Jolteon: Unleashing the Promise of Serverless for Serverless Workflows

Zili Zhang, Chao Jin, and Xin Jin, *School of Computer Science, Peking University*

https://www.usenix.org/conference/nsdi24/presentation/zhang-zili-jolteon

This paper is included in the
Proceedings of the 21st USENIX Symposium on
Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Jolteon: Unleashing the Promise of Serverless for Serverless Workflows

Zili Zhang Chao Jin Xin Jin

School of Computer Science, Peking University

Abstract

Serverless computing promises automatic resource provisioning to relieve the burden of developers. Yet, developers still have to manually configure resources on current serverless platforms to satisfy application-level requirements. This is because cloud applications are orchestrated as serverless workflows with multiple stages, exhibiting a complex relationship between resource configuration and application requirements.

We propose Jolteon, an orchestrator to unleash the promise of automatic resource provisioning for serverless workflows. At the core of Jolteon is a stochastic performance model that combines the benefits of whitebox modeling to capture the execution characteristics of serverless computing and blackbox modeling to accommodate the inherent performance variability. We formulate a chance constrained optimization problem based on the performance model, and exploit sampling and convexity to find *optimal* resource configurations that satisfy user-defined cost or latency bounds. We implement a system prototype of Jolteon and evaluate it on AWS Lambda with a variety of serverless workflows. The experimental results show that Jolteon outperforms the state-of-the-art solution, Orion, by up to 2.3× on cost and 2.1× on latency.

1 Introduction

Serverless computing [1–6] aims to simplify cloud programming and relieve developers from infrastructure management. It exposes cloud functions as a key abstraction to developers. Developers use cloud functions to build cloud applications, and the serverless computing platform handles underlying hardware resource management. Serverless computing provides fine-grained resource elasticity and billing at the granularity of functions.

The potential benefits of serverless computing attract many applications, such as data analytics [7–9], video processing [10–12], and machine learning [13–15]. Applications are typically orchestrated as *serverless workflows* on serverless computing platforms [16–19]. Specifically, an application is decomposed into a set of functions, and a serverless workflow corresponds to a directed acyclic graph (DAG) that organizes these functions to implement the application logic.

Cloud platforms are expected to satisfy application-level requirements for cloud applications [20–22]. These requirements typically refer to latency and cost bounds. For example, a developer may expect the end-to-end latency to process a request for an application (i.e., the corresponding serverless workflow) to be no larger than 100ms, or the per-request cost to be no larger than $1. There exists a trade-off between latency and cost. Provisioning more resources results in lower latency but higher cost, and vice versa. Given a bound on one metric (e.g., a latency bound), it is desirable to provision resources to minimize the other metric (e.g., minimize cost).

Serverless computing promises automatic resource provisioning to relieve the burden of developers. Resource provisioning includes two parts, i.e., resource configuration (i.e., the function instance size and the number of function instances for each stage for a workflow instance) and resource scaling (i.e., the number of workflow instances). While current serverless platforms provide auto-scaling based on the realtime load, developers still have to manually configure resources for workflows to satisfy application-level requirements. AWS Lambda provides Power Tuning [23] for developers to profile the latency-cost curve for a particular function and optimize the configuration for one function based on the latency-cost curve. However, Power Tuning does not support tuning the serverless workflow.

Several recent works have explored resource configuration for serverless workflows [7–9, 24, 25]. Orion [24] uses a blackbox model to approximate the latency-cost curve for a workflow, and a heuristic algorithm to search for resource configurations. The nature of the blackbox approach results in inaccurate models and the resource configurations found by the heuristic are sub-optimal. Ditto [9] enables developers to optimize either latency or cost for a workflow. It only provides two extremes (minimum latency or minimum cost), and does not allow developers to explore other trade-offs between latency and cost. A 7% sacrifice on latency may bring a 1.8× reduction on cost, which may be more preferable than the configuration with the minimum latency.

We present Jolteon, a workflow orchestrator that provides automatic resource configuration to satisfy application-level requirements for serverless applications. Developers only need to specify either a latency or cost bound for a workflow. Jolteon automatically configures resources to minimize the execution latency for a cost bound or minimize the cost for a latency bound. By doing so, Jolteon delivers a serverless experience for workflow orchestration, and more importantly, enables developers to navigate the entire latency-cost Pareto front of the configuration space.

There are two challenges in realizing Jolteon. The first challenge is to build an accurate performance model to capture the complex relationship between resource configuration and application requirements. Existing works [7–9, 24, 25] rely

	Jolteon	Orion [24]	Ditto [9]	Caerus [8]	Locus [7]	Aquatope [25]	Stepconf [26]	CostPre [27]	Power-Tuning [23]	Cose [28]
Analytical performance model?	Y	N	Y	Y	Y	N	Y	N	Y	Y
Distribution-aware performance model?	Y	Y	N	N	N	Y	N	Y	N	N
Achieving the Pareto front and guaranteeing the performance bound?	Y	N	N	N	N	N	N	N	Y	Y
Supporting serverless workflow?	Y	Y	Y	Y	Y	Y	Y	Y	N	N

Table 1: The design space of Jolteon against existing works of serverless computing.

on either blackbox modeling or whitebox modeling. Blackbox models capture the performance variability of serverless computing, but lack explainability and ignore the execution characteristics of serverless workflows. In contrast, whitebox models employ deterministic formulas to represent the execution characteristics, which is faster and more accurate on average prediction. However, whitebox models ignore the inherent variability of serverless computing and fail to guarantee the performance bound. Jolteon builds a stochastic performance model. The model uses analytical formulas to capture the execution characteristics of workflow execution, which make the model more accurate and efficient (i.e., the benefit of whitebox modeling). The model uses stochastic functions to capture the performance variability, which can be used to bound the performance (i.e., the benefit of blackbox modeling). This approach leverages the benefits of whitebox and blackbox models, and avoids their drawbacks.

Given the stochastic performance model, the next challenge is to find the optimal resource configuration under a latency or cost bound. As we use random variables to model the execution variability, the problem is mathematically formulated as a chance constrained optimization problem, which is a type of stochastic optimization problem. To solve the problem, we first convert it to a deterministic formulation via Monte Carlo sampling and guarantee the performance bound through a novel inequality. Enumerating all possible configurations under the deterministic formulation is not practical given the large search space and the vast number of constraints. We prove that our formulation is convex. Thus, we leverage the convexity and use an efficient gradient descent algorithm to find the optimal resource configuration under the bound.

We implement a system prototype of Jolteon and evaluate it on AWS Lambda with a variety of serverless workflows. The experimental results show that Jolteon outperforms Orion [24] by up to 2.3× on cost and 2.1× on latency. Compared to Ditto [9] which provides either minimum latency or minimum cost, Jolteon is able to reduce cost by 1.8× or latency by 3.3×, with a $\leq$11% reduction on the other metric. The evaluation also confirms that Jolteon can satisfy different latency or cost bounds given by users.

In summary, we make the following contributions.

- We present Jolteon, a workflow orchestrator for serverless computing that provides automatic resource configuration to satisfy application-level requirements.
- We propose a stochastic performance model that captures both the execution characteristics and variability, and a bound guaranteed sampler to transform the stochastic optimization problem. We further prove the convexity of the problem and apply a gradient descent algorithm to find the latency-cost Pareto front of the configuration space.
- We implement a Jolteon prototype. The experimental results show that Jolteon outperforms the state-of-the-art solution, Orion, by up to 2.3× on cost and 2.1× on latency.

2 Background and Motivation

In this section, we first introduce the background of serverless computing and application workflows. Then we discuss the limitations of existing work, which motivate the design of Jolteon. Finally, we describe the technical challenges to find the optimal resource configuration that satisfies performance bounds for serverless workflows.

2.1 Serverless Workflows

Serverless computing simplifies cloud programming for cloud application developers. It enables developers to create short-lived, stateless functions triggered by events (e.g., HTTP request). Serverless platforms are responsible for resource scaling and fault tolerance, allowing developers to concentrate on application logic without managing cloud resources [5, 6]. Known for its high elasticity and pay-as-you-go billing, serverless computing meters function run-time at a fine granularity, one millisecond in major platforms [1–3], and bills users only for the resources consumed during function execution.

These benefits lead to a shift of traditional serverful applications to serverless platforms across many fields, including data analytics [7, 8], video processing [10–12], machine learning [13–15], and vector query processing [29]. A serverless application is orchestrated into a workflow, represented as a DAG. Each stage (i.e., node) in the DAG consists of a collection of parallel function instances. The quantity of cloud resources dedicated to a stage is the product of the number of function instances and the function size (e.g., the number of vCPUs). Edges in the DAG denote data dependencies between consecutive stages. The intermediate data is transmitted between stages via external storage services, such as S3 [30],

Developers expect that cloud applications are executed to meet different application-level requirements. For instance, they typically prioritize latency in *online* applications and

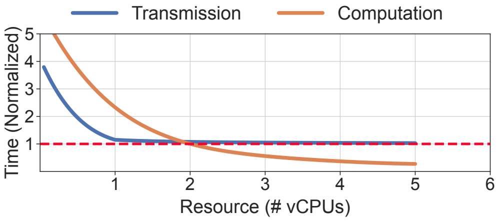


Figure 1: Ditto's performance model.

focus on cost efficiency for *offline* applications. A key requirement for workflow resource configuration is to minimize one metric (e.g., minimize cost) while adhering to the limit on another (e.g., a latency bound). Current serverless platforms only offer auto-scaling to scale the workflow instances, and delegate resource configuration (e.g., the function size and the number of function instances in a workflow stage) to developers. To meet the application-level requirements, developers have to understand the resource-to-performance mapping of serverless workflows. This is cumbersome and error-prone, as identifying the optimal trade-off between latency and cost (i.e., Pareto front) is complex and most developers lack expertise in the underlying system characteristics.

2.2 Existing Work and Challenges

In this subsection, we compare Jolteon with existing works and describe the challenges. Table 1 summarizes the key differences of Jolteon against other resource management systems of serverless computing.

