FACULTY OF ENGINEERING AND ARCHITECTURE

Optimising memory usage of Kubernetes operators using WebAssembly

Tim Ramlot
Student number: 01704656

Supervisors: Prof. dr. Bruno Volckaert, Prof. dr. ir. Filip De Turck
Counsellors: ing. Merlijn Sebrechts, ing. Sander Borny

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2021-2022

GHENT UNIVERSITY

FACULTY OF ENGINEERING AND ARCHITECTURE

Optimising memory usage of Kubernetes operators using WebAssembly

Tim Ramlot
Student number: 01704656

Supervisors: Prof. dr. Bruno Volckaert, Prof. dr. ir. Filip De Turck
Counsellors: ing. Merlijn Sebrechts, ing. Sander Borny

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2021-2022

GHENT UNIVERSITY

Permission of use on loan

Acknowledgements

This master's dissertation concludes a five year computer science engineering programme at Ghent University. This dissertation has helped me deepen my knowledge on Kubernetes and WebAssembly and aims to provide the interested readers with new insights. During my studies, Kubernetes has become my favourite subject, thanks to a couple of interesting projects and a fantastic internship.

I would like to thank ing. Merlijn Sebrechts and ing. Sander Borny for their guidance, feedback and the time they invested in me and my study.
Additionally, I thank my supervisors Prof. dr. Bruno Volckaert and Prof. dr. ir. Filip De Turck.

I am grateful to my friends and family who have helped me by providing much-needed relaxation and distraction.

My parents and brother have supported me throughout my studies where necessary, for which I am extremely grateful. They also reviewed and proofread the dissertation, which helped me a lot.

Tim Ramlot
June 2022

Explanatory note

This master's dissertation is part of an exam. Any comments formulated by the assessment committee during the oral presentation of the master's dissertation are not included in this text.

Abstract

"A new Kubernetes operator architecture, based on WebAssembly, can reduce the control plane memory overhead and makes the orchestrator a better fit for low-resource environments while offering a cost-reduction for existing systems."
High-resource clusters are the primary orchestration target of Kubernetes. Running Kubernetes on low-resource clusters suffers from relatively high control plane operator overhead costs. To capture the interest of low-resource fog, edge and IoT market segments, it is vital to reduce these overhead costs. Complex Kubernetes deployments greatly benefit from the extensibility of the Kubernetes orchestrator. However, this extensibility is one of the main cost drivers of the Kubernetes control plane architecture. In this traditional architecture, each operator runs in a separate pod, leaving many potential optimizations on the table.

This master's dissertation presents a WebAssembly-based operator architecture that allows sharing the low-level operator logic and offers scale-to-zero functionality, based on advancements used by the newest edge serverless platforms. It enables operators to be unloaded, while watching for Kubernetes API events and reloading the operators when required.

Using our proposed WebAssembly operators instead of comparable container-based operators resulted in a significant 64% reduction in memory usage for benchmarking 100 synthetic operators. When idle, the WebAssembly operators used 17% less memory than when active. This idle memory usage was further reduced by 57% by unloading the WebAssembly module. However, this should be done infrequently, as reloading modules comes with a severe latency and memory usage cost.

Keywords: Kubernetes, edge, operators, WebAssembly, WASI

Optimising memory usage of Kubernetes operators using WebAssembly

Tim Ramlot
Supervisors: Prof. dr. Bruno Volckaert, Prof. dr. ir. Filip De Turck
Counsellors: ing. Merlijn Sebrechts, ing. Sander Borny

***Abstract*—High-resource clusters are the primary orchestration target of Kubernetes. Running Kubernetes on low-resource clusters suffers from relatively high control plane operator overhead costs. To capture the interest of low-resource fog, edge and IoT market segments, it is vital to reduce these overhead costs. This paper presents a WebAssembly-based operator solution, that achieves more light-weight isolation compared to container-based operator architectures. Using our proposed WebAssembly operators instead of comparable container-based operators resulted in a significant 64% reduction in memory usage for benchmarking 100 synthetic operators. Some additional improvements are possible in best-case scenarios.**

***Index Terms*—Kubernetes, edge, operators, WASM, WASI**

I. Introduction

The adoption of new technologies relies heavily on potential resulting cost savings. Monetary and environmental cost reductions have been achieved by increasing computational density in cloud computing. Lower latency costs have stemmed from advancements in telecommunications, such as 5G, and the dispersion of cloud to fog, edge and IoT. These two innovation flows have independently led to widely accepted solutions. Here lies an opportunity for cross-pollination between projects. Cloud computing utilizes cloud orchestration for optimal resource allocation and server cluster management. Kubernetes [1], which originated from Google Borg [2, 3], is such an extensible cluster orchestrator. Its extensible architecture parts a cluster in a control plane and a worker plane. Edge environments, which are typically large complex clusters with limited resources, can benefit significantly from having an orchestrator to manage complexity and size.

However, the primary orchestration targets of Kubernetes are high-resource clusters. Running Kubernetes on low-resource clusters suffers from relatively high control plane operator overhead costs, which hinders adaptation in the edge market segment. In complex cloud-native application deployments, operators [4] are used to automate actions on the Kubernetes cluster state, that would otherwise be performed by a human operator. These operators are one of the main cost drivers of the Kubernetes control plane. To react to changes in the Kubernetes cluster state, the operators have to run as long-living processes. Even if the operator's control loop is idle, the container and process still use cluster resources. For complex applications that use many operators, these overhead costs quickly accumulate and account for a significant portion of the resource utilization. This is especially problematic for low-resource deployments.

Since global edge application configuration and deployment is a complex task, it is often abstracted by the service provider and included in a Function as a Service (FaaS) offering. FaaS applications are better suited for low-resource edge environments thanks to their fast on-demand scaling properties. Specifically, some edge FaaS platforms use WebAssembly (WASM), a browser technology designed as a portable binary code format that can be assembled from a range of programming languages and that is well suited to resource-constrained environments. On edge FaaS platforms, like Cloudflare Workers [5] and Fastly Compute@Edge [6], WebAssembly is used to securely isolate workloads with reduced overhead and scale-to-zero capabilities.

Chapter II explains the architecture of our WebAssembly-based operator solution and Chapter III discusses how this architecture is implemented. In Section IV, our benchmark results of the WebAssembly runtime are discussed, as well as the methodology to achieve these results.

II. Solution architecture: WASM operator

Because of Kubernetes' can-always-fail design, an operator application is not supposed to hold any internal state across reconciliation iterations except for caches. The operator generally uses the Kubernetes API to store state. This theoretically should allow running each iteration of the reconciliation loop without storing state in between. This property also holds for FaaS systems, where there is no guarantee for state preservation in between function calls. FaaS solutions for constrained edge environments often utilize Software-based Fault Isolation (SFI) instead of process isolation. WebAssembly lets you create SFI applications based on code written in existing high-level languages.

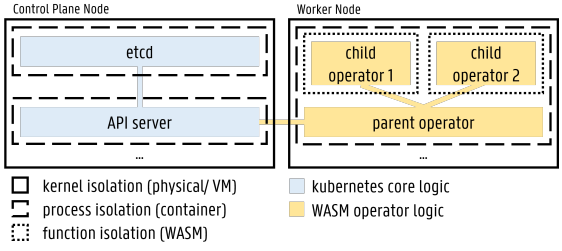


Fig. 1. Design of our WASM operator architecture.

The *WASM operator architecture* presented in this paper and visualized in Figure 1 attempts to realize all the beneficial aspects that solutions in prior work have achieved [7, 8, 9]. The main parts of the architecture are the parent operator and the child operators. The child operators run as WASM instances in the WASM runtime embedded in the parent operator. We use an existing WASM runtime implementation as embedded runtime. The beneficial aspects are listed below.

- **Isolation overhead:** All child operators run in the same process as the parent operator. This process runs inside a single container in a single Kubernetes pod. Isolation is provided by the WASM engine, eliminating the overhead due to container isolation.
- **Modularity:** The WASM runtime makes it possible to add or remove child operators without interfering with the other active child operators.
- **Simple child operator:** In our architecture, the parent operator extends the WASM runtime with host functions that can be used by the child operators to communicate with the Kubernetes API. Low-level operator logic is moved to the parent operator. This reduces the complexity and overhead of the client operators.
- **Scale-to-zero:** To limit the overhead of inactive operators, our architecture allows to dynamically unload inactive operators.

In order to efficiently make Kubernetes API requests, we want child operators to perform them asynchronously. Existing WASM runtimes offer no support for asynchronous calls or offer a solution that is incompatible with idle module unloading. Therefore, we created a new solution that adds support for asynchronous operations to the WASM runtime that is embedded in our parent operator. By extending the WASM runtime, we allow the child operators to wait for host functions asynchronously.

A. Parent operator asynchronous runtime

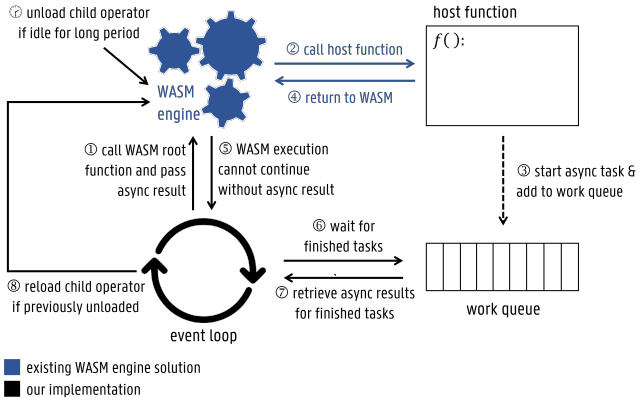


Fig. 2. The design of the parent operator incorporates a WASM runtime event loop which repeatedly performs actions 1-8; making it possible to asynchronously call host functions from within a WASM instance.

Figure 2 shows how the parent operator manages the asynchronous operations of a child operator and unloads an inactive child operator after a long period of inactivity. The main components of the parent operator are the WASM engine, the host functions exposed to the WASM instance and the work queue. For the WASM engine component and some of the host functions, existing solutions can be used.

The solution works as follows: Each WASM module has an entry point that executes the main function in the child operator, shown in Figure 2 as ①. Environment interactions happen through the calling of WASM host functions ②. Some of these actions are asynchronous and do not directly yield a result. These asynchronous actions are started and added to the work queue ③, directly returning control to the WASM module ④. After executing all synchronous logic, the WASM execution stops and the control is returned to the event loop ⑤. This loop checks if any of the actions in the work queue finished ⑥ and passes the results of that finished action ⑦ back to the WASM engine ①, reloading the child operator in case it had been previously unloaded ⑧. When returning to WASM, a new set of synchronous actions are performed by the engine. Long-running operators repeat this process indefinitely. These operators are always waiting for new asynchronous inputs, like events in a watch stream.

The runtime might detect that a certain child operator has not been receiving any asynchronous results over a long period (marked as 🕒). This is indicative for an operator that reached a steady state in its reconciliation process. Most likely, it will only restart its logic after external applications changed the state of the Kubernetes resources that it manages. This could mean that the operator remains idle for multiple hours. In such cases, it can be more resource-efficient to unload and swap the WASM instance to disk.

B. Child operator asynchronous client

All client operators run as single-threaded asynchronous WASM instances. The child operator is started by the host which calls the `start` function that is exposed by the WASM module. This initial function starts the operator reconciliation loop, which makes asynchronous requests. These asynchronous futures [10] are `awaited` by the child operator, but some of these futures `await` asynchronous host function results from the parent runtime. If the child operator cannot continue without new results from the host environment, it stops the execution and returns to the host. If none of the pending asynchronous requests have finished already, the host waits for one of them to finish, as described in Section II-A. Once a request finishes, the host returns the result to the child operator, such that the child operator can finish the linked asynchronous request. This restarts the whole process.

III. Implementation

The latest version of our implementation and the scripts used for the end-to-end tests can be found on Github [11].

A. Prior work

Our operator implementation builds upon the proof of concept (PoC) made by Francesco Guardiani and Markus

Thömmes [12]. This PoC provides a WASM operator solution based on the Wasmer [13] WASM runtime and a hacked version of the kube-rs [14] library. However, at the time of writing, it has been 2 years since this project was updated. Since the API of the Wasmer runtime drastically changed after its v1 release, and the hacks applied to the kube-rs project are not well documented, upgrading the PoC was not straight forward. Furthermore, the Wasmer project lacks the future potential that other open-source initiatives, like Wasmtime, can offer. To update kube-rs more easily in the future, a new project structure was required. Moreover, the original version of the PoC cannot unload inactive operators as its architecture is different from the architecture proposed in Section II. We refactored the PoC and updated it to implement the aforementioned architecture. Finally, we implemented several improvements to further optimize the PoC implementation, such as adding support for caching compiled WASM modules for later reuse.

B. Parent operator: WASM runtime

The parent operator extends the Wasmtime WASM runtime. Wasmtime was chosen over other WASM runtimes, because it is the flagship WASM engine from the Bytecode Alliance, with support from some of the biggest players in the technology industry. Our implementation configures Wasmtime to compile ahead of time (AOT) new WASM modules to machine code to eliminate the compiler memory overhead at runtime. These compiled modules are cached on disk and can be reused when possible. To initiate these compiled modules, Wasmtime only has to map the file to memory and provide the necessary tools to communicate with this initiated module. Because of the use of file-backed memory, for idle operators, these memory locations can be dropped from memory by the kernel when needed. If the memory region needs to be accessed again, a page-fault will be triggered, and the kernel will load the file back into memory. However, the dynamically populated memory of the WASM module will not be unloaded automatically from memory. That is why our implementation adds a custom unloading and disk swapping implementation in the parent operator. This makes unloading and swapping possible, even on systems without swap enabled at operating system level.

C. Parent operator: host functions

WASM host functions are functions exposed by the WebAssembly runtime to the WASM instances. The Web Assembly System Interface (WASI) is a standardised set of these host functions. Our implementation can benefit from the existing Wasmtime library that readily implements these WASI host functions, reducing the implementation and maintenance burden of our solution. A core aspect of Kubernetes operators is communicating with the Kubernetes API server. However, at the time of writing, the WASI spec has not yet standardised sockets as part of the interface [15]. This means that for our implementation, we had to implement custom HTTP host functions to create a working WASM operator setup. As shown in Figure 3, our implementation uses the low-level part of kube-rs for the Kubernetes host function implementation. The high-level kube-rs functionality is implemented in the child operator. The added host functions are asynchronous functions, meaning that they return control to the WASM module immediately, while returning an `async_id` that references a task in the workqueue as described in Section II-A.

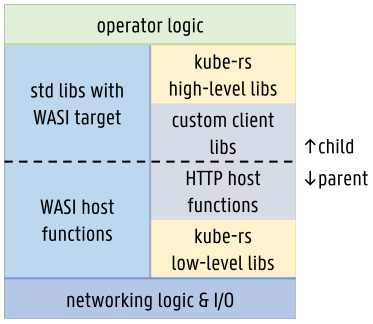


Fig. 3. The operator libraries are split between the parent and child operator and consist out of existing WASI libraries and our own custom libraries based on kube-rs.

D. Child operator: client libraries

All Kubernetes operator domain knowledge is implemented in the custom reconciliation loops, that are defined in the child operators. Our language preference for the child operators is Rust since the Rust standard libraries best support the WASI host function calls. Golang, which is normally used in Kubernetes, has no support for WASI in its default compiler. Additionally, Golang is a garbage collected language, which have been shown to use more memory [16]. Another advantage of choosing Rust as language is that an easier interoperability between the parent and child operator is obtained. The (de)serialisation logic mentioned in Section III-C, can be reused for both the parent and child, since they are both implemented in Rust.

IV. Resource utilisation

A. Test setup

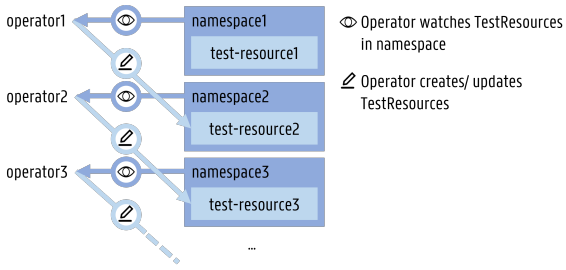
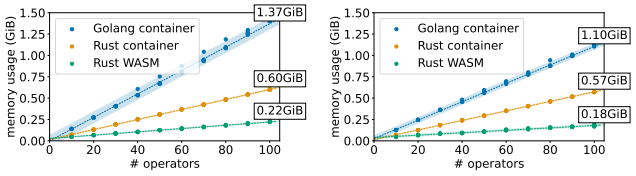


Fig. 4. The test setup for the synthetic-operator workload, each operator is responsible for the propagation of changes from one namespace to another.

The test synthetic-operator, as shown in Figure 4, simulates a workload with N different operators, which depend on each other's actions and are idle for most of the time. Each



(a) active (b) idle

Fig. 5. The memory 95% upper bounds of the different languages/ isolation techniques are ordered as follows: Rust WASM < Rust container < Golang container; all operators use less memory when idle.

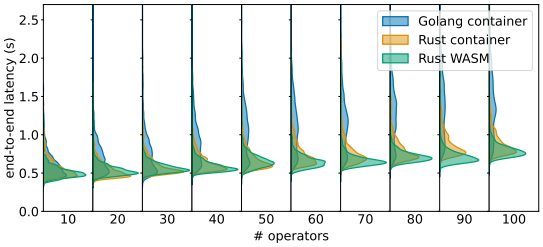


Fig. 6. The end-to-end latency of WASM operators is identical to Rust operators.

operator watches a namespace for `TestResources` and only reconciles, once a resource is created or updated. It then updates/ creates the resource in its destination namespace. For a full update of all resources, all operators must update their resource one-by-one. This means the full end-to-end latency equals the accumulated individual operator latencies. This synthetic workload simulates a highly dependent and interactive operator setup.

Measuring the memory footprint of a workload execution, requires accounting all the memory usage effects that the process has on the system. This is a non-trivial problem. The memory utilization measure that we use is determined by limiting the memory usage, as determined by cgroup v2 [17], until the application is being slowed down as determined by the Linux PSI metric [18]. For each run, we determine a upper bound memory limit. Each upper bound is defined as the memory limit that is not exceeded for 95% of the selected time range duration. For each configuration, which is defined by an operator type and number of operators, five independent runs were performed, each yielding one upper bound for the active and one for the idle period. Per operator type, we tested the number of operators from 10 to 100, in increments of 10. For the active and idle selection separate, based on the resulting 50 upper bounds for each operator type, we trained a linear regression model. Using this linear model, we determined the 95% prediction interval in which we expect with 95% certainty the upper bound memory usage of a new run with the given configuration, as described by Neter et al. [19].

The end-to-end latency is measured by the synthetic-operator test for the active period of the test. Each set of reconciliations starts from an update of the `TestResource` in namespace 1 until the `TestResource` in namespace N is updated. The time from start to end is measured and each reconciliation set is repeated 500 times per run, resulting in $500N$ reconciliation iterations. As described in Section IV-A, for each configuration, which is defined by an operator type and a number of operators, five independent runs are performed.

