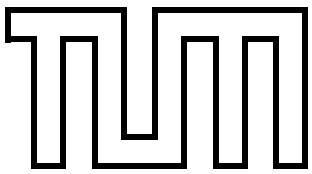


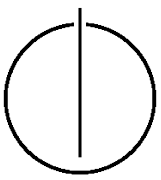
FAKULTÄT FÜR INFORMATIK

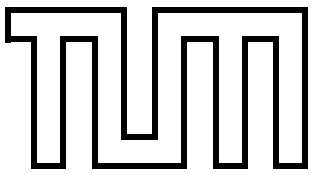
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

λ-Cntr: A Dependable Serverless Architecture

Eduard von Briesen





FAKULTÄT FÜR INFORMATIK

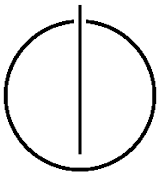
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

λ-CNTR: A Dependable Serverless Architecture

λ-CNTR: Eine zuverlässige Serverless Architektur

Author:	Eduard von Briesen
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Jörg Thalheim, M.Sc.
Date:	12.11.2021



I assure the single handed composition of this bachelor thesis only supported by declared resources,

Munich, 12.11.2021 Eduard von Briesen

Abstract

Function as a Service or FaaS is a modern category of cloud computing services, providing developers with a platform to deploy functionality without having to deal with the complexities of building and maintaining the underlying infrastructure. Serverless functions are build and deployed as minimal black-boxes that offer no additional tooling to keep size and start-up times slim, this makes them notoriously hard to debug.
λ-Cntr provides developers with additional debugging capabilities without modifying the deployment of functions. At run-time λ-Cntr allows users to efficiently deploy a debug container containing any number of tools and dynamically attaching it to a containerized function.
This is achieved using Cntr, a system which allows the combination of two container images in a nested namespace. λ-Cntr deploys Cntr within a privileged pod in a Kubernetes environment and manages its attachment to other containers. In our evaluation, λ-Cntr incurs reasonable performance overheads and shows comparable performance to Kubernetes' ephemeral containers.

Zusammenfassung

Function as a Service oder *FaaS*, ist eine moderne Kategorie von Cloud-Computing-Diensten. Sie bietet Entwicklern eine Plattform zur Bereitstellung von funktionalem Code, ohne sich mit der Komplexität des Aufbaus und der Wartung der darunter liegenden Infrastruktur befassen zu müssen. *Serverless* Funktionen werden als minimale Blackboxen erstellt und bereitgestellt. Sie enthalten keine zusätzlichen Tools, um ihre Größe und Start-Up-Zeiten zu minimieren und dies erschwert das debuggen solcher Anwendung. λ-Cntr bietet Entwicklern zusätzliche Debugging-Möglichkeiten, ohne die Funktionen zu modifizieren. Zur Laufzeit ermöglicht λ-Cntr den Benutzern effizient einen Debug-Container bereitzustellen, der eine beliebige Anzahl von Tools enthält und sich dynamisch an eine containerisierte Funktion anhängen lässt.
Dies wird mit Cntr erreicht, einem System, das die Kombination von zwei Containern in einem geteiltem Namespace ermöglicht. λ-Cntr setzt Cntr innerhalb eines privilegierten pod in einer Kubernetes-Umgebung ein und verwaltet dessen Anbindung an anderen Containern. In unserer Evaluation zeigt λ-Cntr einen angemessenen Leistungs-Overheads und ist vergleichbar mit Kubernetes' Ephemeral Containern.

Contents

Chapter 1

Introduction

1.1 Problem

Serverless computing has seen rapid adoption in recent years, due to its flexible scalability and ease of deployment. Function as a Service (FaaS) allows developers to focus on implementing functionality instead of building and maintaining increasingly complex cloud infrastructures. Numerous frameworks exist that manage everything, from the deployment and scaling of serverless functions to the underlying hardware. Commercial solutions such as Amazon AWS Lambda [2], Microsoft Azure [36] or Google Cloud Functions [18] implement a flexible pay-as-you-go billing model. Open-source alternatives such as OpenFaaS [39] and Apache OpenWhisk [7] allow users to setup their own serverless environments with a greater degree of control, at the cost of additional effort.

Although FaaS offers many valuable advantages, the debugging of serverless applications is one of its major drawbacks. Troubleshooting FaaS applications is challenging due to their complex production environments, the limited access once deployed and the lack of tooling [54, 56].

Frameworks such as Amazon AWS SAM [3] provide a local testing environment, which allows more traditional debugging methods without attached costs of running the application in the cloud. Where local debugging fails is in the simulation of the complex production environments in which a function has to operate. Certain errors might only occur once the function is deployed and interacts with other services.

Debugging in the cloud avoids this shortcoming. Calling a function through its API presents a simple method to inspect if it is running as intended. Developers can gain more insight into distributed applications by combining logs [60], traces [57, 58, 61] and monitoring data [51, 62, 66]. Although this

information is immensely valuable to identify and locate bugs in a production environment, it requires a trade-off between expressiveness and performance or cost. The main disadvantage of this method is the lack of insight into running functions.

1.2 Objectives

λ-Cntr addresses current shortcomings of serverless debugging, by giving users insight into serverless functions and providing them with additional tools. λ-Cntr uses Cntr [63] to combine two running containers within the context of a Kubernetes environment. This enables developers to package additional tooling into a debugging container, that can be deployed and attached to any container at run-time.

