

Introduction to WebAssembly (Wasm)

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



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)
```

**THAT SOUNDS
AWFULLY SIMILAR...**

...TO A JVM

**THAT SOUNDS
AWFULLY SIMILAR...**

...TO JAVASCRIPT

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**

@Divya_Mohan02

Read the full report [here](#)

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?

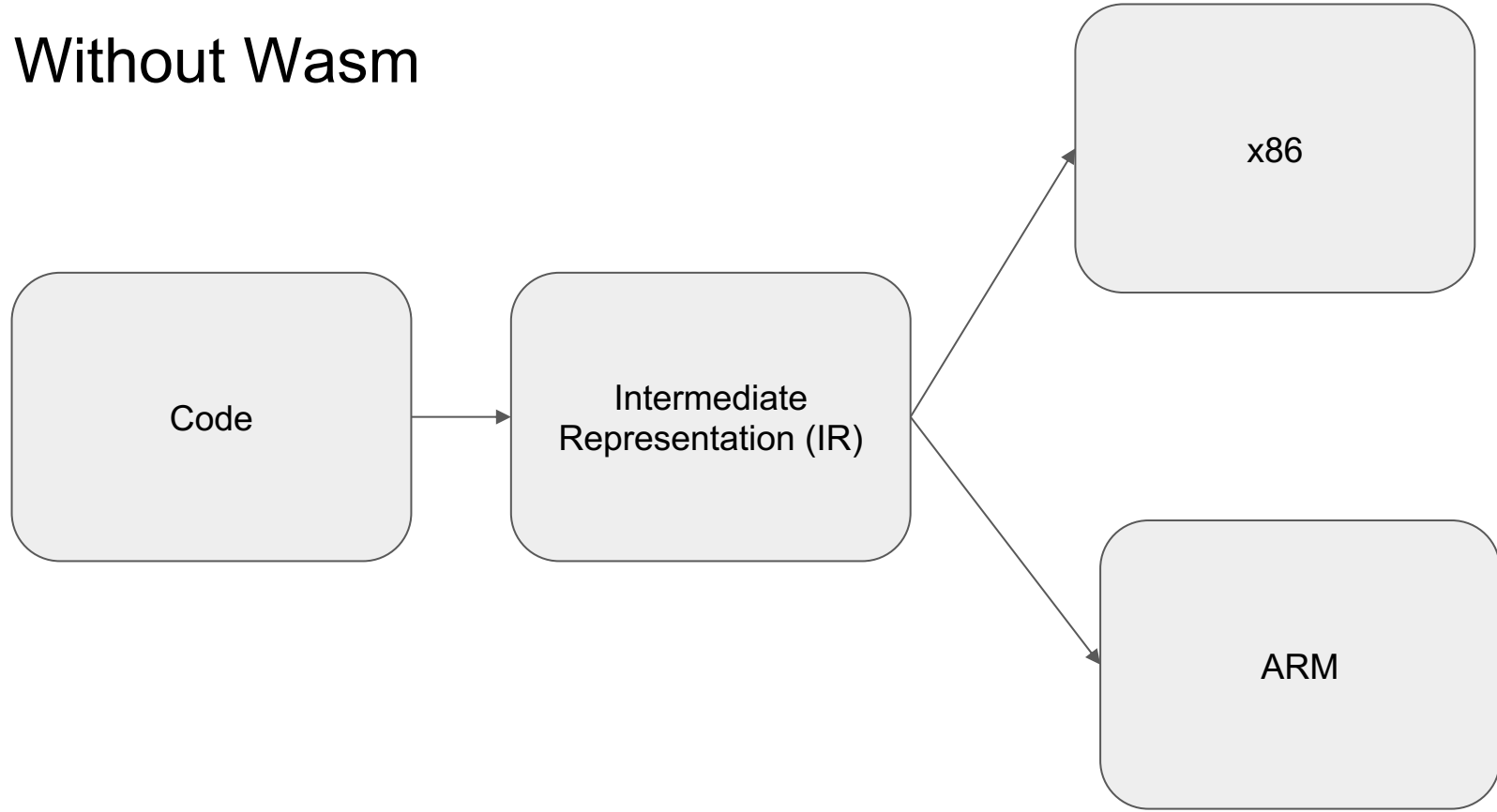
WebAssembly x AI?

Edge Compute x
WebAssembly?

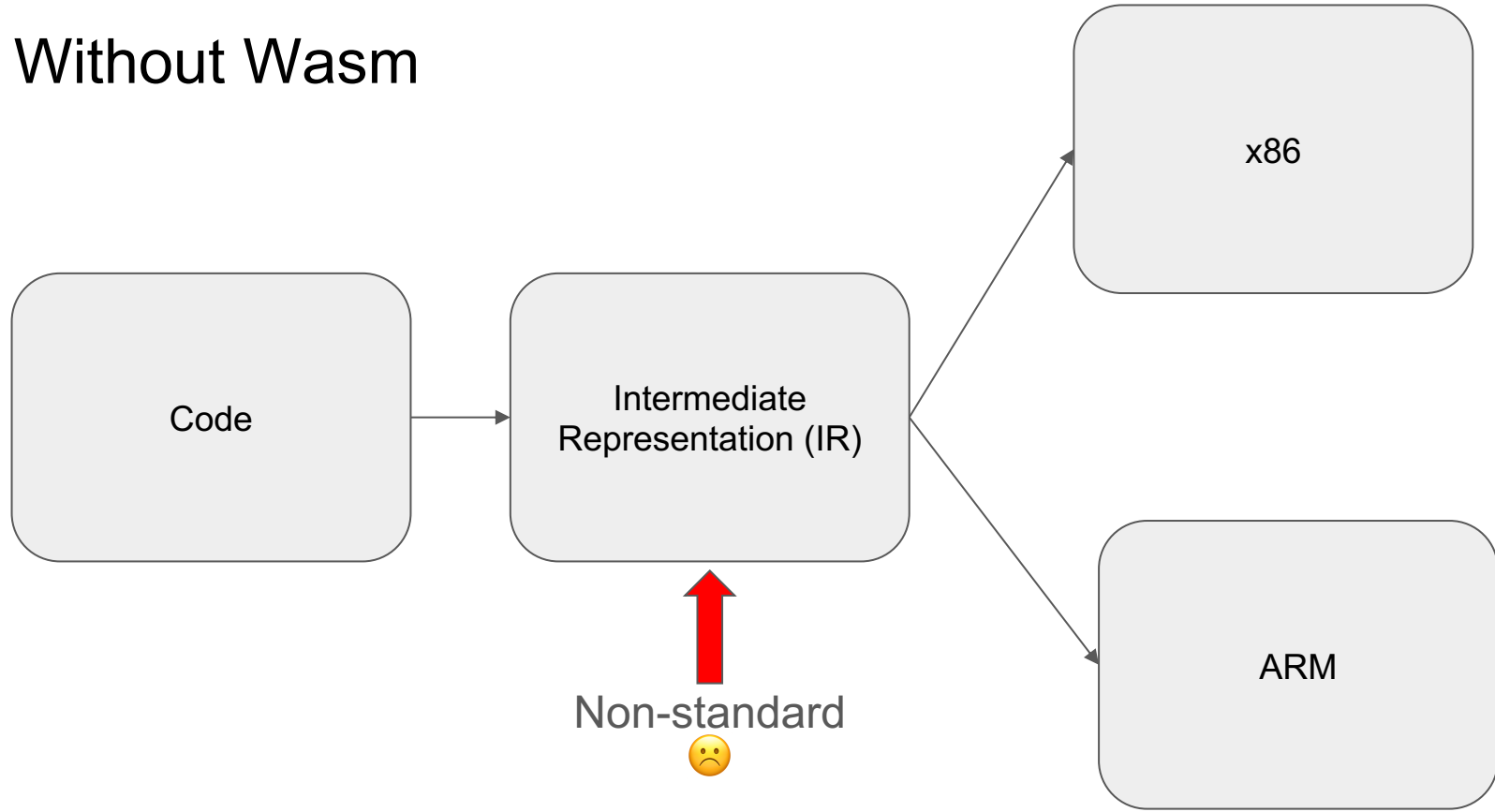
WebAssembly in the
Browser?

