Introduction to WebAssembly (Wasm)

@Divya_Mohan02

Hi there, I’m Divya Mohan!

- Principal Technology Advocate @ SUSE
- Docs maintainer and contributor to Kubernetes, CNCF
- Leading the re-launched CHAOSS chapter for Asia
- Recognized contributor to Bytecode Alliance

@Divya_Mohan02

WTH is WebAssembly?

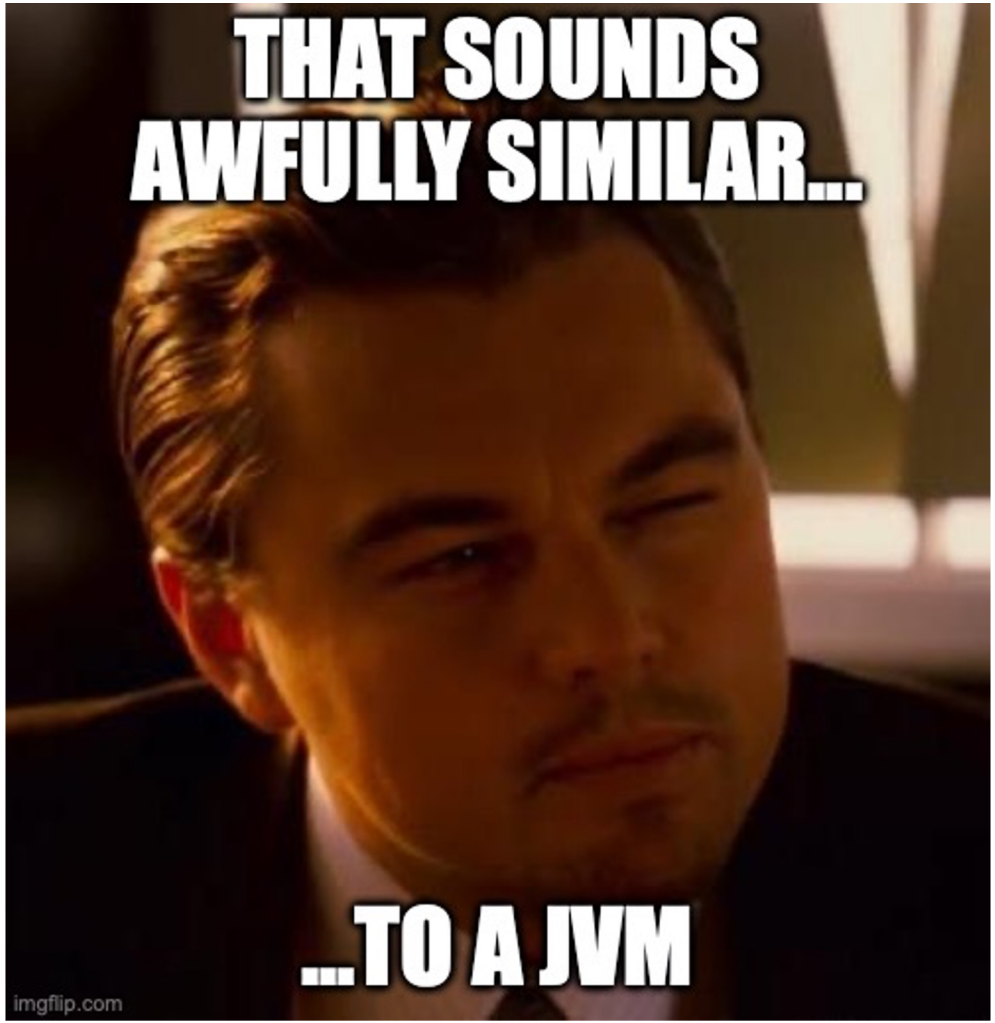
A brief history of WebAssembly

- Announced in 2015
- v1 MVP announced in 2017
- WASI standard announced in 2019
- v2 draft announced in 2022
- WASI Preview 2 announced in 2024

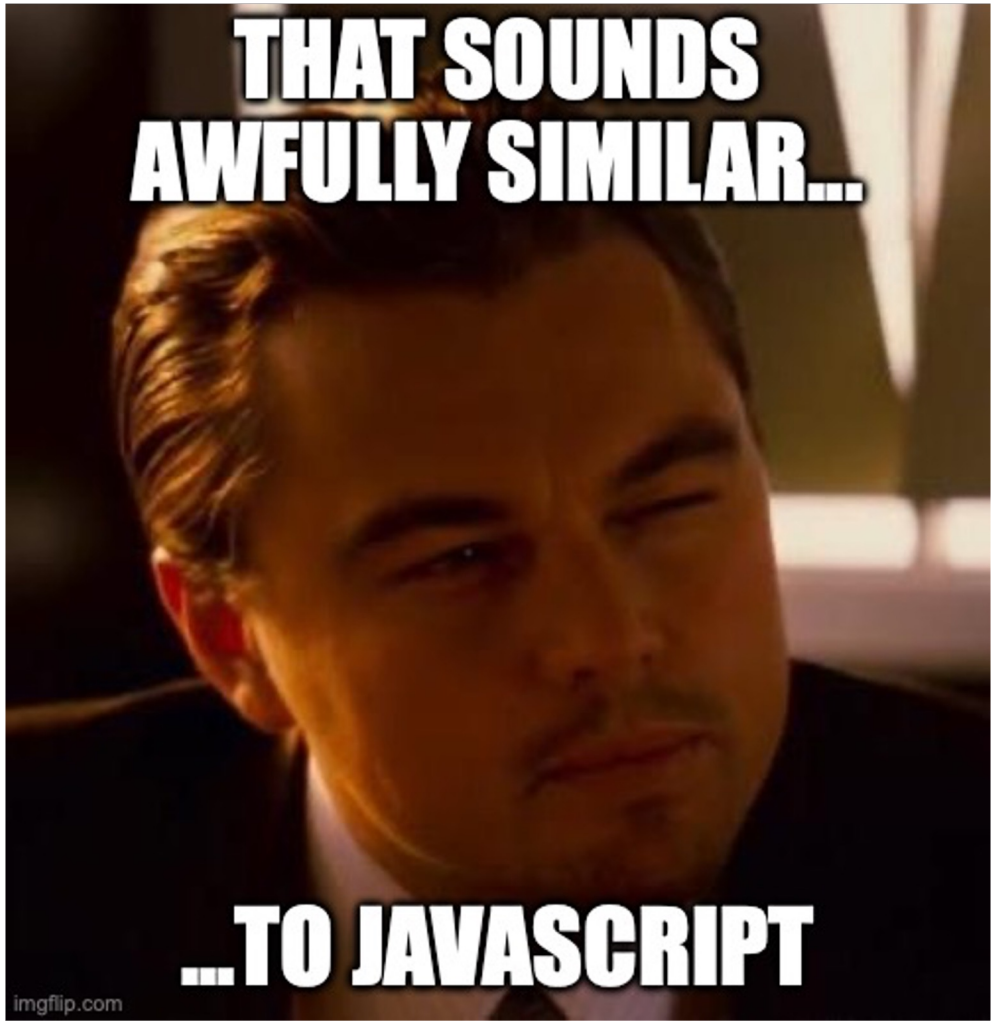
WebAssembly is…

- Bytecode-like format
  - Portable across architectures & languages
  - Sandboxed = Secure
- Not just for the web
  - Can be (and is!) extended to server-side ecosystem
- Also, it is Wasm not WASM!

```
(module
  (type $t0 (func (param i32 i32 i32) (result i32)))
  (type $t1 (func (param i32)))
  (type $t2 (func (param i32 i32 i32 i32) (result i32)))
  (type $t3 (func (param i32 i32) (result i32)))
  (type $t4 (func (param i32 i32 i32 i32 i32 i32) (result i32)))
  (type $t5 (func))
  (type $t6 (func (result i32)))
  (type $t7 (func (param i32) (result i32)))
  (type $t8 (func (param i32 i64 i32) (result i64)))
  (import "env" "putc_js" (func $putc_js (type $t1)))
  (import "env" "__syscall3" (func $__syscall3 (type $t2)))
  (import "env" "__syscall1" (func $__syscall1 (type $t3)))
  (import "env" "__syscall5" (func $__syscall5 (type $t4)))
  (func $__wasm_call_ctors (type $t5))
  (func $main (export "main") (type $t6) (result i32)
    i32.const 1024
    call $puts
    drop
    i32.const 0)
  (func $writev_c (export "writev_c") (type $t0) (param $p0 i32) (param $p1 i32) (param $p2 i32) (result i32)
```



@Divya_Mohan02



@Divya_Mohan02

JVMs are **NOT** polyglot compilation targets!

- Java bytecode was not **designed** to be a compilation target.
- *invokeDynamic* opcode was introduced **especially** to support the dynamic languages targeting the runtime[1]
- Not an adequate compilation target, but a necessary one.

> *The Java Virtual Machine (JVM) has been widely adopted in part because of its classfile format, which is portable, compact, modular, verifiable, and reasonably easy to work with.* ***However, it was designed for just one language—Java****— and so when it is used to express programs in other source languages, there are often "pain points" which retard both development and execution.*

Source: https://www.javaadvent.com/2022/12/webassembly-for-the-java-geek.html

Why are we talking about WebAssembly (**AGAIN**)?

Focus on memory safe programming languages

FEBRUARY 26, 2024

PRESS RELEASE: Future Software Should Be Memory Safe

ONCD ▸ BRIEFING ROOM ▸ PRESS RELEASE

Leaders in Industry Support White House Call to Address Root Cause of Many of the Worst Cyber Attacks

Read the full report here

@Divya_Mohan02

A brief history of WebAssembly

- Announced in 2015
- v1 MVP announced in 2017
- WASI standard announced in 2019
- v2 draft announced in 2022
- WASI Preview 2 announced in 2024

But, there's also…

Component Model?

Kubernetes x WebAssembly?

Serverless x WebAssembly?