1.3 Outline

Chapter 2 summarizes technologies relevant to λ-Cntr. Chapter 3 outlines the design goals behind λ-Cntr and gives a high-level design overview. The implementation of the main three system components is outlined in Chapter 4, followed by the evaluation of λ-Cntr in Chapter 5, where its performance, security and effectiveness are examined. Chapter 6 gives an overview of related works and Chapter 7 concludes the thesis with a summary.

Chapter 2

Background

This chapter outlines technologies relevant to λ-Cntr.

2.1 Cntr

Cntr [63] aims to speed up deployment time and decrease the size of containers, by freeing up resources that usually get wasted by the inclusion of unnecessary tools. With Cntr, containers can be split into two distinct parts, a *slim* and a *fat* image. The *slim* image provides the main functionality and can be deployed efficiently, while the *fat* image contains additional tooling that can be dynamically attached at run-time.
Cntr achieves this by creating a nested namespace and merging the namespaces of the two containers. CntrFS implements a filesystem in userspace (FUSE) that is mounted as the root (`/`) in the new nested namespace. The application filesystem is re-mounted (`/var/lib/cntr`), and any request to tools located on the *fat* image are handled and redirected by the CntrFS server.
Cntr is compatible with any container engine, and can be easily attached to any container given its ID. When attached, it adopts the complete execution context of the target container (container namespaces, environment variables, capabilities, ...). These properties establish Cntr as an immensely powerful tool, whose capabilities λ-Cntr employs in the space of serverless computing.

2.2 Kubernetes

Kubernetes [27] is an open-source container orchestration system that allows the deployment, scaling, and management of containerized applications. A Kubernetes cluster consists of at least one node that runs pods. Pods are

groups of one or more containers with shared resources and specifications managed by a Kubelet running on each node. Kubernetes provides a well documented API that allows the querying and manipulation of objects like pods or namespaces.
There are numerous competitors to Kubernetes such as Docker Swarm [13], Apache Mesos [6] or Ranchers Cattle [44]. While these alternatives show some advantages in small deployments, Kubernetes outperforms them in complex environments [55] and has established itself as the industry standard. λ-Cntr is implemented specifically for the use with Kubernetes, future work could include compatibility with other orchestration engines.

2.3 OpenFaaS

OpenFaaS [39] is a framework for building and deploying serverless functions with Docker and Kubernetes. Every function is exposed and can be invoked with OpenFaaS' Gateway which can be accessed through its REST API, CLI or UI. Functions get deployed as immutable Docker images with faas-netes [14], the faas-provider for Kubernetes. NATS [37] is used for asynchronous execution and queuing of functions, metrics are enabled by Prometheus [42] and auto-scaling trough AlertManager [43].
Functions are created from predefined templates that allow for further customization. Depending on the language of the function the structure of templates may differ, but they all include the functional code, some form of package management and a Dockerfile. When the function is built according to the Dockerfile, the OpenFaaS watchdog [40] is included and a port is exposed to which the container listens to at run-time. The watchdog is responsible for starting and monitoring the function. OpenFaaS does not perform any additional performance optimizations during the build stage, and it is left to the developer to not bloat functions with unnecessary code or dependencies. If desired, users can modify the Dockerfile to include additional utilities at the cost of performance. λ-Cntr offers the flexibility to delay this inclusion until run-time and only for specific functions where the tools are needed.

Chapter 3

Design

λ-Cntr deploys a debugging container in a Kubernetes cluster and lets the user attach it to any other container running on the cluster, in an on-demand fashion using Cntr [63]. (Figure 3.1)

3.1 Design Goals

The design of λ-Cntr is motivated by following design goals.

- *Generality:* λ-Cntr should be compatible with any Kubernetes distribution and should support a wide-range of workflows (e.g. debugging, tracing, profiling).
- *Transparency:* λ-Cntr should support these workflows, without modifying its target, the container runtime, the Kubernetes deployment or the underlying host operating system.
- *Efficiency:* λ-Cntr should introduce minimal performance overheads.

3.2 Overview

λ-Cntr is built as an extension of Cntr and uses it as a black-box without any modifications. As a result λ-Cntr benefits from the generality and transparency provided by Cntr. The design challenges present themselves in adopting Cntr into a Kubernetes environment.
To ensure greater portability, we decided to package the debugging container with the Cntr binary into a pod instead of executing Cntr directly on the node itself. Pods present themselves as the ideal deployment type for λ-Cntr, as they are the most efficient way to deploy and manage a single

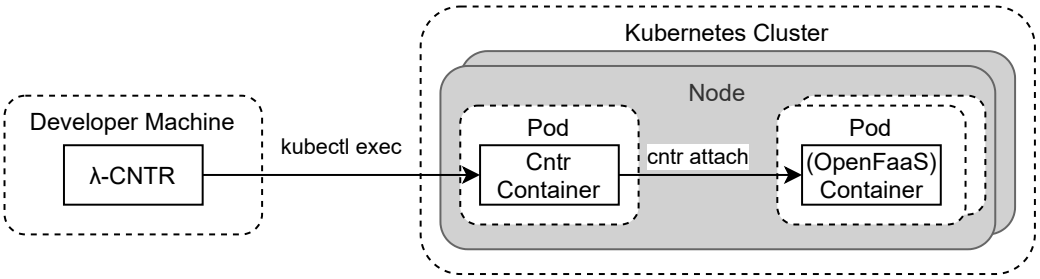


Figure 3.1: λ-Cntr deployment on Kubernetes.

container within Kubernetes. Use-cases targeted by λ-Cntr do not benefit by the additional functionality such as scaling offered by more complex workload resources. The usage of pods allows λ-Cntr to efficiently support multi-node clusters, as the only thing required to support an additional host is the deployment of another pod specific to that node.
DaemonSets [30] offer a mechanism that ensures that all nodes run a copy of a pod, they are intended for services that have to run reliably in the background such as logging or monitoring. λ-Cntr is intended to be only deployed as long as it is needed, but some use-cases may arise where continuos deployment is desireable. We consider the deployment of λ-Cntr with DaemonSets as possible future work.
λ-Cntr provides users with a simple interface, while handling all the intricacies of pod creation and management in the background. This includes providing pods with the necessary privileges and access to the container runtime, managing deployments across multiple nodes and handling attachment to target containers.

