Autoscaling in Serverless Platforms via Online Learning with Convergence Guarantees

Abednego Wamuhindo Kambale, Jonatha Anselmi, Danilo Ardagna, Bruno Gaujal

***Abstract*—As the adoption of serverless computing platforms continue to grow, designing autoscaling policies that strike the right balance between energy efficiency and user-perceived performance has become a central challenge. In this work, we propose an online learning algorithm with theoretical convergence guarantees that dynamically tunes control parameters in a serverless autoscaling environment. The proposed algorithm, grounded in stochastic gradient descent, learns online—during the actual operation of the platform—the optimal values of three key control parameters: (i) the target stock size of pre-warmed (idle) functions, (ii) the threshold triggering provisioning actions, and (iii) the expiration rate of idle resources. We prove that, under Markovian dynamics, the algorithm converges to the parameter set that minimizes a cost function capturing the tradeoff between energy consumption and response latency. In addition, we demonstrate that its structure naturally supports parallelization, significantly accelerating convergence.**

Extensive numerical experiments show that our method outperforms existing baselines, including recent deep learning-based approaches, even under non-Markovian settings—highlighting both its robustness and practical viability for next-generation serverless infrastructures.

***Index Terms*—Autoscaling, serverless functions, online learning, stochastic gradient descent.**

I. Introduction

Serverless computing is a modern approach to building and running applications in the cloud where developers only need to focus on writing and deploying code as small, independent functions. These functions are executed in managed environments without the need to handle servers or infrastructure, which is why this model is often referred to as Function-as-a-Service (FaaS). This execution model makes development faster and more efficient, as cloud providers take care of all the operational overhead [1].

Due to its flexibility and ease of use, many major cloud platforms have introduced their own serverless services. For example, Amazon Lambda [2], IBM Cloud Functions based on OpenWhisk [3], and Microsoft Azure Functions [4] are widely used to support real-world applications ranging from video processing and machine learning, to data analytics and distributed computing tasks [5], [6]. The global FaaS market size was valued at USD 5.51 billion in 2024 and is projected to reach from USD 7.17 billion in 2025 to USD 59.24 billion by 2033, growing at a CAGR of 30.2% [7].

A. W. Kambale, and D. Ardagna are with Politecnico di Milano. Milan, Italy. Email: abednegowamuhindo.kambale@polimi.it; danilo.ardagna@polimi.it

J. Anselmi, B. Gaujal are with Inria of Univ. Grenoble Alpes, Grenoble, France. Email: jonatha.anselmi@inria.fr; bruno.gaujal@inria.fr

Despite its advantages, serverless computing comes with some challenges [1]. One of the most common issues is the function initialization delay, commonly referred to as the *cold start*. This delay happens because the platform needs to prepare a secure, isolated environment before the function can actually run. In contrast, a *warm start* reuses an already-initialized environment, which is much faster. Studies such as [8] show that cold starts incur significant computational and memory overhead, which degrades user experience and increases energy and operational costs. To overcome these limitations, many studies [9]–[21] have explored autoscaling strategies for serverless platforms. Autoscaling is the process of automatically adjusting the number of function instances based on the current workload, such as how many requests are coming in or how many functions are running at the same time. This mechanism helps ensure that applications remain responsive and cost-effective, even when demand changes.

A. Contribution

While autoscaling itself is a well-established mechanism in serverless platforms, *how* to set the parameters that govern it remains largely open: in practice these parameters are fixed by heuristics or hand-tuning, and the few learning-based methods either optimize a single parameter or rely on reinforcement learning (RL) without convergence guarantees. The core problem we address in this paper is therefore the following: *how can a serverless platform learn, online and while it is running, the optimal values of the multiple control parameters that jointly govern its autoscaling policy, using only the transient (non-stationary) observations available at runtime, and with provable convergence to an optimal operating point?* This is challenging because the controller never observes the system in its stationary regime, must optimize several interacting parameters simultaneously, and must do so without a model of the workload. Our goal is to develop an online algorithm that learns the optimal parameters of a generalized version of autoscaling policies [9], [21] commonly used in the literature and implemented in existing serverless infrastructures [2]–[4]. Here, optimality is defined in terms of achieving a balance between user-perceived performance delay and energy consumption. Roughly speaking, the considered autoscaling policy works as follows:

> *"Activate a certain number of function instances if the number of active idle instances belongs to some interval and deactivate a function instance if it remains idle after a certain amount of time. "*

Note that this policy aims to achieve improved energy–performance tradeoffs by controlling at the same time (i) the two control parameters that, inspired by inventory control theory, are interpreted as provisioning thresholds, and (ii) the mean expiration time of idle functions, i.e., the duration an idle function instance remains active before being turned off.

The necessity of jointly controlling these three parameters can be illustrated through a representative scenario. Consider a function serving a bursty workload, such as an image-processing service subject to sudden surges in its request rate. A single provisioning parameter that fixes the target size of the warm pool (θ_{stock}), as learned in [21], determines how many instances are pre-warmed but cannot govern *when* provisioning should be initiated: under bursty arrivals, new instances should begin warming before the pool of idle instances is depleted, which requires a distinct reorder threshold θ_{idle}. Such a parameter likewise cannot govern *how long* an instance should remain warm once a surge subsides: retaining instances for too long incurs unnecessary idle-energy costs, whereas releasing them prematurely increases the cold-start rate of the subsequent surge, a trade-off regulated by the expiration rate θ_{exp}. Since these quantities interact, optimizing any one in isolation yields a suboptimal operating cost, which motivates the joint optimization adopted in this work.

To learn optimally the parameter vector that defines the considered autoscaling policy, we develop a new online learning iterative algorithm, which is *general* in nature and applies to a broader class of problems beyond the autoscaling context. It builds upon the *single-parameter* algorithm introduced in [21], which is a version of the classic Kiefer-Wolfowitz algorithm [22] adapted to the case where only transient (non-stationary) samples of the dynamic system random environment can be observed at any point in time. The main difference is that in this paper we need to consider a *multidimensional* setting, as we aim at optimizing stochastically over multiple parameters at the same time. We provide convergence guarantees for our algorithm and, although these are established under Markovian assumptions, numerical simulations show that it performs robustly even when they do not hold. Empirically, our method substantially outperforms both the single-parameter predecessor [21] and the recent RL-based LSTM-PPO scheme [16] across all evaluated arrival rates, including realistic non-Markovian settings with Pareto-distributed service times [23]. The improvement over the single-parameter baseline exceeds 100%; detailed per-scenario gains are reported in Section VI. In summary, this paper makes the following contributions:

- We develop a new online algorithm which is a generalization of the learning scheme in [21] to a multidimensional setting, allowing for the joint optimization of a vector of control parameters. In addition, we provide a parallel implementation, which accelerates the learning process without compromising accuracy.
- We establish theoretical convergence guarantees for the proposed algorithm. Under mild regularity and Markovian assumptions, the sequence of parameter vectors generated by the algorithm converges to a local minimum of the objective function.
- We have implemented the proposed algorithm, extending the SimFaaS simulation framework, and published it at https://github.com/Wamuhindo/NSGD for reproducibility purposes. Extensive numerical experiments show that our approach consistently outperforms state-of-the-art methods such as the LSTM-PPO scheme recently developed in [16], even under realistic non-Markovian assumptions.

B. Organization

This work is organized as follows. Section II presents a review of the literature on autoscaling technologies in serverless computing and positions our paper. Section III describes the serverless infrastructure and the state model of function instances. In Section IV, we introduce the autoscaling policy considered throughout the paper. Section V presents our main theoretical contributions: we first develop a general stochastic optimization framework, then introduce the online learning algorithm, prove convergence guarantees, describe its parallel implementation, and apply it to the autoscaling setting. Section VI reports extensive experimental results about cost efficiency, robustness, and parallelism of our approach. Section VII compares our approach to state-of-the-art baselines. Finally, Section VIII concludes the paper and outlines directions for future research.

II. Related Work

In this section, we provide a comprehensive review of the background and relevant literature on serverless autoscaling. Subsequently, we discuss how our work positions itself within this context and delineates its salient distinctions from previous approaches.

A. Review of Autoscaling in Serverless Computing

A comprehensive review by Tari et al. [9] outlines three major types of autoscaling strategies: framework-based, model-based, and machine learning-based approaches. Each strategy aims to optimize performance and efficiency by scaling function instances in response to demand patterns. Common mechanisms include scale-per-request, which creates a new instance per request, and concurrency-based scaling, which adjusts based on the number of parallel executions. Other adaptive methods use dynamic thresholds that respond to real-time metrics such as latency, CPU utilization, or request rate, enabling finer control under varying workloads. Mahmoudi et al. [10] introduced a performance modeling approach aligned with modern serverless platforms, focusing on auto-scaling mechanisms driven by concurrency levels and request rates. Their method enables both cloud providers and developers to estimate the performance and cost trade-offs of various deployment configurations. With minimal data collection, the proposed model can effectively capture workload-specific characteristics and predict steady-state performance indicators such as response time.

Some studies have explored reinforcement learning (RL) techniques to address the autoscaling challenge in serverless computing. For example, [13] introduced an RL-based adaptive auto-scaling approach for delay-sensitive serverless services. The method learns resource-performance profiles for each

service to guide dynamic scaling decisions. It combines horizontal and vertical scaling and, unlike threshold-based methods, it balances quality of service with resource efficiency. Experiments show that the approach meets response delay requirements while reducing the cost. Similarly, [14] proposed RL-based auto-scaling mechanisms to tackle the challenges of resource provisioning under Service Level Agreement (SLA) constraints. By monitoring system metrics sutch as CPU usage, memory consumption, and request throughput, the study implements and evaluates Q-learning, DynaQ+, and Deep Q-learning agents. Built on top of the Kubeless framework, their approach enables adaptive scaling to dynamic workloads while promoting efficient resource utilization. Benedetti et al. [15] introduced a Q-learning-based autoscaling method for OpenFaaS, targeting resource-constrained edge environments. Their approach learns a scaling policy based on CPU utilization that reduces service latency with limited training iterations.

Recently, [16] proposed LSTM-PPO, an approach that integrates Long Short-term Memory (LSTM) with Proximal Policy Optimization (PPO), a state-of-the-art RL algorithm. The approach analyzes partial observability and sequential dependence of autoscaling actions and finds a balance between conflicting cloud service providers and user objectives. They proposed a POMDP model that captures function metrics such as CPU and memory utilization, function replicas, average execution time, and throughput ratio, as partial observations, and formulates the scaling problem. The LSTM helps capture the system dynamics well. They deploy the proposed agent to the OpenFaaS framework and utilize open-source function invocation traces from a production environment to perform experiments with a matrix multiplication function.

Beyond the single-function setting considered above, another line of work targets *multi-tenant* serverless platforms, where heterogeneous functions from many users compete for shared resources. Qiu et al. [24] show that single-agent RL fails to converge in this setting, and propose a multi-agent PPO formulation (MA-PPO); SIMPPO [25] extends this into a scalable, incremental multi-agent framework that adapts as functions are added or removed. AWARE [26] instead targets the safe deployment of such RL controllers in production, using meta-learning and bootstrapping to limit the risk of unsafe online exploration.