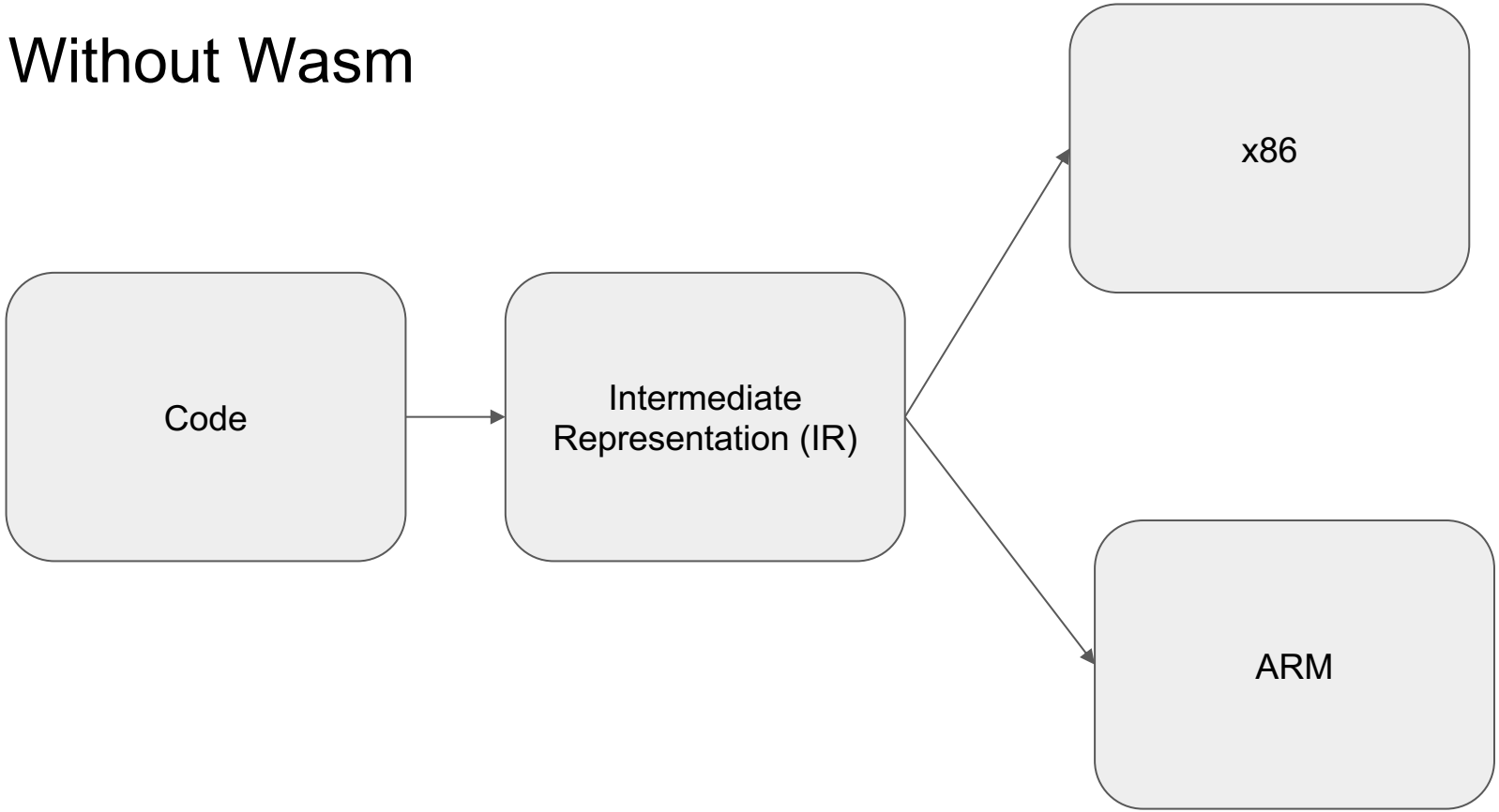
WebAssembly x AI?

Edge Compute x WebAssembly?

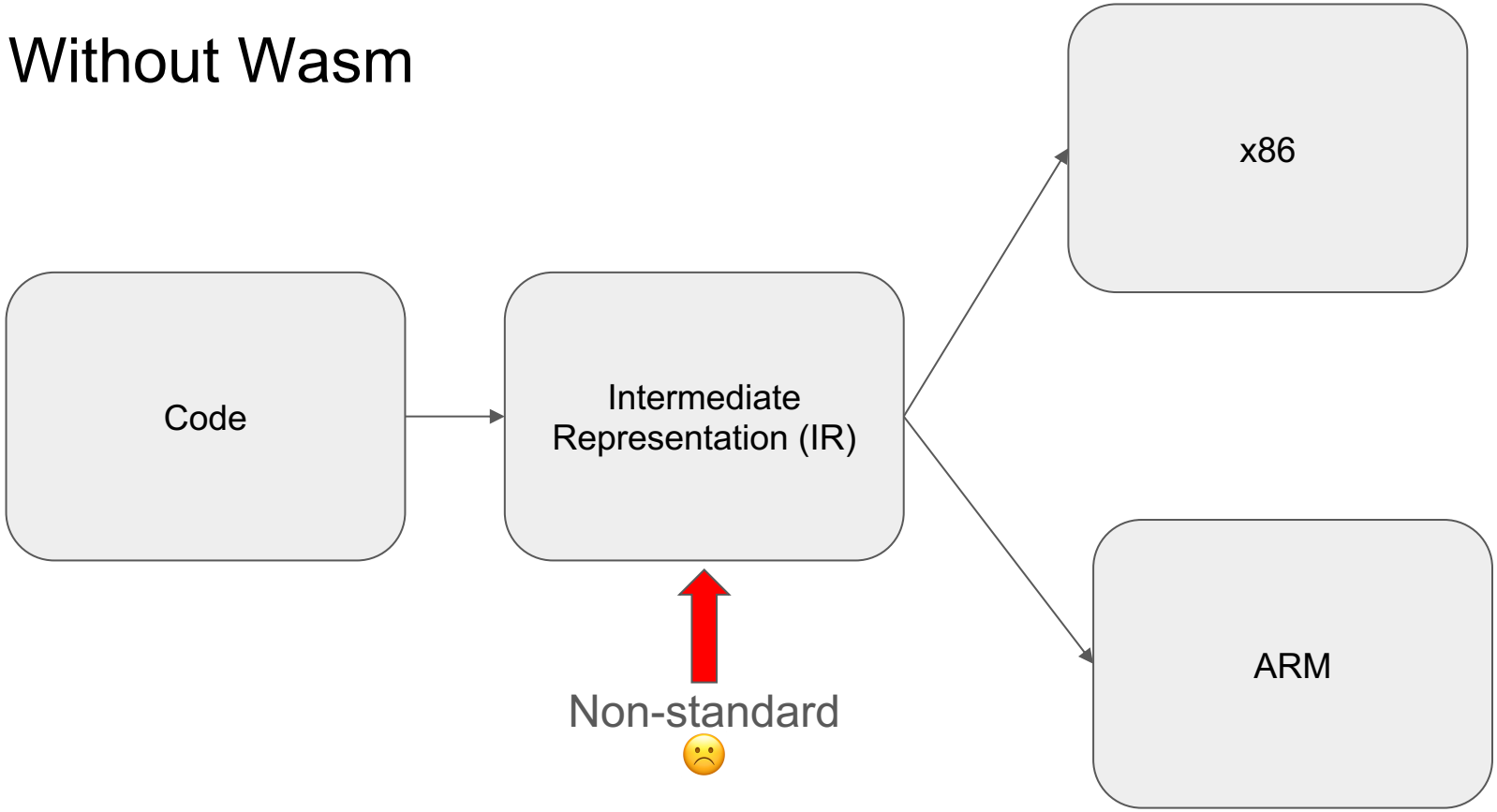
WebAssembly in the Browser?

Bytecode-ish format, huh? How does that work?

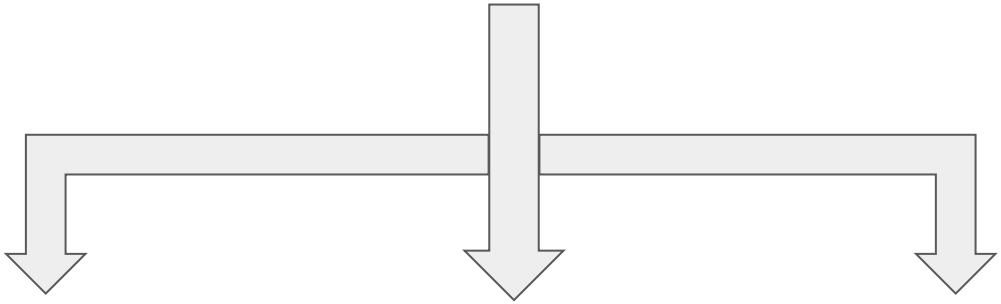
Without Wasm



Without Wasm



Types of Intermediate Representation



Low-level IR

Instructions for a virtual machine

Mid IR

Combines elements of the other two

High-level IR

Closer to a programming language

Portable assembly isn’t new!

