Vitamin-V: Serverless Cloud Software Porting Progress

Thrasyvoulos Iliadis, Nikolaos C. Papadopoulos, Kostantinos Nikas, and Dionisios Pnevmatikatos

Institute of Communication and Computer Systems (ICCS) &
National Technical University of Athens, Athens 15780, Greece
filiadis@cslab.ece.ntua.gr

Abstract. The Vitamin-V project, funded by the Horizon Europe program, aims to develop an open-source full RISC-V software stack for cloud services while matching the performance of x86-based infrastructures. This paper presents our progress towards this goal, consisting of the porting of kata-containers and the establishment of a cloud software stack on x86, to be used for baseline measurements with plans to run on RISC-V. Kata Containers, known for combining the security advantages of virtual machines with the speed of containers, has been successfully ported on RISC-V. Our serverless stack that incorporates FunctionBench, KNative, Kubernetes, Kata Containers and QEMU or Firecracker for benchmarking serverless workloads has succesfully been setup, tested and measured on x86. This setup enabled us to collect initial performance metrics and validate the functionality of the stack in a mature environment.

1 Introduction

The Vitamin-V [1] project addresses the significant challenge of achieving isoperformance with RISC-V for cloud services in an industry dominated by x86 architectures. This project focuses on overcoming limitations in RISC-V [2] environments—such as the lack of hardware support for essential features like virtualization, cryptography, and vectorization—by extending support across three virtual platforms: QEMU [3], gem5, and FPGA prototypes. In addition, Vitamin-V aims to enhance RISC-V toolchain and integrate state-of-the-art cloud management and virtualization software, thereby enabling software development, validation, and benchmarking for classical, modern, and serverless cloud setups, which is where this work mostly focuses.

2 Kata-Containers

Traditional virtual machines (VMs) provide robust isolation by running independent operating systems. Containers, in contrast, share a common kernel and

leverage features such as cgroups and namespaces. Additionally, serverless computing abstracts infrastructure management by automatically scaling code execution without user intervention. Kata Containers represent a hybrid approach in which each container is encapsulated within a lightweight virtual machine. As part of the EU-funded VITAMIN-V project, our initiative focused on extending this architecture to the RISC-V platform by leveraging existing similarities with arm64 configurations.

2.1 Implementation and Testing

The adaptation of Kata Containers to the RISC-V architecture required significant modifications across several core components. Central to the project were three critical elements: the container runtime, the kata agent, and the kata monitor, all of which depend on the `virtcontainers` library. This library contains the architecture-specific and hypervisor-specific code necessary for the operation of Kata Containers and originally supports architectures such as amd64, arm64, ppc64le, and s390x.

The porting of Kata Containers to riscv64 extends the platform's ability to create isolated virtual machines for container workloads by adapting both the guest OS and the root filesystem creation tools to RISC-V. This work involves modifying the processes that build guest disk images and initrds, while meeting the requirements of the Kata Containers runtime. In this effort, we have also made several pull requests as ICCS, contributing to the upstream development and engaging with the community to refine the changes. The adaptation includes integrating the Kata Containers agent—tasked with initializing the containerized environment during boot—into the guest OS startup process, effectively leveraging system managers such as systemd to invoke the agent. By aligning the terminology and operational workflow—spanning rootfs, guest OS (or guest image), and initrd creation—with existing practices, the riscv64 port maintains the core objectives of robust isolation and efficient container deployment.

Key aspects of the porting process include:

- **Architecture-Specific Modifications:** Integration of RISC-V-specific code into the `virtcontainers` library to tailor parameters such as hypervisor options, virtual CPU configurations, and kernel parameters. These changes ensure that the RISC-V target leverages the intrinsic performance and security attributes of Kata Containers.
- **Toolchain and Environment Migration:** Initial development and compilation were carried out on a Debian-based system, which involved manually compiling essential components. However, due to dependency challenges, the environment was migrated to Ubuntu 24.04, which offered improved support and streamlined dependency management.
- **Platform Adaptation:** Testing was primarily conducted on the StarFive VisionFive 2 board running Ubuntu 24.04. Given the current lack of support for the RISC-V H-extension on the board, additional testing was performed using QEMU emulation on an x86_64 host system. This dual approach allowed for thorough evaluation despite hardware limitations.