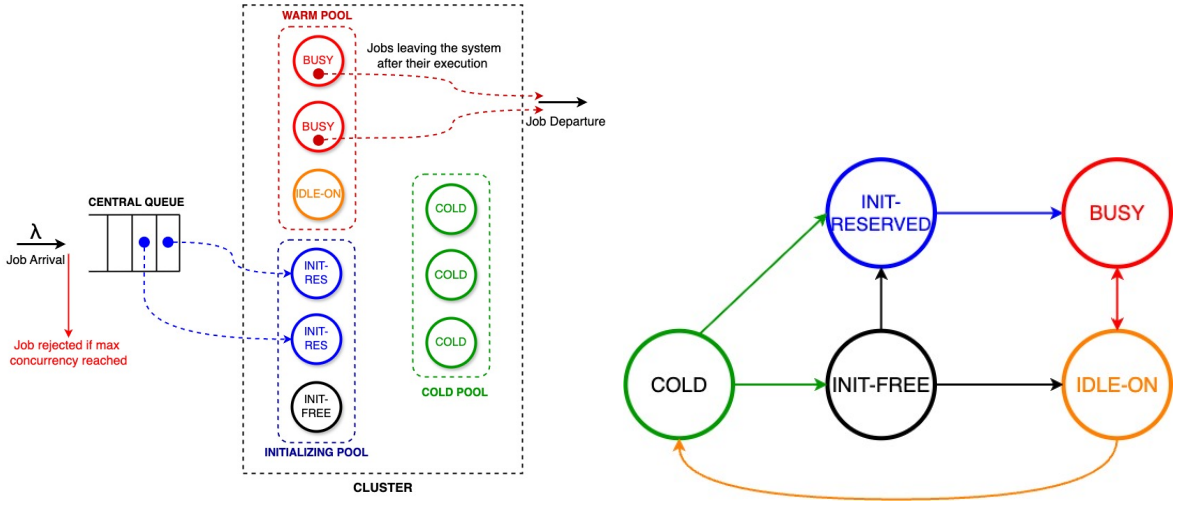
Other studies have explored different optimization techniques to reduce the cold start issue and ensure efficient autoscaling. Authors in [18] introduced *FaaShark*, a cloud-native, end-to-end network traffic analysis system built on a serverless computing platform. It addresses the cold start latency issue, by employing a gradient-based cold start optimization algorithm. This algorithm continuously monitors recent inference request patterns and predicts workload trends using a slope model, enabling pre-warming of containers ahead of increased demand. This pre-warming process helps reduce cold start rates and ensures efficient management of dynamic workloads. Pan et al. [19] proposed a framework that combines pre-warming and automatic resource allocation to improve the performance of serverless workflows by reducing cold start delays and making better use of computing resources. For auto-scaling, the system uses a gradient-based algorithm that monitors trends in function calls to decide when to scale up (pre-warm containers) or scale down (terminate containers). The system lowers the cold start rate and reduces overall costs by adjusting resources based on actual workload. Wen et al. [20] proposed an online autoscaling framework that jointly optimizes parallelism and resource configuration for each step in serverless workflows. Their method uses a heuristic based on the critical path of the workflow to adjust resources while meeting service level objectives and minimizing costs. They also introduced a configuration-aware pre-warming mechanism that predicts future resource needs and launches the required number of function instances in advance to mitigate cold starts and improve autoscaling responsiveness.

A more recent work [21] proposed a stochastic gradient descent algorithm of the Kiefer-Wolfowitz type that runs over a single state evolution to optimize server activation decisions. The algorithm estimates the optimal number of servers to spawn upon job arrival, aiming to minimize a cost function that balances energy consumption and user-perceived performance. The iterative process drives the optimal number of servers toward its optimal value, and improves responsiveness by proactively maintaining idle servers, which is beneficial in scenarios where minimizing cold start latency is critical.

B. Positioning of Our Work

The present work proposes $NSGD$, an autoscaling framework based on a non-stationary gradient descent optimization algorithm. Unlike the approach in [21], which learns a single parameter θ, $NSGD$ considers a vectorized setting with three control parameters: θ_{stock}, θ_{idle}, and θ_{exp}, which define the scale-up and scale-down rule. These parameters directly impact system cost and are jointly optimized to minimize a cost function that balances energy consumption and application performance. Furthermore, $NSGD$ is evaluated under realistic, non-Markovian settings using a state-of-the-art simulator that has been empirically validated against real-world AWS Lambda metrics. While RL-based methods, such as the one proposed by [16], are known to perform well in practice, they typically lack convergence guarantees. In contrast, $NSGD$ offers theoretical convergence guarantees. Moreover, it operates as a fine-grained controller by making autoscaling decisions on a per-request basis, whereas most RL-based solutions are time-triggered and act only at the end of fixed control intervals, which in some cases may result in high costs.

A related line of work models serverless and cloud resource management as a *multi-agent* problem, in which heterogeneous functions or users contend for shared resources. SIMPPO [25] and MA-PPO [24] learn per-function policies with multi-agent reinforcement learning, jointly tuning per-container CPU/memory limits (vertical scaling) and the number of containers (horizontal scaling) to meet per-function SLOs; their main contribution is the coordination of many co-located agents. Our setting differs in two respects. First, we adopt the public-FaaS tenant perspective, in which a function per-container CPU/memory is fixed at deployment time and not autoscaled at runtime [27], so the vertical actuator is unavailable; $NSGD$ controls only horizontal provisioning and instance lifecycle



(a) Snapshot of system at a given time

(b) State transitions for each function.

Figure 1: System snapshot at a given time (a) showing job arrival and the different function pools according to the state. Dashed arrows show job movements. (b) shows state transitions for a single function.

through $(\theta_{stock}, \theta_{idle}, \theta_{exp})$. Second, restricted to a single function and horizontal scaling, these methods reduce to a single-agent PPO controller of the same class as the interval-triggered LSTM-PPO baseline we outperform in Section VII-B, and likewise without convergence guarantees. Extending $NSGD$ to heterogeneous functions and vertical scaling is left for future work.

A complementary line within this multi-agent setting does provide theoretical guarantees: Mao et al. [28] model cloud resource management as a mean-field game and prove finite-time convergence of an actor–critic algorithm to a regularized Nash equilibrium. Our result has a different scope. We do not formulate autoscaling as a multi-agent game, nor do we learn a general RL policy. Instead, $NSGD$ is based on a central control at the system level and not at the user-level (as in [28]). It optimizes a structured autoscaling rule parametrized by $(\theta_{stock}, \theta_{idle}, \theta_{exp})$, with objective $c(\theta) = \mathbb{E}[C(X_\infty^\theta)]$. We prove almost-sure convergence of these parameters iterates to a stationary point of c. Hence, the two approaches are complementary: mean-field RL gives equilibrium guarantees for large interacting set of users, while $NSGD$ gives direct stochastic-optimization guarantees for an interpretable autoscaling controller.

III. System Description

We investigate a serverless computing environment in which computational jobs, arriving from an external source, are executed by stateless functions. Each function invocation corresponds to an instance, and we denote the total number of concurrently active instances by N. The quantity N represents the nominal service capacity allocated to a user. This capacity imposes an upper bound on the number of function instances a user may simultaneously execute. This limit ensures equitable resource distribution across users, and it is enforced by most serverless platforms, as discussed in [29].

Upon arrival, a job is admitted for execution if sufficient resources are available; otherwise, it is rejected once the concurrency limit is reached (see Figure 1a). The specific time at which a job starts its execution is governed by the operational state of the corresponding function. Each function operates among four primary states: *cold*, *initializing*, *busy*, and *idle-on*. A function is considered *idle-on* when it is fully initialized and capable of processing requests. A *busy* function is one that is actively processing a request. In contrast, a *cold* function is inactive and not yet initialized. A function in the *initializing* state is undergoing the transition from *cold* to either *busy*, if it has been assigned a request, or to *idle-on*, if it has not yet been assigned one. When a job is assigned to a function instance, it waits in a central queue as seen in Figure 1a, until the instance completes its initialization phase. Upon completing the processing of a request, the function transitions from *busy* to *idle-on* state, wherein it remains idle yet fully prepared to handle subsequent incoming requests. This mechanism ensures that newly arriving jobs encounter the function in a ready state, thereby facilitating immediate processing without additional initialization overhead. This overhead arises from the fact that, during initialization, the function remains unavailable for execution, as it performs startup operations such as loading required libraries and dependencies, or establishing essential connections, for instance, to a database. At this time, the job waits in a central queue until the function becomes ready.

Furthermore, we distinguish between two types of initializing functions: *init-free* and *init-reserved*. A function is referred to as *init-reserved* when it is explicitly assigned to a specific job – typically the one that triggered its activation – and begins executing that job immediately after completing its initialization. In contrast, *init-free* functions are not necessarily bound to any job. Their provisioning is governed by the system autoscaling policy. Once initialization is complete, a *init-free* function transitions to the *idle-on* state. However, if a job arrives while the function is still initializing and no *idle-on* function is available, it is reclassified as *init-reserved* and subsequently transitions to the *busy* state upon completing initialization. In the remainder of this work, we adopt the terms *init-free* and *init-reserved* instead of *initializing*, as these two states have

different implications for system performance and play distinct roles in the algorithm we propose. Figure 1b illustrates all possible state transitions of a function.

In serverless computing platforms, function lifecycle transitions are influenced by multiple interdependent factors, including request arrival rates, function-specific characteristics, and system-level parameters [30]. For example, the transition from the *cold* to *initializing* state is triggered by job arrivals and influenced by system metrics such as CPU and memory usage, or by the need to establish external connections (e.g., to a database). Similarly, the transition from *busy* to *idle-on* is governed by the function service time and overall system load. Modern commercial FaaS platforms like AWS Lambda define transitions from *idle-on* to *cold* based on configurable expiration policies that terminate idle functions after a timeout to optimize resource usage [29].

Direct analysis of these transitions on production systems is often constrained by limited observability and cost implications. Simulation, therefore, serves as a practical and effective alternative for understanding and optimizing FaaS systems. Several recent open-source serverless simulators [31]–[33] model these transitions and offer a way to analyse the performance of the serverless system without relying on production platforms. However, these simulators do not explicitly model the intermediate *initializing* state, and do not explicitly differentiate between *init-free* and *init-reserved* states. In this work, we extend the SimFaaS framework [31], previously employed in [29], by introducing explicit modeling of the *init-free* and *init-reserved* states. This enhancement is critical for accurately capturing the dynamics required by the proposed autoscaling algorithm. Given that SimFaaS has been empirically validated against real-world AWS Lambda metrics, it provides a reliable foundation for conducting the analyses in this study.

In our simulation model, lifecycle transitions are governed by separate processes: an arrival process controls incoming request dynamics, a warm service process governs transitions from *busy* to *idle-on*, and an expiration process determines transitions from *idle-on* to *cold*. Moreover, we introduce a cold start process to explicitly represent transitions from *init-free* to *idle-on* and from *init-reserved* to *busy*, which were previously unmodeled. Each of these processes can be configured as exponential, deterministic, or entirely custom, offering flexibility to closely mimic real-world platform behaviors and supporting rigorous policy development.

IV. Auto-scaling Policy

This section introduces an autoscaling policy designed to balance the trade-off between energy consumption and user-perceived performance in serverless computing environments. Both metrics are influenced by the current system state, necessitating a dynamic and state-aware control mechanism. A key challenge in this context is workload variability, which is characterized by sudden spikes, unpredictable drops, and irregular patterns in incoming requests. These fluctuations can lead to over-provisioning, resulting in resources and energy waste, or under-provisioning, leading to degraded user experience [1].