Pascal's P-Code

Native Client (NaCl) & Portable Native Client (PNaCl) by Google

IBM TIMI

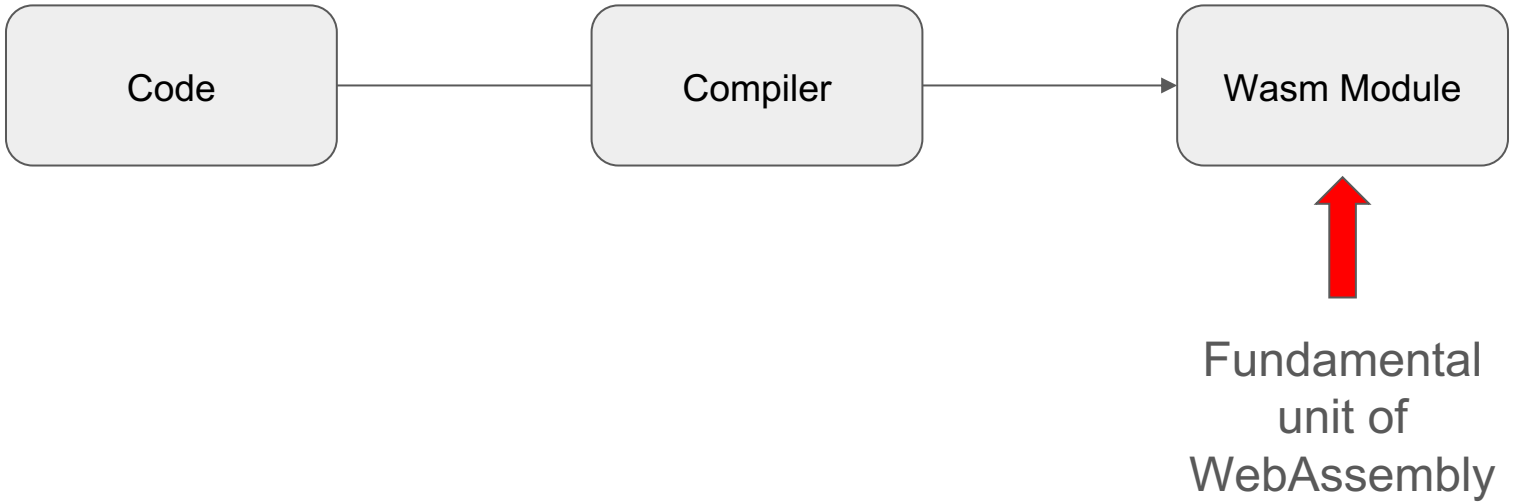
asm.js

Java bytecode & the JVM

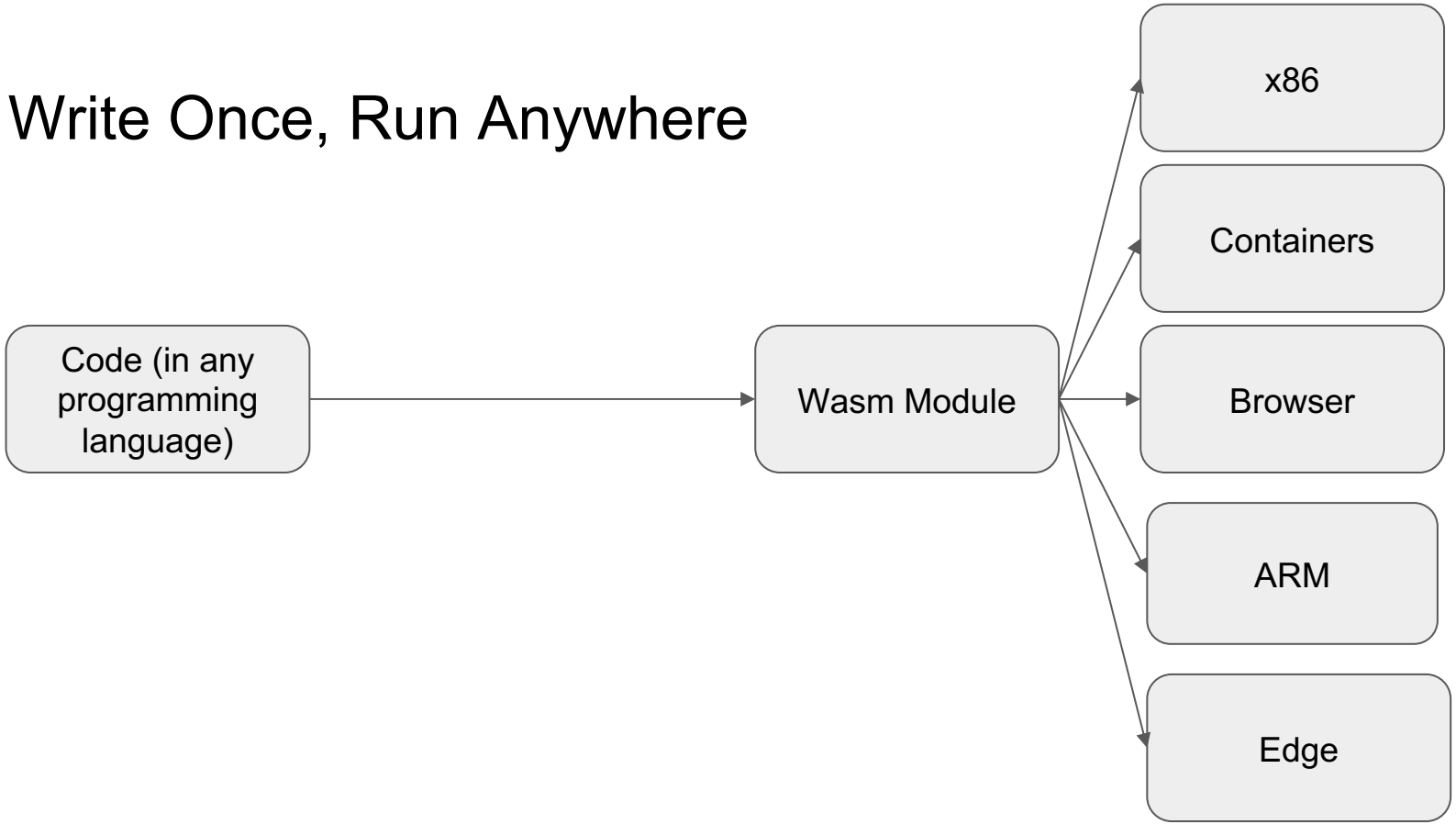
JavaScript

So, where does Wasm fit in?

Write Once, Run Anywhere!

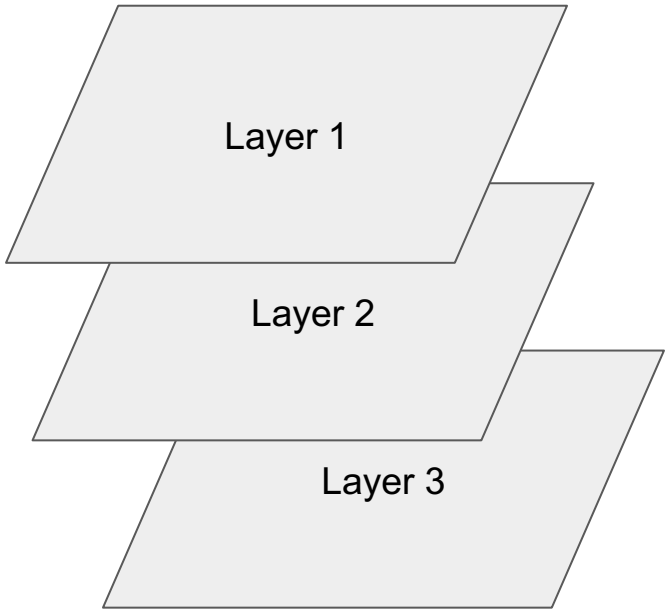


Write Once, Run Anywhere

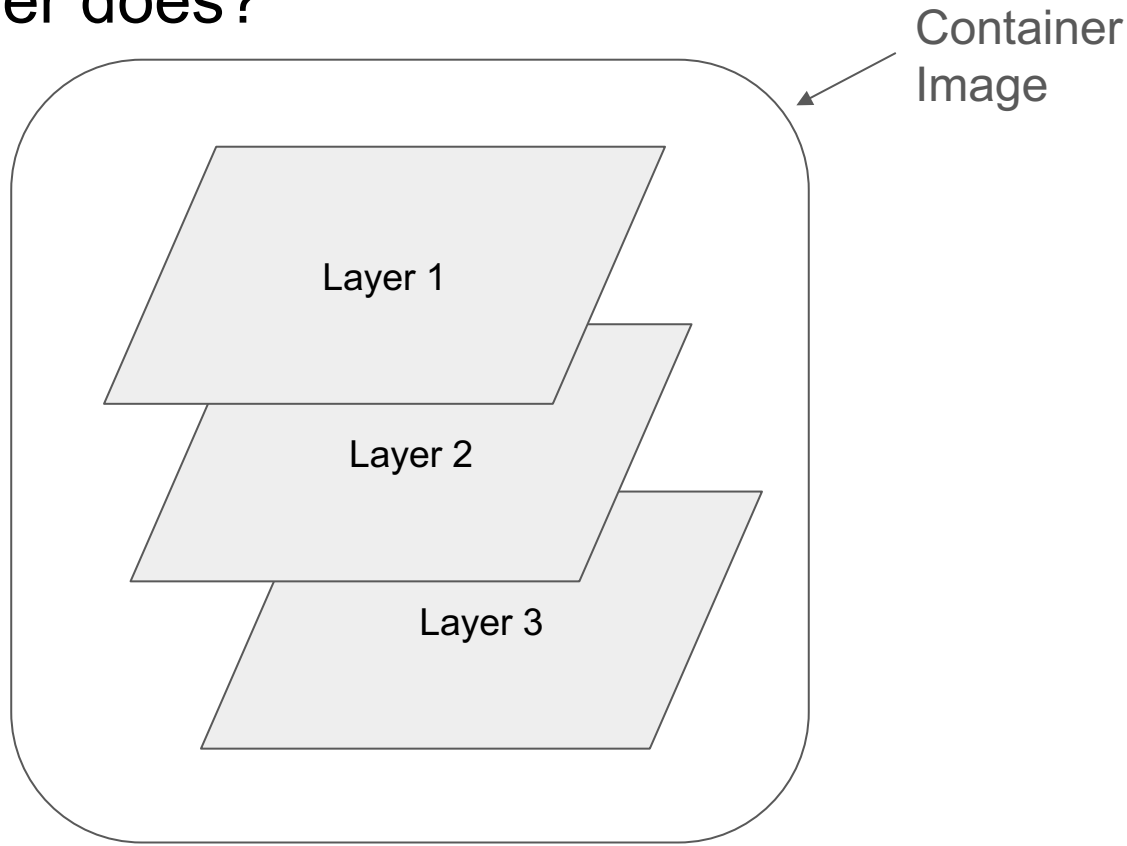


Hasn’t this been achieved with containerization?