B. Golang container, Rust container and Rust WASM compared

Figure 5 shows the obtained memory upper bounds for container-isolated operators written in Golang and Rust and a WASM-isolated Rust operator. The coloured areas represent the 95% prediction intervals for the regression models as described in Section IV-A. Figure 5a shows the results for the active period. The Golang-based operator clearly uses the most memory. For 100 active operators, switching from Golang to Rust resulted in a 56.06% upper bound memory reduction. WASM operators even yielded an 83.81% reduction compared to Golang operators. Compared to Golang, the Rust operators use entirely different operator library and framework implementations. Each implementation has its own memory trade-offs, which can lead to large differences in memory usage. Additionally, as discussed in Section IV-A, garbage collected languages like Golang, typically are less memory efficient than languages without garbage collector like Rust. The Rust container-based operator and the WASM-based operator share much of their source code. However, the WASM-based operators use less memory than container-based operators. This is due to the reduced complexity of the WASM child operator, as much of its low-level operator logic is moved to the parent operator. Moreover, the different isolation techniques used result in a net reduced isolation overhead, which is further explored in Section IV-C.

Figure 5b shows that, as expected, all operator types utilize less memory in case of idle workloads, we observed a 14.21% reduction on average. Compared to 100 idle Golang operators, 100 idle Rust operators utilized 48.04% less memory, which is a smaller reduction than when comparing active operators. However, 100 idle WASM operators still used 83.65% less memory compared to idle Golang operators, similar to the active situation. The smaller reduction in memory usage of container-based Rust operators versus Golang operators is due to Golang experiencing a higher relative reduction in memory consumption when going from active to idle. Based on the typical usage pattern of an operator, which can be idle for a long period of time, it is clear that idle memory usage is important.

End-to-end latency: In Figure 6, the obtained latency distributions for the different operator types are displayed, which were obtained as described in Section IV-A. Based on Jangda et al. [20], WASM performance can be 2.5x slower worst-case compared to native execution. The WASM version of the synthetic-operator, however, did not experience any latency penalties. The latency for the Golang operator increased more than the other operators with increased number of operators.

However, this is most likely due to the memory pressuring algorithm that adds more latency to Golang because its less memory efficient. There was no measured useful difference in latency between the WASM and Rust implementations that was greater than the measured noise. The main bottleneck in the operator's execution is I/O. Therefore, the latencies that occur in CPU-heavy workloads do not affect the synthetic-operator workload much.

C. Cost of isolation

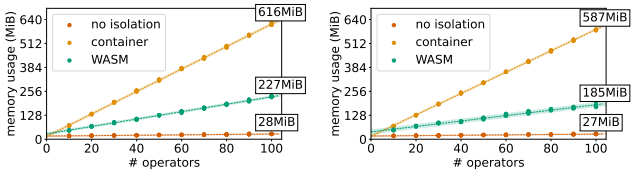
Figure 7 shows the obtained memory upper bounds for Rust operators using no isolation, using containers and using WASM. The coloured areas represent the 95% prediction intervals for the regression models as described in Section IV-A. The solution with no isolation is the most resource efficient. This operator is able to scale to 100 control loops without significant additional memory overhead. Both the WASM-based and container-based setups experience significant per-operator overhead. Additionally, the WASM-based operator has a higher initial constant memory overhead. However, since the container-based solution performs worse per-container, this initial overhead can be compensated. In case of the active situation, the WASM-based solution is more memory efficient than the container-based solution with 95% certainty starting from six operators. For the idle operators, this starts from eight operators.

The container-based operators are managed by Kubernetes and each run in a separate Kubernetes pod. Our Kubernetes setup uses containerd [21] to manage the containers. In our tests, the biggest overhead contributor was the per-pod *containerd-shim* process which equates to about 5MiB per pod. The WASM runtime can isolate the modules without introducing such a big overhead. Instead, it introduces a constant initial overhead that does not depend on the number of operators. This memory overhead is due to the WASM runtime, including the low-level operator logic.

Our tests showed that a major memory usage reduction can be achieved by using no isolation. However, having no isolation between operators means that all operators should be fully trusted even for not having errors. Additionally, it results in a lack of modularity: it is not possible to dynamically add or remove controllers. In an operator design based on Kubernetes pods, operators can be added and removed dynamically. Also, WASM modules can be loaded dynamically by the parent operator, without having to restart the parent operator process. WASM is a good intermediate solution, providing isolation and modularity while still being more memory efficient than the container-based solution.

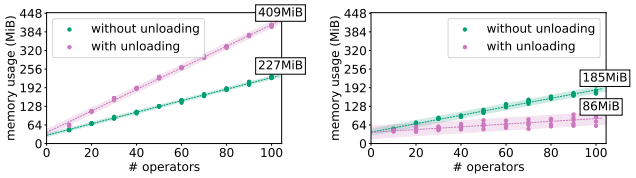
D. Automatically unloading WASM modules

Figure 8 shows the obtained memory upper bounds for the synthetic-operator running as WASM modules. Two versions of the WASM operator are compared: one does not unload the WASM instances and the other unloads each WASM instance in-between each iteration of the reconciliation loop. The coloured areas represent the 95% prediction intervals for



(a) active (b) idle

Fig. 7. The memory 95% upper bounds of non-modular, container-modular and WASM-modular operators show that WASM outperforms container-based isolation, but additional improvements are possible since having no isolation is still much more efficient.



(a) active (b) idle

Fig. 8. The memory 95% upper bounds of the WASM operator with automatic unloading enabled/ disabled; very frequent unloading causes more memory usage, for idle operators it can save memory.

the regression models as described in Section IV-A. Figure 8a shows that the effect of constantly unloading and reloading active WASM operator was an 80.49% increase in memory usage for 100 operators. In Figure 9, the effect of actively unloading and reloading operators on the measured end-to-end latencies is displayed, this figure was obtained as described in IV-A. Figure 8b shows, running 100 operators, we achieved a 52.66% reduction for idle operators compared to not unloading.

Unloading the modules reduces memory usage in case of idle operators. The parent operator writes the memory of idle WASM instances to disk and reloads it later when a Kubernetes watch event is received, as described in Section II-A. Since most operators often stay idle for a long time, this can greatly optimize resource utilization in memory-constrained environments. However, in case of a worst-case unload and reload pattern, memory usage is higher than in case no unloading and reloading takes place. Frequent unloading also introduces a large end-to-end latency penalty due to the disk overhead of swapping the WASM instance, as shown in Figure 9.

To properly benefit from automatic WASM module unloading in a mixed active-idle situation, a predictive scheduler is a necessity, this is considered as future work in this paper. Such a scheduler could help unloading only when it is beneficial to unload a WASM module instead of unloading it in-between each control loop iteration. The optimization opportunity also greatly depends on the heap memory allocated by the operators, necessary for Kubernetes API state caches. This relationship is further discussed in Section IV-E.

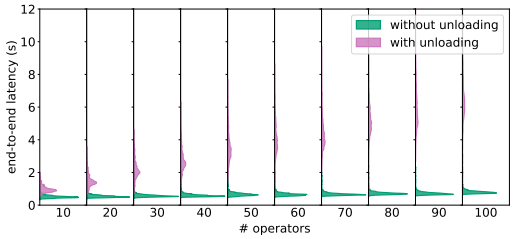
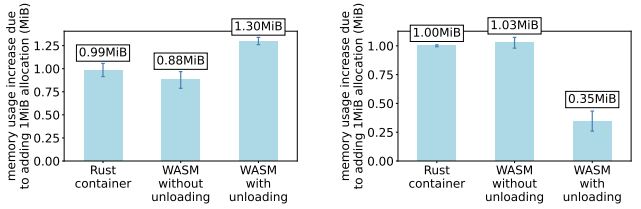


Fig. 9. The end-to-end latency for active WASM operator with unloading enabled/ disabled. Actively unloading and swapping modules introduces significant latency.



(a) The per-operator memory 95% upper bounds increase due to an extra 1MiB of dynamically allocated memory for active operators.

(b) The per-operator memory 95% upper bounds increase due to an extra 1MiB of dynamically allocated memory for idle operators.

Fig. 10. The effects of allocating 1MiB of heap memory in each operator on the 95% upper bounds.

E. Dynamically allocated memory

Figure 10 shows the average memory upper bound increase per operator due to a 1MiB increase in dynamically allocated memory. The metric is obtained based on the slope of the linear regression models trained on 20 upper bound memory usage samples obtained for experiments with allocation sizes of 0MiB to 3MiB, with 5 runs per experiments. Also indicated are the 95% confidence intervals for these slopes.

Figure 10a shows that dynamically allocating 1MiB additional heap memory in each operator resulted in in a memory upper bound increase of roughly 100MiB for 100 active operators with unloading disabled, and in an increase of 130MiB for active WASM operators with unloading enabled. The 30MiB extra overhead originates from the additional memory required to reload the WASM module. Figure 10b shows that the memory consumption for idle operators only increased with 0.35MiB when using our unloading and swapping solution. This is significantly lower than the memory increases for operators without unloading and swapping. As discussed in Section IV-D, adding swapping also adds end-to-end latency. For our experiments, it took about 26ms to swap 1MiB of data to disk per operator, which can be fully attributed to the disk read and write overhead of the hard disk drive in the test server. No latency increase was experienced when using the containerised solution or the WASM solution without unloading.

Operators that watch a large amount of Kubernetes cluster resources will typically keep many of these resources in a cache that they update once the Kubernetes API notifies that a resource change took place. This means that these operators have large amounts of dynamically allocated memory, which directly translates to a memory upper bound increase, as discussed in this section. To reduce this memory usage, it is possible to use our unloading implementation in combination with a tuned scheduler. However, such a solution will result in larger latency overhead due to disk writes. Another solution is to move all operator caches to the parent operator and to deduplicate the resources in these caches.

V. Conclusion

Complex Kubernetes operator workloads are often too heavy for constrained environments. In this dissertation, a novel WebAssembly-based Kubernetes operator solution is proposed. This solution demonstrates that WebAssembly, a technology used by edge FaaS solutions, can also be used to reduce the overhead associated with Kubernetes cluster management. It therefore extends the Wasmtime runtime, adding support for asynchronous Kubernetes API interaction and unloading of idle operators. Our test results show a reduction in memory footprint of 100 active synthetic operators from 1405MiB to 227MiB and of 100 idle operators from 1131MiB to 86MiB by using WASM operators instead of traditional operators. This reduction is due to reduced child operator complexity and the lower WebAssembly isolation overhead. We also found that CPU overhead, identified as a drawback of WASM in prior work [20], does not affect end-to-end latency for our synthetic-operator workload. Unloading WASM operators reduces memory usage for idle operators, while increasing memory usage and end-to-end latency for idle operators. Therefore, future work is needed to add a predictive scheduler that fully optimizes this feature.

Our WASM architecture and implementation demonstrate that initiatives, such as the metacontroller project [8], can integrate a WASM runtime as an alternative to their current webhook solution and benefit from reduced complexity and resource usage. Resource-constrained edge environments are able to run more WebAssembly operators than traditional operators, enabling complex workloads. Cloud deployments become more resource efficient by replacing existing operators with WASM-based operators. The shared benefits of our solution across both edge and cloud segments help accelerate research and adoption.

The biggest open challenges for developing new WASM operators are the WASM and WASI specifications that are still under development. In addition, Golang lacks proper support for WASI, making it more difficult to write operators in Golang. However, Rust operators can more easily take advantage of running as WASM modules. We further propose to obtain additional reductions in memory usage by moving caching logic from the child to the parent operators.

References

[1] E. A. Brewer, “Kubernetes and the path to cloud native,” in *Proceedings of the ACM 6th Symposium on Cloud Computing*. ACM, Aug. 2015, p. 167.

[2] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the 10th European Conference on Computer Systems*. ACM, Apr. 2015.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade," *ACM Queue*, vol. 14, no. 1, pp. 70–93, Jan. 2016.

[4] Shubham, C. Bühler, T. Bannister, and Q. Teng, "Kubernetes operator pattern," Mar. 2022. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

[5] Cloudflare, "Cloudflare workers®," Mar. 2022. [Online]. Available: https://workers.cloudflare.com/

[6] Fastly, "Fastly compute@edge," Mar. 2022. [Online]. Available: https://www.fastly.com/products/edge-compute/serverless

[7] D. Srinivas, J. Liggitt, J. Betz, P. Ohly *et al.*, "kube-controller-manager," May 2022. [Online]. Available: https://github.com/kubernetes/kube-controller-manager

[8] A. Yeh, G. Głąb, Mike, J. X. Tee, S. Bartscher, L. Villard *et al.*, "Metacontroller," Apr. 2022. [Online]. Available: https://github.com/metacontroller/metacontroller

[9] C. Ferris and K. Schlosser, "controller-zero-scaler," May 2019. [Online]. Available: https://github.com/ibm/controller-zero-scaler

[10] T. Cramer, xtutu, stephaneyfx, and I. Dmitrii, "The future trait - asynchronous programming in rust," Apr. 2022. [Online]. Available: https://rust-lang.github.io/async-book/02_execution/02_future.html

[11] T. Ramlot and F. Guardiani, "Wasm operator - master thesis project - optimising memory usage of kubernetes operators using wasm," May 2022. [Online]. Available: https://github.com/thesis-2022-wasm-operators/wasm_operator

[12] F. Guardiani and M. Thömmes, "Kubernetes controllers - a new hope," Jul. 2020. [Online]. Available: https://slinkydeveloper.com/Kubernetes-controllers-A-New-Hope/

[13] S. Akbary, I. Enderlin, M. McCaskey *et al.*, "Wasmer," Apr. 2022. [Online]. Available: https://github.com/wasmerio/wasmer

[14] E. Albrigtsen, T. K. Röijezon, kazk, M. Bagishov, R. Levick *et al.*, "kube-rs," Apr. 2022. [Online]. Available: https://github.com/kube-rs/kube-rs

[15] D. Bakker and L. Clark, "The wasi sockets proposal," Mar. 2022. [Online]. Available: https://github.com/WebAssembly/wasi-sockets

[16] M. Hertz and E. D. Berger, "Quantifying the performance of garbage collection vs. explicit memory management," in *Proceedings of the ACM 20th Conference on Object Oriented Programming Systems and Applications*. ACM, 2005, pp. 313–326.

[17] T. Heo, "cgroupv2 memory," Oct. 2015. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html#memory

[18] J. Weiner, "Psi - pressure stall information," Apr. 2018. [Online]. Available: https://www.kernel.org/doc/html/latest/accounting/psi.html

[19] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman, *Applied Linear Regression Models*, ser. Irwin series in statistics. Irwin, 2005, ch. Chapter 2.6.

[20] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of WebAssembly vs. native code," in *Proceedings of the USENIX 2019 Annual Technical Conference*. USENIX Association, Jul. 2019, pp. 107–120. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/jangda

[21] M. Crosby, L. Liu, P. Estes, D. McGowan, S. Day, A. Suda *et al.*, "containerd," May 2022. [Online]. Available: https://github.com/containerd/containerd

Contents

List of Figures XVII

List of Tables XVIII

List of Listings XIX

List of Abbreviations XX

1 Introduction 1

2 Comparing Kubernetes isolation techniques 3
2.1 Function isolation: software-based fault isolation . . . 4
2.2 Process isolation: containers . . . 6
2.3 Kernel isolation: virtual machines . . . 7

3 Stateless operator control loop and FaaS 8
3.1 Operator control loop . . . 8
3.1.1 Controller example: Deamonset controller . . . 9
3.1.2 Operator example: cert-manager operator . . . 9
3.2 Stateless design and FaaS . . . 9
3.3 FaaS platforms . . . 10

4 WebAssembly 12

5 Existing controller architectures 14
5.1 Containerised operator architecture . . . 15
5.2 Kube-controller-manager architecture . . . 15
5.3 Metacontroller architecture . . . 15
5.4 Controller-zero-scaler architecture . . . 16

6 Solution architecture: WASM operator 17
6.1 Parent operator asynchronous runtime . . . 18
6.2 Child operator asynchronous client . . . 19

7 Implementation 20
7.1 Prior work . . . 20
7.2 Parent operator: WASM runtime . . . 20
7.3 Parent operator: host functions . . . 21
7.4 Child operator: client libraries . . . 22

8 Resource utilisation 24
8.1 Synthetic-operator workload . . . 24
8.2 System details . . . 25

8.3 Memory usage measurement method and analysis . . . 25
8.4 End-to-end latency measurement method and analysis . . . 28
8.5 Golang container, Rust container and Rust WASM compared . . . 28
8.5.1 End-to-end latency . . . 29
8.6 Cost of isolation . . . 30
8.7 Automatically unloading WASM modules . . . 31
8.8 Dynamically allocated memory . . . 32

9 Open challenges and research directions **34**
9.1 Implementation improvements . . . 34
9.2 WebAssembly standards and proposals . . . 34
9.2.1 WASI sockets proposal . . . 35
9.2.2 WASM component model proposal . . . 35
9.3 WASM compilers . . . 35
9.4 SFI-based FaaS . . . 36

Conclusion **37**

Bibliography **38**

List of Figures

Figure 2.1 Hierarchical overview of kernel-, process- and function isolation. . . . 3
Figure 2.2 The full workflow for running an eBPF program in the kernel. . . . 5
Figure 2.3 Security and resource isolation visualised for Linux containers. . . . 6
Figure 2.4 Comparison of full virtualisation versus paravirtualisation. . . . 7

Figure 3.1 The event flow of the Kubernetes operator pattern. . . . 8
Figure 3.2 The Deamonset control loop actions. . . . 9

Figure 4.1 Overview of the WASM runtime components. . . . 12

Figure 5.1 Designs of existing Kubernetes operator and controller architectures. . . . 14

Figure 6.1 Design of our WASM operator architecture. . . . 17
Figure 6.2 The design of the parent operator incorporates a WASM runtime event loop which repeatedly performs actions 1-8; making it possible to asynchronously call host functions from within a WASM instance. . . . 18
Figure 6.3 The design of the child operator allows to perform asynchronous actions in collaboration with the parent operator. . . . 19

Figure 7.1 The operator libraries are split between the parent and child operator and consist out of existing WASI libraries and our own custom libraries based on kube-rs. . . . 22

Figure 8.1 The test setup for the synthetic-operator workload, each operator is responsible for the propagation of changes from one namespace to another. . . . 24
Figure 8.2 The memory usage of a single run for an operator, the operator idles after being active for 7 minutes; samples are selected such that transient behaviour is not captured. . . . 26
Figure 8.3 The distribution of the idle memory WASM operator on the right (b) is a mixture of Gaussians because of temporary memory increases such as shown in the figure on the left (a). . . . 27
Figure 8.4 The memory 95% upper bounds for active and idle periods and the selected memory samples they are based on. . . . 27
Figure 8.5 The memory 95% upper bounds of the different languages/ isolation techniques are ordered as follows: Rust WASM < Rust container < Golang container; all operators use less memory when idle. . . . 28
Figure 8.6 The end-to-end latency of WASM operators is identical to Rust operators. . . . 29
Figure 8.7 The memory 95% upper bounds of non-modular, container-modular and WASM-modular operators show that WASM outperforms container-based isolation, but additional improvements are possible since having no isolation is still much more efficient. . . . 30
Figure 8.8 The memory 95% upper bounds of the WASM operator with automatic unloading enabled/ disabled; very frequent unloading causes more memory usage, for idle operators it can save memory. . . . 31
Figure 8.9 The end-to-end latency for active WASM operator with unloading enabled/ disabled. Actively unloading and swapping modules introduces significant latency. . . . 32
Figure 8.10 The effects of allocating 1MiB of heap memory in each operator on the 95% upper bounds. . . . 33

List of Tables

Table 2.1 The properties of the different Kubernetes isolation techniques. 4

Table 3.1 The most used serverless platforms amongst backend developers in 2021 Q1 annotated with their underlying technology and isolation techniques. 10

Table 8.1 The details of our end-to-end test and benchmarking system. 25

List of Listings

Listing 7.1 Our custom extensions to the WASM parent-child interface. 21

List of Abbreviations

Abbreviation	Explanation
AOT	Ahead Of Time
API	Application Programming Interface
ASI	Address Space Isolation
AWS	Amazon Web Services
cgroup	Control groups
CNCF	Cloud Native Computing Foundation
CPU	Central processing unit
eBPF	extended Berkeley Packet Filter
FaaS	Functon as a Service
I/O	Input output
JIT	Just In Time
kube	Kubernetes
KVM	Kernel-based Virtual Machine
NaCl	Native Client
OCI	Open Container Initiative
PoC	Proof of Concept
PSI	Pressure Stall Information
RBAC	Role Based Access Control
rBPF	Rust eBPF VM
SFI	Software-based Fault Isolation
TCP	Transmission Control Protocol
VM	Virtual Machine
WASI	WebAssembly System Interface
WASM	WebAssembly

1

Introduction

The adoption of new technologies relies heavily on potential resulting cost savings. Monetary and environmental cost reductions have been achieved by increasing computational density in cloud computing. Lower latency costs have stemmed from advancements in telecommunications, such as 5G, and the dispersion of cloud to fog, edge and IoT. These two innovation flows have independently led to widely accepted solutions. Here lies an opportunity for cross-pollination between projects. Cloud computing utilizes cloud orchestration for optimal resource allocation and server cluster management. Kubernetes [1], which originated from Google Borg [2, 3], is such an extensible cluster orchestrator. Its extensible architecture parts a cluster in a control plane and a worker plane. Edge environments, which are typically large complex clusters with limited resources, can benefit significantly from having an orchestrator to manage complexity and size.

