Columbo: A Reasoning Framework for Kubernetes' Configuration Space

Matthijs Jansen
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Sacheendra Talluri
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Krijn Doekemeijer
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Nick Tehrany*
BlueOne Business Software LLC
Beverly Hills, California, USA

Alexandru Iosup
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Animesh Trivedi*
IBM Research Europe
Zurich, Switzerland

Abstract

Resource managers such as Kubernetes are rapidly evolving to support low-latency and scalable computing paradigms such as serverless and granular computing. As a result, Kubernetes supports dozens of workload deployment models and exposes roughly 1,600 configuration parameters. Previous work has shown that parameter tuning can significantly improve Kubernetes' performance, but identifying which parameters impact performance and should be tuned remains challenging. To help users optimize their Kubernetes deployments, we present Columbo, an offline reasoning framework to detect and resolve performance bottlenecks using configuration parameters. We study Kubernetes and define its workload deployment pipeline of 6 stages and 26 steps. To detect bottlenecks, Columbo uses an analytical model to predict the best-case deployment time of a workload per pipeline stage and compares it to empirical data from a novel benchmark suite. Columbo then uses a rule-based methodology to recommend parameter updates based on the detected bottleneck, deployed workload, and mapping of configurations to pipeline stages. We demonstrate that Columbo reduces workload deployment time across its benchmark suite by 28% on average and 79% at most. We report a total execution time decrease of 17% for data processing with Spark and up to 20% for serverless workflows with OpenWhisk. Columbo is open-source and available at https://github.com/atlarge-research/continuum/tree/columbo.

CCS Concepts

• **Software and its engineering** → **Software configuration management and version control systems**; *Software performance*; • **Computer systems organization** → *Cloud computing*.

Keywords

Configuration tuning, resource management, Kubernetes

ACM Reference Format:
Matthijs Jansen, Sacheendra Talluri, Krijn Doekemeijer, Nick Tehrany, Alexandru Iosup, and Animesh Trivedi*. 2025. Columbo: A Reasoning Framework for Kubernetes' Configuration Space. In *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE '25), May 5–9, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3676151.3719374

*Work done partially while the author was at Vrije Universiteit, Amsterdam.

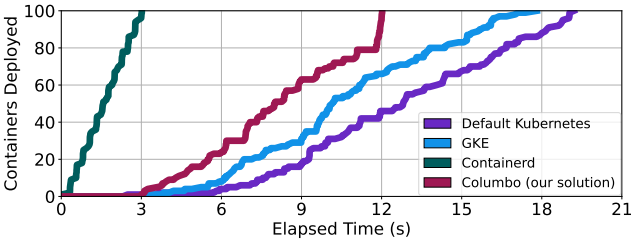

Figure 1: Our approach, Columbo, reduces deployment time for 100 containers by 37% compared to default Kubernetes.

1 Introduction

Containerization is a popular way to package, distribute, and run data processing [9] and serverless [8] applications. Single-machine container runtimes, such as Docker and containerd, start and stop containers, with distributed container managers on top controlling containers across computing infrastructures. Kubernetes is the most popular distributed container manager and is used by 64% of end-users in production [5]. To support diverse workloads and infrastructures, Kubernetes has developed an extremely modular, distributed, configurable, and, consequently, complex (containerized) workload deployment pipeline. Although such complex pipelines have been defined and studied for systems like Spark [31] and Mesos [7], and understanding the pipeline was found to be key in optimizing performance, Kubernetes performance studies are limited to isolated pipeline components [4, 18] and do not consider the pipeline or the configuration space that manages its functionality as a whole. To address this gap, in this work, we conduct the first extensive analysis of Kubernetes' workload deployment pipeline and propose the *Columbo* framework to update Kubernetes configurations for improved container deployment performance.

We show the impact of Kubernetes configurations on the deployment of 100 containers on a single machine in Figure 1. We make three observations. First, the distributed Kubernetes with its default configuration introduces a *significant performance overhead* over the single-machine container runtime containerd, being 6.4x slower. Hence, we conclude that the performance and scalability overheads lie with Kubernetes, not the container deployment. Second, even a provider-managed Kubernetes deployment, such as shown here

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.
ICPE '25, Toronto, ON, Canada
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1073-5/2025/05
https://doi.org/10.1145/3676151.3719374

Table 1: Survey of scientific literature on the use of Kubernetes and its configuration since 2019.

Category	Frequency
Uses Kubernetes	40
Mentions configuration	12
Artifact available	19
Configuration in artifact	9

from Google Kubernetes Engine (GKE), only achieves minimal (7%) performance gains over a default deployment. Third, Columbo, our approach, significantly improves Kubernetes' performance (37%) over the default via an automatic rule-based configuration tuning methodology. From these observations, we conclude that it is imperative for any production-grade setup to evaluate its Kubernetes configuration to meet application demands.

However, optimizing the configuration space beyond the default setting is non-trivial. We identify three challenges toward solving this problem. First, there is a *lack of understanding of Kubernetes' workload deployment pipeline and configuration space.* We report that a modern, fault-tolerant Kubernetes setup (as of version 1.27) can contain upwards of 9 distributed components with multiple parallel replicas, a workload deployment pipeline of up to 26 steps, and 1,600 configuration parameters to configure these steps (Section 2). This is much more complex than related systems such as Mesos (version 1.11.0) and Spark (version 3.4.1), with 279 and 579 parameters, respectively. While the complexity of these systems has been studied in-depth and is well-understood, which has resulted in advanced configuration optimization strategies [7, 31], Kubernetes performance studies are limited (Section 8). We argue that Kubernetes' complexity requires a root-cause analysis approach to facilitate automatic and efficient configuration tuning.

Second, there is a *lack of tools to reason how parameters affect Kubernetes' workload deployment* Existing work focuses on the performance of specific pipeline components [19] or fit to particular applications [17], not on the operation of the pipeline as a whole [22]. We propose a methodology to detect performance bottlenecks in Kubernetes' workload deployment pipeline using analytical models and empirical benchmarks, and construct configuration rules to programmatically find parameters that resolve bottlenecks.

Third, there is a *lack of guidelines regarding how to optimize Kubernetes for a given workload or infrastructure.* We conduct a survey of papers published (Table 1) since 2019 in systems conferences (ATC, EuroSys, HPDC, ICDCS, ICPE, Middleware, NSDI, OSDI, SoCC) with their artifacts. We report that out of the 40 papers that use Kubernetes, less than 30% mention its configuration and none the deployment pipeline. We further analyze the Kubernetes GitHub repository [16], a prominent location for practitioners to discuss Kubernetes, and find that out of 10,000 issues, 20% mention Kubernetes' configuration (2,024), 33% mention the configuration API (3,253), and 10% mention YAML files that store configurations (1,026), and are part of Kubernetes' fragmented configuration system. Our survey demonstrates that Kubernetes' configuration is a source of concern for practitioners because of the large, complex configuration space and lack of structured guidelines on how to navigate the space. We demonstrate how Columbo requires a one-time, expert-driven effort to automate its day-to-day operations.

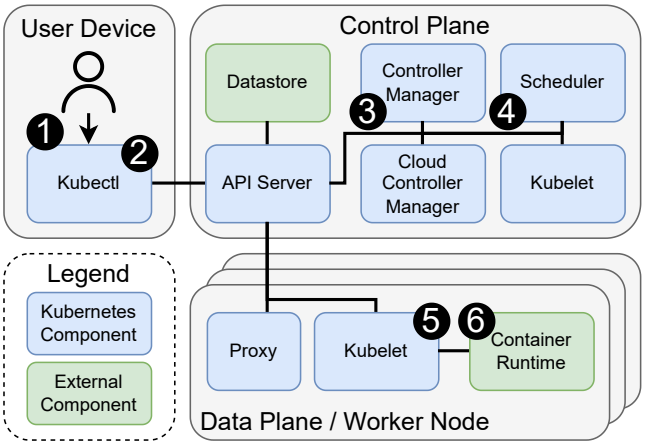

Figure 2: Kubernetes' architecture, distributed over the user device, control plane, and data plane/worker nodes.

