CharmSeeker: Automated Pipeline Configuration for Serverless Video Processing

Miao Zhang, Yifei Zhu, Jiangchuan Liu, *Fellow, IEEE,* Feng Wang, *Senior Member, IEEE,* and Fangxin Wang

***Abstract*—Video processing plays an essential role in a wide range of cloud-based applications. It typically involves multiple pipelined stages, which well fits the latest fine-grained serverless computing paradigm if properly configured to match the cost and delay constraints of video. Existing configuration tools, however, are primarily developed for traditional virtual machine clusters with general workloads. This paper presents CharmSeeker, an automated configuration tuning tool for serverless video processing pipelines. We first carefully examine the key steps and the performance bottlenecks for video processing over modern serverless platforms. Then, we identify the configuration space for processing pipelines and leverage a carefully designed Sequential Bayesian Optimization search scheme to identify promising configurations. We further address the practical challenges toward integrating our solution into real-world systems and develop a prototype with AWS Lambda. Evaluation results show that CharmSeeker can find out the optimal or near-optimal configurations that improve the relative processing time up to 408.77%. It is also more robust and scalable to various video processing pipelines compared with state-of-the-art solutions.**

***Index Terms*—Serverless computing, video processing, configuration tuning, Bayesian optimization**

I. Introduction

A STAGGERING number of videos are captured by ubiquitous cameras and consumed by numerous users and analytical algorithms. For instance, one billion hours of video are watched on YouTube daily [1], and video delivery on the Internet was predicted to account for 82% of IP traffic by 2022 [2]. Processing videos at scale for analysis or delivery purposes greatly challenges today's video processing systems [3]. Shipping them over the Internet to the resource-rich cloud for processing has become a natural way to handle the ever-growing volume of videos and increasingly resource-intensive video processing algorithms [4], [5], [6]. Achieving low-latency and low-cost video processing in the cloud is the key goal pursued by all video service providers. Yet, constrained by heavy-weight virtualization techniques and rigid pricing schemes [7], traditional virtual machine (VM) based cloud infrastructures can only achieve coarse-grained (e.g., node-level) parallelism. It not only prolongs the processing delay but also increases the monetary cost.

To simplify cloud programming and provide fine-grained services, cloud providers abstract cloud resources further and propose serverless computing. It enables developers to build and run their applications without thinking about servers [8]. As typical implementations of serverless computing, Function-as-a-Service (FaaS) offerings (e.g., AWS Lambda [9], Google Cloud Functions [10], Microsoft Azure Functions [11]) provide general-purpose cloud computing infrastructures and are popularizing the serverless paradigm [12]. In FaaS platforms, developers only need to upload application codes as a set of stateless *functions*; cloud providers are responsible for handling the underlying resource provisioning and management. The lightweight implementation and fine-grained pricing schemes make serverless computing promising for low-latency and cost-efficient video processing [13], [14], [15].

Video processing applications migrated to the serverless platform are decoupled into consecutive processing modules, forming a processing pipeline, with each module being implemented as a standalone serverless function. Given a video and a processing pipeline, users usually need to determine configurations for all modules in the pipeline first, i.e., determining the resources allocated to the corresponding serverless function of each module. The recommended industrial practice, a manually "trial and error" approach [16], actually leaves the configuration tuning task to users.

However, the configuration tuning task is exceptionally non-trivial. First, modules in video processing pipelines have diverse resource demands. For instance, based on our measurements, 1GB memory is sufficient for the license plate recognition module in a License Plate Query (LPQ) pipeline (as shown in Fig. 1), but it becomes the performance bottleneck of the object detection module in the same pipeline. Additionally, the extra gain in latency reduction diminishes as the resources allocated to the serverless functions increase [17]. Simply allocating the largest resources [14] inevitably leads to over-provisioning and is not cost-effective. Considering the number of videos that need to be processed in real-world applications, a small increase in the cost per unit can lead to a significant cost problem. While there is a broad range of existing efforts on optimizing configurations for VM clusters [18], [19], [20], [21], the configuration tuning for serverless video processing pipelines remains an unexplored area.

The unique challenges of configuration tuning for serverless video processing pipelines mainly arise from the following aspects. First, the huge configuration space introduced by the

This research was supported by an NSERC Discovery Grant, a CFI JELF/BCKDF Grant, and a MITACS Accelerate Grant. Yifei Zhu's work was supported in part by the SJTU Explore-X grant. Fangxin Wang's work was supported in part by National Natural Science Foundation of China with Grant No.62102342 (Corresponding author: Jiangchuan Liu).

Miao Zhang and Jiangchuan Liu are with the School of Computing Science, Simon Fraser University, BC, Canada.

Yifei Zhu is with UM-SJTU Joint Institute, Shanghai Jiao Tong University, Shanghai, China.

Feng Wang is with Department of Computer and Information Science, The University of Mississippi, Oxford, MS 38677 USA.

Fangxin Wang is with School of Science and Engineering (SSE) and The Future Network of Intelligence Institute (FNii), The Chinese University of Hong Kong, Shenzhen.

pipeline structure and the high parallelism of serverless computing make naive methods impractical (such as grid search, manually tuning [16]). Second, the higher abstraction level of serverless computing hides the underlying infrastructure information and increases the uncertainty, making previous solutions that rely on low-level system information [19], [20] no longer applicable. Third, evaluating a configuration along a pipeline can be expensive. Methods that rely on a large number of training samples to build up performance prediction models can quickly drive up the search cost [22]. Finally, cloud providers usually allow users to enforce a monetary budget on recurring workloads for cost management [23]. Due to the fine-grained pricing strategy of serverless computing, a slight variation in the pipeline budget can significantly change the number of feasible configurations. Moreover, the budget is generally set for the whole pipeline, making the configuration selections of different modules interact with each other.

To address these challenges, we design `CharmSeeker`[1], an automated configuration tuning tool of selecting good configurations for serverless video processing pipelines. Our contributions can be summarized as follows:

- By conducting a measurement study, we confirm the benefits of pipeline configuration tuning for serverless video processing and identify its challenges from both system and algorithm perspectives.
- We propose `CharmSeeker`, which harnesses the high parallelism of serverless computing and a carefully designed configuration picking algorithm to minimize the pipeline processing time within a monetary budget.
- Observing the large gap between the actual cost distribution and the budget allocation space, as well as the natural additive structure of our problem, we design a Sequential Bayesian Optimization (SBO) solution. It can identify good configurations for typical video processing pipelines via two successive optimization steps.
- We further address the practical challenges toward integrating our solution into real-world systems and implement a prototype with AWS Lambda.
- We introduce a percentile pipeline budget measure for pipeline budget settings and conduct extensive evaluation experiments. The results show that `CharmSeeker` can pick up near-optimal configurations with reduced search costs and is robust for various budgets and pipelines.

The rest of this paper is organized as follows. We briefly introduce serverless video processing and highlight the importance of configuration tuning through a measurement study in §II. Then, we present the overview of `CharmSeeker`, formulate the configuration tuning problem, and identify the challenges in §III. We discuss the configuration tuning solutions with BO in §IV and propose a novel SBO algorithm in §V. We then present the evaluation results in §VI. Further discussion is presented in §VII, and related works are reviewed in §VIII. Finally, we conclude this paper in §IX.

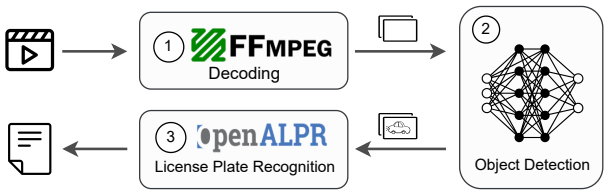


Fig. 1. License plate query (LPQ) pipeline.

II. Background and Motivation

A. Serverless Video Processing

The last decade has witnessed the commercial success of cloud computing represented by low-level VMs [7] (e.g., Amazon EC2 instances [24]). However, the operational complexity of building and maintaining VM clusters presents high barriers to entry for average cloud users [15]. To simplify cloud programming and make cloud resources easier to use, cloud providers propose serverless computing and implement this paradigm in their FaaS offerings. As the unit of computation in FaaS platforms, *functions* are code snippets typically written with a variety of high-level programming languages, such as `Python`, `Node.js`, and `Go` [12]. At deployment, developers register functions to the FaaS platform with minimal configuration efforts (e.g., specify memory) and declare events to trigger their executions. The FaaS platform is responsible for handling every triggering request, scaling resources precisely, and ensuring fault tolerance and service availability.

The FaaS platform handles each triggering event for a function by a short-lived, dedicated *function instance*, which executes the function code with the input message within a specialized container or sandbox [25], [26], [27]. Thanks to these lightweight virtualization technologies, function instances can spin up or down in several milliseconds [13], providing fine-grained, highly parallelizable computing infrastructures. Furthermore, users are charged for the time their function codes are executed in milliseconds, achieving the long-promised "pay-as-you-go". Benefiting from these advantages, serverless computing has been used to address an increasing variety of workloads [12], [27].

Low latency video processing can be achieved by leveraging the inherently parallelizable structures in videos (e.g., frames and groups of pictures) [3]. The fine-grained, highly parallel, readily available, pay-as-you-go computing infrastructures provided by serverless computing make it a perfect match for building low-latency and cost-effective video processing applications. Several efforts have been made to unlock the potential of serverless computing in video processing. For instance, `Excamera` [13] provides a backend for interactive video processing applications by invoking serverless function instances in bulk, thousands at a time. `Sprocket` [14] provides a framework to orchestrate serverless functions in video processing pipelines with a domain-specific language and exploits the intra-video parallelism (the group of pictures) to achieve low latency.

[1]We name the tool `CharmSeeker` as this challenging task is analogous to seeking the proper charm beads to create one's gorgeous charm bracelet.