However, the primary orchestration targets of Kubernetes are high-resource clusters. Running Kubernetes on low-resource clusters suffers from relatively high control plane operator overhead costs, which hinders adaptation in the edge market segment. In complex cloud-native application deployments, operators [4] are used to automate actions on the Kubernetes cluster state, that would otherwise be performed by a human operator. These operators are one of the main cost drivers of the Kubernetes control plane. To react to changes in the Kubernetes cluster state, the operators have to run as long-living processes. Even if the operator's control loop is idle, the container and process still use cluster resources. For complex applications that use many operators, these overhead costs quickly accumulate and account for a significant portion of the resource utilization. This is especially problematic for low resource deployments.

Since global edge application configuration and deployment is a complex task, it is often abstracted by the service provider and included in a Function as a Service (FaaS) offering. FaaS applications are better suited for low-resource edge environments thanks to their fast on-demand scaling properties. Specifically, some edge FaaS platforms use WebAssembly, a browser technology designed as a portable binary code format that can be assembled from a range of programming languages and that is well suited to resource-constrained environments. On edge FaaS platforms, like Cloudflare Workers [5] and Fastly Compute@Edge [6], WebAssembly is used to securely isolate workloads with reduced overhead and scale-to-zero capabilities.

The goal of this dissertation is to investigate if WebAssembly FaaS approaches can be used to make Kubernetes more suitable for the edge. Specifically, it answers the following research questions:

RQ.1 How does WebAssembly isolation compare to traditional Kubernetes isolation techniques?

RQ.2 How do operator workloads compare to FaaS workloads?

RQ.3 What is the current position of WebAssembly in the FaaS market?

RQ.4 How can Kubernetes use WebAssembly to run operator logic?

RQ.5 How does WebAssembly operator overhead compare to regular operator overhead?

RQ.6 What situations affect the overhead difference between WebAssembly and regular operators?

RQ.7 How does unloading modules and swapping modules affect the memory footprint of WebAssembly operators?

In Chapter 2, Software Fault Isolation and more particular WebAssembly is compared to the other isolation techniques that can be used with Kubernetes. Chapter 3 investigates the resemblance between Kubernetes Operator workloads and typical Function as a Service workloads and identifies the position of WebAssembly in the FaaS market. Chapter 4 explains the main components involved in running a WebAssembly application starting from source code. Chapter 5 provides an overview of multiple existing design patterns for running controllers and operators on Kubernetes. Chapter 6 explains the architecture of our WebAssembly-based operator solution and Chapter 7 discusses how this architecture is implemented. In Chapter 8, our benchmark results of the WebAssembly runtime are discussed, as well as the methodology to achieve these results. Lastly, the remaining open challenges and research directions are discussed in Chapter 9.

2

Comparing Kubernetes isolation techniques

Combined execution of multiple workloads generally requires isolation between components into protection domains [7]. Figure 2.1 shows the kernel isolation, process isolation and function isolation techniques and how they can be combined hierarchically. In this chapter, the function isolation technique, more particular WebAssembly (WASM), is compared to the other isolation techniques that can be used with Kubernetes, answering Research Question 1. Kubernetes by default uses process isolation, in the form of containerised workloads [8]. Additionally, extensions like KubeVirt [9] and Kata Containers [10] enable Kubernetes to manage Kernel-based Virtual Machines (KVM) [11]. Recent additions to the Kubernetes ecosystem, like Krustlet [12, 13] and CRUNW [14], add support for WebAssembly [15]. However, these WASM solutions still lack a lot of features, because they only support the current, unfinished [16] version of the WebAssembly System Interface (WASI) [17]. Additionally, many Kubernetes service meshes use the Envoy proxy [18], which uses WebAssembly plugins to add programmability where configuration options alone are not sufficient.

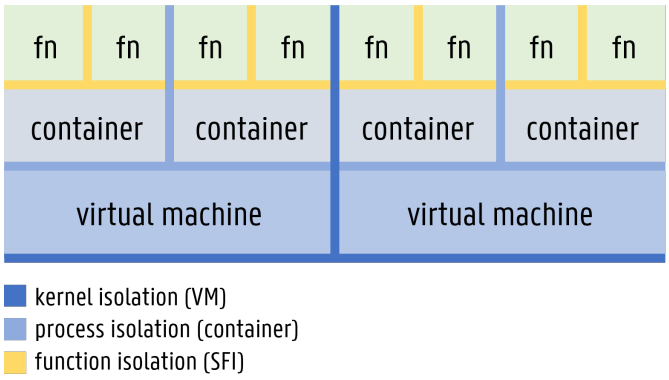


Figure 2.1: Hierarchical overview of kernel-, process- and function isolation.

To compare the different isolation techniques, multiple metrics and properties need to be considered. The first metrics to compare are the overhead costs per isolation unit. Overhead can drastically reduce deployment density if it is extensive for the most limited resource. Secondly, the initialisation time of an isolation unit is compared. This initialisation time is related to the speed of elasticity of an isolation technique, which is inversely correlated to the over-commitment rate necessary for a certain level of service. Each of the isolation techniques guarantees in its own way the three sub-forms of isolation: *fault isolation*, *resource isolation* and *security isolation*. Soltesz et al. [19] explains the three forms of isolation as respectively the ability to limit buggy logic's effects, the ability to enforce fair resource consumption and the ability to limit access/ information leakage. Lastly, as indicative measure for the attack surface, the complexity of the environment API,

which is expressed as the number of different endpoints, is compared. The environment API is the API used by the sandboxed application to interact with the environment.

isolation unit	function			process	kernel	
implementation	WASM	JavaScript	eBPF	container	VM	
API call	WASI	Deno op	helper functions	syscall	VM exit	hypercall
API # endpoints	46[1] [20]	347 [21]	179 [22]	390 [23]	56 [24]	
API latency[2] (# cycles)	6 [25]			72 [26]	1650 [27]	765 [26]
***Fault isolation* [19]**	software			hardware	hardware	
***Resource isolation* [19]**				Linux cgroups [28]	hardware virtualisation	
***Security isolation* [19]**				Linux namespaces [29]	full	
initialisation time [30]	1ms			100ms	100ms - 10ms (Unikernel)	
memory footprint [30]	KB			MB	MB - KB (Unikernel)	

Table 2.1: The properties of the different Kubernetes isolation techniques.

[1] WASI API is still unfinished [16]

[2] CPU cycle measurements are for the older Intel Sandy Bridge architecture; function-isolation latency does not include instrumentation overhead

2.1 Function isolation: software-based fault isolation

To achieve *fault isolation* between functions, functions are sandboxed through instrumentation of instructions that can unintentionally or mal-intentionally alter the execution of another function. This technique is called Software-based Fault Isolation (SFI) [31]. These additional checks are inserted when the machine code is generated to run on the processor. The compilation can happen ahead of time (AOT) or just in time (JIT), depending on the implementation. JavaScript and WebAssembly, both examples of SFI solutions, are web standards supported by the most popular browsers. The Linux kernel supports in-kernel programming through extended Berkeley Packet Filters (eBPF) [32], which is also an SFI solution. Due to the advanced standardisation combined with existing experience and built communities, in most cases reusing these existing SFI solutions makes more sense than creating a new solution. Hence, lots of effort has been invested in running JavaScript and WebAssembly outside of the browser and running eBPF outside of the kernel. NodeJS is the most popular server-side JavaScript/ WASM runtime. However, it does not provide any *resource or security isolation*. On the other hand, Deno is a new upcoming JavaScript/ WASM runtime that does provide *resource and security isolation* [21] similarly to how a browser does. Both solutions reuse the V8 engine developed for use in the Chrome browser. Deno implements the standardised JavaScript interface using a set of underlying environment API functions called *Deno ops*. This ops interface is where Deno enforces its *resource isolation*. Additionally, server-side engines like Wasmtime have been build specifically for WebAssembly. For Wasmtime, the details on *resource and security isolation* depend on the runtime implementation. Lastly, rBPF is an eBPF runtime that runs outside of the Kernel, yielding a lightweight SFI runtime that can target IoT devices [33].

WASM [15] and its NaCl [34] predecessor are supported as the compilation target of multiple high-level programming languages. Existing codebases can be recompiled to these targets for execution in the browser or for server-side SFI with some effort. WASM is also used to protect components within an executable file by preventing error propagation. It prevents third party libraries from possibly compromising whole applications, like Firefox [35] and Gobi [36]. Through SFI it is possible to isolate logical components that are smaller than processes. Recent improvements to WASM runtimes prove that this can be done with minimal overhead [37, 38], which makes it well suited for edge FaaS solutions. For example, as an optimisation, guard zones are used to reduce the number of checks required for secure memory operations [39]. All components run in the same process, meaning no context switching is required. Calling WASM host functions, is equally expensive as executing a call instruction, see Table 2.1. Note that in addition the overhead of the call instruction, checks that make sure the call is permitted, have to be executed. The WASM System Interface is a standardised set of WASM host functions, the implementation of these host functions is left to the WASM runtime implementer, which is also responsible for implementing *resource and security isolation* for those functions. WASI is further explained in Chapter 4. The main security pain point of SFI are speculative execution attacks like Spectre [40] and Meltdown [41]. These attacks can be used to circumvent the instrumentation added by a compiler. A couple of solutions to mitigate these attacks, have been proposed by Narayan et al. [42]. Chromium moved to the Site Isolation browser architecture, a hybrid isolation solution that uses *process isolation* between sites and *function isolation* within sites [43, 44]. Cloudflare its FaaS solution moved to Dynamic Process Isolation [45], which is further explained in Chapter 3.

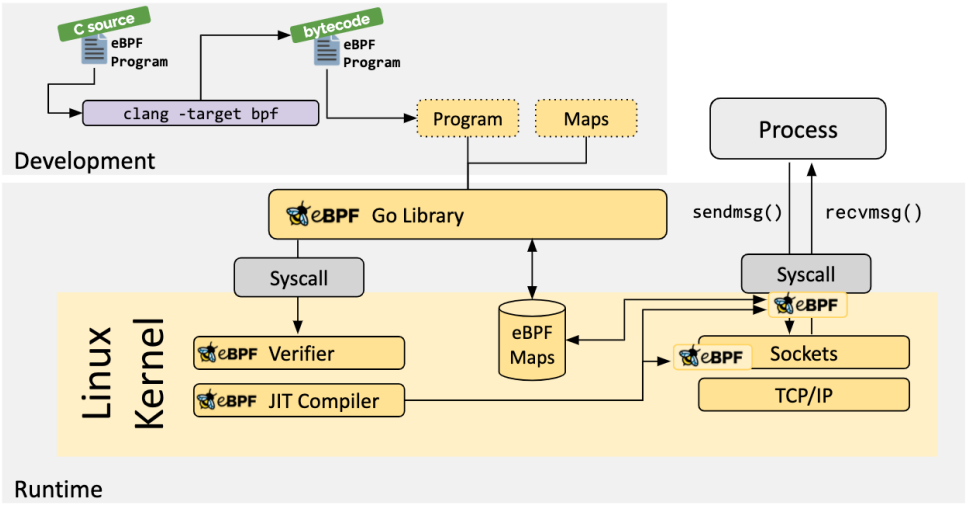


Figure 2.2: The full workflow for running an eBPF program in the kernel.

Extended Berkeley Packet Filter is, similar to WASM, a compilation target of the LLVM compiler [46, 47]. Which means that eBPF programs can be compiled from high-level languages like C and Rust [48]. These eBPF programs can be loaded in the kernel using the bpf syscall. The kernel verifies the eBPF program before successfully completing the load procedure. The verifier makes sure the eBPF program is limited in complexity and does not block the kernel. Depending on the configuration, the eBPF program is executed when a pre-defined hook or a kprobe/ uprobe event occurs [32]. To execute an eBPF program, the eBPF program is compiled to machine code by a JIT compiler in the kernel. This full workflow is visualised by Rudenko et al. [32] in Figure 2.2. An eBPF program cannot call arbitrary kernel functions, instead it has to use the helper functions API [22], as displayed in Table 2.1. The main goal of eBPF is to add programmability to the kernel. This programmability is

often necessary for covering all possible use cases while maintaining a sane amount of configuration options. Compared to WebAssembly, eBPF can be slightly more performant. However, running eBPF programs in userspace requires a lot of extra effort and imposes additional restrictions. Running WebAssembly in userspace is generally a more sensible choice [49].

2.2 Process isolation: containers

Isolation between processes is often referred to as Operating System Virtualisation [19]. Through paging, the operating system makes sure that processes are Address Space Isolated such that one process cannot alter the memory of another process, yielding *fault isolation*. The environment API used by a process is called the syscall API. The Linux syscall API is much more mature compared to the WASI API used by WebAssembly. Process isolation achieves *security isolation* by limiting the resources that are accessible via the syscall API. In Linux, there are 8 namespace types [29] that can each form a protection domain for a type of resource, providing more fine-grained *security isolation* than virtual machines. Furthermore, *resource isolation* is provided by Control Groups (cgroup) [28], which are used to define resource utilisation constraints. A set of Linux processes with their own namespaces and cgroup limits, has all three isolation mechanisms enabled and is called a container (see figure 2.3; based on Bikram [50]).

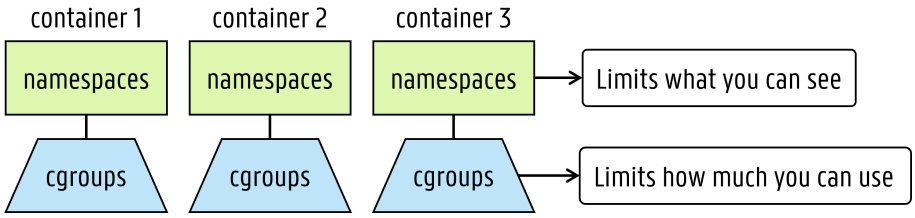


Figure 2.3: Security and resource isolation visualised for Linux containers.

Adding namespaces and cgroups to processes, introduces very limited overhead [51]. Containers are the default isolation technique used by Kubernetes. In Kubernetes however, additional overhead is introduced. The smallest unit of Kubernetes deployment is called a Pod. Pods contain one or more containers that share network and IPC namespaces. A *containerd-shim* instrumentation process is added to each container, and per pod a pause container is added, both adding extra overhead. In WebAssembly, this overhead can be circumvented by modifying the WASM runtime to optimally implement the required features. Generally, the extensibility of the WebAssembly runtime API and implementation can be very advantageous compared to the fixed Linux syscall API and kernel.

Table 2.1 shows that context switches between processes are about 10 times more lightweight than between VMs. Also, resource allocation for containers is more elastic than for VMs, thanks to dynamic fine grained *resource isolation*. Lastly, all kernel logic can be shared between processes, further reducing overhead. In general, containers are more resource efficient than VMs. The performance benefits are countered by additional complexity and thus possible security issues. The full syscall API, for example, is much more complex than the API exposed to VMs. For each of these syscalls, the kernel has to enforce namespace- and cgroup constraints. That makes it hard to guarantee the same level of security offered by VMs. All processes

are exposed to the full syscall API. The gVisor [52] project can be used alongside containers to reduce this syscall attack surface. Internally, gVisor follows the privilege separation pattern [53, 54], combined with seccomp filters. Privileged and non privileged parts of gVisor communicate through inter-process communication [55]. All these in-directions add overhead in return for better security.

2.3 Kernel isolation: virtual machines

For isolation between kernels, *Fault isolation* is implemented in hardware using nested paging [56], yielding Address Space Isolation (ASI) between VMs. *Resource isolation* is done through allocation of dedicated physical resources to a VM or through allocation of virtualised hardware devices that share an underlying physical device. Assuming no unintended crosstalk, access to resources that are not allocated to the VM is fully shielded, yielding full *security isolation* [19]. The API endpoint counts in Table 2.1 show that the attack surface of a virtual machine (VM) is smaller than for container isolation [57]. However, the table also shows that context switching based on a VM exit, are quite costly in terms of CPU cycles. The use of special client paravirtualisation drivers such as virtio [58] can reduce overhead. Figure 2.4 based on Jones [59] shows the differences between full virtualisation and paravirtualisation. These paravirtualisation drivers use hypercalls instead of VM exits to perform context switches, which increases the attack surface by also adding hypercalls as possible entry-points [60, 61]. On Intel processors, a hypercall is performed by executing the vmcall assembly instruction, which switches context to the VM monitor.

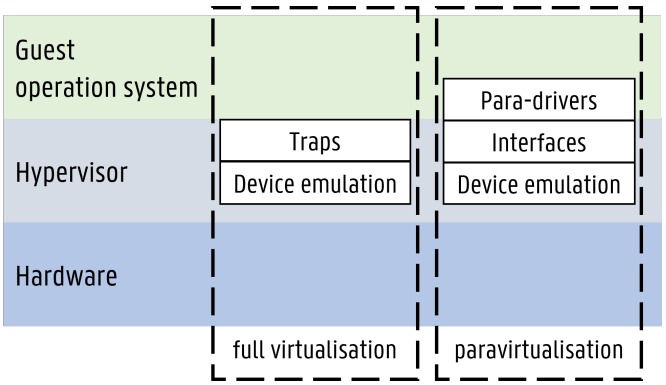


Figure 2.4: Comparison of full virtualisation versus paravirtualisation.

Compared to containers, virtualisation introduces more overhead, especially when dealing with latency-sensitive or I/O intensive applications [51]. Great amounts of effort have been invested in reducing the overhead related to virtualisation. The Firecracker [62] microVM solution reports overheads of 150ms start-up time and a per-VM memory overhead of 3MB. The start-up time can be improved to 50ms through the use of Unikernels [63]. Both examples match the order of magnitude described in Table 2.1 which originates from Shillaker et al. [30].

3

Stateless operator control loop and FaaS

The workload pattern of an operator can be divided into common controller logic and operator-specific logic. The common logic is the same for most operators, while the operator-specific logic is unique per operator and contains the know-how encoded in that operator. In this chapter, the common loop pattern of an operator is discussed and compared to the FaaS function invocation pattern. Section 3.1 discusses the operator control loop. In Section 3.2, it is compared to FaaS, answering Research Question 2. Section 3.3 identifies the position of WebAssembly in the FaaS market, answering Research Question 3, by discussing the most used FaaS solutions.