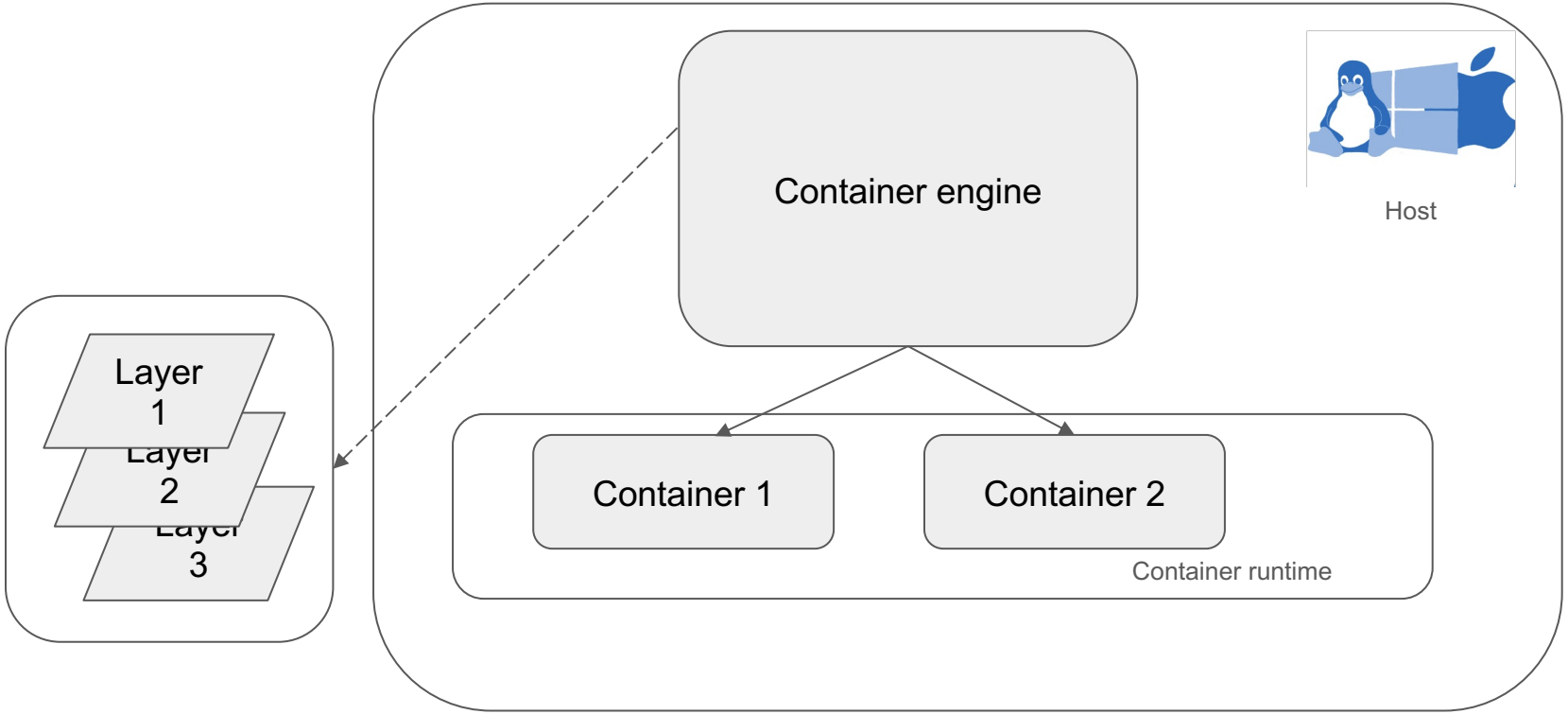
What a container does?



What a container does?

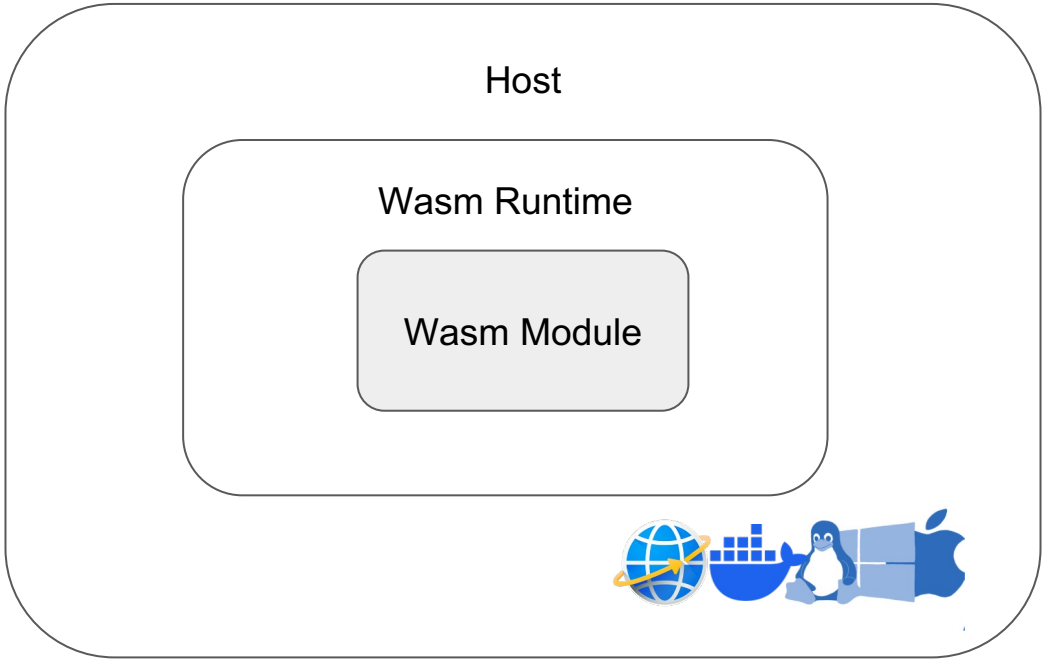


What a container does?



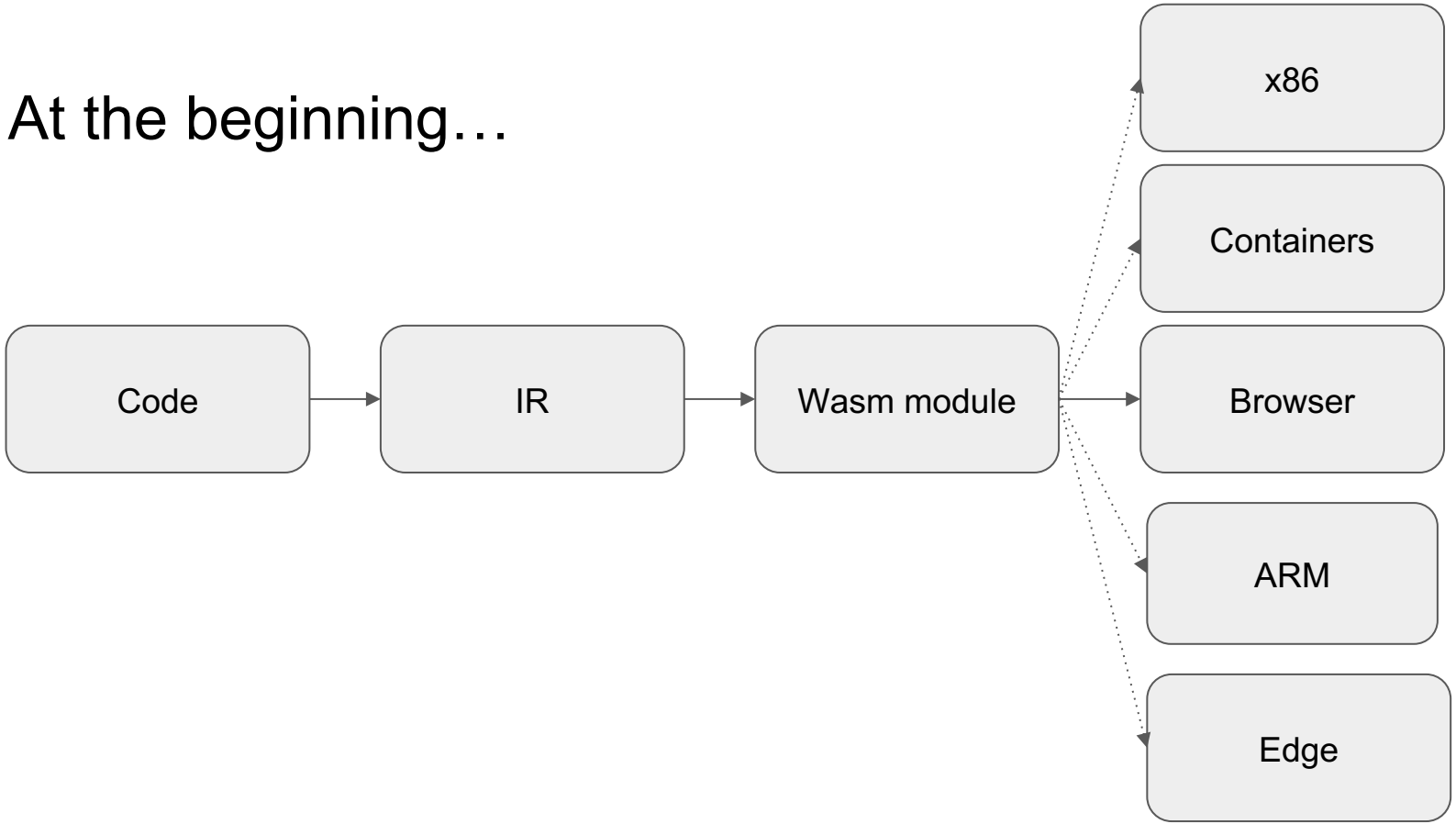
What Wasm does?

What Wasm does?



