regular expressions to extract the information between these markers efficiently. Although the model might generate additional text beyond the expected response, the use of locators allows for the seamless extraction of relevant content while discarding unnecessary text.

B. Experimental evaluation

To evaluate the effectiveness of *SlsDetector* in identifying misconfigurations within serverless applications, we present four research questions (Section VI-B1). To answer these questions, we detail the evaluation metrics (Section VI-B2), baselines for comparison (Section VI-B3), evaluation dataset (Section VI-B4), and experimental settings (Section VI-B5).

1) Research Questions:

• **RQ1:** How does the effectiveness of *SlsDetector* compared to traditional data-driven methods?

• **RQ2:** How effective is *SlsDetector* without considering our multi-dimensional constraints?

• **RQ3:** How does the non-determinism of LLMs influence the effectiveness of *SlsDetector*?

• **RQ4:** How does the generalization capability of *SlsDetector* when using different LLMs?

2) Evaluation Metrics: We use $precision$, $recall$, and $F1\text{-}score$ as evaluation metrics to compare *SlsDetector* against the baseline methods at the configuration parameter level, i.e., configuration entries or values. We check whether the detection approach can accurately determine the validity of each configuration parameter within the configuration file. $precision$ measures the proportion of correctly identified misconfigured parameters among all parameters flagged as misconfigured. $recall$ quantifies the ability of the approach to detect actual misconfigurations by calculating the proportion of true misconfigured parameters that are correctly identified. $F1\text{-}score$ provides a balanced measure that accounts for the significance of both false positives and false negatives. These metrics are calculated through True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN), explained in Table III. $precision = \frac{TP}{TP+FP}$, $recall = \frac{TP}{TP+FN}$, and $F1\text{-}score = 2 \times \frac{precision \times recall}{precision+recall}$. Values range from 0% to 100%, with scores closer to 100% indicating greater effectiveness.

3) Baseline Methods: We implement two types of baselines to evaluate effectiveness. Given the lack of approaches specifically tailored for detecting misconfigurations in serverless computing, we first draw on principles from established data-driven techniques used in prior configuration studies [65]–[68]. By adapting these methods, we create a data-driven baseline suited to the characteristics of serverless applications. Additionally, we introduce a straightforward LLM-based baseline as a second comparison, which does not consider our designed constraints.

• **Baseline 1:** Data-driven method (DD method). We implement a data-driven approach for serverless applications by learning configuration patterns from a dataset of configuration files. As no existing dataset specifically focuses on serverless application configurations, we collect our data from the AWS Serverless Application Repository (SAR) [123], an official repository for serverless applications where each application is packaged with an AWS SAM template and links to relevant configuration files. We include all configuration files associated with serverless applications that have been successfully deployed at least once as of August 18, 2023, which is the date we collected this dataset. This results in a collection of 701 configuration files across 658 serverless applications, with some links providing multiple configuration files representing distinct configurations. Given the correctness of ensuring the dataset, we conduct a careful manual review of the configuration files. This review was performed by the first two authors, who have a background in cloud computing. Identified issues were discussed and resolved with consensus among the authors. To assess the consistency of independent labeling, we employ Cohen's Kappa (κ) [124], a widely used metric for measuring inter-rater agreement. The resulting κ value of 0.916 indicates an almost perfect agreement and a reliable labeling procedure [125].

Using this dataset, we learn configuration patterns, focusing on common resource types, configuration entries, values, and dependencies among entries and values. To streamline this process, we first standardize the configuration files into a uniform representation. Object names for various resource types are identified, with object names replaced by standardized labels (e.g., a placeholder like "PH+resource type") for consistency across configuration entries and values. Leveraging this standardized dataset, we extract the used resource types, entries, and values.

To detect dependencies among both entries and values, we apply association rule mining techniques [126], [127]. Specifically, we use the FP-Growth algorithm [128], which is known for its scalability. We need to set a support threshold for frequent itemsets using the formula $\alpha \times len$, where len represents the total number of configuration files, a deterministic value, and α is a percentage that indicates the desired mining granularity. Leveraging mined frequent itemsets, we generate association rules by utilizing traversal way and dividing items into left and right sets, where items in the right set must appear if those in the left set are present. These rules reveal the configuration dependencies. If the tested file contains all items in a left set, this approach checks whether it includes the corresponding items in the right set. If any items are missing, it reports them.

• **Baseline 2**: Basic LLM method (BL method). It is designed using a straightforward prompt that does not take our multi-

TABLE III: The explanation of TP, FP, TN, and FN in our scenario.

TP	A misconfigured parameter correctly identified as misconfigured
FP	A correctly configured parameter mistakenly flagged as misconfigured
TN	A correctly configured parameter accurately recognized as valid
FN	A misconfigured parameter that is overlooked or incorrectly classified as valid

dimensional constraints into account. This prompt contains the configuration file content followed by a task description. Similarly, the output is enclosed within a locator pair, "¡START¿" and "¡END¿", to delimit the required response. This prompt is shown in Fig. 52.

4) Evaluation Dataset: We conduct experimental evaluations on a dataset comprising three types of configurations. The first type includes error-free configurations, enabling us to evaluate true negatives and false positives in detection. The second type contains configurations with real-world errors, allowing for the assessment of true positives and false negatives. Although this second type is somewhat free of data leakage concerns of LLMs, we include a third type to strengthen the validity of our conclusions. The third type consists of configurations with injected errors, which are not exposed to LLMs during training, thereby eliminating data leakage concerns. By utilizing these diverse configurations, we can achieve a valid evaluation.

• *Configurations without Errors (26).* We manually collect configuration files that have been successfully executed without errors. This data is separate from the one used to mine configuration patterns in the data-driven approach.

We collect real-world configuration cases from GitHub. GitHub issues provide rich information, including developer discussions and related code or configuration fragments. We conduct the following steps. First, on July 2, 2024, the date we collected this data, we searched GitHub using the keywords "AWS," "serverless," and "configuration," which yielded more than 8,000 relevant configuration-related issues. We then manually reviewed these issues to extract correct configuration fragments from the problematic casesa time-consuming and challenging process. To facilitate this task, the first two authors jointly review the configurations. Initially, they filter through the configuration fragments by searching for terms including "successful," "successfully," and "it works" within the issues to identify correct configurations. For the fragments that matched, they conducted a manual verification process to ensure that the configurations were indeed error-free. Over two months, the two authors identified 52 configuration fragments that met our criteria. These error-free real-world configuration fragments are divided into two sets: 26 (naming from case 1 to case 26) are used to evaluate error-free configurations, while the remaining 26 (naming from case 27 to case 52) are reserved for generating configurations with injected errors, which is explained in detail later.

• *Real-world Misconfigurations (58).* To evaluate the effectiveness of approaches in identifying real-world misconfigurations in serverless applications, we construct a relevant dataset by mining real-world configuration issues from GitHub. These issues need to contain clearly identified root causes as ground truths, enabling us to accurately assess the effectiveness of detection results.