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

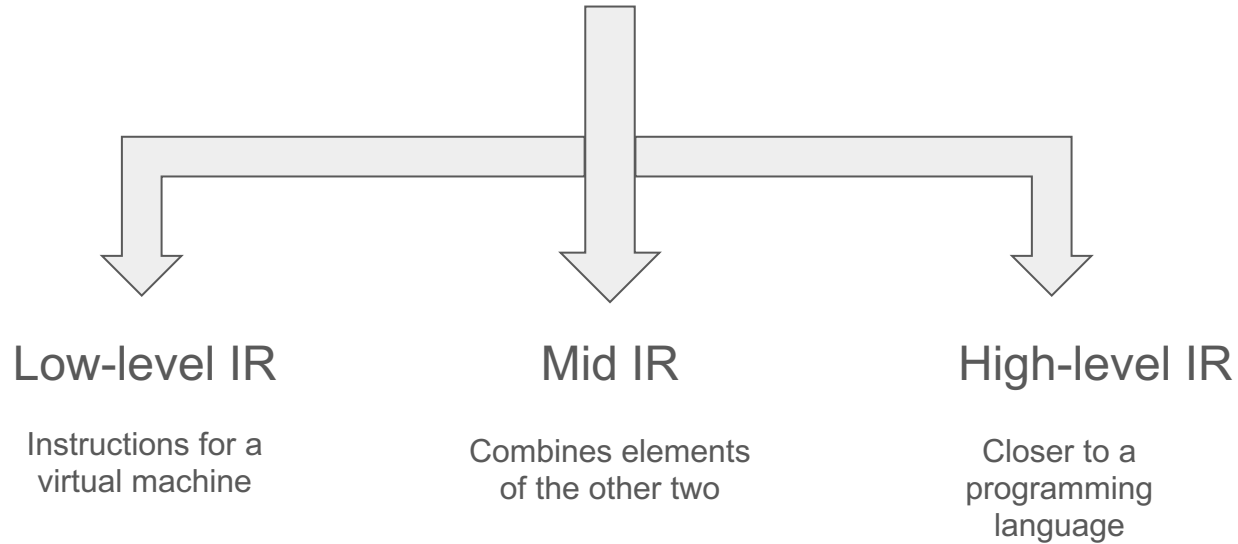
Without Wasm



Without Wasm



Types of Intermediate Representation



Portable assembly isn't new!

Pascal's P-Code

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

IBM TIMI

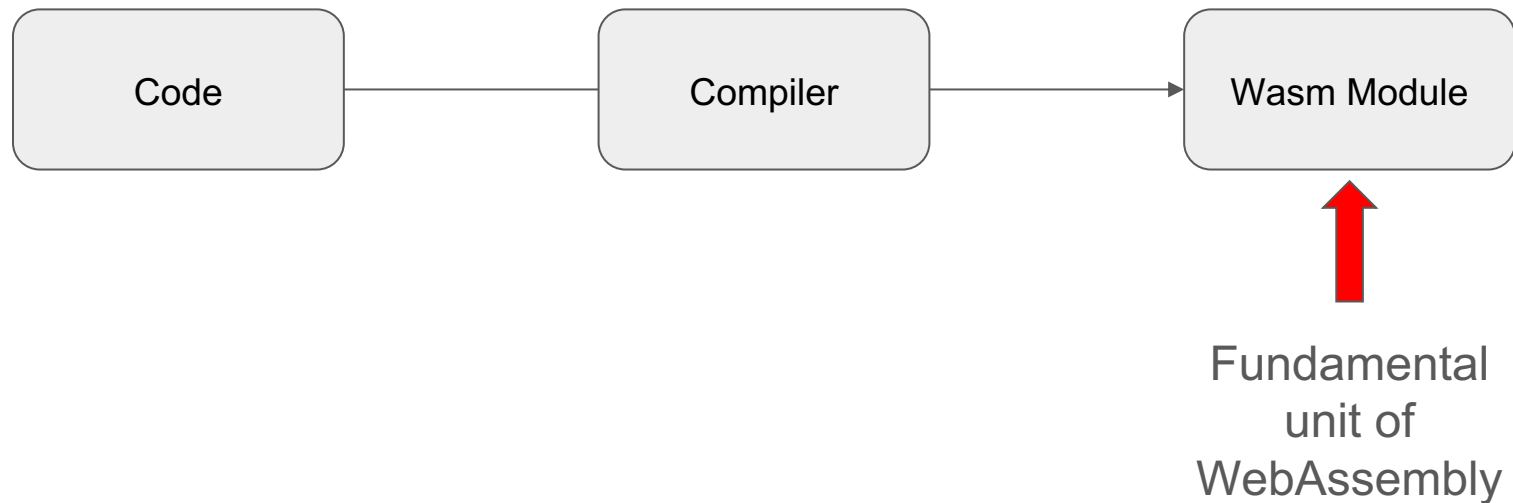
asm.js

Java bytecode & the
JVM

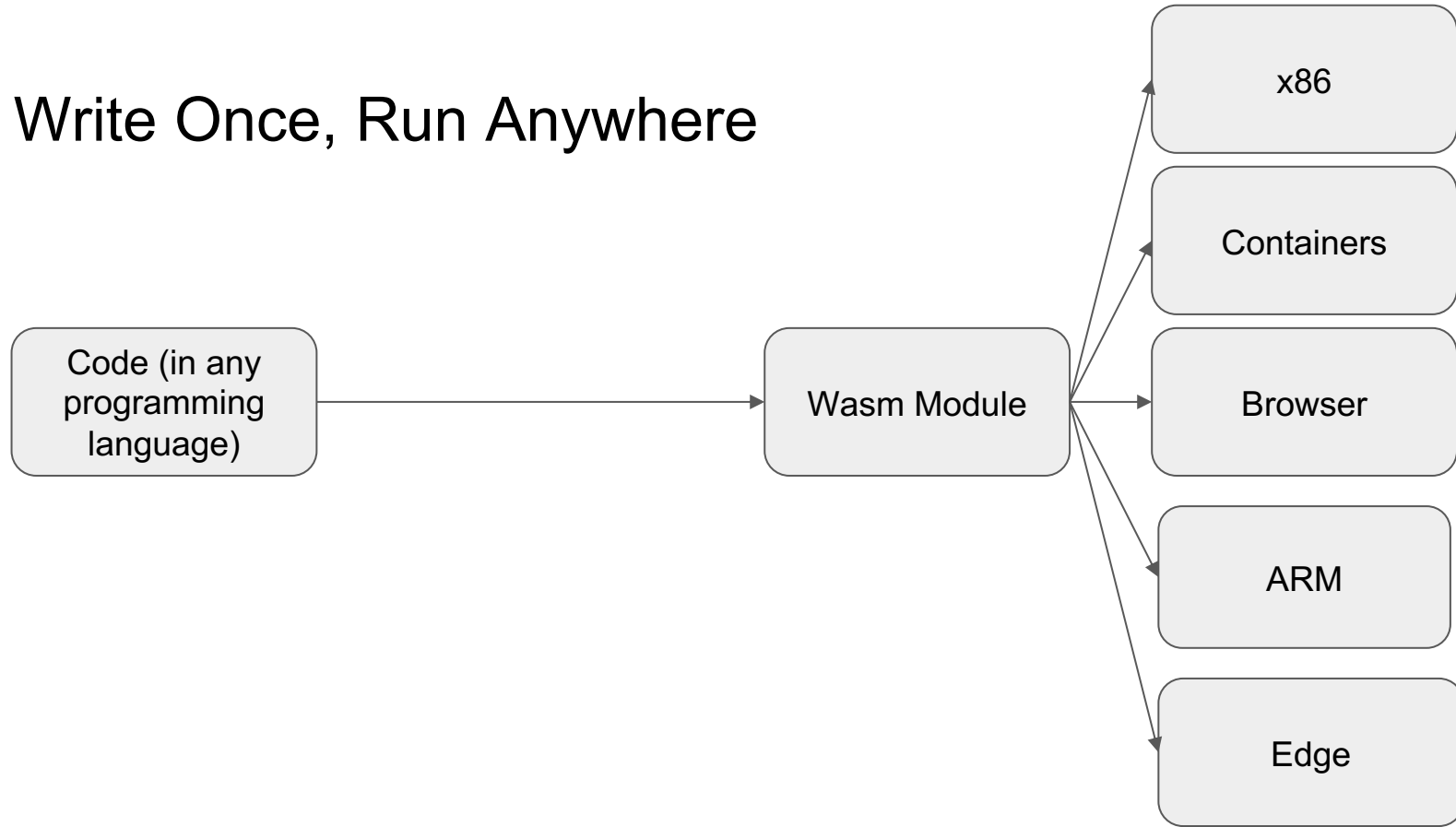
JavaScript

So, where does Wasm fit in?