We introduce a dynamic autoscaling policy whose goal is to keep the number of active idle instances in the interval $[\theta_{idle}, \theta_{stock}]$ and deactivate a function instance if it remains idle after a certain amount of time ($1/\theta_{exp}$ in average). The parameters θ_{stock} and θ_{idle} influence the scale-up behavior of the system, while θ_{exp} regulates the scale-down or expiration of idle resources.

The scale-up strategy is inspired by classical inventory control models, specifically the (s, S)-type policies, where s represents the reorder point and S the order-up-to level. Kalymon [34] established the optimality of such policies under the assumption that demand follows a Markov process. Analogously, we define θ_{idle} as the rescaling threshold (i.e., similar to reorder point) and θ_{stock} as the quantity of function instances (*init-free*+*idle-on*) required to reach the desired scale-up-to level (i.e., similar to order-up-to level). Consequently, upon the arrival of a job request, the number of *init-free* function instances to spawn is determined by the difference between θ_{stock} and the current total number of *init-free* and *idle-on* instances in the system.

The scale-down strategy is simple and used in several serverless computing platforms; see, e.g., [35]–[37]. When a server becomes idle-on, it becomes cold after an *expiration time* if during such time the server received no job. We let this expiration time be $1/\theta_{exp}$ on average. Experiments in [31] showed that each cloud provider sets a default *expiration time*, which users can adjust based on their application needs. This highlights the importance of learning θ_{exp}, as it directly impacts the system operational cost.

Policy 1 summarizes the proposed autoscaling policy that is governed by θ_{stock}, θ_{idle}, and θ_{exp}.

Policy 1: The proposed autoscaling policy.

On *each arrival (with rate λ)* **do**
  if *#idle-on_functions = 0* ***and*** *#cold_functions > 0* **then**
    #init-reserved_functions := 1;
    #init-free_functions_to_spawn := $\pi_{\theta_{stock}}$;
    Initialize #init-reserved_functions;
    Initialize #init-free_functions_to_spawn;
  else if *#idle-on_functions > 0* **then**
    #idle-on_functions := #idle-on_functions - 1;
    if *#idle-on_functions* < $\pi_{\theta_{idle}}$ ***and*** *#cold_functions > 0* **then**
      #init-free_functions_to_spawn := $\pi_{\theta_{stock}}$ − (#idle-on_functions + #init-free_functions);
    Serve the request with an *idle-on* function;
    Initialize #init-free_functions_to_spawn;
  else
    Reject job
end
On *each expiration of an* idle-on *function (with rate θ_{exp})* **do**
  transition to *cold* the expired *idle-on* function;
end

Remark 1. *Note that θ_{stock} and θ_{idle} belong to $\mathbb{R}_+$. However,*

since the number of functions to be launched in the system must be an integer, we introduce the integer-valued random variables $\pi_{\theta_{stock}}$ and $\pi_{\theta_{idle}}$. Specifically, let $\pi_{\theta_{stock}} = Round(\theta_{stock}) \in \mathbb{N}$ such that $\mathbb{E}[\pi_{\theta_{stock}}] = \theta_{stock}$, and analogously, $\pi_{\theta_{idle}} = Round(\theta_{idle}) \in \mathbb{N}$ such that $\mathbb{E}[\pi_{\theta_{idle}}] = \theta_{idle}$. For the time being, the function Round$(\cdot)$ is treated as a black box producing integer outputs that depend on θ_{stock} or θ_{idle}; its exact construction is postponed to Section V since it must be defined on $\mathbb{R}$, be smooth (twice differentiable) and have no bias.

The autoscaling policy in Policy 1 is fundamentally governed by implicit scale-up and scale-down mechanisms, parameterized respectively by $\pi_{\theta_{stock}}$, $\pi_{\theta_{idle}}$, and θ_{exp}. The scale-up logic ensures that the system maintains, on average, a reserve of approximately θ_{stock} functions ready for use in the near future. Furthermore, this mechanism triggers the initialization of additional functions whenever the number of *idle-on* functions drops below a threshold, designed to stabilize around θ_{idle} over time. On the other hand, the scale-down behavior, regulated by θ_{exp}, is critical for minimizing resource overhead. Specifically, θ_{exp} defines the expiration rate at which a function, after transitioning from *busy* to *idle-on*, is eligible to transition to *cold* in the absence of any bound incoming requests.

Upon the arrival of a job request, and given that the control parameters θ_{stock} and θ_{idle} are supplied by the learning algorithm, the function scaling-up mechanism follows a structured decision-making process, as detailed below:

A1) Cold start allocation: if no *idle-on* functions exist but a pool of *cold* functions is available, the system uniformly samples one *cold* function to transition to the *init-reserved* state, and selects an additional $\pi_{\theta_{stock}}$ *cold* functions to transition to the *init-free* state. Meanwhile, the request waits in a central queue for the activation of its associated *init-reserved* function (lines 2–6).

A2) Warm start allocation and preemptive preparation: if one or more *idle-on* functions are available, the request is immediately sent to one of them, selected uniformly at random. To proactively mitigate potential future initialization delays, the system monitors *idle-on* functions. If their count drops below the threshold $\pi_{\theta_{idle}}$, the system initiates additional *cold* to *init-free* transitions in response to new incoming jobs. This preemptive action aims to ensure that the combined number of *init-free* and *idle-on* functions reaches the target level θ_{stock} (lines 7–12).

A3) Resource saturation: if no *idle-on* or *init-free* functions are available, and no *cold* functions exist, all computational resources are saturated, and the incoming job is rejected and the state remains unchanged (lines 13–14).

Each of the three control parameters θ_{stock}, θ_{idle}, and θ_{exp} have a direct impact on the system cost. A more detailed analysis of this impact is provided in Appendix C. By jointly learning θ_{stock}, θ_{idle}, and θ_{exp}, an effective autoscaling policy can be formulated, thereby significantly reducing system costs. Indeed, such a policy not only dynamically adapts to workload fluctuations but also proactively provisions cold functions in anticipation of future demand, thereby efficiently releasing idle resources. This results in improved system responsiveness and energy efficiency as shown in Section VI. In contrast to the approach in [21], which learns only θ_{stock} and triggers scaling-up only when no *idle-on* instances are available – thus relying exclusively on warm starts without proactive provisioning – the proposed autoscaling policy incorporates preemptive readiness by learning θ_{stock}, θ_{idle} and expiration behavior (i.e., θ_{exp}). Furthermore, unlike traditional autoscaling techniques that apply coarse-grained control (e.g., periodic scale-up and scale-down rules) as in [16], our method adopts a fine-grained scale-per-request strategy. The learned control parameters are used at each job arrival to make better and more timely scaling decisions.

Decision-time complexity: The per-request decision in Policy 1 executes in $O(1)$ time, requiring only a constant number of integer-counter comparisons and a single rounded-difference computation. No model inference is performed, and the computational cost is independent of both the arrival rate λ and the concurrency limit N. The per-request control therefore remains inexpensive even under high request volume.

The learning updates are not performed at every request; they are computed infrequently and asynchronously, without blocking request handling, as detailed in Section V. This contrasts with RL controllers, which evaluate a neural network at every decision.

V. Learning Algorithm

We propose an online algorithm named $NSGD$, for the optimal joint learning of the parameter vector $\boldsymbol{\theta} = (\theta_{stock}, \theta_{idle}, \theta_{exp})$. $NSGD$ is *general* in the sense that it is applicable to a setting that extends beyond the autoscaling framework considered in this paper. For this reason, we begin by considering a general stochastic optimization framework, which we will later apply to our specific autoscaling setting in Section V-E.

Assuming the system evolution follows Markovian dynamics, our main result (Theorem 1) states that the sequence of iterates produced by $NSGD$ converges to a local minimum of the specified cost function.

A. Stochastic Optimization Problem

We now formalize the problem under investigation. Assume that $(X_t)_{t\in\mathbb{R}_+}$ (with $X_t \in \mathscr{X}$ where $\mathscr{X}$ denotes the finite *state space*) represents the evolution of the system's state. Let X_∞ be the random variable describing the system in its stationary regime. We assume that the evolution of X_t, and consequently X_∞, depends on a parameter vector $\boldsymbol{\theta} \in \mathbb{R}^d$, $d \in \mathbb{N}$. To make this dependence explicit, we denote the state process by $(X_t^{\boldsymbol{\theta}})_t$ and its stationary behavior by $X_\infty^{\boldsymbol{\theta}}$.

At each time t, a controller observes the system state X_t and incurs a cost $C : \mathscr{X} \to \mathbb{R}$. The controller is allowed to adjust $\boldsymbol{\theta}$ over time with the objective of minimizing the long-run average cost in a setting where *the controller only knows the cost function C and the current state X_t*, nothing else is known about the system. Specifically, the goal is to minimize the expected stationary cost:

$$c(\boldsymbol{\theta}) := \mathbb{E}[C(X_\infty^{\boldsymbol{\theta}})]. \tag{1}$$

To achieve this, our objective is to design an iterative procedure for $\boldsymbol{\theta}$ able to converge to a minimizer of c.

Remark 2. *The problem defined above is the multi-dimensional generalization of the problem studied in [21]. The main difference is that the optimization parameter $\boldsymbol{\theta}$ is a vector rather than a scalar. Therefore, the stochastic gradient descent algorithm and the corresponding convergence results need to be extended as they do not apply in the setting considered here.*

Remark 3. *Note that the cost function C does not depend on $\boldsymbol{\theta}$* directly*, as it is a function from $\mathscr{X}$. It only depends on $\boldsymbol{\theta}$ via $X_\infty^{\boldsymbol{\theta}}$. This poses the technical difficulty that ∇C is not accessible and rules out the application of a large class of stochastic gradient descent algorithms based on stochastic approximations with Markovian noise (see for example [38], [39]).*

Remark 4. *If one could sample directly from $X_\infty^{\boldsymbol{\theta}}$, then the optimization problem under investigation could be solved via the celebrated Kiefer–Wolfowitz algorithm which, under certain conditions, can provide a sequence of $\boldsymbol{\theta}$'s converging to a minimum of $c(\boldsymbol{\theta})$. However, as discussed in the introduction, we have only access to (transient) samples of $(X_t^{\boldsymbol{\theta}})_t$ rather than $X_\infty^{\boldsymbol{\theta}}$ and this makes our problem more difficult.*

B. Non-Stationary Gradient Descent Algorithm

We now propose an online learning algorithm to compute $\boldsymbol{\theta}^*$, which minimizes the long-run average cost (1). Importantly, the algorithm can be executed regardless of the specific assumptions made about the system evolution X_t. On the other hand, its theoretical guarantees, which will be presented below, will hold under a Markovian structure.