To address the aforementioned challenges, in this work, we systematically study Kubernetes and define its workload deployment pipeline to model, measure, and optimize how Kubernetes deploys applications. To this end, we design and implement Columbo, a reasoning framework to detect and resolve performance bottlenecks in pipeline stages using Kubernetes configuration parameters. Columbo consists of an analytical model, a benchmark suite, and a rule-based methodology. We make the following contributions:

(1) We analyze Kubernetes and argue that its complexity requires a root-cause analysis-based approach to efficiently find configuration parameters that optimize workload deployment (Sections 2)
(2) We present the design of Columbo, an offline reasoning framework for faster Kubernetes workload deployment by optimizing its configuration space through performance bottleneck detection and parameter optimization rules (Section 3).
(3) We define Kubernetes' workload deployment pipeline of 6 stages and 26 steps, and capture its best-case deployment time in an analytical model (Section 4).
(4) We synthesize configuration rules that allow Columbo to find parameters updates that resolve detected bottlenecks (Section 5).
(5) We demonstrate Columbo's user-driven and automated day-to-day operations and evaluate its ability to resolve bottlenecks (Section 6). Compared to the default, Columbo decreases deployment time by 28% across a novel benchmark suite and total execution time by 17% for data processing on Spark and up to 20% for serverless workflows on OpenWhisk (Section 7).
(6) We publish Columbo as an open-source artifact: https://github.com/atlarge-research/continuum/tree/columbo.

2 Background and Motivation

We first present a study on Kubernetes' architecture (Section 2.1) and configuration space (Section 2.2) to understand the complexity in optimizing workload deployment performance. Next, we discuss limitations of existing performance optimization approaches and the need for Columbo (Section 2.3).

2.1 Architecture Overview

Kubernetes is a system for deploying containerized applications. It consists of 9 components that form the control plane, responsible for

managing containers, and the data plane, responsible for executing control plane decisions. Users describe containerized workloads in workload files and submit them via the command line tool *kubectl* (❶ in Figure 2) to the control plane (❷). A workload file describes workload objects, which are services that run indefinitely or jobs that run until completion. This paper focuses on job-type workloads because they benefit most from optimizing deployment overhead due to their limited lifespan.

The control plane consists of multiple, possibly replicated, components and, together with the data plane, depends on external components for specialized services, such as a database and container runtime. Central to the control plane is the *API server*. It provides an interface for reading and writing to a persistent *datastore*. This datastore stores the current and desired state of workloads. All other control plane components move a workload's current state toward its desired state using control loops. A control loop (i) watches the datastore through the API server, (ii) reads new or updated objects, (iii) performs an action based on the read object, and (iv) writes back objects. For example, the *controller manager* reads submitted job objects, creates pod objects described in the jobs, and writes them to the datastore (❸). Pods are the scheduling unit of Kubernetes and house containers. The *scheduler* reads the pod objects, makes a scheduling decision, and writes that back (❹).

The data plane is located on worker nodes that execute workloads. Each worker has a *kubelet* that communicates with a *container runtime*. The kubelet reads pod objects that have been scheduled onto its node (❺), tasks the container runtime with creating pods and containers (❻), and writes status updates to the control plane.

2.2 Configuration Complexity

Kubernetes' API defines objects, including workloads (e.g., jobs), infrastructures (e.g., nodes), and control and data plane configurations (e.g., scheduling algorithm). In this work, we focus on optimizing Kubernetes workload deployment performance through control and data plane configurations, which we call the *configuration space*. We survey the documentation of the configuration space and present an overview for Kubernetes version 1.27 in Table 2.

Users modify configuration parameters by submitting *configuration files*, similar to workload files, or via the command line (*CLI*) when starting a component. Kubernetes configuration files cover 234 API objects that group 1,067 parameters, and the CLI covers an additional 531 parameters for a total of 1,598. Additional parameters are not exposed by the API and only accessible via Kubernetes' source code. We leave configurations from external components such as the datastore and container runtime as future work.

2.3 Limitations of Existing Approaches

Existing performance optimization approaches for Kubernetes [13, 32, 34] consider either its whole configuration space (e.g., brute-force) or require users to define which parameters and values to optimize within. The brute-force approach is extremely time-intensive as applying a single configuration update requires the affected components to restart or recompile, which takes up to multiple minutes. Therefore, brute-force approaches either stop preemptively or update many parameters at once to reduce cost. However, both have significant drawbacks: Either only a part of the configuration space is explored (preemptive), or the performance impact of individual parameters cannot be correctly quantified, leaving in those with a negative impact (many at once). The user-input approach considers a subset of the parameter space but is time-intensive nonetheless. Moreover, the subset may not include the configuration leading to optimal performance and requires user expertise to be defined.

Table 2: Summary of Kubernetes v1.27's configuration space: A total of 234 API objects and 1,598 parameters.

Component	File		CLI
	Objects	Params	Params
API server	39	149	152
Controller manager	37	147	132
Kubelet	31	206	133
Proxy	9	61	59
Scheduler	69	315	55
Other	50	189	0
Total	234	1,067	531

To resolve the shortcomings of existing approaches, we argue for a new and automated approach to configuration optimization using root-cause analysis. With root-cause analysis, we reason about what parameters optimize performance, allowing us to consider the entire configuration space time-efficiently. Our approach consists of two parts: First, a one-time, expert-driven effort to define Kubernetes' workload deployment pipeline, model its performance, and create configuration update rules. Second, building Columbo, a workload-dependent, user-driven bottleneck detection framework.

3 Columbo Design

We present the design of Columbo, an automated offline configuration tuning framework for faster container deployment in Kubernetes. We first outline Columbo's day-to-day use by non-expert users, which only requires input on the workload and Kubernetes cluster to tune configurations for. Next, we examine three research questions that form the core of a one-time, expert-driven effort to enable Columbo's highly-automated day-to-day operations.

3.1 Day-to-day Operation

A non-expert user supplies Columbo with a workload configuration to optimize the deployment of. If the user wants to optimize for a range of workloads, they should supply the most resource-demanding workload to Columbo as less demanding workloads will benefit equally from their optimization, but not vice versa. We discuss this in-depth in Section 4. Additionally, the user supplies a Kubernetes configuration which Columbo modifies to bring the deployment performance optimization about (❶ in Figure 3). The default or a modified Kubernetes configuration can be used, allowing any cluster to be optimized by Columbo.

Columbo follows an iterative root-cause analysis approach where it proposes a single configuration update at a time that is predicted to increase deployment performance the most. The iterativity guarantees that the performance impact of each configuration update can be accurately quantified. The root-cause analysis consists of two parts: Bottleneck detection and rule-based parameter attribution. For bottleneck detection, Columbo defines Kubernetes' workload

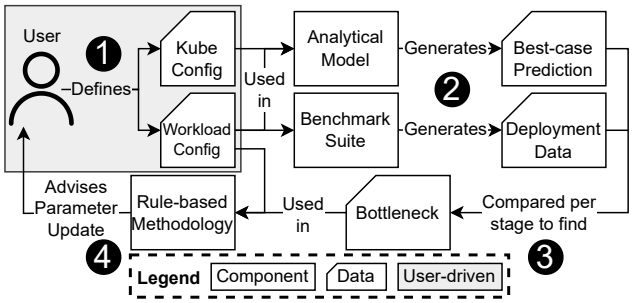

Figure 3: Design of the Columbo Reasoning Framework.

Table 3: Kubernetes architecture mapped to the AtLarge datacenter scheduling reference architecture [1]. proc= processing; man= management; Sched= scheduler; Cont= container.

Responsibility	Kubectl	Controller Manager	Scheduler	Kubelet
Job proc.	Stage 1	Stage 2		
Pod proc.		Stage 3		
Sched. man.			Stage 4	
Pod man.				Stage 5
Cont. man.				Stage 6

deployment pipeline and predicts the time the user's workload spends in each pipeline stage with an analytical model (❷). Similarly, Columbo measures the time spent per pipeline stage using a benchmark suite on a Kubernetes cluster. The stage with the largest discrepancy between the best-case prediction and real-world measurements has the largest performance optimization potential and is marked as the primary workload deployment bottleneck (❸).

Columbo's root-cause analysis depends on the hardware it executes on because application deployment could be constrained by resources rather than Kubernetes configuration. Deploying applications faster typically involves executing more work in less time, hence increasing resource utilization and depending on resource availability. Columbo reports resource utilization and recommends users to add more resource if it detects high utilization. We recommend users to deploy Columbo on the most performant hardware under consideration to uncover configuration bottlenecks that would remain hidden if deploying Columbo only on resource-constrained devices. Users can execute Columbo on multiple hardware setups to adapt to various resource constraints in a single configuration. Columbo does not assume anything about the underlying hardware and adapts to any platform Kubernetes supports.