Through iterative compilation, debugging, and configuration refinements tailored to the RISC-V architecture, all major components were successfully integrated and validated. This process laid the groundwork for further enhancements and eventual full support on native hardware.

2.2 Results and Future Outlook

The porting of Kata Containers to RISC-V has demonstrated a viable blend of performance and security. Encapsulating each container within a lightweight virtual machine maintains robust isolation akin to traditional VMs, while the containerized approach preserves near-native performance. Early testing suggests that the solution offers significant improvements over conventional VM startup times and provides the necessary security assurances for multi-tenant environments.

Key project outcomes include:

- **Optimized Performance and Security Trade-offs:** The hybrid model effectively leverages the speed of containerization and the robust isolation of virtual machines, addressing the limitations inherent in each approach when used independently.
- **Community and Ecosystem Impact:** Active contributions, such as extending support for the kata agent to the RISC-V architecture, have engaged the open-source community. This collaborative effort is expected to promote broader adoption and facilitate continued innovation in secure containerization.
- **Addressing Practical Challenges:** The migration from Debian to Ubuntu 24.04 simplified the development process by mitigating dependency issues, and iterative problem solving resolved compilation errors and architecture-specific challenges. The use of QEMU emulation has enabled comprehensive testing until native support becomes available.

Looking forward, future work will focus on expanding testing to include RISC-V hardware with full virtualization support. Further optimization of hypervisor configurations and refinement of RISC-V-specific parameters will continue to enhance performance. Additionally, extending integration with container orchestration platforms such as Kubernetes remains a priority to ensure scalable deployment in diverse environments.

In summary, this work marks a significant milestone in merging the agility of containers with the secure isolation of virtual machines on next-generation hardware architectures. It lays a strong foundation for the development of secure, scalable, and efficient containerized environments, addressing the evolving needs of cloud and multi-tenant computing infrastructures.

3 FunctionBench

Within VITAMIN-V, ICCS has been working on a serverless cloud use case, and a demonstrator that aims to validate comprehensive end-to-end execution

scenarios utilizing the modern paradigm of serverless computing. The project deploys a full FaaS software stack to verify functional correctness and to benchmark performance through serverless workloads. For this evaluation, we use the FunctionBench [4] suite. This suite offers a fine-grained assessment of resource performance as well as a more complete view of running applications under serverless conditions. Porting of FunctionBench to RISC-V is detailed in [5].

3.1 Requirements and Development Status

For the FunctionBench evaluation, the complete software stack has been implemented to operate on both RISC-V and x86 architectures. A schematic of the software stack, which can be deployed using either QEMU or Firecracker [6], is provided in Figure 1. Detailed status tables (Tables 1 and 2) summarize the progress of each component within the software stack.

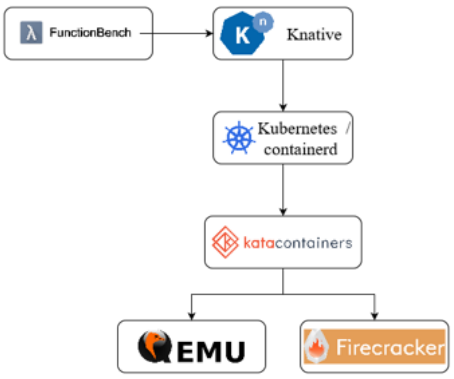

Fig. 1: Complete Software Stack for a serverless use case (using either QEMU or Firecracker)

Software Component	Status
FunctionBench: Benchmark Code	Completed
FunctionBench: Knative Support	Completed
Kubernetes	Completed
Kata Containers	Under Testing
QEMU	Completed
Firecracker	Work In Progress

Table 1: Status of all components needed for RISC-V software stack

Software Component	Status
FunctionBench: Benchmark Code	Completed
FunctionBench: KNative Support	Completed
Kubernetes/containerd	Completed
Kata Containers	Completed
QEMU	Completed
Firecracker	Completed

Table 2: Status of all components needed for x86 software stack

FunctionBench itself includes a range of benchmarks and is available in three variants tailored for Google, Amazon, or Azure. We adopted the version designed for Google Cloud due to its streamlined codebase and the availability of deployment resources. The revised benchmarks have been successfully tested on a RISC-V system simulated via QEMU (Ubuntu 23.10 with kernel version 6.5.0-40-generic). Originally, FunctionBench benchmarks were configured to operate with Google Cloud Storage through the functions-framework. In order to facilitate more flexible local testing without reliance on cloud-based resources, we adapted the benchmarks to run locally using Python Flask. This dual-mode configuration allows us to execute the same benchmarks either in the cloud or in a local environment, enhancing their portability and ease of evaluation. For local testing we used local buckets working with MinIO [7] cloud storage.

