Master Thesis Topic Proposal

Improving A Distributed Workflow Engine With A WebAssembly Runtime

Felix Seidel

This master thesis proposal extends a distributed workflow engine with a WebAssembly runtime. This improves workflow execution performance figures, such as resource utilization, latency, and throughput. The resulting solution will provide a developer-friendly workflow and provide interoperability between Linux containers and WebAssembly modules in the same workflow.

Using WebAssembly in a cloud computing environment benefits users because it provides a third containment option besides Virtual Machines and Linux containers. WebAssembly promises very efficient resource utilization combined with a high level of tenant isolation. Workflow engines profit benefit from this particularly because of their tasks' ephemeral nature. This document first provides context on the application domain. Then it describes the WebAssembly runtime architecture and its integration into a distributed workflow engine. Last, it gives a use case example for benchmarking.

1 Context

Efficient tenant isolation is the primary motivation for this thesis. In the next section, we will introduce WebAssembly and its use cases. Last, we explain workflow engines and container orchestration.

1.1 Cloud Tenant Isolation

Cloud computing provides benefits to its users in two ways. It makes computing resources accessible easier in that it abstracts hardware specifics and provides a common user interface instead. Moreover, the cost per computing resource is lowered because cloud computing allows physical hardware to be efficiently shared between tenants.

However, the economic benefits of shared hardware come with additional challenges. The cloud infrastructure provider must ensure that each tenant gets a fair

share of computing resources. Even more importantly, they must isolate each tenant to provide data security.

When cloud computing resources emerged, Virtual Machines (VMs) were the first available mechanism. VMs provide a strong level of isolation but come with a resource and management overhead. Linux containers reduced the overhead considerably, but do not provide the same isolation level. Existing research has been done to bring the two mechanisms closer together and combine their advantages. With lightweight VMs and application-level kernels like Firecracker [1] and gVisor [2], the distinction between the technologies becomes increasingly blurred.

WebAssembly promises to provide a third method with a simpler implementation, a high level of security and close-to-zero overhead. However, in contrast to VMs and containers, WebAssembly as a runtime in the backend is in its infancy and basic tooling has been implemented only recently.

1.2 WebAssembly – a bytecode for the Web

WebAssembly is a portable, low-level bytecode format for executable programs [3]. It abstracts hardware-specifics, making it independent of a programming language, hardware, or platform.

WebAssembly modules execute at near-native speed [4]. However, compared to native programs, they can be efficiently embedded into existing runtimes and are considered secure because of their sandboxed execution environment [5].

Essentially, WebAssembly serves as a common compilation target and runtime environment for a variety of programming languages [4]. Each language with an LLVM frontend can be compiled into WebAssembly, for example, C, C++, Rust, and Go.

Even though WebAssembly was designed initially for efficient and secure execution of programs within Web browsers, its use cases expand beyond the Web. It utilizes common hardware capabilities on many platforms, from powerful data center servers to edge and IoT devices.

1.3 Use Cases for WebAssembly beyond the Web

Now that Web browser support for WebAssembly is widespread, researchers and practitioners explore headless use cases which do not involve a browser.

There are many incentives to use WebAssembly in backend and server systems, too.

They have significant advantages over more established tenant isolation technologies in cloud infrastructure. Because they execute in a lightweight virtual machine, they do not necessitate the overhead of a separate operating system kernel or file system. Moreover, the sandboxed execution environment has limited and more easily guarded interfaces to the outside world. WebAssembly modules are a good fit for an environment where untrusted code is executed on shared hardware, such as cloud or edge computing applications.

For example, Shillaker et al. described a system for executing serverless computing workloads efficiently, which uses WebAssembly as its execution environment [6]. Li et al. propose a framework allowing developers to create integrated software programs which run the same code on a wide range of hardware, from datacenters to the network edge and IoT devices [4]. Their work is the first step towards ready-to-use WebAssembly runtimes in the cloud. However, the community did not implement tools to run WebAssembly with a good developer experience in container orchestration systems and workflow engines.

1.4 Workflow Engines & Kubernetes

A workflow is "a model for complex processes in which the process is decomposed into discrete, sequential operations" [7].