Columbo uses rule-based parameter attribution and a rate-limiting throughput analysis to recommend configuration parameters and values that resolve the primary bottleneck. Three findings underpin our methodology: (i) each workload pipeline stage processes one type of workload object, such as the scheduler only processing pod objects; (ii) a part of the configuration space consists of numerical parameters that define the (concurrent) processing rate of such objects through stages, such as the *kube-api-qps* parameter defining the number of job and pod objects that may be written to the database per second; (iii) these parameters often limit the object processing rate, for example, when *kube-api-qps* allows for 10 job objects to be written per second while a user wants to create 100 jobs at once, creating artificial rate-limiting delays. Columbo requires a one-time effort to construct a parameter rule per configuration parameter. A rule defines (i) what pipeline stage a parameter affects, (ii) what API workload object a parameter affects, and (iii) the current value (processing limit) of the parameter. Columbo selects the subset of parameters that map to the primary bottleneck stage, and further reduces the selection of parameters to those which limit the processing of objects from the user workload. For the prior *kube-api-qps* example, Columbo suggests the user to update this parameter to the number of job objects in their workload (i.e., 100 objects), and informs the user of the predicted performance gain (❹). The user may update their Kubernetes configuration and restart Columbo for a new configuration suggestion.

3.2 Expert-driven Design Requirements

The day-to-day operation of Columbo is highly-automated and simplified for non-expert users but requires a one-time, expert-driven effort to define Kubernetes' workload deployment pipeline, create a performance model for the pipeline, build a benchmark suite, and construct configuration rules. We pose 3 research questions that drive the development of those components.

RQ1: How do we define and model Kubernetes' workload deployment pipeline? We find that Kubernetes' use of control loops transforms its workload deployment process in a pipeline where each pipeline stage processes a single workload object and generates one or more new objects to be consumed by the next stage. We define Kubernetes' workload deployment pipeline using the AtLarge reference architecture for datacenter scheduling [1], which provides a blueprint for resource management pipelines (Section 4). The mapping results in a pipeline for Kubernetes of 6 stages and 26 steps, which' performance we capture in an analytical model.

RQ2: How do we reason about performance bottlenecks in Kubernetes' pipeline and attribute configuration updates? We build a benchmark suite to gather real-world workload deployment performance data which Columbo compares to the best-case analytical model prediction to find performance bottlenecks (Section 6). Next, we present a methodology to prune the Kubernetes configuration space to only include performance-related parameters and build configuration optimization rules for each (Section 5).

RQ3: How do non-expert users use Columbo to optimize their workloads? We synthesize a methodology for how users can use Columbo and iteratively apply advised configuration parameter updates to optimize workload deployment performance (Section 6.3). Users only have to decide whether to use the suggested parameter update, marked in gray in Figure 3.

4 Columbo Analytical Model for Kubernetes' Workload Deployment Pipeline

To address RQ1, we analyze the Kubernetes workload deployment process to find an architecture to capture the process in. We find this process to resemble a pipeline, with each pipeline stage using a control loop that reads an API object from the datastore and writes one or more new or updated objects back. We define the deployment process as a pipeline by mapping it onto the AtLarge

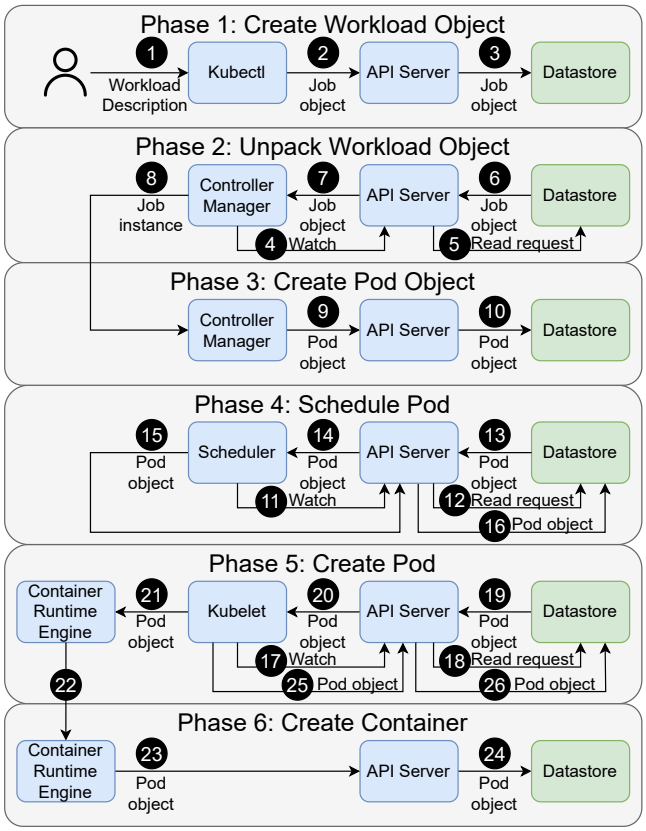

Figure 4: Workload-deployment pipeline in Kubernetes, divided into 6 distinct stages, with a total of 26 steps.

reference architecture for datacenter scheduling [1]. This architecture provides a comprehensive and detailed conceptual model for distributed scheduling with a resource management and deployment pipeline. It provides a blueprint of a pipeline architecture, how to define stages and functional relationships among them, and shows how to apply such model for exemplary systems like Borg [30] (predecessor of Kubernetes). We adapt this model to Kubernetes with minor modification, certifying its comprehensiveness. Working without such reference architecture requires an expensive design and validation process.

The reference architecture defines four major responsibilities: job processing, task processing, scheduler management, and resource management. We map these four responsibilities onto a new workload deployment pipeline of 6 stages (Figure 4 and Table 3) and adapt it to Kubernetes' major workload objects, being jobs, pods, and containers. Each stage processes a single object on a single architectural component. We extend the reference architecture's 4 responsibilities to 6 stages by differentiating between job processing at the user (kubectl) and control plane (controller manager) and splitting resource management into pod and container management. Furthermore, we rename task processing to pod processing, as these are Kubernetes' units of scheduling. The 6 stages are divided into 26 steps, with each step describing a communication between two architectural components. The API server and datastore are used across stages to read and write workload and cluster data. The resulting pipeline describes in detail how a submitted workload gets processed by the control plane and deployed onto worker nodes. Though the pipeline definition requires detailed analysis from an expert, we argue that this is one-off effort as the underlying architecture of Kubernetes has been stable since its inception. We demonstrate the generalizability of the pipeline through experiments in Section 6.6.

Table 4: Overview of model parameters.

Symbol	Definition	Phase
T_{sk}	Time to start kubectl	1
T_{tr}	Time to translate workload	1
L	Latency from user to API server	1
T_w	Time to write object to datastore	1,3-6
T_r	Time to read object from datastore	2,4-6
T_{cpo}	Time to create a pod object	3
T_{sc}	Time to schedule a pod location	4
T_{cg}	Time to create cgroup	5
T_{nn}	Time to create network namespace	5
T_{mv}	Time to mount volume to pod	5
T_{cs}	Time to create pod sandbox	5
C	Number of containers in pod	6
$T_{cc,i}$	Time to create container i	6
$T_{sc,i}$	Time to start container i	6

4.1 Columbo Analytical Model Overview

We synthesize a *first-order analytical model per Kubernetes pipeline-stage*, which, taken together, describes the best-case deployment time of a workload (Section 4.8). This best-case prediction assumes each stage can process objects in parallel without any dependency between objects, following the design of control loops, and parallelism is only limited by available hardware, not Kubernetes configurations. Based on this definition of a best-case execution, Columbo only considers configurations that manage concurrent object processing. The model uses a simple multi-core algorithm to schedule stages onto hardware, which assumes an executed stage to use a single CPU core. Columbo compares the model's best-case predictions against real-world measurements from a benchmark suite to detect performance bottlenecks in Section 6, which also functions as validation of the model. The remainder of this section presents the analytical model per stage, focusing on details pertaining to parallelism management within each stage (parameters in Table 4).