The proposed $NSGD$ algorithm is the multidimensional adapation of the algorithm presented in [21]. It is an adapted version of the Simultaneous-Perturbation Stochastic Approximation (SPSA) approach of the classical Kiefer–Wolfowitz (KW) algorithm, and it works as follows. After each update of $\boldsymbol{\theta}$—from episode $n-1$ to n—the classical SPSA KW algorithm requires sampling two independent states from the stationary measures of $X_\infty^{\boldsymbol{\theta}_n+\delta_n}$ and $X_\infty^{\boldsymbol{\theta}_n-\delta_n}$, where δ_n is the perturbation used to approximate the gradient of the cost function at $\boldsymbol{\theta}_n$. However, since direct access to these stationary distributions is not available, we instead simulate two trajectories of the system evolution, each for τ_n timesteps, to obtain states that are approximately stationary. The length τ_n depends on the system's convergence properties; for instance, when X_t is a Markov chain, τ_n is related to its mixing time and typically grows as $\log n$ times the uniform mixing time of the chain.

We begin with the initial state x_{start} and an initial policy parameter $\boldsymbol{\theta}_0$. Let T_n denote the number of timesteps accumulated by the end of episode $n-1$. For each episode n, we set x_{start} as the initial state for the simulated trajectories and we let $\mathbf{u}_n$ denote a random perturbation vector that we draw independently from the set $\{-1,1\}^d$ and each coordinate $\mathbf{u}_n^{(j)}$ has a Rademacher distribution (*i.e.* +/- 1 Bernoulli with parameter $1/2$).

From T_n to $T_n+\tau_n-1$, we first simulate the system with initial state x_{start} and parameter $\boldsymbol{\theta}_n+\mathbf{u}_n\delta_n$, and observe the states $X_{T_n,T_n+i}^{\boldsymbol{\theta}_n+\mathbf{u}_n\delta_n}$ for $i=1,\dots,\tau_n$, and the random cost $C(\boldsymbol{\theta}_n+\mathbf{u}_n\delta_n, X_{T_n,T_n+\tau_n}^{\boldsymbol{\theta}_n+\mathbf{u}_n\delta_n})$. We reiterate this process K times, and do similar simulations with parameter $\boldsymbol{\theta}_n-\mathbf{u}_n\delta_n$. We then evaluate the average cost with parameters $\boldsymbol{\theta}_n+\mathbf{u}_n\delta_n$ and $\boldsymbol{\theta}_n-\mathbf{u}_n\delta_n$ with the samples $X_{T_n,T_n+(i+1)\tau_n}^{\boldsymbol{\theta}_n+\mathbf{u}_n\delta_n}$ and $X_{T_n+K\tau_n,T_n+(K+i+1)\tau_n}^{\boldsymbol{\theta}_n-\mathbf{u}_n\delta_n}$ for $i=0,\dots,K-1$, from which we may approximate the derivative of the cost function at $\boldsymbol{\theta}_n$ by simultaneous perturbation and eventually compute the following update of the parameter $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}_{n+1}^{(j)} = \boldsymbol{\theta}_n^{(j)} - \gamma_n \frac{\hat{c}(\boldsymbol{\theta}_n+\mathbf{u}_n\delta_n) - \hat{c}(\boldsymbol{\theta}_n-\mathbf{u}_n\delta_n)}{2\mathbf{u}_n^{(j)}\,\delta_n} \tag{2}$$

for all $i=1,\dots,d$, where γ_n is the stepsize of the parameter update, and

$$\hat{c}(\boldsymbol{\theta}_n+\mathbf{u}_n\delta_n) = \frac{1}{K}\sum_{i=1}^{K} C\left(\boldsymbol{\theta}_n+\mathbf{u}_n\delta_n, X_{T_n,T_n+i\tau_n}^{\boldsymbol{\theta}_n+\mathbf{u}_n\delta_n}\right), \tag{3a}$$

$$\hat{c}(\boldsymbol{\theta}_n-\mathbf{u}_n\delta_n) = \frac{1}{K}\sum_{i=1}^{K} C\left(\boldsymbol{\theta}_n-\mathbf{u}_n\delta_n, X_{T_n+K\tau_n,T_n+(K+i)\tau_n}^{\boldsymbol{\theta}_n-\mathbf{u}_n\delta_n}\right). \tag{3b}$$

This process is formalized in Algorithm 2.

Algorithm 2: Non-stationary Gradient Descent Algorithm.

Input: γ_n step-size sequence, δ_n discretization step, initial parameter $\boldsymbol{\theta}_0$ and state x_{start}, T the total simulation time and a parameter τ.

Set $n=0$ the algorithm time-step and $t=0$ the current simulation time-step, $x=x_{\text{start}}$ the initial state.
while $t \le T$ **do**
  $\tau_n = \tau\log(n+1)$
  for *simulation number* $i=1,\dots,K$ **do**
    Simulate the system starting at the state reached at time $T_n+(i-1)\tau_n$ with parameter $\boldsymbol{\theta}_n+\mathbf{u}_n\delta_n$ over τ_n steps
  end
  for *simulation number* $i=1,\dots,K$ **do**
    Simulate the system starting at the state reached at time $T_n+(K+i-1)\tau_n$ with parameter $\boldsymbol{\theta}_n-\mathbf{u}_n\delta_n$ over τ_n steps
  end
  Compute the average over the $2K$ simulation end states to get $\hat{c}_n(\boldsymbol{\theta}_n+\mathbf{u}_n\delta_n)$ and $\hat{c}_n(\boldsymbol{\theta}_n-\mathbf{u}_n\delta_n)$.
  Update the empirical stationary measure: the number of visits of a given state under any parameter, divided by the total number of visits.
  Compute the parameter update (2).
  $t := t+2K\tau_n$ and $n := n+1$
end

C. Convergence Under Markovian Assumptions

We will rely on the following assumptions, also used in [21] and, as discussed below, inspired by the seminal work in [40].

Assumption 1 (Markov process)**.** *For any T,t in $\mathbb{N}$ with $T\le t$, the stochastic process $X_{T,t}^{\boldsymbol{\theta}}$ is a discrete-time Markov chain from time T to time t. Its transition matrix is denoted $P_{\boldsymbol{\theta}}$.*

Note that the transition matrix of the Markov chain X_t is indexed by $\boldsymbol{\theta} \in \mathbb{R}^d$.

Assumption 2 (Regularity of the cost function)**.**

(2.a) $c : \mathbb{R}^d \to \mathbb{R} \in \mathscr{C}^3$ *(3rd derivative is continuous)* .

(2.b) ∇c *is Lipschitz,*

(2.c) There exists $L > 0$ *and* $r > 0$ *such that for any* $x \in \mathscr{X}$, $\langle \nabla c(\boldsymbol{\theta}, x), \boldsymbol{\theta} \rangle < r$ *for* $||\boldsymbol{\theta}|| > L$. *If* $||\boldsymbol{\theta}|| > L$, *then* $\sup_{\boldsymbol{\theta}} \mathrm{Var}_{\boldsymbol{\theta}}(c(\boldsymbol{\theta}, X_\infty^{\boldsymbol{\theta}})) < \infty$.

While Assumption 2 is quite technical and may look restrictive, it will be satisfied in the autoscaling case by extending the cost function to $\mathbb{R}^3$ in a controlled way. This will be detailled in Section V-E below.

Assumption 3 (Uniform mixing time)**.** *For the system* $X_t^{\boldsymbol{\theta}}$, *let* x *be an initial state and let* $m_{\boldsymbol{\theta}}$ *denote the corresponding stationary measure within parameter* $\boldsymbol{\theta}$. *There exists* $C_1 > 0$ *and* $\rho < 1$ *independent of* $\boldsymbol{\theta}$ *such that*

$$\left\| P_{\boldsymbol{\theta}}^k(x, \cdot) - m_{\boldsymbol{\theta}} \right\|_1 \leq C_1 \rho^k$$

for any $\boldsymbol{\theta}$ *and* $k \in \mathbb{Z}_+$.

The previous assumption establishes a connection between the transient regime – observable over time – and the stationary properties of the system. It is uncommon in standard stochastic gradient descent algorithms because these typically rely on the ability to sample directly from the stationary distribution, which is not possible in our case. Note that $P_{\boldsymbol{\theta}}$ is the transition matrix of the underlying Markov chain (Assumption 1).

We can now state our main theoretical result, which states the almost sure convergence of the sequence $\boldsymbol{\theta}_n$ produced by $NSGD$ algorithm, under the following parametrization: The sequences $(\gamma_n)_n$, $(\delta_n)_n$ and $(\tau_n)_n$ are such that

$$\lim_{n\to\infty} \delta_n = 0, \qquad \lim_{n\to\infty} \tau_n = +\infty \tag{4a}$$

$$\sum_n \gamma_n = \infty, \qquad \sum_n \gamma_n^2 \delta_n^{-2} < \infty \tag{4b}$$

and

$$(\gamma_n)_n, (\delta_n)_n \text{ and } \left(\frac{\gamma_n}{\delta_n}\right)_n \text{ are decreasing.} \tag{5}$$

Theorem 1. *Let* $(\boldsymbol{\theta}_n)_n$ *be the sequence of random variables generated by Algorithm 2 with parametrization* (4)-(5)*. Under Assumptions 1, 2 and 3, then* $c(\boldsymbol{\theta}_n)$ *is decreasing and*

$$\boldsymbol{\theta}_n \overset{a.s.}{\to} \boldsymbol{\theta}^* \tag{6}$$

where $\boldsymbol{\theta}^*$ *is such that* $\nabla c(\boldsymbol{\theta}^*) = 0$.

The proof is lengthy and technical, and therefore deferred to Appendix A. It follows the analysis of stochastic gradient descent in [40], under the same set of assumptions. Under these conditions, the only general theoretical guarantee is that the gradient vanishes at the limit point once convergence is reached. In our autoscaling case, however, we find that the cost function is quasi-convex, implying that any such limit point corresponds to a global minimum. A discussion on the assumption validation is presented in Appendix B.

D. Parallel Implementation

Algorithm 2 (NSGD) requires, for each iteration of the loop in Line 4, the simulation of two trajectories of X_t for τ_n transition steps using the control parameters perturbed in opposite directions. Specifically, $NSGD$ performs one run with the control parameters incremented by a perturbation vector $\mathbf{u}_n \delta_n$ and another with the parameters decremented by the same perturbation vector. This setup aligns well with scenarios where the algorithm operates directly on the real system, allowing online learning of the control parameters. However, when a reliable simulator is available, the algorithm can be parallelized. The two trajectories can be evaluated simultaneously on separate simulator instances, yielding equivalent outcomes. Moreover, all the K runs used to reduce variance in the algorithm can be performed in parallel, which eliminates the need for sequential execution and significantly accelerates the overall training process. Furthermore, distinct perturbations can be applied across simulators, and the resulting gradients can be averaged. This approach enhances training stability and mitigates the risk of large, destabilizing parameter updates. The simulators and parallelization enable offline learning of the optimal control parameters, eliminate the energy overhead associated with online exploration, and offer substantial computational speedup. The learned policy can then be seamlessly deployed on the real system. Section VI presents a comparative performance analysis of the algorithm in both parallel and sequential configurations, highlighting the advantages of leveraging parallelism during the learning phase.

E. Application to Autoscaling

We now illustrate how the general framework developed above applies to our autoscaling setting. It is important to emphasize that Algorithm 2 is applicable regardless of the nature of the observation process $(X_t)_t$ and independent of any specific assumptions about its dynamics. However, the convergence guarantee stated in Theorem 1 does rely on the presence of a Markovian structure in the system evolution.

