Reducing cold starts during elastic scaling of containers in Kubernetes

Emad Heydari Beni, Eddy Truyen, Bert Lagaisse, Wouter Joosen
{firstname.lastname}@cs.kuleuven.be
imec-DistriNet, KU Leuven, Belgium

Jordy Dieltjens
jordy.dieltjens@gmail.com
KU Leuven, Belgium

ABSTRACT

Automatic scaling of containers is an important feature to handle fluctuating workloads. However, the delay caused by the time to bootstrap a container has an impact on applications with deadline-based Service-Level Objectives (SLOs). This delay is called the cold start problem. Many techniques have already been proposed to tackle this problem but not all techniques have been integrated and evaluated in Kubernetes, the de-facto standard in container orchestration. This paper combines and evaluates three techniques in the context of Kubernetes: (i) pre-creation of network containers, (ii) using container images that enable sharing of linked libraries in memory and (iii) extending the declarative configuration management approach of Kubernetes with imperative configuration for creating multiple application containers in parallel. A prototype of the approach is implemented and tested on a Java-based Spring Boot application where the cold start problem occurs due to various library dependencies. Our findings illustrate that the use of containers that allow for library sharing already has a large, positive impact when starting up a single container. The pre-creation of network containers in combination with imperative configuration enables the application to meet deadline-driven SLOs without a non-deterministic delay that appears in Kubernetes when multiple containers are created in parallel. We conclude that the use of container images that allow for library sharing is a must for all applications that require fast container start-ups in Kubernetes. Pre-creation of network containers when combined with imperative configuration also has a positive impact on SLO compliance during elastic scaling of containers.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; *Cloud computing*; *Software as a service orchestration system*;

KEYWORDS

Cold start, Auto-scaling, Container frameworks, Serverless

ACM Reference Format:
Emad Heydari Beni, Eddy Truyen, Bert Lagaisse, Wouter Joosen and Jordy Dieltjens. 2021. Reducing cold starts during elastic scaling of containers in Kubernetes. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3412841.3441887

1 INTRODUCTION

New cloud computing execution models such as serverless computing have become popular recently [1]. Software providers offer their services through microservices or purely serverless architecture. In that regard, one of the emerging and widely adopted paradigms of delivering software is the use of OS-level virtualization such as containers [20]. To facilitate managing an application as a distributed set of containers, container orchestration frameworks are used. The popular examples are Docker containers and Kubernetes [2]. Kubernetes has pioneered in declarative configuration management, where a control loop detects differences between a desired and actual system state, and a policy-rich scheduler that takes into account expressive placement constraints for placing containers on a cluster of worker nodes.

Automatic scaling of functions is an inherent feature of serverless computing [1, 12]. Upon each function call, one or more containers need to be started or elastically scaled out based on different workloads. Even though containers are considered to have faster startup times compared to traditional virtual machines, the latency caused by the time to (i) bootstrap containers, (ii) prepare the software environments, and (iii) initialize the user code has an impact on some applications [4, 12], especially multi-tenant services with strict service level agreements (SLAs). This problem is called "cold start". It sometimes takes seconds or minutes to have a container and the application up and running.

To reduce this cold start latency, various techniques have been already introduced, e.g. using snapshots [6], lazy fetching of Docker images [8] and container queues [15, 18, 19]. However, there are always trade-offs. In particular, the queue-based approaches mostly sacrifice memory to obtain a faster start-up time since containers are pre-launched. In the context of the serverless middleware OpenWhisk, Mohan et al. [19] improved this deficiency by pre-creating and caching a pool of reusable networking endpoints, namely network containers. When a function container is created, an existing network container gets bound to it. This approach results in 80% reduction in execution time in comparison to cold queues and several orders of magnitude reduction in memory footprint. However, this technique has not yet been evaluated in the context of Kubernetes.

Moreover, in a serverless setting, or auto-scaling in general, software dependencies are redundantly loaded in memory when containers are replicated. In particular, recent research has shown that replicated containers can share common libraries in memory, provided that the used container image encapsulates the libraries in separate image layers. [9]. This technique has also been reported to reduce startup times as well [24].

Contributions. In this paper, we extend Kubernetes with an imperative configuration management approach to reduce the cold start problem when auto-scaling containers in parallel.

We have found that in Kubernetes a non-deterministic delay appears in container start-up times when creating multiple application containers in parallel on the same node. The cause of this delay is due the declarative configuration management approach of Kubernetes where various controllers act upon differences between desired and actual system state.

It has already been shown before that declarative and imperative configuration management are complementary techniques [3]. Our imperative approach uses a script to create multiple containers in parallel. However, it builds upon the technique introduced by Mohan et al. [19] to still benefit from the advantages of Kubernetes' declarative configuration management approach and its policy-rich scheduler. More specifically, to realize a queue of reusable network containers in Kubernetes, we create a pool of empty pods (i.e. pod is a group of containers). Each pod comes with a network container since network containers can only be instantiated in a pod. To scale out the application in an imperative fashion, our scaler selects one or more of these pods and it launches a number of application containers by injecting these into the selected pods.