4.2 Stage S1: Create Workload Object (CWO)

Users submit workload description files to kubectl (❶ in Figure 4 and T_{sk} in Equation 1). Kubectl translates the workload objects in the files (T_{tr}), such as jobs, and requests the API server to create the objects (❷ and L). To protect control plane components from flooding the API server with requests, the rate with which components can send requests to the API server can be configured in terms of *queries-per-second (QPS)*. Requests that would exceed the QPS are withheld, slowing down the deployment process. The API server verifies each request, creates the requested objects, and writes them to the datastore (❸ and T_w). These write requests can also be rate-limited to protect the datastore.

$$T_{S1} = T_{sk} + T_{tr} + L + T_w \quad (1)$$

4.3 Stage S2: Unpack Workload Object (UWO)

The controller manager watches the datastore for newly created workload objects (e.g., jobs) and forwards them to the corresponding controller (e.g., a job controller). These watches go through the API server, which verifies the operation (❹–❼ and T_r in Equation 2).

$$T_{S2} = T_r \tag{2}$$

4.4 Stage S3: Create Pod Object (CPO)

Workload objects contain pods, which the controller unpacks, creates (❽ and T_{cpo} in Equation 3), and writes to the datastore (T_w and ❾–❿). Pods are unpacked in batches, and the remaining pods must wait for the current batch to be processed, delaying deployment. The batch size is configurable in source code only. The writes to the datastore go via the API server and are subject to the API server's concurrency limit, previously mentioned in stage 1, forming a scalability bottleneck when creating many pods at once.

$$T_{S3} = T_{cpo} + T_w \tag{3}$$

4.5 Stage S4: Schedule Pod (SP)

The scheduler watches the datastore for newly created pod objects and adds them to a scheduling queue after verifying their eligibility for scheduling (⓫–⓮ and T_r in Equation 4). The scheduler processes the pods in its queue, for each gathering the current state of all worker nodes and filtering them based on user permissions and application specifications (e.g., requires a GPU). The best-fitting node is selected out of this list (T_{sc}) and written back to the datastore as the pod's scheduled target (⓯–⓰ and T_w).

$$T_{S4} = T_r + T_{sc} + T_w \tag{4}$$

The scheduler is heavily reliant on the API server to read pod objects and node statuses and write back scheduling decisions. However, the cluster's state can be cached at the scheduler, reducing the number of read requests. The sequential processing of pods in the queue might create a performance bottleneck as well, which is difficult to mitigate without major changes to Kubernetes' scheduling design. However, the scheduling time per pod is often a fraction of the time required to create the pod and containers unless heavy, possibly AI-enhanced scheduling algorithms are used.

4.6 Stage S5: Create Pod (CP)

Each worker node runs one kubelet that watches the datastore for pod objects that are scheduled onto its node (⓱). The kubelet reads the pod object (⓲–⓴ and T_r in Equation 5) and asks the container runtime to create the pod. This includes using cgroups to restrict the resources the pod and the containers in the pod can use (T_{cg}) and setting up network (T_{nn}) and storage namespaces. The latter comprises inventorying what volumes the pod requires, such as container images, comparing them to the volumes already available on the node, fetching the missing volumes, and mounting them to the pod's file system (T_{mv}). Finally, the kubelet sets up a sandbox that integrates the cgroups and network/storage namespaces for the containers to live in (㉑ and T_{cs}).

$$T_{S5} = T_r + T_{cg} + T_{nn} + T_{mv} + T_{cs} \tag{5}$$

Fetching volumes is an important bottleneck when creating pods, as a single image can be hundreds of megabytes in size or more, which can take minutes to download. Additionally, many of the pod and container creation steps are CPU and memory-demanding. This becomes an even more important problem as the kubelet, container runtime, and operating system processes on worker nodes have to compete for resources with each other and deployed applications.

4.7 Stage S6: Create Container (CC)

Finally, containers are started in the pod (㉒). Pods can contain multiple types of containers, such as ephemeral containers, which are used to inspect other containers; init containers, which are guaranteed to execute only once at pod startup; and application containers with the user's workload. All containers (C in Equation 6) inside a pod are created in sequence (T_{cc}), and then started in sequence (T_{sc}). This is not configurable. However, containers from different pods can be started in parallel.

$$T_{S6} = \sum_{i=0}^{C}(T_{cc,i}) + \sum_{j=0}^{C}(T_{sc,j} + T_w) \tag{6}$$

Status updates are sent to the datastore for every started container, and once all containers in a pod have started (㉓–㉖ and T_w). These status updates can be rate-limited by the API server.

4.8 Deployment Time Prediction

Columbo deploys a single container of a user's workload and measures the time spent in each deployment step. This data is used to fill the model's pipeline step parameters (e.g., T_{cpo}, T_{cg}). Then, Columbo calculates how much each pipeline stage is executed using workload properties such as the number of jobs J, pods per job P, and containers per pod C. Finally, mimicking operating system schedulers, Columbo uses a multi-core scheduling algorithm to schedule the pipeline step executions onto the available hardware, using the user's defined number of worker nodes in use N (we assume a single control plane node), and CPU cores on the control plane U_c and per worker node U_w.

$$\begin{aligned} T = \quad & (T_{S1} + T_{S2}) \times \lceil J/U_c \rceil \\ & + (T_{S3} + T_{S4}) \times \lceil (J \times P)/U_c \rceil \\ & + (T_{S5} + T_{S6}) \times \lceil (J \times P)/(N \times U_w) \rceil \end{aligned} \tag{7}$$

We demonstrate the use of the model in practice in Section 6.3. There, we also measure the accuracy of the model by comparing its predicted optimal deployment time to Columbo-optimized performance of various deployments. The closer these are, the better the model captures Kubernetes' workload deployment pipeline.

5 Rule-based Configuration Space Pruning

Once the workload pipeline is defined by an expert, it can be applied to study the role of configuration parameters in the performance of Kubernetes (RQ2). We do this by mapping the configuration space to model stages and generate rules on how to navigate the space based on Columbo's bottleneck detection. In this section, we present Columbo's methodology for finding parameters and parameter values to update to resolve bottlenecks.

Table 5: Selection of Columbo optimization rules to resolve common Kubernetes workload deployment bottlenecks with default parameter values for Kubernetes version 1.27.

Step	Stage	Parameter	Object	Value
T_w	1,3,6	max-mut.-requests-inflight	*#containers*	200
T_{cpo}	3	SlowStartInitialBatchSize	*#pods*	1
T_w	3	kube-api-qps	*#pods*	20
T_w	4	kube-api-qps	*#pods*	50
T_{mv}	5	registryPullQPS	*#containers*	5
T_{mv}	5	maxParallelImagePulls	*#containers*	1
T_w	6	eventRecordQPS	*#containers*	50
T_w	6	kubeAPIQPS	*#containers*	50

5.1 Configuration Space Pruning

Kubernetes' configuration space includes many parameters that do not affect performance, such as the cluster's name or security credentials, and should not be considered when building parameter rules. Columbo prunes these parameters using a one-time, expert-driven construction of inclusion and exclusion criteria. With limited criteria, Columbo finds only a small fraction of performance-sensitive parameters, which it can build rules for.

These criteria are as follows:

(1) We exclude experimental and deprecated API features.
(2) We exclude parameters that do not affect deployment performance or primarily affect other properties (e.g., security).
(3) We only include numerical parameters. Columbo applies a rate-limiting throughput analysis (Section 3.1) to reason about parameter updates, and rates are numerical by nature. Non-numerical parameters typically enable, disable, or set functionalities, such as encryption or scheduling algorithms, and don't have such a performance intuition. We will include non-numerical parameters in future work.

Columbo uses these criteria to programmatically filter the configuration documentation of Kubernetes version 1.27, and finds 61 parameters that satisfy. These criteria are version independent. Columbo can use the same criteria to find 59 parameters for Kubernetes version 1.26. Columbo also applies these criteria to filter variable declarations in Kubernetes' source code and finds 62 configuration parameters for Kubernetes version 1.27 that satisfy and are not exposed through the Kubernetes API. Columbo's static text-based filtering is not guaranteed to find only parameter that affect performance. We use a one-time manual pruning to reduce the selection of parameters to 42 for Kubernetes 1.27, which will have configuration rules built. Moreover, expert users can continuously update the parameter selection by analyzing the effectiveness of configuration recommendations produced by Columbo. For future work, we plan on investigating more elaborate semantic search and machine learning approaches.

5.2 Configuration Rule Construction

Columbo automatically builds a parameter optimization rule for each of the 42 selected parameters using information from the static text-based filtering of Kubernetes' configuration documentation. A rule defines the pipeline stage and step a parameter maps to, the workload object the parameter affects, and the current value of the parameter. We show a subset of the rules in Table 5. Columbo uses the aforementioned rate-limiting throughput analysis to recommend configuration updates through parameter rules. For example, if Columbo finds a bottleneck in stage 3, which processes pods, and the user's workload includes 10 pods, Columbo finds the parameter rule for the *SlowStartInitialBatchSize* parameter to limit performance as it only allows for 1 pod to be processed at a time. As such, Columbo advices the user to update this parameter to a value of 10.

6 Columbo Evaluation

We construct a benchmark suite of various Kubernetes workload deployment patterns and apply Columbo to the suite to evaluate its ability to detect and resolve bottlenecks. Specifically, we answer the following questions with Columbo:

Q1. How does Kubernetes' workload deployment time scale when deploying containerized applications concurrently? Columbo deploys workloads with 1 and 100 containers and finds a 20x difference in deployment time and a 2.8x difference for the fastest deployed container (Section 6.2).

