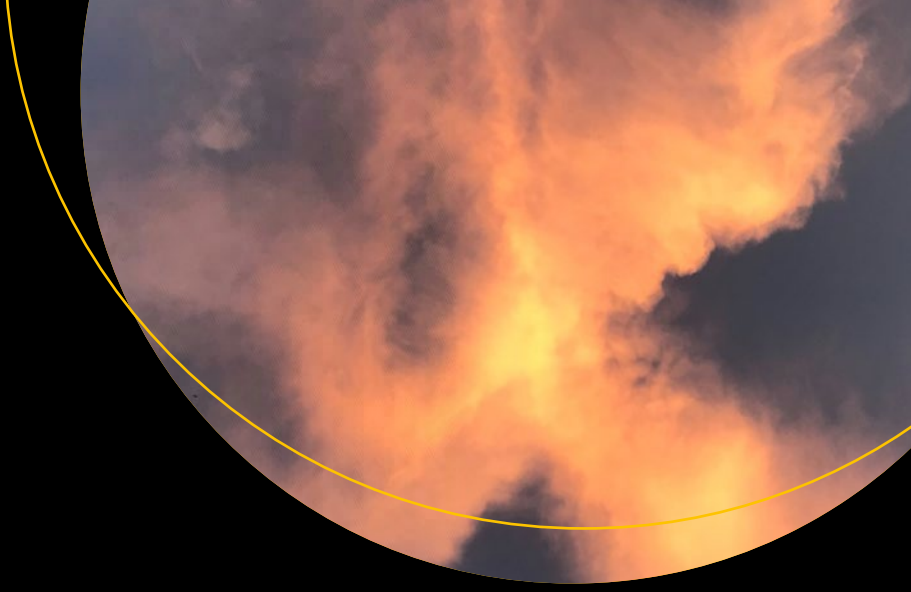

WebAssembly for Cloud

A Basic Guide for Wasm-Based Cloud Apps

Shashank Mohan Jain

Apress®

WebAssembly for Cloud

A Basic Guide for Wasm-Based Cloud Apps

Shashank Mohan Jain

Apress®

WebAssembly for Cloud: A Basic Guide for Wasm-Based Cloud Apps

Shashank Mohan Jain
Bangalore, India

ISBN-13 (pbk): 978-1-4842-7495-8 ISBN-13 (electronic): 978-1-4842-7496-5
https://doi.org/10.1007/978-1-4842-7496-5

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Spandana Chatterjee
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Jas Le on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484274958. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

I dedicate this book to my parents and their blessings, without which this book was not at all possible.

I also dedicate this book to my dear wife. I would not have been able to write it without her constant pushing and support.

I appreciate my angel of a daughter for allowing me the time to write this book.

Finally, I thank a dear friend who constantly pushed me into writing this.

Table of Contents

About the Author

Shashank Mohan Jain has worked in the IT industry for 20 years, mainly in cloud computing and distributed systems. He has a keen interest in virtualization techniques, security, and complex systems.

Shashank has more then 30 software patents in cloud computing, IoT, and machine learning. He has been a speaker at many cloud conferences. In addition, he holds Sun, Microsoft, and Linux kernel certifications.

About the Technical Reviewer

Srinivasa Reddy Challa is an expert developer at SAP. He has experience developing applications in various programming languages, including Java, Kotlin, Node.js, Rust, Golang, and Python, and frameworks like Spring, Django, and Express. Srinivasa also has extensive experience working with cloud providers like AWS, Azure, and AliCloud and has cloud certification in AWS. He has a bachelor's degree in computer science engineering.

Acknowledgments

I would like to acknowledge Kevin Hoffman, whose work in WebAssembly is an inspiration. Kevin is the creator of the waPC library, which is used in a chapter in the book.

Introduction

Somewhere, something incredible is waiting to be known.

—Carl Sagan

I start this journey with a quote from the eminent scientist and science communicator Carl Sagan. This short book introduces the amazing world of WebAssembly. The book's main theme is to create a simple WebAssembly program from scratch and take it to the cloud. In doing this, you'll gain a solid introduction to the valuable features offered by WebAssembly. Consider this book an introduction to WebAssembly and how it is powering browser-based applications and cloud applications. 'To get the most out of this book, you should have a bit of understanding of cloud fundamentals and basic knowledge of programming languages like Rust, golang and javascript.'

CHAPTER 1

WebAssembly Introduction

Before introducing WebAssembly, it's important to get a brief history of virtualization to better understand the context of WebAssembly.

When VMware started the virtualization revolution, virtual machines were positioned as the unit of computation. This meant that you could create and deploy software compatible with a virtual machine (VM). The VM-based approach provided great isolation because it introduced a kernel boundary between software and the host on which the workload ran (called a *hypervisor*). Although they were secure, VMs were heavy in nature and took time to spin up.