Whitebox and analytical performance model. To achieve automatic resource configuration, an essential prerequisite is to capture the resource-to-performance relationship with a comprehensive and accurate performance model. Several works [7–9] leverage whitebox approaches to deterministically predict the function execution time with analytical equations. For instance, Locus [7] observes that the data shuffling time decreases with the increase in resources. Based on Locus, Caerus [8] and Ditto further divide the execution into several fine-grained steps (e.g., transmission and computation). They build individual equations for each step based on its logic.

As depicted in Figure 1, Ditto's performance model divides function execution into transmission and computation steps. The model recognizes that the transmission time remains constant when the number of vCPUs surpasses one, given that a single vCPU can fully saturate the network bandwidth. Hence, it employs a piecewise function, using one vCPU as the boundary, to characterize the relationship between resources and transmission time. These efforts capture the underlying logic of serverless computing and achieve fast and accurate predictions of *average* performance. However, the model neglects the intrinsic performance *variability* in serverless computing, thus failing to guarantee the performance bound.

Blackbox and distribution-aware performance model. Other works [24, 25, 27] note that real-world serverless platforms suffer from performance *variability*. This variability stems from the skew of the input data characteristics, network traffic, workflow structure, and invocation pattern, etc. To address this, these works [24, 25, 27] employ distribution-aware performance models to capture such performance variability. Specifically, they first collect function execution traces as sampling data, and then fit a blackbox model to cover the function execution process.

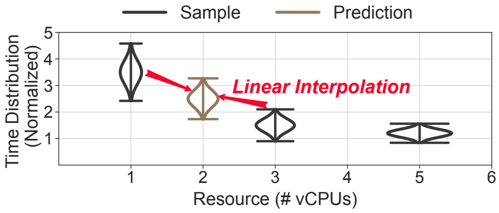


Figure 2: Orion's performance model.

Among these, Orion [24] utilizes a linear interpolation method. As depicted in Figure 2, the two sampling distributions are illustrated by probability distribution functions. The predicted distribution is fitted through linear interpolation. Aquatope [25] and CostPre [27] rely on Bayesian optimization and mixture density networks, respectively. Although these techniques capture the performance variability, they overlook underlying serverless computing characteristics, resulting in imprecise predictions and time-consuming blackbox fitting. The characteristics of serverless computing refer to the step-by-step breakdown of function execution, i.e., initialization, transmission, and computation. We can exploit the characteristics of each step to build a more accurate and efficient performance model, e.g., the time of transmission step is influenced by the available network bandwidth. We will discuss the details in §4.1.

Pareto front and performance bound. Optimizing the cost or latency for serverless workflows has been explored in prior work. Caerus [8] and Locus [7] utilize proportional resource allocation based on input data size. However, for applications insensitive to data size, these approaches yield sub-optimal performance. To rectify this, Ditto [9] introduces a bottom-up DAG traversal algorithm to minimize either cost or latency, and achieves near-optimal results. Nevertheless, these studies do not support finding the optimal resource configuration under specific performance bounds, and their deterministic models fail to guarantee the performance bound under the variability of serverless computing.

Orion [24], Stepconf [26] and Aquatope [25] aim to reduce cost within a pre-defined latency budget. Orion, considered as the state-of-the-art, leverages a best-first search algorithm that explores resource configurations and selects the first one that satisfies the given latency constraint. Similarly, Stepconf applies heuristic search methods, while Aquatope resorts to Bayesian techniques. Notably, the search space of the three systems is limited, and they cannot enumerate all possible resource configurations. Consequently, they fail to achieve the Pareto front between latency and cost while guaranteeing the performance bound.

Supporting serverless workflows. At the function level, some works such as AWS Lambda Power Tuning [23] and Cose [28] characterize the deterministic latency-cost curve for specific functions. They profile the latency-cost curve for a particular function, and allow users to optimize latency or cost under a given bound of the other metric. However, a serverless workflow (i.e., a DAG) is composed of many functions with varying data dependencies. These works are inadequate for supporting the optimization of serverless workflows.

Challenges. Our goal is to provide automatic resource configuration to meet application-level requirements. Specifically, the system is expected to configure resources automatically to minimize the execution latency for a user-specified cost bound or minimize the cost for a user-specified latency bound. To achieve this goal, two main challenges must be addressed.

The first challenge is to build a performance model to accurately capture the complex relationship between resource configurations and application requirements (e.g., latency and cost). As described above, whitebox models successfully capture the underlying logic of serverless computing and therefore provide fast and accurate predictions of average performance. However, their deterministic nature fails to capture the performance variability. Blackbox models, on the other hand, concentrate on the performance variability but overlook the underlying system characteristics, resulting in imprecise predictions and time-consuming blackbox fitting. It is desirable to develop a performance model that leverages the benefits of whitebox and blackbox while mitigating their shortcomings.

The second challenge is to identify the optimal resource configuration within the given performance bound. With the performance model including distributional factors, the optimization problem becomes stochastic which cannot be solved directly. Even if the problem is deterministic, the huge search space of resource configurations and intricate formulation of the problem make it hard to find the optimal solution. In short, the second challenge lies in the formulation of the stochastic optimization problem with performance guarantees and efficiently solving the problem to find the optimal result.

3 Jolteon Overview

Jolteon is a serverless workflow orchestrator that facilitates automatic resource configuration to meet application-level requirements (i.e., latency or cost). It employs a novel stochastic performance model to accurately profile the resource-to-performance relationship in serverless computing (§4.1). It formulates the chance constrained optimization problem. It then converts it into a deterministic problem with Monte-Carlo sampling and guarantees the performance bound via a novel inequality (§4.2). It employs a gradient descent algorithm that leverages the problem's convexity to achieve the optimal result (§4.3). Figure 3 shows the overview.

User interface. Users define their serverless workflow DAGs and requirements (latency or cost bounds) as input to Jolteon. A workflow DAG includes the functions for each stage and the data dependencies between the stages. Jolteon assesses the validity of a DAG (e.g., confirming the absence of cycles) and deploys the functions to the serverless platform. Jolteon then processes the invocation requests for the workflow, and executes the workflow with the optimal resource configuration. The results are returned upon the completion of the requests.

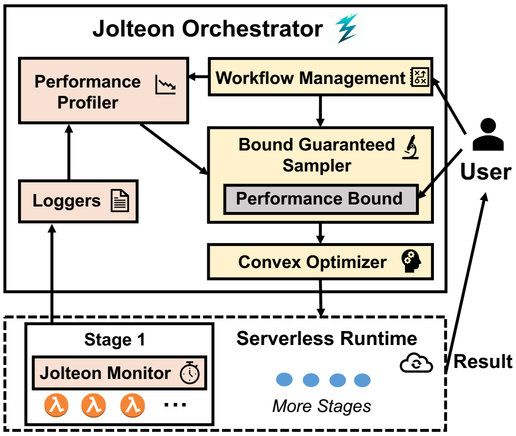


Figure 3: Jolteon overview.

Jolteon orchestrator. The orchestrator receives workflow DAGs and requirements from users, and generates resource configurations. It contains the following components.

Performance profiler. After a workflow is registered, Jolteon's performance profiler periodically polls the data logs of the corresponding workflow. It learns and updates the performance model (§4.1) of each workflow stage, modeled by stochastic functions. The stochastic functions are able to leverage the advantages of both whitebox and blackbox models while mitigating their shortcomings. Specifically, the performance model treats every parameter in the analytical formula as a random variable and fits it as a distribution.

Bound guaranteed sampler. Jolteon formulates the chance constrained optimization problem based on the workflow information and user-specified bound. In the formulation, resource configurations are regarded as independent variables. Since chance constrained optimization cannot be solved directly, this module converts it into a deterministic problem through Monte-Carlo sampling. More samplings lead to a higher confidence level, but more complicated formulation with larger solving time. Jolteon proposes a novel inequality to decide the minimal sample size that guarantees the performance bound with a high confidence level (§4.2).

Convex optimizer. After sampling, the chance constrained optimization problem is converted into a deterministic optimization problem. Solving such problem with existing algorithms [31] is both time-consuming and sub-optimal. Jolteon leverages one key insight: the problem is convex. Consequently, Jolteon employs a gradient descent algorithm to efficiently find the optimal resource configuration (§4.3). Moreover, Jolteon prunes the constraints with the support constraint

Symbol	Description
d_i	The number of function instances of the i_{th} stage
v_i	The number of vCPUs of one function in the i_{th} stage
$\mathbf{d}$	The array of d_i of all stages
$\mathbf{v}$	The array of v_i of all stages
$T_j(d,v)$	The transmission time of the j_{th} function in one stage
$C_j(d,v)$	The computation time of the j_{th} function in one stage
$LS_i(d,v)$	The latency of the i_{th} stage
$CS_i(d,v)$	The cost of the i_{th} stage
$LW(\mathbf{d},\mathbf{v})$	The latency of the entire workflow
$CW(\mathbf{d},\mathbf{v})$	The cost of the entire workflow

Table 2: Key notations in the design.

technique to further reduce formulation complexity. The configuration is then sent to the serverless runtime for execution.

Serverless runtime. The serverless runtime receives the serverless workflow and the optimal resource configuration from the orchestrator. It then executes (i.e., through function invocations) the workflow and records the execution logs. The logs are used by the profiler to build the performance model. The execution results are returned to the users.

4 Jolteon Design

In this section, we first describe the stochastic performance model (§4.1). Then we formulate the chance constrained optimization problem and introduce the bound guaranteed sampler (§4.2). Finally, we introduce the convex optimizer to solve the problem (§4.3). The key notations are listed in Table 2. Lowercase symbols represent deterministic values (e.g., x), uppercase symbols represent random variables (e.g., X), and bold symbols represent vectors (e.g., $\mathbf{x}$).

4.1 Performance Profiler

Jolteon's performance profiler collects the function logs to periodically update the stochastic performance model. In this subsection, we introduce the details of the stochastic performance model which integrates the benefits of both whitebox and blackbox models while mitigating their drawbacks.