Q2. How do users use Columbo to optimize their workload's deployment performance? We present a use-case of how Columbo reduces the deployment time of a particular workload by 79.1% over three iterations, with minimal user input.

Q3. Does the deployment method affect deployment time and the performance optimization? Columbo deploys a workload with 100 containers with containers partitioned between multiple workload description files, jobs, pods, and containers per pod (Section 6.4). It finds a performance difference of up to 58% between deployment methods and decreases deployment time between 5% and 37%, limited by CPU usage.

Q4. How does workload deployment time scale with multiple worker nodes using the Kubernetes default and Columbo-optimized configuration? Columbo applies weak and strong scalability, deploying workloads with 100 containers per node and across all nodes (Section 6.5). Columbo achieves an average reduction in deployment time of 27.6% across all 14 microbenchmarks, and up to 79.1%.

Q5. What is the performance difference between Kubernetes versions? Columbo finds Kubernetes version 1.26 to deploy 100 containers 1.4x slower than version 1.27 due to a difference in default configuration settings (Section 6.6). Columbo eliminates this difference through configuration updates.

6.1 Benchmark Design and Setup

Columbo creates a Kubernetes cluster with a default or user-defined configuration (Figure 3). On the cluster, it deploys a workload from the benchmark suite. A benchmark describes, among others, application parameters, the container image to use, how many jobs, pods, and containers to use, and the number of worker nodes in the cluster. To achieve this, Columbo uses Continuum [12], an infrastructure deployment and benchmarking framework.

The benchmark suite covers several common workload deployment patterns across 14 microbenchmarks: Scaling containers on a single node (Q1) or across multiple nodes (Q4) using multiple deployment methods (Q3). These patterns cover the primary ways

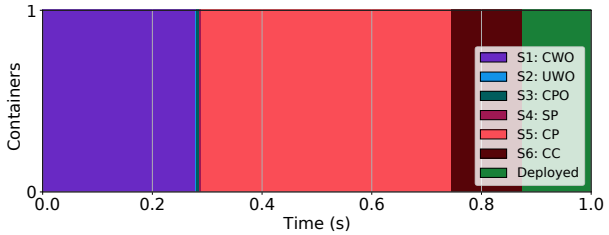

Figure 5: Time to deploy 1 container on Kubernetes, split between the 6 workload deployment pipeline stages (S).

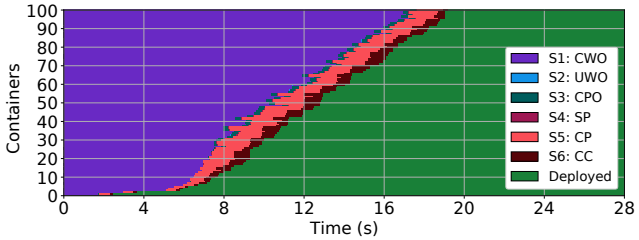

Figure 6: Time to deploy 100 containers on Kubernetes.

users can deploy and scale up workloads and thus show the primary scalability limitations users can expect when scaling up. We use a standard Python 3 container image (114 MB in size) as the application container and cache the image on all worker nodes to eliminate downloading overhead. Unless stated otherwise, Columbo deploys Kubernetes version 1.27 (Q5) with its default configuration on *e2-highmem-8* virtual machines from Google Cloud with 8 CPU cores and 64 GB RAM. It uses one machine to host Kubernetes' control plane and up to 16 machines as worker nodes. Columbo uses a customized Kubernetes implementation to measure the time spent in each pipeline step (*deployment data* in Figure 3). We repeat all experiments 5 times and report the best run.

6.2 Concurrent Deployments

To answer Q1, Columbo deploys a single container (Figure 5) and 100 containers on a single node (Figure 6), and measures their deployment time per pipeline stage. It takes Kubernetes 0.94 seconds to deploy the single container and 19.2 seconds for the 100 containers, a 20x difference. Most interestingly, it takes Kubernetes 2.5 seconds to deploy the first of the 100 containers. These differences indicate a limitation in Kubernetes' workload deployment pipeline to process workload requests in parallel. Columbo shows the origin of the limitation to be kubectl in stage 1, which takes up as much as 90% of deployment time for the last deployed container (Figure 6). Columbo finds the CPU of the control plane, where kubectl is executed, to be fully utilized during the stage 1 execution (Figure 7). Hence, Columbo concludes that kubectl is resource-constrained and, therefore, slows down stage 1. We demonstrate in Section 6.5 how Columbo optimizes such a performance bottleneck.

Finding 1: Kubernetes takes almost one second to deploy a single container, which is 8.5x slower than containerd (Figure 1).

Finding 2: Containers deploy significantly slower in parallel, with the fastest deployed container of a 100-container workload being 2.6x slower than a single-container workload.

Finding 3: Kubectl uses significant CPU resources to translate and submit container descriptions, which could cause a resource bottleneck and slow down container deployment.

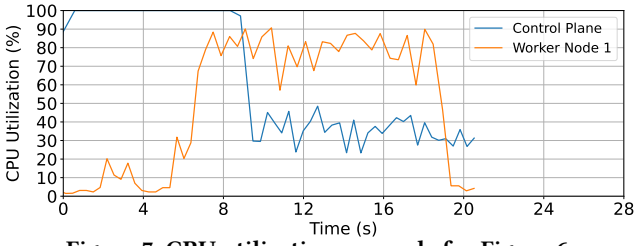

Figure 7: CPU utilization per node for Figure 6.

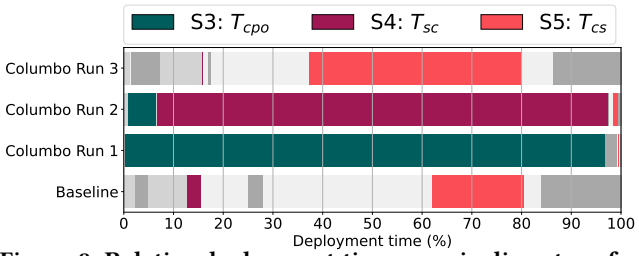

Figure 8: Relative deployment time per pipeline stage for a workload of 1600 containers across 16 nodes. Columbo analyzes the deployment in three iterations, finding and resolving bottlenecks in specific pipeline stages through configuration updates. The total deployment time reduces from 80.9 seconds (run 1) to 16.9 seconds (run 3), converging to a predicted best-case time of 11.9 seconds (baseline).

6.3 Day-to-day Columbo Use Case

To answer Q2, we demonstrate how Columbo deploys 1600 containers over 16 nodes and improves Kubernetes' default configuration in three iterations to reduce deployment time by 79.1%. We plot Columbo's measured relative deployment time per pipeline step in Figure 8 as *Columbo Run 1*, with the sum of all steps being 100%.

Columbo uses the analytical model to predict the best-case deployment time for this workload. It deploys a single container from the workload and measures the time spent in each pipeline step. This information and the workload specification are fed into the analytical model, resulting in a best-case deployment prediction (Section 4.8). We show this prediction as *Baseline* in Figure 8.

Columbo selects the pipeline step with the most discrepancy in execution time between the real-world execution (Columbo Run 1) and the analytical prediction (Baseline) as primary bottleneck. In Figure 8, this is the step that creates pod objects (T_{cpo}), with a predicted execution time share of 1%, to an empirical share of 97%. The workload requires 1600 pod objects to be created. Columbo finds the parameter rule of the *SlowStartInitialBatchSize* configuration parameter to map to the T_{cpo} step and violate the required parallelism of 1600, with a default value of 1. Therefore, Columbo recommends the user to change this parameter to 1600, with a predicted deployment time reduction of 96%.

The user applies the parameter update and runs Columbo for a second iteration. Columbo benchmarks the workload with the new configuration, which we show as *Columbo Run 2* in Figure 8. Deployment time decreases from 80.9 seconds in the first iteration to 31.1 in the second, and the primary bottleneck in T_{cpo} reduces from 97% of execution time to 6%, showing the effectiveness of Columbo's parameter tuning. Columbo identifies a new bottleneck and violating parameter rule, similar to the first iteration, which

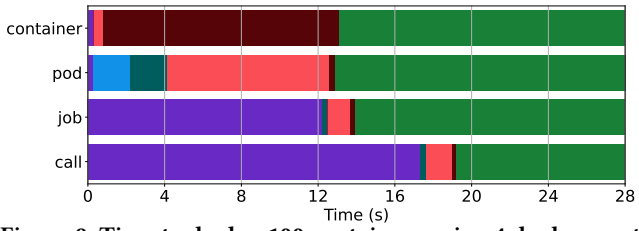

Figure 9: Time to deploy 100 containers using 4 deployment methods: (i) *call*, distributed over 100 kubectl calls; (ii) *job*, one kubectl call with 100 jobs; (iii) *pod*, a single job with 100 pods; and (iv) *container*, a single pod with 100 containers.