As cloud technology progressed, we saw the advent of *container-based virtualization*, which was mainly facilitated by structures within the Linux kernel. Containers on the same host shared the Linux kernel but have adequate mechanisms for security, like namespaces, seccomp profiles, and SELinux, which offered multilayered security for containers. In 2018, a new technology called WebAssembly has emerged. It was created by Mozilla and started as a browser-based technology. Since then, developers have employed it on the cloud and server-side apps. WebAssembly allows an extra level of virtualization by running the Wasm computation within a Linux process.

Things began with virtual machines (which are complete operating systems) and then moved to Linux containers (Linux processes protected and isolated by the Linux kernel). Now there is WebAssembly,

S. M. Jain, *WebAssembly for Cloud*, https://doi.org/10.1007/978-1-4842-7496-5_1

a computation unit within the Linux process. The goal is to provide a computation unit that can quickly spin up and be suitable for serverless workloads.

WebAssembly (also known as Wasm) is the new universal bytecode for interoperable compute units. *Interoperable* means that the compute unit should be able to run on any compatible Wasm runtime. A *compute unit* is a Wasm module. The basic idea is to have a bytecode format that is universal and standard.

Languages like JavaScript, Rust, Golang, and Java can be compiled to a Wasm-based bytecode. Once this bytecode is generated, it can be executed on any Wasm runtime.

Wasm is a small and efficient stack-based virtual machine that abstracts the target architecture by compiling the code to a universal bytecode representation. Wasm is based on an industry-wide collaborative effort to get a performant and secure close to assembly language. The Bytecode Alliance, set up to create shared implementations of WebAssembly standards, includes major players like Arm, Intel, Google, Microsoft, Mozilla, and Fastly.

Wasm is also well suited to run code in a multitenant way because it has the right security primitives built into it. Since it's launched within a process but is not a process itself, it also provides a means to avoid cold start problems, which are typical of serverless environments. Wasm is gaining tractions in areas like

- Providing data filtering capabilities in case of gateways like Envoy
- Policy engines like Open Policy Agent
- Kubernetes admission controller
- Databases like Postgres with custom extensions supporting Wasm

Wasm is now seen as a forefront technology in the cloud-native community. According to the Cloud Native Computing Foundation's CTO, Chris Aniszczyk, "Any project that has an extension mechanism will probably take advantage of Wasm to do that."

> *The promise, and excitement, is around a mix of portability and speed.*
>
> —Fintan Ryan, a senior analyst at Gartner

With low resource overhead and speed up in startup time compared to JavaScript, Wasm can be provisioned on IoT devices with resource-constrained memory, CPU, and storage. With no cold start issues, the portability and low resource consumption would make WebAssembly ideal for serverless deployments on the cloud and the edge. Initially started as a sandboxing technique for browser-based applications (for example, running image processing, decoding video and audio on the browser), it has now made inroads into server-side technologies due to powerful sandboxing capabilities and low overhead.

The security capabilities of Wasm make it a good fit for preventing security vulnerabilities like buffer overflows and control flow integrity issues. Wasm separates code and data. It has a static type system with type checking and a very structured control flow designed to make it easier to write code that compiles to be safe, with linear memory, global variables, and stack memory accessed separately. These aspects are discussed in the later chapters in regards to how Wasm provides neat mechanisms to avoid such security challenges.

Under the hood, Wasm runtime is a stack-based virtual machine operating on the Wasm bytecode by pushing and popping data off the stack. The closest comparison would be to the working of a JVM. One major difference is that JVM bytecode isn't universal (i.e., only programming languages like Kotlin and Scala can be compiled as Java

bytecode). But, almost all the programming languages like C, C++, Rust, Golang, and JavaScript can be compiled into Wasm bytecode.

Wasm currently only supports numeric data types, although there's a proposal to add reference types like strings, sequences, records, variants to make it easier for Wasm modules to interact with modules running in other runtimes or written in different languages. Though this is not a limitation, other data types, such as strings, can still be realized with these numeric types, just that it makes programming Wasm directly a bit tedious. A Wasm module doesn't have access to APIs and system calls in the OS. If you want it to interact with anything outside the module, you must explicitly import it, so the only code that could be executed is the code that is packaged as part of the module. This interaction with the operating system calls is facilitated by a new spec known as WASI (WebAssembly System Interface). The WASI spec allows an interoperable Wasm code that can be ported to any Wasm runtime (i.e., runtimes like Lucet, Wasmer, and Node.js) once the Wasm compiler generates the bytecode.