The selection process is as follows: First, we use the same keywords (i.e., "AWS," "serverless," and "configuration") to search for relevant issues on GitHub on July 2, 2024. Next, we identify satisfied issues based on the following criteria: (i) the issue is marked as closed, indicating that it has been resolved; (ii) the issue includes a configuration fragment based on AWS SAM for analysis; and (iii) the discussion concludes with a clearly identified root cause of the problem. Using these criteria, we select 58 real-world configuration problems encountered in serverless applications, surpassing the scale of prior studies on configuration-related research [65], [127].

To ensure the accuracy of the configuration errors to be detected, we meticulously review each real-world configuration file in conjunction with its identified root cause. During this process, we also manually identify and address any potential configuration issues (e.g., outdated runtime) that could influence the evaluation.

• *Injected Misconfigurations (26).* We construct injected misconfigurations by generating various errors in the correct configuration files. To achieve this, we use 26 error-free configuration files named from case 27 to case 52. Misconfigurations of different types are then generated, following misconfiguration generation rules from prior studies [61], [62], [69], [129], [130]. Prior studies [69], [129] showed that these rules can cover most configurations. In addition to utilizing existing rules, we extend specific misconfiguration generation rules tailored to serverless application configurations, as outlined in Table IV. For each selected configuration file, we randomly sample a configuration parameter that aligns with the subcategories in Table IV and generate invalid configurations, creating a new erroneous configuration file for detection. In total, we generate 26 configuration files with injected misconfigurations for evaluation.

Our evaluation dataset contains 110 configuration files with corresponding ground-truth answers. Fig. 53 shows its details. Of these, 26 are error-free configuration files, 58 contain real-world errors, and 26 have injected errors. Across all configuration parameters, there are 4,108 correct configuration parameters and 308 misconfigured ones. Among the misconfigured parameters, 90 involve incorrect resource types, 108 have misconfigured entries, 48 contain incorrect values, 39 exhibit entry dependency issues, and 23 have value dependency issues. We analyze the detection results across all configuration parameters to obtain TP, FP, TN, and FN. We then calculate $precision$, $recall$, and $F1\text{-}score$ to evaluate the effectiveness of the detection.

Configuration File	Task Description	Response
Resources: BucketEventConsumer: Type: AWS::Serverless::Function Properties: Handler: BucketEventConsumer.main. lambda_handler Runtime: python3.6 CodeUri: bundle.zip	■ **Question**: Are there any misconfigurations in the above configuration file?	➢ Answer format (You **MUST** follow this): Detected errors are written between <START> and <END> tags:

Fig. 52: The prompt of BL method.

TABLE IV: Misconfiguration generation rules (we use generation rules from previous work [61], [62], [129], [130] and customize them in our scenario.)

Category	Subcategory	Specification	Generation Rules
Syntax	Resource type	Value set = {AWS::Serverless::Function, AWS::Serverless::Api, ...}	Generate a resource type that does not belong to the value set
	Entry	Value set = {entry1, entry2, ...}, specific entries are used in a certain resource type	Generate an invalid entry for a resource type
Range	Basic numeric	Valid range constrained by data type	Generate values outside the valid range (e.g., max value+1)
	Enum	Options, value set = {enum1, enum2, ...}, specific values are used in a certain configuration entry	Generate a value that does not belong to set
Dependency	Entry relationship	$(P_1, V, \diamond) \mapsto P_2, \diamond \in \{>, \geq, =, \neq, <, \leq, occurrence\}$	Generate invalid entry relationships for configuration entries $(P_1, V, \neg\diamond)$
	Value relationship	$(P_1, P_2, \diamond), \diamond \in \{>, \geq, =, \neq, <, \leq, occurrence\}$	Generate invalid value relationship for configuration entry values $(P_1, P_2, \neg\diamond)$

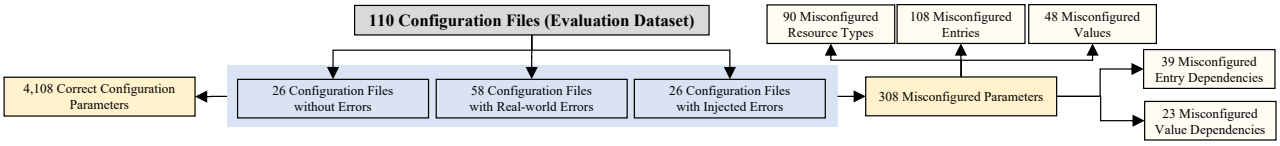


Fig. 53: The Details of Evaluation Dataset.

5) Experimental Settings: We introduce our parameter settings, experimental repetitions, and experimental environment.

Parameter Settings. For **RQ1**, the compared DD method needs to specify a frequent threshold, α. We experiment with various threshold levels: low (1%), medium (3% and 5%), and high (10%). A lower threshold corresponds to a lower support value, enabling the discovery of more dependencies. For comparisons with *SlsDetector*, we use a default α value of 5%. Experimental results also show that 5% is optimal for achieving the best effectiveness results in DD methods. We also report results for both *SlsDetector* and DD method across other thresholds. For **RQ2**, we compare *SlsDetector* with the BL method, both of which leverage LLMs. We select ChatGPT-4o as the default LLM due to the widespread use and outstanding performance of ChatGPT in recent research [12], [69]. A crucial parameter of LLMs is the temperature, which controls the level of randomness in the generated responses. To ensure reproducibility and consistency, we follow the previous work [11], [118], [131], [132] to set the temperature to 0 for all identical queries. For **RQ3**, there are no specific parameters to be set. For **RQ4**, we evaluate the generalization capability of *SlsDetector* across various LLMs, excluding ChatGPT-4o. Specifically, we utilize an open-source model, Llama 3.1 (405B) Instruct Turbo, and a proprietary model, Gemini 1.5 Pro. These models are among the top-ranked LLMs [133]. As with RQ2, we set the temperature of LLMs to 0 to maintain consistent outputs across repeated queries.

Experimental Repetitions. For experiments involving stochastic processes, we follow established best practices [11], [131], repeating each experiment five times and reporting the mean evaluation metrics to reduce the impact of random variations.

Experimental Environment. Our experiments were conducted on an Ubuntu 18.04.4 LTS server with an Intel Xeon (R) 4-core processor and 24 GiB of memory. The LLMs were accessed through their respective APIs. While all methods are implemented in Python, their misconfiguration detection capabilities are independent of the underlying programming language.

C. Evaluation Results

This section gives and discusses the results of each research question.

1) RQ1: Effectiveness of SlsDetector and Data-Driven Method (DD method): This section explores the effectiveness of *SlsDetector* in comparison to the DD method. *SlsDetector* has a significant advantage in the effectiveness aspect. Table V presents their results in detecting misconfigurations in serverless applications. Specifically, *SlsDetector* achieves a $precision$ of 72.88%, $recall$ of 88.18%, and $F1\text{-}score$ of 79.75%. In contrast, the DD method, with its default threshold

TABLE V: RQ1: Results about *SlsDetector* and DD method.

Methods	$precision$	$recall$	$F1\text{-}score$
DD method with 5% threshold (default)	19.06%	70.78%	30.03%
SlsDetector (vs DD method with 5% threshold)	72.88% (↑ 53.82%)	88.18% (↑ 17.40%)	79.75% (↑ 49.72%)
DD method with 10% threshold	17.70%	64.61%	27.79%
DD method with 3% threshold	18.83%	70.13%	29.69%
DD method with 1% threshold	18.85%	70.45%	29.75%

of 5%, only reaches a $precision$ of 19.06%, $recall$ of 70.78%, and $F1\text{-}score$ of 30.03%. *SlsDetector* outperforms the DD method, increasing $precision$ by 53.82 percentage points, $recall$ by 17.40 percentage points, and $F1\text{-}score$ by 49.72 percentage points, showing its superior effectiveness.