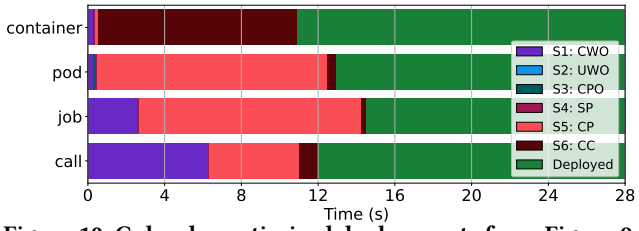

Figure 10: Columbo-optimized deployments from Figure 9.

reduces deployment time to 16.9 seconds (*Columbo Run 3*), a 79.1% decrease compared to the original configuration. While this deployment is still 42% above the predicted best-case performance of 11.9 seconds, it will be difficult in practice to improve the performance further. Columbo measures a CPU utilization of up to 100% in the control plane and up to 90% in the worker nodes. Given this high utilization, there is little more performance to be gained, so the user decides to stop tuning with Columbo. We ascribe the difference between the actual and predicted performance to the model's multi-core scheduling algorithm. It assumes that each pipeline step execution uses one CPU core. However, in practice, many steps use more resources. As a result, the actual execution is sooner limited by CPU resources rather than configuration parameters, leading to reduced performance. A more fine-grained scheduling algorithm would change the predicted resource use per pipeline stage and hence change the predicted performance per stage under resource constraints. However, we find that bottlenecks typically dominate deployment time (e.g., up to 96% in Figure 8) so we do not expect a more detailed scheduling algorithm to change Columbo's bottleneck detection.

6.4 Deployment Methods

To investigate how a workload's structure affects deployment time (Q2), we compare four deployment patterns. Each pattern deploys 100 containers and wraps them differently in pods and jobs. These patterns are: *per-call*, invoke kubectl 100 times, with 1 job per invocation; *per-job*, deploy 100 jobs through a single kubectl call; *per-pod*, deploy 100 pods in a single job; and *per-container*, deploy 100 containers in a single pod. We present our results in Figure 9.

Columbo shows that the per-call method is, on average 47% slower in deploying all 100 containers than the other deployment methods. This deployment, as well as the per-job deployment, seem to be slowed down due to a lack of CPU resources in phase 1, as discussed in finding 3. However, Columbo attributes this bottleneck to a limited job submission rate between kubectl and the API server, which it resolves via a configuration update (Figure 10). Despite

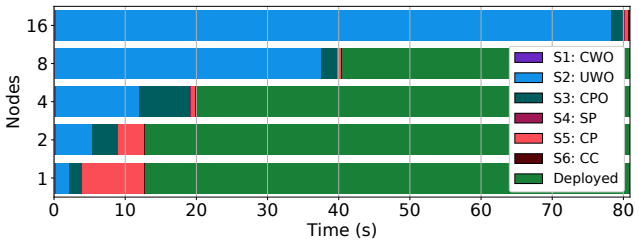

Figure 11: Time to deploy 100 containers per node with the per-pod deployment method (weak scalability).

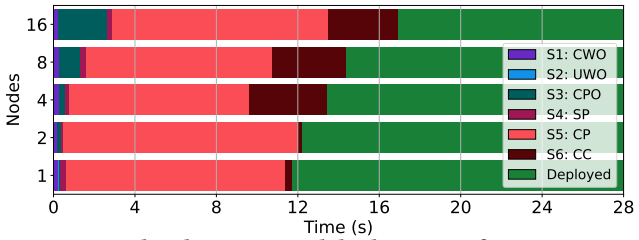

Figure 12: Columbo-optimized deployments from Figure 11.

the high CPU utilization, both deployments see a sharp reduction in time spent in stage 1 of the deployment pipeline, and the per-call deployment even sees a 37% reduction in total deployment time.

The per-container and per-pod methods have the lowest deployment times of around 13 seconds and spend the deployment time primarily on the worker nodes. The per-container method sees a decrease in deployment time of 16% but is still bottlenecked in the container creation phase. This deployment's 100 containers are created in sequence, which is the intended behavior for containers in the same pod and cannot be changed. The per-job method sees no change in deployment time as it is always constrained by CPU resources on the worker node.

Finding 4: Workloads with fewer jobs and more containers per job decrease kubectl's CPU usage, and potentially deployment time.

Finding 5: Pods are expensive to create. However, grouping containers in a pod is more expensive as these are created sequentially.

6.5 Multi-node Deployments

To answer Q4, we examine how deployment time scales with workloads distributed across worker nodes. We study weak scalability, where Columbo deploys 100 containers per node for up to 1600 containers on 16 nodes (Figure 11), and strong scalability, where Columbo deploys 100 containers across up to 16 nodes (Figure 13). Columbo uses the per-pod deployment method.

With weak scalability, the load on the control plane increases as there are more containers to process, but the load on worker nodes remains constant as each always deploys 100 containers. As a result, Columbo shows that stage 2 on the control plane is a primary bottleneck and increases deployment time from 12.9 seconds with 1 worker to 80.9 seconds with 16 workers, or a 6.3x increase. Columbo finds this bottleneck to be because of configuration limitations rather than resource limitations, and reduces deployment time to 16.9 seconds with 16 nodes, a 79.1% decrease (Figure 12).

With strong scalability (Figure 13), the load on the control plane remains constant as it always processes 100 containers. In contrast, the load on the worker nodes decreases as each deploys fewer containers. As a result, workload deployment time will stop decreasing

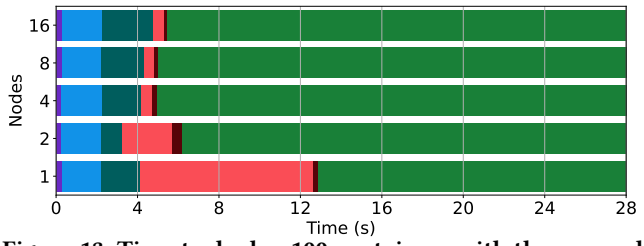

Figure 13: Time to deploy 100 containers with the per-pod deployment method across 1 to 16 nodes (strong scalability).

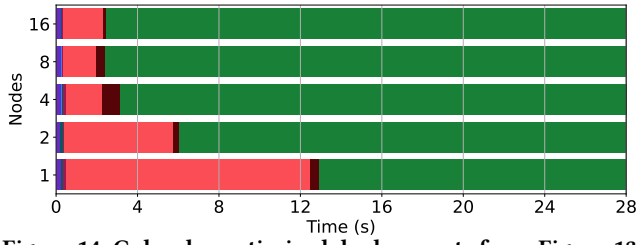

Figure 14: Columbo-optimized deployments from Figure 13.

when using more than 4 workers as the load on the workers is already negligible. Columbo reduces the control plane deployment time via several configuration updates (Figure 14). As a result, deployment time decreases from 5.5 seconds to 2.5 seconds with 16 workers, while performance still stagnates from 4 workers on.

Finding 6: Deploying containers across more worker nodes decreases deployment time. There is a limit, however, as larger workloads increase the pressure on the control plane.

Finding 7: Columbo achieves an average decrease in deployment time of 27.6% across its benchmark suite of 14 microbenchmarks compared to the default configuration. It resolves configuration-induced bottlenecks and makes the CPU the new limiting factor.

6.6 Kubernetes Versions

To answer Q5, Columbo deploys the same 100 container workload in Kubernetes versions 1.26 (Figure 15) and 1.27 (Figure 6) to investigate performance differences. Although Kubernetes is an evolving platform, its high-level architecture has remained stable, allowing Columbo to seamlessly work across Kubernetes versions. Columbo finds that version 1.26 deploys all containers in 26.5 seconds, a 1.4x increase compared to the 19.2 seconds of version 1.27. Furthermore, the first deployed container in version 1.27 takes 2.6 seconds compared to 4.6 seconds for version 1.26, a 1.8x increase. The primary bottleneck is identified in the pod deployment stage 5. We discover a difference in default values between the Kubernetes versions for several configuration parameters related to stage 5, including a 10-fold difference in the number of read and write requests a kubelet may submit to the API server per second, to be the root cause of the performance difference. These parameters slow down pod creation at different rates. Columbo resolves the bottleneck in both versions, resulting in a similar (less than 5% difference in deployment time) optimized performance between the versions, which is visualized in Figure 10 for the call method in Kubernetes 1.27.