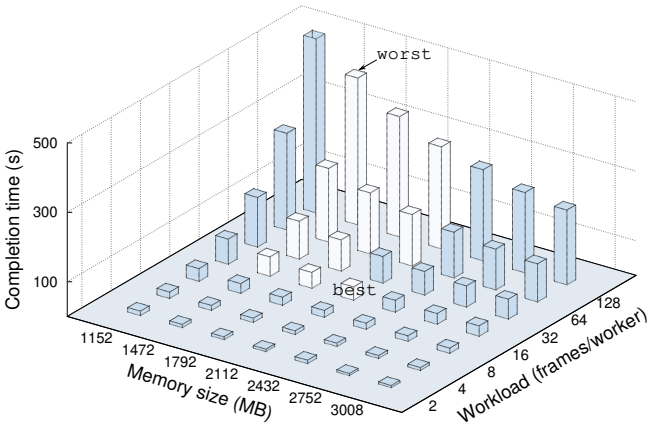


Fig. 2. Completion time of the *object detection* stage under different configurations. The white bars indicate feasible configurations under the budget of $0.151.

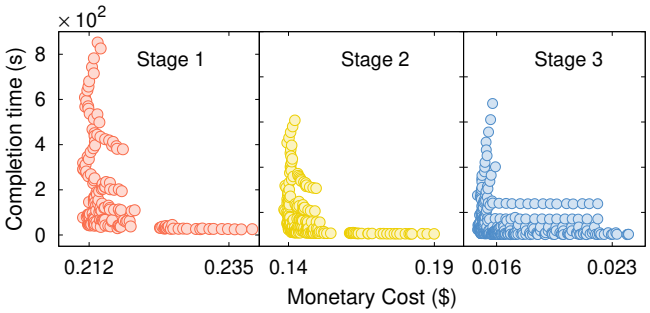


Fig. 3. Completion time and monetary cost of three stages in the LPQ pipeline under various configurations. Each circle represents a subconfiguration for the corresponding stage.

In this paper, we focus on the pipeline configuration tuning problem for these serverless-powered video processing applications and take the typical LPQ pipeline shown in Fig. 1 as a case study. This pipeline queries license plate numbers appearing in an input video. To deploy it with FaaS platforms, we break the monolithic code into a series of standalone *functions* with each implementing a processing module. Specifically, the input video is first decoded by the *decoding* module into a set of frames before being sent to the *object detection* module, which detects the existence of cars in each frame. Frames with cars are further processed by the following *license plate recognition* module, from which the actual license plate numbers in this video are extracted. We define the processing procedure of a module as a **stage**. The serverless function corresponding to each stage can be instantiated by multiple function instances, which we call **worker**s of this stage. Workers in the same stage execute the same function code to process different parts of a video in parallel so that the stage processing latency can be reduced.

B. Why Do We Need to Optimize Configuration?

Generally, given an input video (i.e., the total workloads for each stage is fixed), two types of knobs can be tuned for each worker j in stage i: the allocated resource and the assigned workload. In AWS Lambda, memory size is the only resource knob controlled by users. Other resources (such as CPU power) are allocated proportionally to it. Therefore, we only consider memory size as the resource knob in this work. Depending on the task type of the stage, the workload assigned to each worker can be video chunks, frames, images, etc. For ease of exposition, we assume that concurrent workers of the same stage i are allocated the same memory and workload, denoting by m^i and w^i, respectively. This also helps to mitigate stragglers in practice. Note that when the total workload for stage i (denoting by W^i) is determined, the number of concurrent workers (serverless function instances) is W^i/w^i. Therefore, the number of concurrent workers, namely concurrency, and w^i can be seen as the same knob. Further, we define a **configuration** $\boldsymbol{c}^i$ for stage i as a vector $[m^i\ w^i]$, and a **pipeline configuration** $\boldsymbol{c}$ for a k-stage pipeline as $[m^1\ w^1\ \ldots\ m^k\ w^k]$.

Finding the optimal configuration is critical to achieving low-latency and cost-effective video processing. To better understand the benefits of applying an optimized configuration, we take the second stage in the LPQ pipeline as a case study. In particular, we deploy an object detection function to AWS Lambda to detect objects included in 1950 frames derived from the Seattle video (see §VI for details). We record the completion time and monetary cost under different configurations. The results are shown in Fig. 2. It is easy to see that the completion time of the slowest configuration [1152 128] (i.e., 1152MB and 128 frames per worker) is 127.7× longer than that of the fastest configuration [3008 2], while the processing cost of [3008 2] is 36.43% more expensive than the cheapest configuration [1472 64]. Assuming a practical budget of $0.151 is set by the user for this processing task via the cost management API, like AWS Budgets [23], we further identify the feasible configurations within this budget as white bars shown in the figure. Under this budget, the worst feasible configuration is 11.07× slower than the best feasible one.

The performance gap between good and bad configurations can be even more drastic when it comes to the whole processing pipeline. Fig. 3 shows the cost of various subconfigurations for the LPQ pipeline to process 390 video chunks derived from the Seattle video. With a given pipeline budget $0.384, if a bad configuration [1280 32 1152 128 512 1024] (the uppermost circle in each subfigure) was chosen, the processing time of the whole pipeline can be 40.5× longer than that of a proper configuration [2432 2 2048 4 1600 8] (located lower left in each subfigure). Therefore, significant benefits can be obtained by picking up a proper configuration.

III. CharmSeeker: Design and Challenges

A. Design Overview

Fig. 4 demonstrates the overview of `CharmSeeker`. On the cloud-side, video processing pipelines are deployed to the serverless computing platform, and a remote-shared storage system (such as Amazon S3) is used to handle intermediate data exchange between consecutive pipeline stages. On the client-side, the Profiling module, the Configuration Picking

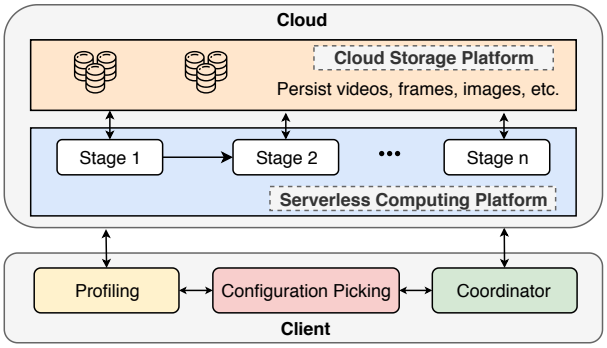


Fig. 4. Overview of `CharmSeeker`.

module, and the Coordinator module are in charge of picking configurations and invoking serverless functions.

Specifically, by running our configuration tuning algorithm, the Configuration Picking module can propose promising configurations for the Profiling module, which can invoke serverless profiling functions through Remote Procedure Call (RPC). The profiling functions execute the video processing tasks with the configuration received from the Profiling module and report the results (processing time and monetary cost) back to it. The results are further sent back to the Configuration Picking module for the next configuration selection. When the Configuration Picking module is convinced that it has found a near-optimal configuration, it will report the configuration to the Coordinator module, which will run regular video processing tasks with the reported configuration.

Recently, several frameworks have been proposed to orchestrate the execution of serverless functions in a video processing pipeline, like `mu` [13] and `Sprocket` [14]. They focus on the function orchestration in a pipeline but lack discussions on how to configure memory and workloads to optimize performance and costs. As such, they can take the *Coordinator* role in `CharmSeeker`. The main responsibility of the Profiling module is to invoke serverless functions and receive execution results, which can be easily implemented based on APIs provided by serverless computing platforms. As a result, we will focus on the design of the Configuration Picking module in the remainder of this paper.

B. Configuration Tuning: Problem Formulation

For a given video v, a video processing pipeline p and budget $\mathcal{B}$, the goal of the Configuration Picking module is to minimize the pipeline processing time T under the budget constraint by tuning pipeline configuration knobs. The pipeline processing time with a configuration $\boldsymbol{c}$ is defined as $T(\boldsymbol{c})$ and processing monetary cost is denoted as $B(\boldsymbol{c})$. Then our problem can be formulated as follows:

$$\begin{aligned} \underset{\boldsymbol{c}}{\textit{minimize}} \quad & T(\boldsymbol{c}) \\ \textit{subject to} \quad & B(\boldsymbol{c}) \leq \mathcal{B} \end{aligned} \tag{1}$$

The pipeline processing time is the sum of completion time of all stages in the pipeline, i.e., $T(\boldsymbol{c}) = \sum_i T^i(\boldsymbol{c}^i)$. Here we ignore the scheduling and invoking overheads across different pipeline stages because the choices of configuration do not affect this overhead, which mainly depends on the implementations of cloud platforms. In practice, the execution duration of workers in the same stage may not be completely identical due to the complex execution environment and noises. The tail latency determines the completion time of one stage, and thus, we define $T^i(\boldsymbol{c}^i)$ as the execution duration of the slowest worker in stage i, i.e., $\max_j T^i_j(\boldsymbol{c}^i)$, where for any worker j in stage i, $T^i_j(\boldsymbol{c}^i)$ is reported by the serverless platform and can be easily accessed. The pipeline's monetary cost $B(\boldsymbol{c})$ is the aggregated cost for all workers from all stages in the pipeline, i.e., $\sum_{i,j} B^i_j(\boldsymbol{c}^i)$. The typical serverless platform charges users by the allocated resource (memory) and the execution duration of function instances. According to the pricing strategy, we can easily obtain the monetary cost $B^i_j(\boldsymbol{c}^i)$ of each worker j by calculating:

$$B^i_j(\boldsymbol{c}^i) = r \times m^i \times \lceil T^i_j(\boldsymbol{c}^i) \rceil + I \tag{2}$$

where r is the price for every MB-second, I is the price for each invocation, and $\lceil T^i_j(\boldsymbol{c}^i) \rceil$ is the execution duration is seconds after rounding up to the nearest billing unit.