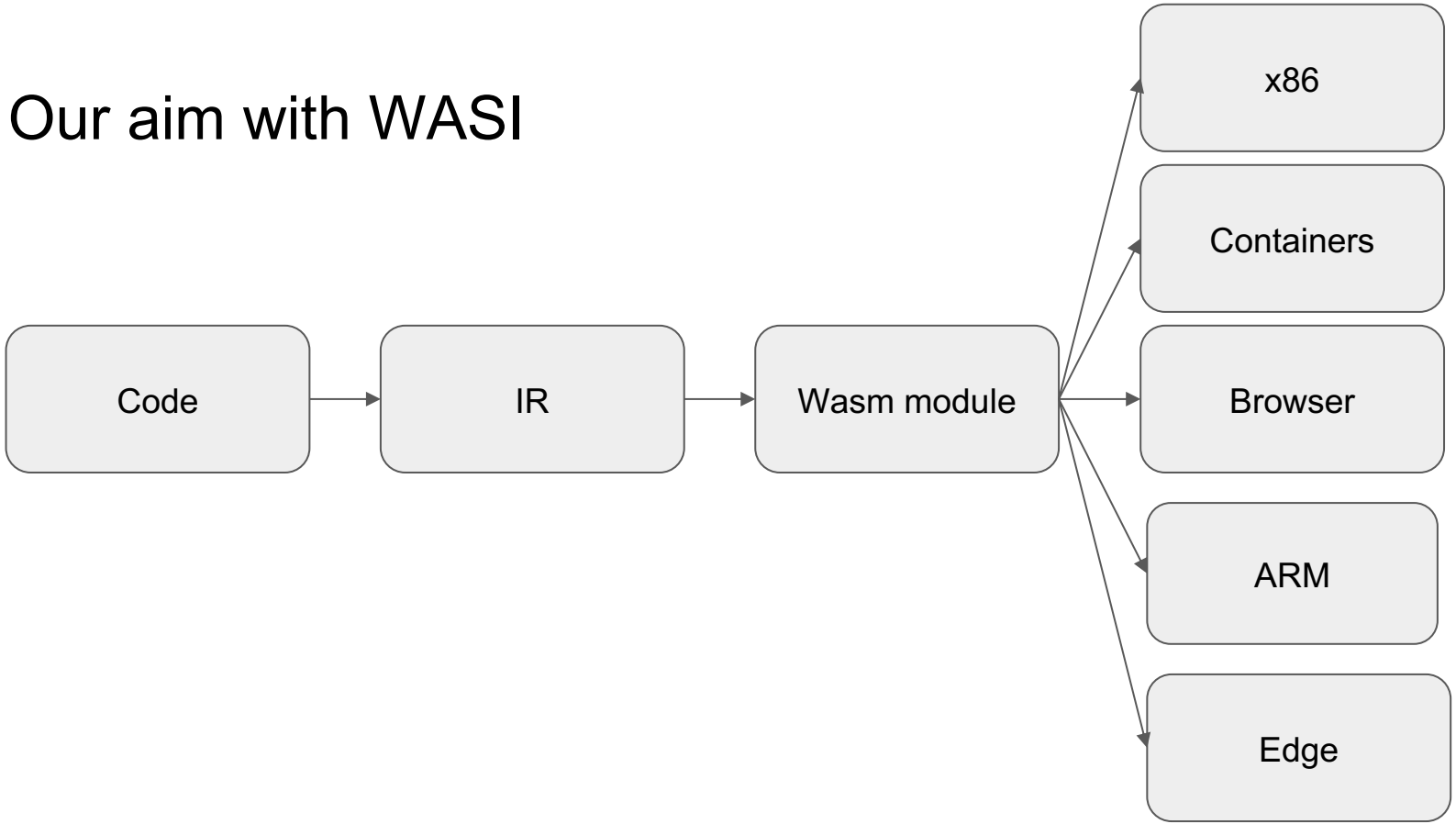
WASI

At the beginning…

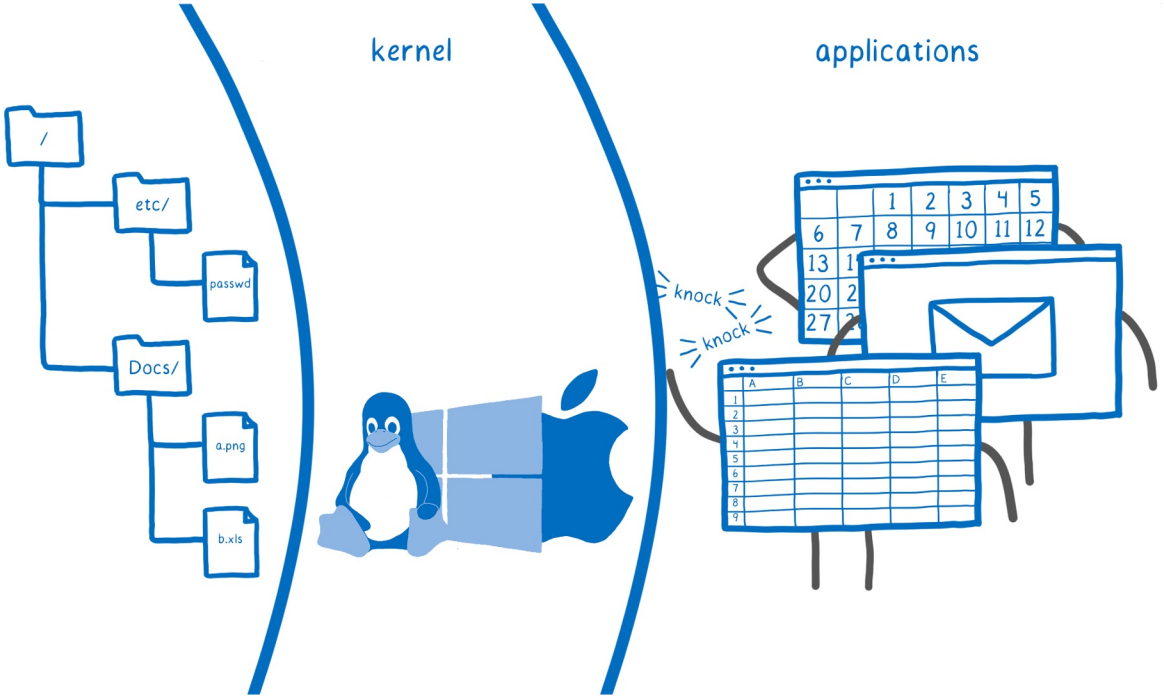


The WebAssembly System Interface (WASI) is a set of APIs for WASI being developed for eventual standardization by the WASI Subgroup, which is a subgroup of the WebAssembly Community Group.

Our aim with WASI

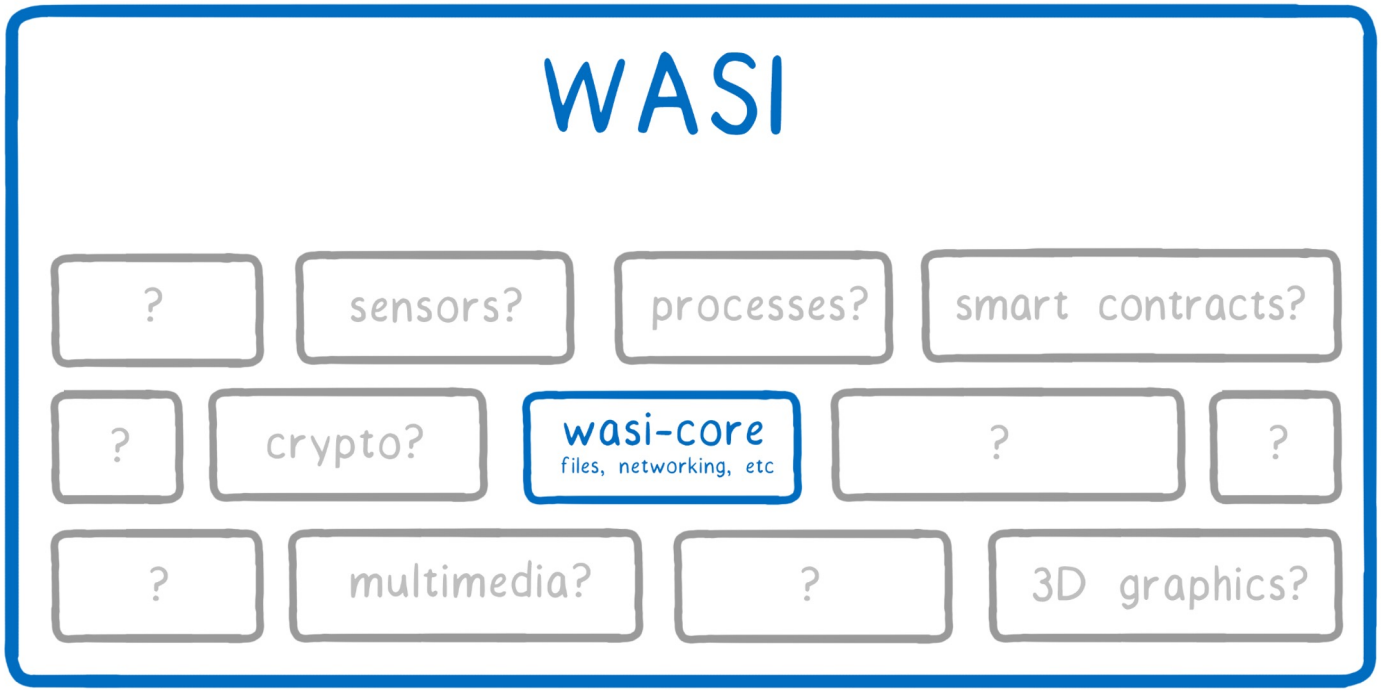


What's a system interface?



Source : https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

How WASI proposes to implement it?



Source: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

WASI Preview 1

- APIs heavily influenced by
  - Portable Operating System Interface (POSIX)
    - Set of APIs, command line shells, and user interfaces
    - Defines a set of standard system interfaces based on UNIX
  - Cloud Application Binary Interface (ABI)
    - ABI for UNIX-like OS based on capability-based security
- What were we trying to do?
  - Porting Unix ideas to Wasm to give developers the capability to interact with the world

WASI Preview 1

- Challenges:
  - Monolithic ABI implementation
  - Didn’t make sense for some platforms = Low on portability
    - Windows -> Requirement of a compatibility layer
    - Web embeddings

What did WASI Preview 1 support?