Write Once, Run Anywhere!

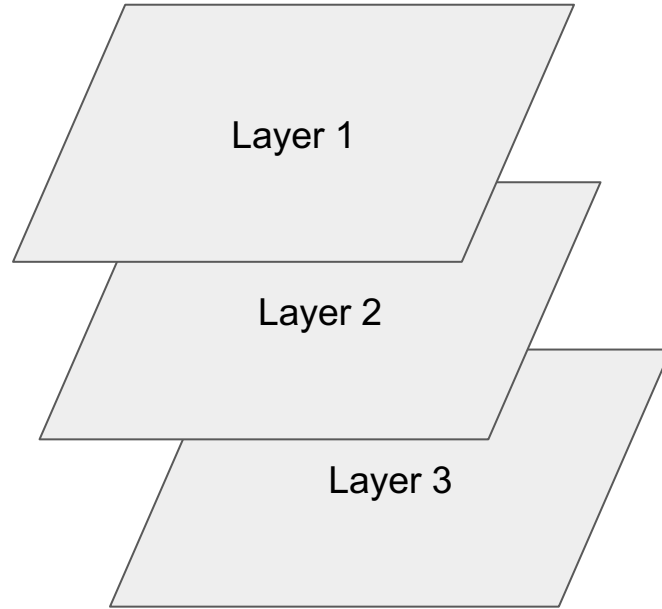


Write Once, Run Anywhere

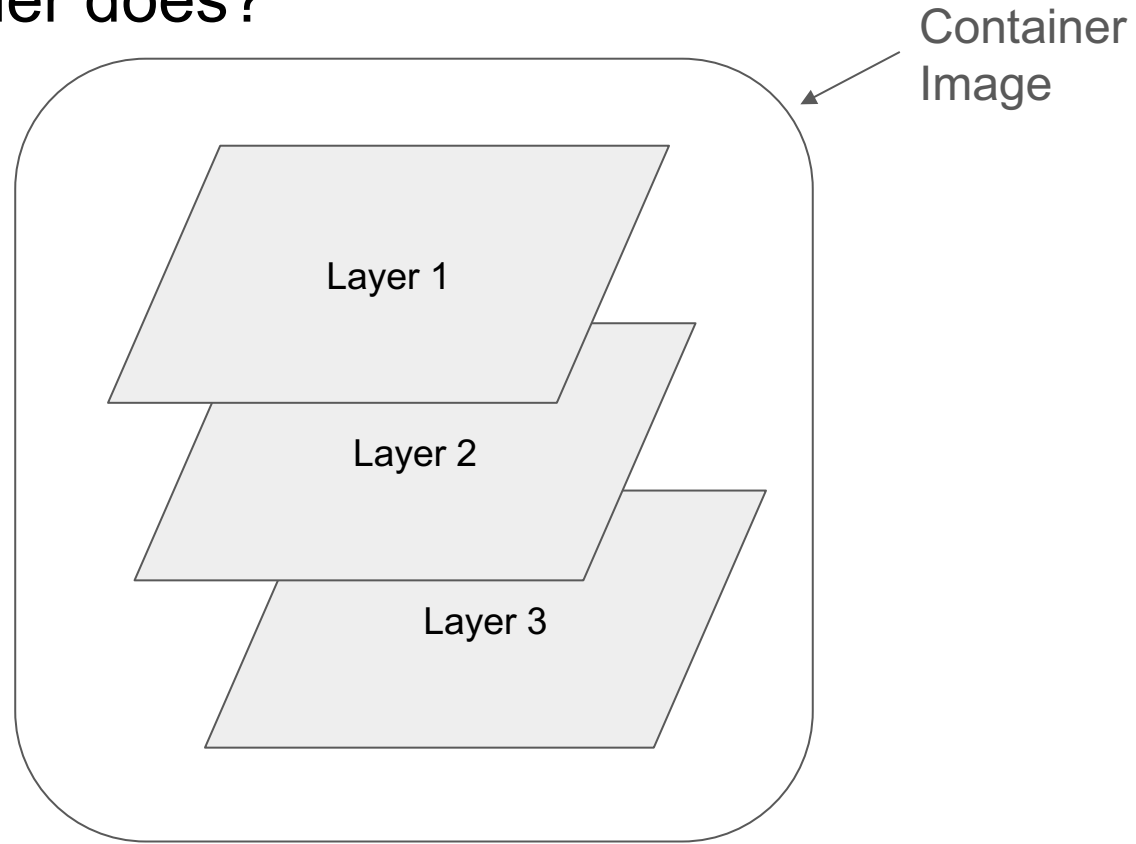


Hasn't this been achieved with
containerization?