Our research goals are threefold:

(1) The first goal is to see whether creating a pool of network containers (also called `pause` containers) in advance has a positive effect on the cold start problem in Kubernetes.
(2) The second goal is to investigate the impact of container-based library sharing on the cold start problem when combined with network containers technique in Kubernetes.
(3) The last goal is to compare imperative and declarative configuration management along with the two aforementioned techniques. The imperative approach creates multiple containers by means of an imperative script, whereas the declarative approach employs a control loop with a policy-rich scheduler.

A prototype of the approach is implemented and tested on a Java-based Spring Boot application [21] where the cold start problem occurs. This application is a simple job processing microservice, which has a variable amount of users. The intention is to test the effectiveness of the approach with multiple users submitting jobs at the same time. As a validation, we compare our approach with using the default declarative approach of Kubernetes for increasing or decreasing the number of replicated Pods.

Findings. We have evaluated our approach with fluctuating workloads. We could deduce from the experiments that using container images that support library sharing has the greatest impact on the "cold start" problem. Library sharing also had a positive but small effect on the start-up of multiple containers in parallel. After all, this provides an extra reduction in time for starting up application dependencies (e.g. the Java Virtual Machine (JVM) and Tomcat server). Finally, the effectiveness of the network container queue is limited, but in conjunction with the imperative approach, it results in faster boot time of containers since it can be done in parallel. The imperative approach further enables the application to meet SLA targets without a non-deterministic delay that appears in the declarative approach when multiple Pods must be created at the same time due to concurrent user requests.

The remainder of this paper is structured as follows. In §2 and §3, we present the necessary background and related work on containers and the cold start problem. Then, §4 discusses how reusable network containers, library sharing and imperative configuration can be realized in Kubernetes. Subsequently, §5 presents our evaluation. Finally, §6 presents our conclusions.

2 BACKGROUND

In this section, we introduce: (i) Kubernetes and some of its underpinning components, (ii) the differences between declarative and imperative configuration, and (iii) a brief definition of the cold start problem in the context of containers.

2.1 Kubernetes

Kubernetes is a container orchestration framework commonly deployed and used by researchers and practitioners. It facilitates the deployment, (auto)scaling and management of container-based applications through declarative configuration files. A *pod* is a group of containers that share storage and network resources [13]. Pods are the smallest unit of deployment in Kubernetes. A *deployment* is used to get a pod or a *ReplicaSet* of a pod to a certain state. The *deployment controller* is responsible for changing the current state to the requested state. CPU and memory of compute resources (containers or pods) can be managed by guaranteed *resource requests* and *maximum resource limits.*

`Pause` ***containers.*** Each pod has a `pause` container. A `pause` container is responsible for the creation of a shared network and a namespace for the other containers in the pod [14]. If any container within a pod fails, the entire pod does not restart thanks to `pause` containers. This is because this container ensures that the network namespace and PID namespace remain. We use the terms `pause` and network container interchangeably throughout this paper.

Internal components. In Kubernetes, the control plane, which manages the worker nodes and pods, is composed of schedulers, API servers, and an `etcd` database. Schedulers are responsible for suitable pod placements based on different scheduling decisions. API server exposes the Kubernetes functionalities. The `etcd` database is a consistent and highly-available key-value store as a backing store for all cluster data [13]. The worker nodes host pods. *Kubelet* is an agent on each node responsible for making sure that containers are running and healthy [13].

2.2 Declarative and imperative systems.

Declarative approach. In a declarative system, the client is aware of a desired state, and the system is provided with a representation of this state, through which it can come up with a set of instructions to reach that state from the current state [23]. In Kubernetes, this approach is managed by various controllers. A representation of the pods is sent to the API server. After a few security checks, it stores the resource in the `etcd` database. The scheduler afterwards performs pod placement process based on this information. Based on scheduling decisions, the kubelets are contacted to start the containers. If more pods are planned to be started, that does not happen simultaneously and in parallel.

Imperative approach. In an imperative system, the client is aware of a set of instructions to bring the system to a desired state [23]. Kubernetes operates as a declarative system through the API server. An imperative approach would bypass many steps such as scheduling and operate by communicating directly with the container runtime (e.g. the Docker daemon) on the nodes. That means pods can now be created in parallel if it is required.

2.3 Cold Start

When a serverless application serves an invocation request, or a microservice scales out due to a particular workload, one or more containers get created and started. To achieve this, the system requires to (i) bootstrap containers, (ii) prepare the software environments, and (iii) initialize the user code [4, 12]. This is called cold start. The execution of these steps might cause latency due to memory footprint, runtime, code package size, and more [5].

3 RELATED WORK

We present the related work in two groups: (i) rapid deployments where the focus is on speeding up the container ecosystem, and (ii) queue-based approaches where different techniques based on pools of containers are researched.

3.1 Rapid deployments

Caching and snapshots. Cadden et al. [7] present a technique based on dependency planning by introducing two caching solutions to improve the startup time of a task. In brief, their cache-aware scheduler schedules tasks by creating containers on nodes where there is an exact container image cached, or a sub-layer of it. Although unikernels are inherently different in comparison to Docker containers, Fu et al. [10] introduce an approach based on creating snapshots of unikernels ready to execute functions. Upon a function invocation, unikernels request the snapshots of this function to speed up its invocation.