For many practical use cases, computational workflows have many heterogeneous components [8]. In the following, these components are called tasks. The heterogeneity between the tasks is because of differences in their programming language, runtime dependencies, first-party or third-party origin, and input/output interface definition.

This reduces workflow portability, sharing, and re-use. Some of these challenges resolve themselves when workflow tasks are packaged into standardized, self-contained packages, such as Linux container images.

Kubernetes [9] is popular software that solves this problem in the domain of distributed computing. It is an open-source container orchestration platform developed by Google. The orchestrator manages the lifecycle of Linux containers, which is one solution to the portability problem for backend workloads. Using Kubernetes, containers can be run reliably and at scale.

To run workflows, users deploy them to a workflow engine. This software allows them to manage workflows, create workflow instances, and retrieve workflow results.

Most workflow engines can run in Kubernetes. However, most of them have an architecture that does not incorporate container orchestration into their core concept. Specifically, they implement task distribution and scaling *on top of*, not *with* Kubernetes. In contrast, Argo Workflows is a workflow engine that tightly integrates with the Kubernetes API [10]. It adds a data model to Kubernetes, allowing users to describe and run workflows using Kubernetes itself. Argo's primary use cases are workflows for Data Science, Machine Learning, Extract-Transform-Load (ETL) processes, Batch Processing, and Continuous Integration/Delivery.

2 Thesis Topic

This master thesis integrates WebAssembly modules with a distributed workflow engine. The primary motivation for this research is the assumption that key performance figures improve when WebAssembly modules replace Linux containers.

The container runtime isolates containers. To do so, it manages a considerable number of resources for each container, such as a process and network namespace, an IP address, and allocates resources for the user space processes. In Kubernetes clusters, this overhead may further increase because the orchestrator adds sidecar containers (e.g., service mesh proxies). Container management adds cost in terms of CPU and memory consumption and adds latency to the workflow execution.

In contrast, a WebAssembly runtime can spawn a module with very little overhead. The performance characteristics and behavior of workflow tasks implemented as WebAssembly modules should be comparable to the benchmarks by Shillaker et al. [6, p. 428]. The reduced overhead increases the capacity of the system due to improved resource utilization. Moreover, it reduces the workflow latency, i.e., workflows complete faster. For a fair comparison, the WebAssembly runtime must provide the same or a better level of isolation between each running module.

For interoperability, the solution must meet two sub-goals. First, existing APIs and integrations, such as workflow task artifacts and output logs, must remain compatible between the container and WebAssembly runtime. Second, users should use the same tools to make the modules available to the workflow

engine like for containers. Specifically, the system should be able to retrieve both container images and modules from the same OCI (Docker) registry.

In addition to end-user usability aspects, important goals with respect to the WebAssembly runtime integration into Kubernetes must be met. The target architecture should be native to a distributed system. Thus, it will insert another runtime layer on top of containers, as getting around the container management overhead is one important motivation to integrate WebAssembly in the first place. On the other hand, to keep the system simple and easy to comprehend, the runtime layer must be built based upon Kubernetes primitives. This approach also ensures that the solution is re-usable and likely accepted by the Kubernetes and Argo communities.

The Krustlet [11] project shows an interesting solution to this problem. Krustlet implements the Kubelet API such that it presents itself as another node to the Kubernetes cluster. WebAssembly programs present themselves to the cluster as pods, which is the same way as containers do. WebAssembly pods use the same scheduling and management facilities as regular containers. This is useful for the WebAssembly workflow runtime to re-use existing distribution and scaling functionality. While Krustlet already supports scheduling WebAssembly workloads as Kubernetes pods, an efficient remote module invocation API must be implemented. Moreover, means for horizontal scaling are necessary. For this, one can either implement this feature within Krustlet, or re-use standard tooling such as the Horizontal Pod Autoscaler [1].

The WebAssembly runtime for Kubernetes is independent of a specific workflow orchestrator. However, the thesis will focus on Argo Workflows since Argo's design goals are well-aligned with ours in that both approaches achieve a Kubernetes-native solution, in contrast to building a custom implementation around Kubernetes. Argo creates a new container for each workflow task with the existing executor. With the extension, users can instead specify a WebAssembly module as the execution target for a workflow task. As a design goal, both implementations should be interoperable, so the workflow designer should not need to differentiate between a container and a module.