First, we construct a continuous-time Markov chain that models the dynamics induced by the autoscaling policy described in Section IV. This model is very similar to the Markov model presented in [21]; the main difference lies in the scale-up policy, which is more complex in the present setting as discussed in Section IV. We assume that jobs join the system following a Poisson process with rate λ. Service, initialization and expiration times associated to each function are exponentially distributed random variables with rate μ, β, θ_{exp}, respectively. The sequences of job inter-arrival, service, initialization and expiration times are i.i.d. and independent of each other.

It is easy to see that a Markovian representation of the system dynamics is given by the state variable $x \in \mathscr{X} := \left\{ x = (x_i \in \mathbb{N} : i = 1, \ldots, 4) \;\middle|\; \sum_{i=1}^3 x_i \leq N,\ x_4 \leq x_3 \right\}$. Here, x_1, x_2, and x_3 denote the number of *idle-on*, *busy*, and *initializing* (both *init-free* and *init-reserved*) functions, respectively. The fourth component, x_4, corresponds to the number of init-reserved functions.

Note that the number of init-reserved functions can also be interpreted as the number of *blocked* jobs that are currently waiting in the central queue. Consequently, the number of cold functions in state x is $N - \sum_{i=1}^{3} x_i$.

The Markov chain $(X_t)_t$ has the following transition matrix (using e_i the null vector except at coordinate i where its value is one).

$$P_{\boldsymbol{\theta}}(x, x') = \begin{cases} \Omega\lambda \mathbb{I}_{\{E_1(x)\}} & \text{if } x' = x + e_4 \\ & \quad +(1 + \pi_{\theta_{stock}}(x))e_3 \\ \Omega\lambda \mathbb{I}_{\{E_2(x)\}} & \text{if } x' = x - e_1 + e_2 \\ & \quad +e_3(\pi_{\theta_{stock}} - x_1 - x_3) \\ \Omega\mu x_2 & \text{if } x' = x + (e_1 - e_2)\mathbb{I}_{\{x_4=0\}} \\ & \quad - e_4\,\mathbb{I}_{\{x_4>0\}} \\ \Omega\theta_{exp}\, x_1 & \text{if } x' = x - e_1 \\ \Omega\beta x_3 \mathbb{I}_{\{x_4>0\}} & \text{if } x' = x + e_2 - e_3 - e_4 \\ \Omega\beta x_3 \mathbb{I}_{\{x_4=0\}} & \text{if } x' = x + e_1 - e_3 \end{cases}$$

for all states $x, x' \in \mathscr{X}$, with $x' \neq x$, where Ω is a uniformisation constant ($\Omega^{-1} := \lambda + \beta N + \mu N + \theta_{exp} N$).

Then, we let $\boldsymbol{\theta} \in \mathbb{R}^d$ with $d = 3$, where the components are interpreted as follows: $\boldsymbol{\theta}_1 = \theta_{stock}$, $\boldsymbol{\theta}_2 = \theta_{idle}$, and $\boldsymbol{\theta}_3 = \theta_{exp}$. This is the vector that Algorithm 2 learns optimally.

Finally, in the set $[0, N] \times [0, N] \times [0, \infty) \times \mathscr{X}$ for $\theta_{stock}, \theta_{idle}\theta_{exp}, x$ let the instant random cost be

$$C(\boldsymbol{\theta}, x) := \sum_{i=1}^{4} w_i x_i + \mathbb{I}_{\{x_2+x_4=N\}} w_{\text{rej}}, \tag{7}$$

where w_i, for $i = 1, \ldots, 4$, and w_{rej} are positive weights. w_1, w_2, w_3, w_4 correspond respectively to the power consumption of a function in respective states *idle-on*, *busy*, *initializing* and *init-reserved*. w_{rej} denotes the penalty weight for rejecting a job, as the condition $\mathbb{I}_{\{x_2+x_4=N\}}$ indicates that the system is fully occupied, i.e., all N servers are either busy or reserved, thus preventing the acceptance of new jobs. (7) defines a tradeoff between energy consumption and user-perceived delay performance.

VI. Experimental Evaluation

This section presents a comprehensive evaluation of $NSGD$ to assess its practical applicability and verify its theoretical properties. We aim to answer the following key questions:

i) To what extent does the joint use of all three control parameters improve cost efficiency compared to using one or two parameters in isolation?
ii) Does the algorithm converge reliably across different configurations?
iii) Can the algorithm leverage parallelism effectively for improved training performance?

Section VI-A describes the experimental setup and implementation details. Section VI-B discusses the convergence behavior and effectiveness of the algorithm across various scenarios. Finally, Section VI-C evaluates the benefits of parallel execution during the offline learning phase and examines its impact on computational efficiency.

A. Experimental Setup

To evaluate NSGD, we build upon the SimFaaS framework [31], a simulator designed to emulate public serverless platforms such as AWS Lambda, Google Cloud Functions, and Azure Functions. SimFaaS includes models for autoscaling behavior and tools for performance analysis and visualization. Its strong validation against real cloud systems The simulator demonstrates high fidelity with real-world observations, achieving an average error of only 12.75% for cold start probability and Mean Absolute Percentage Errors of 3.43% and 0.17% for instance count and idle capacity, respectively. To enable the implementation of our autoscaling algorithm, we extend SimFaaS by explicitly modeling the *init-free* and *init-reserved* states, which are not represented in the original framework. In addition, we incorporate our custom autoscaling policy and the learning algorithm that adapts scaling decisions over time based on observed system dynamics. Our implementation can be found at https://github.com/Wamuhindo/NSGD.

The system arrival process is modeled as a Poisson distribution, which is a good approximation of production request (job) arrival distribution [10], [41]. The initialization, service, and expiration processes are assumed to follow exponential distributions. These modeling choices are made to facilitate evaluation within a well-understood analytical framework where convergence to an optimal policy is theoretically established. However, it is important to note that the algorithm itself is not inherently dependent on these stochastic assumptions for its functionality or applicability. To assess this claim, we consider scenarios where service times follow a Pareto distribution (more representative of enterprise systems, see [23] and where the expiration of an instance is deterministic (as implemented by AWS Lambda, Google Cloud Functions, IBM Cloud Functions and Azure Functions, [29]). We report the results of these analyses in Section VII. Below are the different considerations made during the simulation:

1) For Algorithm 2 we choose $\tau = 10^4$, $T = 4 \cdot 10^6$, $\gamma_n = 1/n$, $\mathbf{u}_n\delta_n = [0.5u_{n_1},\ 0.5u_{n_2},\ u_{n_3}] \times n^{-2/3}$. The components u_{n_1}, u_{n_2}, and u_{n_3} of the perturbation vector $\mathbf{u}_n$ are chosen uniformly at random in $\{-1, +1\}$ at every algorithm iteration. For convergence analysis, we consider four distinct initialization triplets $(\theta^0_{stock},\ \theta^0_{idle},\ \theta^0_{exp}) \in \{(1,\ 1,\ 5), (1,\ 5,\ 5), (3,\ 4,\ 10), (10,\ 2,\ 1)\}$.
The expiration rate used in the learning algorithm, denoted θ_{exp}, is designed to be on the same order of magnitude as the other control parameters, θ_{stock} and θ_{idle}, to ensure balanced learning dynamics. However, the actual simulator expiration rate, denoted ξ, is typically several orders of magnitude smaller. To reconcile this scale difference, we define $\theta_{exp} = K_{exp} \times \xi$, where K_{exp} is a fixed scaling factor. This formulation preserves the ability to learn the optimal expiration behavior, since optimizing θ_{exp} is equivalent to optimizing ξ, via $\xi^* = \theta^*_{exp}/K_{exp}$. In our experiments, we use $K_{exp} = 10^3$.
2) For the weights defining the cost function we set $w_1 = 2$, $w_2 = 1$, $w_3 = 5$, $w_4 = 100$, and $w_{rej} = 200$. Initializing functions incur high energy costs due to operations like launching a virtual machine, loading dependencies, and

connecting to external services. To capture this, we set $w_3 \geq \max\{w_1, w_2\}$. To prioritize user-perceived performance and penalize rejected requests, we assign significantly larger values to w_4 and w_{rej} compared to w_3. Since w_4 and w_{rej} reflect subjective trade-offs related to user satisfaction and provider reputation–rather than directly influencing energy metrics–we conduct a sensitivity analysis in Section VII by varying these weights. This allows us to evaluate the robustness of $NSGD$ under changes in these parameters.

3) For the Markov chain parameters, we set $N = 50$, the arrival rate λ such that $\lambda/N \in \{0.1,\ 0.2,\ 0.3,\ 0.7\}$, the initialization rate $\alpha = 0.1$ and the service rate $\mu = 1$ jobs per time units. With a time unit set at 10 milliseconds, these values reflect realistic service rates for serverless applications according to data discussed in [21], [29], [42]. Function execution times in serverless environments exhibit considerable variability due to heterogeneous workloads and varying computational requirements. It is well known that the Pareto distribution is well-suited for modeling scenarios where a small fraction of invocations require disproportionately large execution times; e.g., [43]. We parametrize the Pareto distribution with the shape $= 2.12$ and scale $= 0.53$ to have for the distribution mean $\approx \mu = 1$ and the variance $Var \approx 3.96$. Moreover, in practice, many cloud providers implement deterministic timeout policies rather than probabilistic expiration mechanisms [31].According to [31] this value is by default 20 minutes for Azure Functions and 10 minutes for Google Cloud Functions, AWS Lambda, IBM Cloud, and Apache OpenWhisk. Therefore, we evaluate the performance of $NSGD$ under both stochastic service time distributions and deterministic expiration policies to ensure our findings are applicable to production environments.
4) In Section 3, we established that θ_{stock} and θ_{idle} belong to $\mathbb{R}_+$, while the number of function instances to be launched in the system must be an integer. We define $\pi_{\theta_{stock}} = \text{Round}(\theta_{stock})$ as a randomized rounding scheme and choose $\pi_{\theta_{stock}} = \lfloor \theta_{stock} \rfloor$ with probability $\theta_{stock} - \lfloor \theta_{stock} \rfloor$ and $\pi_{\theta_{stock}} = \lceil \theta_{stock} \rceil$ with probability $\lceil \theta_{stock} \rceil - \theta_{stock}$. This rounding ensures $\mathbb{E}[\pi_{\theta_{stock}}] = \theta_{stock}$. We adopt the same rounding for θ_{idle} leading to $\mathbb{E}[\pi_{\theta_{idle}}] = \theta_{idle}$. While this rounding procedure makes the cost function non-differentiable at integer values of θ_{stock} and θ_{idle}, the finite-difference gradient estimator $\frac{\hat{c}(\boldsymbol{\theta}_n + \mathbf{u}_n \delta_n) - \hat{c}(\boldsymbol{\theta}_n - \mathbf{u}_n \delta_n)}{2(\mathbf{u}_n)_i\, \delta_n}$ provides correct descent direction so that convergence of the optimization algorithm should be preserved as confirmed by our numerical experiments

B. Algorithm Convergence Analysis

This section evaluates the convergence of the proposed learning algorithm, detailed in Section V. The learning dynamics of the three parameters, θ_{stock}, θ_{idle}, and θ_{exp}, for a representative example are illustrated in Figure 2, which presents their evolution across learning iterations under a system arrival rate of $\lambda = 5$. Each subplot within the figure comprises four distinct initialization triplets $(\theta^0_{stock},\ \theta^0_{idle},\ \theta^0_{exp}) \in \{(1,\ 1,\ 5), (1,\ 5,\ 5), (3,\ 4,\ 10), (10, 2, 1)\}$. The reason behind the choice of these four initialization triplets is to ascertain if the algorithm converges even when starting from different initial values. It is important to note that convergence does not imply that θ_{stock}, θ_{idle}, and θ_{exp} must attain identical values for different training runs. Rather, it indicates that the values to which a particular training run converges yield equivalent (or near-equivalent) minimum cost. This phenomenon can be observed given the three-dimensional parameter space, where multiple distinct configurations can lead to equivalent optimal costs.