Finding 8: Kubernetes versions vary in default configurations, which can result in a 77% increased workload deployment time for version 1.26 compared to 1.27. Columbo's recommendations protect users against such variations.

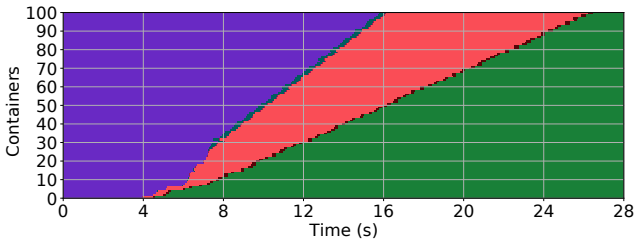

Figure 15: Time to deploy 100 containers on Kubernetes 1.26.

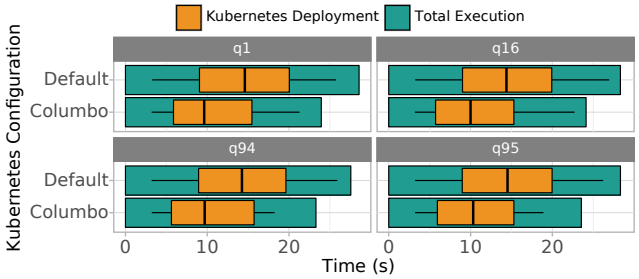

Figure 16: Time to deploy 400 Spark workers as pods on Kubernetes and execute TPC-DS queries q1, q16, q94, and q95, using the default and Columbo-optimized configurations.

7 End-to-end Impact of Columbo

We evaluate the performance impact of Columbo on a data analytics application from the TPC-DS [26] benchmark using Apache Spark and a serverless workflow on OpenWhisk. Both use Kubernetes.

7.1 TPC-DS on Spark

Apache Spark is a data processing systems that process data on workers through user queries. Spark can deploy these workers through resource managers such as Kubernetes. Columbo deploys Spark on Kubernetes and measures the time it takes to deploy the workers and its impact on end-to-end application performance when switching between Kubernetes's default and Columbo's optimized configuration.

Setup: Columbo deploys Kubernetes on 15 *e2-standard-32* virtual machines from Google Cloud with 32 CPU cores and 128GB RAM. Two machines run the Kubernetes and Spark control planes, and the remaining 13 function as workers. We configure Spark to deploy 400 workers across the 13 worker nodes, each using 1 CPU core and 3GB RAM. We use Spark 3.4.1 with the Kubernetes Pod API to start workers. For the workload, we use a 100GB TPC-DS dataset stored on Google Cloud Storage. Out of the 100 queries that the TPC-DS benchmark offers, we use queries 1, 16, 94, and 95, as these represent different compute and network characteristics [27].

Evaluation: Columbo executes one query from the TPC-DS benchmark at a time, restarting all 400 workers for each (cold start). We show the total execution time for each query in Figure 16. The total execution time consists of three parts: First, Spark prepares to deploy the user workload and requests Kubernetes to create workers. Second, Kubernetes creates the Spark workers as pods (marked as *Kubernetes Deployment* in Figure 16). The figure also shows the distribution of pod deployment times. Third, a worker executes the query as soon as it is deployed by Kubernetes.

We observe that the median worker deployment time on Kubernetes decreases from 14.5 seconds using Kubernetes' default

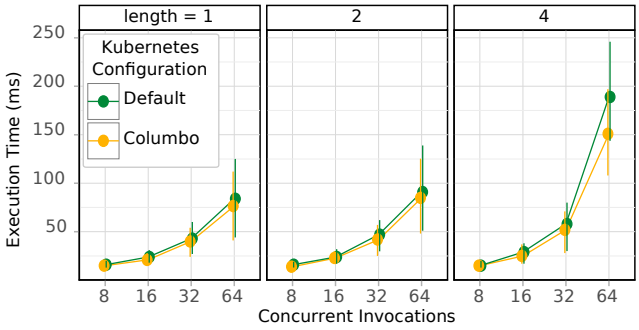

Figure 17: Execution time of serverless functions across different function chain lengths and concurrent executions.

configuration to 10.3 seconds after applying Columbo. This average decrease of 29% aligns with the 28% decrease found across Columbo's microbenchmarks (finding 7), showing the generalizability of Columbo's recommendations. The reduced deployment time translates into a 17% lower total execution time on average.

7.2 Serverless Workflows on OpenWhisk

Serverless platforms such as OpenWhisk [2] and Knative [15] deploy serverless functions using Kubernetes. Serverless workflows are complex applications that chain serverless functions so the results of one function are passed to the next. Many scientific [3] and business [24] applications are structured as serverless workflows. Columbo deploys a serverless microbenchmark using OpenWhisk on Kubernetes. OpenWhisk starts workers using Kubernetes before executing functions on the workers.

Setup: Columbo deploys Kubernetes on 10 *e2-standard-8* virtual machines from Google Cloud with 8 CPU cores and 32GB RAM. One machine runs the Kubernetes control plane, another the OpenWhisk control plane, and the remaining 8 function as workers. The microbenchmark sends a JSON object through a chain of 1 to 4 functions. Each chain is evaluated with 8 to 64 concurrent invocations. This microbenchmark is representative of common function depths found in chained workloads [3, 20], and is similar to serverlessbench [33]. We measure the cold-start performance as warm-starts do not interact with the Kubernetes control plane.

Evaluation: In Figure 17, we observe that optimizing Kubernetes' configuration has minimal impact at lower concurrency levels. At higher concurrency levels (32-64) and low chain lengths (1-2), Columbo-optimized configurations improve performance by 10-13% as the number of deployed OpenWhisk workers exceeds the default concurrency of several Kubernetes pipeline stages. Therefore, as expected, Columbo improves deployment performance further by up to 20% at even higher concurrency levels and chain lengths. Optimizing the Kubernetes configuration has no significant impact on performance variability.

Finding 9: Columbo's optimization strategy also applies to systems that rely on Kubernetes to manage resources, achieving an average workload deployment time reduction of 29% for Spark on Kubernetes.

Finding 10: Columbo's optimization of workload deployment time lowers total execution time for highly parallel applications on Spark by 17% on average and for deep serverless function chains on OpenWhisk by up to 20%.

8 Related Work

Many works focus on Kubernetes to study, benchmark, and analyze its performance and components. Chiba et al. present ConfAdvisor, a tool for tuning container-specific configurations for Kubernetes [4]. However, the container runtime is one of many components in Kubernetes' architecture, so optimizing it has limited impact [6]. Truyen et al. analyze the differences in Kubernetes' configuration between managed Kubernetes services [29]. Medel et al. model the lifecycle of a pod and build a tool for detecting performance bottlenecks in Kubernetes clusters [11]. Khan et al. integrate performance models into a simulator, allowing them to quickly iterate between Kubernetes configurations [14]. However, these models operate with higher granularity than Columbo, making it challenging to identify the root cause of discovered bottlenecks. Geldenhuys et al. optimize configurations of distributed stream processing platforms such as Apache Flink [10], which can be deployed on Kubernetes and be used on top of Columbo.

Closest to our approach are SelfTune, BestConfig, and SmartConf. SelfTune [13] leverages the iterative operation of cluster managers, such as Kubernetes, to update parameters. BestConfig [34] uses a divide-and-conquer method for configuration-space sampling, with broad coverage (bounded by resource utilization) to explore an optimal configuration for systems (not just cluster manager). SmartConf [32] proposes a control-theoretic approach to navigate toward user-defined performance goals using dynamic configuration adjustments. Compared to these approaches, Columbo does not require users to define relevant parameters, their bounds, and reward functions, provides finer-grained bottleneck analysis (Kubernetes stages), identifies relevant configuration parameters with each run (rule-based), and recommends parameter updates to optimize workload deployment time.

Beyond the work described here, there is a large body of work around distributed performance anomaly detection, resource utilization analysis, and (failure/bottleneck) root cause analysis [21, 25]. However, these works focus on general modeling and performance, bottleneck, and failure analysis, without attributing the cause to a configuration option as Columbo does. More broadly, there is a class of work that targets scaling and optimizations of various distributed workloads and services [23, 28]. We consider these efforts orthogonal to ours, which aim to extract the maximum performance from configuration change only.