C. Configuration Tuning: Challenges

C1: Large configuration space. The memory allocated for each serverless function can be in 1MB increments on AWS Lambda, and this platform currently supports 1000 concurrent function instances by default, and this limit can be increased by request [28]. This leads to a huge configuration space for a processing stage, not to mention the whole processing pipeline. For example, the number of all pipeline configurations in configuration space for the LPQ pipeline we evaluated in §VI is as many as $11,289,600$. Therefore, the search cost of a brute-force method is unacceptable.

C2: Complicated processing time model: The execution duration of a serverless function instance is affected by the allocated resources, the input workload, and the internal code logic of functions. The uncertain network conditions further introduce noises when functions communicate with other cloud services. Capturing such complicated dependencies using white-box methods from the perspective of cloud users is impractical. This is because only minimal information of actually allocated resources is exposed to users (e.g., memory in AWS Lambda) in serverless platforms, and there is limited access to low-level system information. Experimentally, a measurement study on real-world serverless platforms [17] has revealed that the relationship between the execution duration of a function instance and its memory size is *non-linear*. Overprovisioning resources beyond the function's requirement only gains marginal improvements.

Fig. 5 shows the completion time and monetary cost distributions of all configurations for *the third stage* in the LPQ pipeline. As demonstrated, the effects of the allocated memory and the input workload on the completion time are correlated. When the per-worker workload is small, completion time cannot benefit from large memory sizes; in comparison, when the per-worker workload becomes large, allocating more memory can lead to considerable benefits.

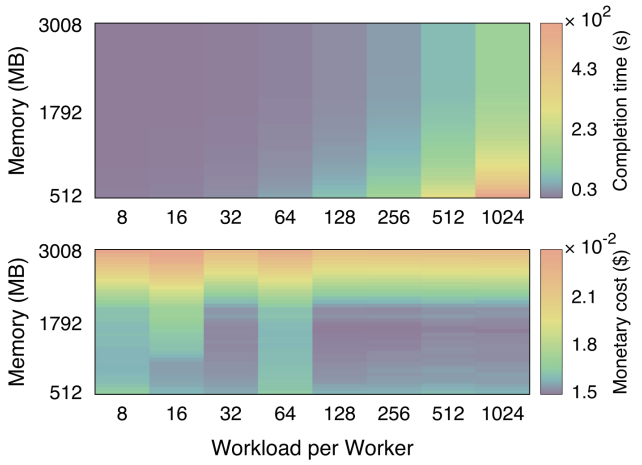


Fig. 5. Completion time and monetary cost distributions of various configurations. (License plate recognition stage of the LPQ pipeline; Video source: Seattle video)

C3: Complicated monetary cost model: Compared with the completion time, the monetary cost of a function instance has more complex relations to the allocated memory. On one hand, large memory may lead to a short completion time, which helps save money. On the other hand, large memory also means a higher price per unit time, which drives up the monetary cost. Thus, it is hard to summarize a general rule of how much memory will result in low monetary costs. The situation becomes much trickier with the introduction of the per-worker workload knob. As shown in Fig. 5, there are multiple local minima in the configuration-money space, which makes monetary cost optimization challenging.

C4: Budget-constrained pipeline: As shown in Fig. 3, configurations of different stages with the same completion time can cost differently. This indicates that different video processing modules in a pipeline have different resource requirements. For example, 1GB of memory is sufficient for the license plate recognition function. However, it will be the resource leading to performance bottlenecks for the object detection function that runs a deep learning model. Consequently, we cannot simply choose the same configuration for all stages in a pipeline. Furthermore, the optimal configuration for each stage cannot be separately found because they are coupled to each other by the pipeline budget. Although allocating more budget to one stage improves its performance, it also deprives other stages of the opportunity to explore better configurations.

IV. Configuration Tuning with BO

A. Why Do We Choose BO?

A straightforward way of configuration tuning is to run all possible configurations and find out the best one. However, the challenge C1 we identified in §III-C makes brute-force methods (e.g., exhaustive search) not practical given the prohibitively high search cost. Approaches that assume the dependence between performance and configurations can be characterized by simple analytical models are also impractical due to the challenge C2 and C3. These challenges further preclude methods that build complex machine learning models to predict performance under various resource configurations [20], [22], since they require access to low-level system information and many training samples to guarantee high accuracy. Solutions that employ simple sampling strategies along with heuristic searching algorithms [18] are not ideal for solving the challenges of our problem as well. This is because our performance functions are non-convex and have multiple local extrema. Thus, heuristic searching algorithms (such as those based on gradient-descent) can easily fall into local extrema.

The challenges of our problem make it easier to experiment with than to understand, so it is better to consider our problem as a black box [29]. Bayesian Optimization (BO) is a powerful black-box optimization tool [30], [31], [32] that can overcome the disadvantages of aforementioned solutions, and it has been applied to optimize VM cluster configuration in recent years [19], [21]. By adopting BO, `CharmSeeker` can benefit from the following aspects. First, BO is designed to find the extrema of black-box functions that are expensive to evaluate [30] and works well even in settings where the functions are non-convex and have multiple local extrema [32]. Second, non-parametric BO does not make any assumptions about the parameters of the black-box objective, which makes it robust enough to deal with various video types, processing modules, and pipeline topologies. Furthermore, BO can get near-optimal configurations with a high probability by only evaluating a small *carefully selected* subset of configurations, leading to a low-latency and economical configuration selection. Finally, BO's capability in integrating out noises can help `CharmSeeker` work well with real-world platforms.

B. Vanilla BO: Preliminary

Formally, the goal of vanilla BO is to solve the global optimization problem of finding $x_\star = \text{argmin}_{x\in\mathcal{X}} f(x)$, where f is an expensive black-box function. Normally, we do not have access to $f(x)$, but only a noisy version: $y = f(x) + \varepsilon$ with $\varepsilon \sim N(0, \sigma^2)$. Fundamentally, BO is a sequential model-based optimization solution where promising candidates are evaluated one by one [31]. A typical process of BO starts with initializing a *probabilistic surrogate model* M with a small set of samples $\mathcal{D}_0$. The probabilistic surrogate model represents the prior we place on all possible objective functions in the function space. BO assumes that the observations of the unknown objective function are sampled from it. Then during each iteration t, based on the surrogate model, an *acquisition function* α is optimized to select the next point x_t for evaluation. Specifically, the acquisition function identifies promising points based on the distributions at all not-yet-evaluated points predicted by the surrogate model. After observing the noisy output y_t of the objective, the surrogate model is updated by the newly generated sample (x_t, y_t) under the Bayes' rule. This process proceeds until the optimum is found.

The success of BO can be attributed to its two key ingredients: the probabilistic surrogate model and the acquisition function. The former maintains our belief about the objective function. It can be implemented as parametric and non-parametric models [31]. In this paper, we choose the non-parametric Gaussian Process (GP) [33] due to its flexibility

and tractability [34]. BO with the GP prior relies on GP regression to predict the mean and variance of the unknown function at any point in the input space. Based on the surrogate model posterior, the acquisition function determines which point to evaluate next. The principle is automatically trading off between exploration and exploitation. Exploration means selecting from where the surrogate model has high uncertainty, and exploitation indicates sampling from locations where the predicted value of the objective function is small. Good implementations, such as Expected Improvement (EI) and GP Upper Confidence Bound [30], take both exploration and exploitation into consideration. We choose EI in this work as it performs well without additional parameters [34].

The basic idea of EI is to pick up the point with the max expected improvement over the current best observation. Assuming we have obtained n observation samples $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$ and the best observation so far is $y_t = \min\{y_i\}_{i=1}^{n}$. The EI of x can be calculated by:

$$\mathrm{EI}(x) = \mathbb{E}(\max\{y_t - f(x), 0\} \,|\mathcal{D}) \tag{3}$$

With GP prior, the EI criterion has the following closed-form expression:

$$\mathrm{EI}(x) = \begin{cases} (y_t - m(x))\Phi(Z) + \sigma(x)\phi(Z), & \text{if } \sigma(x) > 0 \\ 0, & \text{if } \sigma(x) = 0 \end{cases}$$

where $Z = \frac{y_t - m(x)}{\sigma(x)}$. $m(x)$ and $\sigma^2(x)$ are the predicted mean and variance of function f at x, respectively. $\Phi(\cdot)$ is the standard normal Cumulative Distribution Function (CDF) and $\phi(\cdot)$ is the Probability Density Function (PDF). Note that compared with the expensive objective function, the acquisition function is much cheaper to compute and gradient-based optimization methods are applicable to it.

V. Configuration Tuning in CharmSeeker

A. BO-based Strawman Solutions

1) Applying BO to Our Problem: For unconstrained BO or when the constraint value can be calculated from the objective value [21], only one surrogate model (for the objective function) is needed. Unfortunately, for our constrained problem, we cannot calculate the pipeline monetary cost (i.e., the summation of all workers' monetary costs in the pipeline) directly from the pipeline processing time (i.e., the summation of the slowest worker's processing time for each stage). Thus, we need to build independent surrogate models for the processing time and monetary cost as recommended in [35]. Note that these two performance metrics can still be observed together when evaluating configuration $\boldsymbol{c}$ since the FaaS platform reports the execution duration of each worker, i.e., $T_j^i(\boldsymbol{c}^i)$, from which both performance metrics can be easily calculated. In addition, the processing time and monetary cost are both non-negative for all configurations and not well-modeled by a GP prior. We thus model them in the logarithmic units by placing the GP prior to $\log T(\boldsymbol{c})$ and $\log B(\boldsymbol{c})$. Consequently, the solution to the problem (1) can be found by solving the following equivalent problem:

$$\begin{aligned} \underset{\boldsymbol{c}}{\textit{minimize}} \quad & \log T(\boldsymbol{c}) \\ \textit{subject to} \quad & \log B(\boldsymbol{c}) \le \log \mathcal{B} \end{aligned} \tag{4}$$

To encourage the acquisition function to select promising configurations within the feasible region, we introduce the probability of constraint satisfaction $P(\log B(\boldsymbol{c}) \le \log \mathcal{B})$ and the constrained-weighted EI (CEI) [35]:

$$\mathrm{CEI}(\boldsymbol{c}) = P(\log B(\boldsymbol{c}) \le \log \mathcal{B}) \times \mathrm{EI}(\boldsymbol{c}) \tag{5}$$

The probability of constraint satisfaction for the not-yet-evaluated configuration $\boldsymbol{c}$ can be easily calculated from the marginal mean and variance predicted by the surrogate model for the constraint function.