3.1 Operator control loop

Kubernetes operators adhere to the controller control loop pattern: "In Kubernetes, controllers are control loops that watch the state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state." [64]. For operators, the watched state of the cluster typically consists of the custom resources associated with the operator. Changes to these custom resources trigger a reconcile loop iteration that results in a state update in case the cluster has not yet converged to the desired state, as shown in Figure 3.1 by Perzyna [65]. The reconciliation loop body contains all the operator-specific logic.

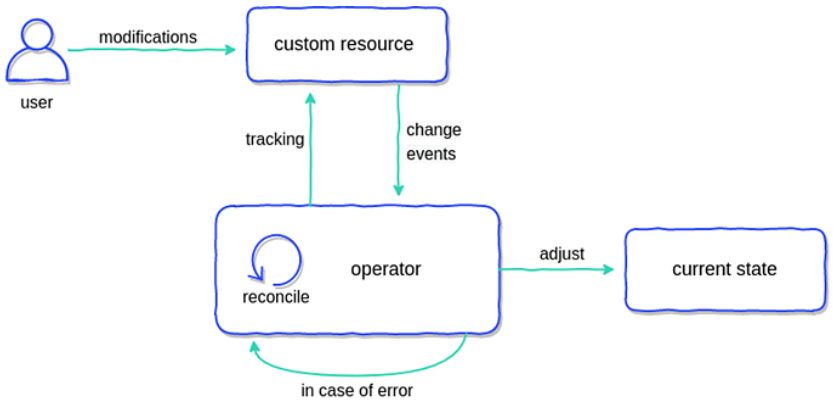


Figure 3.1: The event flow of the Kubernetes operator pattern.

3.1.1 Controller example: Deamonset controller

A simple example of a controller workload is the Kubernetes Deamonset controller, which is displayed in figure 3.2 by Rawat [66]. The controller looks for Deamonset resources and creates the desired amount of Pod resources. Operators, like the cert-manager operator 3.1.2, generally house more complex logic, as they have an equivalent function to what human operators would otherwise do manually. However, it is not always possible to distinguish between controllers and operators.

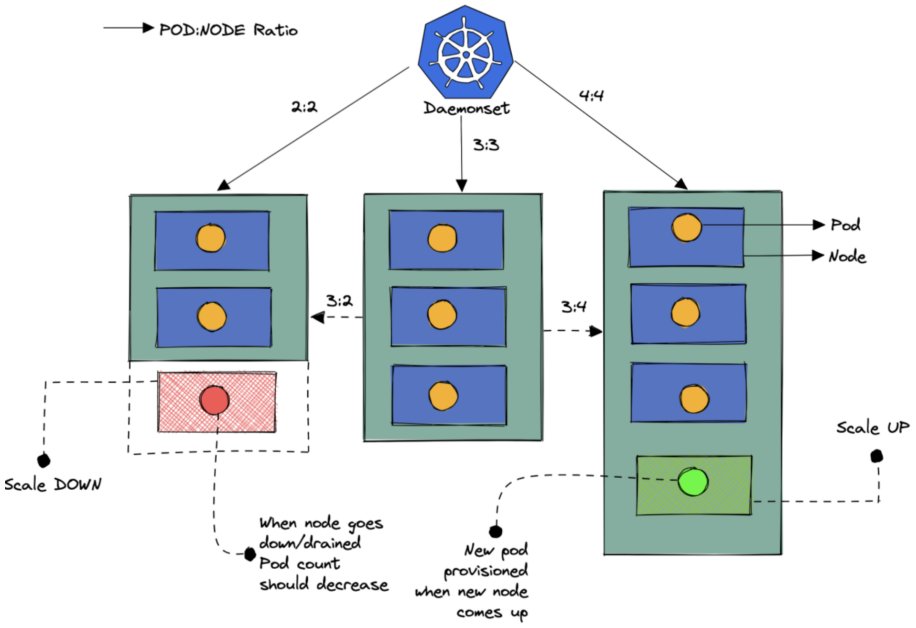


Figure 3.2: The Deamonset control loop actions.

3.1.2 Operator example: cert-manager operator

The cert-manager operator [67] is an example of an advanced operator that manages certificate renewal for the certificates in a Kubernetes cluster. This operator manages multiple CRDs and executes multiple reconciliation loops. The operator waits for new or changed Custom Resources (CRs) and issues new certificates if needed. Additionally, it makes sure that certificates that are about to expire are reissued. The complexity of the operator stems from the variety of configuration options combined with the variety of issuer integrations.

3.2 Stateless design and FaaS

Because of Kubernetes' can-always-fail design, an operator application is not supposed to hold any internal state across reconciliation iterations except for caches. The operator generally uses the Kubernetes API to store state. This theoretically should allow running each iteration of the reconciliation loop without storing state in between. This property also holds

for FaaS systems, where there is no guarantee for state preservation in between function calls. This implies that advancements to the state-of-the-art FaaS solutions are prime candidates to improve Kubernetes operators. In Section 3.3, the most used offerings in the FaaS market with their underlying technologies are listed. The section also discusses the position of WebAssembly within that FaaS market.

3.3 FaaS platforms

FaaS platform	used by [68]	Underlying technology	Tenant isolation	Function isolation
AWS Lambda	54%	Firecracker [62]	VM	
Google Cloud Functions	41%	gVisor [69]	container	
Azure Functions	35%	Hyper-V [70, 71]	VM	container
Google Cloud Run	30%	gVisor [69]	container	
self-hosted platforms	22%			
Cloudflare Workers	16%	V8 [5]	SFI	

Table 3.1: The most used serverless platforms amongst backend developers in 2021 Q1 annotated with their underlying technology and isolation techniques.

Table 3.1 lists the most used FaaS platforms among backend developers. The platforms in the table are annotated with their underlying technology and isolation technique, as explained in Chapter 2. All three isolation techniques are used by at least one platform. Each platform has its perks and drawbacks, often related to the underlying isolation technology. Analogously, for Kubernetes operator isolation, different isolation techniques yield different (dis)advantages.

AWS lambda is the most used platform. The underlying microVM technology of the platform was open-sourced under the name Firecracker [62]. Lambda supports execution of a wide range of programming languages and can be extended with custom OCI-images defining a new runtime and/ or libraries [72]. The latest version uses VM isolation between each function, both within and between tenants. A preceding version instead used container isolation within a tenant [71]. Cold-start latencies are omitted by swapping the contents of a VM instead of rebooting. This content-swapping technique, however, requires a pool of already running VMs. It therefore is a great solution for creating a FaaS platform on large scale but lacks the small-scale potential that is required for low-memory Kubernetes clusters. AWS also offers an OCI-image compatible FaaS solution, called AWS Fargate. The main selling point of this solution is that it can directly run all existing applications packaged as an OCI-image. OCI-images are an open format that can be run on multiple platforms, limiting vendor lock-in. AWS Fargate uses, identically to AWS lambda, microVMs for isolation by utilizing Firecracker [73]. Lambda also has an edge variant called Lambda@Edge, which uses the microVM technology to globally execute JavaScript or Python FaaS functions from 13 edge locations. Lastly, AWS offers an edge FaaS solution called Cloudfront Functions which uses process-based isolation model to execute serverless JavaScript functions at 218+ edge locations [74]. Interestingly, AWS chose process-based isolation over software-based fault isolation by reason of better security like higher resilience against speculative

execution attacks [75]. This is an important challenge to overcome when using WASM isolation.

In second and fourth place there are Google FaaS solutions, which are running on Borg, which is Google's internal container orchestrator and predecessor of Kubernetes [3]. Additionally, gVisor [52] is used as an extra isolation layer, improving *security isolation* [69]. These FaaS solutions are also compatible with applications packaged as an OCI-image. The cold-start overhead of starting containers and runtime overhead of gVisor, as discussed in Chapter 7, cause this solution to be less efficient.

Microsoft's Azure Functions are in third place with their solution that uses VM isolation cross-tenant and container isolation for functions of a same tenant. Before May 2018 the platform shared VMs across tenants [70, 71], which was likely changed due to security concerns.

The last entry included in Table 3.1 is Cloudflare's serverless edge solution. The Cloudflare Workers platform is a JavaScript and WebAssembly SFI solution. It builds on the V8 engine that is also used by the Chrome browser [5]. Other commercial offerings that use SFI are Fastly Compute@Edge [6] and Deno Deploy [76]. Deno Deploy builds on the open-source Deno runtime [21], which runs on top of v8 as discussed in Chapter 2. To mitigate speculative execution vulnerabilities in V8 [40, 41, 77], the Cloudflare Workers platform uses Dynamic Process Isolation [45]. Dynamic Process Isolation detects malicious programs and runs these in a separate process, preventing speculative execution that could circumvent SFI. This hybrid approach allows benefiting from the low SFI overhead, without the security trade-off. Fastly Compute@Edge provides a FaaS platform for running WASM modules, it therefore uses the open-source Lucet WASM engine [78]. As discussed in Chapter 2, deduplicating and sharing logic between sandboxes is easier for SFI solutions compared to container or VM solutions. The host function call overhead is minimal and they are simple to define through extension of the SFI runtime. SFI FaaS runtimes, for example, can deduplicate the networking and routing logic, and drastically reduce the complexity of the applications that run sandboxed. This is promising for Kubernetes WASM operators, which can also benefit from moving complexity from the operator to the shared operator runtime.

Fifth place is for self-hosted solutions. Many of these self-hosted FaaS platforms like Fission [79], OpenFaaS [80], and Knative [81] build on top of the Kubernetes ecosystem. By default, they use containers to run FaaS functions distributed as OCI-images, vm-based isolation is supported through the use of a Kubernetes vm-based OCI runtime like kata containers. Since Krustlet [12] and CRUNW [14] require compilation to WASM and still have limited support for sockets, using SFI as underlying isolation technique is not yet supported for these Kubernetes-based FaaS platforms. Other solutions like OpenWhisk [82] and Nuclio [83] support deployment on Kubernetes, but do not depend on Kubernetes for isolating FaaS workloads, making it harder to switch the underlying isolation technique away from containers. Self-hosted WASM-based FaaS solutions like Faasm [30], Sledge [84] and Spin [85] aim to provide an open-source alternative to the Cloudflare Workers [5] and Fastly Compute@Edge [6] commercial offers.

4

WebAssembly

A full WebAssembly solution consists out of a lot of different components. This chapter aims to demystify the high-level workings of WebAssembly solutions and the technical terms used for internal components. Figure 4.1 shows the full flow, from START to DONE, for running a Rust program using a WebAssembly runtime.

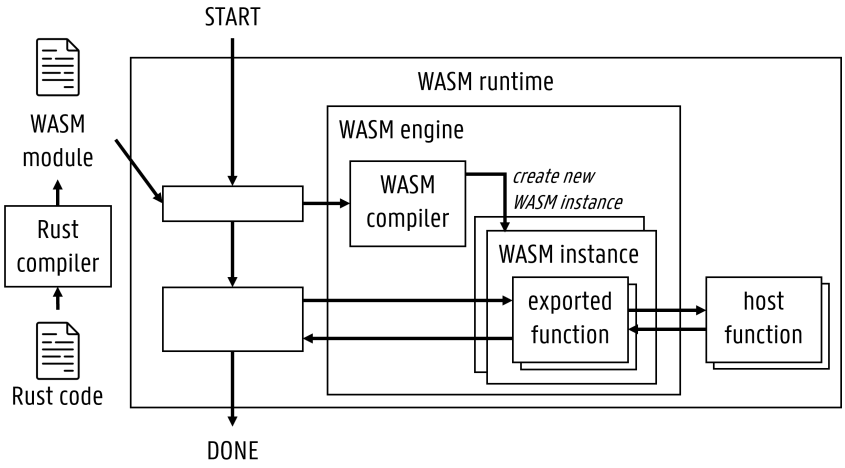


Figure 4.1: Overview of the WASM runtime components.

The components and their interactions as shown in Figure 4.1 are explained in detail below.

WASM module: A WASM module is a portable binary-code file that can be compiled from a range of programming languages and executed by a WASM runtime.

to-WASM compiler: Programming languages like Rust, C, C++, Golang, ... can be compiled to WASM by configuring the compilation target architecture of their default compilers. Most of these compilers first compile the high-level language to the LLVM-IR and use LLVM to compile to the target architecture. The LLVM compiler supports WASM [47], which makes it easier to add support.

WASM-to-bytecode compiler: To run a WASM module, the binary WASM-code has to be translated to instructions that can be executed by the target machine, like x86-64. Herefore, a compiler has to perform this translation. This can be done ahead-of-time or just-in-time. The compiler also has to generate instructions that prevent insecure operators. This way Software-based Fault Isolation is implemented.

WASM instance: An instantiated version of a WASM module. One module can have multiple instances, but each instance originates from one module.

WASM engine: The WASM engine is responsible for running the WASM module. It uses the WASM-to-bytecode compiler to generate the instructions. It initiates the memory layout of the WASM instance and initiates the objects and functions the instance needs. It also provides an API to interact with the WASM instance.

WASM runtime: The WASM runtime is responsible for letting the WASM instances interact with the environment/ outside world. It therefore provides a set of host functions that can be called by the WASM instance. Similarly, the WASM instance can expose a set of functions that can be used by the runtime to let the WASM instance perform a certain action. The WASM runtime can also directly read and write to the linear memory used by the WASM instance via the WASM engine. This way it can copy complex values into the memory of the WASM instance.

WASM host (imported) function: Host functions are functions implemented by the WASM runtime and provided via the WASM engine to a WASM instance. These host functions can be referred to, directly in the WASM binary, and it is the WASM engine's task to make sure that they are run when that part of the WASM binary is executed. The instance can use these functions to perform actions that alter the environment, for example printing text to the screen.

WASM module (exported) function: Functions exported by a WASM module are defined in the WASM file and can be invoked by the runtime via the engine. These functions, when called on the WASM instance, will execute the logic as defined in the WASM module. They are required by the runtime, such that it can initiate an action or to transfer control. For example, WASM modules export an entry point function that should be called to start executing the module.

WASI: Web Assembly System Interface is a standardised set of host functions, used by non-browser runtimes. Most of the popular programming languages have a standard library that supports WASI as compilation target, meaning that the appropriate WASI host functions are used to perform environment interactions when the standard library functions are called by the WASM instance. For example, a controller running in a WASM sandbox can use the WASI interface to read environment variables set by the runtime and uses this WASI interface to log messages to stdout or a log file. At the time of writing, WASI is still unfinished. Most limiting is that it does not yet include host functions for working with sockets [16].

The current WASM standard only supports passing very simple arguments (integers and floats) for imported and exported functions as arguments or as return value. This means that more complex values have to be passed as references to memory locations inside the WASM sandbox, such that the runtime can fetch the complex values there. If the runtime wants to pass a complex value to the WASM module, it first has to ask the module to allocate a region of memory inside the sandbox. The complex values that are stored at those memory locations, have to be encoded before sending, and decoded when received. The libraries for serialising host function arguments and deserialising the results of a host function call, thus have to be functionally identical in both the WASM host and the WASM client libraries. The WebAssembly community has spent a lot of effort on creating a canonical ABI to (de)serialize complex values in all languages the same way. Once this is fully supported and standardised by WebAssembly it will further simplify creating new WASM runtimes with custom host functions. The canonical ABI is part of the *WASM component model proposal* which is further discussed in Section 9.2.2.

5

Existing controller architectures

In Kubernetes, the cluster-state is stored in a distributed database and is query-able via the Kubernetes API server. Automated operations performed on that state are typically done by controllers. Some of these controllers are part of the Kubernetes core and are installed as part of all Kubernetes deployments. Most Kubernetes deployments also extensively utilize 3rd party controllers that extend the Kubernetes control plane, some of which are operators. This extensibility is one of Kubernetes' most valuable properties. In Section 5.1 the design of 3rd party operators is explained, Section 5.2 explains the core Kubernetes controllers' design. Section 5.3 shows the architecture of controllers that are part of the metacontroller project [86] and Section 5.4 shows the architecture of the controller-zero-scaler project [87].

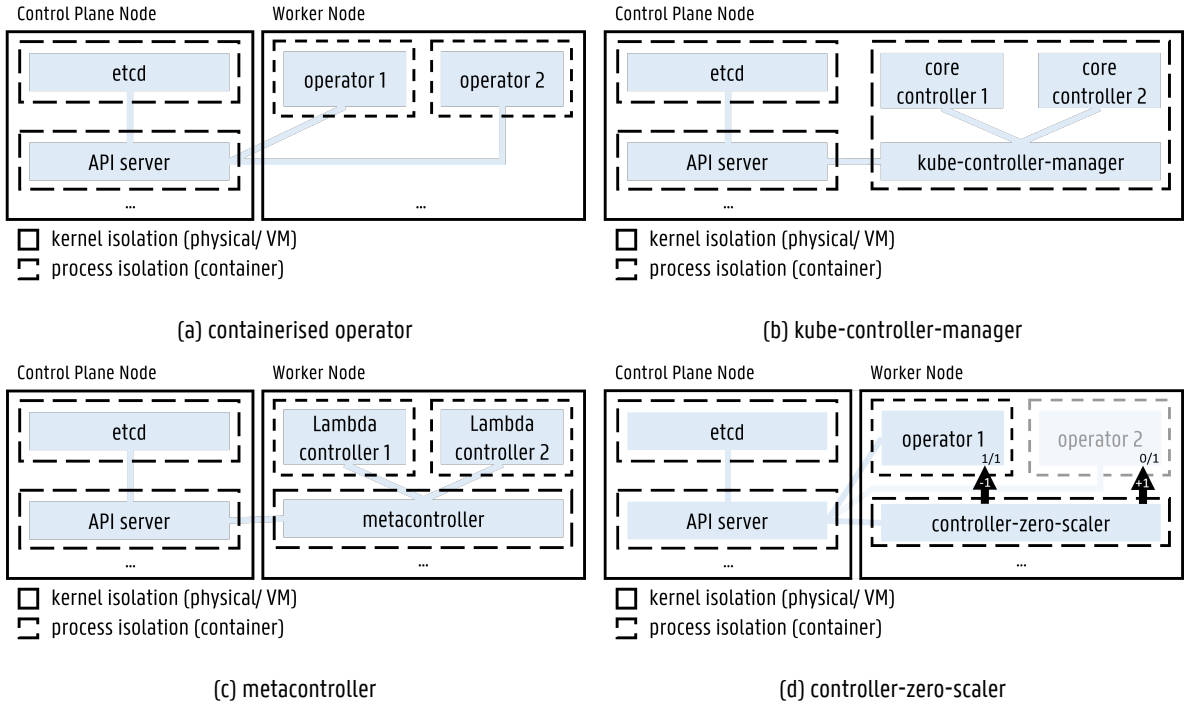


Figure 5.1: Designs of existing Kubernetes operator and controller architectures.

5.1 Containerised operator architecture

Kubernetes operators automate actions that a human operator would otherwise perform on a Kubernetes cluster. The operators often come with their respective Kubernetes custom resources. Figure 5.1a, shows the default Kubernetes operator architecture. Each operator runs in its own container and its own Kubernetes pod. This architecture allows to easily add or remove operators by adding or removing pods to Kubernetes. Each operator is isolated, and its capabilities can be limited. However, this architecture introduces quite some overhead. Each operator runs in its own pod and has its own Kubernetes state cache and low-level operator logic, resulting many operators that have a lot of duplicated logic in common. Most operator implementations limit the number of concurrently running instances of an operator to a single instance, with no option to scale horizontally. If a second instance were added, it would idle because of leader-election. This is done to prevent two operator instances from simultaneously updating the same Kubernetes resources, resulting in update conflicts that slow down the reconciliation progress. However, to make progress it is necessary that one instance is running at all times. The operators thus have to run continuously, wasting valuable resources.