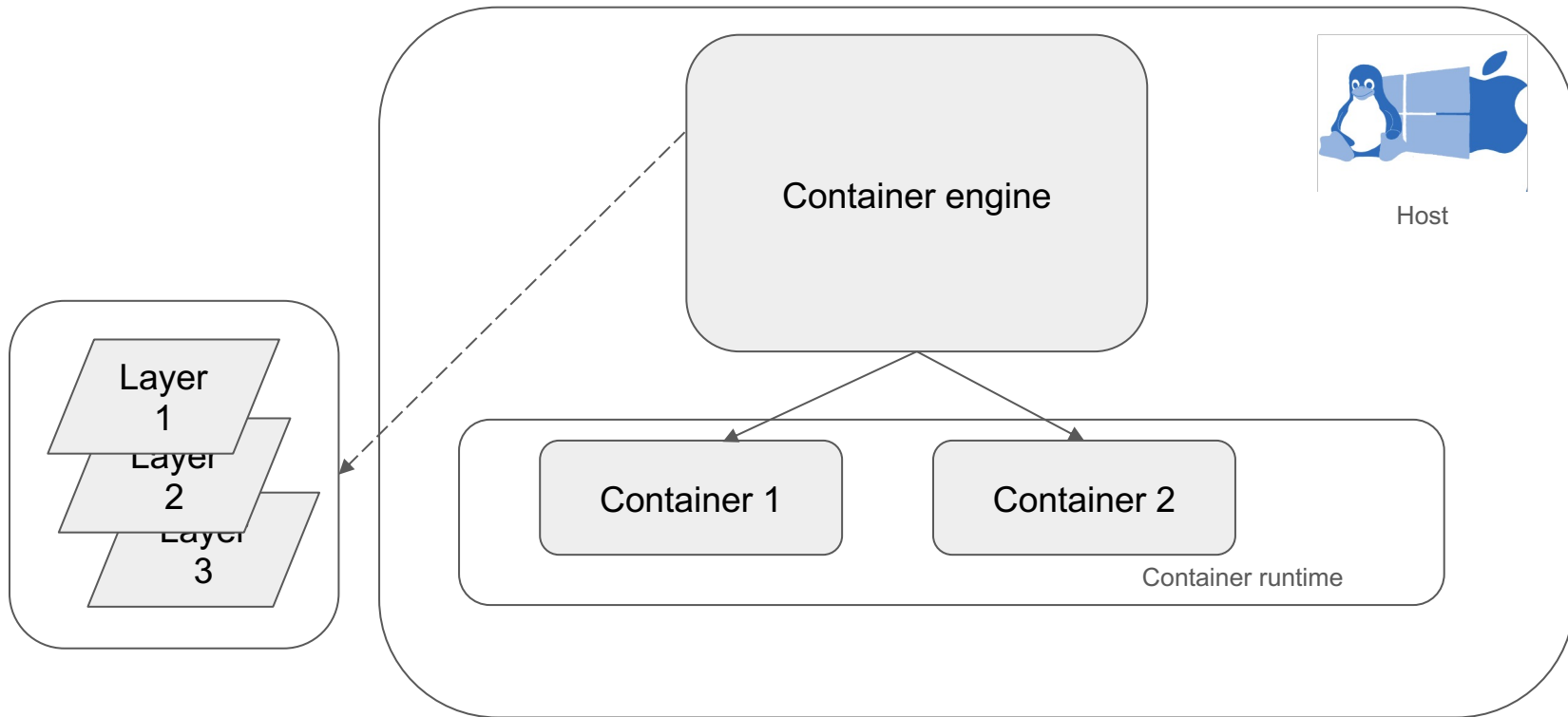
What a container does?



What a container does?



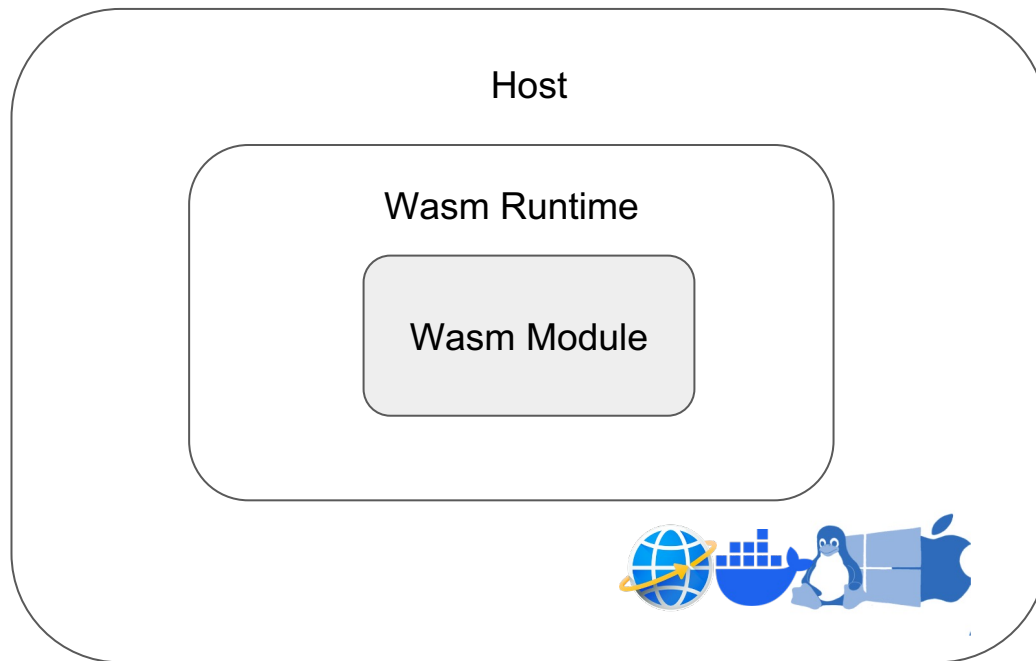
What a container does?



What Wasm does?

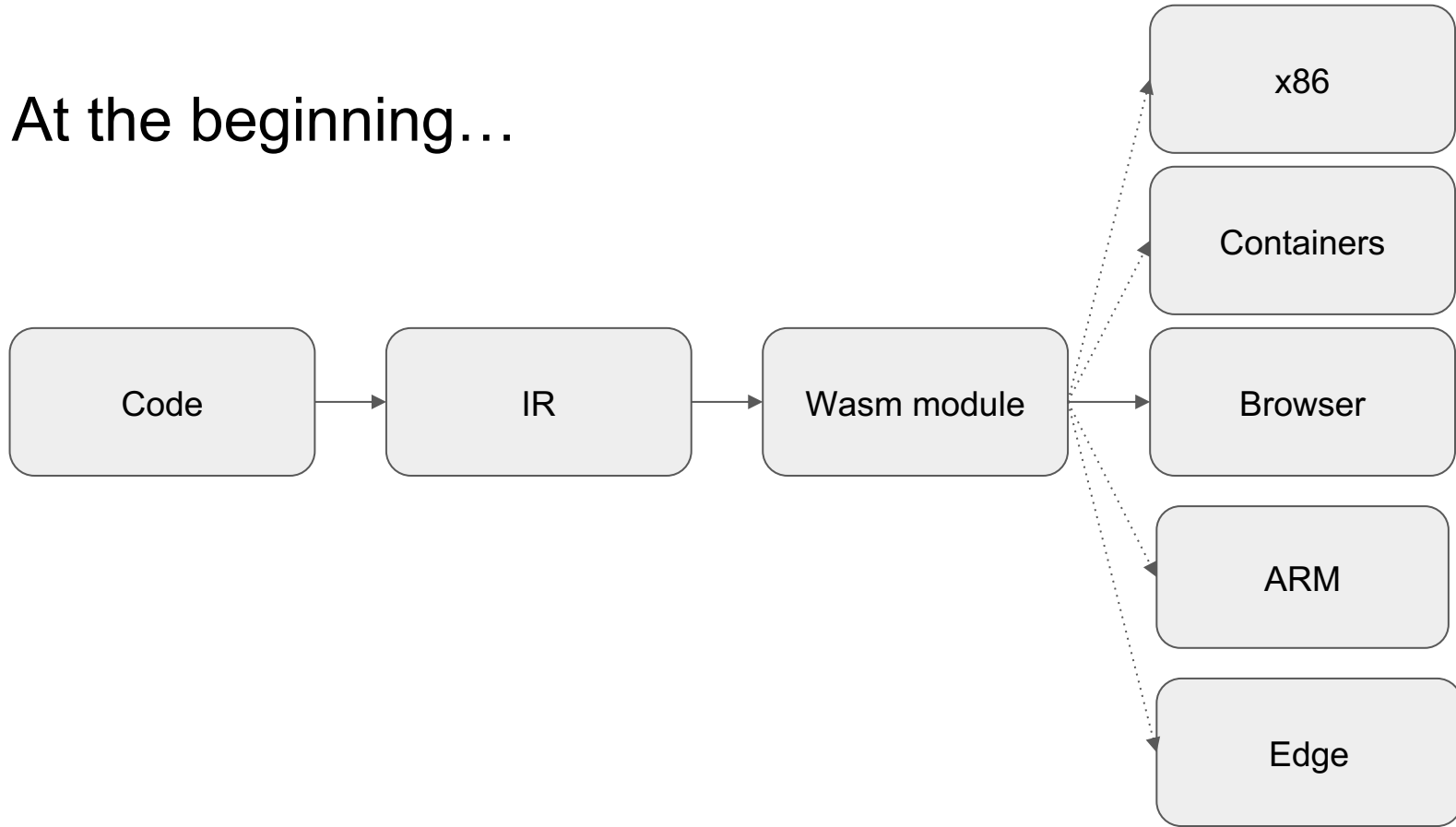


What Wasm does?