Although serverless function instances are typically implemented by lightweight virtualization technology with strong isolation [27], the shared underlying resources in the public cloud unavoidably lead to some uncertainty. For example, different executions of the same function may take different times. Following best practices in the previous work [21], we leverage the ability of BO to handle additive noises to address the cloud uncertainty. To be specific, the cloud uncertainty will be modeled as the observation noises in BO. The evaluation results $\log \widetilde{T}(\boldsymbol{c})$ and $\log \widetilde{B}(\boldsymbol{c})$ for configuration $\boldsymbol{c}$ will be considered as the noisy observations of $\log T(\boldsymbol{c})$ and $\log B(\boldsymbol{c})$.

2) Strawman Solutions and Their Drawbacks: Based on the optimization framework provided by vanilla BO, if we take the entire pipeline as a whole, model the pipeline processing time and monetary cost as independently GPs, and adopt CEI as the acquisition function to pick up the next pipeline configuration, we can obtain a strawman solution to our optimization problem. We call this solution CherryPick* since it is a specialized version of CherryPick [21] for our problem. Unfortunately, despite its simplicity, this solution can suffer from performance degradation as the vast pipeline configuration space (search space) increases the difficulties in converging to the best feasible configuration.

Exploiting the unique structure in the objective function to accelerate the convergence of BO has proven a promising approach [36], [37]. Specifically, if the objective function has an additive structure, decomposing it into a set of low-dimensional disjoint subproblems can bring significant statistical and computational benefits [37]. For our pipeline optimization problem, the objective function is naturally additive. For instance, the optimized objective of the LPQ pipeline can be decomposed as follows:

$$T(\boldsymbol{c}) = T^1(\boldsymbol{c}^1) + T^2(\boldsymbol{c}^2) + T^3(\boldsymbol{c}^3) \tag{6}$$

where $T^i(\cdot)$ has its own input parameters $\boldsymbol{c}^i$ and can be optimized and evaluated independently and concurrently. These facts inspire us to borrow ideas from the *divide-and-conquer* algorithm design paradigm, i.e., dividing the problem into independent subproblems that can be solved directly and then merging the results of subproblems.

Considering that the pipeline budget only affects the upper bound of each stage's monetary cost, we can break the pipeline optimization problem into independent subproblems if we know how to allocate the pipeline budget to each stage. Assuming there are k stages in a video processing pipeline. Another strawman solution is to randomly generate a budget allocation vector $\boldsymbol{b} = [b^1\ b^2\ \cdots\ b^k]$ satisfying $0 < b^i < \mathcal{B}$

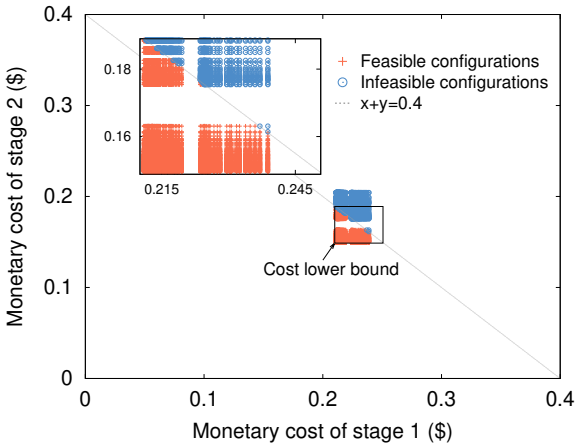


Fig. 6. Comparison of the potential budget allocation space and the actual monetary costs distribution for the first two stages of the LPQ pipeline with the budget of $0.4.

and $\sum_i b^i \leq \mathcal{B}$. With such a budget allocation vector, our pipeline optimization problem can be addressed by solving the following budget-constrained optimization subproblem for stage i:

$$\begin{aligned} \underset{\boldsymbol{c}^i}{\textit{minimize}} \quad & \log T^i(\boldsymbol{c}^i) \\ \textit{subject to} \quad & \log B^i(\boldsymbol{c}^i) \leq \log b^i \end{aligned} \tag{7}$$

The above subproblem for stage i can be solved directly by a constrained BO, i.e., CherryPick*. Benefiting from the reduced parameter dimensions (only m^i and w^i), it can quickly converge to find the near-optimal subconfiguration that satisfies the sub-budget constraint b^i.

However, it is exceptionally challenging to allocate the pipeline budget properly across all stages by generating the budget allocation vector at random. Assuming our pipeline budget is $\mathcal{B}$, and the allowed budget range for each stage is $(0, \mathcal{B})$ when there is no additional information. This demands us to pick up a budget allocation vector from $(0, \mathcal{B})^k$, which is a huge space since the budget can be any continuous numeric numbers in this range. Thus, the strawman solution is highly likely to find an inappropriate budget allocation vector, leading to infeasible solutions and wasting the scarce search cost. Here we take the first two stages of the LPQ pipeline as an example to elaborate on this issue. Assuming that the 2-stage pipeline budget for processing the Seattle video is $0.4, without extra information, the sub-budget allocated to each stage can be any value in the range $(0, 0.4)$. Despite this, the actual monetary costs of all configurations only take up a small part of the $(0, 0.4)^2$ space, as shown in Fig. 6. This indicates the high difficulties in obtaining a proper budget allocation vector without the knowledge of the cost distribution.

B. Sequential Bayesian Optimization

Motivated by the observations, we propose an SBO solution with two successive optimization steps to overcome the drawbacks of the strawman solutions. To break the pipeline optimization problem into independent subproblems, SBO first runs a *cost optimization* step. Specifically, it leverages the vanilla BO to unearth the subconfiguration leading to the minimal cost for each stage. Following the previous example in Fig. 6, if the minimal costs we get is $[b_1\ b_2]$, the search space can be significantly reduced to $[b_1,\ 0.4-b_2]\times[b_2,\ 0.4-b_1]$ (the small rectangle shown in Fig. 6). Then, SBO employs Latin Hypercube Sampling (LHS) [38] to sample a set of uniformly distributed budget allocation vectors from the reduced search space. As such, our problem is converted to a set of independent subproblems, and each subproblem is to optimize the pipeline processing time under a fixed budget allocation vector. Next, SBO enters the *time optimization* step. With each sampled budget allocation vector, SBO decouples all pipeline stages and directly applies a constrained BO to solve each stage's optimization subproblem (7). Finally, the solutions to all subproblems are merged to obtain the optimal pipeline configuration.

Algorithm 1 shows how SBO works in detail. For each stage i, we maintain one surrogate model M_1^i for the monetary cost function and the other M_2^i for the processing time function. In the *cost optimization* step, the algorithm tries to find the cost lower bound for each stage (lines 1-11). This is equivalent to solve the following optimization problem for each stage i:

$$\underset{\boldsymbol{c}^i}{\textit{minimize}} \quad \log B^i(\boldsymbol{c}^i) \tag{8}$$

The acquisition function α_1^i is based on the surrogate model M_1^i to pick up the next configuration $\boldsymbol{c}_\star^i$ for evaluation. The noisy observations $\log\widetilde{T^i}(\boldsymbol{c}_\star^i)$ and $\log\widetilde{B^i}(\boldsymbol{c}_\star^i)$ are then added to the known sample set for the surrogate model update. We use EI to implement α_1^i since problem (8) is an unconstrained optimization. This step stops when the sum of the second smallest monetary cost of each stage is not greater than the pipeline budget $\mathcal{B}$. This stopping condition ensures that the next time optimization step has at least two initialization samples. Note that this step does not guarantee to find the lowest cost for each stage, but an approximation is sufficient.

Based on the lowest cost $\boldsymbol{b}_\star$ observed in the first step, we define the left budget $\gamma = \mathcal{B} - \sum_i b_\star^i$. Immediately, the reduced budget search space can be denoted by:

$$\mathcal{X}_\mathcal{B} = [\ b_\star^1,\ b_\star^1+\gamma\ ] \times \cdots \times [\ b_\star^k,\ b_\star^k+\gamma\ ] \tag{9}$$

We then use the LHS to obtain budget allocation vectors from the reduced space $\mathcal{X}_\mathcal{B}$, since this sampling method can generate near-random samples. The number of sampled vectors is determined by the allowed search cost (lines 13-14). For each sampled budget allocation vector $\boldsymbol{b} = [b^1 \cdots b^k]$, we set $\log b^i$ as the monetary cost upper bound for stage i and then conduct m evaluations to solve the problem (7) (lines 15-25). The acquisition function α_2^i implements CEI because it biases the search towards the feasible region. Each stage optimizes its objective function independently and concurrently. All stages are loosely coupled to each other through the constrained cost upper bound. Next, we construct all evaluated pipeline configurations by computing the Cartesian Product of the sets of evaluated configurations for each stage. The algorithm finally outputs the feasible configuration with the minimum pipeline processing time (lines 26-27).