5.2 Kube-controller-manager architecture

The core control loops, shipped with Kubernetes are combined in a single kube-controller-manager binary [88], see Figure 5.1b. This reduces size and operational overhead and simplifies management and deployment. Thanks to all controllers being in a single binary and being run as as single process, code can be deduplicated, and caches and Kubernetes informers can be shared. However, this design comes with some limitations. All controllers have to be known at compile time. It is not possible to dynamically add a control loop to the set of running controllers, without creating a new version of the binary and restarting all running controllers. If one of the controllers fails, this will cause the full kube-controller-manager process to fail and will affect all other running controllers. Lastly, untrusted controllers can overtake the full kube-controller-manager process because of lack of isolation between controllers.

5.3 Metacontroller architecture

Metacontroller [86] is a project that aims to simplify the development and creation of new Kubernetes controllers. It consists of a main controller, the metacontroller, and a set of simple Lambda controllers. These Lambda controllers implement the business logic. They are simple applications that act as a webhook. A Lambda controller is typically run in a separate pod and is language agnostic. Similarly to the architecture in Section 5.1, the Kubernetes pods that serve the webhooks have to continuously run, introducing overhead. Alternatively, the webhooks can be hosted by a (self-hosted) serverless platform, as described in Section 3.3, to achieve a lower overhead solution. The simple Lambda controllers do not each watch their set of Kubernetes resources, as is the case for the *operator architecture* described in Section 5.1. Instead, this low-level operator logic is implemented in the metacontroller, which calls the webhooks containing the business logic when needed. Using the configuration in the form of the metacontroller Kubernetes API resources, the metacontroller determines which webhooks to

use and when to use them. These metacontroller Kubernetes API resources can be used to dynamically add or remove control loops. The metacontroller houses multiple control loops, like the *kube-controller-manager architecture* from Section 5.2.

5.4 Controller-zero-scaler architecture

The controller-zero-scaler project [87] is a concept project developed by IBM. It tries to solve the operator overhead problem by dynamically downscaling the traditional operator deployments. The architecture starts from the *operator architecture* described in Section 5.1. Additionally, the controller-zero-scaler controller is running in the cluster and watches the Kubernetes API for changes. If it detects that the resources managed by an operator have not been altered for a long period of time, it downscales that operator deployment to zero instances. This results in Kubernetes stopping that operator, freeing resources. If the controller-zero-scaler notices new changes, it upscales the deployment, such that the necessary operator logic is executed. Note that this project is not actively developed and more of a concept than a project used in production. The main advantage of the setup is that the resource utilization of idle operators is fully eliminated. However, the downscaling has to happen infrequently, because of the overhead that results from stopping and restarting the operators. All cached cluster state is also lost when restarting, resulting in additional load on the Kubernetes API server.

6

Solution architecture: WASM operator

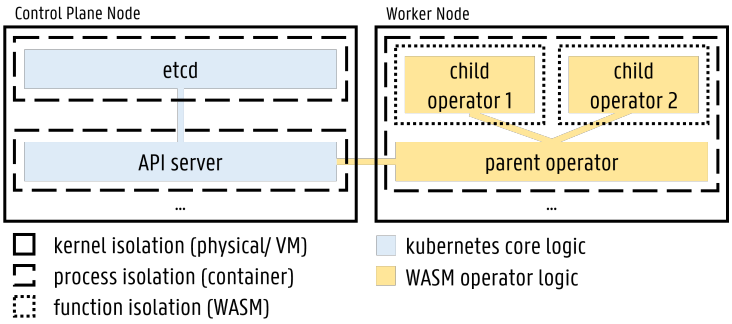


Figure 6.1: Design of our WASM operator architecture.

The *WASM operator architecture* presented in this dissertation and visualized in Figure 6.1 attempts to realize all the beneficial aspects of the four architectures described in Chapter 5. This chapter explains how our architecture uses WebAssembly to achieve this, answering Research Question 4. The main parts of the architecture are the parent operator and the child operators. The child operators run as WASM instances in the WASM runtime embedded in the parent operator. We use an existing WASM runtime implementation as embedded runtime. The beneficial aspects are listed below.

Isolation overhead: All child operators run in the same process as the parent operator. This process runs inside a single container in a single Kubernetes pod. Isolation is provided by the WASM engine, eliminating the overhead due to container isolation, similar to the *kube-controller-manager architecture*.

Modularity: The WASM runtime makes it possible to add or remove child operators without interfering with the other active child operators. This modularity is missing from the *kube-controller-manager architecture*, but is provided by the other architectures mentioned in Chapter 5.

Simple child operator: In our architecture, the parent operator extends the WASM runtime with host functions that can be used by the child operators to communicate with the Kubernetes API. Low-level operator logic is moved to the parent operator. This reduces the complexity and overhead of the client operators similarly to the *metacontroller architecture*.

Scale-to-zero: To limit the overhead of inactive operators, our architecture allows to dynamically unload inactive operators, similar to the *controller-zero-scaler architecture*.

In order to efficiently make Kubernetes API requests, we want child operators to perform them asynchronously. Existing WASM runtimes offer no support for asynchronous calls or offer a solution that is incompatible with idle module unloading. Therefore, we created a new solution that adds support for asynchronous operations to the WASM runtime that is embedded in our parent operator. By extending the WASM runtime, we allow the child operators to wait for host functions asynchronously. Section 6.1 describes how we extended the WASM runtime, such that it meets these requirements and additionally supports unloading idle operators. Section 6.2 explains how the client libraries within the child operator interact with the parent asynchronous runtime.

6.1 Parent operator asynchronous runtime

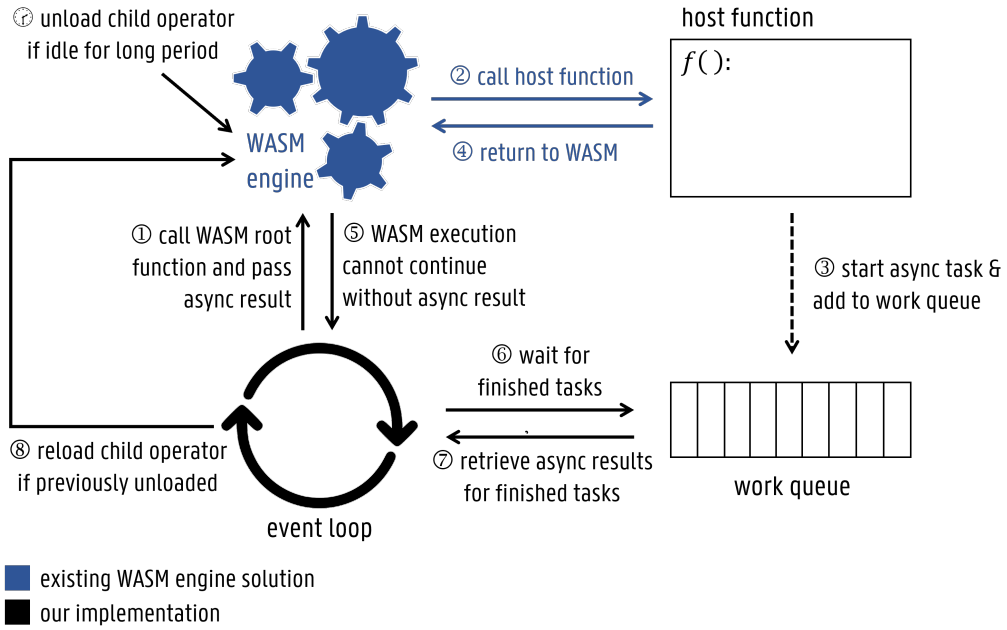


Figure 6.2: The design of the parent operator incorporates a WASM runtime event loop which repeatedly performs actions 1-8; making it possible to asynchronously call host functions from within a WASM instance.

The parent operator architecture extends a WASM runtime, as described in Chapter 4, by adding support for asynchronous host functions and support for unloading of idle modules. Figure 6.2 shows how the parent operator manages the asynchronous operations of a child operator and unloads an inactive child operator after a long period of inactivity. The main components of the parent operator are the WASM engine, the host functions exposed to the WASM instance and the work queue. For the WASM engine component and some of the host functions, existing solutions can be used.

The solution works as follows: Each WASM module has an entry point that executes the main function in the child operator, shown in Figure 6.2 as ①. Environment interactions happen through the calling of WASM host functions ②. Some of these actions are asynchronous and do not directly yield a result. These asynchronous actions are started and added to the work

queue ③, directly returning control to the WASM module ④. After executing all synchronous logic, the WASM execution stops and the control is returned to the event loop ⑤. This loop checks if any of the actions in the work queue finished ⑥ and passes the results of that finished action ⑦ back to the WASM engine ①, reloading the child operator in case it had been previously unloaded ⑧. When returning to WASM, a new set of synchronous actions are performed by the engine. Long-running operators repeat this process indefinitely. These operators are always waiting for new asynchronous inputs, like events in a watch stream.

The runtime might detect that a certain child operator has not been receiving any asynchronous results over a long period (marked as 🕒). This is indicative for an operator that reached a steady state in its reconciliation process. Most likely, it will only restart its logic after external applications changed the state of the Kubernetes resources that it manages. This could mean that the operator remains idle for multiple hours. In such cases, it can be more resource-efficient to unload and swap the WASM instance to disk.

6.2 Child operator asynchronous client

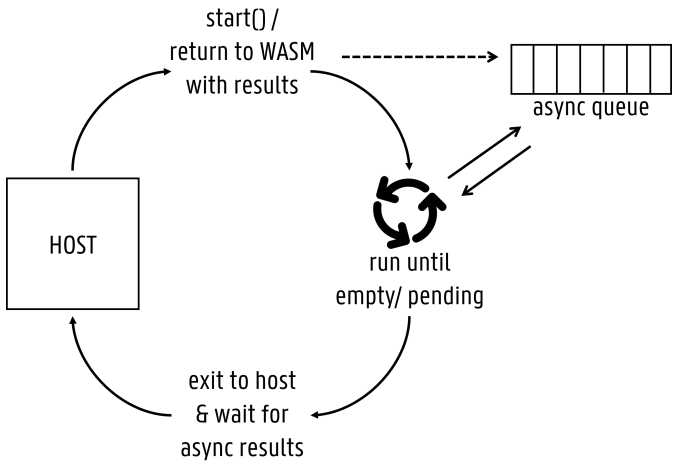


Figure 6.3: The design of the child operator allows to perform asynchronous actions in collaboration with the parent operator.

Figure 6.3 shows the design of the asynchronous request passing and response processing, inside the child operator. All client operators run as single-threaded asynchronous WASM instances. The child operator is started by the host which calls the `start` function that is exposed by the WASM module. This initial function starts the operator reconciliation loop, which makes asynchronous requests that populate the async queue. These asynchronous futures [89] are `awaited` by the child operator, but some of these futures `await` asynchronous host function results from the parent runtime. If the child operator cannot continue without new results from the host environment, it stops the execution and returns to the host. If none of the pending asynchronous requests have finished already, the host waits for one of them to finish, as described in Section 6.1. Once a request finishes, the host returns the result to the child operator, such that the child operator can finish the linked asynchronous request. This restarts the whole process.

7

Implementation

The latest version of our implementation and the scripts used for the end-to-end tests can be found on Github [90]. As explained in Chapter 6, our architecture foresees a parent operator and a set of child operators. Section 7.2 discusses the parent operator async WASM runtime implementation. Section 7.3 explains how all in- and outbound communication from the WASM modules happens for our implementation. In Section 7.4 the WASM client library implementations are discussed.

7.1 Prior work

Our operator implementation builds upon the proof of concept (PoC) made by Francesco Guardiani and Markus Thömmes [91]. This PoC provides a WASM operator solution based on the Wasmer [92] WASM runtime and a hacked version of the kube-rs [93] library. However, at the time of writing, it has been 2 years since this project was updated. Since the API of the Wasmer runtime drastically changed after its v1 release, and the hacks applied to the kube-rs project are not well documented, upgrading the PoC was not straight forward. Furthermore, the Wasmer project lacks the future potential that other open-source initiatives, like Wasmtime, can offer. To update kube-rs more easily in the future, a new project structure was required. Moreover, the original version of the PoC cannot unload inactive operators as its architecture is different from the architecture proposed in Chapter 6. We refactored the PoC and updated it to implement the aforementioned architecture. Finally, we implemented several improvements to further optimize the PoC implementation, such as adding support for caching compiled WASM modules for later reuse.

7.2 Parent operator: WASM runtime

The parent operator extends the Wasmtime WASM runtime. Wasmtime was chosen over other WASM runtimes, because it is the flagship WASM engine from the Bytecode Alliance, with support from some of the biggest players in the technology industry. As a consequence, it will quickly support newly standardised features. The Lucet [78] engine even joined forces with Wasmtime, such that the Wasmtime engine can evolve more quickly. Our implementation configures Wasmtime to compile ahead of time new WASM modules to machine code to eliminate the compiler memory overhead at runtime. These compiled modules are cached on disk and can be reused when possible. To initiate these compiled modules, Wasmtime only has to map the file to memory and provide the necessary tools to communicate with this initiated module. Because of the

use of file-backed memory, for idle operators, these memory locations can be dropped from memory by the kernel when needed. If the memory region needs to be accessed again, a page-fault will be triggered, and the kernel will load the file back into memory. However, the dynamically populated memory of the WASM module will not be unloaded automatically from memory. That is why our implementation adds a custom unloading and disk swapping implementation in the parent operator. This makes unloading and swapping possible, even on systems without swap enabled at operating system level. The parent operator is developed in Rust, because Wasmtime is also written in that language. Adding additional functionality to the Wasmtime runtime, like host functions, is thus best supported for a parent operator that is also written in Rust.

As described in our design, a parent operator runs an event loop for each of its child operators. The main task of an event loop is to let child operators perform I/O heavy operations asynchronously. Our implementation implements event loops as Rust futures [89], which are polled when the loop is initialised and later on, when it is awoken because of the occurrence of a desired I/O event. Our implementation is similar to the implementation used by Deno [21], including their FaaS solution as described in Chapter 3. Our parent operator passes the event loop futures to Tokio [94], which is an existing asynchronous Rust runtime library that assures that the future execution advances and the futures are awakened when new awaited I/O events occur. Tokio therefore schedules these futures on green threads, called *tasks*. These tasks can be executed on multiple cores simultaneously, implementing the many-to-many thread model [95].

7.3 Parent operator: host functions

```
// exported WASM module functions
fn wakeup(async_id: u64, finished: u32, ptr: *const u8, len: u32) { ... }
fn allocate(size: u32) -> *mut u8 { ... }

// provided WASM host functions
fn delay(millis: u64) -> u64 { ... }
fn request(ptr: *const u8, len: u32, stream: u32) -> u64 { ... }
```

Listing 7.1: Our custom extensions to the WASM parent-child interface.

As described in Chapter 4, WASM host functions are functions exposed by the WebAssembly runtime to the WASM instances. WASI is a standardised set of these host functions. Our implementation can benefit from the existing Wasmtime library that readily implements these WASI host functions, reducing the implementation and maintenance burden of our solution. A core aspect of Kubernetes operators is communicating with the Kubernetes API server. However, at the time of writing, the WASI spec has not yet standardised sockets as part of the interface [16], this is further discussed in Section 9.2.1. This means that for our implementation, we had to implement custom HTTP host functions to create a working WASM operator setup. In Listing 7.1, the *request* and *delay* host function signatures are listed. The *request* function makes it possible to perform Kubernetes HTTP requests and receive answers with a WASM module. As shown in Figure 7.1, our implementation uses the low-level part of kube-rs for the Kubernetes `request` host function implementation, kube-rs manages setting up the connection with the Kubernetes API and performs the authentication. The high-level kube-rs functionality is implemented

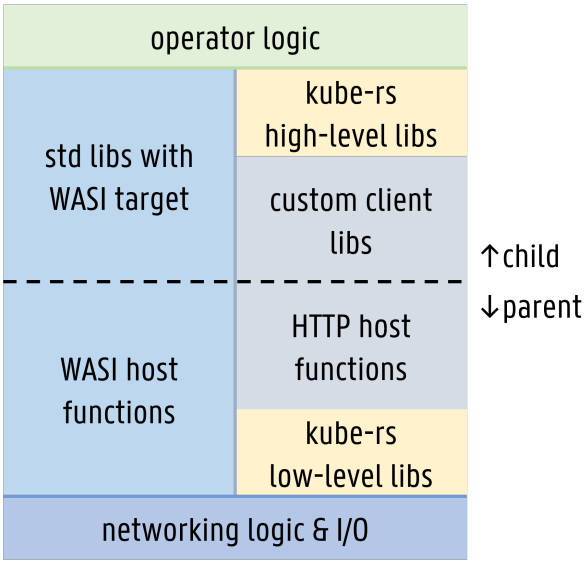


Figure 7.1: The operator libraries are split between the parent and child operator and consist out of existing WASI libraries and our own custom libraries based on kube-rs.

in the child operator. The *delay* function allows the module to wait for a specified period of time, which was another missing feature required by the client operators. The functions `delay` and `request` are both asynchronous functions, meaning that they return control to the WASM module immediately, while returning an `async_id` that references a task in the work queue as described in Section 6.1.

7.4 Child operator: client libraries

All Kubernetes operator domain knowledge is implemented in the custom reconciliation loops, that are defined in the child operators. Our language preference for the child operators is Rust since the Rust standard libraries best support the WASI host function calls. Golang, which is normally used in Kubernetes, has no support for WASI in its default compiler. However, the TinyGo [96] compiler supports the WASM-WASI target but cannot perform serialisation due to its limited support for Golang reflection. Additionally, Golang is a garbage collected language, which have been shown to use more memory [97]. Another advantage of choosing Rust as language is that an easier interoperability between the parent and child operator is obtained. The (de)serialisation logic mentioned in Section 7.3, can be reused for both the parent and child, since they are both implemented in Rust.

To use the HTTP host functions, we implemented a custom client library that can be used by the WASM child operator. This library provides an API to interact with the HTTP host functions from within the child operator. It also exposes two additional functions to the parent operator, as visible in Listing 7.1. The exported `wakeup` function is used by the parent operator to pass the results of asynchronous requests. As mentioned before, these results often are complex values. The parent operator uses the exported `allocate` function to allocate a region in the WASM instance's memory to write the complex value

to. As mentioned in Section 7.3 and listed in Listing 7.1, the operator exposes the `request` host function to make HTTP requests against the Kubernetes API. It then receives the response to that request asynchronously via the `wakeup` exported function, as explained in Section 6.2. The implementation also supports streaming responses. Therefore, the WASM module; passes `stream = 1` as argument in the `request` function, since boolean values are not directly supported as function arguments. The `wakeup` function indicates, using the `finished` argument, whether the result is the last result in the response stream or not. On top of the library for passing HTTP requests and responses to the parent operator, the child operator uses the high-level part of the kube-rs library to fabricate requests that the Kubernetes API can understand and to decode the responses that are returned. The kube-rs library enables implementation of an operator control loop, which watches the desired state as described by Kubernetes Custom Resources and performs the actions to achieve this state.