Lazy image loading. In this approach, it is shown that only a small part of the container images (e.g. 6% in Docker images [11]) is enough to start a container, and the rest can be loaded lazily. FogDocker [8] presents a base image, after an analysis process on an image, with the essential files required to boot the container. The rest of the original image will be downloaded asynchronously. However, it is hard to mitigate application crashes at runtime when a required file is not yet fetched. Likewise, Slacker [11] uses the same technique mixed with layer-based snapshots and lazy cloning, i.e. this means that if Slacker wants to clone a particular layer of a Docker image, it only clones that specific layer and not the whole image. The advantage of this approach is that the pushing and pulling of containers run more smoothly, and as a result, the start-up time is reduced. A small trade-off is that the full runtime of applications becomes longer in experiments with a large load.

These techniques are not always compatible with each other. Some of the approaches based on Docker images are not always feasible in every situation, e.g. the "Lazy" approaches [8, 11], where we do not directly download all files from an image. This can become problematic with large applications that use a lot of files to operate.

3.2 Queue-based approaches

These techniques aim at preparing the infrastructural components (e.g. containers) as early as possible and keep them in pools. A warm queue of containers is a queue in which the containers are ready to process clients' requests. Lin et al. [15] reduce cold start latency by 85% through employing a pool of warm pods using Knative. However, the pod migration takes 2 seconds in this approach. Likewise, McGrath et al. [18] use cold queues to monitor the memory capacity of the worker nodes to start up new containers, and warm queues to keep track of existing warm containers. The other approach is to use pre-warmed queues where containers are started but the application is not yet initialized [22]. However, this technique is appropriate for stateless applications, especially those that do take much time to initialize, e.g. scripting languages like Python rather than compiled runtimes like Java and .NET [5].

The warm and pre-warmed queues are effective at the cost of high memory consumption. Mohan et al. [19] present an approach based on reusable network containers in OpenWhisk, inspired by `pause` containers in Kubernetes. In this way, they reduced the cold start latency by skipping the networking step.

4 AN IMPERATIVE APPROACH TO COLD START

In this section, we present three strategies, namely using (i) a pool of reusable network containers, (i) a layer-based library sharing, and (iii) imperative configuration of application containers, to mitigate cold start problem in Kubernetes.

4.1 Reusable network containers

In this approach, we aim at pre-creating the network infrastructure of application containers to decrease the cold start time. To achieve this, a queue of warm network containers can be used. When scaling-out is required and a new application container must be started, our scaler binds the newly started container to a warm network container. Afterwards, the network container is removed from the queue and placed in a hot queue, meaning that it is occupied. Using the hot queue, the network container can be released and put back in the network container queue when the application container is no longer required. That ensures *reusability*.

In Kubernetes, network containers, also known as `pause` containers, cannot be created on a stand-alone basis. To have a queue of `pause` containers, our approach is based on creating lightweight pods. Each pod is composed of a `pause` container as a result.

Pod injection. When a scale-out request arrives, an application container is injected to a pod as follows: our scaler controller starts an application container; it picks an entry from the queue (i.e. the entry includes the `pause` container ID, `IPC` namespace and `cgroup` of each pod in the queue); it configures the container to use the network namespace, `cgroup` and `IPC` namespace of the pod; and it places the pod in the hot queue to know that the pod is already running a worker instance. We call this "pod injection" as an application container is injected into a pod without using the Kubernetes API, and instead by communicating directly with the worker nodes.

4.2 Layered-based Library Sharing

Layered-based library sharing is a technique with which external components and software dependencies of an application are encapsulated in various layers of container images to mitigate the cold start problem. Other work showed that library sharing reduces the memory footprint by sharing common portions of memory between containers [9]. Our hypothesis is that a layer-based approach has an effect on the startup time of an application. The previous research uses this technique to reduce the image size. That means if a layer with the library already exists, it can be used by multiple containers. This basically makes the container image (e.g. Docker) smaller for the second container, as it no longer needs to provide the libraries for its applications; therefore, it reduces the cold start.

Programming languages influence the startup time of applications [17]. For example, Java applications typically include most dependencies in JAR files. In a JAR file, there are application classes, libraries, frameworks and a manifest file. If we work with a large JAR file, we only have 1 layer containing the entire application. So it is impossible to benefit from library sharing. To speed up the container image retrieval, a JAR file can be split into multiple layers.

4.3 Imperative scaling in Kubernetes

Using an imperative approach in Kubernetes, our scaler communicates directly with the worker nodes, bypassing the Kubernetes controller components. In a declarative approach, a state change request (e.g. deployment of a pod) is processed by Kubernetes API, various controllers and schedulers to implement a control loop to mitigate unintended states (e.g. a node or container crash). Moreover, pods are started sequentially and placed one by one. But, our imperative approach communicates directly with the container runtime (e.g. the Docker daemon), start the application containers and inject them into the network pods. Therefore, the entire process can be done in parallel and at the same time. Our hypothesis is that it considerably improves the application start up time when an application requires to scale out.

5 EVALUATION

We aim to evaluate the effectiveness of the presented techniques, namely (i) library sharing, (ii) employing a pool of network pods, and (iii) the imperative approach, to mitigate the cold start problem. We also evaluate how our imperative approach affects applications with strict SLO requirements based on job completion time.

Firstly, a job-oriented microservice application is introduced as our test application for the experiments. Secondly, we present our research methodology and the experiments conducted in that regard. Thereafter, we present our experiment setup and conclude with the experimental findings and their discussion.