The original design of the FunctionBench benchmarks requires integration with Google Cloud Storage. This requires a valid Google account, credits, and the necessary configurations. While Google's functions-framework provides an alternative for local testing, it still depends on Google-specific packages, which may not be readily available for RISC-V ones and includes a dependency on a google product. To address these limitations, we modified the benchmarks so that they could also run locally using Python Flask [8]. By eliminating the dependency on any online storage solution, this local configuration simplifies testing and evaluation. Consequently, we have developed two versions for each benchmark: one designed for operation with Google Cloud Storage (or functions-framework) and another that executes in a local environment using Python Flask.

3.2 Evaluation

In order to evaluate and compare the performance of the ported RISC-V software stack with that of other architectures, we deemed it necessary to use an identical experimental setup across platforms. To use for a fair comparison, initial measurements have been collected from x86 systems. Table 3 provides a detailed description of the machine used for these tests, while Figures 2, 3, 4 and 5 illustrate the results obtained. Note the logarithmic scale on the Y-Axis in each graph.

A methodology was employed to capture both warm and cold execution times for every benchmark. Cold execution time was measured by including the full

system startup, meaning that the container was deployed from scratch before generating a response. Warm execution time was recorded as the period required to execute a function once the container was already running. For each benchmark, we took 100 measurements under both conditions: for warm times, the container was initialized once followed by 100 sequential requests, whereas for cold times, a single request was made, the container was then terminated, and this process was repeated 100 times.

The experiments tracked both durations and latencies. Durations are the duration of the function execution just on the server side, and latencies are the total latency experienced by the client, so is an end-to-end metric.

Feature	**Details**
Name	icy7
Architecture	x86
CPU	4 x Intel Xenon Silver 4314 (16 cores)
Cores	64 total (4 x 16)
Frequency	2.40 GHz
OS	Debian 11 (bullseye)
Docker	version 20.10.5+dfsg1, build 55c4c88
Kubernetes	v1.30.2
Kustomize	v5.0.4-0.20230601165947-6ce0bf390ce3

Table 3: Details of machine used for initial x86 measurements

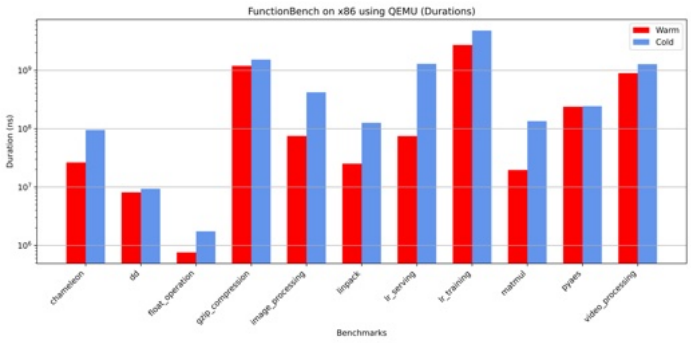

Fig. 2: FunctionBench: Warm and Cold duration time measurements using QEMU

Figures 2 and 3 describe the results using QEMU. From figure 2, we see that simple benchmarks consistently have the shortest durations in both cold and warm states, expected due to their minimal overhead (e.g. dd, chameleon). More complex benchmarks such as lr_training - lr_serving show higher durations as expected. The latency figure 3 reinforces this view. Cold runs have consistent latencies, while warm runs vary a bit. However, the general trend of execution