New client operators can be implemented by creating a Rust project that uses our custom client library and compiling that project to the WASM-WASI target. The obtained WASM module file can be loaded by the parent operator, which also starts the execution of the module's reconciliation loop. Our implementation foresees a simple configuration file that specifies what WASM files should be loaded by the parent operator, and we assume that the files are available on the local file system. However, to achieve a better user experience, ideally, the WASM files are distributed via OCI images and the configuration is stored in Kubernetes resources. This improvement is left as a possible future improvement and should not impose new challenges.

8

Resource utilisation

The operator control loop, as described in Section 3.1, is typically bound by I/O waiting. While the operator waits for new I/O events, it remains idle. If an operator achieves its desired state for the resources it manages, the operator can remain idle for multiple hours. To achieve optimal operator density, the memory usage of the operator and its runtime are usually most limiting. As described in Chapter 6 and 7, the proposed WASM operator is designed to deduplicate low-level operator logic and unload WASM modules after long periods of idle time. Section 8.5 and 8.6 investigate the WebAssembly operator overhead compared to regular operator overhead with regards to memory usage, answering Research Question 5. To get a full image of the factors that affect overhead differences between the different solutions, answering Research Question 6, all tests are performed for both active and idle operators and in Section 8.8 multiple sizes of dynamically allocated memory are compared. Since unloading and reloading child operators to and from disk takes time, it is also important to compare the end-to-end latency of the operators. Long end-to-end latencies can result in configuration change unresponsiveness, harming the experience of administrators and developers. In Section8.7, Research Question 7 is answered by comparing the memory usage and end-to-end latencies of the WASM operator with unloading and swapping enabled versus disabled.

8.1 Synthetic-operator workload

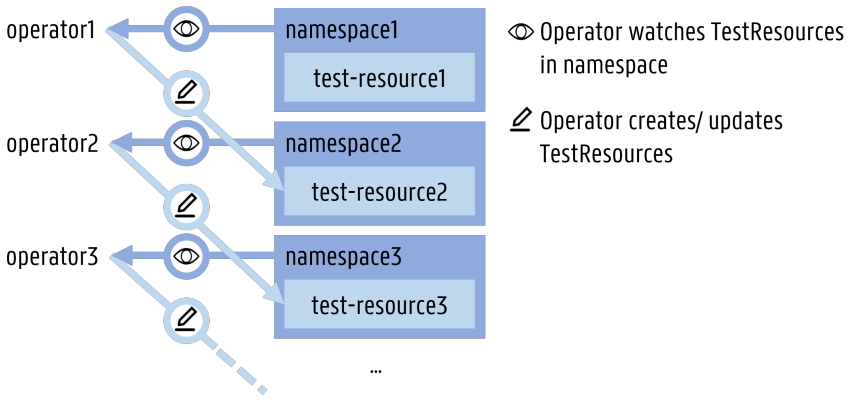


Figure 8.1: The test setup for the synthetic-operator workload, each operator is responsible for the propagation of changes from one namespace to another.

The test synthetic-operator, as shown in Figure 8.1, simulates a workload with N different operators, which depend on each other's actions and are idle for most of the time. Each operator watches a namespace for `TestResources` and only reconciles, once a resource is created or updated. It then updates/ creates the resource in its destination namespace. The input watch and output create/ update namespaces for each operator can be configured dynamically at runtime. All used operators contain a unique nonce to prevent trivial optimizations, such that we correctly simulate a situation where all N operators are different. For a full update of all resources, all operators must update their resource one-by-one. This means the full end-to-end latency equals the accumulated individual operator latencies. This synthetic workload simulates a highly dependent and interactive operator setup.

8.2 System details

All tests were performed on a machine with the hardware details described in Table 8.1a. The versions of the software installed on the machine are described in Table 8.1b.

CPU	Intel Xeon E3-1220 v3 @ 3.10GHz
RAM	2x8GB Samsung M391B1G73QH0-CMA
disk	250GB Travelstar Z7K500 HTS725025A7E630

(a) hardware details

kernel version	5.15.0-30-generic
go version	1.18.2
rustc version	1.60.0
containerd version	1.6.4
kind version	v0.13.0
kubernetes version	v1.24.0
wasmtime version	0.36.0

(b) software details

Table 8.1: The details of our end-to-end test and benchmarking system.

8.3 Memory usage measurement method and analysis

Measuring the memory footprint of a workload execution, requires accounting all the memory usage effects that the process has on the system. This is a non-trivial problem. The Linux kernel provides multiple measures for memory usage, both for individual processes and control groups (cgroups). The process-based metrics do not account for resources used by the workload outside of the process, like kernel data structures. The cgroup v2 measure `memory.current` [98] is the most complete measure currently exposed by the Linux kernel [99, 100]. It includes "Userland memory - page cache and anonymous memory", "Kernel data structures such as dentries and inodes" and "TCP socket buffers" [98]. This metric is also used by Kubernetes to limit memory usage and prevent rogue processes from starving other processes.

In case of limited available memory, the kernel instantiates a set of procedures to reduce the system's memory usage. Memory can be reclaimed, which means that data is dropped from memory. For memory mapped from a file, this can be

done instantly. For other memory regions, first the memory contents are stored to a swap file on disk. Additionally, memory management systems within processes can reduce memory consumption. Garbage collected programs for example can additionally free memory by more frequently performing garbage collection. It is necessary to take into account all these memory reducing procedures when determining the minimal memory footprint of an application. Objects or memory pages that are rarely used, or that were only used on process startup, would otherwise inflate the memory usage metric.

All memory reducing procedures come with a runtime performance overhead. Loading and saving a frequently used memory region from and to disk, will slow down the process, which makes it unusable. That is one of the the reasons why swapping is not broadly supported on Kubernetes nodes [101]. If garbage collection has five times as much memory available as is required, its runtime performance matches or slightly exceeds that of explicit memory management. However, performance degrades substantially when it must use smaller heaps. With three times as much memory, it runs 17% slower on average. With twice as much memory, it runs 70% slower [97]. This slowdown, for example, is very noticeable for Golang applications that are running in a very constrained environment. Since execution advancement of an operator workload is mainly I/O-bound, most of the time, the operator is waiting for responses from the Kubernetes API server. A small slowdown of the execution might thus not directly cause the operator to advance slower. To detect execution slowdown because of memory limitations, Linux exposes a Pressure Stall Information (PSI) metric [102].

The memory utilization measure that we use is determined by limiting the memory usage until the application is being slowed down as determined by the PSI metric. Figure 8.2 shows an example run. The accumulated current memory value `memory.current` of a set of operators is displayed together with the accumulated `memory.high` cgroup v2 memory usage throttle limit. The figure shows that the throttle limit is updated based on the PSI metric. After triggering and completing a sequence of successful resource reconciliations, the synthetic test stops the resources updates. This simulates an idle operator and allows reclaiming of a big part of the memory resources. For each run, we selected memory usage samples from the active and idle period. The selections are the last 100 second that the operator is active or idle, to prevent measuring transitional behaviour at the start of the active or idle period. Since the WASM operators show a periodic behavior when idle, we selected 300 seconds, instead of 100 seconds, for those experiments as that roughly matches its periodicity. Figure 8.2 shows an example of such a selection.

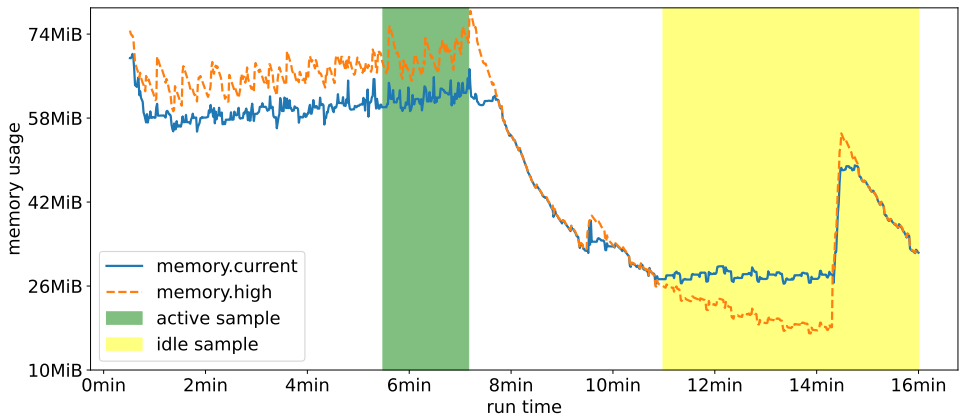
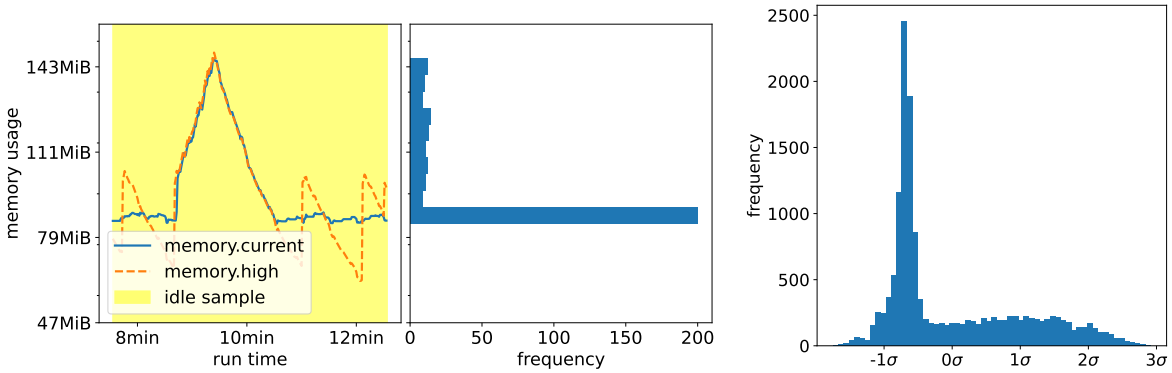


Figure 8.2: The memory usage of a single run for an operator, the operator idles after being active for 7 minutes; samples are selected such that transient behaviour is not captured.



(a) example run and frequency plot

(b) full frequency plot of all idle WASM operators

Figure 8.3: The distribution of the idle memory WASM operator on the right (b) is a mixture of Gaussians because of temporary memory increases such as shown in the figure on the left (a).

The memory measurements within a selected period are distributed as a mixture of Gaussians. This can be explained by temporary increases in memory usage combined with periods of stable memory usage, as is visible in Figure 8.3. Furthermore, we notice that within a run the memory usage samples are more strongly correlated than for samples across runs for the same configuration. In order to prevent obtaining an obscured view on our measurements, we eliminate the correlation by reducing our samples for one run to a single value. In order to also take into account the different distributions, we decided to reduce each run to a single 95% quantile measure. Each upper bound is defined as the memory limit that is not exceeded for 95% of the selected time range duration. Figure 8.4 shows these upper bounds for five runs that are part of the same configuration.

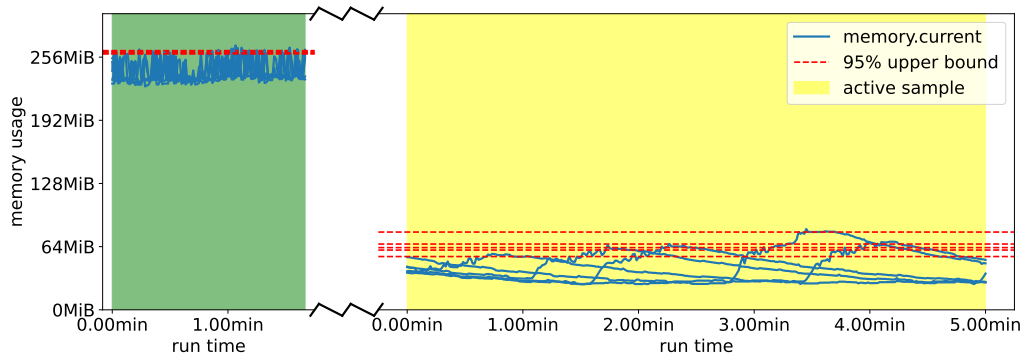


Figure 8.4: The memory 95% upper bounds for active and idle periods and the selected memory samples they are based on.

For each configuration, which is defined by an operator type and a number of operators, five independent runs were performed, each yielding one upper bound for the active and one for the idle period. Per operator type, we tested the number of operators from 10 to 100, in increments of 10. For the active and idle selection separate, based on the resulting 50 upper bounds $q_j : j = 1..50$ for each operator type, we trained a linear regression model. Using this linear model, we determined the 95% prediction interval in which we expect with 95% certainty the upper bound memory usage of a new run with

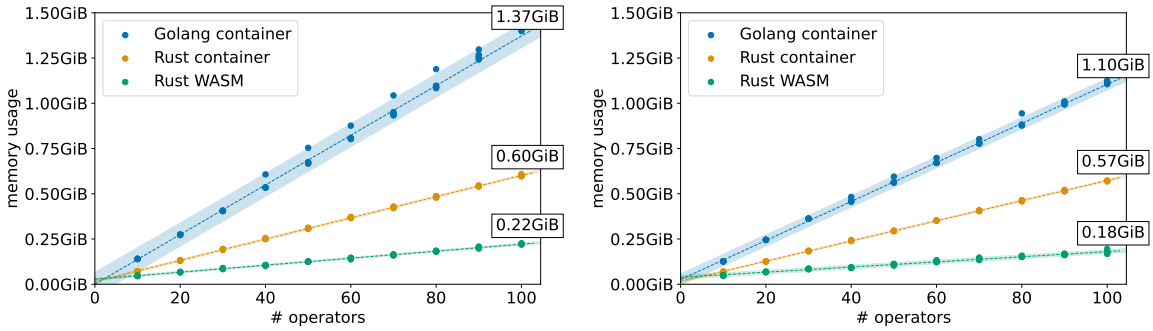
the given configuration. Formula 8.1, by Neter et al. [103], gives us the relation between this 95% prediction interval for an unknown upper bound Q_i of an new run i, and the predicted upper bound $\hat{q}_i$ for that run, with x_i the number of operators used in the configuration.

$$Pr\left(|Q_i - \hat{q}_i| < t_{97,5\%} * \frac{\sum_{j=1}^{50}(q_j - \hat{q}_j)^2}{50-2}\sqrt{1 + \frac{1}{50} + \frac{(x_i - \bar{x})^2}{\sum_{j=1}^{50}(x_j - \bar{x})^2}}\right) = 95\% \tag{8.1}$$

8.4 End-to-end latency measurement method and analysis

The end-to-end latency is measured by the synthetic-operator test for the active period of the test. Each set of reconciliations starts from an update of the `TestResource` in namespace 1 until the `TestResource` in namespace N is updated. The time from start to end is measured and each reconciliation set is repeated 500 times per run, resulting in $500N$ reconciliation iterations. As described in Section 8.3, for each configuration, which is defined by an operator type and a number of operators, five independent runs are performed.

8.5 Golang container, Rust container and Rust WASM compared



(a) active (b) idle

Figure 8.5: The memory 95% upper bounds of the different languages/ isolation techniques are ordered as follows: Rust WASM < Rust container < Golang container; all operators use less memory when idle.

Figure 8.5 shows the obtained memory upper bounds for container-isolated operators written in Golang and Rust and a WASM-isolated Rust operator. The coloured areas represent the 95% prediction intervals for the regression models as described in Section 8.3. Figure 8.5a shows the results for the active period. The Golang-based operator clearly uses the most memory. For 100 active operators, switching from Golang to Rust resulted in a 56.06% upper bound memory reduction. WASM operators even yielded an 83.81% reduction compared to Golang operators. Compared to Golang, the Rust operators use entirely different operator library and framework implementations. Each implementation has its own memory trade-offs, which can lead to large differences in memory usage. Additionally, as discussed in Section 8.3, garbage collected

languages like Golang, typically are less memory efficient than languages without garbage collector like Rust. The Rust container-based operator and the WASM-based operator share much of their source code. However, the WASM-based operators use less memory than container-based operators. This is due to the reduced complexity of the WASM child operator, as much of its low-level operator logic is moved to the parent operator. Moreover, the different isolation techniques used result in a net reduced isolation overhead, which is further explored in Section 8.6.

Figure 8.5b shows that, as expected, all operator types utilize less memory in case of idle workloads, we observed a 14.21% reduction on average. Compared to 100 idle Golang operators, 100 idle Rust operators utilized 48.04% less memory, which is a smaller reduction than when comparing active operators. However, 100 idle WASM operators still used 83.65% less memory compared to idle Golang operators, similar to the active situation. The smaller reduction in memory usage of container-based Rust operators versus Golang operators is due to Golang experiencing a higher relative reduction in memory consumption when going from active to idle. Based on the typical usage pattern of an operator, which can be idle for a long period of time, it is clear that idle memory usage is important.

8.5.1 End-to-end latency

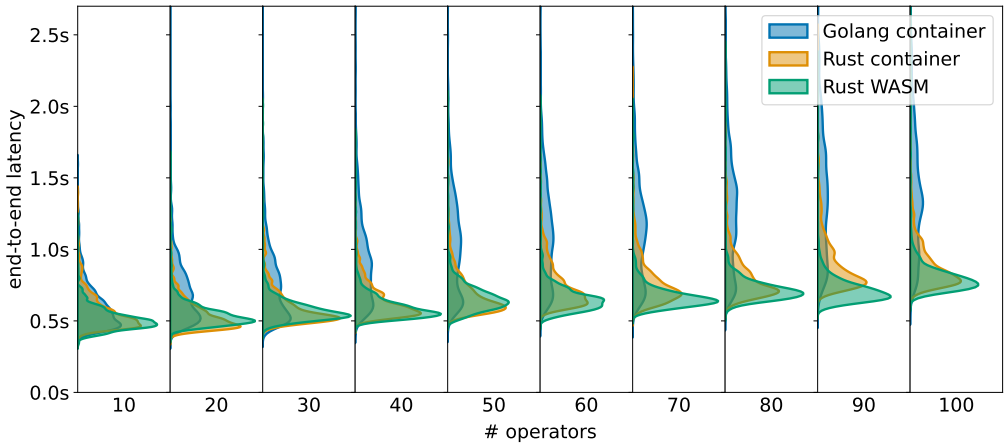


Figure 8.6: The end-to-end latency of WASM operators is identical to Rust operators.

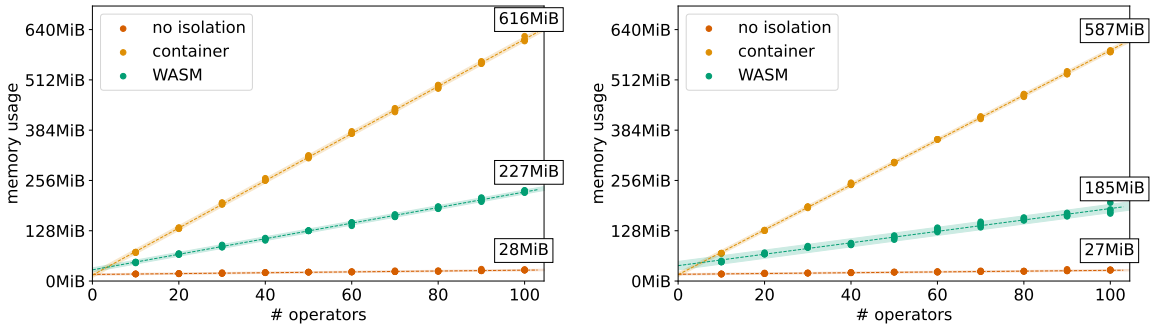
In Figure 8.6, the obtained latency distributions for the different operator types are displayed, which were obtained as described in Section 8.4. Based on Jangda et al. [104], WASM performance can be 2.5x slower worst-case compared to native execution. The WASM version of the synthetic-operator, however, did not experience any latency penalties. The latency for the Golang operator increased more than the other operators with increased number of operators. However, this is most likely due to the memory pressuring algorithm that adds more latency to Golang because its less memory efficient. There was no measured useful difference in latency between the WASM and Rust implementations that was greater than the measured noise. The main bottleneck in the operator's execution is I/O. Therefore, the latencies that occur in CPU-heavy workloads do not affect the synthetic-operator workload much. Additionally, if there would be a situation in which an operator workload suffers from such a slowdown, it is possible to expose the operation as a natively implemented host function from the runtime, since the WASM runtime is quite easily extendable.