The model first divides each workflow stage (i.e., each node of the DAG) into *functions* and *phases*. A stage consists of many parallel function instances. The latency of the *stage* is the *maximum* latency of all function instances. Each function instance is divided into two phases: *initialization* and *execution*. The initialization phase refers to setting up the function environment while the execution phase is to run the user code. The phase is further divided into fine-grained *steps* as follows.

Initialization phase. The initialization phase is to receive the request and set up the function environment. The first step involves a network delay and front door execution. The network delay spans from the moment the function is triggered to when it reaches the API gateway. It is inherently variable, influenced by the network status. Subsequently, front door execution includes request authentication, function routing, and load balancing. The latency of this process is likewise unpredictable, shaped by the overall system load. This step is inevitable and irrelevant to resource configuration. We have gathered data on the latency from AWS Lambda, where functions are triggered at 30-second intervals via HTTP requests. We omit the failed invocation requests and only record the time of requests with a status code of 200. Evidently, this latency is not stable, as illustrated in Figure 4. We model the latency with a random variable, denoted by D.

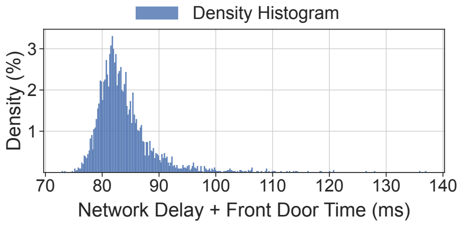


Figure 4: Time distribution of the first step in initialization.

The next step is to set up the function environment. The serverless platform determines if a corresponding function instance (e.g., VM or container) is present in the instance pool. In the absence of such an instance (i.e., cold start), the platform will allocate a new one, pull the function image, and initiate the function runtime. Conversely, if an instance does exist (i.e., warm start), the existing function instance is reused. The latency is negligible for warm start. Jolteon profiles cold or warm start according to the specific policy (e.g., pre-warming or keep-alive) and employs the following formula to approximate the time of this step, where C is a random variable of cold start time.

$$G = \begin{cases} 0, & \textit{Warm start} \\ C, & \textit{Cold start}. \end{cases}$$

Execution phase. In the execution phase, the function includes two steps: data transmission and computation. Typically, a workflow stage downloads the data from its preceding stage, processes the data, and uploads the result to its subsequent stage. The allocation of computational resources has a notable influence on the latency of these two steps. Specifically, more vCPUs result in higher network bandwidth and higher computational capacity.

Transmission step. In the data transmission step, data is either downloaded from or uploaded to external storage. We model downloading as an example. Relevant factors for this step include the data size, the available network bandwidth, and the API overhead. The input data may vary due to the diversity of data sources and the stochastic nature of data generation. We model input data size (denoted as S) as a random variable. The stage comprises d parallel function instances. Let the available network bandwidth and the number of vCPUs of a function instance be b and v, respectively. Our goal is to formulate a model for the transmission time of a function, based on the given resource configuration, d and v.

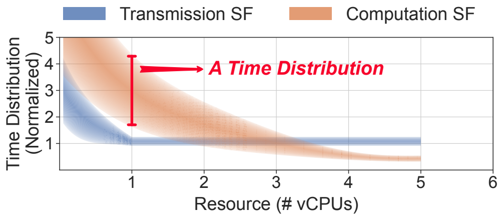


Figure 5: Jolteon's stochastic performance model.

The data transmission time for a serverless function has two components: the propagation time and API invocation overhead. The propagation time is influenced by the data size and the available network bandwidth. The data of the entire stage (with size of S) is partitioned into d parts, one for each function instance. The propagation time of each function instance is $S/(d \times b)$, where b represents the available bandwidth. b scales with the number of allocated vCPUs (i.e., v). However, the bandwidth reaches a saturation point when v is sufficiently large. Thus, b is approximated by $min(v \times W, B)$, where B is the maximum bandwidth and W is the per-vCPU bandwidth. The API invocation overhead is independent of resource allocation and denoted by O_T. Therefore, the transmission time for a function instance is summarized as follows:

$$T(d,v) = \frac{S}{d \times min(v \times W,\ B)} + O_T. \tag{1}$$

Computation step. In the computation step, each function handles a data subset of size S/d with v vCPUs. However, the function logics are diverse. For example, in the frame extraction stage of video analytics, the function ingests S/d videos and cycles through each frame with multi-core processing. The computation time is $A \times \frac{S}{dv}$, where A is the time to process one video. Conversely, in the map stage of data analytics, the function sorts the S/d data units. Since the time complexity of sort is $O(x \ln x)$, the time is $B \times \frac{S}{dv} \times \ln \frac{S}{dv}$, where B is the time to operate one data unit with one vCPU.

To tackle such complexity, we propose a model that mathematically characterizes the distinct computational logics with appropriate equations. Specifically, we focus on two typical function logics: those with polynomial complexity and logarithmic complexity. The execution time of polynomial logic is represented by $\sum_{i=0}^{l} A_i \times x^i$, and the time for logarithmic logic is $\ln x \times (\sum_{i=0}^{m} B_i \times x^i)$. Here, A_i and B_i are positive random variables and x is defined as $\frac{S}{dv}$. For functions without the relevant logic, the coefficients are set to zero. Consequently, the computation time of a function instance is expressed as:

$$C(d,v) = \sum_{i=0}^{l} A_i \times (\frac{S}{dv})^i + \ln \frac{S}{dv} \times \sum_{i=0}^{m} B_i \times (\frac{S}{dv})^i. \tag{2}$$

l and m are decided when fitting the parameters (A_i, B_i) with historical data. Specifically, A_i is fitted one by one in ascending order. The fitting process terminates when a specific number of zero A_i are encountered, and l is the last index i. The profiler monitors if the function supports multi-processing. If latency remains the same when $v > 1$, it is deemed single-processing. Consequently, the function's performance model restricts v's feasible domain to $(0,1]$. Another concern is that Formula 2 is not applicable to function logic with other complexities (e.g., exhaustive search with exponential complexity rather than polynomial complexity). However, since the feasible domain of v and d is discrete and finite (e.g., $v \in (0,6]$), other complexities can be approximated by the above polynomial complexity, i.e., Lagrange interpolating polynomial.

Function model to stage model. The aforementioned performance model pertains to an individual function instance. We must consider the d parallel function instances for a stage. The latency of the entire stage is the maximum latency of the d parallel functions. We define the stage latency as $LS(d,v)$:

$$LS(d,v) = \max\left\{D_j + G_j + T_j(d,v) + C_j(d,v), j = 1 \dots d\right\}. \tag{3}$$

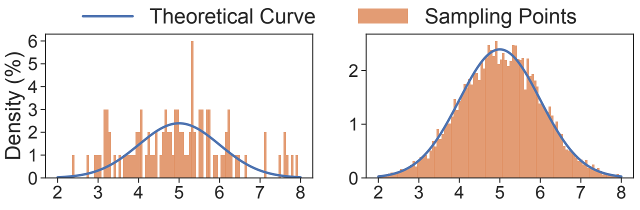
Regarding the cost of a stage, current serverless platforms do not bill users for initialization time. Therefore, the cost is defined as follows, where α is the cost per second of one vCPU and β is the cost per invocation.

$$CS(d,v) = \sum_{j=1}^{d} \big((T_j(d,v) + C_j(d,v)) \times v \times \alpha + \beta\big). \tag{4}$$

Stochastic model. The above formulas serve as analytical (i.e., whitebox) models, and the parameters (e.g., C, W, A_i) are fitted with historical data. The independent variables d and v are the resource configuration. As we discussed in §2.2, treating the parameters (e.g., C, W, A_i) as static values fails to account for the inherent variability. We instead model these parameters as random variables, transforming the deterministic functions to stochastic functions. These random variables are fitted as distributions and marked as uppercase letters. As shown in Figure 5, when the resource configuration is given, the stochastic function (SF) outputs a distribution of the predicted latency rather than a fixed value. Compared to blackbox models, this stochastic model captures the underlying system characteristics through the simple yet potent formulas, which allows higher accuracy and lower fitting overhead. In short, this stochastic model captures not only the variability but also the underlying system characteristics, which leverages the benefits of both whitebox and blackbox models.

4.2 Bound Guaranteed Sampler

Formulation of chance constrained optimization. We first extend the above stage performance model to the entire workflow. In the workflow DAG, the latency of the complete workflow is modeled as the `maximum` latency of paths in the DAG. One path is a sequence of stages that are connected by edges and its latency is the `sum` of all sequential stages' latency. The workflow latency, denoted as $LW(\mathbf{d}, \mathbf{v})$, is calculated through



(a) Sample size = 100.

(b) Sample size = 10,000.

Figure 6: Sampling under different sample size.

either addition or maximum on stage latency, $LS_i(d_i, v_i)$. Here, $\mathbf{d}$ and $\mathbf{v}$ are p-dimensional arrays where d_i and v_i specify the resource configuration of the i-th stage (p stages in total). As for the cost of the workflow, denoted as CW, it is the sum of all stage cost, i.e., $CW(\mathbf{d}, \mathbf{v}) = \sum_{i=1}^{p} CS_i(d_i, v_i)$.

We then mathematically formulate this stochastic problem as a chance constrained optimization problem. The objective is to minimize either latency or cost, subject to the constraint of guaranteeing the performance bound. The problem is formulated as follows (with cost as the objective function):

$$\begin{aligned} & Minimize \quad CW(\mathbf{d}, \mathbf{v}). \\ & s.t. \quad Confidence\big(LW(\mathbf{d}, \mathbf{v}) \leq \varepsilon_l\big) \geq \delta. \end{aligned} \tag{5}$$

In this formulation, certain parameters are random variables and the constraint mandates that the latency bound ε_l be guaranteed with a predetermined confidence level δ. The formulation to minimize latency under a cost bound is similar.

Bound guaranteed sampling. Chance constrained optimization, a subfield of stochastic optimization, deals with uncertain parameters in constraint functions, while ensuring the constraints are guaranteed with a high confidence level. Such problem has been extensively studied in various research fields [32–34]. However, adapting this method to our specific problem introduces two challenges.