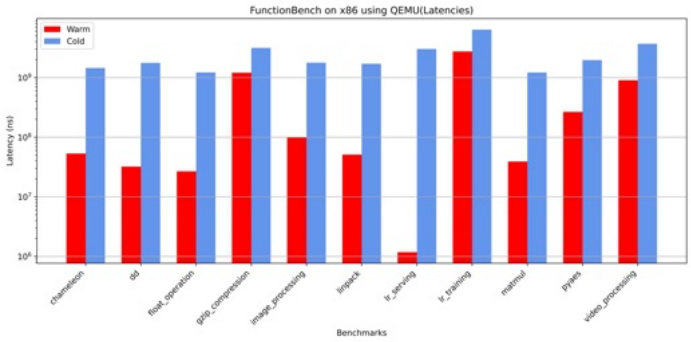

Fig. 3: FunctionBench: Warm and Cold latency time measurements using QEMU

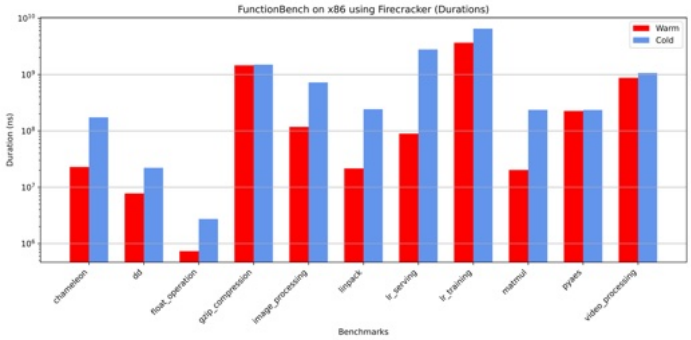

Fig. 4: FunctionBench: Warm and Cold duration time measurements using Firecracker

times from the durations chart applies here too. We can see this by looking at the warm times for the quickest benchmarks for the durations (2), and noticing that these are still among the quickest here - chameleon, dd, etc. A notable exception is the lr_serving model, which could be explained by the nature of the benchmark itself. Despite these results only being initial x86 measurements, we can already observe that the execution times are heavily affected by the nature of the running benchmark, especially when measuring actual server-side durations. The same applies when using Firecracker (figures 4-5). We therefore observe that both metrics show a similar behavior with both deployments, an observation which remains to be further evaluated once RISC-V results are available.

4 Future Work

As part of our future objectives, we aim to deploy the complete serverless cloud software stack on RISC-V platforms. This will involve thorough evaluations on both RISC-V FPGA prototypes and available hardware to assess performance

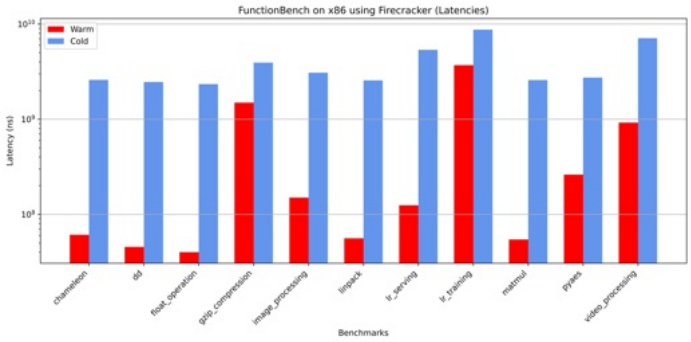

Fig. 5: FunctionBench: Warm and Cold latency time measurements using Firecracker

and compatibility. However, the limited availability of RISC-V hardware supporting the Hypervisor extension presents a challenge. To mitigate this, initial development is being conducted using QEMU-based virtual environments, allowing us to emulate necessary hardware features and validate the code's functionality on RISC-V. These efforts are an obvious next step in advancing RISC-V's capabilities to meet the demands of modern cloud infrastructures.

Acknowledgement

This work was supported by project Vitamin-V (Project number: 101093062) funded by the European Union. Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union or the HaDEA. Neither the European Union nor the granting authority can be held responsible for them.

References

1. R. Canal *et al.*, "Vitamin-v: Virtual environment and tool-boxing for trustworthy development of risc-v based cloud services," in *2023 26th Euromicro Conference on Digital System Design (DSD)*, 2023.
2. A. Waterman *et al.*, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. RISC-V International, 2024. Accessed: 2025-04-16.
3. "QEMU: A generic and open source machine emulator and virtualizer."
4. J. Kim *et al.*, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 502–504, 2019.
5. T. Iliadis *et al.*, "Vitamin-v: Serverless cloud computing porting on risc-v," in *SAMOS 2024 Proceedings, Part II*, 2024.
6. A. Agache *et al.*, "Firecracker: Lightweight virtualization for serverless applications," in *Conference on Virtual Execution Environments*, ACM, 2020.
7. MinIO, Inc., "Minio: High performance object storage." `https://min.io`, 2025.
8. "Python-flask documentation." `https://flask.palletsprojects.com/en/stable/`, 2010.