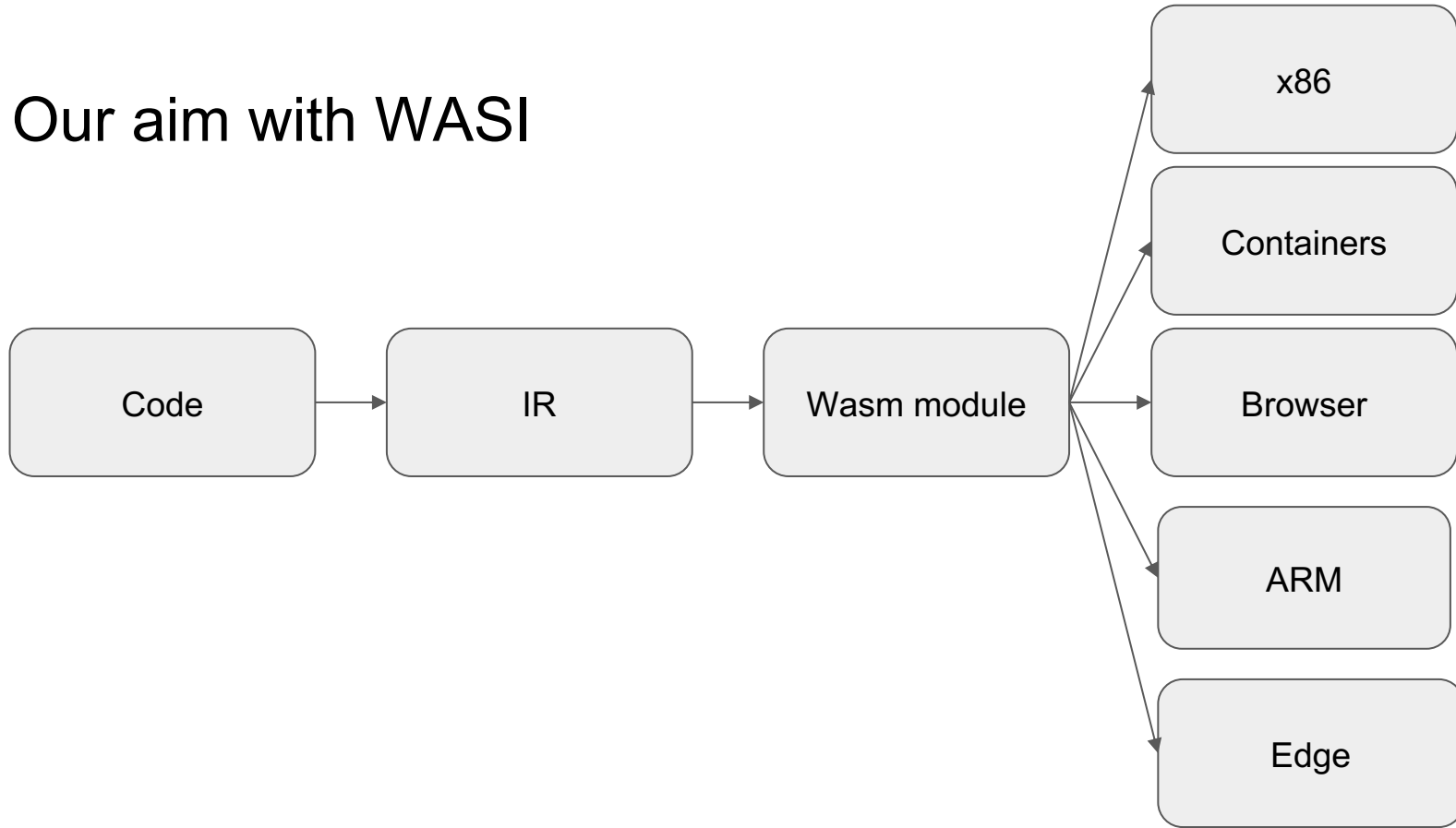
WASI

At the beginning...

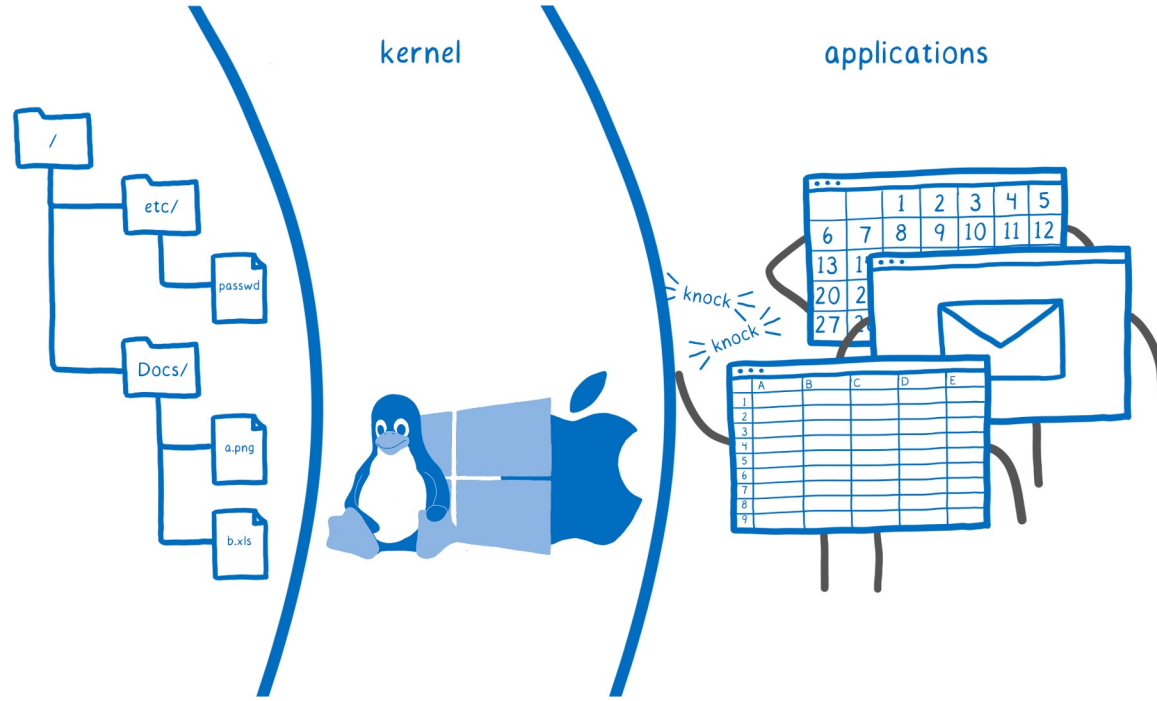


The WebAssembly System Interface (WASI) is a set of APIs for WASI being developed for eventual standardization by the WASI Subgroup, which is a subgroup of the WebAssembly Community Group.