3.3 λ-Cntr workflow

λ-Cntr is easy to use, the user only has to provide the name of the target pod and its namespace. λ-Cntr then exposes the shell to the user, from where they can access the filesystem of the container with the additional tools defined in the λ-Cntr-container. The user can define and build their own Docker container to include specific development tools. In the background λ-Cntr executes the following steps.

1. *Resolve pod name to container ID:* λ-Cntr queries the Kubernetes API to get the container ID(s) of the target pod, given its name. In case the pod contains multiple containers the user is required to specify

the exact one. The ID is required by CNTR to further resolve it to the process ID and obtain the target containers context.

2. *Deploy debugging pod:* λ-CNTR initiates the startup of a new pod containing the debugging container on the node of the target pod, and waits until it is ready. If there already is a λ-CNTR-pod deployed in the specified namespace on the same node, this step is omitted.

3. *Initiate interactive shell and attach to pod:* λ-CNTR initiates an interactive shell with the λ-CNTR-pod and attaches to the container using CNTR, forwarding the I/O to the user.

4. *Close shell and terminate pod:* When the user exits the shell of the pod, λ-CNTR terminates the deployed pod. Simply closing the shell leaves the pod running for future use.

Chapter 4

Implementation

λ-Cntr is implemented in Rust as it offers good performance with strong security and a wide range of libraries. The following chapter outlines the detailed implementation of its components.

4.1 Command line parser

We use clap [9] as a simple to use and efficient library for command line parsing. This provides a powerful framework in which it is only necessary to define possible commands (154 LoC).

Subcommands. The two similar subcommands `attach` and `execute` are defined. They both take the same arguments and options, the only difference being that `execute` takes an additional `command` argument. The workflow as described in Section 3.3 is initiated with `attach`. The `execute` command is mainly used for testing and debugging purposes: instead of opening a shell, it only executes the passed command in the pod without terminating it afterwards.

Arguments. The user is required to pass the name of the target pod as the first argument. In case the pod contains more than one container, one must be specified by passing its name as a second argument.

Flags. Clap automatically generates the two common flags `--help` and `--version`, both provide their expected functionality. λ-Cntr does not define any additional flags of its own.

Options. The following options can be set by the user:

`-i, --image <image>` If no argument is provided, the environment variable `CNTR_IMAGE` is checked, if no image is provided here either, a default image is used.

`-s, --socket-path <socket-path>` Similar to `image`, if no argument is passed `SOCKET_PATH` is used if present, otherwise the standard containerd socket path `/run/containerd/containerd.sock` is used.

`-n, --namespace <namespace>` If no argument is provided, `default` is used.

4.2 Kube-controller

The kube-controller handles all communication with the Kubernetes API using the kube-rs crate [26] (237 LoC). In this section we describe its core functionality.

Initialization. Before the kube-controller handles any request by the user, it creates and initializes a Kubernetes client, using the in-cluster environment variables first, falling back to the local `kubeconfig`. If this step succeeds, the pods of the specified namespace are fetched, and the name of the target pod is resolved to a container ID using an auxiliary function.

Deploy. If the initialization is successful, a λ-Cntr-pod can be deployed. The pod is deployed using the manifest created by the templater (§ 4.3), the program waits until it is ready. If a λ-Cntr-pod already exists in the given namespace on the same node, this process is skipped.

Attach. Once the λ-Cntr-pod is up and running, an attached process is started, similar to the `kubectl exec --tty` command. Before the `stdin` and `stdout` of the debug pod get passed through to the user, `cntr attach` is called using the container ID which was resolved in the initialization phase. This starts an interactive shell to the target container in the debug pod with which the user can now interact. The process is kept alive until the user exits the shell to the pod.

Auxiliary. The auxiliary functions include methods to resolve the target pods name to its containers ID, identify the container runtime and determine the node λ-Cntr has to be deployed on.

4.3 Templater

The templater builds a small deployment manifest for the λ-Cntr-pod (75 LoC). Important fields that give the pod the necessary privileges to run Cntr are statically set here. These include:

`hostPID`: With this option set to `true`, the container is given access to the hosts process ID namespace.

`securityContext.privileged`: With this flag enabled, the container is given access to all the hosts devices.

`securityContext.runAsUser`: With this option set to `0`, the container is run as the `root` user.

Relevant fields that are set at runtime are:

`image`: The debugging image that is deployed. By defining the `image` at runtime, the user can use their own debugging container image, provided it includes the Cntr binary.

`volumes.hostPath.path`: Has to point to the container socket of the node to which λ-Cntr is deployed. The container socket is required by Cntr, in order to resolve the container ID to the process ID. Its path on the host varies based on the Kubernetes distribution and the container runtime.

`volumeMounts.mountPath`: The mount path of the container socket within the pod. The path depends on the container runtime and is determined by λ-Cntr.

`nodeSelector`: To ensure the λ-Cntr-pod is deployed to the target pods node, the well known label `kubernetes.io/hostname` is set to the nodes hostname. The value is determined by λ-Cntr based on the target pod.