5.1 Test application

We perform our experiments on a job processing microservice application (see Fig. 1). The application is composed of three parts: (i) the users who create jobs and add them to the queue, (ii) the queue where the active jobs are stored and (iii) the workers who are responsible for periodically checking the queue for jobs. Users add jobs to the queue. A job is a collection of tasks. When a lot of users are actively busy and producing different workloads, extra workers can be added to handle all possible requests. The Queue service and the workers have been implemented using RESTful Spring Boot in Java based on the Tomcat server.

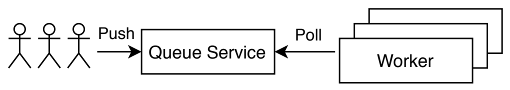


Figure 1: Test application

5.2 Experiment Methodology

Methodology. To evaluate the techniques introduced in this paper, four deployment variations of the test application, presented in Section 5.1, are employed:

(1) *DCL* where pods are created in a declarative way via Kubernetes and its policy-based scheduler, i.e. the pause and application containers are created together.
(2) *IMP* where network pods are created in advance so that we can use them to inject our application containers, i.e. the injection of these application containers is done asynchronously by running a script on the node on which a pod are available.
(3) *DCL + LIB* where everything is identical to *DCL* except now the Docker image of the application container consists of multiple layers to allow library sharing.
(4) *IMP + LIB* where everything is identical to *IMP* but with library sharing as *DCL + LIB*.

We employ the Locust [16] load testing tool to allow multiple users to join the experiments. They register jobs in the queue to start a scaling action. Each job that is being submitted by a user runs in its own container. We repeat these workload tests for each of the abovementioned deployment variations of the test application. Throughout this process, we inspect the occurrence of different events to measure the duration of: (i) starting the pause container, (ii) application container startup delay, (iii) starting up the application container, (iv) starting up the JVM, (v) starting up Tomcat server, (vi) finalizing Spring Boot startup, and (vii) finishing the first task. We analyze the effect of the deployment variations on these durations to understand how the different techniques of our approach affect the cold start problem.

Library sharing validation. Instead of using a large JAR file, we extract 3 layers to reuse in our Docker file. To speed up the build time of the Docker image, we have placed the layers that change the least, such as libraries, above the classes. If the dependencies are not changed, then the library layer does not need to change, resulting in a faster build time. The library layer can also be easily shared if we want to work with other applications.

No dependency layers

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ADD target/worker.jar app.jar
ENTRYPOINT exec java -jar /app.jar
```

Layered library sharing

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG LIB=target/
COPY $LIB/BOOT-INF/lib /app/lib
COPY $LIB/META-INF /app/META-INF
COPY $LIB/BOOT-INF/classes /app
ADD README.md README.md
RUN apk update java=8u191+
ENTRYPOINT exec java -cp "app:app/lib/*"
"be.kuleuven.WorkerApplication"
```

To validate that library sharing is properly implemented, we use the pmap and smaps tools as used in [9]. It allows us to check which libraries are being used, how much memory they share with the other processes, and obtain more detailed information about memory usage. We create two containers with the same image. We perform this validation once with library sharing and once without. Our results show that the libraries such as Tomcat, Spring, Avalon, etc. have been successfully shared in the case with library sharing; however, pmap and smaps shows no libraries in the "shared memory" section in the other case. We can therefore conclude that we have successfully shared our libraries with each other in our library sharing Docker container.

Experiment setup. To ensure that the results are comparable, we use the same environment for all experiments. This environment is an OpenStack-based private cloud. We created four virtual machines (VMs) where three of them belong to the Kubernetes cluster while the fourth one is used for load testing. The operating system of the VMs is Ubuntu 16.04. We used Kubernetes v1.12. The resource allocation of the VMs is organized as follows: (i) one Kubernetes master node with 2vCPU and 4GB RAM as the master node of the cluster, (ii) one worker node with 4vCPU and 8GB RAM. This worker node hosts both the test application and the scaler that is responsible for deploying the worker pods, (iii) one worker node monitoring with 2vCPU and 4GB RAM, and (iv) the load generator node with 4vCPU and 8GB RAM to simulate the load of an external user submitting jobs.

5.3 Experiments

In this subsection, we present our experiments and our findings. We have performed two experiments:

(1) *Experiment 1* to measure the total runtime of containers individually (i.e. 1 – 6 containers were created) and understand the impact of the presented techniques on cold start
(2) *Experiment 2* to measure the start and end time of 6 containers with 1000 tasks, especially to understand the impact on a scenario where there is a strict Service Level Objective (SLO) such as *job return time*

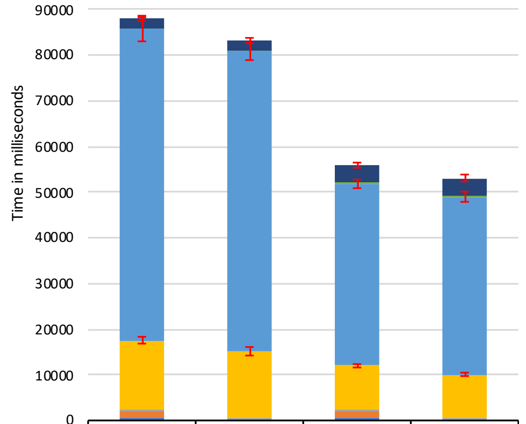
5.3.1 Experiment 1. This experiment includes 6 rounds, and each round has been executed against each of the deployment variations, namely declarative (*DCL*), imperative (*IMP*), declarative with library sharing (*DCL* + *LIB*) and imperative with library sharing (*IMP* + *LIB*). In the first round, only one container is launched for one user, in the second round, 2 containers are launched for 2 users, and so on up to 6 containers/users. We inspect the start-up time of the pods with a granularity described in Section 5.2. We ran each experiment at least 20 times and then averaged these results. Only 10 experiments have been performed with the six containers. We also show the standard deviation on the graphs in red showing how much the values differ from each other. In this paper, we investigate the extreme cases: Round 1 with one container, and Round 6 with six containers.