Algorithm 1: Sequential Bayesian Optimization

Input: Number of iterations m; Surrogate model M_1^i, M_2^i; Acquisition function α_1^i, α_2^i; Known sample set $\mathcal{D}^i$
Output: c_{best}

Initialize M_1^i with $\mathcal{D}^i$;
repeat
  for $i = 1, 2, \cdots, k$ **do**
    $c_\star^i \leftarrow \text{argmax}_{c^i}\, \alpha_1^i(c^i; M_1^i)$, and evaluate $c_\star^i$;
    $\mathcal{D}^i \leftarrow \mathcal{D}^i \cup \{(c_\star^i, \log \widetilde{T}^i(c_\star^i), \log \widetilde{B}^i(c_\star^i))\}$;
    Update M_1^i with $\mathcal{D}^i$;
    $b_\star^i \leftarrow$ the smallest cost B^i from $\mathcal{D}^i$;
    $b_2^i \leftarrow$ the second smallest cost B^i from $\mathcal{D}^i$;
  end
until $\sum_i b_2^i \le \mathcal{B}$;
$\boldsymbol{b}_\star \leftarrow [b_\star^1, \cdots, b_\star^k]$;
Compute reduced space $\mathcal{X}_\mathcal{B}$ based on $\boldsymbol{b}_\star$;
Compute the left number of evaluations n ;
$S \leftarrow$ Sample n/m budget allocation vectors from $\mathcal{X}_\mathcal{B}$;
Initialize M_2^i with $\mathcal{D}^i$;
foreach *budget allocation vector* $\boldsymbol{b} \in S$ **do**
  for $i = 1, 2, \cdots, k$ **do**
    while *Number of iterations* $\le m$ **do**
      $c_\star^i \leftarrow \text{argmax}_{c^i}\, \alpha_2^i(c^i; \log b^i, M_1^i, M_2^i)$;
      Evaluate $c_\star^i$;
      $\mathcal{D}^i \leftarrow \mathcal{D}^i \cup \{(c_\star^i, \log \widetilde{T}^i(c_\star^i), \log \widetilde{B}^i(c_\star^i))\}$;
      Update M_1^i, M_2^i with $\mathcal{D}^i$;
    end
  end
end
$C \leftarrow \{(c^1, \cdots, c^k) \mid c^i \in \mathcal{D}^i \wedge \sum_i B^i(c^i) \le \mathcal{B})\}$;
$c_{best} \leftarrow \text{argmin}_{c \in C}\, T(c)$;

VI. Evaluation

We evaluate `CharmSeeker` with video processing pipelines deployed on a leading public cloud platform AWS. We choose AWS since AWS Lambda is a good representative of serverless computing platform [26]. At its core, `CharmSeeker` is designed to be *independent* of serverless computing platforms. It only needs access to platform-specific APIs for changing function configurations, invoking functions, and reading function execution logs. These are basic APIs that serverless computing platforms expose for their users [39], [40]. Thus, users can extend `CharmSeeker` to another serverless computing platform with minimal efforts.

A. Experiment Setup

Video dataset: We evaluate `CharmSeeker` using a typical set of videos as shown in TABLE I, which covers different camera types (static or moving), illuminations (daytime or night), and resolutions (720p or 4K). Before feeding a video to the processing pipeline, we first chop it into equal-length (about 5 seconds) chunks as the common practice in industry [41] and store them in Amazon S3. Specifically, 390 chunks for each video are evaluated.

TABLE I
The Video Dataset

Name	Description
Seattle	Daytime drive through Seattle, dashcam, 4K [42]
L.A.	Night drive through Los Angeles, dashcam, 4K [43]
Road	A motorway traffic video, static camera, 720p [44]

Serverless video processing pipeline: We implement each module in the LPQ pipeline as an independent serverless function with `Python` and deploy them to AWS Lambda. The decoding function downloads video chunks from Amazon S3 and decodes them using `FFmpeg` [45]. It then selects 5 frames from each video chunk and uploads them to S3 for further processing. The workload in this stage refers to the number of video chunks processed by each worker. The following object detection function leverages a deep neural network (DNN) model, `YOLOv3` [46], to detect objects from frames downloaded from S3, extract the images with recognized cars and upload them back to S3. The number of frames is the workload in this stage. The last license plate recognition function automatically detects and recognizes license plates using the `OpenALPR` [47] library. The number of car images processed by each worker is the workload in this stage. From a perspective of resource management, the LPQ pipeline is very representative, as it includes typical methods to implement serverless video processing functions, binary executables (`FFmpeg`), DNN-based algorithms (`YOLOv3`), and traditional computer vision algorithms (`OpenALPR`).

Alternative solutions: Since there are no solutions directly solving our budget-constrained pipeline configuration optimization problem, we compare our solution with the specialized versions of state-of-the-art approaches. (1) *Randomized Grid Search (RGS)*: Random Search has been proved to be more efficient than Grid Search for hyperparameter optimization when different hyperparameters show different importances [48]. Generating a random configuration for the whole pipeline is inefficient, as the number of possible pipeline configurations is exponential in the number of knob dimensions. Considering that each stage can be evaluated independently, we propose a randomized grid search solution to strengthen naive random search. This method randomly samples n configurations for each stage to evaluate. The best feasible configuration is generated from the Cartesian Product of all the subconfigurations sampled for each stage. (2) *CherryPick* (CPS)*: This solution applies the constrained BO (i.e., methods used in CherryPick [21]) to the entire pipeline and models the objective and constraint as independently GPs to match the requirements of our problem (recall §V-A). (3) *Exhaustive Search (ES)*: This solution searches all pipeline configurations. We regard the results reported by this solution as the ground truth.

Metrics: We use two performance metrics. (1) We say a configuration is good if its performance is close to the optimal configuration. To quantify how good a configuration reported by a configuration tuning algorithm is, we define relative processing time (RPT). It is the pipeline processing time of

TABLE II
DETAILED CONFIGURATION SPACE

	Memory (MB) (step size 64)	Workload (2^i/worker)	Configs #
Stage 1	$[1280 \cdots 3008]$	$i \in [0 \cdots 5]$	168
Stage 2	$[1152 \cdots 3008]$	$i \in [1 \cdots 7]$	210
Stage 3	$[512 \cdots 3008]$	$i \in [3 \cdots 10]$	320
	2-stage pipeline configurations #:		35, 280
	LPQ pipeline configurations #:		11, 289, 600
	5-stage pipeline configurations #:		645, 388, 800

the configuration reported by a specific search scheme (RGS, CPS, or `CharmSeeker`) normalized by the optimal pipeline processing time reported by ES. (2) `CharmSeeker` is a configuration tuning tool for video processing workloads running on real-world cloud platforms. All sampled configurations are evaluated in public cloud platforms, which leads to direct monetary charges for users. A good search scheme should be able to find a good configuration with as little expense as possible. Therefore, we define the relative search cost (RSC). It is the search expense of a specific search scheme normalized by the expense of ES.

Configuration space: Although the minimum memory that can be configured for AWS Lambda functions is 128MB, small memory sizes are insufficient for complicated video processing functions. For example, the minimum memory demand for the object detection function (LPQ pipeline) is 1152MB. The FaaS platform also has limits on the maximum concurrency and execution duration [28], which determine the bounds of workload assigned to each worker. To better understand how the number of stages in a pipeline affects the performance of different solutions, we isolate the first two stages of the LPQ pipeline to construct a 2-stage pipeline and synthesize a 5-stage pipeline by duplicating the last two stages in the LPQ pipeline. TABLE II shows the detailed configuration space. To facilitate the exhaustive search for the 5-stage pipeline, we set the step size of its memory increment to 256MB.

B. Experiment methodology

The execution duration of a serverless function instance is vulnerable to a broad range of noise sources, such as underlying heterogeneous infrastructures and network variations. Given a pipeline budget, we find that the optimal configurations are not identical when conducting ES multiple times. To account for the instability and report reliable experimental results, we repeat the evaluation of one configuration 5 times and take the averages as the evaluation results.

The pipeline budget indicates users' trade-off between processing latency and monetary cost. One user may be willing to pay more for a short processing time, while another may tolerate processing delays to save money. A good solution should meet various budget preferences, i.e., finding a good configuration robustly regardless of whether the budget is tight or loose. The absolute budget of a pipeline is affected by the pipeline topology, the processing task, the resolution of videos, etc. As a result, we need a measure that reflects the difficulty levels of finding a feasible configuration for performance evaluation.

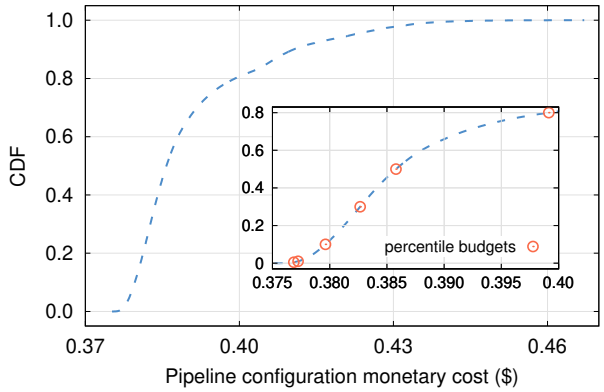


Fig. 7. CDF of LPQ pipeline's monetary costs under different configurations (Video source: Seattle video).