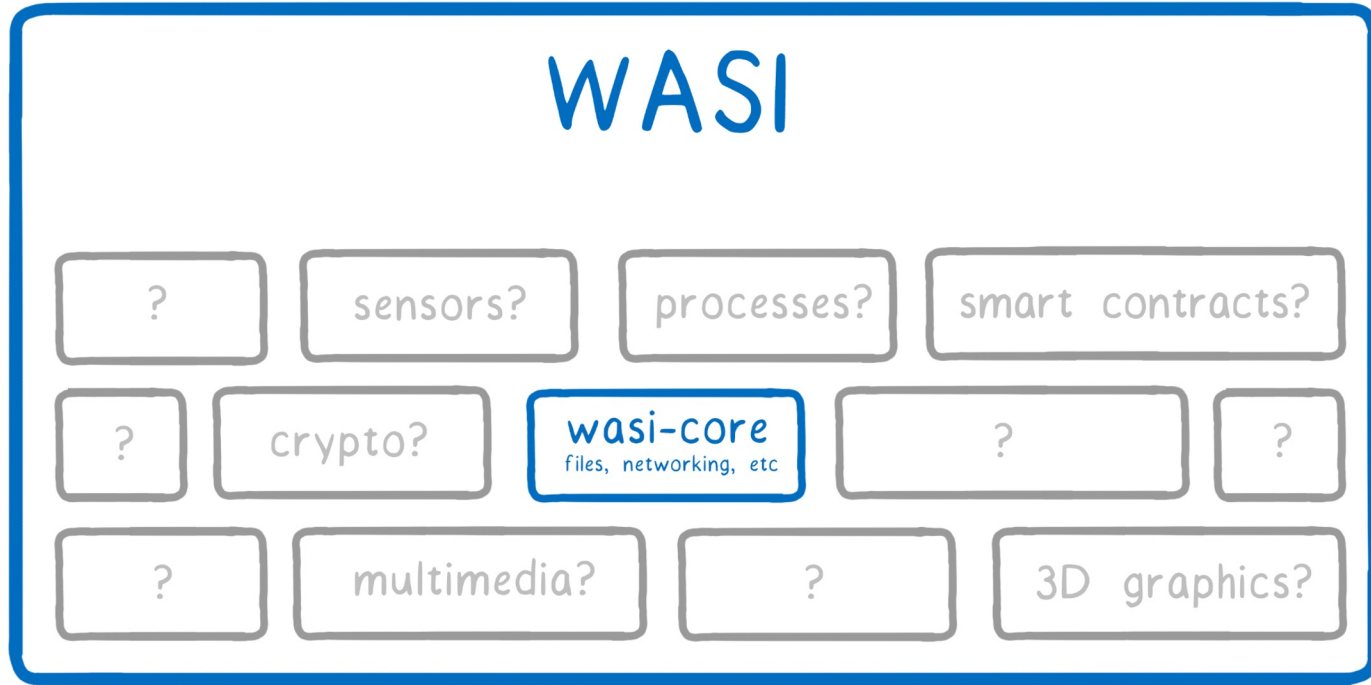
Our aim with WASI



What's a system interface?



How WASI proposes to implement it?



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

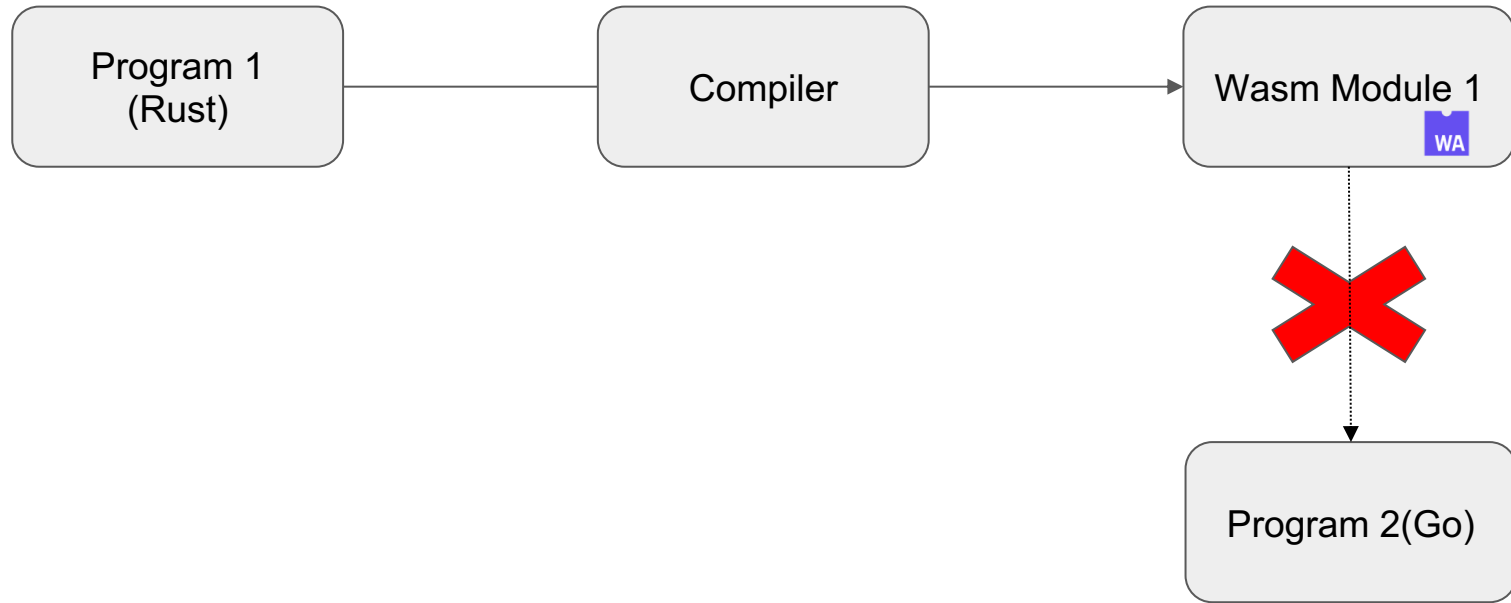
- wasi-io
- wasi-clocks
- wasi-random
- wasi-filesystem
- wasi-sockets
- wasi-cli
- wasi-http

Did you say the component model?

Let's go back here...

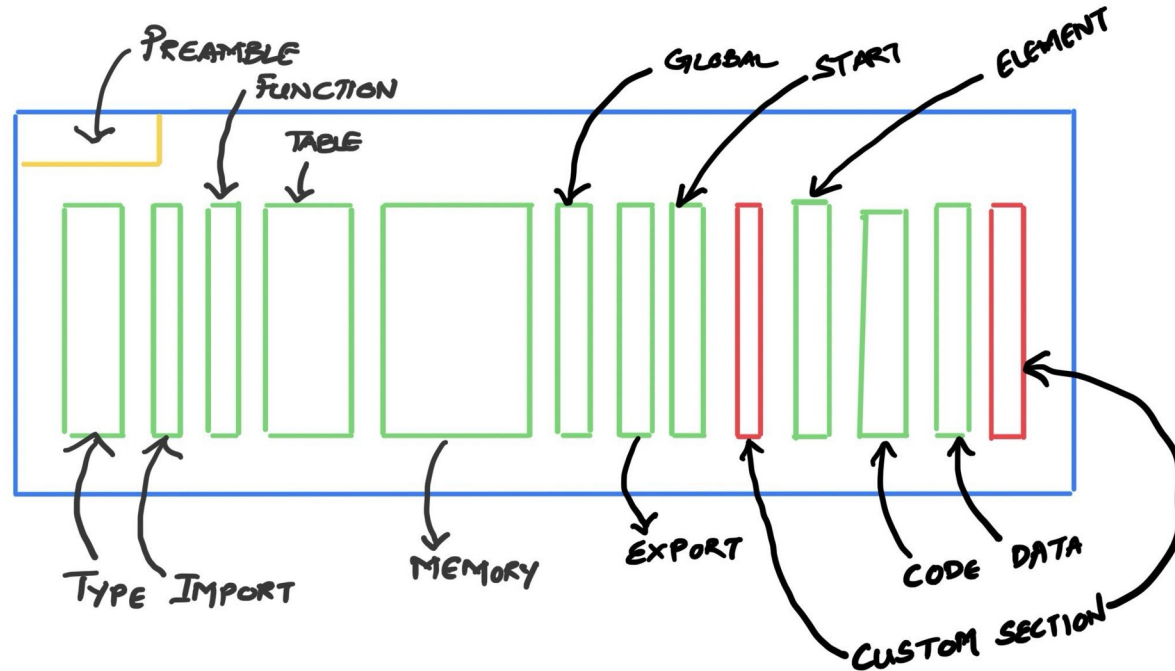


With WASI Preview 1



Why tho? 🤔

Cause a module kinda looks like this...