Round 1 (one container). Fig. 2a illustrates the first round in which only 1 container has been started (i.e. the results of registering one user). We collected fine-grained information regarding the startup of this container. We note that the strategies that use library sharing have a major impact on the boot time of the Tomcat server as well as when starting a JVM instance. The reason for this is that the libraries that Tomcat uses are provided with a higher layer in the Docker image and we therefore benefit from the layer-based library sharing that we introduced in Section 4.2. A small trade-off is the execution time of the first task with the library sharing strategies. We notice an increase when the Spring boot is completed.

The imperative approach, in Fig. 2a, does not seem to have considerable impact on the whole picture. That is because the Java ecosystem, and most compiled runtime enterprise languages, are slower at startup in comparison to scripting languages such as Python and Ruby. However, to figure out the real advantage of creating the network containers in advance, we zoom in on the bottom events of this chart regardless of the application deployed within the container. Fig. 3a illustrates a closer look at the container-related startup events. In this figure, the impact of creating a network container in advance is more clearly indicated. We can understand, that thanks to the creation of a network container in advance, we gain about two seconds speedup. This is certainly an improvement, as it does not depend on the programming language and the application dependencies. We obtain these results since with the imperative approach the pause containers were picked up from the pool of pods. Moreover, the declarative approaches typically create the pause containers first and then the application containers; therefore, there is a delay in between, which is avoided in the imperative approaches as a consequence. In Fig. 3b, we zoom in on the duration of the first task and notice that the standard deviation lines overlap. When the standard deviation lines do not or almost not overlap, the results are significantly different. In our case, however, we observed that the durations of the first task completion were not significantly different in this experiment.

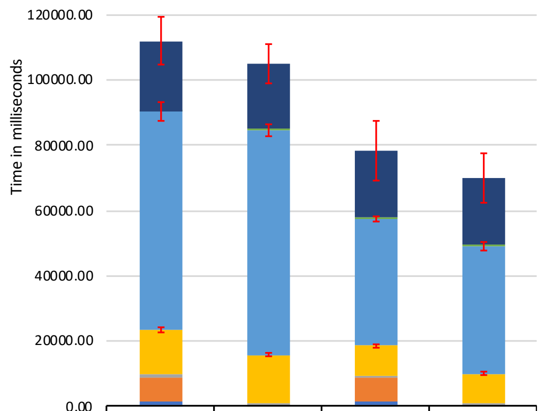
Round 6 (six containers). Fig. 2b shows the timings of the different deployment variations for the sixth round where 6 containers are started for 6 users. We performed the same experiments as in to Round 1, but we seek to see the differences or correlations with the previous observations in the other rounds. In this figure, we observe that the library sharing technique continues to have a major impact on the boot time of the Tomcat server and JVM. The duration of starting up an instance of the Tomcat server is even almost halved. However, the time of "finalizing spring boot startup" is doubled, but the numbers are only in the range of ~150 milliseconds, which is little compared to the benefit we get from the speedup of the other dependencies (Tomcat and JVM).

In Fig. 4a, the startup times from Fig. 2b are illustrated separately in which 6 containers (pods) have been started. Here we can deduce that creating network containers in advance has a minimal impact. However, if we do this together with an imperative configuration, we notice a great improvement. The advantage is greatest with successive containers. We notice this especially with container 6 of this experiments. The reason is that Kubernetes needs to wait for all pause containers and the predecessors to be created; that introduces considerable delay. More precisely, to create 6 pods, Kubernetes starts with creating the pause containers one by one. Once all the pause containers have been created, it starts with creating the application containers one by one. For instance, if we want to know the total startup time of the 4th pod with the *DCL* approach,



	DCL	IMP	DCL+LIB	IMP+LIB
🟦 Duration finish first task	2044.70	2130.70	3778.81	3867.10
🟩 Duration finalizing spring boot startup	154.10	169.90	396.57	304.80
🟦 Duration starting up Tomcat server	68230.95	65627.05	39771.05	38769.10
🟨 Duration starting up the JVM	15156.50	14801.55	9643.38	9639.75
⬜ Duration starting up the application container	434.55	463.60	436.10	480.85
🟧 Application container startup delay	1413.20	0.00	1426.57	0.00
🟦 Duration starting the pause container	606.80	0.00	592.71	0.00

(a) The fined-grained start-up times and execution time of (round 1, container 1) when only 1 container is scheduled to be created