We thus introduce a *percentile pipeline budget* measure based on the monetary cost distribution of all configurations reported by ES. Fig. 7 demonstrates the CDF of the monetary cost of all the 11, 289, 600 configurations for the LPQ pipeline. The nth ($n \in \{0.5, 1, 10, 30, 50, 80\}$) percentile pipeline budgets are marked in the scaled subfigure, and they indicate the probabilities of a configuration satisfying a specific monetary cost constraint. For instance, the point at $(0.37963, 0.1)$ represents there are 10% pipeline configurations whose monetary costs are not greater than \$0.37963, so 0.37963 is set as the 10th percentile pipeline budget. Generally, a decreasing trend in n indicates the increasing difficulties in finding a feasible configuration, especially when the number of evaluations allowed is limited. To make fair comparisons, for a given video, all search solutions (RGS, CPS, and `CharmSeeker`) are evaluated with the same percentile pipeline budgets.

C. Effectiveness of `CharmSeeker`

In this subsection, we evaluate the effectiveness of `CharmSeeker` for different pipelines and budgets. For each video, we allow `CharmSeeker` to perform 20 trials, i.e., running 20 iterations to sample 20 different configurations for each stage. Then, we run RGS and CPS under a similar search cost (cloud expenditures for running samples), i.e., we use the search cost of `CharmSeeker` to decide when the searching process of baselines should be terminated.

Compared to the baselines, `CharmSeeker` is more stable and can pick up a better configuration with fewer search costs. Fig. 8 shows the RPT distribution (first quartile, median, third quartile) of 20 experiments under different search schemes for the Seattle video. The results of the other two videos are similar. We find that compared to baselines, `CharmSeeker` can find a better configuration under the same pipeline budget with fewer search costs. Specifically, for the LPQ pipeline, `CharmSeeker` improves the median RPT by 158.45% (245.47%) over RGS (CPS) under the $0.5th$ percentile pipeline budget. When the budget is sufficient, `CharmSeeker` can pick up a configuration on the median very close to the optimal one. For the 2-stage pipeline, the

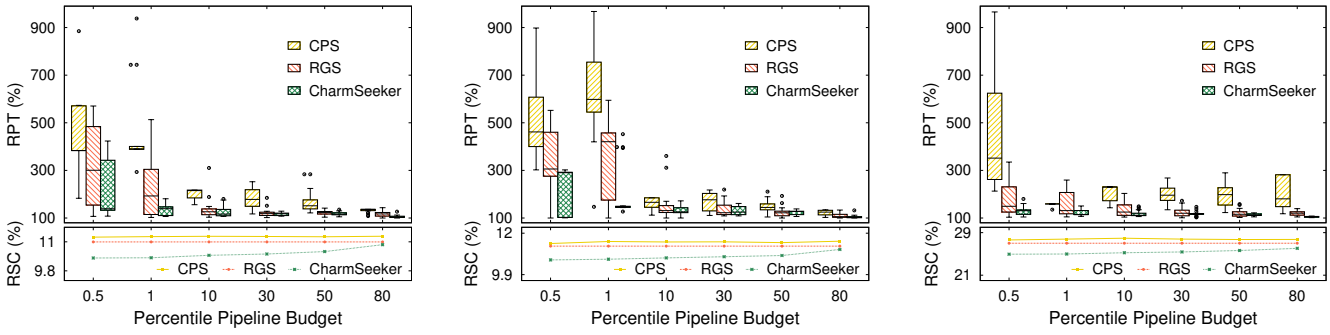


Fig. 8. Comparison of RTP and RSC for different pipelines. The circles in the boxplots represent outliers.

TABLE III
PERCENTAGE OF EXPERIMENTS THAT CAN FIND A FEASIBLE CONFIGURATION.

Budget	CPS		RGS		CharmSeeker	
	$0.5th$	$1st$	$0.5th$	$1st$	$0.5th$	$1st$
LPQ	90%	100%	95%	95%	100%	100%
2-stage	60%	90%	60%	85%	100%	100%
5-stage	65%	100%	60%	100%	100%	100%

median RPT improvements of `CharmSeeker` are more apparent and can achieve up to 408.77% (under the $1st$ percentile pipeline budget, over CPS). As the number of stages increases to 5, the search space of each stage becomes smaller (with a memory step of 256MB) while the configuration space for the whole pipeline increases (the configuration dimension rises). In this case, CPS maintains a significant variance on RPT for the 0.5th percentile pipeline budget, while the variances of RGS and `CharmSeeker` are much smaller since they search subconfigurations for each stage independently.

`CharmSeeker` is more robust than baselines for varying pipeline budgets and pipeline lengths. As a supplement to Fig. 8, TABLE III summarizes the percentage of experiments that found feasible configurations when the pipeline budgets were tight. For those relatively loose budgets (not shown in the table), the percentages are 100%. As the table shows, `CharmSeeker` is robust enough to handle video processing pipelines with different numbers of stages. It can reliably find feasible configurations even when the pipeline budget is exceptionally tight. By comparison, with a tight pipeline budget, the baselines often cannot find feasible configurations, and the performance of CPS further suffers from increasing stages of the pipeline.

As more samples are evaluated, `CharmSeeker` converges to the optimal or near-optimal configuration faster compared with RGS. A search scheme with good convergence should exhibit stability in reporting the optimal configuration as the number of configurations evaluated (or collected samples) increases. Fig. 9 shows the convergence of `CharmSeeker` and RGS under the 0.5th percentile pipeline budget for the LPQ and 2-stage pipeline. The probability of RGS for reporting a feasible configuration increases as the

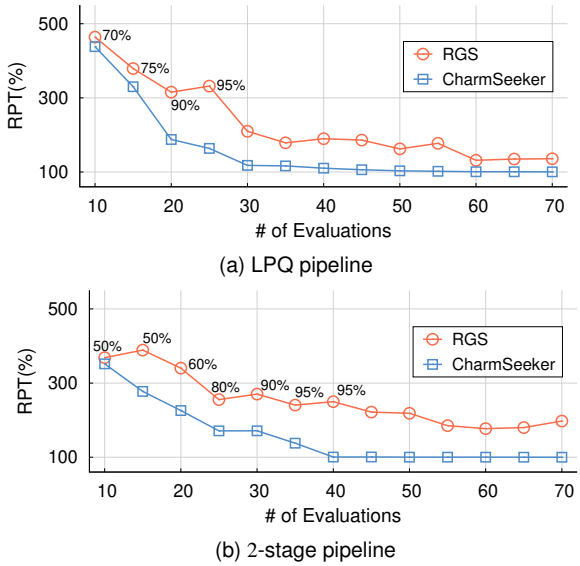


Fig. 9. Variations of RPT with the changing number of evaluations in RGS and `CharmSeeker`. The label next to each data point indicates the frequency of finding a feasible configuration out of 20 experiments. The default value is 100% if not marked out.

number of evaluated configurations increases. Nevertheless, when it is allowed to sample 70 configurations, it still has a high chance of picking a configuration that is far from the optimal one (1.5× the optimal processing time or higher). By contrast, `CharmSeeker` guarantees to report feasible configurations even when the number of allowed configurations for evaluation is small (e.g., 10). It starts to converge to the optimal configuration when the number of evaluated configurations reaches about 40.

D. How `CharmSeeker` Works?

The three subfigures from top to bottom in Fig. 10 illustrate how `CharmSeeker` searches through the feasible configuration space and finds out the optimal configuration. The allowed evaluation number is 20. Each subfigure has 176 columns that denote all 176 feasible configurations for the 2-stage pipeline under the 0.5th percentile pipeline budget. These pipeline configurations are arranged from left to right based on their processing time, i.e., the leftmost column

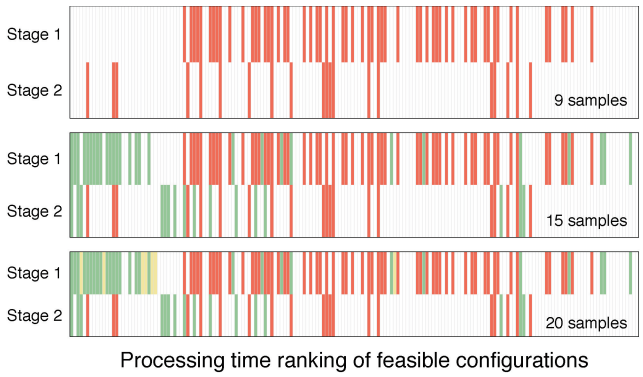


Fig. 10. Snapshots of the optimal configuration searching results after evaluating 9, 15, 20 samples (Colorized in red, green, yellow, respectively).

represents the configuration with the shortest processing time (the optimal configuration under the budget). Each column is further divided into two rows representing the subconfigurations for stage 1 and stage 2 in the corresponding pipeline configuration. When the subconfiguration for a stage is evaluated, all cells representing this subconfiguration in the same stage are colorized. Note that the same subconfiguration may appear in multiple pipeline configurations. For example, subconfiguration [3008 1] for stage 1 can appear both in pipeline configuration [3008 1 2048 8] and [3008 1 3008 2]. Thus, multiple cells in a row can be colorized simultaneously after evaluating one subconfiguration.

The top subfigure in Fig. 10 displays the searched feasible subconfigurations after the first cost optimization step in SBO. This optimization step ends with 9 samples (total evaluated subconfigurations, including feasible and infeasible) for each stage. These searched subconfigurations construct 9 feasible pipeline configurations (both rows in one column are colorized as red). Since this step aims to optimize the monetary cost for each stage regardless of its processing time, searched pipeline configurations are not necessarily located to the left.

According to the number of remaining subconfigurations to be evaluated (11 in this case), two budget allocation vectors are sampled for the next budget-constrained time optimization step. The middle and bottom subfigures in Fig. 10 show all evaluated subconfigurations after applying the first and second budget allocation vectors, respectively. The new subconfigurations added in the optimization process of the first (second) budget allocation vector are colorized as green (yellow). The evaluated configurations in this step are close to the left, indicating that the second step of SBO is indeed searching towards the direction of the optimal feasible configuration.