Chapter 5

Evaluation

In this section, we present an experimental evaluation of λ-Cntr. Our evaluation answers the following questions:

1. How performant is λ-Cntr? (§ 5.1)
2. How secure is λ-Cntr? (§ 5.2)
3. How effective is λ-Cntr for in-production debugging of real-world serverless applications? (§ 5.3)

5.1 Performance and Overheads

Experimental Testbed. To evaluate in a realistic environment we setup a Kubernetes v1.21.5 node using k3s [24]. The machine has a Intel Xeon Gold 6238R (28 Cores / 56 Threads) and 256GB of memory. A Samsung MZ7LH960 provided 960GB of storage using the `ext4` filesystem and the Linux kernel was at version 5.8.0.

5.1.1 Performance.

We first evaluate the start-up performance of λ-Cntr in direct comparison to Kubernetes' ephemeral containers (§ 6.2).

Methodology. We attach a debugging container to pods in a production environment with λ-Cntr and ephemeral containers. We measure the cold-start time without the debugging container being deployed, and the warm-start time with the debugging container already deployed. The time is measured from command execution to the opening of the interactive shell in milliseconds.

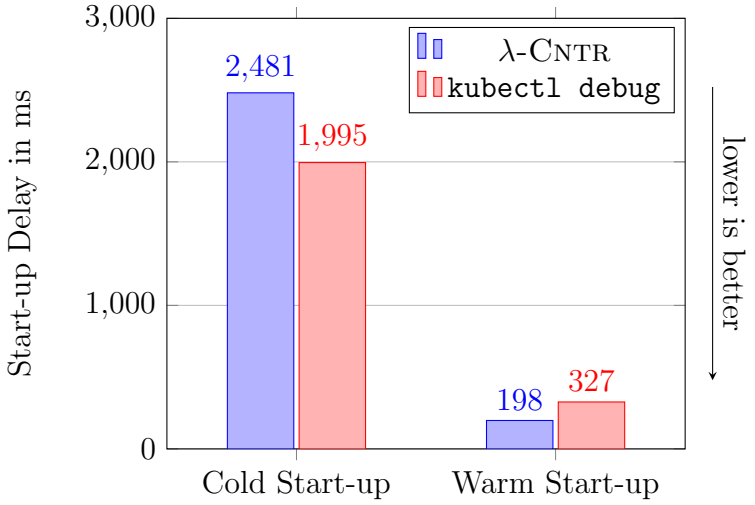


Figure 5.1: Start-up latency to open a shell with λ-Cntr and `kubectl debug`.

Workloads. We deploy a debugging container-, including tools representative of a multitude of debugging use-case such as coding (vim [49], git [17]), networking (netcat [38], iperf [22]) and processes monitoring (htop [21], strace [47]). The complete list of tools can be found in the Dockerfile [33]. The container image is based on `alpine` [1] and is sized at 738.17MB without, and 861.39MB with the Cntr binary (including auxiliary binaries). These two images are used with `kubectl debug` and `lambda-cntr attach` respectively. In all tests we attached to a `busybox` pod [8].

Results. Our results presented in Figure 5.1 show similar outcomes for both approaches. The cold start-up without the debugging container being deployed takes much longer, most time here is spent on the deployment. λ-Cntr is about half a second slower since a whole new pod has to be started, while `kubectl debug` adds the ephemeral container to an existing pod. Once the container is deployed, the attaching process only takes a fraction of a second and is again similar between both methods. The major advantage of λ-Cntr over ephemeral containers is that a cold start-up only has to be performed once, and the warm start-up time holds for every following attachment to any pod of the same node. Since ephemeral containers are deployed within the specific target pod, any first attachment to a pod requires a cold start-up.