8.6 Cost of isolation

Figure 8.7 shows the obtained memory upper bounds for Rust operators using no isolation, using containers and using WASM. The coloured areas represent the 95% prediction intervals for the regression models as described in Section 8.3. The solution with no isolation is the most resource efficient. This operator is able to scale to 100 control loops without significant additional memory overhead. Both the WASM-based and container-based setups experience significant per-operator overhead. Additionally, the WASM-based operator has a higher initial constant memory overhead. However, since the container-based solution performs worse per-container, this initial overhead can be compensated. In case of the active situation, the WASM-based solution is more memory efficient than the container-based solution with 95% certainty starting from six operators. For the idle operators, this starts from eight operators.

The container-based operators are managed by Kubernetes and each run in a separate Kubernetes pod. Our Kubernetes setup uses containerd [105] to manage the containers. As described in Section 2.2, each Kubernetes pod comes with some overhead. In our tests, the biggest overhead contributor was the per-pod *containerd-shim* process which equates to about 5MiB per pod. The WASM runtime can isolate the modules without introducing such a big overhead. Instead, it introduces a constant initial overhead that does not depend on the number of operators. This memory overhead is due to the WASM runtime, including the low-level operator logic.

In Chapter 6 we noted that the *kube-controller-manager* does not incorporate isolation between controllers, partly because this improves efficiency. Our tests showed that indeed a major memory usage reduction can be achieved by using no isolation. However, having no isolation between operators means that all operators should be fully trusted even for not having errors. Additionally it results in a lack of modularity: it is not possible to dynamically add or remove controllers. In an operator design based on Kubernetes pods, operators can be added and removed dynamically. Also, WASM modules can be loaded dynamically by the parent operator, without having to restart the parent operator process. WASM is a good intermediate solution, providing isolation and modularity while still being more memory efficient than the container-based solution.

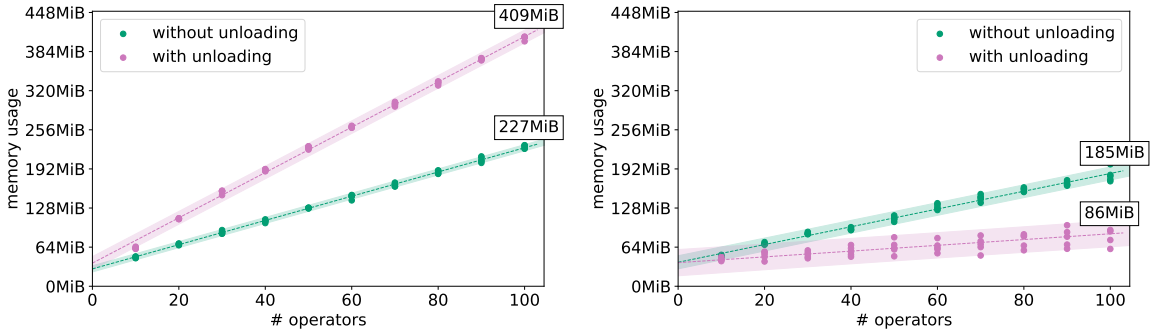


(a) active

(b) idle

Figure 8.7: The memory 95% upper bounds of non-modular, container-modular and WASM-modular operators show that WASM outperforms container-based isolation, but additional improvements are possible since having no isolation is still much more efficient.

8.7 Automatically unloading WASM modules



(a) active

(b) idle

Figure 8.8: The memory 95% upper bounds of the WASM operator with automatic unloading enabled/ disabled; very frequent unloading causes more memory usage, for idle operators it can save memory.

Figure 8.8 shows the obtained memory upper bounds for the synthetic-operator running as WASM modules. Two versions of the WASM operator are compared: one does not unload the WASM instances and the other unloads each WASM instance in-between each iteration of the reconciliation loop. The coloured areas represent the 95% prediction intervals for the regression models as described in Section 8.3. Figure 8.8a shows that the effect of constantly unloading and reloading active WASM operator was an 80.49% increase in memory usage for 100 operators. In Figure 8.9, the effect of actively unloading and reloading operators on the measured end-to-end latencies is displayed, this figure was obtained as described in 8.4. Figure 8.8b shows, running 100 operators, we achieved a 52.66% reduction for idle operators compared to not unloading.

Unloading the modules reduces memory usage in case of idle operators. The parent operator writes the memory of idle WASM instances to disk and reloads it later when a Kubernetes watch event is received, as described in Section 6.1. Since most operators often stay idle for a long time, this can greatly optimize resource utilization in memory-constrained environments. However, in case of a worst-case unload and reload pattern, memory usage is higher than in case no unloading and reloading takes place. Frequent unloading also introduces a large end-to-end latency penalty due to the disk overhead of swapping the WASM instance, as shown in Figure 8.9.

To properly benefit from automatic WASM module unloading in a mixed active-idle situation, a predictive scheduler is a necessity, this is considered as future work in this dissertation. Such a scheduler could help unloading only when it is beneficial to unload a WASM module instead of unloading it in-between each control loop iteration. The optimization opportunity also greatly depends on the heap memory allocated by the operators, necessary for Kubernetes API state caches. This relationship is further discussed in Section 8.8.

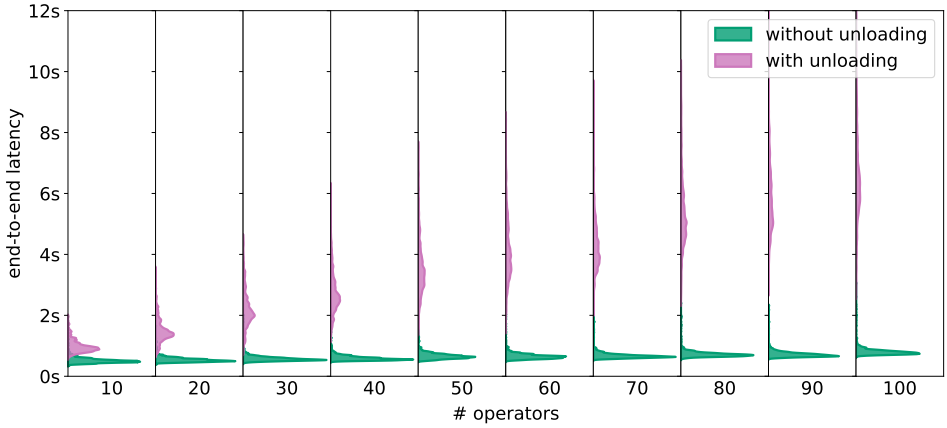


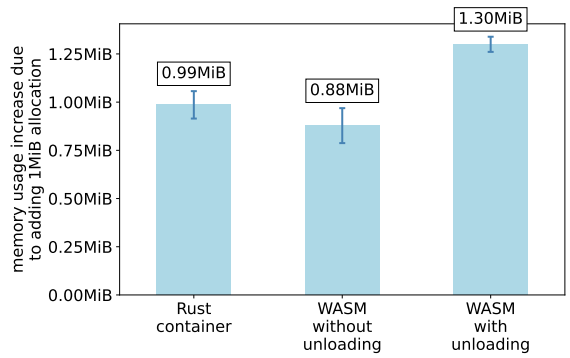
Figure 8.9: The end-to-end latency for active WASM operator with unloading enabled/ disabled. Actively unloading and swapping modules introduces significant latency.

8.8 Dynamically allocated memory

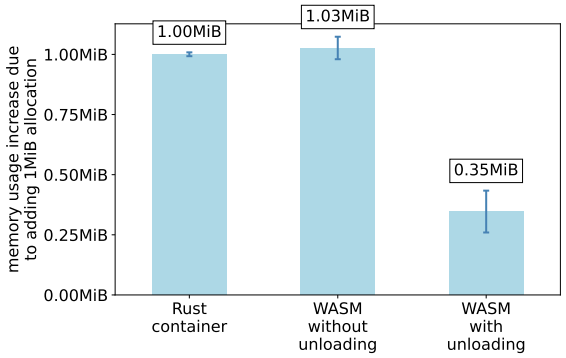
Figure 8.10 shows the average memory upper bound increase and average end-to-end latency increase per operator due to a 1MiB increase in dynamically allocated memory. The metric is obtained based on the slope of the linear regression models trained on 20 upper bound memory usage samples obtained for experiments with allocation sizes of 0MiB to 3MiB, with 5 runs per experiments. Also indicated are the 95% confidence intervals for these slopes.

Figure 8.10a shows that dynamically allocating 1MiB additional heap memory in each operator resulted in in a memory upper bound increase of roughly 100MiB for 100 active operators with unloading disabled, and in an increase of 130MiB for active WASM operators with unloading enabled. The 30MiB extra overhead originates from the additional memory required to reload the WASM module. Figure 8.10b shows that the memory consumption for idle operators only increased with 0.35MiB when using our unloading and swapping solution. This is significantly lower than the memory increases for operators without unloading and swapping. As discussed in Section 8.7, adding swapping also adds end-to-end latency. Figure 8.10c shows that, for our experiments, it took about 26ms to swap 1MiB of data to disk per operator, which can be fully attributed to the disk read and write overhead of the hard disk drive in the test server, listed in Section 8.2. No latency increase was experienced when using the containerised solution or the WASM solution without unloading.

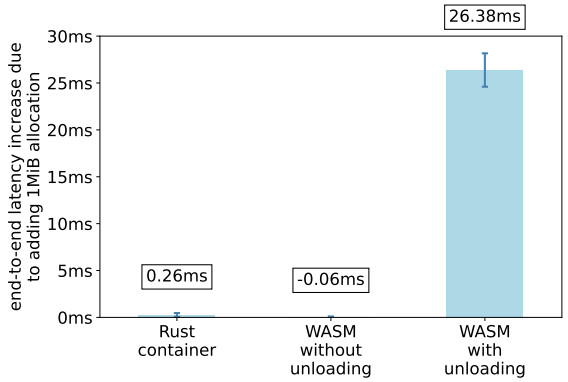
Operators that watch a large amount of Kubernetes cluster resources will typically keep many of these resources in a cache that they update once the Kubernetes API notifies that a resource change took place. This means that these operators have large amounts of dynamically allocated memory, which directly translates to a memory upper bound increase, as discussed in this section. To reduce this memory usage, it is possible to use our unloading implementation in combination with a tuned scheduler. However, such a solution will result in larger latency overhead due to disk writes. Using native Linux swapping could also further improve performance, since that swap implementation is more optimized than our custom swap solution. Another solution is to move all operator caches to the parent operator and to deduplicate the resources in these caches, this is further discussed in Section 9.



(a) The per-operator memory 95% upper bounds increase due to an extra 1MiB of dynamically allocated memory for active operators.



(b) The per-operator memory 95% upper bounds increase due to an extra 1MiB of dynamically allocated memory for idle operators.



(c) The per-operator end-to-end latency increase for active operators due to an extra 1MiB of dynamically allocated memory

Figure 8.10: The effects of allocating 1MiB of heap memory in each operator on the 95% upper bounds.

9

Open challenges and research directions

9.1 Implementation improvements

We have identified a number of open challenges that can directly improve our operator implementation. Firstly, as shown in Section 8.8, it is possible to reduce memory consumption by implementing a shared Kubernetes cluster state cache in the parent operator that replaces the individual caches in the child operators. This will eliminate duplicated and fragmented resource caches. It would also allow us to faster swap inactive child operators. Furthermore, a future caching algorithm can drop resources from cache in case memory becomes too limited. Secondly, we identified a limitation that originates from the Kubernetes pod resource constraints. Running more child operators in our WASM runtime directly results in higher memory usage, this was shown for all experiments in Chapter 8. However, since all operators run in a single Kubernetes pod, as described in Chapter 6, the resource constraints for the workload do not adapt dynamically when adding new operators dynamically. The resource constraints in Kubernetes are configured per pod. The operator architectures that run operators in separate pods, do have dynamically adapting constraints. As a workaround, our current implementation does not limit the resources for the WASM operator pod. Instead the parent operator and its embedded runtime are responsible for limiting the per-operator resources.

9.2 WebAssembly standards and proposals

A major open challenge identified as part of the dissertation is the shortcomings of the WebAssembly system interface specifications and libraries. As mentioned in Chapter 4, a missing feature in the WASI specification is support for socket handling [16]. The Golang standard libraries lack proper support for WASI if compiled using the default compiler. The TinyGo [96] compiler supports the WASM-WASI target but cannot perform serialisation due to its limited support for Golang reflection, as discussed in Chapter 7. Support for profiling or debugging WebAssembly is very limited. Many of the features are also lacking documentation and the overall tooling landscape is quite fragmented. This deteriorates the developer experience and will impact adoption, if not improved upon in the future. The standardisation groups that are working on improving this experience are organised as follows: "The open standards for WebAssembly are developed in a W3C Community Group (that includes representatives from all major browsers) as well as a W3C Working Group." [106] Both groups are mostly managed by representatives from Google. Additionally, the WebAssembly System Interface is standardised in a subgroup of the

WebAssembly W3C Community Group [17].

9.2.1 WASI sockets proposal

The WASI sockets proposal aims to standardise an interface for interacting with network sockets, like TCP and UDP sockets [16]. These sockets can be used to make network requests or setup a network server in WebAssembly. An important part of this API would be to limit the capabilities related to these sockets. Once this proposal is finalised and accepted, it can be used to replace some of the custom host functions that we currently use, as described in Section 7.3. When this proposal gets implemented, client operators will be able to make generic web requests, furthering the capabilities of WASM operators.

9.2.2 WASM component model proposal

As identified in Section 2.1, an advantage of WASM is that the runtime can be easily extended with extra host functions, exposing new functionality to the WASM module. This however still requires the host function to be updated. The WASM component model proposal [107] aims to add support for calling functions exposed by other WASM modules in a simple and dynamic manner. This would allow to place dependencies in separate WASM modules and deduplicate the logic for all modules that depend on them. This proposal, when available, could further help reduce resource utilization of our WASM operators.

An important part of the component model proposal is the ability to pass complex data values to and from WASM modules in canonical manner. This is the task of the Canonical ABI sub-proposal. This also prevents the repetitive task of writing (de)serializers in different languages for all possible client operator languages. Instead, these functions can be generated when required. An example implementation of this proposal can be found in the witx DSL language and code generators [17].

9.3 WASM compilers

As described in Section 9.4, there is still large fragmentation when it comes to the WASM compilers used by the most popular WASM-based projects. Each of these compilers has its own performance characteristics, with many of them claiming to be the fastest [108, 109]. The bytecode alliance develops three compilers: Wasmtime, Lucet [78], and WAMR. However, they promote Wasmtime as their flagship compiler, and WAMR as a solution for environments that are extremely resource limited. The Wasmtime compiler is developed mainly by Mozilla and Fastly and internally uses the cranelift compiler, while the WAMR compiler is created mainly by Intel. An alternative WASM compiler like WAMR, might further reduce the memory utilization of our implementation since such a compiler optimizes runtime overhead for extremely resource-limited environments. The WasmEdge compiler, formerly SSVM, was developed by SecondState, and is now part of the Cloud Native Computing Foundation (CNCF). Another frequently used compiler is Wasmer [92], which can switch between a LLVM and cranelift backend. Google's v8 Javascipt engine also supports WASM, and much of the research that Google has done on their Native Client

solution [34, 110] has been repurposed by WASM. Additionally to performance, support for the latest WebAssembly proposals, is a very important selection criterion, as described in Section 9.2. These new proposals can drastically improve the developer possibilities when working with WASM.

9.4 SFI-based FaaS

As mentioned in Chapter 3, Cloudflare Workers [5] and Fastly Compute@Edge [6] are currently the most used FaaS platforms based on SFI. Although these platforms are not fully open-sourced, the companies have heavily invested in improving the underlying technologies. Cloudflare collaborated with Graz University of Technology, Sapienza University of Rometo and CISPA Helmholtz Center for Information Security to create Dynamic Process Isolation [45] for example. Cloudflare's platform is build on v8, which is mainly developed by Google and helps to define new standards for Javascript and WebAssembly. Fastly open-sourced its WebAssembly-to-bytecode compiler as part of the Bytecode Alliance and helps to incorporate their findings in the other compilers that are open-sourced under the Bytecode Alliance, like Wasmtime. The offering of fully open-source SFI-based FaaS platforms is very limited. A startup called Fermyon [111], founded by the authors of Krustlet [12], has created an open-source alternative to the closed-source offerings of Cloudfare and Fastly. Their WASM FaaS solution, called Spin [85], uses Wasmtime as WASM-to-bytecode compiler. Another WASM-based FaaS platform is WasmCloud, which was developed by Cosmonic and donated to the Cloud Native Compute Foundation. WasmCloud utilizes Wasmer as WASM-to-bytecode compiler. Other open-source solutions originate from academic research, like Faasm and Sledge. Faasm [30] was proposed by Shillaker et al. from Imperial College London and Sledge [84] was proposed by Gadepalli et al. from George Washington University in collaboration with Arm Research. Faasm uses WAMR or WAVM and Sledge uses its own compiler, called aWsm. All these startups and projects are still under active development and still require a lot of additional features to cover the most frequent use-cases at the time of writing. Many of these possible improvements are linked to standards that are still in the process of being created, as described in Section 9.2.

Conclusion

Complex Kubernetes operator workloads are often too heavy for constrained environments. In this dissertation, a novel WebAssembly-based Kubernetes operator solution is proposed. This solution demonstrates that WebAssembly, a technology used by edge FaaS solutions, can also be used to reduce the overhead associated with Kubernetes cluster management. It therefore extends the Wasmtime runtime, adding support for asynchronous Kubernetes API interaction and unloading of idle operators. Our test results show a reduction in memory footprint of 100 active synthetic operators from 1405MiB to 227MiB and of 100 idle operators from 1131MiB to 86MiB by using WASM operators instead of traditional operators. This reduction is due to reduced child operator complexity and the lower WebAssembly isolation overhead. We also found that CPU overhead, identified as a drawback of WASM in prior work [104], does not affect end-to-end latency for our synthetic-operator workload. Unloading WASM operators reduces memory usage for idle operators, while increasing memory usage and end-to-end latency for idle operators. Therefore, future work is needed to add a predictive scheduler that fully optimizes this feature.

Our WASM architecture and implementation demonstrate that initiatives, such as the metacontroller project [86], can integrate a WASM runtime as an alternative to their current webhook solution and benefit from reduced complexity and resource usage. Resource-constrained edge environments are able to run more WebAssembly operators than traditional operators, enabling complex workloads. Cloud deployments become more resource efficient by replacing existing operators with WASM-based operators. The shared benefits of our solution across both edge and cloud segments help accelerate research and adoption.

The biggest open challenges for developing new WASM operators are the WASM and WASI specifications that are still under development. In addition, Golang lacks proper support for WASI, making it more difficult to write operators in Golang. However, Rust operators can more easily take advantage of running as WASM modules. We further propose to obtain additional reductions in memory usage by moving caching logic from the child to the parent operators.