We investigate why the DD method produces less effective results. One major issue is its low $precision$ (19.06%) and $F1\text{-}score$ (30.03%). We further observe TP, FN, FP, and TN values obtained by the DD method across all configuration parameters, as shown in Table VI. Results show that the FP value is 926, indicating that 22.54% of the 4,108 correct configuration parameters are mistakenly flagged as misconfigurations. In contrast, on average, *SlsDetector* misclassifies only 2.48% of correct configuration parameters as misconfigurations. Thus, the low effectiveness of the DD method is attributed to high false positives. As a data-driven approach, the DD method learns configuration patterns based on historical data, which mainly includes previously used configurations. This reliance makes it difficult to accurately identify configurations that are either rare or newly supported, resulting in numerous false positives. Thus, the DD method fails to detect some valid configurations that are indeed supported, leading to its low $precision$ and $F1\text{-}score$.

We also compare the effectiveness of the DD method under different thresholds α: 10%, 3%, and 1%, with the results presented in Table V. As α decreases from 10% to 1%, the evaluation metrics show improvement. Specifically, $precision$ increases from 17.70% to 18.85%, $recall$ rises from 64.61% to 70.45%, and $F1\text{-}score$ improves from 27.79% to 29.75%. To further explore the reasons for their changes, we give TP, FN, FP, and TN results of the DD method under different thresholds, as shown in Table VII. The primary reason for improvements is that lower α mines more dependencies among entries or values. This enables the accurate identification of a larger number of misconfigured parameters. Specifically, the TP value for the DD method at a 10% threshold is 199, whereas at a 1% threshold, it increases to 217. This improvement leads to a higher $recall$, increasing from 64.61% to 70.45%. However, a lower α also increases the risk of generating potentially invalid dependencies, resulting in correctly configured parameters being mistakenly flagged as misconfigurations. This is evident from the FP values: the FP value for the DD method at a 10% threshold is 925, while at a 1% threshold, it increases to 934. As a result, $precision$ shows only a modest improvement, from 17.70% to 18.85%. For $F1\text{-}score$, lowering α enhances the effectiveness of the DD method, reaching a value of 29.75%. However, it still significantly lags behind the 79.75% achieved by *SlsDetector*.

In addition, we observe that a threshold of 5% for the DD method yields superior results compared to 1%, 3%, and 10%, suggesting that 5% is an optimal threshold for the data-driven method in this scenario. In the threshold of 5%, the FP-growth algorithm can effectively mine relationships without losing valid dependencies or generating an excessive number of invalid dependencies. However, even at 5%, the effectiveness of the DD method remains significantly lower than that of *SlsDetector*, with particularly low $precision$ and $F1\text{-}score$.

> **Ans. to RQ1:** *SlsDetector* achieves a $precision$ of 72.88%, $recall$ of 88.18%, and $F1\text{-}score$ of 79.75%, surpassing data-driven methods across all metrics. It shows significant improvements, with increases of 53.82 percentage points in $precision$, 17.40 percentage points in $recall$, and 49.72 percentage points in $F1\text{-}score$. These results suggest the high effectiveness of *SlsDetector*.

2) RQ2: Effectiveness of SlsDetector and Basic LLM-based Method (BL method): We explore the effectiveness of *SlsDetector* in comparison to the BL method using the default ChatGPT-4o for detecting misconfigurations in serverless applications. Table VIII presents their results, showing that *SlsDetector* is more effective than the BL method. Specifically, *SlsDetector* achieves a $precision$ of 72.88%, $recall$ of 88.18%, and an $F1\text{-}score$ of 79.75%. The BL method achieves a $precision$ of 51.65%, $recall$ of 65.00%, and an $F1\text{-}score$ of 57.55%. *SlsDetector* outperforms the BL method across all metrics, with increases in $precision$ by 21.23 percentage points, $recall$ by 23.18 percentage points, and $F1\text{-}score$ by 22.20 percentage points.

We investigate the reasons for the low effectiveness of the BL method. Table IX shows TP, FN, FP, and TN values obtained by the BL method across all configuration parameters. The results indicate that the BL method has a low TP value of 200, successfully identifying only 64.94% of the 308 misconfigured parameters. In contrast, *SlsDetector* accurately identifies an average of 272 (88.31%) misconfigured parameters. To further explore the root causes of the BL method's low effectiveness, we examine the average number of misconfigured parameters correctly identified across different categories. As presented in Table X, the BL method identifies fewer errors than *SlsDetector* in each category, including resource types, entries, values, entry dependencies, and value dependencies. Particularly, the BL method only detects 17.95% of misconfigured entry dependencies, while *SlsDetector* detects

TABLE VI: RQ1: Results* of TP, FN, FP, and TN for DD method and *SlsDetector*.

Methods	308 misconfigured parameters		4,108 correct configuration parameters	
	TP	**FN**	**FP**	**TN**
DD method (default)	218 (70.78%)	90 (29.22%)	**926 (22.54%)**	3,182 (77.46%)
SlsDetector (default)	272 (88.31%) ✓	36 (11.69%) ✓	102 (2.48%) ✓	4,006 (97.52%) ✓

* Higher TP and TN are preferable, while lower FN and FP are desired.

TABLE VII: RQ1: Results of TP, FN, FP, and TN for DD method with different thresholds α.

Methods	308 misconfigured parameters		4,108 correct configuration parameters	
	TP	**FN**	**FP**	**TN**
DD method with 10% threshold	199	109	925	3,183
DD method with 3% threshold	216	92	931	3,177
DD method with 1% threshold	217	91	934	3,174

TABLE VIII: RQ2: Results about *SlsDetector* and BL method using the default LLM (ChatGPT-4o).

Baseline	$precision$	$recall$	$F1\text{-}score$	**Ours**	$precision$	$recall$	$F1\text{-}score$
BL method	51.65%	65.00%	57.55%	***SlsDetector*** (vs BL method)	72.88% (↑ 21.23%)	88.18% (↑ 23.18%)	79.75% (↑ 22.20%)

TABLE IX: RQ2: Results of TP, FN, FP, and TN for BL method and *SlsDetector*

Methods	308 misconfigured parameters		4,108 correct configuration parameters	
	TP	**FN**	**FP**	**TN**
BL method (default)	**200 (64.94%)**	107 (34.74%)	188 (4.58%)	3,920 (95.42%)
SlsDetector (default)	272 (88.31%) ✓	36 (11.69%) ✓	102 (2.48%) ✓	4,006 (97.52%) ✓

* Higher TP and TN are preferable, while lower FN and FP are desired.