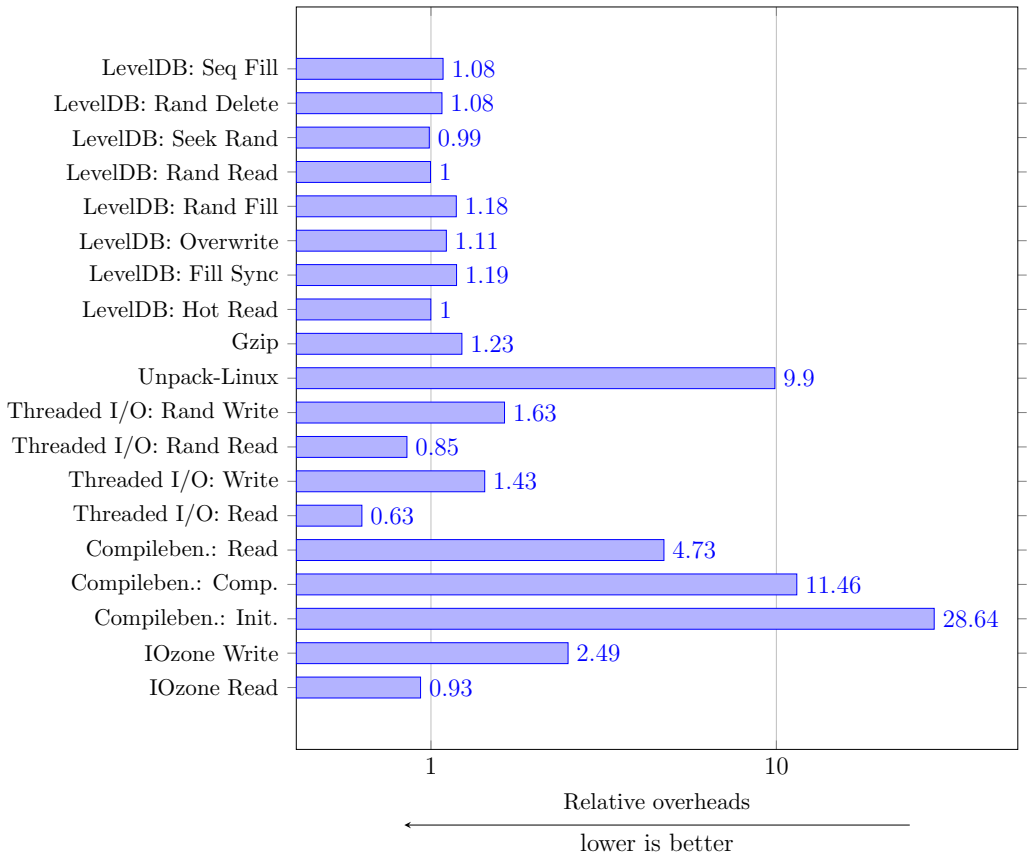


Figure 5.2: Relative performance overheads of λ-Cntr when compared to the native filesystem performance.

5.1.2 Overheads

Next we evaluate the overhead introduced by λ-Cntr. Our approach closely follows the one presented in the evaluation of Cntr [63].

Benchmark: Phoronix suite. Performance measurements are done using the Phoronix test suite [41]. Phoronix is an open-source meta benchmark which provides a wide range of filesystem benchmarks and workloads.

Methodology. In order to compute the relative overheads of λ-Cntr, we ran the benchmarks directly on the native container filesystem as a baseline and compare the performance results when accessing the filesystem trough Cntr. We ran the benchmarks within containers on the Kubernetes cluster.

Summary of the Results. Figure 5.2 shows the relative overheads of λ-Cntr compared to running the benchmarks natively, the lower the result the better. All results are publicly available on the openbenchmark platform [45].
In the experiment, 11 out of 19 benchmarks show a moderate overhead of less than 2×. Only four tests show a substantial decrease of performance with λ-Cntr, caused by frequent `inode` lookups, this includes the *Compilebench* (up to 28.6×) and *Unpack-Linux* (9.9×) benchmarks. The benchmarks *IOzone: Read* (0.93×), *Threaded I/O: Read* (0.63×) and *Threaded I/O: Rand Read* (0.85×) showed a slight increase in throughput speeds when compared to the native filesystem, due to additional caching performed by the underlying CntrFS.
In general, performance of reads stays mostly unaffected, while writes encounter overheads of up to 2.5×. The performance of λ-Cntr in total is decent, and it is notable that our setup represents a worst case with I/O-heavy operations all performed through the debugging container. In addition the target use-cases of λ-Cntr, such as debugging with auxiliary tools, are generally not as I/O intensive as the presented workloads.

IOzone. IOzone is a file system benchmark utility. We performed sequential read and write operations with a file size of 8GB and a 4KB block size. Read performance is on par with the native filesystem and surpasses it slightly (0.93×), due to the additional caching performed by CntrFS. Writes show a substantial performance hit (2.49×) as a result of the additional filesystem layers and due to not being able to utilize caching as well as during read operations.

Compilebench. Compilebench simulates different stages in the compilation process of the Linux kernel. These stages include (a) the `initial create` stage simulating a tarball unpack, (b) the `compile` stage compiling the kernel module, and (c) the `read compiled tree` stage recursively reading a source tree. In our testing, Compilebench showed by far the greatest overheads. Due to its meta-data oriented nature the benchmark performs many `inode` lookups, resulting in a considerable amount of time intensive context-switches.

Threaded I/O. The Threaded I/O benchmark performs both sequential and random read and write operation of 64MB files, with 16 threads simultaneously. We see better results as shown with IOzone, with fast read (0.63×) and good write (1.43×) performance when compared to the native filesystem.

Again reads can be mostly served from the cache and λ-CNTR overall benefits from the multithreading optimization of CNTRFS. Performance marginally decreases for random reads (0.85×) and writes (1.63×).

Unpack-Linux. This benchmark measures the time it takes to extract the Linux kernel package from a compressed tarball. The workload is comparable to the Compilebench benchmark and encounters similar bottlenecks as outlined above, resulting in a similar high overhead (9.9×).

Gzip Compression. This test measures the time needed to compress two copies of the Linux kernel source tree using `Gzip`. The data is served faster by λ-CNTR and the native filesystem than the `Gzip` compression processes it, thus we only see a small performance overhead (1.23×).

LevelDB. LevelDB is a fast key-value storage library written at Google using Snappy [46] for data compression. Across the benchmarks, the overhead stays minimal for read heavy (`hot read, rand read, seek rand`) and increases up to about 20% on write heavy workloads (`fill sync, overwrite, rand fill, rand delete, seq fill`). These results adhere to the behavior observed in previous read/write workloads, although to a lesser extend and without the substantial performance hits seen in the (de-)compression benchmarks.

5.1.3 Memory footprint

As the final part of the performance evaluation we measure the memory footprint of λ-CNTR.

Methodology. We measure the memory footprint of the λ-CNTR-pod and compare the results between λ-CNTR and ephemeral containers. We measure memory usage of pods and their containers using the Kubernetes Metrics Server [31] after starting up a shell and idling.

Workloads. We use the same debugging images as described in the performance section of this evaluation (§ 5.1) and attache to a `busybox` pod.