The first challenge is that the objective and constraint functions contain random variables. Jolteon employs Monte Carlo sampling to transform the chance constraint into a set of deterministic constraints. As for the objective function, the random variables are usually regarded as their expectation values in optimization. Let function $G(\mathbf{d}, \mathbf{v}, \mathbf{X})$ be the constraint function in Formula 5, where $\mathbf{X}$ is the set of the random variables defined in §4.1, represented as a random vector. For example, $G(\mathbf{d}, \mathbf{v}, \mathbf{X}) = LW(\mathbf{d}, \mathbf{v}) - \varepsilon_l$ if user specifies a latency bound. Monte Carlo sampling is used to sample n vectors from the random variables in the constraint functions, e.g., $\mathbf{x_i}$ is a deterministic sample vector of $\mathbf{X}$. This converts the original stochastic constraint into a series of deterministic constraints:

$$\{G(\mathbf{d}, \mathbf{v}, \mathbf{x_i}) \leq 0, \; for \; i = 1...n\}. \tag{6}$$

All random variables are replaced by their corresponding sample values. The deterministic problem can be solved directly.

The second challenge is to determine the number of samples to ensure the performance bound with a high confidence level. Conceivably, the more samples, the higher confidence the solution has. For example, assume that the random vector, $\mathbf{X}$, only contains one random variable, X. X conforms to normal distribution as shown in Figure 6. The figure contrasts two different sample sizes: Figure 6(a) uses 100 samples, while Figure 6(b) uses 10,000 samples. More samples yield a higher confidence. Specifically, the prediction error for the 95% percentile is 13% in Figure 6(a) and only 0.15% in Figure 6(b). However, a large sample size introduces high sampling overhead and intricate problem formulation with large solving time. The two illustrations in Figure 6 exemplify a single random variable on two specific sample sizes. However, the actual problem arises with many random variables, which further complicates the determination of sample size.

Jolteon leverages Hoeffding's inequality and sample approximation theory to find the lower bound of sample size n. This lower bound allows Jolteon to guarantee the performance bound at confidence level δ while minimizing the sampling overhead and simplifying the problem formulation. The lower bound of sample size is as follows:

$$\frac{1}{2 \times (1 - percentile)^2} log(\frac{|D|}{1-\delta}). \tag{7}$$

In this content, D denotes the domain of independent variables $\mathbf{v}$ and $\mathbf{d}$. δ refers to the confidence level. Jolteon sets δ to 99.9% which is large enough for most user cases. *percentile* refers to the percentile for the target bound, such as P95 latency bound. The two parameters are both configurable for developers. The theoretical proof of the lower bound of n to guarantee the performance bound is given in Appendix A.1. The proof is based on Hoeffding's inequality [35] and sample approximation theory [36, 37].

The definition domain, D, depends on the structure of workflow DAG and is a large value due to the large resource configuration space. In real workflow executions, n may reach up to thousands, according to Formula 7. However, the subsequent convex optimizer reduces such number with pruning technique and the evaluation (§6.4) shows that solving with thousands of constraints is still efficient and fast.

4.3 Convex Optimizer

The sampler transforms the chance constrained optimization problem (Formula 5) into a deterministic problem with a set of constraints, i.e., random variables become the sampling values. Thus, we replace the uppercase symbols with lowercase symbols. However, solving such problem at runtime remains a challenge due to the complex constraints and huge search space. Jolteon leverages an important insight: the optimization problem is *convex*. A convex problem means the objective function and all constraints are convex functions. This confers a significant advantage—the local extremum also serves as the global extremum, enabling rapid, optimal solutions via established algorithms [31]. We first prove the convexity of the above deterministic problem.

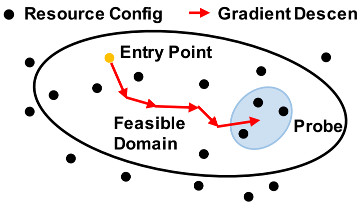


Figure 7: Process diagram of optimization algorithm.

Convexity analysis. According to the definition of convex optimization problems [38], we need to prove that $CW(\mathbf{d},\mathbf{v})$ and $LW(\mathbf{d},\mathbf{v})$ are convex, where $(\mathbf{d},\mathbf{v})$ are independent variables and all other parameters are constants after sampling.

Since the sum and maximum of convex functions remain convex, we only need to prove that $CS_i(d_i,v_i)$ and $LS_i(d_i,v_i)$ are convex. Formula 3 and 4 further simplify the proof by narrowing the focus on proving $T(d,v)$ and $C(d,v)$ are convex. We focus on one stage and omit the index i. All variables and constants are positive due to their real-world meanings.

For the computation step, $C(d,v)$ (i.e., Formula 2), is the sum of two parts: the polynomial part and the logarithmic part. Function $f(x)=x^i$ is convex and monotonically increasing. $\frac{s}{dv}$ is convex because its Hessian matrix is positive-definite. Therefore, the polynomial part is convex due to the convexity of composite functions. Since s is significantly larger than $d \times v$ due to its real-world meaning (i.e., $\frac{s}{dv} > 1$), the logarithmic part is also convex according to its positive-definite Hessian matrix. Therefore, $C(d,v)$, the sum of two parts, is a convex function. For the transmission step, $T(d,v)$ (i.e., Formula 1), is also a convex function due to the convexity of piecewise maximum functions, where s,w,o_t are constants.

The detailed analysis of the convexity of the above sum, maximum, composite, logarithmic, polynomial, and piecewise maximum functions is given in Appendix A.2. In summary, the sampling deterministic problem is convex.

Optimization algorithm. Based the convexity, we propose an efficient algorithm to optimally solve the problem, as outlined in Figure 7. Each point in the figure represents one resource configuration (i.e., $\mathbf{d},\mathbf{v}$). The feasible domain circles out the points that satisfy the sampling constraints. The first procedure is a gradient descent algorithm. It starts from a random entry point and takes iterative steps following the gradient until it approaches a local extremum. The second procedure is a probe process. Due to the discrete nature of resource configurations in the real world, the continuous local extremum may not be feasible. To address this, the probe process iteratively examines feasible points surrounding this extremum to find the final configuration. Given that the problem is convex, the local extremum is also globally optimal. Thus, the optimization algorithm is capable of identifying the optimal solution under a given performance bound (i.e., Pareto front).

An excessive number of sample constraints complicate the optimization process and exacerbate the constraints checking overhead. To mitigate this, we employ a support constraint technique to prune redundant constraints. This technique identifies a minimal subset of $\{c_1,c_2,...,c_\alpha\}$ while preserving the feasible domain. If $\mathbf{x_i} < \mathbf{x_j}$ ($\mathbf{x_i}$ is the parameter vector of c_i), then the feasible domain defined by c_j becomes a subset of that defined by c_i. Thus, c_i can be pruned. This is valid under the assumption that all constants and variables are positive and the inequality sign of constraint is $\leq$. For example, if c_1 is $x+y \leq 1$ and c_2 is $2x+2y \leq 1$, the feasible domain of c_2 is encompassed by that of c_1 when $x,y>0$.

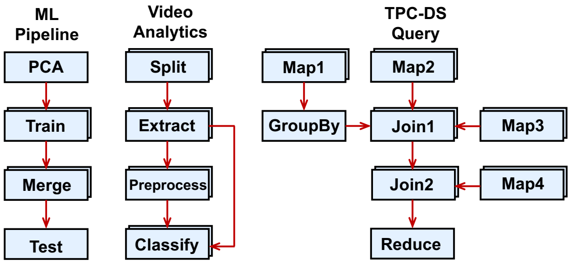


Figure 8: DAGs of the serverless workflows in the evaluation.

The solving time of convex optimizer occupies the majority of Jolteon's orchestration time. In particular, the solving time escalates rapidly with the increasing number of constraints and complexity of the workflow DAG. In §6.4, we evaluate the solving time of the optimizer and show that it is efficient and fast under most use cases. In summary, Jolteon's convex optimizer is capable of identifying the Pareto front, and is efficient with the support constraint pruning technique.

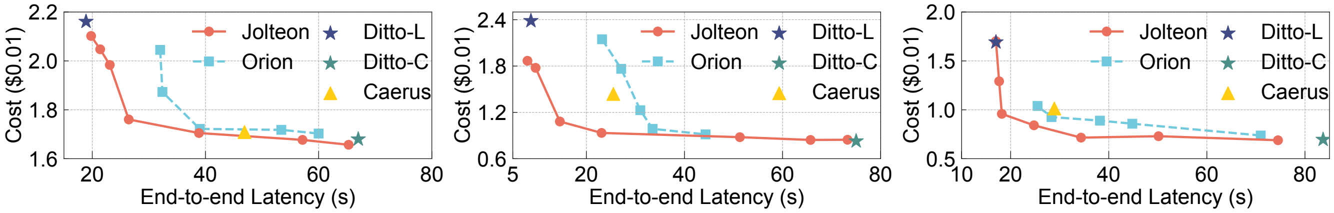
5 Implementation

We implement a system prototype with ~3,800 lines of code in Python. Our prototype supports AWS Lambda [1] as the serverless platform, and uses AWS S3 [30] as the external storage. Jolteon handles execution failures by re-executing the serverless workflow. Existing fault tolerance mechanisms for serverless workflows [39–41] are orthogonal to Jolteon. The code of Jolteon is open-source and is publicly available at https://github.com/pkusys/Jolteon.

Performance profiler. The performance profiler periodically polls the data from the serverless runtime and fits a stochastic performance model through `SciPy` [42]. We employ the non-linear least squares method to fit the parameters. The profiler updates the expectation value and covariance matrix of each parameter to characterize the distribution. These distributions are then used to generate samples by the sampler.

Bound guaranteed sampler. The sampler module generates a set of sample constraints to transform the original problem into a deterministic problem. The sample size is determined through Formula 7. Since the distributional parameters constitute a multivariate distribution, the sampling process is implemented through joint probability distribution with each parameter's expectation value and covariance matrix.