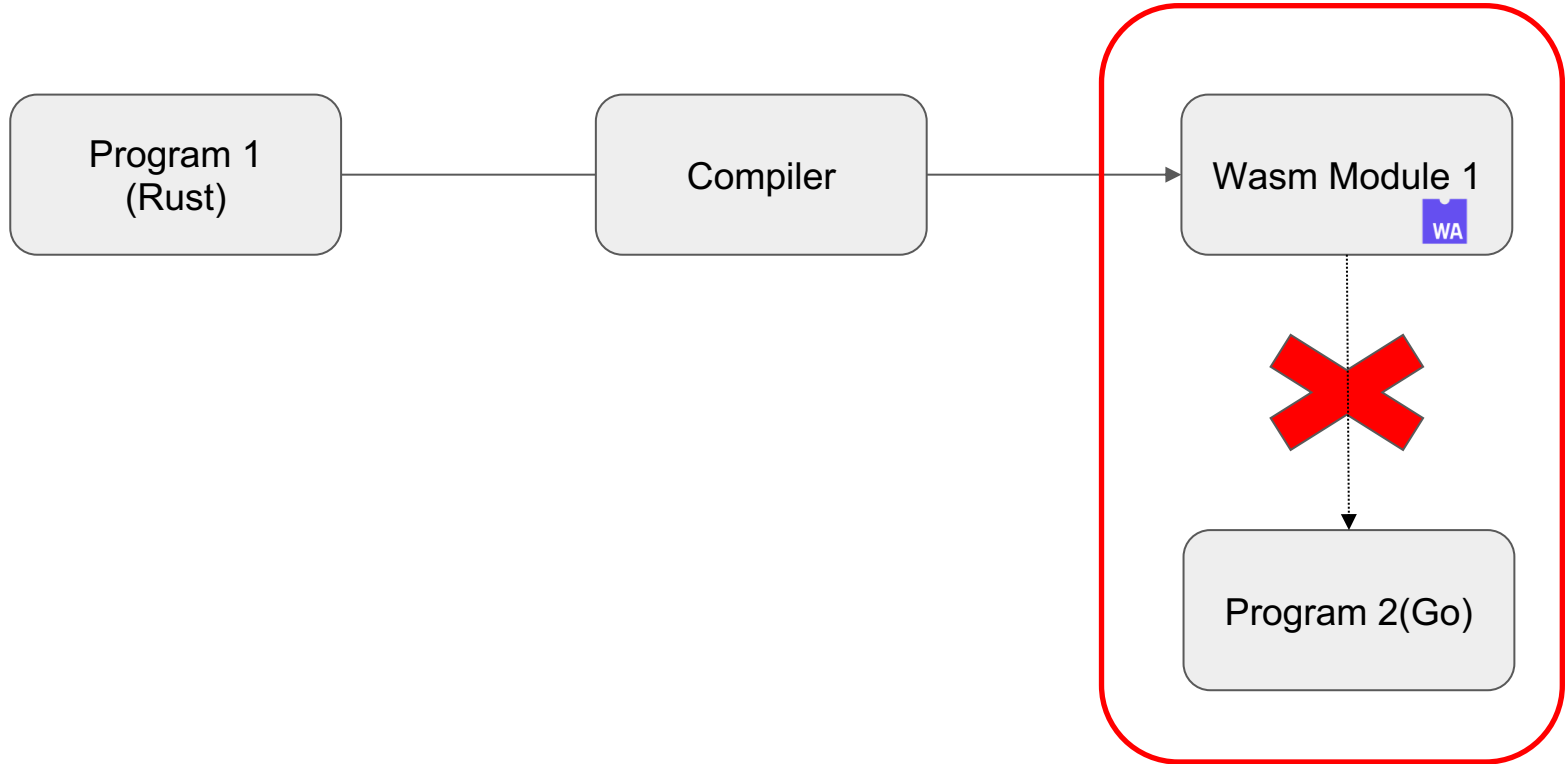
And supports...

```
(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)
```

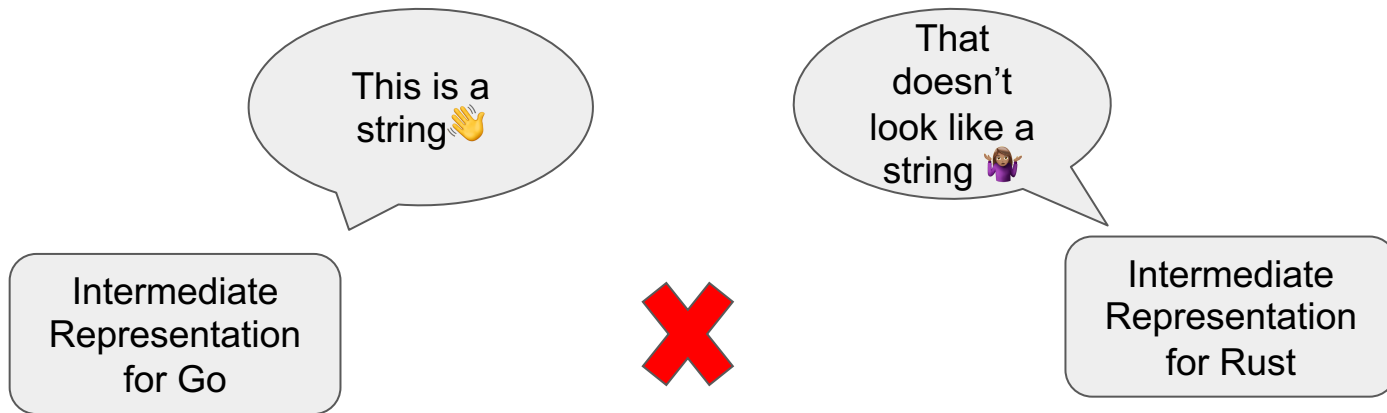
Therefore,

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

With WASI Preview 1



Zooming in...