	DCL	IMP	DCL+LIB	IMP+LIB
🟦 Duration finish first task	21498.18	19829.33	20608.80	20441.60
🟩 Duration finalizing spring boot startup	171.45	182.89	331.60	371.80
🟦 Duration starting up Tomcat server	66814.09	68945.44	38899.30	39053.90
🟨 Duration starting up the JVM	13897.64	14867.89	9097.70	9168.50
⬜ Duration starting up the application container	611.36	1041.78	618.60	997.70
🟧 Application container startup delay	7660.00	0.00	7503.40	0.00
🟦 Duration starting the pause container	1401.82	0.00	1349.50	0.00

(b) The fined-grained boot-up and execution time of (round 6, container 6) when 6 containers are scheduled to be created

Figure 2: Experiment 1 (round 1 and round 6)

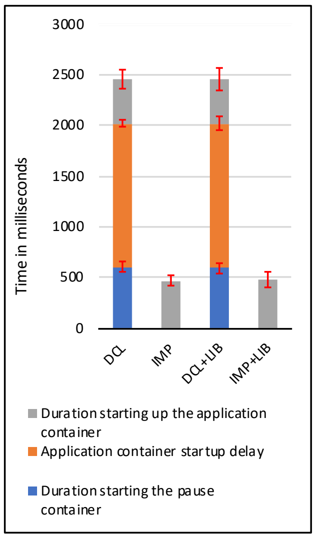
application container 4 needs to wait till all pause containers (1...6), and the application container 1, 2 and 3 are created and started. Then, the container 4 can get created and started; by starting up we mean the container not the application. The only drawback of the imperative approach is that the duration of "starting up the application container" increases. But it is a small trade-off. We can conclude that the imperative approach is considerably beneficial when multiple containers get started up simultaneously.

Findings. Based on the results of the experiment 1 (see Section 5.3.1), several findings can be concluded as follows:

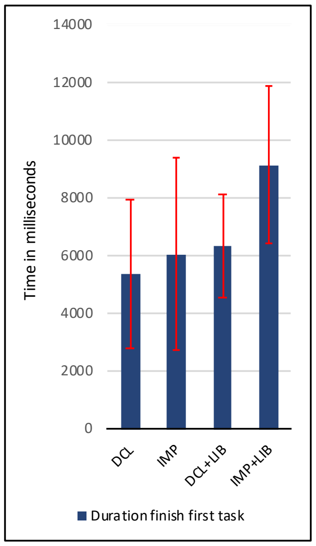
(1) The library sharing technique has the greatest impact across all experiments. It resulted in a reduction in the start-up time of the JVM and the Tomcat server. The only drawback is that the duration of "finalizing spring boot" is getting longer, but this is very little compared to the overall improvement.
(2) The library sharing technique has a negative impact on the execution time of the first task. However, the standard error bars overlap, which indicates that statistically the results of these experiments are not significantly different. This means that library sharing across multiple experiments does not negatively impact the execution time of the first task.
(3) The approach to pre-create the network containers seems to have a small impact on cold start when starting only one container. The only advantage is that we can skip the network creation step. However, the impact is greater when multiple containers start up simultaneously. The reason for this is the way Kubernetes creates its pods and containers. Firstly, it creates the necessary pause containers before the application containers. As a result, we observe a considerable impact on the boot-up time, especially with the last container. In the Kubernetes approach, the last container has to wait until all pause containers and application containers have been created before it can start its own creation process. A small drawback with this technique is when several containers are started simultaneously, the start-up of the application container takes longer. However, this is again negligible in comparison to the overall improvement.

5.3.2 Experiment 2. In this experiment, we primarily focus on the start time of the application containers; creating the network containers in advance does not affect this experiment. The aims of this experiment are (i) to examine the possible impact of different strategies on the start and end time of the application, (ii) to realize which technique is more suitable in case of having service layer objectives (SLO) such as job completion time, and (iii) to understand the impact of each approach on the CPU usage. We expect that the strategies that start their containers the fastest will also reach their optimal CPU usage faster. We test the techniques on the aforementioned deployment variations of the test application discussed earlier.

We let six users register in the application simultaneously. After that, each user sends 1000 tasks to the application. The end time is equal to the time when a user finished his 1000 tasks. We use Locust

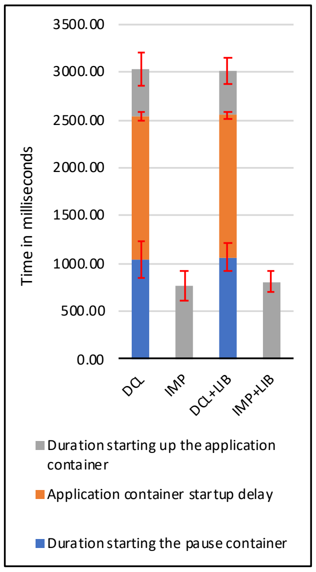


(a) The container startup events, (round 1, container 1) based on Fig. 2a

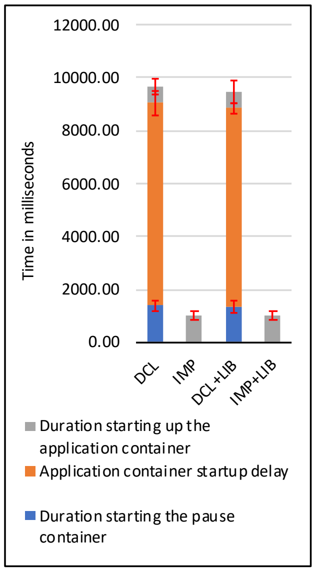


(b) The duration of completion of the first task (round 2, container 1)