Results. The results depicted in Figure 5.3 show the memory footprint of the individual containers for λ-CNTR and `kubectl debug`. The `busybox` container is not manipulated in both approaches, thus showing the same memory consumption. The pod created by λ-CNTR requires about four

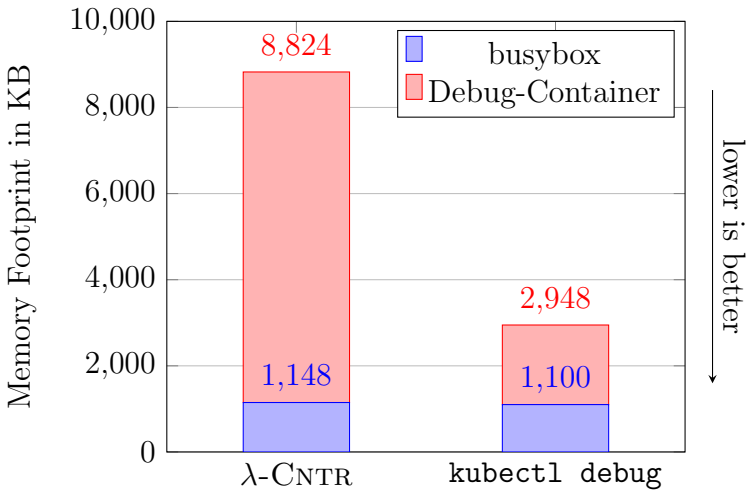


Figure 5.3: Memory footprint of debugging containers with λ-Cntr and `kubectl debug`.

times the memory when compared to the ephemeral container created in the existing `busybox` pod. This is due to the volumes mounted in the λ-Cntr-pod and the `cntr attach` process running in the background.

5.2 Security

In this section we analyze the necessary privileges required by λ-Cntr in regards to security and present a threat model.

5.2.1 Kubernetes CIS Benchmark.

This benchmark, published by the Center for Internet Security (CIS) [29], specifies a set of best practices to ensure a secure Kubernetes environment. We checked λ-Cntr against the CIS Kubernetes v1.20 Benchmark v1.0.0. In the following we will address policies violated by λ-Cntr.

Minimize the admission of privileged container (5.2.1). Privileged containers are given access to all devices on the host, allowing them to do almost anything the host can. It is generally not recommend to run containers with the corresponding flag enabled and to define at least one *PodSecurity-Policy* prohibiting privileged containers.

Minimize the admission of containers wishing to share the host-process ID namespace (5.2.2). When given access to the hosts PID namespace, a container can read all processes running on the host. Again, it is generally not recommended to give containers this capability, and a *PodSecurityPolicy* should be in place preventing this.

Recommendation. λ-Cntr is required to run a container with both these privileges in order to find and attach to other containers. In such cases the CIS benchmark recommends to define a separate *PodSecurityPolicy* with minimal access permissions which allows for the creation of such a container.

5.2.2 Threat Model.

In a typical cloud deployment scenario we consider for λ-Cntr, a Kubernetes cluster is set up across multiple physical nodes with numerous services isolated into containers running on them. We include the hardware, the host OS, the Kubernetes deployment, as well as the container runtime in λ-Cntrs trusted computation base (TCB).
We can assume users of λ-Cntr already have access to the cluster and do not obtain any additional privileges through the application. It is notable that the λ-Cntr application itself does not need any special privileges in order to run, and the process started by Cntr within the container also runs with the privileges of the application. As future work we consider a stronger encapsulation of the attachment process to prevent users from getting access to the λ-Cntr-pod shell. Instead, developers would only get access to the shell opened by Cntr, this limits all privileges to those of the target container. In such a scenario we imagine administrative roles handling the deployment of privileged debugging pods and allowing developers to work on deployed containers without giving them direct access to the cluster.
This security optimization gives administrators more control over the system. As a consequence of splitting roles into cluster owner and λ-Cntr-user, potential malicious actors are introduced. Users might try to get access to the cluster through λ-Cntr, by breaking out of the Cntr-shell into the pod that is running with privileges. Preventing this should be the main goal in the implementation of such a feature.

5.3 Use cases

In this section we present possible real-world use-cases for λ-Cntr.

5.3.1 Debug shell

As we made clear in the motivation behind λ-Cntr serverless functions are notoriously hard to debug. λ-Cntr can easily be deployed in existing container-based environments and attach to pods running serverless functions. Developers can get shell access to functions and are able to explore their environment and filesystem with additional auxiliary tools. This is especially relevant for administrators troubleshooting issues in production, such as the debugging of configuration issues.

5.3.2 Remote development

With the ability to access a containers filesystem and to bring in additional tools, as described in the previous use-case, a more involved debugging approach is possible. By including tools such as `gdbserver` [16] in the λ-Cntr-container, developers can debug C/C++ code running in the remote container from a familiar local environment. Debuggers for other languages such as JPDA [23] for Java, Delve [12] for Golang or `debugpy` [11] for Python offer similar functionality. These debuggers can be tightly integrated into IDEs such as VS Code [50], enabling a powerful debugging setting and other development workflows for code running in an otherwise inaccessible production environment. With λ-Cntr this can be achieved at runtime, omitting the inclusion of debuggers at deployment-time and keeping production containers lean.

5.3.3 Network probing

Networking of distributed applications is done through numerous virtualization layers. Every pod creates its own network namespace that is shared between its containers. The communication between pods on the same node traverses the container integration bridge, while traffic between pods on separate hosts has to pass through an additional container overlay network, the host network and the infrastructure network. With NetworkPolicies networking within Kubernetes can get even more intricate. This complexity is troublesome when services or functions show networking issues and debugging is only possible on the host level. λ-Cntr enables network troubleshooting directly from within a pods network namespace and with any networking tool. Developers can inspect the network configuration from the pods perspective, monitor all its traffic and, if needed, generate test traffic between probe-points throughout the cluster.