97.44%. We check specific configurations and observe that the BL method struggles to identify configuration entries related to cloud service resources that should co-occur with the event sources defined by serverless functions. For instance, the configuration entry `RestApiId` under an event source of type "Api" should be associated with configuration entries of the "AWS::Serverless::Api" resource type. Overall, these results indicate that relying solely on the raw capabilities of LLMs, as done in the BL method, is inadequate for the complex task of detecting misconfigurations in serverless applications. A key factor contributing to the improved effectiveness of *SlsDetector* is its ability to incorporate multi-dimensional constraints for guiding LLM inferences. These constraints are designed across various dimensions. By integrating them into the analysis, *SlsDetector* enhances the decision-making process, resulting in a more effective identification of misconfigurations.

Ans. to RQ2: *SlsDetector* outperforms the BL method across all metrics using the default ChatGPT-4o, with increases in $precision$ by 21.23 percentage points, $recall$ by 23.18 percentage points, and $F1\text{-}score$ by 22.20 percentage points. This suggests that integrating multi-dimension constraints is beneficial for handling misconfiguration detection in serverless applications.

3) RQ3: Impact of Non-determinism on SlsDetector: We explore how the non-determinism of LLMs impacts our evaluation results. As detailed in Section VI-B5, each experiment is repeated five times. We analyze their results shown in Table XI to assess the reliability of our conclusions. Results show that while the non-determinism of LLMs can influence evaluation results, its effect is relatively minor. *SlsDetector* consistently achieves high effectiveness across different trials. $precision$ ranges from 70.35% to 76.14%, $recall$ varies between 84.74% and 91.88%, and $F1\text{-}score$ falls between 76.88% and 81.21%. Even the lowest values, i.e., $precision$ at 70.35%, $recall$ at 84.74%, $F1\text{-}score$ at 76.88%, are still higher than $precision$ (19.06%), $recall$ (70.78%), and $F1\text{-}score$ (30.03%) of the data-driven approach. Furthermore, the lowest metric values for *SlsDetector* remain approximately 20 percentage points higher than the average results (i.e., $precision$ at 51.65%, $recall$ at 65.00%, $F1\text{-}score$ at 57.55%) of the basic LLM-based method, as shown in Table VIII. This suggests that our conclusions regarding *SlsDetector* are not affected by the non-determinism of LLMs.

Ans. to RQ3: Our conclusions are not impacted by the non-determinism of LLMs.

4) RQ4: Generalization Capability of SlsDetector: To explore the generalization of *SlsDetector*, we use two additional models: the open-source Llama 3.1 (405B) Instruct Turbo model and the proprietary Gemini 1.5 Pro model. *SlsDetector* consistently achieves high effectiveness across all metrics, with $precision$, $recall$, and $F1\text{-}score$ values exceeding 70%, regardless of the LLM utilized. Table XII shows their results. Specifically, with the Llama 3.1 (405B) Instruct Turbo, *SlsDetector* achieves a $precision$ of 70.27%, $recall$ of 78.38%, and an $F1\text{-}score$ of 74.05%. With the Gemini 1.5 Pro model, *SlsDetector* yields a $precision$ of 71.72%, $recall$ of 74.35%,

TABLE X: RQ2: The average number of misconfigured parameters correctly identified as misconfigured across different categories.

Methods	Misconfigured resource types (90)	Misconfigured entries (108)	Misconfigured values (48)	Misconfigured entry dependencies (39)	Misconfigured value dependencies (23)
BL	62 (68.89%)	83 (76.85%)	39 (81.25%)	7 (17.95%)	12 (52.17%)
SlsDetector	84 (93.33%) ✓	93 (86.11%) ✓	43 (89.58%) ✓	38 (97.44%) ✓	19 (82.61%) ✓

TABLE XI: RQ3: Evaluation metrics results of *SlsDetector* across five repetitions.

Metrics	Repetition 1	Repetition 2	Repetition 3	Repetition 4	Repetition 5	**Mean**
$precision$	71.83%	70.78%	70.35%	75.28%	76.14%	72.88%
$recall$	91.88%	91.23%	84.74%	86.04%	87.01%	88.18%
$F1\text{-}score$	80.63%	79.72%	76.88%	80.30%	81.21%	79.75%

TABLE XII: RQ4: Results about *SlsDetector* and BL method using various LLMs.

BL Method	$precision$	$recall$	$F1\text{-}score$	**Ours**	$precision$	$recall$	$F1\text{-}score$
BL (GPT-4o)	51.65%	65.00%	57.55%	***SlsDetector*** (GPT-4o) (vs BL)	72.88% (↑ 21.23%)	88.18% (↑ 23.18%)	79.75% (↑ 22.20%)
BL (Llama)	48.88%	58.38%	53.09%	***SlsDetector*** (Llama) (vs BL)	70.27% (↑ 21.39%)	78.38% (↑ 20.00%)	74.05% (↑ 20.96%)
BL (Gemini)	44.41%	22.86%	30.11%	***SlsDetector*** (Gemini) (vs BL)	71.72% (↑ 27.31%)	74.35% (↑ 51.49%)	72.93% (↑ 42.82%)

and an $F1\text{-}score$ of 72.93%. Among these, *SlsDetector* with ChatGPT-4o offers the highest effectiveness, while *SlsDetector* with the Gemini 1.5 Pro model shows comparatively lower metrics but still achieves a high $F1\text{-}score$ of 72.93%.

We also evaluate the BL method with different LLMs, shown in Table XII. We observe considerable variability. While the BL method achieves $precision$, $recall$, and $F1\text{-}score$ values approaching or exceeding 50% when using ChatGPT-4o and Llama 3.1 (405B) Instruct Turbo, its effectiveness drops substantially with the Gemini 1.5 Pro model, where $precision$ is 44.41%, $recall$ is 22.86%, and $F1\text{-}score$ is 30.11%. This indicates a key limitation of the BL method: its effectiveness is dependent on the specific LLM used. In contrast, *SlsDetector* provides the ability to maintain consistent effectiveness across different models, showing its generalization.

We compare the effectiveness differences between *SlsDetector* and the BL method when using the same LLM. As discussed in RQ2, *SlsDetector* outperforms the BL method with ChatGPT-4o by over 20 percentage points across all evaluation metrics. From Table XII, when utilizing the Llama 3.1 (405B) Instruct Turbo model, *SlsDetector* also achieves improvements of over 20 percentage points across all evaluation metrics compared to the BL method. With the Gemini 1.5 Pro model, *SlsDetector* outperforms the BL method with even greater gains, achieving 27.31 percentage points higher in $precision$, 51.49 percentage points higher in $recall$, and 42.82 percentage points higher in $F1\text{-}score$. The effectiveness gap is especially pronounced with Gemini 1.5 Pro, showing an effectiveness difference of around 50% in $recall$ and $F1\text{-}score$, underscoring the effectiveness of our approach.

Ans. to RQ4: *SlsDetector* exhibits generalization capability, consistently achieving highly effective results across various LLMs. In contrast, the effectiveness of the BL method varies significantly depending on the chosen LLM. When using the Gemini 1.5 Pro model, *SlsDetector* outperforms the BL method by approximately 50 percentage points in both $recall$ and $F1\text{-}score$.