An extension point for the workflow engine integration is adding a Just-in-Time (JIT) compiler for WebAssembly modules. Workflows often include custom logic that cannot be generalized ahead of time to be compiled and uploaded to a module registry. Compared to containers, WebAssembly modules have a small file size and compile quickly.

[1] `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/` (retrieved on 2021-11-21)

The JIT compiler would pick up custom logic when the workflow is submitted to the workflow orchestrator, then compile and upload a module into a registry where it is picked up during workflow execution. Containers running a scripting language (e.g., Python) can achieve the same behavior. However, compiled WebAssembly modules run at near-native speed and are orchestrated more efficiently, as explained in the previous sections. If JIT-built workflow steps were to be implemented with the existing Argo Workflows architecture, container images would have to be built for each custom workflow step. Container images are computationally expensive during build time and adds overhead to the workflow execution. The added overhead is because the system must distribute the custom-built images and retrieve them before execution. A small WebAssembly module can achieve this with less overhead than required for a container.

3 Use Case

The WebAssembly module runtime will be demoed and evaluated based on a document processing workflow as an exemplary use case. This workflow demonstrates the developer experience for WebAssembly in the Argo workflow engine. The following section describes the document processing workflow in detail. Moreover, a benchmarking concept is introduced. Last, we explain how we will demonstrate the workflow and the benchmarks in a presentation.

3.1 Document Processing Workflow

The use case depicted in Figure 1 implements a document processing workflow for scanned images. It is derived from the Argo Workflows examples library [2]. The workflow transforms an incoming document into an optimized and searchable archive format that is suitable for storage in a Document Management System (DMS). First, the pages of an incoming document are split apart. For each page, it optimizes the scan, performs Optical Character Recognition (OCR), extracts addresses, and enhances found information using online geocoding and tokenization. Last, pages are merged and sent to the DMS.

The workflow has some I/O-heavy tasks such as loading a document from external storage, geocoding addresses with an online web service, and saving a document to an external document management system. These workflow tasks

[2] `https://github.com/argoproj/argo-workflows/tree/master/examples` (retrieved on 2021-11-21)

are probably not suited for WebAssembly since it currently lacks universal APIs for network access.

A sub-workflow can be run data-parallel for each page in the document. Substantial performance gains are to be expected when the majority of the tasks are transformed to WebAssembly, compared to when the same tasks run as containers. For a 10-page document, 64 workflow tasks execute, of which 52 can be replaced with WebAssembly module runs.

Figure 2 shows an example of a YAML document with a workflow that consists of a workflow step based on a container image, one based on a WebAssembly module and a JIT-compiled step.

This workflow will be implemented with Argo Workflows in two variants. The baseline variant implements each task with a container. Contrary, the improved variant replaces containers with WebAssembly modules for the tasks highlighted in green in Figure 1.

3.2 Evaluation

Both a qualitative and quantitative evaluation of the use case example will be conducted. On the qualitative side, it identifies challenges for a good developer experience with WebAssembly modules on Kubernetes. For the quantitative evaluation, a performance benchmark will be conducted. The target metrics follow the ones utilized by Shillaker et al. [6, p. 427]:

1. Execution time over the number of parallel workflow instances
2. Task latency (i.e., time until a given workflow task starts processing – using a distributed tracing system[3])
3. Task throughput (i.e., the number of active workflow tasks at a single point in time)
4. Hardware utilization (measured as billable time of active memory – this is how cloud providers usually bill serverless functions)

3.3 Demonstration

The use case demonstration consists of three sections.

The first section introduces the Argo workflow engine and the developer experience for using it together with WebAssembly modules. Specifically, the audience

[3] `https://opentracing.io` (retrieved on 2021-11-24)

will see how the module integrates with the whole system, e.g. for using input and output parameters and artifacts. We use a Golang WebAssembly module which implements a part of the document processing workflow as an example.

The second section demonstrates the system's runtime behavior and performance. Based on the tooling we implemented for the performance benchmarks, workflow instances are submitted to the cluster. We show the WebAssembly module invocations are invoked, the horizontal scaling capabilities of the WebAssembly runtime, and how this is presented to the user in the Argo UI.