The plots demonstrate the robustness and consistency of the learning algorithm. As shown in Figure 2a, the parameter θ_{stock} converges to values within the narrow range of approximately 3.0 to 3.2 across all initializations. Similarly, Figure 2a (center) shows that θ_{idle} stabilizes between 2.6 and 2.8, while Figure 2a (right) indicates that θ_{exp} converges within the interval of 2.7 to 2.8.

$\boldsymbol{\theta}^0$	$\boldsymbol{\theta}^0_1$	$\boldsymbol{\theta}^0_2$	$\boldsymbol{\theta}^0_3$	$\boldsymbol{\theta}^0_4$
Cost	12.53	12.57	12.82	12.37

Table I: Cost yielded by the optimal values from 4 initializations under Pareto-distributed service times.

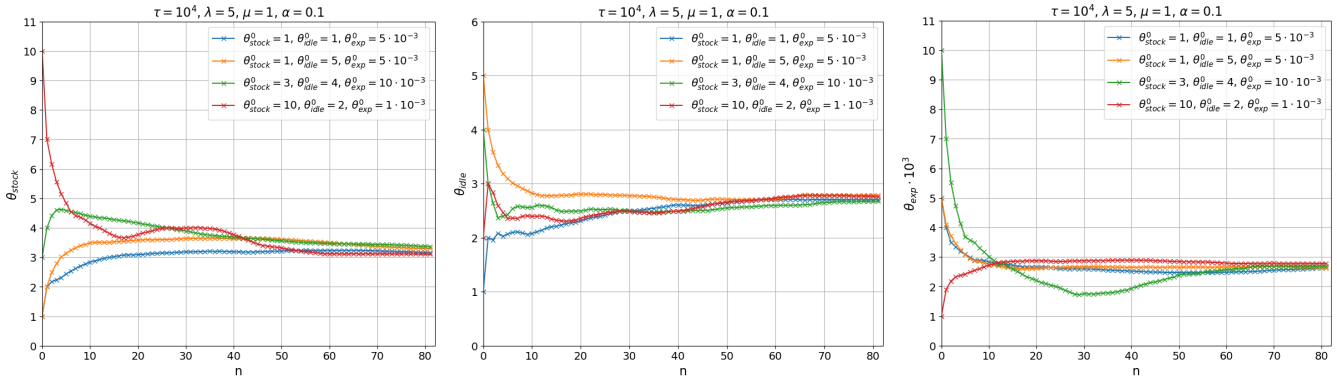
Figures 2b illustrate the convergence behavior of the control parameters θ_{stock}, θ_{idle}, and θ_{exp} under Pareto-distributed service times and $\tau = 10^5$. While the control parameters do not converge to identical values across different initial values, the resulting values yield equivalent costs, as shown in Table I. Performing the evaluation of the resulting values over 10 different runs for each initial value $\boldsymbol{\theta}^0$ (i.e., setting different seeds), the costs associated with the four final parameters fall within a narrow range of 12.37 to 12.82. This indicates that the algorithm converges to different but equally optimal solutions.

An observation from the plots in Figure 2 is the role of initialization in determining the speed of convergence. In particular, configurations initialized with lower values of θ_{stock} and θ_{exp} tend to converge more quickly compared to those initialized with higher values. The algorithm produces stable and comparable optimal values across all tested parameter configurations, despite variations in convergence speed. This robustness shows that although initialization affects early-stage learning efficiency, the quality of the final solution is unaffected. In service-oriented computing environments, where minimizing convergence time can drastically reduce resource consumption without compromising performance guarantees, such behavior is especially desirable. In quantitative terms, across all initializations and arrival rates the control parameters converge within roughly 35–80 iterations (Figure 2); by Table II, this corresponds to only a couple of minutes of parallel training, making the learning phase practical for deployment.

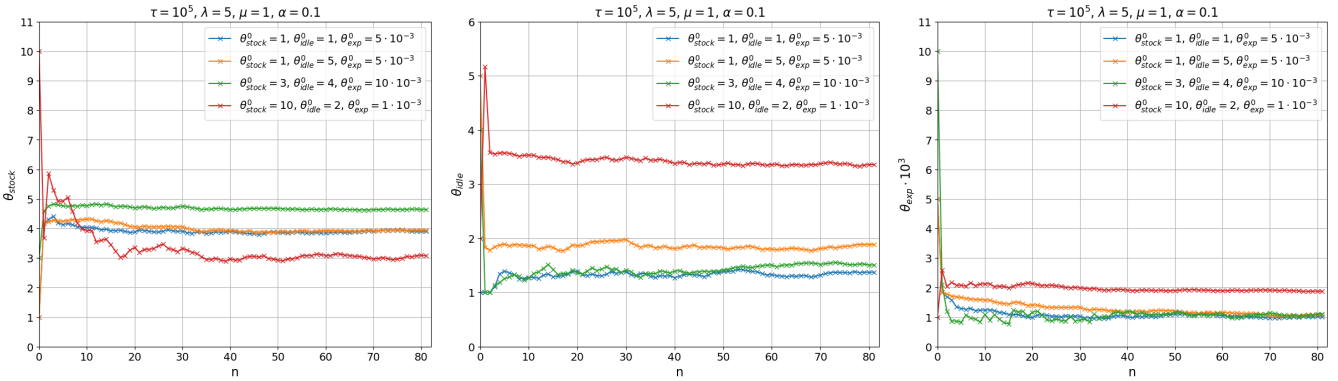
Overall, the results presented in Figure 2 confirm the effectiveness of $NSGD$ in learning the control parameters, thereby enabling efficient autoscaling in serverless platforms.

C. Parallel vs Sequential Learning

This section presents a comparative performance analysis of the algorithm in both parallel and sequential configurations.



(a) Control parameters θ_{stock} (left), θ_{idle} (center), and θ_{exp} (right) under exponentially distributed service times.



(b) Control parameters θ_{stock} (left), θ_{idle} (center), and θ_{exp} (right) under Pareto-distributed service times.

Figure 2: Evolution of control parameters θ_{stock}, θ_{idle}, and θ_{exp} over 80 iterations under two types of service time distributions: exponential and Pareto.

To compare the parallel and sequential execution, we run the proposed $NSGD$ algorithm on a MacBook Pro, with a 12-core Apple Silicon M2, and 32GB of RAM. We perform experiments with $\tau = 10^4$, $K \in \{1, 2, 4\}$ and $\lambda/N \in \{0.1, 0.2, 0.3\}$ to assess the speedup yielded by parallel execution. Each value in the Table II is an average of 5 different runs. The results presented in Table II demonstrate that parallel execution is more time-efficient than sequential execution. In particular, the observed speedup increases with the value of K. Since the two trajectories with opposite perturbations of $\boldsymbol{\theta}$ are executed in parallel, the ideal speedup is $2 \times K$. However, this ideal speedup is not fully realized in practice due to factors such as CPU scheduling overhead, memory access latency, and synchronization overhead.

For $K = 1$, the observed speedup ranges between 1.7 and 1.8. When $K = 2$, the speedup increases to between 3.0 and 3.4, corresponding to approximately $[1.5,\ 1.7] \times K$. With $K = 4$, the speedup further improves, ranging from 5.65 to 6.45, which falls within $[1.4,\ 1.6] \times K$. Overall, parallel execution is consistently more time-efficient compared to sequential execution with a roughly constant time complexity. However the time complexity of the sequential execution grows with K. It is important to note that this speed up is observed since there is enough computing capacity (CPUs) that enable parallel execution.

λ/N	**#Cores** ($2K$)	**Sequential**	**Parallel**	**Speedup**
0.1	2	248	138	1.80
0.2		290	163	1.78
0.3		314	184	1.71
0.1	4	464	136	3.41
0.2		506	158	3.20
0.3		553	181	3.05
0.1	8	941	146	6.45
0.2		1032	172	6.00
0.3		1107	196	5.65

Table II: Sequential vs parallel training time (in seconds) and speedup for 80 iterations across three different arrival rates.

VII. Numerical Comparison with Baselines

This section presents a numerical comparison between $NSGD$ and two recent state-of-the-art baselines. Specifically, Section VII-A evaluates $NSGD$ against the method proposed in [21], while Section VII-B provides a detailed comparison with the LSTM-PPO approach proposed in [16].

A. Comparison with NSGD-1D Algorithm

The approach proposed in [21] focuses on learning a single control parameter, θ_{stock}, to drive the autoscaling policy. In contrast, $NSGD$ jointly learns three control parameters. Figure 3 presents a performance comparison between $NSGD$ and the baseline from [21], which we refer to as $NSGD$-1D. All reported results are averaged over 10 independent runs to ensure robustness. $NSGD$ consistently outperforms NSGD-1D

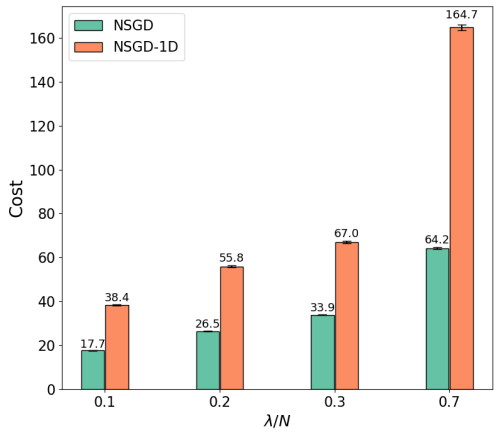


Figure 3: Cost comparison of $NSGD$ and $NSGD$-1D algorithm in [21] where only θ_{stock} is learned. Results are averages over 10 runs.

across all arrival rates. Specifically, relative cost reductions of 120%, 113%, 100%, and 156% are observed for $\lambda/N = 0.1$, $\lambda/N = 0.2$, $\lambda/N = 0.3$, and $\lambda/N = 0.7$, respectively. Notably, the improvements are more substantial at higher workload levels (e.g., $\lambda/N = 0.7$), where the performance gap between the two approaches becomes particularly large, indicating the scalability and robustness of our method under heavy load. These results highlight the effectiveness of our multi-parameter learning approach. In fact, by jointly optimizing θ_{stock}, θ_{idle}, and θ_{exp}, $NSGD$ achieves more efficient autoscaling policies than those based on a single control variable.