Convex optimizer. After sampling, we employ the SLSQP (Sequential Least Squares Quadratic Programming) method as the gradient descent algorithm in Figure 7. This method uses a



(a) TPC-DS Query.

(b) Video Analytics.

(c) ML Pipeline.

Figure 9: Overall performance for different serverless workflows.

series of quadratic approximations to find the optimum under a set of nonlinear constraint functions. Given that it performs optimally when the objective function and constraints are convex, it aligns well with the nature of our problem. The probe procedure is implemented by breadth first search at a fixed depth to iterate the points around the local extremum.

6 Evaluation

In this section, we evaluate Jolteon from the following aspects: (*i*) overall performance against state-of-the-art solutions (§ 6.1); (*ii*) performance guarantees provided by Jolteon (§ 6.2); (*iii*) effectiveness of the performance model (§ 6.3); (*iv*) time analysis for running Jolteon (§ 6.4). We also evaluate the sensitivity of Jolteon to various initial configurations (i.e., entry points) in Appendix A.3.

Setup. We conduct all experiments on AWS. The Jolteon orchestrator is deployed on one c5.12xlarge EC2 instance with 48 vCPUs and 96 GB memory. Jolteon executes the serverless workflow applications on AWS Lambda [1] and uses AWS S3 [30] as the external storage.

Workloads. Our experiments use three representative serverless applications. The workflows of these applications have different characteristics in terms of the number of stages, data dependencies, I/O and compute demands. Figure 8 shows the three workflow DAGs, and the details are as follows.

- **ML Pipeline** is a machine learning workflow adopted from Cirrus [13]. It consists of four stages connected as a chain: dimensionality reduction (PCA), model training, merging, and testing. Using the `LightGBM` library [43], the training stage runs multiple parallel functions to train a set of decision tree models, and the merging stage combines them into a random forest. Each workflow execution trains on 5K images and tests on 2K images of MNIST dataset [44].
- **Video Analytics** is adopted from Pocket [45] and is composed of four stages: video splitting, frame extraction, preprocessing and frame classification. There exists one branch in the workflow, where some frames are preprocessed and then classified by a pre-trained YOLO model [46], while the others are directly dispatched to classification. Each workflow execution processes 32 YouTube videos in "Music" and "News" categories, each with one minute duration.
- **TPC-DS Query** is a data analytics job (Query 95 in TPC-DS benchmark [47]). This workflow is composed of eight stages with complex dependencies. The stages perform a series of `filter`, `groupby`, and `join` operations. Each workflow execution processes 10 GB TPC-DS data.

Baselines. We compare Jolteon with the following baselines.

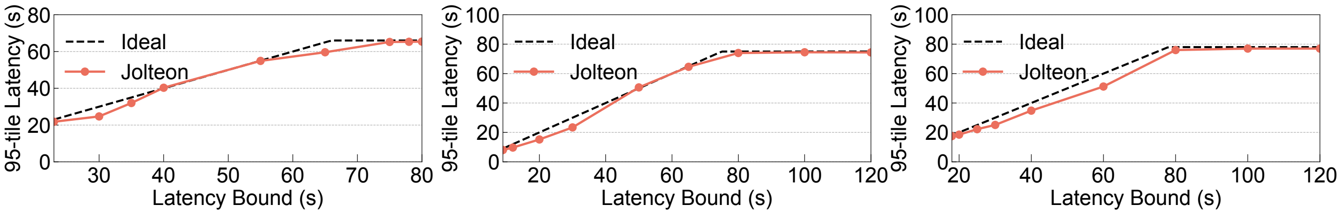
- **Caerus** [8] is a serverless scheduler that uses a heuristic proportional resource allocation strategy based on input data size to optimize latency and cost for a workflow.
- **Ditto** [9] is a state-of-the-art serverless scheduler that utilizes whitebox modeling to provide either minimum latency or minimum cost for a workflow execution, which we refer to as **Ditto-L** and **Ditto-C**, respectively.
- **Orion** [24] is a state-of-the-art serverless scheduler that employs blackbox modeling on the function instance size and uses deterministic numbers of parallel functions. It then uses a heuristic search algorithm with the blackbox model to minimize cost under varying latency requirements.

Metrics. We use end-to-end latency and cost of the workflow execution as the main metrics. The latency refers to the time span from the submission of a workflow request to the receipt of its execution results by the user. The cost is the expense for the serverless function execution, which is extracted from AWS Lambda's "Billed Duration" log entry. We do not use throughput, because throughput is determined by the number of workflow instances, which is controlled by the auto-scaling mechanism of serverless platforms, and the throughput of a single instance, which is the reciprocal of latency.

6.1 Overall Performance

We first compare the overall performance of Jolteon against the baselines. For systems that can meet varying performance requirements (i.e., Jolteon and Orion), we set different latency and cost bounds to obtain the latency-cost curve. For those that do not explore the latency-cost curve (i.e., Ditto and Caerus), we measure the best performance they can achieve. We run the workflows under each resource configuration several times and report the average latency and cost. The results are shown in Figure 9, which we summarize as follows.

- Jolteon outperforms Caerus in *both* latency and cost. Specifically, Jolteon achieves 1.75× lower latency and 1.33× lower cost than Caerus for Video Analytics. This is because the performance model of Caerus only considers the input data size rather than the inherent logic of a serverless function. As a result, it fails to capture the performance accurately and thus the resource configuration is sub-optimal.

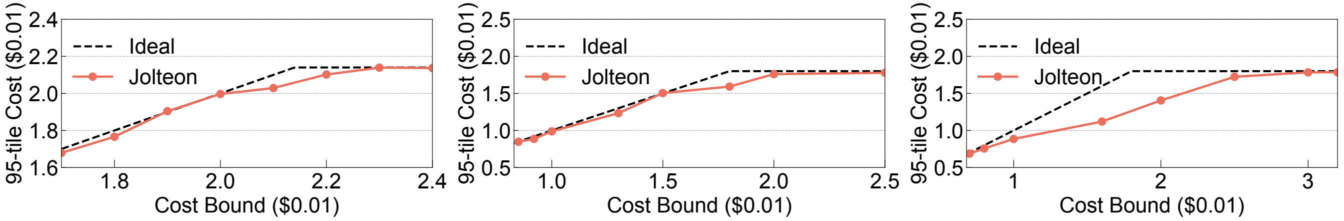


(a) TPC-DS Query.

(b) Video Analytics.

(c) ML Pipeline.

Figure 10: Latency guarantee of Jolteon.



(a) TPC-DS Query.

(b) Video Analytics.

(c) ML Pipeline.

Figure 11: Cost guarantee of Jolteon.

- Ditto-L and Ditto-C achieve minimum latency and cost, respectively. However, Ditto does not support the trade-off on the Pareto-optimal latency-cost curve. In serverless workflow execution, a small sacrifice on one metric (e.g., latency) may bring a significant reduction on the other metric (e.g., cost). Compared to Ditto, Jolteon is able to trade-off between latency and cost. With only 7% sacrifice on latency, Jolteon reduces the cost by up to 1.77× compared with Ditto-L. And an 11% increase on cost enables Jolteon to reduce the latency by 2.44–3.25× against Ditto-C.
- Orion approximates the latency-cost curve and is able to meet varying performance requirements. But it uses heuristic search that returns as soon as one configuration meets the performance requirement and only adjusts function sizes. This approach significantly narrows the algorithm's search space and is far from the Pareto-optimal. Compared to Orion, Jolteon achieves 1.45–2.07× on latency with the same cost, and 1.04–2.3× on cost with the same latency.

6.2 Performance Guarantees

This set of experiments evaluates the effectiveness of Jolteon to provide performance guarantees. We set the percentile of the target performance requirement to 95%.

Latency guarantee. To evaluate the latency guarantee of Jolteon, we set different latency bounds and measure the latency. Figure 10 shows the actual 95% latency and the ideal latency for the three workflows. The measured actual 95% latency is consistently less than and close to the ideal line, which demonstrates the capability of Jolteon to provide latency guarantee. When we increase the latency bound, the actual 95% latency monotonically increases, indicating that Jolteon can adapt to varying latency bounds. The actual latency does not increase when the bound is loose (e.g., >80 seconds for Video Analytics in Figure 10(b)), since the resource configuration hits the floor to enable the execution.

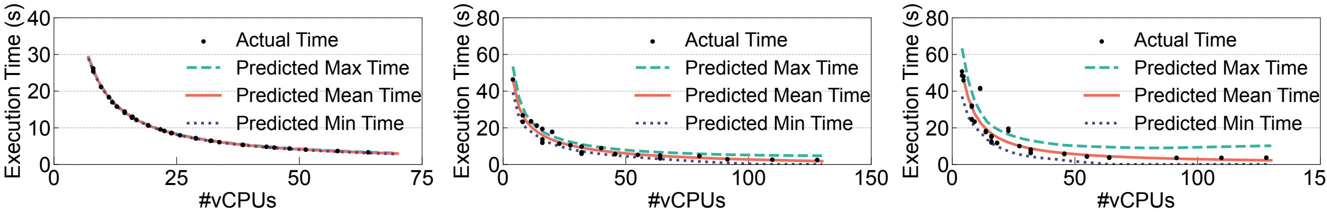
Cost guarantee. We also evaluate the cost guarantee of Jolteon by setting different cost bounds. Figure 11 shows the actual and ideal 95% cost for the three workflows. Similar to the latency guarantee, Jolteon is able to provide bounded cost for all workflows. When the cost bound is relatively loose (e.g., >$0.02 per run in Figure 11(b)), the actual cost is nearly unchanged. This is because the optimization objective is latency. Increasing resources does not further reduce latency. In such cases, Jolteon avoids unnecessary allocation of resources.

6.3 Effectiveness of the Performance Model

To evaluate the performance model, we conduct step-level, stage-level, and workflow-level experiments.