Last, we discuss the implications for developer workflows. Based on the document processing workflow, we can show which aspects of WebAssembly are a good fit for distributed workflow processing, and which aspects can still be improved in the future. For example, with networking APIs still missing in WebAssembly, certain tasks are hard to implement whereas WebAssembly is a very good fit for compute-heavy tasks that are invoked many times.

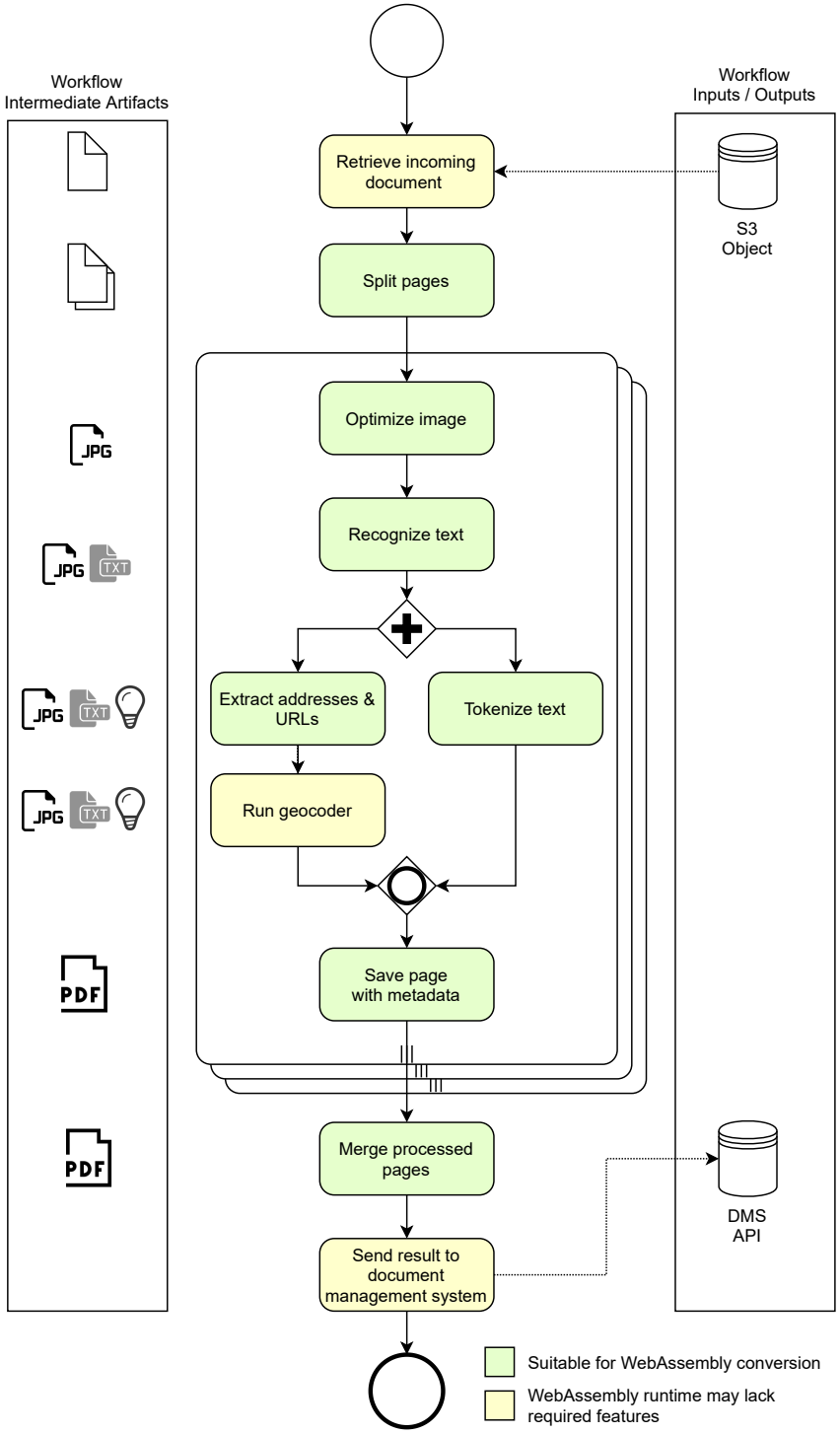

Figure 1: Use Case Progress Diagram. Depicts the document processing workflow as a BPMN process. The sub-process inside the highlighted parallel task can be worked on data-parallel per page. Tasks which are suitable for WebAssembly replacement are highlighted green.