B. Comparison with LSTM-PPO

This section compares the performance of $NSGD$ against the one yielded by LSTM-PPO presented in [16]. LSTM-PPO relies on two core components to ensure the autoscaling: the Long Short-Term Memory (LSTM) network to capture the environment stochasticity, and the Proximal Policy Optimization (PPO) to learn the autoscaling policy. LSTM-PPO observes some system metrics in a control time period, takes an action, and computes a reward. The observation comprises the average memory and CPU utilizations, the average execution time, the number of actual function replicas in the system, the proportion of requests successfully served in the control time window, and the total number of requests. In the reward function, LSTM-PPO considers the number of requests successfully served, the average memory and CPU utilization, and the margin between the actual replicas and the minimum number of replicas. The RL agent action space is defined by $\{-2, -1, 0, +1, +2\}$, where $+A$ means scale-up by A replicas, $-A$ means scale-down by A replicas and 0 means do nothing. We implemented LSTM-PPO in ray (RLlib) following the exact implementation done in [16]. Since SimFaaS [31] does not give direct access to CPU and memory utilization, we extend it to provide these two metrics required by LSTM-PPO.

As discussed in Section VI-A, we trained $NSGD$ for 4×10^6 simulation steps and then evaluated it. An evaluation run consists of 4×10^6 steps. To have a fair comparison, we also first trained LSTM-PPO and then evaluated the trained policy for the same number of steps. During the evaluation, we computed the LSTM-PPO cost with $NSGD$ cost function in Equation 7.Moreover, we provide a comparison with LSTM-PPO when it is trained considering the reward as the negative of the cost that we use for $NSGD$.We refer to this case as LSTM-PPO-COST.

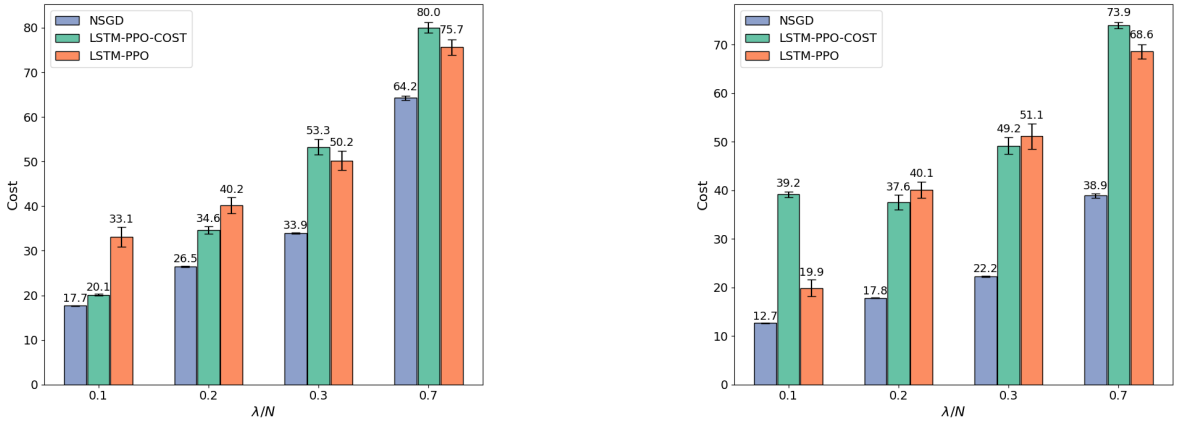
1) Exponentially distributed service times: Figure 4 compares the performance of $NSGD$ and LSTM-PPO across 10 different runs and four arrival rates $\lambda/N = 0.1$, $\lambda/N = 0.2$, $\lambda/N = 0.3$, and $\lambda/N = 0.7$ with exponentially or Pareto distributed arrival rates. Specifically, when under exponentially distributed service times and LSTM-PPO trained with its original reward from [16], $NSGD$ consistently yields lower costs as shown in Figure 4a (orange) with average improvements of 48%, 35%, 34%, and 15% for $\lambda/N = 0.1$, $\lambda/N = 0.2$, $\lambda/N = 0.3$, and $\lambda/N = 0.7$, respectively. Moreover, $NSGD$ also exhibits lower variance across all arrival rates.

Even when LSTM-PPO is trained using the same cost function as $NSGD$, $NSGD$ still outperforms LSTM-PPO, across all arrival rate settings. As shown in Figure 4a (green), $NSGD$ achieves lower cost by margins of 3, 8, 19, and 16 for $\lambda/N = 0.1$, 0.2, $= 0.3$, and 0.7, respectively. LSTM-PPO-COST performs better than its original version under low arrival rates, but the advantage decreases as the load increases. This behavior can be explained by the agent ability, at low workloads, to leverage the rejection-related cost in the reward function. As the workload increases, the time-driven nature of LSTM-PPO limits its responsiveness, reducing its effectiveness in handling request rejections.

2) Pareto-distributed service times: As illustrated in Figure 4b, the performance trend under Pareto-distributed service times mirrors the one of the exponential case. However, $NSGD$ demonstrates adaptability to the heavier-tailed distribution, achieving lower costs at all arrival rates compared to the exponential scenario. LSTM-PPO, using its original reward, yields similar costs for $\lambda/N = 0.2$ and $\lambda/N = 0.3$, and slightly better results for 0.1 and 0.7. Nevertheless, $NSGD$ maintains a clear advantage, with relative cost margin exceeding 100% for $\lambda/N = 0.2$ and $\lambda/N = 0.3$, and reaching 67% and 75% for 0.1 and 0.7, respectively. LSTM-PPO-COST fails to adapt effectively to the change in service time distribution and is still outperformed by $NSGD$ across all arrival rates. The relative cost margin exceeds 100% in all cases except for $\lambda/N = 0.7$, where it remains high at 90%.

C. Robustness of $NSGD$ with Respect to Expiration Time

As previously discussed, many cloud providers use deterministic expiration policies [31]. To reflect this operational reality, we evaluate the robustness of $NSGD$ under non-Markovian but realistic scenarios with Pareto-distributed execution time and deterministic expiration policy. All reported results are averaged over 10 independent runs. Figure 5 presents the results across different settings, including combinations of exponential and Pareto service time distributions with both exponential and deterministic expiration policies. The results show that $NSGD$ effectively adapts to each scenario, learning control parameters that minimize operational costs. In particular, scenarios with Pareto-distributed service times consistently yield lower costs than their exponential counterparts. Furthermore, $NSGD$ yields comparable cost under deterministic



(a) Exponentially distributed service times.

(b) Pareto distributed service times.

Figure 4: Cost comparison of $NSGD$ with LSTM-PPO [16] under 10 different runs and three arrival rates. The service times follow an exponential distribution in subfigure (a) and a Pareto distribution in subfigure (b). LSTM-PPO considers two scenarios, one with its original reward (subfigure (a) orange) and another (LSTM-PPO-COST) with our cost as the negative of the reward (subfigure (b) green).

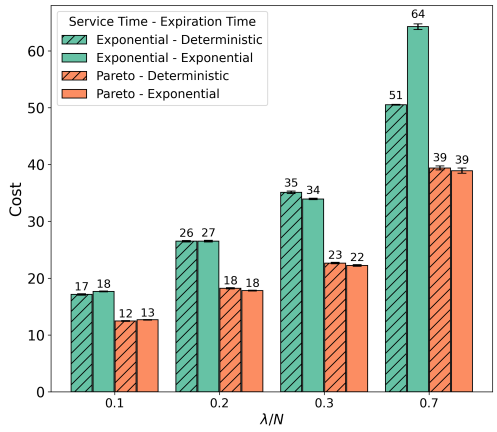


Figure 5: Cost comparison of $NSGD$ across different combinations of service time and expiration time distributions. Service times follow either an exponential or Pareto distribution, while expiration times are modeled using either an exponential or deterministic distribution.

expiration policies to that observed with exponential ones, for which convergence guarantees are established. These findings highlight the practical applicability and robustness of $NSGD$ in realistic, non-Markovian serverless environments.

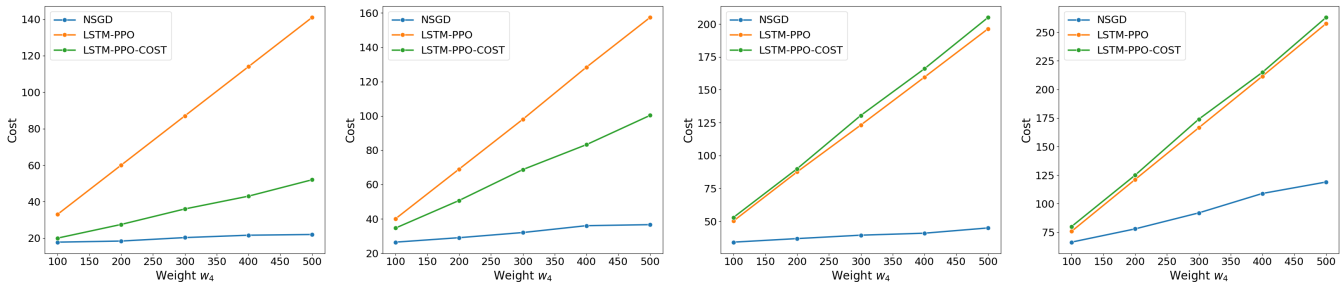
Regardless of the reward formulation for LSTM-PPO and the service time distribution considered, $NSGD$ consistently outperforms it. This superiority can be attributed to two key factors. First, $NSGD$ operates in a scale-per-request fashion, making scaling decisions at each request arrival based on the current control parameter vector $\boldsymbol{\theta}$. In contrast, LSTM-PPO is time-driven: it updates scaling decisions at fixed control intervals. As a result, during these intervals, multiple requests may be rejected depending on the current system state (e.g., number of busy or idle instances), leading to a higher rejection cost for LSTM-PPO. Second, the distinction lies in the way the control is performed: LSTM-PPO adjusts only the total number of replicas in the system, whereas $NSGD$ modulates this number through fine-grained control over individual parameters in $\boldsymbol{\theta}$, each of which has a distinct and direct impact on the overall cost.

D. Cost Sensitivity Analysis

In this section we perform a sensitivity analysis of $NSGD$ with respect to the cost function weights. w_4 and w_{rej} are designed to reflect user-perceived performance rather than direct energy consumption. While the costs associated with them are not supported directly by the service provider, they may result in indirect costs through degraded reputation or user satisfaction. Because of this, these weights can be tuned subjectively by cloud providers based on how they prioritize user experience. For the results presented in Figures 2 and 4, we set $w_4 = 100$ and $w_{rej} = 200$. To evaluate the robustness of our learning algorithm with respect to these subjective cost parameters, we performed a sensitivity analysis by varying w_4 from 100 to 500, and setting $w_{rej} = 2 \times w_4$. The results, presented in Figure 6, illustrate the system cost variation under different arrival rates. Our approach demonstrates strong robustness since, regardless of the specific values of w_4 and w_{rej}, the algorithm consistently learns policies that minimize the total cost as observed in all the plots. In contrast, LSTM-PPO with its original cost function fails to adapt to the changing weight settings. Its total cost increases linearly with w_4, exhibiting a sharp rise across all arrival rates. Although LSTM-PPO-COST shows improved rejection handling under low arrival rates (Figures 6a and 6b), it still struggles at higher arrival rates, similar to its original formulation. This supports our earlier findings (see Figure 4) that the time-driven design of LSTM-PPO limits its ability to handle job rejections effectively in certain workload scenarios.