VII. Conclusion

This paper provides a comprehensive exploration of the challenges and opportunities within serverless computing, offering innovative solutions across four pivotal topics: cold start performance, programming frameworks, resource management, and security. In this section, we summarize the contributions with relevant background context for each area:

Cold Start Performance: Serverless computing excels in providing on-demand, event-driven services, but cold start latency remains a significant bottleneck, especially in high-demand scenarios. Cold starts occur when serverless functions are invoked after being idle, requiring container initialization and resulting in delays. Traditional solutions, such as function preloading or caching, often fail to balance performance and resource efficiency. In this work, we delve into the mechanics of cold starts, proposing advanced methods like container prewarming, snapshot restoration, and optimized scheduling. These approaches reduce latency while maintaining scalability, significantly enhancing the responsiveness of serverless platforms.

Programming Frameworks: The rise of serverless architectures demands robust programming frameworks that simplify the development and deployment of applications. Existing frameworks like OpenFaaS and Knative provide essential tools for orchestrating serverless functions, but their support for emerging workloads, such as machine learning (ML) and AI inference, remains limited. Our research evaluates these frameworks strengths and limitations, focusing on their

usability, multi-cloud compatibility, and workload adaptability. We propose improvements to enhance developer productivity, streamline function orchestration, and enable support for advanced workflows, contributing to a more seamless application-building experience.

Resource Management: Resource elasticity is a hallmark of serverless computing, enabling systems to scale dynamically in response to fluctuating workloads. However, achieving this elasticity efficiently is a complex challenge. Ineffective resource allocation can lead to underutilization or bottlenecks, hampering system performance. Our work addresses these issues by introducing techniques for dynamic provisioning, workload prediction, and task scheduling. Additionally, we analyze the trade-offs between cost and performance, proposing tailored solutions for heterogeneous environments, thereby ensuring optimal resource usage in various deployment scenarios.

Security: The flexibility of serverless computing comes with unique security challenges, including configuration mismanagement, vulnerabilities in third-party dependencies, and risks of data breaches during function execution. These risks are exacerbated by the ephemeral and distributed nature of serverless environments. We identify these vulnerabilities and present novel strategies to mitigate them, such as leveraging machine learning for anomaly detection, implementing fine-grained access control, and enhancing runtime monitoring. These solutions strengthen the security posture of serverless systems without compromising their inherent agility.

By addressing these key areas, this paper contributes to overcoming critical barriers in serverless computing, paving the way for its broader adoption in diverse and complex application domains. Our findings provide actionable insights to developers, architects, and researchers striving to optimize serverless systems for the demands of modern computing.

ACKNOWLEDGMENT

This work was carried out as a report of my project for the Advanced Operating Systems (CSC 6032) course. I would like to express my sincere gratitude to Professor Yeh-Ching Chung for his guidance and support throughout the project. I also wish to thank Teaching Assistant ShiHao Hong for his valuable assistance and feedback. Their contributions were instrumental in the completion of this work.

REFERENCES

[1] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, Nov. 2019.

[2] S. Eismann, J. Scheuner, E. V. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, and A. Iosup, "The state of serverless applications: collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, 2021.

[3] "Amazon s3," https://aws.amazon.com/s3/, 2024.

[4] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The serverless trilemma: function composition for serverless computing," in *ACM SIGPLAN 2017*. ACM, 2017.

[5] HuggingFace, "Safetensors: ML Safer for All," https://github.com/huggingface/safetensors, 2023, accessed on 2024-01-22.

[6] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. T. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "OPT: Open Pre-trained Transformer Language Models," *CoRR*, vol. abs/2205.01068, 2022.

[7] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[8] E. Almazrouei, H. Alobeidli, A. Alshamsi, A. Cappelli, R. Cojocaru, M. Debbah, E. Goffinet, D. Heslow, J. Launay, Q. Malartic, B. Noune, B. Pannier, and G. Penedo, "Falcon-40B: an open large language model with state-of-the-art performance," 2023.

[9] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021.

[10] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[11] J. Xu, R. Yang, Y. Huo, C. Zhang, and P. He, "Divlog: Log parsing with prompt enhanced in-context learning," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[12] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1703–1726, 2024.

[13] M. X. Weng, J. Lu, and H. Ren, "Unrelated parallel machine scheduling with setup consideration and a total weighted completion time objective," *International Journal of Production Economics*, vol. 70, no. 3, 2001.

[14] S. Webster and M. Azizoglu, "Dynamic programming algorithms for scheduling parallel machines with family setup times," *Computers & Operations Research*, vol. 28, no. 2, 2001.

[15] "Gurobi optimizer reference manual," Gurobi Optimization, LLC, 2019. [Online]. Available: http://www.gurobi.com

[16] A. Ilyushkin and D. Epema, "The impact of task runtime estimate accuracy on scheduling workloads of workflows," in *CCGRID*, 2018.

[17] W. Shao, F. Xu, L. Chen, H. Zheng, and F. Liu, "Stage delay scheduling: Speeding up dag-style data analytics jobs with resource interleaving," in *ICPP, Proc.*, 2019.

[18] R. Sakellariou and H. Zhao, "A hybrid heuristic for dag scheduling on heterogeneous systems," in *IPDPS 2004*. IEEE, 2004.

[19] B. Gacias, C. Artigues, and P. Lopez, "Parallel machine scheduling with precedence constraints and setup times," *Computers & Operations Research*, vol. 37, no. 12, 2010.

[20] F. Wu, Q. Wu, and Y. Tan, "Workflow scheduling in cloud: a survey," *The Journal of Supercomputing*, vol. 71, no. 9, 2015.

[21] M. Afzalirad and J. Rezaeian, "Resource-constrained unrelated parallel machine scheduling problem with sequence dependent setup times, precedence constraints and machine eligibility restrictions," *Computers & Industrial Engineering*, vol. 98, 2016.

[22] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 1049–1062. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/zhang-chengliang

[23] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[24] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "Infless: a native serverless system for low-latency, high-throughput inference," in *Proceedings of the 27th ACM International Conference*

on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 768781. [Online]. Available: https://doi.org/10.1145/3503222.3507709

[25] M. Yu, Z. Jiang, H. C. Ng, W. Wang, R. Chen, and B. Li, "Gillis: Serving large neural networks in serverless functions with automatic model partitioning," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2021, pp. 138–148.

[26] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 397–411.

[27] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, "Tetris: Memory-efficient serverless inference through tensor sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[28] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 199–216. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[29] Microsoft, "Azure ML," https://learn.microsoft.com/en-us/azure/machine-learning, 2023, accessed on 2024-01-22.

[30] T. K. Authors, "Kserve," https://github.com/kserve/kserve, 2023, accessed on 2024-01-22.

[31] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 443–457. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/wang-ao

[32] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid task provisioning with Serverless-Optimized containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 57–70. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/oakes

[33] Z. Li, J. Cheng, Q. Chen, E. Guan, Z. Bian, Y. Tao, B. Zha, Q. Wang, W. Han, and M. Guo, "Rund: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing," in *USENIX Annual Technical Conference*. USENIX Association, 2022, pp. 53–68.

[34] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. 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*, 2020, pp. 467–481.

[35] L. Ao, G. Porter, and G. M. Voelker, "Faasnap: Faas made fast using snapshot-based vms," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 730746. [Online]. Available: https://doi.org/10.1145/3492321.3524270

[36] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "Seuss: skip redundant paths to make serverless fast," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3392698

[37] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 559572. [Online]. Available: https://doi.org/10.1145/3445814.3446714

[38] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 419–433. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shillaker

[39] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 205–218. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad

[40] L. Mai, G. Li, M. Wagenländer, K. Fertakis, A.-O. Brabete, and P. Pietzuch, "{KungFu}: Making training in distributed machine learning adaptive," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 937–954.