Bibliography

[1] E. A. Brewer, "Kubernetes and the path to cloud native," in *Proceedings of the ACM 6th Symposium on Cloud Computing*. ACM, Aug. 2015, p. 167.

[2] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the 10th European Conference on Computer Systems*. ACM, Apr. 2015.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade," *ACM Queue*, vol. 14, no. 1, pp. 70–93, Jan. 2016.

[4] Shubham, C. Bühler, T. Bannister, and Q. Teng, "Kubernetes operator pattern," Mar. 2022. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

[5] Cloudflare, "Cloudflare workers®," Mar. 2022. [Online]. Available: https://workers.cloudflare.com/

[6] Fastly, "Fastly compute@edge," Mar. 2022. [Online]. Available: https://www.fastly.com/products/edge-compute/serverless

[7] B. W. Lampson, "Protection," *ACM SIGOPS Operating Systems Review*, vol. 8, no. 1, pp. 18–24, Jan. 1974.

[8] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[9] R. Mohr, D. Vossel, S. Gott, V. Romanovsky, and A. Lukianov, "Kubevirt.io," Dec. 2021. [Online]. Available: https://kubevirt.io/

[10] A. Randazzo and I. Tinnirello, "Kata containers: An emerging architecture for enabling MEC services in fast and secure way," in *Proceedings of the IEEE 6th International Conference on Internet of Things: Systems, Management and Security*. IEEE, Oct. 2019.

[11] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux virtual machine monitor," in *Proceedings of the Ottawa Linux Symposium*, 2007. [Online]. Available: https://www.kernel.org/doc/mirror/ols2007v1.pdf#page=225

[12] M. Fisher, "Krustlet," Apr. 2020. [Online]. Available: https://deislabs.io/posts/introducing-krustlet/

[13] R. Squillace, "WebAssembly meets Kubernetes with Krustlet," Apr. 2020. [Online]. Available: https://cloudblogs.microsoft.com/opensource/2020/04/07/announcing-krustlet-kubernetes-rust-kubelet-webassembly-wasm/

[14] M. Yuan, T. McCallum, S.-T. Hsieh, hydai *et al.*, "Crunw," Apr. 2022. [Online]. Available: https://github.com/second-state/crunw

[15] A. Rossberg, B. L. Titzer, A. Haas, D. L. Schuff, D. Gohman, L. Wagner, A. Zakai, J. F. Bastien, and M. Holman, "Bringing the web up to speed with webassembly," *Communications of the ACM*, vol. 61, no. 12, pp. 107–115, Nov. 2018.

[16] D. Bakker and L. Clark, "The wasi sockets proposal," Mar. 2022. [Online]. Available: https://github.com/WebAssembly/wasi-sockets

[17] D. Gohman, P. Hickey, L. Clark, A. Crichton, A. Brown *et al.*, "Wasi," Apr. 2022. [Online]. Available: https://github.com/WebAssembly/WASI

[18] A. Wilk, M. Klein, htuch, phlax *et al.*, "envoy," Apr. 2022. [Online]. Available: https://github.com/envoyproxy/envoy

[19] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *Proceedings of the ACM 2Nd European Conference on Computer Systems*. ACM, 2007, pp. 275–287.

[20] P. Hickey, A. Crichton, M. Frysinger, N. McCallum, J. Konka, and katelyn martin, "Wasi snapshot preview1," Jan. 2022. [Online]. Available: https://github.com/WebAssembly/WASI/blob/main/phases/snapshot/witx/wasi_snapshot_preview1.witx

[21] R. Dahl, B. Iwańczuk, K. Kelly, B. Belder, L. Casonato, and C. Beyer, "Deno," Mar. 2022. [Online]. Available: https://github.com/denoland/deno

[22] M. Kerrisk, "bpf-helpers(7) - linux manual page," Sep. 2021. [Online]. Available: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html

[23] L. Torvalds, dhowells, C. Brauner, and A. Lutomirski, "linux/syscall_64.tbl," Jan. 2022. [Online]. Available: https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl

[24] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *Proceedings of the ACM 18th Conference on Computer and Communications Security*. ACM, 2011, pp. 401–412.

[25] S. Soller, "Measurements of system call performance and overhead," Jan. 2017. [Online]. Available: http://arkanis.de/weblog/2017-01-05-measurements-of-system-call-performance-and-overhead

[26] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang, "Hyperkernel: Push-button verification of an os kernel," in *Proceedings of the ACM 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 252–269.

[27] Y. Shalabi, "The cost of virtualization exits," Dec. 2014. [Online]. Available: http://yshalabi.github.io/VMExits/

[28] R. Rosen, "Resource management: Linux kernel namespaces and cgroups," p. 70, May 2013. [Online]. Available: https://sites.cs.ucsb.edu/~rich/class/cs293b-cloud/papers/lxc-namespace.pdf

[29] M. Kerrisk, "namespaces(7) - linux manual page," Sep. 2021. [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html

[30] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *Proceedings of the USENIX 2020 Annual Technical Conference*. USENIX Association, Jul. 2020, pp. 419–433. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shillaker

[31] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5, pp. 203–216, Dec. 1993.

[32] A. Rudenko, Q. Monnet, D. Thaler, T. Graf, D. Borkmann *et al.*, "ebpf," Apr. 2022. [Online]. Available: https://ebpf.io/

[33] K. Zandberg and E. Baccelli, "Minimal virtual machines on iot microcontrollers: The case of berkeley packet filters with rbpf," *arXiv preprint arXiv:2011.12047*, vol. abs/2111.12047, Nov. 2020.

[34] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," *Communications of the ACM*, vol. 53, no. 1, pp. 91–99, Jan. 2010.

[35] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting fine grain isolation in the Firefox renderer," in *Proceedings of the USENIX 29th Security Symposium*. USENIX Association, Aug. 2020, pp. 699–716. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

[36] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan, "Gobi: WebAssembly as a practical path to library sandboxing," *arXiv preprint arXiv:1912.02285*, vol. abs/1912.02285, Dec. 2019.

[37] M. Kolosick, S. Narayan, E. Johnson, C. Watt, M. LeMay, D. Garg, R. Jhala, and D. Stefan, "Isolation without taxation: Near-zero-cost transitions for webassembly and sfi," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, Jan. 2022.

[38] G. Tan, "Principles and implementation techniques of software-based fault isolation," *Foundations and Trends® in Privacy and Security*, vol. 1, no. 3, pp. 137–198, 2017.

[39] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *Proceedings of the ACM 18th Conference on Computer and Communications Security*. ACM, 2011, pp. 29–40.

[40] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the IEEE 40th Symposium on Security and Privacy*, IEEE. IEEE, May 2019, pp. 1–19.

[41] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, and R. Strackx, "Meltdown: Reading kernel memory from user space," *Communications of the ACM*, vol. 63, no. 6, pp. 46–56, May 2020.

[42] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening WebAssembly against spectre," in *Proceedings of the USENIX 30th Security Symposium*. USENIX Association, Aug. 2021, pp. 1433–1450. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/narayan

[43] A. Barth, C. Jackson, and C. Reis, "The security architecture of the Chromium browser," Stanford University, Tech. Rep., 2009. [Online]. Available: http://css.csail.mit.edu/6.858/2018/readings/chromium.pdf

[44] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: Process separation for web sites within the browser," in *Proceedings of the USENIX 28th Security Symposium*. USENIX Association, Aug. 2019, pp. 1661–1678. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/reis

[45] M. Schwarzl, P. Borrello, A. Kogler, K. Varda, T. Schuster, D. Gruss, and M. Schwarz, "Dynamic process isolation," *arXiv preprint arXiv:2110.04751*, vol. abs/2110.04751, 2021.

[46] A. Starovoitov, "BPF backend," Dec. 2014. [Online]. Available: https://reviews.llvm.org/D6494

[47] J. Bastien and D. Gohman, "WebAssembly: Here be dragons," Oct. 2015. [Online]. Available: https://llvm.org/devmtg/2015-10/slides/BastienGohman-WebAssembly-HereBeDragons.pdf

[48] A. Decina, D. Tucker, W. Findlay, D. Everton *et al.*, "Aya," Apr. 2022. [Online]. Available: https://github.com/aya-rs/aya

[49] W. Huang and M. Paradies, "An evaluation of webassembly and ebpf as offloading mechanisms in the context of computational storage," *arXiv preprint arXiv:2111.01947*, vol. abs/2111.01947, 2021.

[50] Bikram, "Docker namespace vs cgroup," Oct. 2021. [Online]. Available: https://bikramat.medium.com/namespace-vs-cgroup-60c832c6b8c8

[51] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, Mar. 2015, pp. 171–172.

[52] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The true cost of containing: A gVisor case study," in *Proceedings of the USENIX 11th Workshop on Hot Topics in Cloud Computing*. USENIX Association, Jul. 2019. [Online]. Available: https://www.usenix.org/conference/hotcloud19/presentation/young

[53] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *Proceedings of the USENIX 12th Security Symposium*. USENIX Association, Aug. 2003, pp. 231–242. [Online]. Available: https://www.usenix.org/conference/12th-usenix-security-symposium/preventing-privilege-escalation

[54] T. Combe, A. Martin, and R. D. Pietro, "To Docker or not to Docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, Sep. 2016.

[55] L. Lamport, "On interprocess communication," *Distributed Computing*, vol. 1, no. 2, pp. 77–85, 1986.

[56] N. Natu and P. Grehan, "Nested paging in bhyve," in *2014 AsiaBSDCon*, 2014. [Online]. Available: https://people.freebsd.org/~neel/bhyve/bhyve_nested_paging.pdf

[57] 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 ACM 26th Symposium on Operating Systems Principles*. ACM, Oct. 2017.

[58] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[59] M. Jones, "Virtio: An i/o virtualization framework for linux," Jan. 2010. [Online]. Available: https://developer.ibm.com/articles/l-virtio/

[60] A. Milenkoski, B. D. Payne, N. Antunes, M. Vieira, and S. Kounev, "Experience report: An analysis of hypercall handler vulnerabilities," in *Proceedings of the IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 100–111.

[61] A. Milenkoski, M. Vieira, B. D. Payne, N. Antunes, and S. Kounev, "Technical information on vulnerabilities of hypercall handlers," *arXiv preprint arXiv:1410.1158*, vol. abs/1410.1158, 2014.

[62] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *Proceedings of the USENIX 17th Symposium on Networked Systems Design and Implementation*. USENIX Association, Feb. 2020, pp. 419–434. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache

[63] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 461–472, Mar. 2013.

[64] T. Bannister, Ray, Y. Yongsu, Q. Teng, and R. Hvaara, "Controllers," Jun. 2021. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/controller/

[65] P. Perzyna, "Kubernetes operators explained," Jul. 2020. [Online]. Available: https://blog.container-solutions.com/kubernetes-operators-explained/

[66] B. Rawat, "Kubernetes: Daemonset," Dec. 2021. [Online]. Available: https://blog.opstree.com/2021/12/07/kubernetes-daemonset/

[67] J. Munnelly, J. V. Leeuwen, M. Eyskens, I. Krumina *et al.*, "cert-manager," Apr. 2022. [Online]. Available: https://cert-manager.io/

[68] J. Witkowski and K. Korakitis, "The state of cloud native development," Jan. 2021. [Online]. Available: https://www.cncf.io/wp-content/uploads/2021/12/Q1-2021-State-of-Cloud-Native-development-FINAL.pdf

[69] J. Polites, "Life of a serverless event: Under the hood of serverless on google cloud platform (cloud next '18)," Jan. 2018. [Online]. Available: https://youtu.be/MBBQ6P3GauY

[70] F. Cavalcante and E. Laureano, "Azure functions internals - brk4020," Sep. 2018. [Online]. Available: https://youtu.be/9Ep6N4PtAxc

[71] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proceedings of the USENIX 2018 Annual Technical Conference*. USENIX Association, Jul. 2018, pp. 133–146. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/wang-liang

[72] Amazon Web Services, "Creating lambda container images," May 2022. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/images-create.html

[73] A. Srikanta and O. Filiz, "Aws fargate under the hood," Dec. 2019. [Online]. Available: https://d1.awsstatic.com/events/reinvent/2019/CON423-R1_REPEAT%201%20AWS%20Fargate%20under%20the%20hood_No%20Notes.pdf

[74] D. Poccia, "Introducing cloudfront functions – run your code at the edge with low latency at any scale," May 2021. [Online]. Available: https://aws.amazon.com/blogs/aws/introducing-cloudfront-functions-run-your-code-at-the-edge-with-low-latency-at-any-scale/

[75] Amazon Web Services, "Cloudfront faqs," May 2022. [Online]. Available: https://aws.amazon.com/cloudfront/faqs/

[76] R. Dahl, B. Iwańczuk, K. Kelly, B. Belder, L. Casonato, and C. Beyer, "Deno deploy," Apr. 2022. [Online]. Available: https://deno.com/deploy

[77] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript," in *Proceedings of the 21th International Conference on Financial Cryptography and Data Security*, A. Kiayias, Ed. Springer International Publishing, 2017, pp. 247–267.

[78] P. Hickey, "Announcing Lucet: Fastly's native WebAssembly compiler and runtime," Mar. 2019. [Online]. Available: https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime

[79] S. Vasani, T.-C. Chen, S. Sudake, Vishal *et al.*, "Fission," May 2022. [Online]. Available: https://github.com/fission/fission

[80] A. Ellis, B. Rheutan, V. K. Singh, L. Roesler, J. McCabe, R. Dimitrov *et al.*, "Openfaas," May 2022. [Online]. Available: https://github.com/openfaas/faas

[81] M. Moore, V. Agababov, M. Thömmes, J. Friedman, K. Nakayama *et al.*, "Knative serving," May 2022. [Online]. Available: https://github.com/knative/serving

[82] R. Rabbah, M. Thömmes, J. Dubee, C. Mehrotra, C. Santana *et al.*, "Openwhisk," May 2022. [Online]. Available: https://github.com/apache/openwhisk

[83] E. Duchan, L. BG, E. Nussbaum, O. Messer *et al.*, "Nuclio," May 2022. [Online]. Available: https://github.com/nuclio/nuclio

[84] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: A serverless-first, light-weight wasm runtime for the edge," in *Proceedings of the 21st International Conference on Middleware*. ACM, 2020, pp. 265–279.

[85] R. Matei, L. Martin, M. Noorali, I. Towlson *et al.*, "Spin," Apr. 2022. [Online]. Available: https://github.com/fermyon/spin

[86] A. Yeh, G. Głąb, Mike, J. X. Tee, S. Bartscher, L. Villard *et al.*, "Metacontroller," Apr. 2022. [Online]. Available: https://github.com/metacontroller/metacontroller

[87] C. Ferris and K. Schlosser, "controller-zero-scaler," May 2019. [Online]. Available: https://github.com/ibm/controller-zero-scaler

[88] D. Srinivas, J. Liggitt, J. Betz, P. Ohly *et al.*, "kube-controller-manager," May 2022. [Online]. Available: https://github.com/kubernetes/kube-controller-manager

[89] T. Cramer, xtutu, stephaneyfx, and I. Dmitrii, "The future trait - asynchronous programming in rust," Apr. 2022. [Online]. Available: https://rust-lang.github.io/async-book/02_execution/02_future.html

[90] T. Ramlot and F. Guardiani, "Wasm operator - master thesis project - optimising memory usage of kubernetes operators using wasm," May 2022. [Online]. Available: https://github.com/thesis-2022-wasm-operators/wasm_operator

[91] F. Guardiani and M. Thömmes, "Kubernetes controllers - a new hope," Jul. 2020. [Online]. Available: https://slinkydeveloper.com/Kubernetes-controllers-A-New-Hope/

[92] S. Akbary, I. Enderlin, M. McCaskey *et al.*, "Wasmer," Apr. 2022. [Online]. Available: https://github.com/wasmerio/wasmer

[93] E. Albrigtsen, T. K. Röijezon, kazk, M. Bagishov, R. Levick *et al.*, "kube-rs," Apr. 2022. [Online]. Available: https://github.com/kube-rs/kube-rs

[94] C. Lerche, A. Crichton, A. Ryhl, T. Endo, I. Petkov *et al.*, "Tokio," Apr. 2022. [Online]. Available: https://github.com/tokio-rs/tokio

[95] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. John Wiley & Sons, Inc., Apr. 2018, pp. 166–167.

[96] Ayke, R. Evans, Nia, sago35 *et al.*, "Tinygo - a go compiler for small places," Apr. 2022. [Online]. Available: https://tinygo.org/

[97] M. Hertz and E. D. Berger, "Quantifying the performance of garbage collection vs. explicit memory management," in *Proceedings of the ACM 20th Conference on Object Oriented Programming Systems and Applications*. ACM, 2005, pp. 313–326.

[98] T. Heo, "cgroupv2 memory," Oct. 2015. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html#memory

[99] C. Down, "5 years of cgroup v2: The future of linux resource control," in *Conference at 34th USENIX Large Installation System Administration Conference*. USENIX Association, Jun. 2021. [Online]. Available: https://www.usenix.org/conference/lisa21/presentation/down

[100] V. Biriukov, "How much memory my program uses or the tale of working set size," Sep. 2021. [Online]. Available: https://biriukov.dev/docs/page-cache/7-how-much-memory-my-program-uses-or-the-tale-of-working-set-size/

[101] E. Hashman, "New in kubernetes v1.22: alpha support for using swap memory," Aug. 2021. [Online]. Available: https://kubernetes.io/blog/2021/08/09/run-nodes-with-swap-alpha/

[102] J. Weiner, "Psi - pressure stall information," Apr. 2018. [Online]. Available: https://www.kernel.org/doc/html/latest/accounting/psi.html

[103] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman, *Applied Linear Regression Models*, ser. Irwin series in statistics. Irwin, 2005, ch. Chapter 2.6.

[104] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of WebAssembly vs. native code," in *Proceedings of the USENIX 2019 Annual Technical Conference*. USENIX Association, Jul. 2019, pp. 107–120. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/jangda

[105] M. Crosby, L. Liu, P. Estes, D. McGowan, S. Day, A. Suda *et al.*, "containerd," May 2022. [Online]. Available: https://github.com/containerd/containerd

[106] A. Rossberg, D. Gohman, L. Wagner, Ms2ger, B. Smith *et al.*, "Wasm - spec," Apr. 2022. [Online]. Available: https://github.com/WebAssembly/spec

[107] L. Wagner, D. Gohman, A. Crichton, N. Fitzgerald *et al.*, "Component model design and specification," May 2022. [Online]. Available: https://github.com/WebAssembly/component-model

[108] Y.-Y. He, S.-T. Hsieh, hydai, X. Liu, M. Yuan *et al.*, "Wasmedge," Apr. 2022. [Online]. Available: https://github.com/WasmEdge/WasmEdge

[109] V. Shymanskyy, S. Massey, M. Graey, I. Grokhotkov *et al.*, "Wasm3," Apr. 2022. [Online]. Available: https://github.com/wasm3/wasm3

[110] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary CPU architectures," in *Proceedings of the USENIX 19th Security Symposium*. USENIX Association, Aug. 2010, pp. 1–11. [Online]. Available: https://www.usenix.org/conference/usenixsecurity10/adapting-software-fault-isolation-contemporary-cpu-architectures

[111] M. Butcher, A. Reese, B. Hardock, I. Towlson, M. Fisher, M. Dhanani, R. Matei *et al.*, "Fermyon technologies," Apr. 2022. [Online]. Available: https://www.fermyon.com/