VIII. Conclusion

This paper presented $NSGD$, an online multi-parameter stochastic gradient descent algorithm for autoscaling in serverless platforms. The proposed approach optimizes system costs by dynamically balancing energy consumption and user-perceived performance. Unlike heuristic-based or RL approaches in the literature, which often lack convergence guarantees, $NSGD$ provides theoretical convergence in Markovian environments. Furthermore, extensive simulations demonstrate that $NSGD$ remains effective and converges even in non-Markovian settings with Pareto-distributed service times and



(a) $\lambda/N = 0.1$ (b) $\lambda/N = 0.2$ (c) $\lambda/N = 0.3$ (d) $\lambda/N = 0.7$

Figure 6: Cost sensitivity analysis when changing weights associated with *init-reserved* (w_4) and rejection (w_{rej}) for different values of λ/N.

deterministic expiration thresholds. Comparative evaluations against two recent state-of-the-art baselines–including a single-parameter control strategy and the LSTM-PPO RL algorithm–highlight the superior performance of $NSGD$. Across a range of arrival rates and workload conditions, $NSGD$ achieves significant cost reductions (up to 100%), while maintaining robust autoscaling decisions in both Markovian and non-Markovian settings. These results demonstrate $NSGD$ as a reliable autoscaling solution for modern serverless environments that offers strong performance guarantees and adaptability to real-world workload variability.

Building on this foundation, future research will investigate the integration of $NSGD$ into serverless function pipelines with multiple interdependent stages, aiming to optimize resource provisioning and scaling decisions across pipeline layers. Further exploration will also consider support for heterogeneous function profiles.

ACKNOWLEDGMENT

This work has been funded by the EMERGENCE 2025 DynamicEnergy project entitled “Reinforcement Learning for Dynamic Minimization of Energy Consumption in Real Time Embedded Systems.”, the IRGA (Initiatives de Recherche Grenoble Alpes) ALB project entitled “Auto-scaling and Load Balancing for Serverless Platforms” and FOUNDRY-ANR-23-PEIA-0003.

REFERENCES

[1] Y. Li, Y. Lin, *et al.*, “Serverless computing: State-of-the-art, challenges and opportunities”, *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1522–1539, 2023.

[2] Amazon, *Serverless computing - aws lambda - amazon web services*, https://aws.amazon.com/lambda/.

[3] Apache, *Apache openwhisk is a serverless, open source cloud platform*, https://openwhisk.apache.org/.

[4] Microsoft, *Azure functions – serverless functions in computing | microsoft azure*, https://azure.microsoft.com/en-us/products/functions/.

[5] J. Peng, Z. Zhou, *et al.*, “Cost-efficient cloud-edge video analytics with hybird iaas and faas resources”, in *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, 2023, pp. 2515–2522.

[6] E. Paraskevoulakou and D. Kyriazis, “Ml-faas: Toward exploiting the serverless paradigm to facilitate machine learning functions as a service”, *IEEE Transactions on Network and Service Management*, vol. 20, no. 3, pp. 2110–2123, 2023.

[7] Straits Research, *Function-as-a-service market size, share & trends report 2033*. https://straitsresearch.com/report/function-as-a-service-market, 2025.

[8] M. Shahrad, J. Balkind, *et al.*, “Architectural implications of function-as-a-service computing”, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52, Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 1063–1075.

[9] M. Tari, M. Ghobaei-Arani, *et al.*, “Auto-scaling mechanisms in serverless computing: A comprehensive review”, *Computer Science Review*, vol. 53, p. 100 650, 2024.

[10] N. Mahmoudi and H. Khazaei, “Performance modeling of metric-based serverless computing platforms”, *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 1899–1910, 2023.

[11] S. Agarwal, M. A. Rodriguez, *et al.*, “A reinforcement learning approach to reduce serverless function cold start frequency”, in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021, pp. 797–803.

[12] L. Schuler, S. Jamil, *et al.*, “Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments”, in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021, pp. 804–811.

[13] Z. Zhang, T. Wang, *et al.*, “Adaptive auto-scaling of delay-sensitive serverless services with reinforcement learning”, in *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2022, pp. 866–871.

[14] A. Zafeiropoulos, E. Fotopoulou, *et al.*, “Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms”, *Simulation Modelling Practice and Theory*, vol. 116, p. 102 461, 2022.

[15] P. Benedetti, M. Femminella, *et al.*, “Reinforcement learning applicability for resource-based auto-scaling in serverless edge applications”, in *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2022, pp. 674–679.

[16] S. Agarwal, M. A. Rodriguez, *et al.*, “ A Deep Recurrent-Reinforcement Learning Method for Intelligent AutoScaling of Serverless Functions ”, *IEEE Transactions on Services Computing*, vol. 17, no. 05, pp. 1899–1910, Sep. 2024.

[17] X. Li, P. Kang, *et al.*, “Kneescale: Efficient resource scaling for serverless computing at the edge”, in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 180–189.

[18] H. Zhao, S. Pan, *et al.*, “Faashark: An end-to-end network traffic analysis system atop serverless computing”, *IEEE Transactions on Network Science and Engineering*, vol. 11, no. 3, pp. 2473–2484, 2024.

[19] S. Pan, H. Zhao, *et al.*, “Sustainable serverless computing with cold-start optimization and automatic workflow resource scheduling”, *IEEE Transactions on Sustainable Computing*, vol. 9, no. 3, pp. 329–340, 2024.

[20] Z. Wen, Q. Chen, *et al.*, “Joint optimization of parallelism and resource configuration for serverless function steps”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 4, pp. 560–576, 2024.

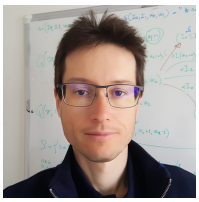
[21] J. Anselmi, B. Gaujal, *et al.*, “Non-stationary gradient descent for optimal auto-scaling in serverless platforms”, *IEEE Transactions on Networking*, pp. 1–14, 2025.

[22] J. Kiefer and J. Wolfowitz, “Stochastic estimation of the maximum of a regression function”, en, *Ann. Math. Statist.*, vol. 23, no. 4, pp. 462–466, 1952.

[23] D. Ardagna, M. Ciavotta, *et al.*, “A hierarchical receding horizon algorithm for qos-driven control of multi-iaas applications”, *IEEE Transactions on Cloud Computing*, vol. 9, no. 2, pp. 418–434, 2021.

[24] H. Qiu, W. Mao, *et al.*, "Reinforcement learning for resource management in multi-tenant serverless platforms", in *Proceedings of the 2nd European Workshop on Machine Learning and Systems (EuroMLSys '22)*, ACM, 2022, pp. 20–28.
[25] H. Qiu, W. Mao, *et al.*, "SIMPPO: A scalable and incremental online learning framework for serverless resource management", in *Proceedings of the 13th ACM Symposium on Cloud Computing (SoCC '22)*, ACM, 2022, pp. 306–322.
[26] H. Qiu, W. Mao, *et al.*, "AWARE: Automate workload autoscaling with reinforcement learning in production cloud systems", in *2023 USENIX Annual Technical Conference (USENIX ATC '23)*, No DOI assigned by the publisher, USENIX Association, 2023, pp. 387–402.
[27] Amazon Web Services, *Configuring AWS Lambda function memory*, https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html, Accessed: 2026-06-27, 2024.
[28] W. Mao, H. Qiu, *et al.*, "A mean-field game approach to cloud resource management with function approximation", in *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, 2022, pp. 36 243–36 258.
[29] N. Mahmoudi and H. Khazaei, "Performance modeling of serverless computing platforms", *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2834–2847, 2022.
[30] Y. Li, Y. Lin, *et al.*, "Serverless computing: State-of-the-art, challenges and opportunities", *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1522–1539, 2023.
[31] N. Mahmoudi and H. Khazaei, "SimFaaS: A Performance Simulator for Serverless Computing Platforms", in *International Conference on Cloud Computing and Services Science*, ser. CLOSER '21, Springer, 2021, pp. 1–11.
[32] M. Hanaforoosh, M. Ashtiani, *et al.*, "Mfs: A serverless faas simulator", in *2023 9th International Conference on Web Research (ICWR)*, 2023, pp. 81–86.
[33] P. Raith, T. Rausch, *et al.*, "Faas-sim: A trace-driven simulation framework for serverless edge computing platforms", *Software: Practice and Experience*, vol. 53, pp. 2327–2361, 12 Dec. 2023.
[34] B. A. Kalymon, "Stochastic prices in a single-item inventory purchasing model", *Operations Research*, vol. 19, pp. 1434–1458, 6 1971.
[35] N. Mahmoudi and H. Khazaei, "Performance modeling of serverless computing platforms", *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2834–2847, 2022.
[36] L. Wang, M. Li, *et al.*, "Peeking behind the curtains of serverless platforms", in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18, Boston, MA, USA: USENIX Association, 2018, pp. 133–145.
[37] J. Anselmi, "Asynchronous load balancing and auto-scaling: Mean-field limit and optimal design", *IEEE/ACM Transactions on Networking*, vol. 32, no. 4, pp. 2960–2971, 2024.
[38] S. Allmeier and N. Gast, "Computing the bias of constant-step stochastic approximation with markovian noise", 2024.
[39] S. Chandak, V. S. Borkar, *et al.*, "Concentration of contractive stochastic approximation and reinforcement learning", *Stochastic Systems*, vol. 12, no. 4, pp. 411–430, 2022.
[40] D. P. Bertsekas and J. N. Tsitsiklis, "Gradient convergence in gradient methods with errors", *SIAM Journal on Optimization*, vol. 10, no. 3, pp. 627–642, 2000.
[41] W. Li, X. Ma, *et al.*, "Analytical model and performance evaluation of long-term evolution for vehicle safety services", *IEEE Transactions on Vehicular Technology*, vol. 66, no. 3, pp. 1926–1939, 2017.
[42] J. Dean and L. A. Barroso, "The tail at scale", *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.
[43] N. AsSadhan and C. R. Johnson, "Modeling end-to-end delay using pareto distribution", in *Proceedings of the Second International Conference on Internet Monitoring and Protection (ICIMP 2007)*, IEEE, 2007, pp. 18–18.

Abednego Wamuhindo Kambale obtained his B.Sc. degree from ULPGL/Goma in Congo D.R. He is currently a Ph.D. student at Politecnico di Milano where he obtained his Master degree in computer sciences and engineering. His research interests include Artificial Intelligence for resource management in computing continuum.



Jonatha Anselmi is a tenured researcher at the French National Institute for Research in Digital Science and Technology (Inria) since 2014. Prior to this, he was a full-time researcher at the Basque Center for Applied Mathematics (BCAM) and a postdoctoral researcher at Inria. He received his PhD in computer engineering from Politecnico di Milano (Italy) in 2009. His research lies at the intersection of applied mathematics and computer science, with a focus on decision making under uncertainty.

Danilo Ardagna is Associate Professor at the Dipartimento di Elettronica Informazione and Bioingegneria at Politecnico di Milano. He received a Ph.D. degree in computer engineering in 2004 from Politecnico di Milano from which he also graduated in December 2000. His work focuses on the design, prototype and evaluation of optimization algorithms for resource management of computing systems.

Bruno Gaujal is a research director at Inria. Till Dec. 2015, he was the head of the large-scale computing team iat Inria and is currently a member of the learning theory team at Inria Grenoble-Alpes. He has also held several positions in AT&T Bell Labs, Loria (Nancy, France) and École Normale Supérieure of Lyon. He is a founding partner of a start-up company, RTaW, since 2007. His main interests are in performance evaluation, optimization and control of discrete event dynamic systems as well as learning algorithms for dynamic stochastic systems.