[41] M. Wagenländer, L. Mai, G. Li, and P. Pietzuch, "Spotnik: Designing distributed machine learning for transient cloud resources," in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[42] X. Wei, F. Lu, T. Wang, J. Gu, Y. Yang, R. Chen, and H. Chen, "No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 497–517. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/wei-rdma

[43] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance serverless computing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 923–935. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/akkus

[44] M. Yu, A. Wang, D. Chen, H. Yu, X. Luo, Z. Li, W. Wang, R. Chen, D. Nie, and H. Yang, "FaaSwap: SLO-Aware, GPU-Efficient Serverless Inference via Model Swapping," *CoRR*, vol. abs/2306.03622, 2023.

[45] J. Jeong, S. Baek, and J. Ahn, "Fast and efficient model serving using multi-gpus with direct-host-access," in *EuroSys*. ACM, 2023, pp. 249–265.

[46] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 427–444. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/klimovic

[47] F. Romero, G. I. Chaudhry, I. n. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$t: A transparent auto-scaling cache for serverless applications," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 122137. [Online]. Available: https://doi.org/10.1145/3472883.3486974

[48] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: stateful functions-as-a-service," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 24382452, jul 2020. [Online]. Available: https://doi.org/10.14778/3407790.3407836

[49] Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 691707. [Online]. Available: https://doi.org/10.1145/3477132.3483541

[50] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "SONIC: Application-aware data passing for chained serverless applications," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 285–301. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/mahgoub

[51] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 303–320. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/mahgoub

[52] M. Abdi, S. Ginzburg, X. C. Lin, J. Faleiro, G. I. Chaudhry, I. Goiri, R. Bianchini, D. S. Berger, and R. Fonseca, "Palette load balancing: Locality hints for serverless functions," in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 365380. [Online]. Available: https://doi.org/10.1145/3552326.3567496

[53] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX

Association, Apr. 2023, pp. 1489–1504. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/yu

[54] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for Transformer-Based generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 521–538. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/yu

[55] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, and I. Stoica, "AlpaServe: Statistical multiplexing with model parallelism for deep learning serving," in *OSDI*. USENIX Association, 2023, pp. 663–679.

[56] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *SOSP*. ACM, 2023, pp. 611–626.

[57] P. Patel, E. Choukse, C. Zhang, igo Goiri, A. Shah, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative LLM inference using phase splitting," 2023.

[58] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, and Y. He, "Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22. IEEE Press, 2022.

[59] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Re, I. Stoica, and C. Zhang, "FlexGen: High-throughput generative inference of large language models with a single GPU," in *Proceedings of the 40th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202. PMLR, 23–29 Jul 2023, pp. 31 094–31 116. [Online]. Available: https://proceedings.mlr.press/v202/sheng23a.html

[60] S. Mehta, R. Bhagwan, R. Kumar, C. Bansal, C. Maddila, B. Ashok, S. Asthana, C. Bird, and A. Kumar, "Rex: Preventing bugs and misconfiguration in large services using correlated change analysis," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, 2020, pp. 435–448.

[61] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing configuration changes in context to prevent production failures," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020, pp. 735–751.

[62] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013, pp. 244–259.

[63] J. Toman and D. Grossman, "Staccato: A bug finder for dynamic configuration updates," in *Proceedings of the 30th European Conference on Object-Oriented Programming*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[64] T. Wang, Z. Jia, S. Li, S. Zheng, Y. Yu, E. Xu, S. Peng, and X. Liao, "Understanding and detecting on-the-fly configuration bugs," in *Proceedings of the 45th International Conference on Software Engineering*, 2023.

[65] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: Exploiting system environment and correlation information for misconfiguration detection," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 687–700.

[66] M. Santolucito, E. Zhai, and R. Piskac, "Probabilistic automated language learning for configuration files," in *Proceedings of the Computer Aided Verification: 28th International Conference*. Springer, 2016, pp. 80–87.

[67] Y. Zhou, W. Zhan, Z. Li, T. Han, T. Chen, and H. Gall, "Drive: Dockerfile rule mining and violation detection," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–23, 2023.

[68] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac, "Synthesizing configuration file specifications with association rule learning," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–20, 2017.

[69] X. Lian, Y. Chen, R. Cheng, J. Huang, P. Thakkar, and T. Xu, "Configuration validation with large language models," *arXiv preprint arXiv:2310.09690*, 2023.

[70] (2020) Apache OpenWhisk. Apache Software Foundation. [Online]. Available: https://openwhisk.apache.org

[71] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," *Elastic*, vol. 60, 2016.

[72] K. Kritikos and P. Skrzypek, "A review of serverless frameworks," in *2018 IEEE/ACM UCC Companion*. IEEE, 2018.

[73] Kubernetes. [Online]. Available: https://kubernetes.io

[74] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *MICRO, Proc.*, 2019.

[75] P. Brucker, *Scheduling algorithms*. Springer, 2007.

[76] P. Zuk and K. Rzadca, "Scheduling methods to reduce response latency of function as a service," *arXiv preprint arXiv:0000.00000*, 2020.

[77] C. Chekuri and S. Khanna, "On multi-dimensional packing problems," in *SODA, Proc.*, 1999.

[78] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *EuroSys*. ACM, 2015.

[79] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand *et al.*, "Autopilot: workload autoscaling at Google," in *Euro-Sys, Proc.*, 2020.

[80] M. R. Garey and D. S. Johnson, *Computers and intractability*, 1979, vol. 174.

[81] H. Zhao and R. Sakellariou, "An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm," in *Euro-Par*. Springer, 2003.

[82] L. F. Bittencourt, R. Sakellariou, and E. R. Madeira, "DAG scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm," in *PDP 2010*. IEEE, 2010.

[83] J. Wilkes, "Google cluster-usage traces v3," Google Inc., Mountain View, CA, USA, Technical Report, Apr. 2020, posted at https://github.com/google/cluster-data/blob/master/ClusterData2019.md.

[84] "Serverless endpoints for leading open-source models," https://www.together.ai/products#inference, 2024, accessed on 2024-06-02.

[85] "Custom LLMs," https://deepinfra.com/docs/advanced/custom_llms, 2024, accessed on 2024-06-02.

[86] "Run AI with an API," https://replicate.com/, 2024, accessed on 2024-06-02.

[87] "Your data. your AI. your future." https://www.databricks.com/, 2024, accessed on 2024-06-02.

[88] "Generative AI tailored to you," https://fireworks.ai/, 2024, accessed on 2024-06-02.

[89] "The leading enterprise AI platform," https://cohere.com/, 2024, accessed on 2024-06-02.

[90] S. Gugger, L. Debut, T. Wolf, P. Schmid, Z. Mueller, S. Mangrulkar, M. Sun, and B. Bossan, "Accelerate: Training and inference at scale made simple, efficient and adaptable." https://github.com/huggingface/accelerate, 2022.