- System Clock
- Random Number Generator
- Environment Variables
- Filesystem
- Standard I/O and error streams

WASI Preview 2

- Set of APIs influenced by
  - The Canonical ABI
  - Core Wasm spec
- Adheres to the component model
- ABI is more modular
- What are we trying to do?
  - Provide a stable definition of common interfaces for library developers

Introduced in WASI 0.2

- Worlds
  - Complete description of the imports & exports of a component
  - May be used to represent the execution environment of a component
    - CLI World: CLI-like environment
    - HTTP Proxy World: Ability to concurrently stream in/out any number of HTTP requests

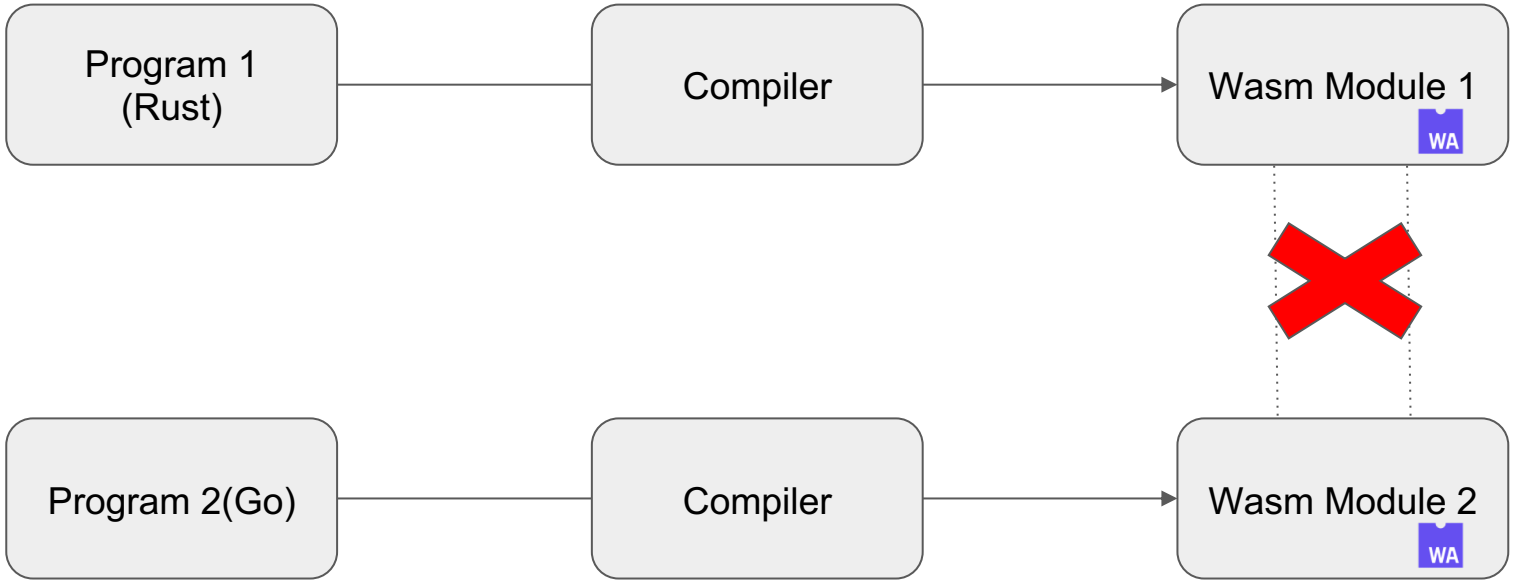
APIs launched as part of WASI 0.2

- wasi-io
- wasi-clocks
- wasi-random
- wasi-filesystem
- wasi-sockets
- wasi-cli
- wasi-http

Did you say the component model?

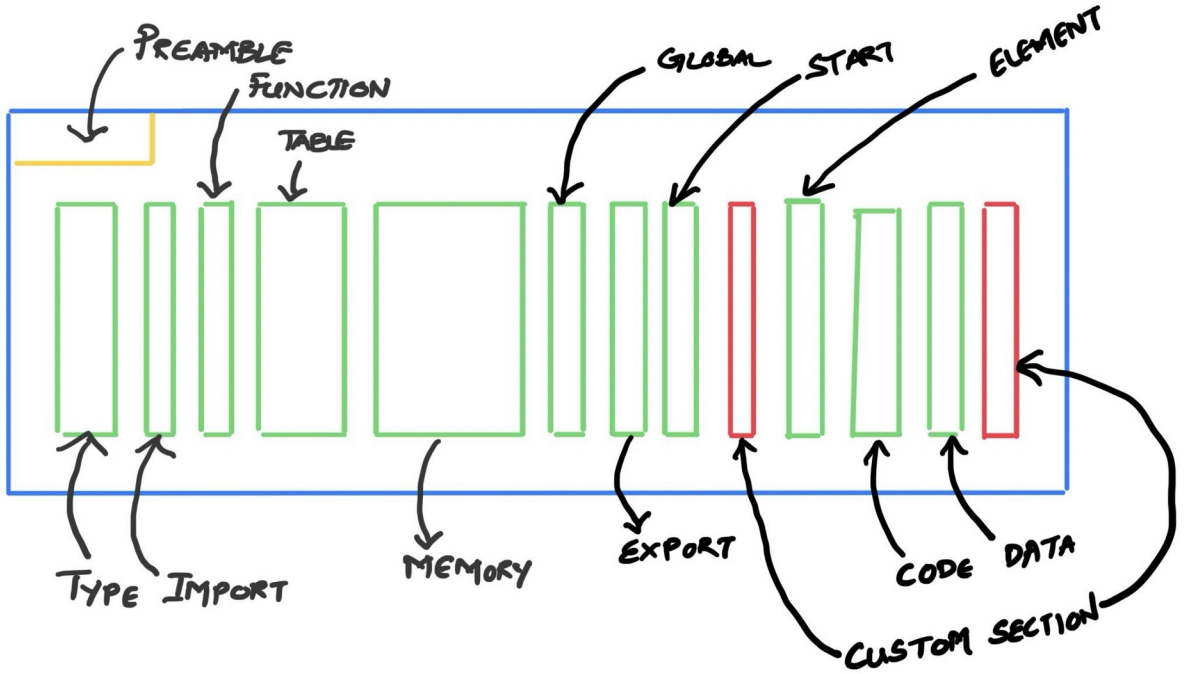
Let’s go back here…

With WASI Preview 1



Why tho? 🤔

Cause a module kinda looks like this…



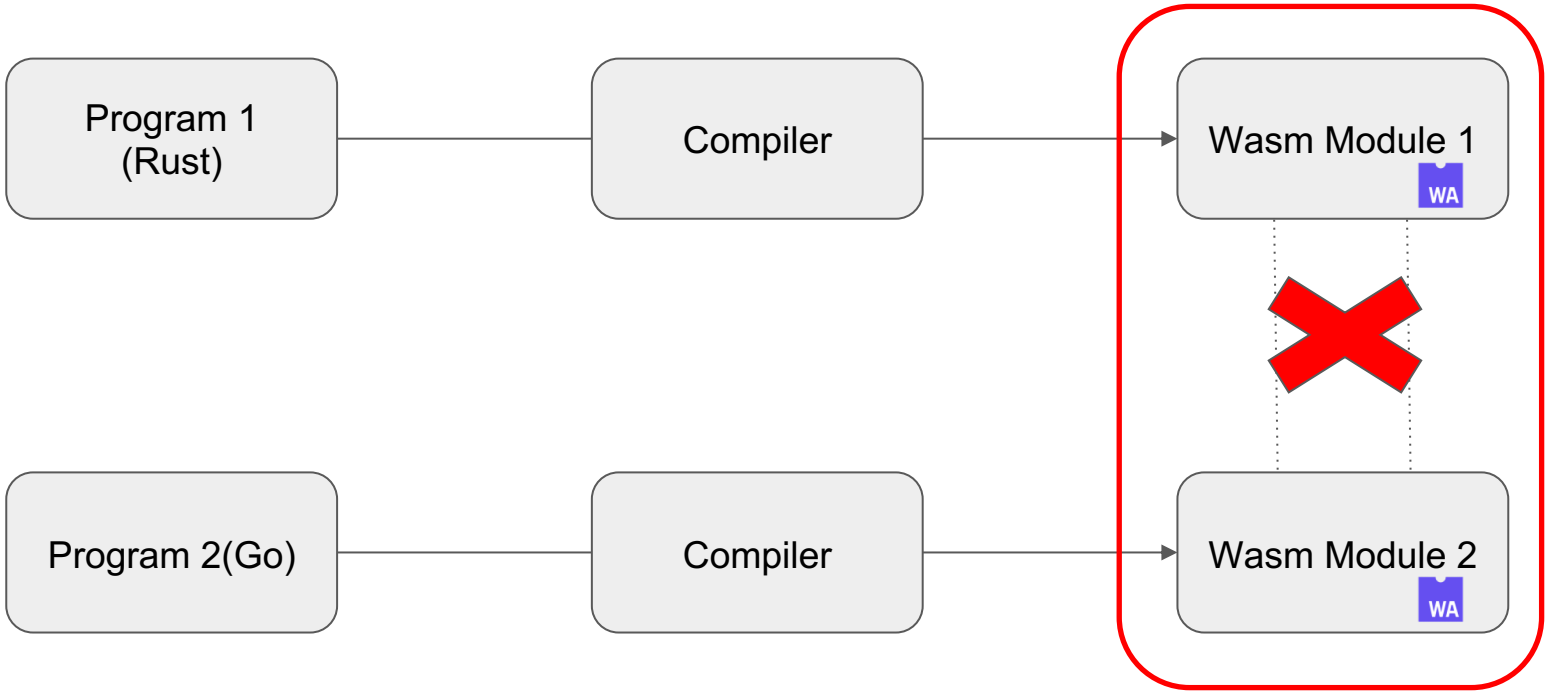
And supports…

```
(module
  (type $t0 (func (param i32 i32 i32) (result i32)))
  (type $t1 (func (param i32)))
  (type $t2 (func (param i32 i32 i32 i32) (result i32)))
  (type $t3 (func (param i32 i32) (result i32)))
  (type $t4 (func (param i32 i32 i32 i32 i32 i32) (result i32)))
  (type $t5 (func))
  (type $t6 (func (result i32)))
  (type $t7 (func (param i32) (result i32)))
  (type $t8 (func (param i32 i64 i32) (result i64)))
  (import "env" "putc_js" (func $putc_js (type $t1)))
  (import "env" "__syscall3" (func $__syscall3 (type $t2)))
  (import "env" "__syscall1" (func $__syscall1 (type $t3)))
  (import "env" "__syscall5" (func $__syscall5 (type $t4)))
  (func $__wasm_call_ctors (type $t5))
  (func $main (export "main") (type $t6) (result i32)
    i32.const 1024
    call $puts
    drop
    i32.const 0)
  (func $writev_c (export "writev_c") (type $t0) (param $p0 i32) (param $p1 i32) (param $p2 i32) (result i32)
```