Chapter 6

Related Work

6.1 Serverless Debugging

Current debugging approaches of serverless application are focused on combining logging, tracing and monitoring data, in order to troubleshoot unwanted behavior. This information can be obtained through systems like Grafana Jaeger [20], Amazon CloudWatch [4] or AWS X-Ray [5]. There exist numerous frameworks compile this information to provide developers with great insight into their distributed systems. Providers such as Thundra [48], Faasly [15], Dashbird [10] or Lumigo [35] offer comprehensive user interfaces, with key features such as distributed tracing, health monitoring and alerting mechanisms.
Although useful, workflows relying solely on these systems are often inefficient, as application have to be re-deployed and re-evaluated when problems occur in production. With λ-Cntr we can avoid this by providing a way to analyze and troubleshoot applications as they are running.

6.2 Ephemeral Containers

Ephemeral containers are a new kind of container introduced in Kubernetes. They run temporarily in an existing pod, by inserting a new container into its namespace. This enables the `kubectl debug` command which offers similar functionality to what we achieve with λ-Cntr. This feature is currently still *alpha* and is not recommended for use in a production environment.
Ephemeral containers come with some restrictions, as they are not intended for building services and do not fit within the normal pod lifecycle. They can only be used with pods once they've already been created, they cannot be restarted, it is not possible to reserve resources or specify a resource

configuration, and fields used for services such as port may not be specified. By default they do not share the process namespace of the target pod and have no access to its filesystem, restricting their use in many debugging use-cases. Kubernetes allows for process namespace sharing between containers of the same pod [28]. With this feature enabled, ephemeral containers get much more powerful, as they get insight into processes running in the target pod and can access their filesystem through `/proc/$pid/root`.
λ-Cntr offers a much closer integration to the container, providing direct access to the filesystem and adopting its environment. The debug container deployed by λ-Cntr offers additional advantage by being deployed in its own pod. It is not bound to a specific pod and can be reattached to any container on the same host, as well as being able to connect to multiple containers in parallel. In our evaluation of λ-Cntr (Chapter 5), we show that this comes at little to no cost of performance.

6.3 Virtualization

Containers used to provide minimal performance overheads at the cost of weaker security, when compared to traditional VMs. But when running untrusted code, as it is the case for cloud providers, the isolation provided by containers no longer suffices. Current approaches such as Google's gVisor [19] aim to overcome this tradeoff, by providing each container with its own application kernel. Other systems such as SAND [53] provide isolation by sandboxing on application-level. Further isolation can be achieved using hypervisor-based virtualization as implemented by Kata Containers [25] or LightVM [59]. This approach is improved by Amazon's Firecracker [52], a new virtualization technology with minimal overhead optimized for serverless and containerized workloads. Firecracker is adopted by vHive [67], an open-source framework for serverless experimentation that also introduces REAP, a mechanism to further improve cold-start delays with pre-fetching.
In the domain of VMs, tools such as *vmsh* [65] exist that are comparable to Cntr. With *vmsh*, developers can spawn debug containers with access to virtual machines. In a similar vein to λ-Cntr, *lambda-pirate* [64] utilizes this functionality in a serverless environment.

Chapter 7

Summary

We presented λ-Cntr, a system to debug and maintain lightweight production containers in a Kubernetes environment. λ-Cntr specifically targets serverless deployments, as they offer developers only very limited debugging capabilities and little insight into deployed functions. λ-Cntr builds on Cntr [63], a tool that enables containers to be dynamically expanded at run-time, by combining it with a debugging container. The troubleshooting capabilities of Cntr are brought into a Kubernetes context, enabling users to attach to any container within a cluster.
Comparable tools, such as ephemeral containers exist, but to the best of our knowledge, none of them adopt the context of the target container as complete as λ-Cntr. In our evaluation, we have extensively tested the performance of λ-Cntr, analysed possible security concerns and present possible real-world use-cases.
As future work, a tighter integration of λ-Cntr with serverless frameworks such as OpenFaaS [39] is possible. This would include functionality preventing functions from scaling down while attached with λ-Cntr. Another possible expansion is a stronger encapsulation of the debugging shell, to limit the privileges users gain through the privileged pod. Further the use of DaemonSets [30] as a deployment mechanism for λ-Cntr and the integration into other container orchestration platforms would be possible.
λ-Cntr is publicly available on GitHub [34] and the used Docker images are published on Docker Hub [32].

List of Figures

Bibliography

[1] Alpine Linux. https://alpinelinux.org/, 11/2021.

[2] Amazon AWS Lambda. https://aws.amazon.com/lambda, 11/2021.

[3] Amazon AWS SAM. https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/what-is-sam.html, 11/2021.

[4] Amazon CloudWatch. https://aws.amazon.com/cloudwatch/, 11/2021.

[5] Amazon X-Ray. https://aws.amazon.com/xray/, 11/2021.

[6] Apache Mesos. http://mesos.apache.org/, 11/2021.

[7] Apache OpenWhisk. https://openwhisk.apache.org/, 11/2021.

[8] busybox. https://github.com/kubernetes/kubernetes/blob/master/hack/testdata/recursive/pod/pod/busybox.yaml, 11/2021.

[9] clap crate. https://docs.rs/clap/, 11/2021.

[10] Dashbird. https://dashbird.io/, 11/2021.

[11] debugpy - a debugger for Python. https://github.com/microsoft/debugpy/, 11/2021.

[12] Delve. https://github.com/go-delve/delve, 11/2021.

[13] Docker Swarm. https://docs.docker.com/engine/swarm/, 11/2021.