[91] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[92] xAI, "Grok-1," 2024, accessed: 2024-05-31. [Online]. Available: https://github.com/xai-org/grok-1

[93] T. M. R. Team, "Introducing DBRX: A new state-of-the-art open LLM," *Mosaic AI Research*, March 2024, accessed: 2024-05-31. [Online]. Available: https://www.databricks.com/blog/introducing-dbrx-new-state-art-open-llm

[94] M. A. Team, "Better, faster, stronger," Mistral AI News, April 2024, accessed: 2024-05-31. [Online]. Available: https://mistral.ai/news/mixtral-8x22b/

[95] A. T. Blog, https://www.anyscale.com/blog/loading-llama-2-70b-20x-faster-with-anyscale-endpoints, 2023, accessed on 2024-01-22.

[96] Amazon, "Serverless inference - minimizing cold starts," https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints.html, 2023, accessed on 2024-01-22.

[97] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up," in *OSDI*. USENIX Association, 2020, pp. 443–462.

[98] W. Bai, S. S. Abdeen, A. Agrawal, K. K. Attre, P. Bahl, A. Bhagat, G. Bhaskara, T. Brokhman, L. Cao, A. Cheema, R. Chow, J. Cohen, M. Elhaddad, V. Ette, I. Figlin, D. Firestone, M. George, I. German, L. Ghai, E. Green, A. Greenberg, M. Gupta, R. Haagens, M. Hendel,

R. Howlader, N. John, J. Johnstone, T. Jolly, G. Kramer, D. Kruse, A. Kumar, E. Lan, I. Lee, A. Levy, M. Lipshteyn, X. Liu, C. Liu, G. Lu, Y. Lu, X. Lu, V. Makhervaks, U. Malashanka, D. A. Maltz, I. Marinos, R. Mehta, S. Murthi, A. Namdhari, A. Ogus, J. Padhye, M. Pandya, D. Phillips, A. Power, S. Puri, S. Raindel, J. Rhee, A. Russo, M. Sah, A. Sheriff, C. Sparacino, A. Srivastava, W. Sun, N. Swanson, F. Tian, L. Tomczyk, V. Vadlamuri, A. Wolman, Y. Xie, J. Yom, L. Yuan, Y. Zhang, and B. Zill, "Empowering azure storage with RDMA," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 49–67. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/bai

[99] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan *et al.*, "When cloud storage meets {rdma}," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 519–533.

[100] "Amazon elasticache pricing," https://aws.amazon.com/elasticache/pricing/?nc=sn&loc=5, Amazon Web Services, accessed: 2024-05-31.

[101] NVIDIA, "NVIDIA DGX H100," https://resources.nvidia.com/en-us-dgx-systems/ai-enterprise-dgx, 2023, accessed on 2024-01-22.

[102] PyTorch, "torch.load," https://pytorch.org/docs/stable/generated/torch.load.html, 2023, accessed on 2024-01-22.

[103] TensorFlow, "tf.saved_model.load," https://www.tensorflow.org/api_docs/python/tf/saved_model/load, 2023, accessed on 2024-01-22.

[104] O. Runtime, "API Reference, onnxruntime.backend.prepare," https://onnxruntime.ai/docs/api/python/api_summary.html, 2023, accessed on 2024-01-22.

[105] H. Zhang, Y. Tang, A. Khandelwal, and I. Stoica, "SHEPHERD: Serving DNNs in the Wild," in *NSDI*. USENIX Association, 2023, pp. 787–808.

[106] TensorFlow, "TensorFlow Documentation: Using the Saved Model Format," https://www.tensorflow.org/guide/saved_model, 2023, accessed on 2024-01-22.

[107] PyTorch, "PyTorch Documentation: Saving and Loading Models," https://pytorch.org/tutorials/beginner/saving_loading_models.html, 2023, accessed on 2024-01-22.

[108] Amazon, "AWS S3," https://aws.amazon.com/s3/, 2023, accessed on 2024-01-22.

[109] D. Shukla, M. Sivathanu, S. Viswanatha, B. Gulavani, R. Nehme, A. Agrawal, C. Chen, N. Kwatra, R. Ramjee, P. Sharma, A. Katiyar, V. Modi, V. Sharma, A. Singh, S. Singhal, K. Welankar, L. Xun, R. Anupindi, K. Elangovan, H. Rahman, Z. Lin, R. Seetharaman, C. Xu, E. Ailijiang, S. Krishnappa, and M. Russinovich, "Singularity: Planet-scale, preemptive and elastic scheduling of AI workloads," 2022.

[110] F. Strati, S. Mcallister, A. Phanishayee, J. Tarnawski, and A. Klimovic, "Déjàvu: Kv-cache streaming for fast, fault-tolerant generative LLM serving," 2024.

[111] C. N. C. Foundation, "etcd," https://etcd.io, 2023, accessed on 2024-01-22.

[112] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. USA: USENIX Association, 2010, p. 11.

[113] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, C. Hesse, and J. Schulman, "Training verifiers to solve math word problems," *CoRR*, vol. abs/2110.14168, 2021.

[114] M. Xu, https://huggingface.co/datasets/shibing624/sharegpt_gpt4, 2023.

[115] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: Warming serverless functions better with heterogeneity," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 753–767.

[116] M. M. Imran, P. Chatterjee, and K. Damevski, "Uncovering the causes of emotions in software developer communication using zero-shot llms," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[117] Z. Xie, Y. Chen, C. Zhi, S. Deng, and J. Yin, "Chatunitest: a chatgpt-based automated unit test generation tool," *arXiv preprint arXiv:2305.04764*, 2023.

[118] X. Yin, C. Ni, and S. Wang, "Multitask-based evaluation of open-source llm on software vulnerability," *IEEE Transactions on Software Engineering*, 2024.

[119] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–38, 2024.

[120] "Chain-of-thought prompting," https://learnprompting.org/docs/intermediate/chain_of_thought, 2024.

[121] Z. Chu, J. Chen, Q. Chen, W. Yu, T. He, H. Wang, W. Peng, M. Liu, B. Qin, and T. Liu, "A survey of chain of thought reasoning: Advances, frontiers and future," *arXiv preprint arXiv:2309.15402*, 2023.

[122] J. Li, G. Li, Y. Li, and Z. Jin, "Structured chain-of-thought prompting for code generation," *ACM Transactions on Software Engineering and Methodology*, 2023.

[123] "Aws serverless application repository," https://serverlessrepo.aws.amazon.com/applications, 2024.

[124] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[125] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977.

[126] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 117–132.

[127] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based online configuration-error detection," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, 2011, pp. 28–28.

[128] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM SIGMOD Record*, vol. 29, no. 2, pp. 1–12, 2000.

[129] S. Li, W. Li, X. Liao, S. Peng, S. Zhou, Z. Jia, and T. Wang, "Confvd: System reactions analysis and evaluation through misconfiguration injection," *IEEE Transactions on Reliability*, vol. 67, no. 4, pp. 1393–1405, 2018.

[130] W. Li, Z. Jia, S. Li, Y. Zhang, T. Wang, E. Xu, J. Wang, and X. Liao, "Challenges and opportunities: An in-depth empirical study on configuration error injection testing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 478–490.