9 Conclusion

In this paper, we introduced Columbo, a reasoning framework to optimize application deployment time in Kubernetes through configuration updates. We proposed a highly automated and detailed model-based and rule-driven performance reasoning framework that can be used with a benchmark suite to detect and resolve performance bottlenecked stages in Kubernetes' workload deployment pipeline. Columbo finds performance bottlenecks across the entire pipeline because of Kubernetes' default configuration and achieves an average decrease in deployment time of 28% across 14 microbenchmarks. Moreover, it achieves a 17% reduction in total execution time for data processing with Apache Spark and a 20% reduction for serverless workflows with OpenWhisk. Columbo is available at https://github.com/atlarge-research/continuum/tree/columbo.

Acknowledgments

We want to thank the shepherd for their expert guidance, as well as the anonymous reviewers for their detailed feedback. This work was supported by the EU Horizon Graph Massivizer and the EU MSCA Cloudstars projects. This research was partly supported by a National Growth Fund through the Dutch 6G flagship project "Future Network Services" and NWO TOP project OffSense.

References

[1] Georgios Andreadis, Laurens Versluis, Fabian Mastenbroek, and Alexandru Iosup. 2018. A reference architecture for datacenter scheduling: design, validation, and experiments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE / ACM, 37:1–37:15. http://dl.acm.org/citation.cfm?id=3291706

[2] Apache. 2023. OpenWhisk. https://openwhisk.apache.org/. Accessed: 2023-05-23.

[3] Ryan Chard, Yadu N. Babuji, Zhuozhao Li, Tyler J. Skluzacek, Anna Woodard, Ben Blaiszik, Ian T. Foster, and Kyle Chard. 2020. funcX: A Federated Function Serving Fabric for Science. In *HPDC '20: The 29th International Symposium on High-Performance Parallel and Distributed Computing, Stockholm, Sweden, June 23-26, 2020*, Manish Parashar, Vladimir Vlassov, David E. Irwin, and Kathryn M. Mohror (Eds.). ACM, 65–76. https://doi.org/10.1145/3369583.3392683

[4] Tatsuhiro Chiba, Rina Nakazawa, Hiroshi Horii, Sahil Suneja, and Seetharami Seelam. 2019. ConfAdvisor: A Performance-centric Configuration Tuning Framework for Containers on Kubernetes. In *IEEE International Conference on Cloud Engineering, IC2E 2019, Prague, Czech Republic, June 24-27, 2019*. IEEE, 168–178. https://doi.org/10.1109/IC2E.2019.00031

[5] CNCF. 2022. CNCF 2022 Anual Survey. https://www.cncf.io/reports/cncf-annual-survey-2022/. Accessed: 2024-01-19.

[6] Greg Cusack, Maziyar Nazari, Sepideh Goodarzy, Erika Hunhoff, Prerit Oberai, Eric Keller, Eric Rozner, and Richard Han. 2022. Escra: Event-driven, Sub-second Container Resource Allocation. In *42nd IEEE International Conference on Distributed Computing Systems, ICDCS 2022, Bologna, Italy, July 10-13, 2022*. IEEE, 313–324. https://doi.org/10.1109/ICDCS54860.2022.00038

[7] Renan Delvalle, Gourav Rattihalli, Angel Beltre, Madhusudhan Govindaraju, and Michael J. Lewis. 2016. Exploring the Design Space for Optimizations with Apache Aurora and Mesos. In *9th IEEE International Conference on Cloud Computing, CLOUD 2016, San Francisco, CA, USA, June 27 - July 2, 2016*. IEEE Computer Society, 537–544. https://doi.org/10.1109/CLOUD.2016.0077

[8] Alex Ellis. 2022. OpenFaaS - Serverless Functions Made Simple. https://github.com/openfaas/faas. Accessed: 2024-01-19.

[9] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, Aditya Akella and Jon Howell (Eds.). USENIX Association, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[10] Morgan K. Geldenhuys, Dominik Scheinert, Odej Kao, and Lauritz Thamsen. 2024. Demeter: Resource-Efficient Distributed Stream Processing under Dynamic Loads with Multi-Configuration Optimization. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering, ICPE 2024, London, United Kingdom, May 7-11, 2024*, Simonetta Balsamo, William J. Knottenbelt, Cristina L. Abad, and Weiyi Shang (Eds.). ACM, 142–153. https://doi.org/10.1145/3629526.3645048

[11] Víctor Medel Gracia, Rafael Tolosana-Calasanz, José Ángel Bañares, Unai Arronategui, and Omer F. Rana. 2018. Characterising resource management performance in Kubernetes. *Comput. Electr. Eng.* 68 (2018), 286–297. https://doi.org/10.1016/j.compeleceng.2018.03.041

[12] Matthijs Jansen, Linus Wagner, Animesh Trivedi, and Alexandru Iosup. 2023. Continuum: Automate Infrastructure Deployment and Benchmarking in the Compute Continuum. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering, ICPE 2023, Coimbra, Portugal, April 15-19, 2023*, Marco Vieira, Valeria Cardellini, Antinisca Di Marco, and Petr Tuma (Eds.). ACM, 181–188. https://doi.org/10.1145/3578245.3584936

[13] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. 2023. SelfTune: Tuning Cluster Managers. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 1097–1114. https://www.usenix.org/conference/nsdi23/presentation/karthikeyan

[14] Michel Gokan Khan, Javid Taheri, Auday Al-Dulaimy, and Andreas Kassler. 2023. PerfSim: A Performance Simulator for Cloud Native Microservice Chains. *IEEE Trans. Cloud Comput.* 11, 2 (2023), 1395–1413. https://doi.org/10.1109/TCC.2021.3135757

[15] Knative. 2024. Knative. https://knative.dev/. Accessed: 2024-05-24.

[16] Kubernetes. 2023. Kubernetes. https://github.com/kubernetes/kubernetes. Accessed: 2024-01-19.

[17] Lars Larsson, Harald Gustafsson, Cristian Klein, and Erik Elmroth. 2020. Decentralized Kubernetes Federation Control Plane. In *13th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2020, Leicester, United Kingdom, December 7-10, 2020*. IEEE, 354–359. https://doi.org/10.1109/UCC48980.2020.00056

[18] Lars Larsson, William Tärneberg, Cristian Klein, Erik Elmroth, and Maria Kihl. 2020. Impact of etcd deployment on Kubernetes, Istio, and application performance. *Softw. Pract. Exp.* 50, 10 (2020), 1986–2007. https://doi.org/10.1002/SPE.2885

[19] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 53–68. https://www.usenix.org/conference/atc22/presentation/li-zijun-rund

[20] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, Carlo Curino, Georgia Koutrika, and Ravi Netravali (Eds.). ACM, 412–426. https://doi.org/10.1145/3472883.3487003

[21] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 378–393. https://doi.org/10.1145/2815400.2815415

[22] Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, George Candea (Ed.). USENIX Association. https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry

[23] Themis Melissaris, Kunal Nabar, Rares Radut, Samir Rehmtulla, Arthur Shi, Samartha Chandrashekar, and Ioannis Papapanagiotou. 2022. Elastic cloud services: scaling snowflake's control plane. In *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*, Ada Gavrilovska, Deniz Altinbüken, and Carsten Binnig (Eds.). ACM, 142–157. https://doi.org/10.1145/3542929.3563483

[24] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 115–130. https://doi.org/10.1145/3318464.3389758

[25] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 293–307.

[26] Meikel Pöss, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why You Should Run TPC-DS: A Workload Analysis. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold (Eds.). ACM, 1138–1149. http://www.vldb.org/conf/2007/papers/industrial/p1138-poess.pdf

[27] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[28] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 16, 16 pages. https://doi.org/10.1145/3342195.3387524

[29] Eddy Truyen, Nane Kratzke, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. 2020. Managing Feature Compatibility in Kubernetes: Vendor Comparison and Analysis. *IEEE Access* 8 (2020), 228420–228439. https://doi.org/10.1109/ACCESS.2020.3045768

[30] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, Laurent Réveillère, Tim Harris, and Maurice Herlihy (Eds.). ACM, 18:1–18:17. https://doi.org/10.1145/2741948.

2741964

[31] Guolu Wang, Jungang Xu, and Ben He. 2016. A Novel Method for Tuning Configuration Parameters of Spark Based on Machine Learning. In *18th IEEE International Conference on High Performance Computing and Communications; 14th IEEE International Conference on Smart City; 2nd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2016, Sydney, Australia, December 12-14, 2016*, Jinjun Chen and Laurence T. Yang (Eds.). IEEE Computer Society, 586–593. https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0088

[32] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. 2018. Understanding and Auto-Adjusting Performance-Sensitive Configurations. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 154–168.

[33] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 30–44. https://doi.org/10.1145/3419111.3421280

[34] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 338–350.

Notes: IBM is a trademark of International Business Machines Corporation, registered in many jurisdictions worldwide. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates. Other products and service names might be trademarks of IBM or other companies.