Figure 3: A closer look at Experiment 1 (round 1 and round 2)



(a) The container startup events, (round 6, container 1)



(b) The container startup events, (round 6, container 6)

Figure 4: A closer look at Experiment 1 (round 6)

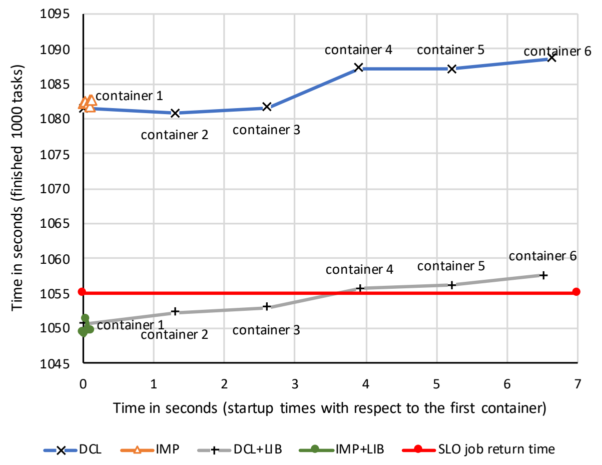


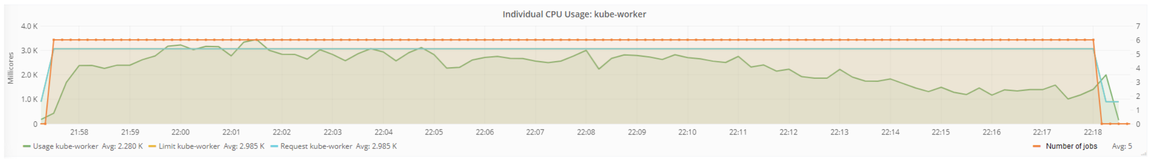
Figure 5: Experiment 2 with 6 containers

to simulate the workload generated by the users. To determine when the 1000th task was executed, we look at the logs of the Docker application to extract the time of the 1000th task. The start and end time of each case are measured in relation to the start time of the first container. In other words, if container 2 has started three seconds after container 1, the startup time of container 2 is three seconds.

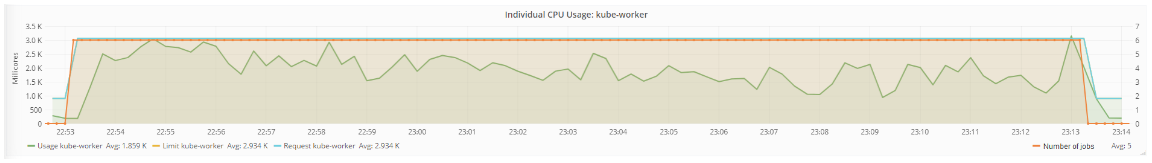
Impact on the duration of processing all tasks. Fig. 5 illustrates that the imperative approach (*IMP* and *IMP + LIB*) starts all 6 containers before the start of container 2 of the ones with declarative approaches (*DCL* and *DCL + LIB*). This is due to the asynchronous creation of the application containers using the imperative approach. Therefore, it means that the other containers do not have to wait for container 1 to initiate their own start-up process. Moreover, our library sharing approach has a positive impact on the end times of this experiment. This impact comes from the previously seen benefits of faster boot times for the Tomcat server and JVM (see Experiment 1).

Meeting job completion time objective (SLO). In Fig. 5, the red line represents the job return time as an SLO. An SLO is an agreement that is made between the service provider and the customer. The agreement is that 1000 tasks will be performed per container in 1055 seconds. If we look at our imperative approach, we conclude that this deadline is easily achievable by all containers. With the declarative approach, this SLO is no longer feasible for the fourth container. This is because the 4th container has to wait for the creation of container 1, 2, 3 and all pause containers. We can deduce that our imperative approach is effective to meet SLO deadlines because these containers can start up in parallel.

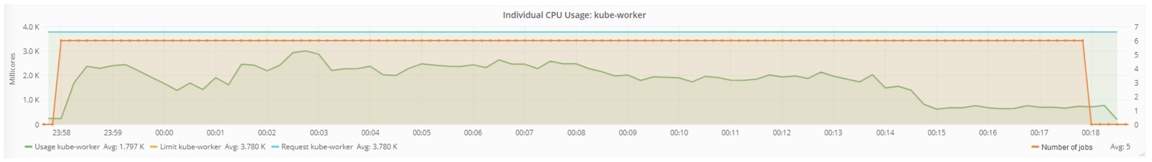
Fastest containers reach optimal CPU usage faster. The cold start problem also affects CPU usage. This problem causes the CPU usage to increase steadily as shown in Fig. 6a. If we look at the impact of library sharing in Fig. 6b, we can already observe an improvement. We see a straight rise to the highest point. Fig. 6c shows the effect of the imperative approach. The start-up phase of this experiment shows an improvement in the CPU usage, but it eventually fades. This is because this approach is mainly effective for the start-up of the containers. Finally, we combine these two strategies in Fig. 6d, where the two previously seen benefits are combined. Our CPU usage is now rising in the beginning and right to its peak.