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: incoming-document
spec:
  entrypoint: main
  templates:
  - name: main
    dag:
      tasks:
      - name: retrieve
        inputs: {parameters: [{name: object_id, value: 2021/11/18/0001.zip}]}
        container:
          image: ghcr.io/Shark/document-processor/s3
          args: ["download", "{{inputs.parameters.object_id}}"]
        outputs: {artifacts: [{name: document, path: /tmp/document.zip}]}

      - name: split
        dependencies: ["retrieve"]
        inputs: {artifacts: [{name: document, path: /tmp/document.zip}]}
        wasm_module:
          image: ghcr.io/Shark/document-processor/split_pages
          args: ["/tmp/document.zip"]
        outputs: {artifacts: [{name: pages, path: /tmp/pages/*.tiff}]}

      - name: filter
        dependencies: ["split"]
        arguments: {parameters: {name: document, value: "{{item}}"}}
        wasm_code:
          language: golang
          code: |
            func handle(in *workflow.Input) (workflow.Output, error) {
              name, ok := in.Arguments["name"]
              if !ok {
                return nil, errors.New("name argument not given")
              }
              doc, err := analyzer.Load(name)
              if err != nil {
                return nil, err
              }
              // Drop document if it is older than one month
              filtered := time.Now().Sub(doc.CreatedAt) > time.Hour * 24 * 30
              return &workflow.Output{
                "is_filtered": filtered
              }, nil
            }
        withItems: "{{steps.split.outputs.artifacts.pages}}"
        output: {parameters: [{name: is_filtered}]}
```

Figure 2: Abbreviated Use Case Process as Argo Workflows YAML template. Note how the retrieve and split steps use a container and a WebAssembly module, respectively. Apart from the key, both are used interchangeably and with an image from the same OCI repository. The `filter` step gives an example for a JIT-compiled workflow task.

Bibliography

[1] A. Agache, M. Brooker, A. Iordache, *et al.*, "Firecracker: Lightweight virtualization for serverless applications", in *17th {usenix} symposium on networked systems design and implementation ({nsdi} 20)*, 2020, pp. 419–434 (cit. on p. ii).

[2] 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 *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019 (cit. on p. ii).

[3] A. Haas, A. Rossberg, D. L. Schuff, *et al.*, "Bringing the web up to speed with WebAssembly", in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, Barcelona, Spain: Association for Computing Machinery, Jun. 2017, pp. 185–200 (cit. on p. ii).

[4] B. Li, W. Dong, and Y. Gao, "WiProg: A WebAssembly-based approach to integrated IoT programming", in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, May 2021, pp. 1–10 (cit. on pp. ii, iii).

[5] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of webassembly", in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 217–234 (cit. on p. ii).

[6] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing", in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 419–433 (cit. on pp. iii, iv, vii).

[7] M. R. Gryk and B. Ludäscher, "Workflows and provenance: Toward information science solutions for the natural sciences", en, *Libr. Trends*, vol. 65, no. 4, pp. 555–562, 2017 (cit. on p. iii).

[8] M. R. Crusoe, S. Abeln, A. Iosup, *et al.*, "Methods included: Standardizing computational reuse and portability with the common workflow language", May 2021. arXiv: 2105.07028 [cs.DC] (cit. on p. iii).

[9] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes", *Commun. ACM*, vol. 59, no. 5, pp. 50–57, Apr. 2016 (cit. on p. iii).

[10] *Argo workflows - the workflow engine for kubernetes*, https://argoproj.github.io/argo-workflows/, Accessed: 2021-11-16 (cit. on p. iv).

[11] *Krustlet*, https://krustlet.dev, Accessed: 2020-11-18 (cit. on p. v).