Step-level effectiveness. We first evaluate the performance model for the critical execution steps in the workflow. The critical steps have the longest execution time and the highest cost in the workflow. Figure 12 shows the actual execution time and the predicted time distributions against the total number of vCPUs. The total number of vCPUs is defined as the product of the number of function instances and the number of vCPUs allocated to each function for the step.

The three workflows exhibit different characteristics. TPC-DS Query is an I/O-intensive workflow, where the critical step is to read the data from S3. Since S3 offers a steady bandwidth for reading [7], the execution time of the step under the same vCPU allocation remains stable in Figure 12(a), and can be precisely predicted by the performance mode with less than 1% error. Video Analytics and ML Pipeline are compute-intensive, where the critical steps are computation steps in video splitting and model training stages, respectively. These steps have more variable execution time due to the performance variability of serverless computing, as shown in Figure 12(b) and Figure 12(c). The stochastic performance model predicts the execution time distributions that cover most of the actual execution time.



(a) TPC-DS Query, map1, read.

(b) Video Analytics, splitting, compute.

(c) ML Pipeline, training, compute.

Figure 12: Effectiveness of the performance model for critical execution steps.

Video Splitting Stage		
Resource Config: (d, v)	**P50**	**P95**
Large: (16, 4)	-6.07%	0.29%
Medium: (8, 2)	6.87%	5.47%
Small: (4, 1)	-5.80%	-4.50%
MAPE	**6.25%**	**3.42%**

Table 3: Errors of Jolteon for stage execution time prediction.

Resource Config	#Funcs (**d**)	#vCPUs/Func (**v**)	#vCPUs in Total
Config 0	(32, 16, 8, 8)	(2.5, 4, 2.5, 2.5)	184
Config 1	(32, 16, 16, 16)	(2, 1.5, 1.5, 2)	144
Config 2	(16, 32, 8, 16)	(2, 1.5, 1.5, 1.5)	116
Config 3	(16, 16, 16, 8)	(1, 1.5, 1.5, 1.5)	76
Config 4	(8, 16, 4, 4)	(1.5, 1, 1, 1)	36
Config 5	(4, 8, 4, 4)	(1.5, 1.5, 1.5, 1.5)	30

Table 4: Six resource configurations to run Video Analytics.

Stage-level effectiveness. Then we use large, medium, and small amounts of resources to examine the effectiveness of the stage-level performance model. Table 3 lists the prediction errors for the video splitting stage (i.e., the critical stage) in Video Analytics. d represents the number of function instances, and v represents the number of vCPUs per function. The error range is [-6.07%, 6.87%]. The mean absolute percentage error (MAPE) is 6.25% for P50 and 3.42% for P95 time prediction, indicating that Jolteon is accurate and is able to capture the performance variability for the stage.

Workflow-level effectiveness. Finally, we evaluate Jolteon's errors in predicting the end-to-end latency distributions for the entire workflow, and compare it with Orion's blackbox and Ditto's whitebox models. We vary the resource configurations with different number of function instances (i.e., **d**) and different number of vCPUs per function (i.e., **v**). The total number of vCPUs is the dot product of vector **d** and **v**. The detailed configurations are shown in Table 4. As shown in Table 5 for Video Analytics, Jolteon has an error range in [-7.59%, 9.29%] for P50 and P95 latency prediction, with a MAPE under 4%. The two baselines perform much worse. Orion experiences a MAPE exceeding 30% for both P50 and P95 latency predictions since its linear interpolation is inaccurate under large configuration space. Ditto has a MAPE of 33.98% for P50 and 28.00% for P95 latency predictions due to its inability to accommodate performance variability in its analytical model.

Video Analytics							
Resource	#vCPUs	Jolteon		Orion		Ditto	
Config	in Total	**P50**	**P95**	**P50**	**P95**	**P50**	**P95**
Config 0	184	0.88%	-7.58%	20.00%	14.62%	51.75%	40.91%
Config 1	144	3.23%	-1.21%	34.38%	36.03%	51.71%	40.89%
Config 2	116	9.29%	5.26%	32.86%	30.20%	75.71%	61.84%
Config 3	76	4.21%	2.56%	52.63%	57.52%	10.88%	8.03%
Config 4	36	3.46%	-1.54%	43.53%	44.28%	9.39%	16.31%
Config 5	30	-2.19%	-1.38%	12.23%	20.12%	-4.46%	-0.02%
MAPE		**3.88%**	**3.26%**	**32.62%**	**33.80%**	**33.98%**	**28.00%**

Table 5: Errors for end-to-end latency prediction of different models.

6.4 Time analysis for Jolteon

Performance model training time. We evaluate the offline time to train the performance model. For each workflow, Jolteon collects tens of execution profiles with different resource configurations. Then, it uses non-linear least squares to fit the distributions of the random variables (parameters) in the performance model. Table 6 reports the training time for the three workflows. The training time is less than 70 milliseconds, which is negligible compared to the end-to-end latency of tens of seconds.

Solving time. We measure the time of Jolteon's gradient descent algorithm in the convex optimizer. We vary the sample size from 10 to 10,000. Figure 13 shows the results. In the case of latency bound, the algorithm finishes under 0.5 seconds when sample size is less than 5,000. When a cost bound is specified, the algorithm finishes within 0.05 seconds. The higher solving time under a latency bound is due to the specifics of our implementation. With `SLSQP` as the algorithm, constraint functions are smooth and differentiable. The presence of the max operator in the latency constraint violates this condition. To address this, we replace the single latency constraint for the entire workflow with individual latency constraints for each DAG path. It eliminates the use of the max operator but increases the number of constraints.

More complex workflows result in longer solving time. Among the three workflows, TPC-DS Query has the most stages and complicated data dependencies (e.g., eight stages). Our experiment on TPC-DS Query involves 4048 sampling constraints. As Figure 13(a) shows, Jolteon can solve the problem within 0.5 seconds. According to the characterization on Azure Durable Functions [24], 95% of the serverless workflows have fewer than eight stages, which indicates that Jolteon provides sub-second solving time for most use cases.

	Training Time
TPC-DS Query	0.065 s
Video Analytics	0.016 s
ML Pipeline	0.014 s

Table 6: Training time for the performance model.

7 Discussion

Memory in serverless computing. Besides vCPU, the memory size of a serverless function also impacts its performance. In AWS Lambda, the memory size of a function is proportional to the configured vCPU. Therefore, the Jolteon prototype, which is on top of AWS Lambda, leverages this property and uses the vCPU as the main influence factor. Recent work [48] decouples the memory size from the vCPU in the resource configuration of a serverless function. In such cases, Jolteon chooses the memory size greater than the peak memory usage for the function and adjusts the number of vCPUs to meet different application requirements.

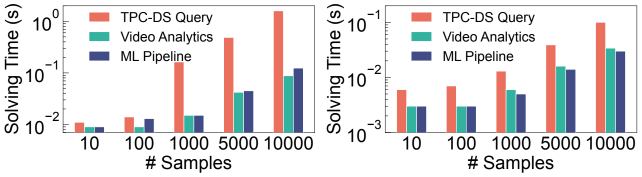
Auto-scaling. Resource provisioning includes two parts: resource configuration and resource scaling. Resource scaling (e.g., auto-scaling) is used to scale the workflow instances to meet the realtime execution load. It is orthogonal to resource configuration and is not the focus of Jolteon's design.

Limitations of Jolteon's performance model. One limitation is that the IO model does not reflect intricate IO patterns. For example, the serverless function may issue a SQL query to an external database. The IO time is determined by the SQL query's logic rather than the returned data size. Another limitation is that the performance model does not consider pipelining in serverless workflows. The upload of the upstream function and the download of the downstream function can be pipelined to reduce latency. Jolteon's performance model can be extended to capture these complex scenarios.

8 Related Work

Characteristics of serverless computing. Different from other cloud computing paradigms, serverless computing exhibits unique performance characteristics. Serverless-Wild [49] and Orion [24] analyze the performance of serverless functions in Azure Functions [2], and emphasize the performance variability and the impact of cold starts. FaaS-Cache [50] categorizes a serverless function's lifespan into initialization and execution phases. Caerus [8] and Ditto [9] view the function execution as fine-grained transmission and computation steps. Jolteon integrates the above insights and introduces a novel stochastic performance model to capture the characteristics of serverless computing.

Resource configuration for serverless computing. Resource configuration is critical to satisfy application-level requirements in cloud computing [20, 22, 51–53] and even more important in serverless computing due to its fine-grained resource allocation [7, 8]. Existing works either use whitebox models [7–9, 23, 26, 28] or blackbox models [24, 25, 27] to predict the performance of serverless functions and perform optimization. Jolteon leverages the advantages of both whitebox and blackbox models while mitigating their drawbacks in the performance model.



(a) Latency bound.

(b) Cost bound.

Figure 13: Solving time of Jolteon.

Serverless workflow orchestration. Many cloud providers identify the necessity for serverless workflow orchestration to facilitate the development of complex cloud applications. Centralized workflow orchestrators, such as AWS Step Functions [16], Google Workflows [19], and Azure Durable Functions and Logic Apps [17, 18], are adopted by major serverless platforms. Recent works [54–56] propose decentralized workflow orchestrators to improve scalability and reduce network latency. Jolteon can be integrated with them to provide automatic resource configuration.

Serverless workflow execution. Some works [39–41, 57] focus on the fault tolerance of serverless workflows to guarantee the exactly-once semantics. Jolteon can be integrated with them by capturing the overhead of logging in the performance model. Some works [9, 58, 59] exploit shared memory to reduce data transmission time by co-locating two adjacent stags in serverless workflows. Jolteon uses S3 as the external storage and its performance model is also applicable to data transmission through shared memory.

9 Conclusion

We present Jolteon, a serverless workflow orchestrator that facilitates automatic resource configuration to satisfy application-level requirements for serverless applications. Jolteon employs a novel stochastic performance model to capture both the execution characteristics and variability, a bound guaranteed sampler to transform the stochastic problem and an efficient convex optimizer to find Pareto-optimal configurations. We evaluate Jolteon with a variety of serverless workflows. The experimental results show that Jolteon outperforms the state-of-the-art solution by up to 2.3× on cost and 2.1× on latency.