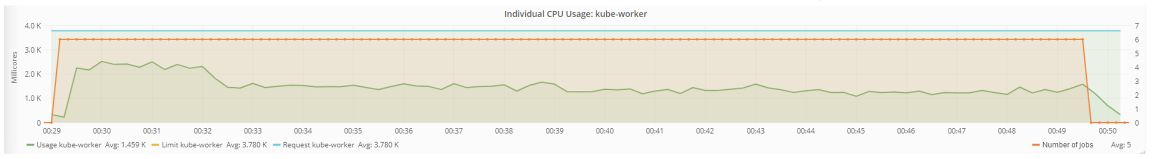
(a) The declarative approach without library sharing



(b) The declarative approach with library sharing



(c) The imperative approach without library sharing



(d) The imperative approach with library sharing

Figure 6: CPU usage in Experiment 2

Findings. Based on the results of the experiment 2, our findings are as follows:

(1) The imperative approach further enables the application to meet SLO targets without a non-deterministic delay that appears in the declarative approach when multiple Pods must be created at the same time due to concurrent user requests.
(2) These two techniques not only reduce the cold start but also provide the application with the processing power (CPU) as early as possible, resulting in a faster overall job completion time.

6 CONCLUSION

This paper has investigated how a queue of reusable network containers, layer-based library sharing, and imperative configuration management, when combined together, can improve the "cold start" problem in Kubernetes. Cold start is a well-known problem in the elastic scaling of containers, especially in serverless computing. Sometimes several containers are required to be started simultaneously, and the applications deployed on these containers are the same or share software dependencies, increasing the impact of this problem. We evaluated the above-mentioned techniques extensively in a deadline-oriented job processing microservice.

Our findings show that (i) the library sharing approach results in a large reduction in the start-up time of software dependencies (e.g. the JVM and Tomcat server), (ii) pre-creating network containers has greater impact when multiple application containers are started in parallel, (iii) the imperative configuration approach introduces start-up time determinism and predictability, making this approach more reliable for applications with SLOs such as job completion deadlines.

REFERENCES

[1] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. *Serverless Computing: Current Trends and Open Problems*. Springer Singapore, 1–20. https://doi.org/10.1007/978-981-10-5026-8_1
[2] David Bernstein. 2014. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing* 1, 3 (2014), 81–84. https://doi.org/10.1109/MCC.2014.51
[3] U. Breitenb ucher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger. 2014. Combining Declarative and Imperative Cloud Application Provisioning Based on TOSCA. In *2014 IEEE International Conference on Cloud Engineering*. 87–96.
[4] Eric A Brewer. 2015. Kubernetes and the path to cloud native. In *Proceedings of the sixth ACM symposium on cloud computing*. 167–167.
[5] Renato Byrro. 2019. Can We Solve Serverless Cold Starts? https://dashbird.io/blog/can-we-solve-serverless-cold-starts/
[6] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2019. SEUSS: Rapid serverless deployment using environment snapshots. *arXiv preprint arXiv:1910.01558* (2019).
[7] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2019. SEUSS: Rapid serverless deployment using environment snapshots. *arXiv preprint arXiv:1910.01558* (2019).
[8] Lorenzo Civolani, Guillaume Pierre, and Paolo Bellavista. 2019. FogDocker: Start container now, fetch image later. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*. 51–59.
[9] José Bravo Ferreira, Marco Cello, and Jesús Omana Iglesias. 2017. More sharing, more benefits? A study of library sharing in container-based infrastructures. In *European Conference on Parallel Processing*. Springer, 358–371.

[10] Silvery Fu, Radhika Mittal, Lei Zhang, and Sylvia Ratnasamy. 2020. Fast and Efficient Container Startup at the Edge via Dependency Scheduling. In *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*.

[11] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Slacker: Fast distribution with lazy docker containers. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*. 181–195.

[12] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[13] Kubernetes. 2020. Kubernetes Documentation. https://kubernetes.io/docs/concepts/workloads/pods/

[14] Ian Lewis. 2017. The Almighty Pause Container. https://www.ianlewis.org/en/almighty-pause-container

[15] Ping-Min Lin and Alex Glikson. 2019. Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach. *arXiv preprint arXiv:1903.12221* (2019).

[16] Locust. [n. d.]. Locust: An open source load testing tool. https://locust.io/. [Last visited on September 28, 2020].

[17] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. 2018. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 181–188.

[18] Garrett McGrath and Paul R Brenner. 2017. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 405–410.

[19] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile cold starts for scalable serverless. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

[20] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. *SIGOPS Oper. Syst. Rev.* 41, 3 (March 2007), 275–287. https://doi.org/10.1145/1272998.1273025

[21] Spring. [n. d.]. Getting Started | Building an Application with Spring Boot. https://spring.io/guides/gs/spring-boot/

[22] Markus Thommes. 2017. Squeezing the milliseconds: How to make serverless platforms blazing fast! https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0

[23] Dominik Tornow. 2018. Imperative vs Declarative. https://medium.com/@dominik.tornow/imperative-vs-declarative-8abc7dcae82e

[24] W. Wang, L. Zhang, D. Guo, S. Wu, H. Cui, and F. Bi. 2019. Reg: An Ultra-Lightweight Container That Maximizes Memory Sharing and Minimizes the Runtime Environment. In *2019 IEEE International Conference on Web Services (ICWS)*. 76–82.