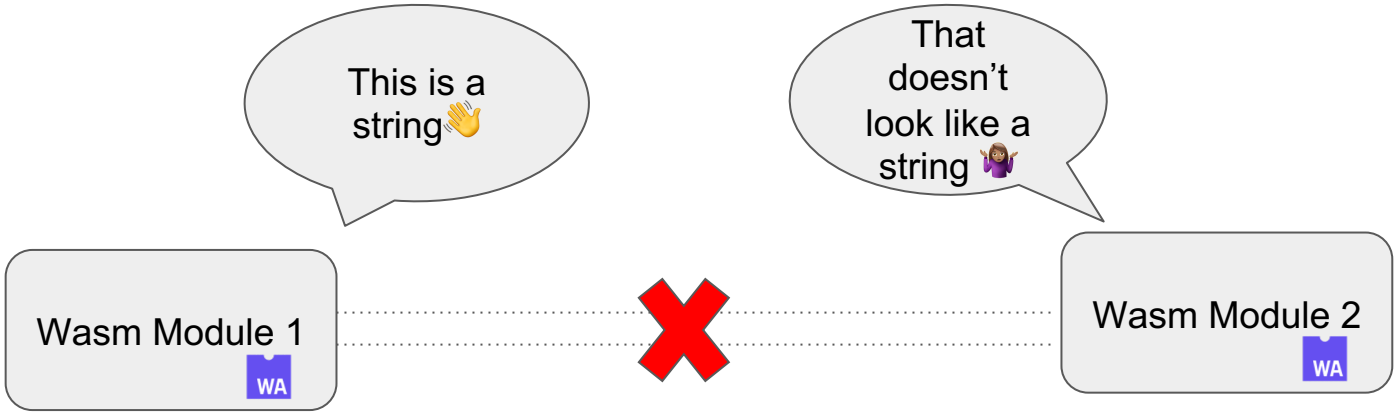
Therefore,

- Richer types such as strings/structs have to be represented by floats & integers
- These representations aren’t interchangeable

With WASI Preview 1



Zooming in…

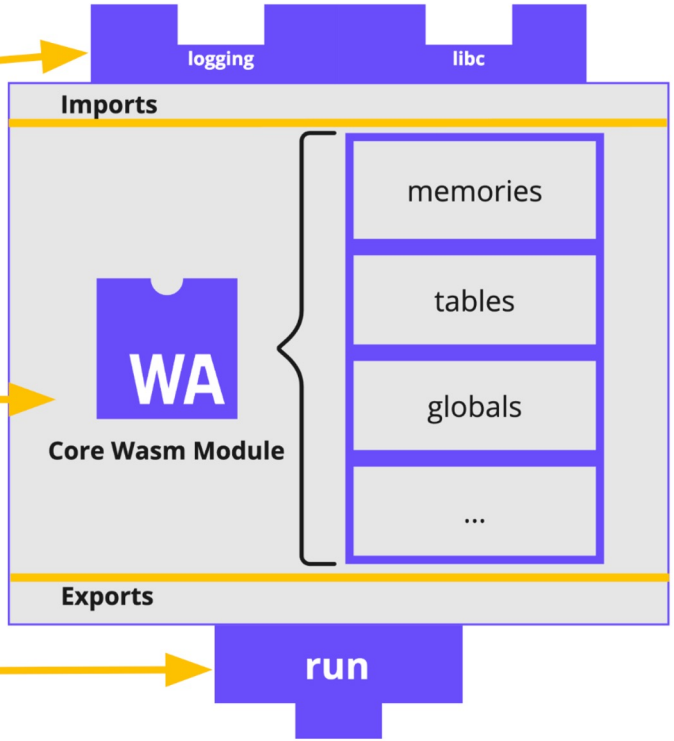

This is a string 👋
That doesn't look like a string 🤷🏽‍♀️
Wasm Module 1
WA
Wasm Module 2
WA

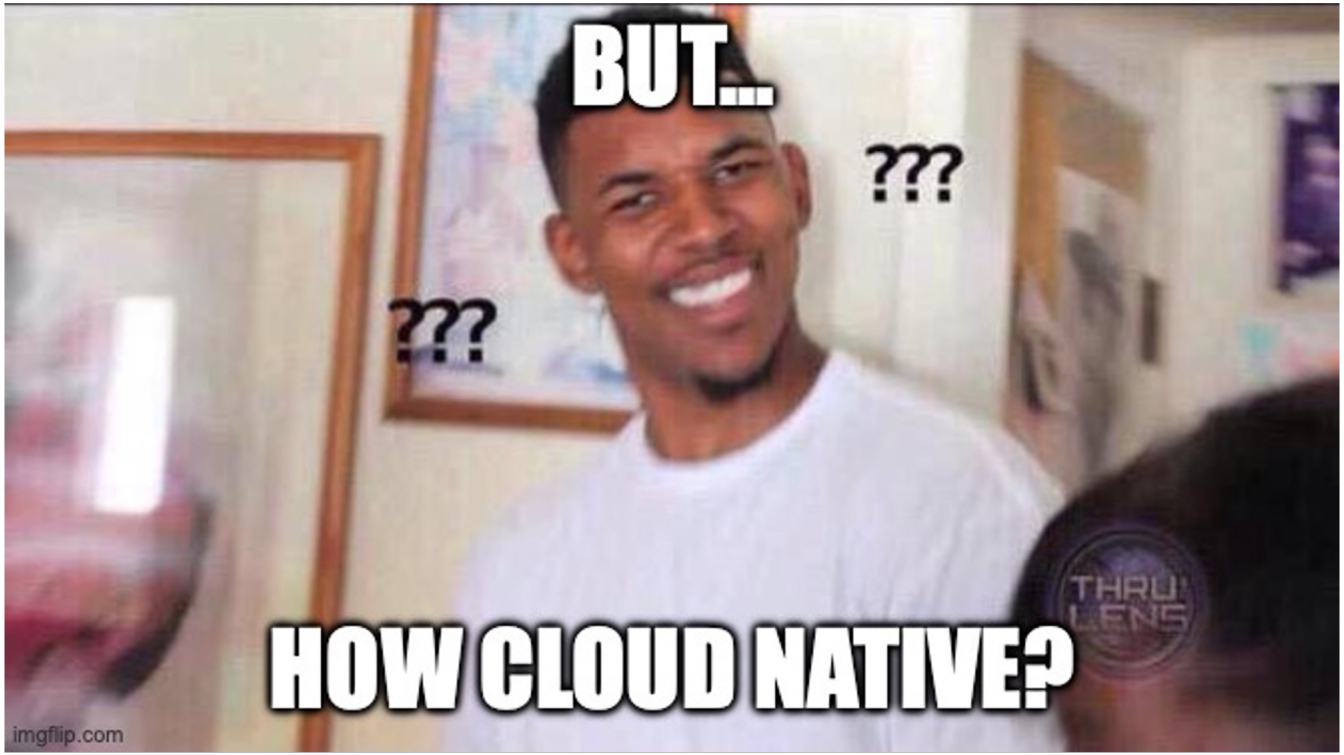
Enter the component model

- A component provides a wrapper around the core Wasm module (~ shared dictionary)
- Richer types are defined using WIT
- Translated to bits and bytes using Canonical ABI

So, what does a component look like?

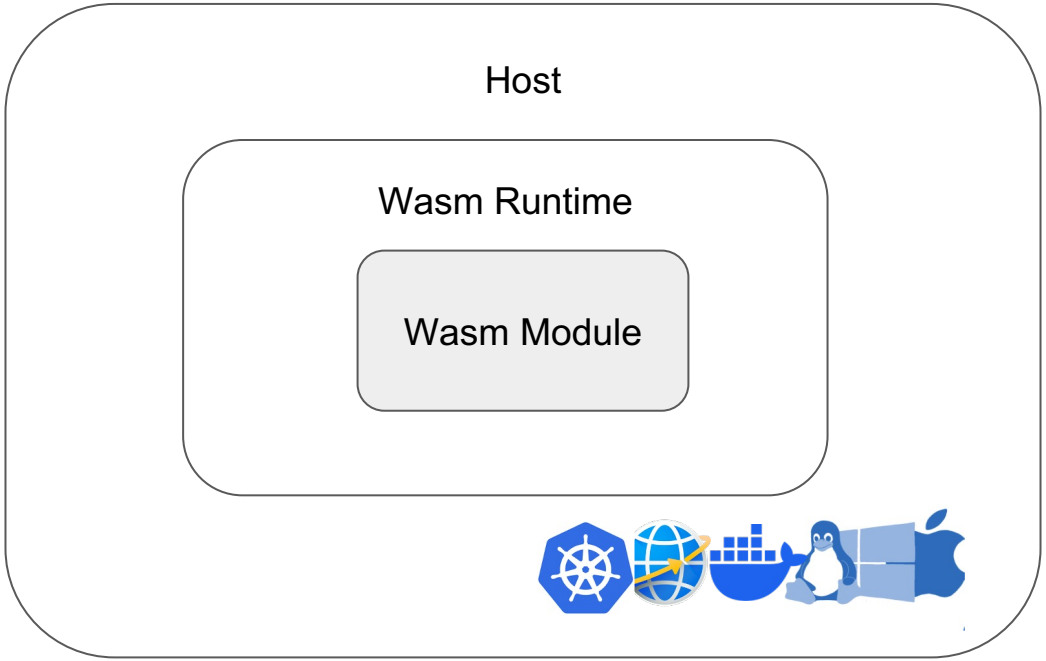
```
(component
  (import "logging" (instance $logging
    (export "log" (func (param string)))
  ))
  (import "libc" (core module $Libc
    (export "mem" (memory 1))
    (export "realloc" (func (param i32 i32) (result i32)))
  ))
  (core instance $libc (instantiate $Libc))
  (core func $log (canon lower
    (func $logging "log")
    (memory (core memory $libc "mem")) (realloc (func $libc "realloc"))
  ))
  (core module $Main
    (import "libc" "memory" (memory 1))
    (import "libc" "realloc" (func (param i32 i32) (result i32)))
    (import "logging" "log" (func $log (param i32 i32)))
    (func (export "run") (param i32 i32) (result i32)
      ... (call $log) ...
    )
  )
  (core instance $main (instantiate $Main
    (with "libc" (instance $libc))
    (with "logging" (instance (export "log" (func $log))))
  ))
  (func $run (param string) (result string) (canon lift
    (core func $main "run")
    (memory (core memory $libc "mem")) (realloc (func $libc "realloc"))
  ))
  (export "run" (func $run))
)
```