Acknowledgments. We sincerely thank our shepherd Rodrigo Bruno and the anonymous reviewers for their valuable feedback on this paper. This work was supported by the National Key Research and Development Program of China under the grant number 2022YFB4500700. Xin Jin is the corresponding author. Zili Zhang, Chao Jin and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] "AWS Lambda." https://aws.amazon.com/lambda/.

[2] "Azure Functions." https://azure.microsoft.com/products/functions/.

[3] "Google Cloud Functions." https://cloud.google.com/functions/.

[4] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, 2019.

[5] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," *Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley*, 2019.

[6] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Communications of the ACM*, 2021.

[7] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *USENIX NSDI*, 2019.

[8] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: Nimble task scheduling for serverless analytics.," in *USENIX NSDI*, 2021.

[9] C. Jin, Z. Zhang, X. Xiang, S. Zou, G. Huang, X. Liu, and X. Jin, "Ditto: Efficient serverless analytics with elastic parallelism," in *ACM SIGCOMM*, 2023.

[10] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads," in *USENIX NSDI*, 2017.

[11] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, "Lavea: Latency-aware video analytics on edge computing platform," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, 2017.

[12] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *ACM Symposium on Cloud Computing*, 2018.

[13] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *ACM Symposium on Cloud Computing*, 2019.

[14] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *IEEE INFOCOM*, 2019.

[15] C. Zhang, M. Yu, W. Wang, and F. Yan, "Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving," in *USENIX ATC*, 2019.

[16] "AWS Step Functions." https://aws.amazon.com/step-functions/.

[17] "Azure Logic Apps." https://azure.microsoft.com/products/logic-apps/.

[18] "Azure Durable Functions." https://learn.microsoft.com/azure/azure-functions/durable/.

[19] "Google Workflows." https://cloud.google.com/workflows.

[20] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *USENIX NSDI*, 2017.

[21] 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 *EuroSys*, 2020.

[22] R. Bhardwaj, K. Kandasamy, A. Biswal, W. Guo, B. Hindman, J. Gonzalez, M. Jordan, and I. Stoica, "Cilantro: Performance-aware resource allocation for general objectives via online feedback," in *USENIX OSDI*, 2023.

[23] "AWS Lambda Power Tuning." https://github.com/alexcasalboni/aws-lambda-power-tuning.

[24] 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 *USENIX OSDI*, 2022.

[25] Z. Zhou, Y. Zhang, and C. Delimitrou, "Aquatope: Qos-and-uncertainty-aware resource management for multistage serverless workflows," in *ACM ASPLOS*, 2023.

[26] Z. Wen, Y. Wang, and F. Liu, "Stepconf: Slo-aware dynamic resource configuration for serverless function workflows," in *IEEE INFOCOM*, 2022.

[27] S. Eismann, J. Grohmann, E. Van Eyk, N. Herbst, and S. Kounev, "Predicting the costs of serverless workflows," in *Proceedings of the ACM/SPEC international conference on performance engineering*, 2020.

[28] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, “Cose: Configuring serverless functions using statistical learning,” in *IEEE INFOCOM*, 2020.

[29] “AWS Serverless Vector Engine.” https://aws.amazon.com/opensearch-service/serverless-vector-engine/.

[30] “Amazon simple storage service (S3).” https://aws.amazon.com/s3/.

[31] S. Ruder, “An overview of gradient descent optimization algorithms,” *arXiv preprint arXiv:1609.04747*, 2016.

[32] M. A. Lejeune and A. Ruszczyński, “An efficient trajectory method for probabilistic production-inventory-distribution problems,” *Operations Research*, 2007.

[33] M. C. Campi and S. Garatti, “A sampling-and-discarding approach to chance-constrained optimization: feasibility and optimality,” *Journal of optimization theory and applications*, 2011.

[34] A. K. Takyi and B. J. Lence, “Surface water quality management using a multiple-realization chance constraint method,” *Water Resources Research*, 1999.

[35] W. Hoeffding, “Probability inequalities for sums of bounded random variables,” *The collected works of Wassily Hoeffding*, 1994.

[36] J. Luedtke and S. Ahmed, “A sample approximation approach for optimization with probabilistic constraints,” *SIAM Journal on Optimization*, 2008.

[37] M. C. Campi and S. Garatti, “The exact feasibility of randomized solutions of uncertain convex programs,” *SIAM Journal on Optimization*, 2008.

[38] S. P. Boyd and L. Vandenberghe, *Convex optimization*. 2004.

[39] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, “Fault-tolerant and transactional stateful serverless workflows,” in *USENIX OSDI*, 2020.

[40] Z. Jia and E. Witchel, “Boki: Stateful serverless computing with shared logs,” in *ACM SOSP*, 2021.

[41] S. Zhuang, S. Wang, E. Liang, Y. Cheng, and I. Stoica, “Exoflow: A universal workflow system for exactly-once dags,” in *USENIX OSDI*, 2023.

[42] “Scipy.” https://scipy.org/.

[43] “LightGBM Python Library.” https://lightgbm.readthedocs.io/en/latest/Python-Intro.html.

[44] L. Deng, “The mnist database of handwritten digit images for machine learning research [best of the web],” *IEEE Signal Processing Magazine*, 2012.

[45] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, “Pocket: Elastic ephemeral storage for serverless analytics,” in *USENIX OSDI*, 2018.

[46] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[47] “TPC-DS.” https://www.tpc.org/tpcds/.

[48] M. Bilal, M. Canini, R. Fonseca, and R. Rodrigues, “With great freedom comes great opportunity: Rethinking resource allocation for serverless functions,” in *EuroSys*, 2023.

[49] 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 *USENIX ATC*, 2020.

[50] A. Fuerst and P. Sharma, “Faascache: keeping serverless computing alive with greedy-dual caching,” in *ACM ASPLOS*, 2021.

[51] S. Chen, C. Delimitrou, and J. F. Martínez, “Parties: Qos-aware resource partitioning for multiple interactive services,” in *ACM ASPLOS*, 2019.

[52] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, “Firm: An intelligent fine-grained resource management framework for {SLO-Oriented} microservices,” in *USENIX OSDI*, 2020.

[53] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, “Sinan: Ml-based and qos-aware resource management for cloud microservices,” in *ACM ASPLOS*, 2021.

[54] D. H. Liu, A. Levy, S. Noghabi, and S. Burckhardt, “Doing more with less: Orchestrating serverless applications without an orchestrator,” in *USENIX NSDI*, 2023.

[55] M. Yu, T. Cao, W. Wang, and R. Chen, “Following the data, not the function: Rethinking function orchestration in serverless computing,” in *USENIX NSDI*, 2023.

[56] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, “Faasflow: Enable efficient workflow execution for function-as-a-service,” in *ACM ASPLOS*, 2022.

[57] S. Qi, X. Liu, and X. Jin, “Halfmoon: Log-optimal fault-tolerant stateful serverless computing,” in *ACM SOSP*, 2023.

[58] S. Shillaker and P. Pietzuch, “Faasm: Lightweight isolation for efficient stateful serverless computing,” in *USENIX ATC*, 2020.

[59] S. Qi, L. Monis, Z. Zeng, I.-c. Wang, and K. Ramakrishnan, “Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing,” in *ACM SIGCOMM*, 2022.

A Appendix

A.1 Lower bound of sample size

Theorem A.1 *One of the lower bounds of the sample size, n, to guarantee the performance bound under confidence level* δ, $(0 < \delta < 1)$ *is:*

$$\frac{1}{2 \times (1 - percentile)^2} log(\frac{|D|}{1-\delta})$$

Proof. Let Pr be the probability function, i.e., the percentile function of a random variable, D be the definition domain and FD be the feasible domain of $(\mathbf{d},\mathbf{v})$. FD (i.e., the ideal feasible domain) is constrained by $Confidence\big(Pr\{G(\mathbf{d},\mathbf{v},\mathbf{x_i}) \ \le 0\} \ge percentile\big) \ge \ \delta$ with percentile as Pr function, while FD' (i.e., the sampling feasible domain) is constrained by n samples $\{G(\mathbf{d},\mathbf{v},\mathbf{x_i}) \ \le 0, \ \ for\ i = 1...n\}$.

For simplicity, *percentile* is replaced by p. With the percentile of performance metric (i.e., p), we have: $Pr\{G(\mathbf{d},\mathbf{v},\mathbf{x_i}) \ \le 0\} \ge p$ if $(\mathbf{d},\mathbf{v}) \in FD$ and $Pr\{G(\mathbf{d},\mathbf{v},\mathbf{x_i}) \ \le 0\} < p$ if $(\mathbf{d},\mathbf{v}) \notin FD$. We define the random variable Z_i by $Z_i = 1$ if $G(\mathbf{d},\mathbf{v},\mathbf{x_i}) \ \le 0$ and $Z_i = 0$ otherwise. Now, we consider the probability of $\exists(\mathbf{d},\mathbf{v}) \in FD'$ but $(\mathbf{d},\mathbf{v}) \notin FD$ (denoted by *pro*).

$$pro = Pr\{\sum_{i=1}^{n} Z_i \ge n\ and\ E(Z_i) < p\}$$
$$\le Pr\{\big(\sum_{i=1}^{n} Z_i - E(\sum_{i=1}^{n} Z_i)\big) \ge n - np\}$$

According to Hoeffding's inequality ($0 \le Z_i \le 1$), we have:

$$pro \le e^{-2n(1-p)^2}$$

Since there may be $|D|$ points in this domain, one of the upper bounds of $1-\delta$ (i.e., the probability of violating the confidence level) is:

$$1-\delta < 1-(1-pro)^{|D|} < |D| \times e^{-2n(1-p)^2}$$

Therefore, one of the lower bounds of sample size n is $\frac{1}{2\times(1-p)^2} log(\frac{|D|}{1-\delta})$.