Wasm in the Cloud

There are differences in running Wasm in a browser vs. running it on a cloud or an edge application (e.g., on an IoT device). When running Wasm on a browser, the interface to the OS is handled by the browser on behalf of the Wasm module. For servers or edge applications, this must be facilitated by the Wasm runtime hosting the Wasm module. The types of system calls would be like a file system I/O or network I/O.

One approach was to have each hosting Wasm runtime implement how to facilitate the system call on behalf of the Wasm module. This was the approach so far, but this led to portability issues as each runtime exposes different methods for the Wasm module to consume for making the system calls. The WASI spec evolved in the Wasm community to provide standardization. It's a modular set of system interfaces that looks

like an abstracted OS, with low-level interfaces like I/O and high-level interfaces like cryptography, keeping WebAssembly code portable. This also provided better security as with this fine-grained access control can be achieved. For example, a certain Wasm module can only access certain files and not the whole file system. This considerably reduces the possible attack surface originating from a specific Wasm module, even if it's malicious.

Many runtimes have emerged to support running Wasm-based workloads in the cloud and edge. Node.js is a prominent player with the V8 runtime supporting the execution of the Wasm modules by loading them within JavaScript code. The Bytecode Alliance had three runtimes. Two (Wasmtime and Fastly's Lucet) recently merged, optimizing edge compute using ahead-of-time compilation to reduce latency. It is rewritten on top of Wasmtime. WAMR, the micro runtime, is for embedded devices with limited resources; that remains a separate runtime.

There are other runtimes, such as Wasmer and TeaVM (for Java bytecode to Wasm). As the community grows, and thereby the number of runtimes grows, it becomes important to keep an eye on the performance aspects of these runtimes. There is a set of benchmarks that measure different aspects of Wasm's runtime performance.

Figure 1-1 shows that the wavm runtime is fastest, followed by the node runtime.

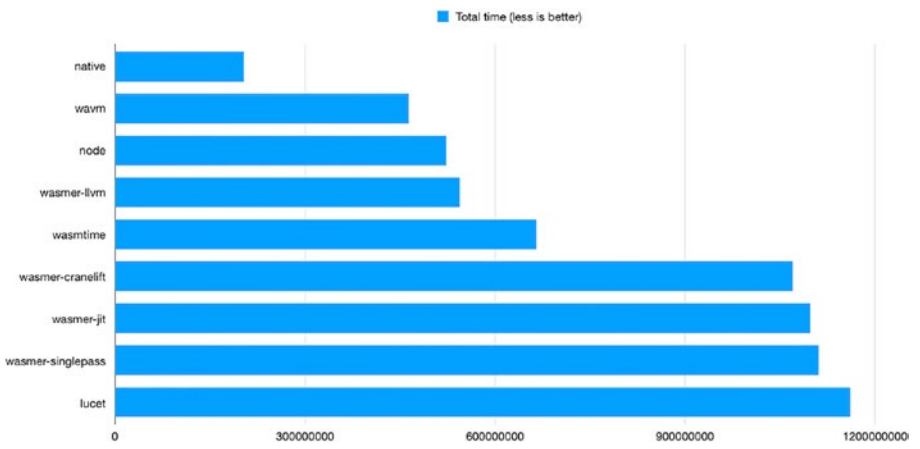

Figure 1-1. *Wasm runtime performance*

Figure 1-2 shows the performance aspects where again wavm is the fastest, followed by the node runtime.

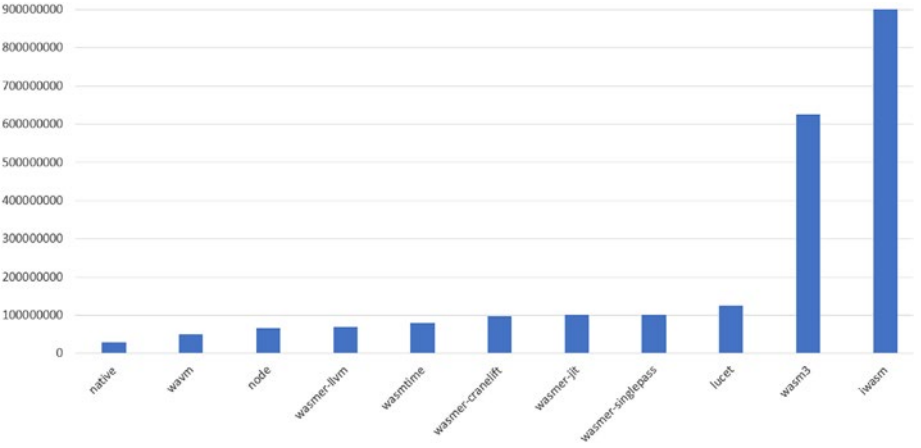

Figure 1-2. *Wasm runtime performance (interpreted mode)*

The following are the main benefits of using WebAssembly.

- Near-native performance
- Lightweight