E. Configuration Extrapolation

1) Performance for Recurring Workloads: `CharmSeeker` relies on representative sample videos to pick up the optimal configuration for recurring video processing workloads. For example, consider a recurring job that counts cars driving through an intersection at rush hours on weekdays. We can choose one day's video as a representative and run `CharmSeeker` to pick up the optimal configuration. The picked configuration is then applied to process the videos for other days. The first issue is how to pick up a representative sample video. Designing an automated method is challenging since it requires deep insights into the video content and processing algorithms. Complex models are needed to extract video content features and build metrics to measure the similarity of workloads. Although this is an interesting exploration direction, it is beyond the scope of this work. Following previous works [21], `CharmSeeker` currently relies on human intuitions to select representative videos.

With the representative sample video, we need to verify if the configuration searched on the video can maintain its performance for similar videos. We divide each video in TABLE I into two equal-length segments (each includes 195 chunks), marked as video#1 and video#2, respectively. We take video#1 as the representative sample video and video#2 as the test video. We then run `CharmSeeker` on video#1 and apply the picked configuration to video#2 for the performance test. Intuitively, the two videos generated in this way are very similar. They have the same video specifications, such as resolution and framerate. More importantly, they were shot continuously in a relatively small time window (33 minutes) by the same camera. We inspected the video content and found that the video#1 roughly covers the video#2's scenes, illuminations, and movement conditions.

The pipeline budgets in this experiment indicate how much users expect to spend on processing 195 video chunks. The budget values are set with the method mentioned in §VI-B based on the ES results on video#1. Given the pipeline budget, we refer to the configuration reported on video#1 as the *configuration of interest*. Ideally, the processing cost of video#2 under the *configuration of interest* should be that of video#1. However, in reality, influenced by video content, the processing costs of the representative video and the test video may not be exactly the same. Being aware of this, users usually can tolerate performance variations in an acceptable range, e.g., ±10 % for the processing time and ±2 % for the monetary cost in our assumption.

To quantify the performance of the *configuration of interest* on video#2, we calculate the percent deviations from video#1's processing time and monetary cost. Fig. 11 shows the extrapolation performance of the *configurations of interest* for all videos in our dataset. As shown, the percent deviation of `CharmSeeker` falls in ±10 % for the processing time, and ±2 % for the monetary cost in all cases. It indicates that the *configurations of interest* can perform well on similar videos, and `CharmSeeker` can pick up a good configuration for recurring video processing workloads.

In practice, the representative video may gradually lose its representativeness for recurring jobs over time, making the actual performance of the *configurations of interest* suboptimal. A simple mechanism to address this issue is setting a threshold. Users can rerun SBO when the gap between the picked configuration's expected performance and actual performance exceeds the threshold. The current implementation of SBO relies on random initialization to construct the initially known sample sets. A potential improvement is to rerun SBO from successful configurations in previous runs.

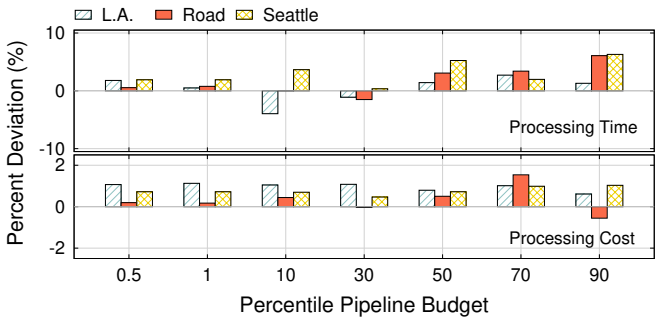


Fig. 11. Percent deviation of the *configuration of interest* on video#2. The configuration of interest is the searched optimal configuration with the highest frequency out of 20 repeat experiments on video#1 (2-stage pipeline).

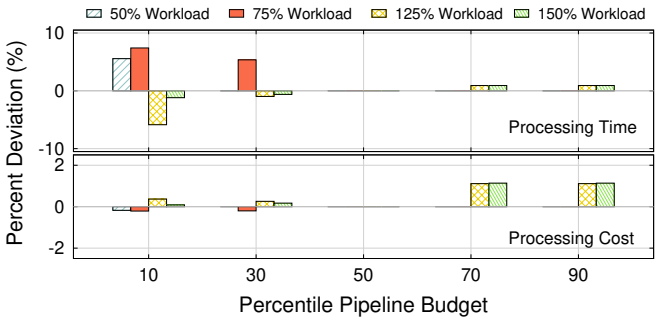


Fig. 12. Percent deviation of the *configurations of interest* on varying input workload sizes. (2-stage pipeline).

Such a "warm-start" technique has been successfully applied in BO hyperparameter optimization [49] to reduce the number of evaluations. Integrating this technique into `CharmSeeker` is worthy of further exploration.

2) Sensitivity to Varying Input Workload Sizes: For recurring video processing jobs, the size of the input workload (e.g., the number of video chunks) fed to a pipeline at one time is normally fixed. Nevertheless, it is interesting to analyze `CharmSeeker`'s sensitivity to varying input workload sizes as it may extend the applications of `CharmSeeker`. As the absolute pipeline budget is affected by the input workload size, we assume that their relationship is linear. For example, the absolute pipeline budget for processing half of a video is half of that for processing the entire video.

The experiment is conducted as follows. We first obtain 10-minute (120 video chunks), 15-minute (180 video chunks), 20-minute (240 video chunks), 25-minute (300 video chunks), and 30-minute (360 video chunks) video clips from the Seattle video, representing the 50%, 75%, 100%, 125% and 150% input workload, respectively. Then, we choose the 20-minute clip (100% workload) as the representative to pick up configurations under various pipeline budgets. We refer to the obtained configurations as *configurations of interest*. Next, we scale the absolute pipeline budgets according to the input workload sizes and employ ES to obtain the ground-truth optimal configurations for varying input workload sizes.

We then evaluate the performance of the *configurations of interest* on 50%, 75%, 125%, and 150% input workloads and compare the results with that of the ground-truth optimal configurations. As shown in Fig. 12, the percent deviations of the *configurations of interest* fall in ±10 % for the processing time, and ±2 % for the monetary cost in all cases. It indicates that `CharmSeeker` is robust to varying input workload sizes.

VII. Further Discussions

The LPQ pipeline has a serial structure, which is very common for video processing pipelines. However, parallel structures can also be found in a pipeline. For instance, cascading a license plate recognition module and a face recognition module parallel after the object detection module to simultaneously query people and license plates. These parallel modules may have different resource requirements, so we cannot consider them as a single stage. One straightforward way to solve this issue is to treat them as different serial stages but only count the larger one of their processing times in the pipeline processing time. Thus, our solution can be easily extended to support different topologies.

VIII. Related Work

Video Processing and Analytics systems: Recent years have witnessed the proliferation of live video analytics systems [50], [51]. The key idea is to balance resources and accuracy by adjusting general configuration knobs, such as framerate, resolution, and processing models. For example, `VideoStorm` [50] profiled the resource demand and accuracy of different knob combinations for each live video query offline. It then adjusted configuration knobs for large-scale concurrent queries according to their quality and lag goals online. `VideoEdge` [51] is a geo-distributed live video analytics system. It tuned the general configuration knobs and the placement knob (private clusters or public clouds) to strike the best trade-off between resources and accuracy. Different from these VM cluster-based video analytics systems, `CharmSeeker` is not specifically designed for live or geo-distributed video analytics. It aims to optimize the processing time and monetary costs for serverless video processing applications. Users can pre-tune these general knobs according to their resource and accuracy goals. After the videos are streamed to the cloud, users can further utilize `CharmSeeker` to tune serverless functions' memory size and workload. In this sense, `CharmSeeker` complements these existing systems and provides them with a readily available tool once they migrate to serverless computing platforms.

Applications of BO: With the minimal assumptions about the optimization problem and only a few samples required to find an appropriate solution, BO is preferred by researchers and engineers to solve black-box optimization problems in automatic machine learning systems. For instance, `Auto-sklearn` [52] resorted to BO to achieve the joint optimization of machine learning algorithm selections and associated hyperparameter configuration. Google `Vizier` [29] harnessed BO to optimize machine learning models and other systems.

As the cloud computing represented by VMs gains popularity, BO has been used to configure resources for VM-based cloud systems [19], [21]. For example, `CherryPick` [21]

unearthed the potential of BO in picking up the best VM cluster configuration for big data analytic workloads. `Arrow` [19] leveraged low-level information to improve the performance of the vanilla BO in configuring VM instances. Unlike existing BO-based configuration tuning tools designed for VMs, our work targets configuration tuning for serverless functions. In particular, our work well complements these existing efforts by providing a novel SBO to ease the configuration tuning of serverless video processing pipelines.

IX. Conclusion

In this paper, we demonstrated the great potential in reducing processing time and monetary cost via efficient configuration tuning and identified its practical challenges. Motivated by the observations from our measurement study, we designed `CharmSeeker`, a tool that uses a carefully designed SBO algorithm to automatically tune configurations for serverless video processing pipelines. A prototype with AWS Lambda is further implemented for evaluation. Our extensive experiments showed the effectiveness and superiority of `CharmSeeker` over state-of-art solutions.

References

[1] YouTube, "Youtube for press," https://www.youtube.com/about/press/, [Online].

[2] Cisco, "Cisco visual networking index: Forecast and trends, 2017-2022 (white paper)," February 2019.

[3] Q. Huang, P. Ang, P. Knowles, T. Nykiel, I. Tverdokhlib, A. Yajurvedi, P. Dapolito IV, X. Yan, M. Bykov, C. Liang *et al.*, "Sve: Distributed video processing at facebook scale," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, 2017, pp. 87–103.

[4] AWS, "Netflix on aws," https://aws.amazon.com/solutions/case-studies/netflix/, [Online].