A.2 Convexity Analysis

Lemma A.1 *Given two convex functions* $f(x)$ *and* $g(x)$, $h(x) = f(x) + g(x)$ *is also a convex function,* $x \in \mathcal{D}$.

Proof. Functions $f(x)$ and $g(x)$ are convex, which satisfy the following inequality:

$$f(\lambda x_1 + (1-\lambda)x_2) \le \lambda f(x_1) + (1-\lambda) f(x_2)$$
$$g(\lambda x_1 + (1-\lambda)x_2) \le \lambda g(x_1) + (1-\lambda) g(x_2)$$

Here, λ is an arbitrary value in $[0,1]$ and $x_1, x_2 \in \mathcal{D}$. We have:

$$h(\lambda x_1 + (1-\lambda)x_2) = f(\lambda x_1 + (1-\lambda)x_2) + g(\lambda x_1 + (1-\lambda)x_2)$$
$$\le \lambda f(x_1) + (1-\lambda) f(x_2) + \lambda g(x_1) + (1-\lambda) g(x_2)$$
$$= \lambda h(x_1) + (1-\lambda) h(x_2)$$

Therefore, $h(x)$ is a convex function.

Lemma A.2 *Given two convex functions* $f(x)$ *and* $g(x)$, $h(x) = \max(f(x), g(x))$ *is also a convex function,* $x \in \mathcal{D}$.

Proof. First, we prove a basic inequality for max operation:

$$\begin{aligned} &\max(\alpha_1,\alpha_2) + \max(\beta_1,\beta_2) \\ &= \max(\alpha_1 + \max(\beta_1,\beta_2), \alpha_2 + \max(\beta_1,\beta_2)) \\ &\ge \max(\alpha_1+\beta_1, \alpha_2+\beta_2) \end{aligned} \tag{8}$$

We can set $\alpha_1 = \lambda f(x_1)$, $\alpha_2 = \lambda g(x_1)$, $\beta_1 = (1-\lambda) f(x_2)$ and $\beta_2 = (1-\lambda) g(x_2)$, where $x_1, x_2 \in \mathcal{D}$. Based on Formula 8 and Lemma A.1, we have:

$$h(\lambda x_1 + (1-\lambda)x_2) = \max(f(\lambda x_1 + (1-\lambda)x_2), g(\lambda x_1 + (1-\lambda)x_2))$$
$$\le \max(\lambda f(x_1) + (1-\lambda) f(x_2), \lambda g(x_1) + (1-\lambda) g(x_2))$$
$$\le \max(\lambda f(x_1), \lambda g(x_1)) + \max((1-\lambda) f(x_2), (1-\lambda) g(x_2))$$
$$= \lambda h(x_1) + (1-\lambda) h(x_2)$$

Therefore, $h(x)$ is a convex function.

Lemma A.3 *Given two convex functions* $f(x)$ *and* $g(x)$ *(*$f(x)$ *is a monotonically increasing function),* $h(x) = f(g(x))$ *is also a convex function,* $x, g(x) \in \mathcal{D}$.

Proof. $\forall x_1, x_2 \in \mathcal{D}$ and $\lambda \in [0,1]$, we have $f(g(\lambda x_1 + (1-\lambda)x_2)) \le f(\lambda g(x_1) + (1-\lambda) g(x_2))$ since $f(x)$ is monotonically increasing and $g(x)$ is convex. Based on this, we derive:

$$h(\lambda x_1 + (1-\lambda)x_2) = f(g(\lambda x_1 + (1-\lambda)x_2))$$
$$\le f(\lambda g(x_1) + (1-\lambda) g(x_2))$$
$$\le \lambda f(g(x_1)) + (1-\lambda) f(g(x_2)) = \lambda h(x_1) + (1-\lambda) h(x_2)$$

Therefore, $h(x)$ is a convex function.

Lemma A.4 $(\frac{s}{xy})^{\alpha} \times \ln\frac{s}{xy}$ *is a bivariate convex function, where s is a positive constant,* $\frac{s}{xy} \in (1,\infty)$*,* $x, y \in (0,\infty)$ *and* α *is a nonnegative integer.*

Proof. We divide the proof into two cases.

Case one: $\alpha = 0$**.** Since $\ln\frac{s}{xy}$ has second derivatives when $\frac{s}{xy} \in (1,\infty)$. We calculate the Hessian matrix of the bivariate function:

$$Hessian(\ln\frac{s}{xy}) = \begin{bmatrix} \frac{1}{x^2} & 0 \\ 0 & \frac{1}{y^2} \end{bmatrix}$$

Evidently, such matrix is positive semi-definite and $\ln\frac{s}{xy}$ is a convex function.

Case two: $\alpha \ge 1$**.** We define $f(z) = z^{\alpha} \times \ln z$, and $g(x,y) = \frac{s}{xy}$. We first calculate the derivative of $f(z)$ $(z > 1)$: $f'(z) = (1 + \alpha \ln z) \times z^{\alpha-1} > 0$. Therefore, $f(z)$ is a monotonically increasing function. Then, we prove that $f(z)$ is a convex

Initial Config	#Funcs (**d**)	#vCPUs/Func (**v**)	#vCPUs in Total
Large	(16, 16, 16, 16)	(4, 4, 4, 4)	256
Medium	(8, 8, 8, 8)	(2, 2, 2, 2)	64
Small	(4, 4, 4, 4)	(1, 1, 1, 1)	16
Mix1	(4, 8, 16, 4)	(1, 2, 4, 1)	88
Mix2	(16, 4, 8, 16)	(4, 1, 2, 4)	148

Table 7: Detailed initial configurations.

function when $z > 1$. When $\alpha = 1$, we calculate the second derivative $f(z) = z^{\alpha} \times \ln z$ as follows:

$$f''(z) = (\ln z + 1)' = \frac{1}{z} > 0$$

When $\alpha \geq 2$, we calculate the second derivative as follows:

$$f''(z) = ((1 + \alpha \ln z) \times z^{\alpha-1})'$$
$$= \alpha(\alpha-1)z^{\alpha-2}\ln z + (2\alpha-1)z^{\alpha-2} > 0 \quad (z > 1, \alpha \geq 2)$$

Therefore, $f(z)$ is a convex function when $z > 1$. Last, we need to prove that $g(x,y)$ is a convex function. We calculate the Hessian matrix of $g(x,y)$ as follows (s,x,y are positive and s is constant):

$$Hessian(g(x,y) = \frac{s}{xy}) = \begin{bmatrix} \frac{2s}{x^3y} & \frac{s}{x^2y^2} \\ \frac{s}{x^2y^2} & \frac{2s}{xy^3} \end{bmatrix}$$

Since s,x,y are all positive, this matrix is positive semi-definite. Therefore, $g(x,y)$ is a convex function. In summary, $f(z)$ is a monotonically increasing convex function, and $g(x,y)$ is a convex function. Based on Lemma A.3, we derive that $h(x,y) = f(g(x,y))$ is a convex function.

Lemma A.5 $f(x,y) = \frac{s}{x \times min(y,w)}$ *is a bivariate convex function, where s,w are positive constants and x,y are positive independent variables.*

Proof. We are able to convert to $f(x,y)$ to $max(\frac{s}{x \times w}, \frac{s}{x \times y})$. The Hessian matrix of $\frac{s}{x \times w}$ is:

$$\begin{bmatrix} \frac{2s}{wx^3} & 0 \\ 0 & 0 \end{bmatrix}$$

It is a positive semi-definite matrix, and $\frac{s}{x \times w}$ is a convex function. As for $\frac{s}{x \times y}$, its convexity is proved in Lemma A.4. According to Lemma A.2, $f(x,y) = \frac{s}{x \times min(y,w)} = max(\frac{s}{x \times w}, \frac{s}{x \times y})$ is a convex function.

A.3 Sensitivity of Jolteon

To evaluate the sensitivity of Jolteon's convex optimizer algorithm (§4.3) to various initial configurations (i.e., entry points), we use five different initial configurations for Video Analytics: large, medium, small, mix1, and mix2, with total number of vCPUs ranging from 16 to 256. Table 7 shows them in detail, where **d** and **v** represent the vectors (four stages) of the number of function instances and the number of vCPUs per function instance, respectively. We run Jolteon with three different latency bounds: 12 seconds for tight bound, 36 seconds for moderate bound, and 80 seconds for loose bound.

Initial	Output Config		
Config	Tight Bound (12 s)	Moderate Bound (36 s)	Loose Bound (80 s)
Large	Config A	Config B	Config C
Medium	Config A	Config B	Config C
Small	Config A	Config B	Config D
Mix1	Config A	Config B	Config C
Mix2	Config A	Config B	Config C

Table 8: Output configurations of Jolteon for Video Analytics under different initial configurations and bounds.

Output Config	#Funcs (**d**)	#vCPUs/Func (**v**)	#vCPUs in Total
Config A	(32, 16, 16, 16)	(5, 1.5, 1.5, 5)	288
Config B	(8, 8, 8, 8)	(1.5, 1.5, 1.5, 1.5)	48
Config C	(4, 4, 4, 4)	(1, 1, 1.5, 1.5)	20
Config D	(4, 4, 4, 4)	(1, 1.5, 1.5, 1)	20

Table 9: Detailed output configurations of Jolteon.

As Table 8 indicates, Jolteon produces identical configurations across all initial configurations under the tight and moderate bounds. Under the loose bound, Jolteon generates a slightly different configuration for the small initial configuration. Table 9 further illustrates these output configurations. Notably, Config C and Config D under the loose bound share the same number of function instances and total vCPUs. The slight discrepancy between them lies in the specific allocation of vCPUs, i.e., (1, 1, 1.5, 1.5) vs. (1, 1.5, 1.5, 1). We run Video Analytics under these two configurations and obtain similar performance. Config C and Config D yield end-to-end latency of 75.9 and 74.5 seconds and cost of $0.00916 and $0.00874, respectively. The difference between the two configurations in latency and cost are negligible, which are 1.9% and 4.8%, respectively. In summary, Jolteon's convex optimizer algorithm is insensitive to initial configurations.