[14] faas-netes - Kubernetes controller for OpenFaaS. https://github.com/openfaas/faas-netes/, 11/2021.

[15] Faasly. https://faasly.io/, 11/2021.

[16] gdbserver. https://www.man7.org/linux/man-pages/man1/gdbserver.1.html, 11/2021.

[17] git. https://www.git-scm.com/, 11/2021.

[18] Google Cloud Functions. https://cloud.google.com/functions, 11/2021.

[19] Google gVisor. https://github.com/google/gvisor, 11/2021.

[20] Grafana Jaeger. https://grafana.com/docs/grafana/latest/datasources/jaeger/, 11/2021.

[21] htop. https://htop.dev/, 11/2021.

[22] iperf. https://sourceforge.net/projects/iperf2/, 11/2021.

[23] Java Platform Debugger Architecture (JPDA). https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/, 11/2021.

[24] k3s. https://k3s.io/, 11/2021.

[25] Kata Containers. https://katacontainers.io/, 11/2021.

[26] kube-rs crate. https://docs.rs/kube/, 11/2021.

[27] Kubernetes. https://kubernetes.io/, 11/2021.

[28] Kubernetes - Share Process Namespace between Containers in a Pod. https://kubernetes.io/docs/tasks/configure-pod-container/share-process-namespace/, 11/2021.

[29] Kubernetes CIS Benchmark. https://www.cisecurity.org/benchmark/kubernetes/, 11/2021.

[30] Kubernetes DaemonSet. https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/, 11/2021.

[31] Kubernetes Metrics Server. https://github.com/kubernetes-sigs/metrics-server, 11/2021.

[32] Lambda-cntr docker hub. https://hub.docker.com/repository/docker/onestone070/lambda-cntr/general, 11/2021.

[33] Lambda-cntr Dockerfile. https://github.com/EduardvonBriesen/Lambda-Cntr/blob/evaluate/Dockerfile, 11/2021.

[34] Lambda-cntr github. `https://github.com/EduardvonBriesen/Lambda-Cntr`, 11/2021.

[35] Lumigo. `https://lumigo.io/`, 11/2021.

[36] Mircosoft Azure. `https://azure.microsoft.com/`, 11/2021.

[37] NATS - The Cloud Native Messaging System. `https://github.com/nats-io`, 11/2021.

[38] Netcat-openbsd. `https://packages.debian.org/sid/netcat-openbsd`, 11/2021.

[39] OpenFaaS. `https://www.openfaas.com/`, 11/2021.

[40] OpenFaaS Watchdog. `https://docs.openfaas.com/architecture/watchdog/`, 11/2021.

[41] Phoronix test suite. `https://www.phoronix-test-suite.com/`, 11/2021.

[42] Prometheus. `https://prometheus.io/`, 11/2021.

[43] Prometheus AlertManager. `https://prometheus.io/docs/alerting/latest/overview/`, 11/2021.

[44] Rancher Cattle. `https://github.com/rancher/cattle`, 11/2021.

[45] Raw benchmark report generated by phoronix test suite. `https://openbenchmarking.org/result/2110140-IB-2110146IB83`, 11/2021.

[46] Snappy. `https://github.com/google/snappy`, 11/2021.

[47] strace. `https://strace.io/`, 11/2021.

[48] Thundra. `https://www.thundra.io/`, 11/2021.

[49] Vim. `https://www.vim.org/`, 11/2021.

[50] VS Code. `https://code.visualstudio.com/`, 11/2021.

[51] G. Aceto, A. Botta, W. de Donato, and A. Pescapè. Cloud monitoring: Definitions, issues and future directions. In *2012 IEEE 1st International Conference on Cloud Networking (CLOUDNET)*, pages 63–67, 2012.

[52] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, 02/2020. USENIX Association.

[53] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, 07/2018. USENIX Association.

[54] I. Baldini, P. C. Castro, K. S. Chang, P. Cheng, S. J. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter. Serverless computing: Current trends and open problems. *CoRR*, abs/1706.03178, 2017.

[55] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, and A. Palopoli. Container orchestration engines: A thorough functional and performance comparison. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pages 1–6, 2019.

[56] V. Lenarduzzi and A. Panichella. Serverless testing: Tool vendors' and experts' points of view. *IEEE Software*, 38(1):54–60, 2021.

[57] W.-T. Lin, C. Krintz, and R. Wolski. Tracing function dependencies across clouds. *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 253–260, 2018.

[58] W.-T. Lin, C. Krintz, R. Wolski, M. Zhang, X. Cai, T. Li, and W. Xu. Tracking causal order in aws lambda applications. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 50–60, 2018.

[59] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.

[60] J. Manner, S. Kolb, and G. Wirtz. Troubleshooting serverless functions: a combined monitoring and debugging approach. *SICS Software-Intensive Cyber-Physical Systems*, 34(2):99–104, 06/2019.

[61] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[62] D. Sun, M. Fu, L. Zhu, G. Li, and Q. Lu. Non-intrusive anomaly detection with streaming performance metrics and logs for devops in public clouds: A case study in aws. *IEEE Transactions on Emerging Topics in Computing*, 4(2):278–289, 2016.

[63] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 199–212, Boston, MA, 07/2018. USENIX Association.

[64] J. Thalheim and P. Okelmann. lambda-pirate. `https://github.com/pogobanane/lambda-pirate`, 11/2021.

[65] J. Thalheim and P. Okelmann. vmsh. `https://github.com/Mic92/vmsh`, 11/2021.

[66] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer. Sieve: Actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, page 14–27, New York, NY, USA, 2017. Association for Computing Machinery.

[67] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery.

Eduard von Briesen