[131] F. Hadadi, Q. Xu, D. Bianculli, and L. Briand, "Anomaly detection on unstable logs with gpt models," *arXiv preprint arXiv:2406.07467*, 2024.

[132] Y. Chen, H. Xie, M. Ma, Y. Kang, X. Gao, L. Shi, Y. Cao, X. Gao, H. Fan, M. Wen *et al.*, "Automatic root cause analysis via large language models for cloud incidents," in *Proceedings of the 19th European Conference on Computer Systems*, 2024, pp. 674–688.

[133] "Chatbot arena llm leaderboard: Community-driven evaluation for best llm and ai chatbots," https://lmarena.ai/, 2024.

[134] D. Tomaras, I. Boutsis, and V. Kalogeraki, "Modeling and predicting bike demand in large city situations," in *2018 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 2018, pp. 1–10.

[135] "Crowdalert application." [Online]. Available: http://crowdalert.aueb.gr

[136] "Waze." [Online]. Available: https://www.waze.com/

[137] D. Kinane *et al.*, "Intelligent synthesis and real-time response using massive streaming of heterogeneous data (insight) and its anticipated effect on intelligent transport systems (its) in dublin city, ireland," *ITS, Dresden, Germany*, 2014.

[138] V. A. Leal Sobral *et al.*, "A cloud-based data storage and visualization tool for smart city iot: Flood warning as an example application," *Smart Cities*, vol. 6, no. 3, pp. 1416–1434, 2023.

[139] G. Ortiz *et al.*, "A microservice architecture for real-time iot data processing: A reusable web of things approach for smart ports," *Computer Standards & Interfaces*, vol. 81, p. 103604, 2022.

[140] T. Elgamal, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *SEC*, 2018.

[141] J. Jiang *et al.*, "Towards demystifying serverless machine learning training," in *ICMD*, 2021, pp. 857–871.

[142] H. El Hafyani *et al.*, "A microservices based architecture for implementing and automating etl data pipelines for mobile crowdsensing applications," in *Big Data*. IEEE, 2021, pp. 5909–5911.

[143] F. Romero *et al.*, "Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines," in *ACM SCC*, 2021, pp. 1–17.

[144] A. Peri, M. Tsenos, and V. Kalogeraki, "Orchestrating the execution of serverless functions in hybrid clouds," in *2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2023, pp. 139–144.

[145] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Middleware*, 2020, pp. 1–13.

[146] E. Oakes *et al.*, "$SOCK$: Rapid task provisioning with serverless-optimized containers," in *USENIX ATC*, 2018, pp. 57–70.

[147] M. Shahrad and et. al, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *USENIX*, 2020, pp. 205–218.

[148] A. Fuerst and P. Sharma, "Faascache: keeping serverless computing alive with greedy-dual caching," in *ASPLOS*, 2021, pp. 386–400.

[149] V. Cardellini, E. Casalicchio, and L. Silvestri, "Service level provisioning for cloud-based applications service level provisioning for cloud-based applications," in *Grid and Cloud Computing: Concepts, Methodologies, Tools and Applications*. IGI Global, 2012, pp. 1479–1500.

[150] S. Hendrickson *et al.*, "Serverless computation with $OpenLambda$," in *HotCloud 16*, 2016.

[151] Z. Zeng and et. al, "Comparing stars: On approximating graph edit distance," *VLDB*, vol. 2, no. 1, pp. 25–36, 2009.

[152] A. Pérez and et. al, "Serverless computing for container-based architectures," *Future Generation Computer Systems*, vol. 83, pp. 50–59, 2018.

[153] J. Cheng, D. T. A. Nguyen, L. Wang, D. T. Nguyen, and V. K. Bhargava, "A bandit approach to online pricing for heterogeneous edge resource allocation," *arXiv preprint arXiv:2302.06953*, 2023.

[154] Y. Peng *et al.*, "A generic communication scheduler for distributed dnn training acceleration," in *SOSP*, 2019, pp. 16–29.

[155] M. Bilal *et al.*, "With great freedom comes great opportunity: Rethinking resource allocation for serverless functions," in *EuroSys*, 2023, pp. 381–397.

[156] T. Donkers *et al.*, "Sequential user-based recurrent neural network recommendations," in *RecSys*, 2017, pp. 152–160.

[157] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," *NeurIPS*, vol. 25, pp. 1097–1105, 2012.

[158] R. D. Luce, "The choice axiom after twenty years," *Journal of mathematical psychology*, vol. 15, no. 3, pp. 215–233, 1977.

[159] Z. Zhang and M. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels," *NeurIPS*, vol. 31, 2018.

[160] E. Gordon-Rodriguez *et al.*, "Uses and abuses of the cross-entropy loss: Case studies in modern deep learning," 2020.

[161] F. Cao *et al.*, "Deconvolutional neural network for image super-resolution," *Neural Networks*, vol. 132, pp. 394–404, 2020.

[162] M. Magill *et al.*, "Neural networks trained to solve differential equations learn general representations," *NeurIPS*, vol. 31, 2018.

[163] H. Yu and et. al, "Faasrank: Learning to schedule functions in serverless platforms," in *ACSOS*. IEEE, 2021, pp. 31–40.

[164] S. Kornblith and et. al, "Similarity of neural network representations revisited," in *ICML*. PMLR, 2019, pp. 3519–3529.

[165] N. Zacheilas *et al.*, "Dione: Profiling spark applications exploiting graph similarity," in *Big Data*. IEEE, 2017, pp. 389–394.

[166] J. Xin *et al.*, "Locat: Low-overhead online configuration auto-tuning of spark sql applications," in *SIGMOD*, 2022, pp. 674–684.

[167] M. Obetz, S. Patterson, and A. L. Milanova, "Static call graph construction in aws lambda serverless applications." in *HotCloud*, 2019.

[168] H. Bunke *et al.*, "A graph distance metric based on the maximal common subgraph," *Pattern recognition letters*, vol. 19, no. 3-4, pp. 255–259, 1998.

[169] F. Lai *et al.*, "$ModelKeeper$: Accelerating DNN training via automated training warmup," in *NSDI '23*, 2023, pp. 769–785.

[170] M. Tsenos *et al.*, "Amesos: a scalable and elastic framework for latency sensitive streaming pipelines," in *DEBS*, 2022.

[171] OpenFaas, "Function chaining." [Online]. Available: https://ericstoekl.github.io/faas/developer/chaining_functions/\#server-side-access-via-gateway

[172] V. Salis and et. al, "Pycg: Practical call graph generation in python," in *ICSE*. IEEE, 2021, pp. 1646–1657.

[173] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *CLOUD*. IEEE, 2019, pp. 502–504.

[174] I. Müller *et al.*, "Lambada: Interactive data analytics on cold data using serverless cloud infrastructure," in *SIGMOD*, 2020.

[175] J. Manner *et al.*, "Optimizing cloud function configuration via local simulations," in *CLOUD*. IEEE, 2021, pp. 168–178.

[176] Y. K. Kim and et. al, "Automated fine-grained cpu cap control in serverless computing platform," *TPDS*, vol. 31, no. 10, pp. 2289–2301, 2020.

[177] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *USENIX*, 2020, pp. 419–433.

[178] X. Fu *et al.*, "Edgewise: a better stream processing engine for the edge," in *USENIX ATC*, 2019.