[5] G. Cloud, "Fastly: Building a high-quality video delivery service for vimeo," https://cloud.google.com/customers/vimeo/, [Online].

[6] AWS, "Aws case study: Encoding.com," https://aws.amazon.com/solutions/case-studies/encoding/, [Online].

[7] Y. Zhu, S. D. Fu, J. Liu, and Y. Cui, "Truthful online auction toward maximized instance utilization in the cloud," *IEEE/ACM Transactions on Networking*, vol. 26, no. 5, pp. 2132–2145, 2018.

[8] Amazon, "Serverless computing - amazon web services," https://aws.amazon.com/serverless/, [Online].

[9] AWS, "Aws lambda," https://aws.amazon.com/lambda/, [Online].

[10] Google, "Cloud functions," https://cloud.google.com/functions/, [Online].

[11] Microsoft, "Azure functions," https://azure.microsoft.com/en-ca/services/functions/, [Online].

[12] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[13] S. Fouladi, R. S. Wahby, B. Shacklett, K. Balasubramaniam, W. Zeng *et al.*, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, 2017, pp. 363–376.

[14] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC'18)*, 2018, pp. 263–274.

[15] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*, 2017, pp. 445–451.

[16] Amazon, "Best practices for working with aws lambda functions," https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html#function-configuration, [Online].

[17] M. Zhang, Y. Zhu, C. Zhang, and J. Liu, "Video processing with serverless computing: A measurement study," in *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'19)*, 2019, pp. 61–66.

[18] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: Tapping the performance potential of systems via automatic configuration tuning," in *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*, 2017, pp. 338–350.

[19] C. Hsu, V. Nair, V. W. Freeh, and T. Menzies, "Arrow: Low-level augmented bayesian optimization for finding the best cloud vm," in *Proceedings of the 38th International Conference on Distributed Computing Systems (ICDCS'18)*. IEEE, 2018, pp. 660–670.

[20] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, 2016, pp. 363–378.

[21] 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 *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, 2017, pp. 469–482.

[22] A. Klimovic, H. Litz, and C. Kozyrakis, "Selecta: Heterogeneous cloud storage configuration for data analytics," in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC'18)*, 2018, pp. 759–773.

[23] AWS, "Aws budgets - amazon web services," https://aws.amazon.com/aws-cost-management/aws-budgets/, [Online].

[24] Amazon. Amazon ec2 - amazon web services. https://aws.amazon.com/ec2/. [Online].

[25] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC'18)*, 2018, pp. 923–935.

[26] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC'18)*, 2018, pp. 133–146.

[27] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, 2020, pp. 419–434.

[28] AWS, "Aws lambda limits," https://docs.aws.amazon.com/lambda/latest/dg/limits.html, [Online].

[29] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, pp. 1487–1495.

[30] E. Brochu, V. M. Cora, and N. De Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," *arXiv preprint arXiv:1012.2599*, 2010.

[31] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2016.

[32] Z. Wang, M. Zoghi, F. Hutter, D. Matheson, and N. De Freitas, "Bayesian optimization in high dimensions via random embeddings," in *Twenty-Third international joint conference on artificial intelligence (IJCAI'13)*, 2013.

[33] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[34] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems (NeurIPS'12)*, 2012, pp. 2951–2959.

[35] M. A. Gelbart, J. Snoek, and R. P. Adams, "Bayesian optimization with unknown constraints," *arXiv preprint arXiv:1403.5607*, 2014.

[36] J. Gardner, C. Guo, K. Weinberger, R. Garnett, and R. Grosse, "Discovering and exploiting additive structure for bayesian optimization," in *Artificial Intelligence and Statistics*, 2017, pp. 1311–1319.

[37] K. Kandasamy, J. Schneider, and B. Póczos, "High dimensional bayesian optimisation and bandits via additive models," in *International conference on machine learning (ICML'15)*, 2015, pp. 295–304.

[38] M. Bilal and M. Canini, "Towards automatic parameter tuning of stream processing systems," in *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*, 2017, pp. 189–200.

[39] AWS, "Boto3 documentation," https://boto3.amazonaws.com/v1/documentation/api/latest/index.html, [Online].

[40] G. APIs, "Python client for cloud functions," https://github.com/googleapis/python-functions/tree/master/docs, [Online].

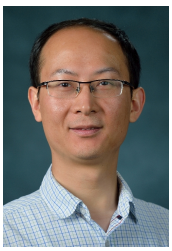
[41] A. Lottarini, A. Ramirez, J. Coburn, M. A. Kim, P. Ranganathan, D. Stodolsky, and M. Wachsler, "Vbench: Benchmarking video transcoding in the cloud," in *Proceedings of the Twenty-Third International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18), 2018, pp. 797–809.
[42] YouTube, "Driving downtown - seattle 4k - usa," https://www.youtube.com/watch?v=rjVD1-1mbuE, [Online].
[43] ——, "Los angeles 4k - night drive," https://www.youtube.com/watch?v=lTvYjERVAnY&t=3961s, [Online].
[44] ——, "M6 motorway traffic," https://www.youtube.com/watch?v=PNCJQkvALVc&t=43s, [Online].
[45] FFmpeg, "A complete, cross-platform solution to record, convert and stream audio and video." https://ffmpeg.org/, [Online].
[46] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.
[47] OpenALPR, "Automatic license plate recognition," https://www.openalpr.com/, [Online].
[48] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *Journal of machine learning research*, vol. 13, no. 2, 2012.
[49] J. Kim, S. Kim, and S. Choi, "Learning to warm-start bayesian hyperparameter optimization," *arXiv preprint arXiv:1710.06219*, 2017.
[50] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, 2017, pp. 377–392.
[51] C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose, "Videoedge: Processing camera streams using hierarchical clusters," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing (SEC'18)*, 2018.
[52] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Advances in Neural Information Processing Systems (NeurIPS'15)*, 2015, pp. 2962–2970.

Miao Zhang received her B.Eng. degree from Sichuan University in 2015, and her M.Eng. degree from Tsinghua University in 2018. She is currently a Ph.D. student at Simon Fraser University, Canada. Her research areas include cloud and edge computing, and multimedia systems and applications.

Yifei Zhu received his MPhil degree from Hong Kong University of Science and Technology in 2015, and his Ph.D. degree in computer science from Simon Fraser University, Canada, in 2020. He is currently an assistant professor at Shanghai Jiao Tong University. His research areas include edge computing and multimedia networking.

Jiangchuan Liu (S'01-M'03-SM'08-F'17) is a Professor in the School of Computing Science, Simon Fraser University, British Columbia, Canada. He is a Fellow of The Canadian Academy of Engineering, an IEEE Fellow, and an NSERC E.W.R. Steacie Memorial Fellow. He is also a Distinguished Guest Professor of Tsinghua Shenzhen International Graduate School. He received the BEng degree (cum laude) from Tsinghua University, Beijing, China, in 1999, and the PhD degree from The Hong Kong University of Science and Technology in 2003, both in computer science. He is a co-recipient of the inaugural Test of Time Paper Award of IEEE INFOCOM (2015), ACM SIGMM TOMCCAP Nicolas D. Georganas Best Paper Award (2013), ACM Multimedia Best Paper Award (2012), and IEEE MASS Best Paper Award (2021). His research interests include multimedia systems and networks, cloud and edge computing, social networking, online gaming, and Internet of things/RFID/backscatter. He has served on the editorial boards of IEEE/ACM Transactions on Networking, IEEE Transactions on Network Sciences and Engineering, IEEE Transactions on Big Data, IEEE Transactions on Multimedia, IEEE Communications Surveys and Tutorials, and IEEE Internet of Things Journal. He is a Steering Committee member of IEEE Transactions on Mobile Computing and Steering Committee Chair of IEEE/ACM IWQoS (2015-2017). He was TPC Co-Chair of IEEE INFOCOM'2021 and IEEE Satellite'2022.



Feng Wang (S'07-M'13-SM'18) received both the Bachelor's degree and Master's degree in Computer Science and Technology from Tsinghua University, Beijing, China in 2002 and 2005, respectively. He received the PhD degree in Computing Science from Simon Fraser University, Burnaby, British Columbia, Canada in 2012. He is currently an Associated Professor in the Department of Computer and Information Science at the University of Mississippi, University, MS, USA. He is a member of IEEE and a recipient of IEEE ICME Quality Reviewer Award (2011). He is a Technical Committee Member of Elsevier Computer Communications. He served as Program Vice Chair in International Conference on Internet of Vehicles (IOV) 2014, and as TPC co-chair in IEEE CloudCom 2017 for Internet of Things and Mobile on Cloud track. He also serves as TPC member in various international conferences such as IEEE INFOCOM, ICPP, IEEE/ACM IWQoS, ACM Multimedia, IEEE ICC, IEEE GLOBECOM and IEEE ICME.

Fangxin Wang (S'15-M'20) is an assistant professor at The Chinese University of Hong Kong, Shenzhen (CUHKSZ). He received the Ph.D., M.Eng., and B.Eng. degree all in Computer Science and Technology from Simon Fraser University, Tsinghua University, and Beijing University of Posts and Telecommunications, respectively. Before joining CUHKSZ, he was a postdoctoral fellow at the University of British Columbia. Dr. Wang's research interests include Multimedia Systems and Applications, Cloud and Edge Computing, Deep Learning and Big Data Analytics, Distributed Networking and System. He lead the intelligent networking and multimedia lab at CUHKSZ. He has published more than 30 papers at top journal and conference papers, including INFOCOM, Multimedia, ToN, TMC, IOTJ, etc. He served as the publication chair of IEEE/ACM IWQoS, TPC member of IEEE ICC, and reviewer of many top conference and journals, including INFOCOM, Multimedia, ToN, TMC, IOTJ.