@Divya_Mohan02

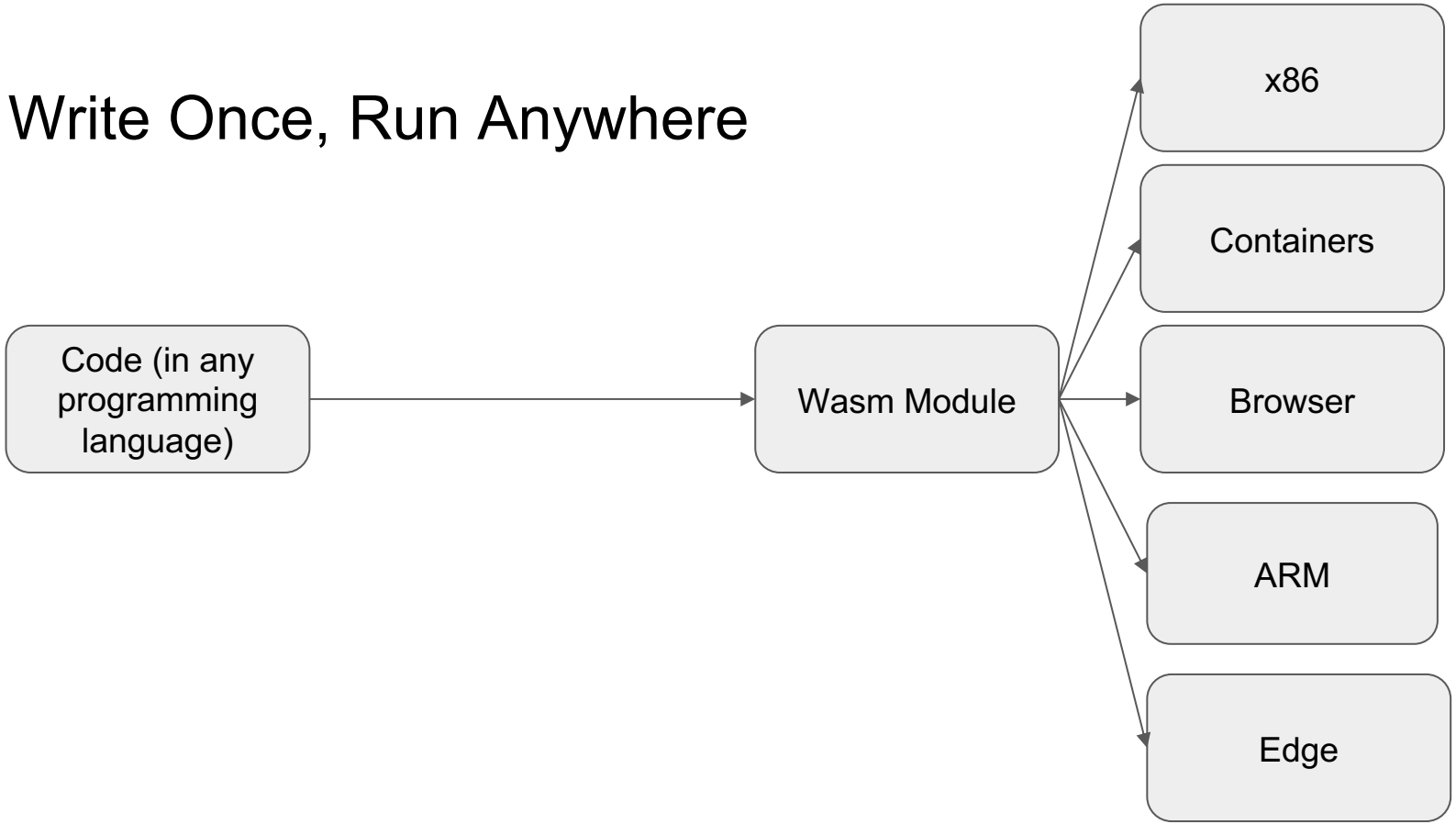
Bringing this back…



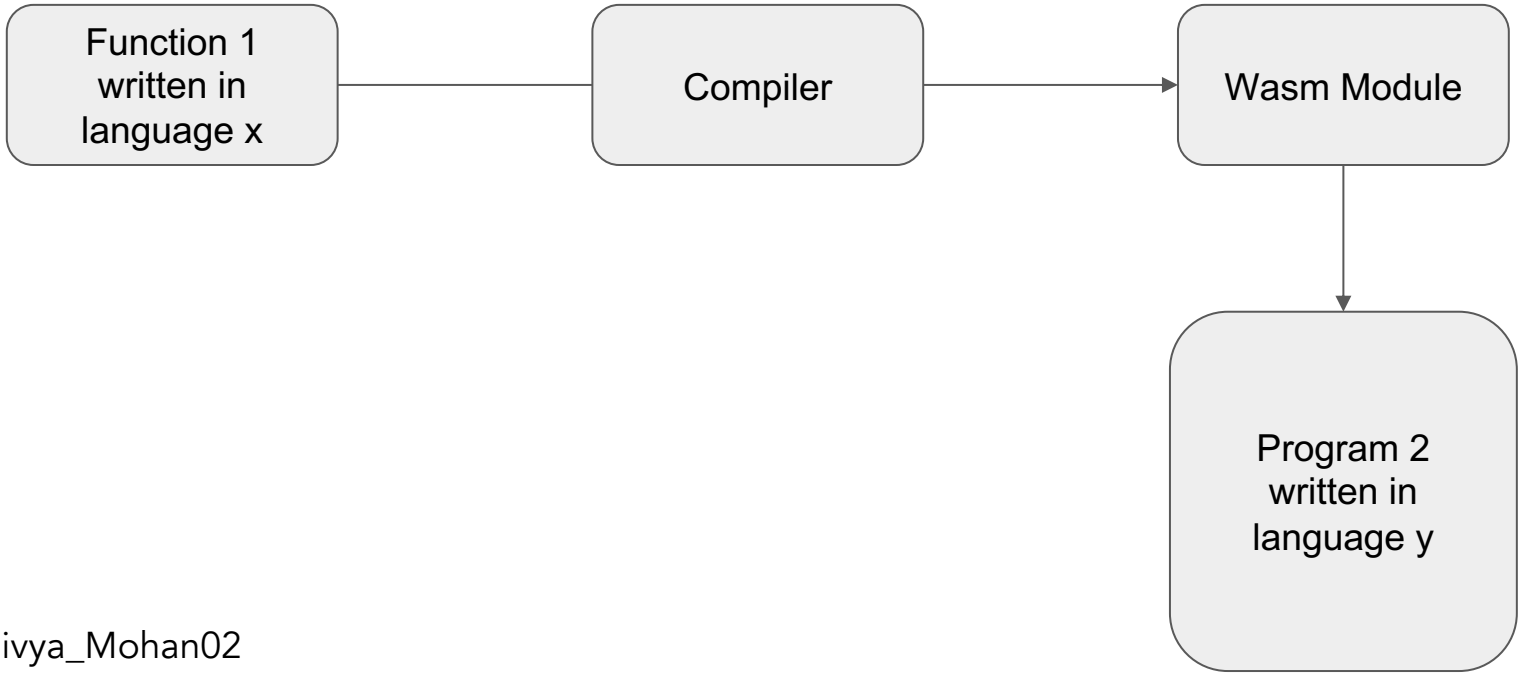
Live demo!

Cool info, why should I care?

Write Once, Run Anywhere



Code reusability



But…that’s a lot of work!

Yep, and here's how YOU can help!

- Pick an area
  - Libraries
  - Runtimes & toolchains
  - Ecosystem integrations
    - Browser
    - Cloud Native
    - Serverless
  - Standards
  - Documentation
  - Community & outreach

A few good places to start

- CNCF Slack - #wg-wasm
- Bytecode Alliance Zulip
- WebAssembly Discord Server
- Bytecode Alliance GitHub

Resources

- Bytecodes meet combinators: invokedynamic on the JVM
- WebAssembly spec
- WASI Preview 2 spec
- WASI Preview 2 Discussion – Bailey Hayes (WasmEdge community meeting Feb)
- Rancher Live: WASI 0.2 - Deep dive – YouTube
- ACM paper on IR execution

Resources

- The White House Press Release from February 26, 2024
- Bytecode Alliance Documentation
- Rancher Desktop
- Blogs by Dan Gohman
- An empirical study of Real-World WebAssembly Binaries
- Wikipedia’s entry on capability-based security
- Containerd Wasm shims

Where you can find me

- X (formerly Twitter): https://x.com/Divya_Mohan02
- GitHub: https://github.com/divya-mohan0209
- LinkedIn: https://linkedin.com/in/divya-mohan0209

Thank you!

@Divya_Mohan02