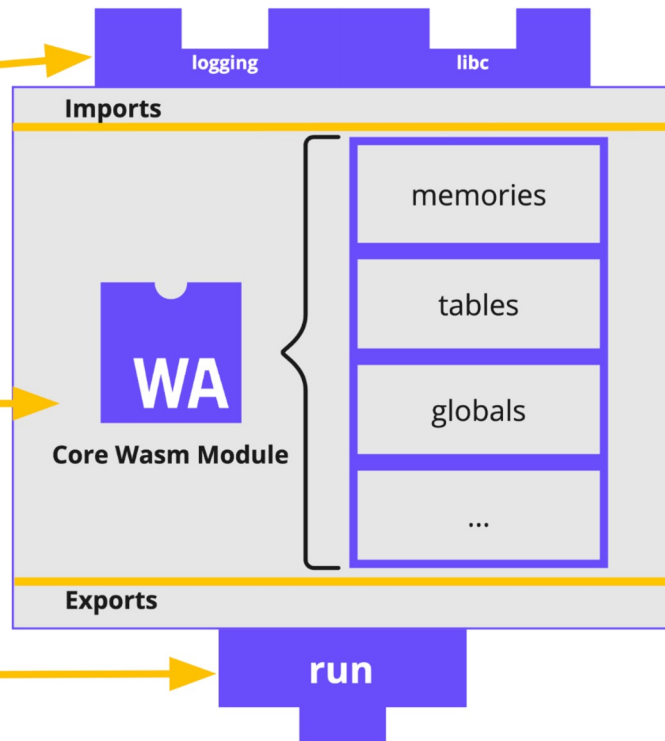


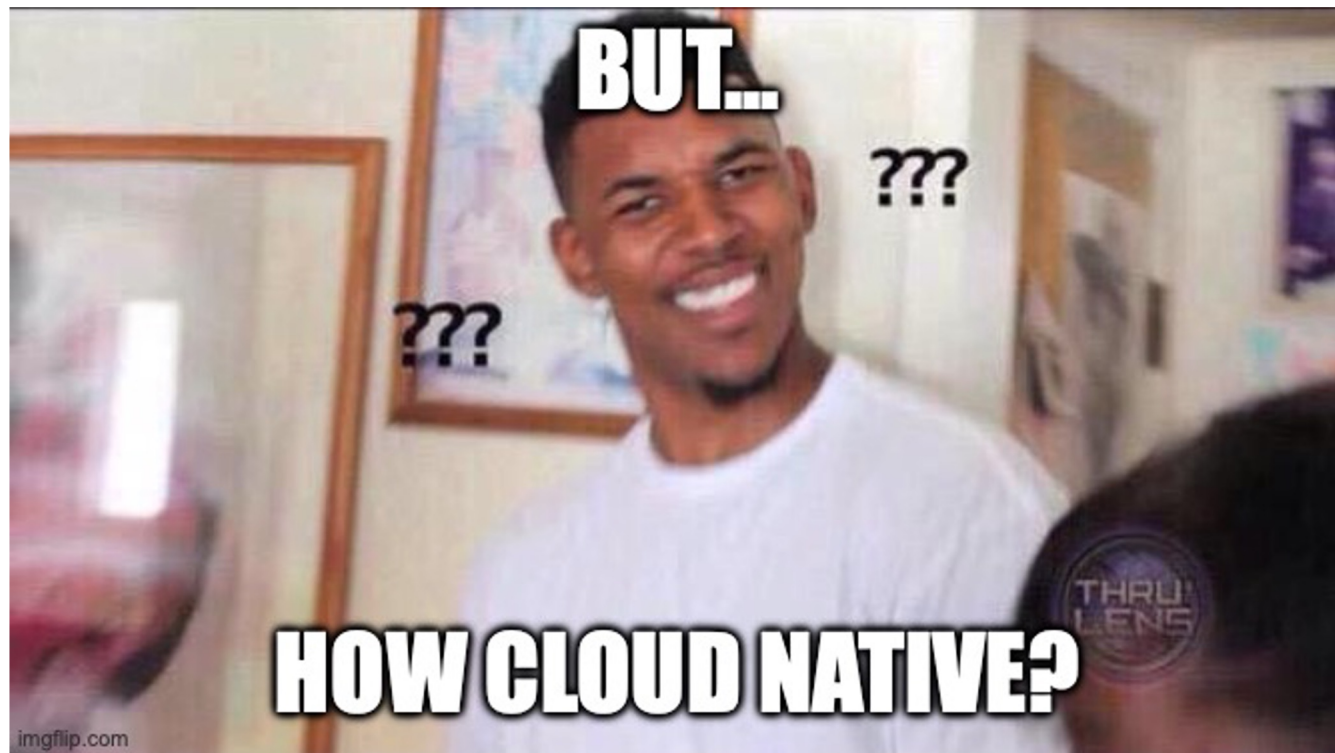
Enter the component model

- 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

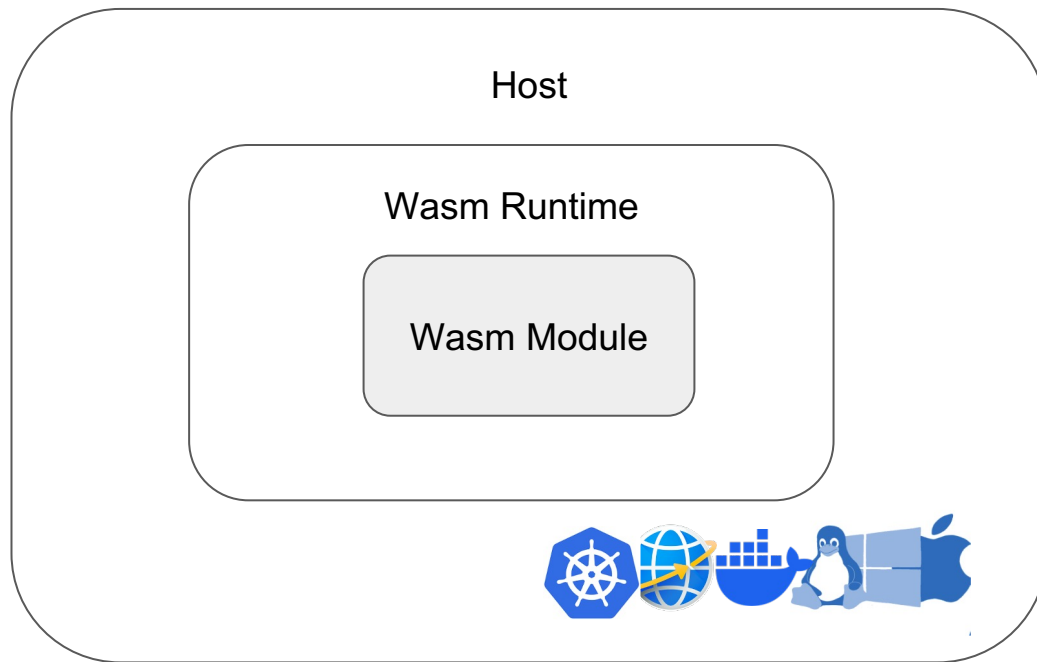
So, what does a component look like?

```
1 (component
2   (import "logging" (instance $logging
3     (export "log" (func (param string)))
4   ))
5   (import "libc" (core module $libc
6     (export "mem" (memory 1))
7     (export "realloc" (func (param i32 i32) (result i32)))
8   ))
9   (core instance $libc (instantiate $libc))
10  (core func $log (canon lower
11    (func $logging "log")
12    (memory (core memory $libc "mem")) (realloc (func $libc "realloc"))
13  ))
14  (core module $Main
15    (import "libc" "memory" (memory 1))
16    (import "libc" "realloc" (func (param i32 i32) (result i32)))
17    (import "logging" "log" (func $log (param i32 i32)))
18    (func (export "run") (param i32 i32) (result i32)
19      ... (call $log) ...
20    )
21  )
22  (core instance $main (instantiate $Main
23    (with "libc" (instance $libc))
24    (with "logging" (instance (export "log" (func $log))))
25  ))
26  (func $run (param string) (result string) (canon lift
27    (core func $main "run")
28    (memory (core memory $libc "mem")) (realloc (func $libc "realloc"))
29  ))
30  (export "run" (func $run))
31)
```





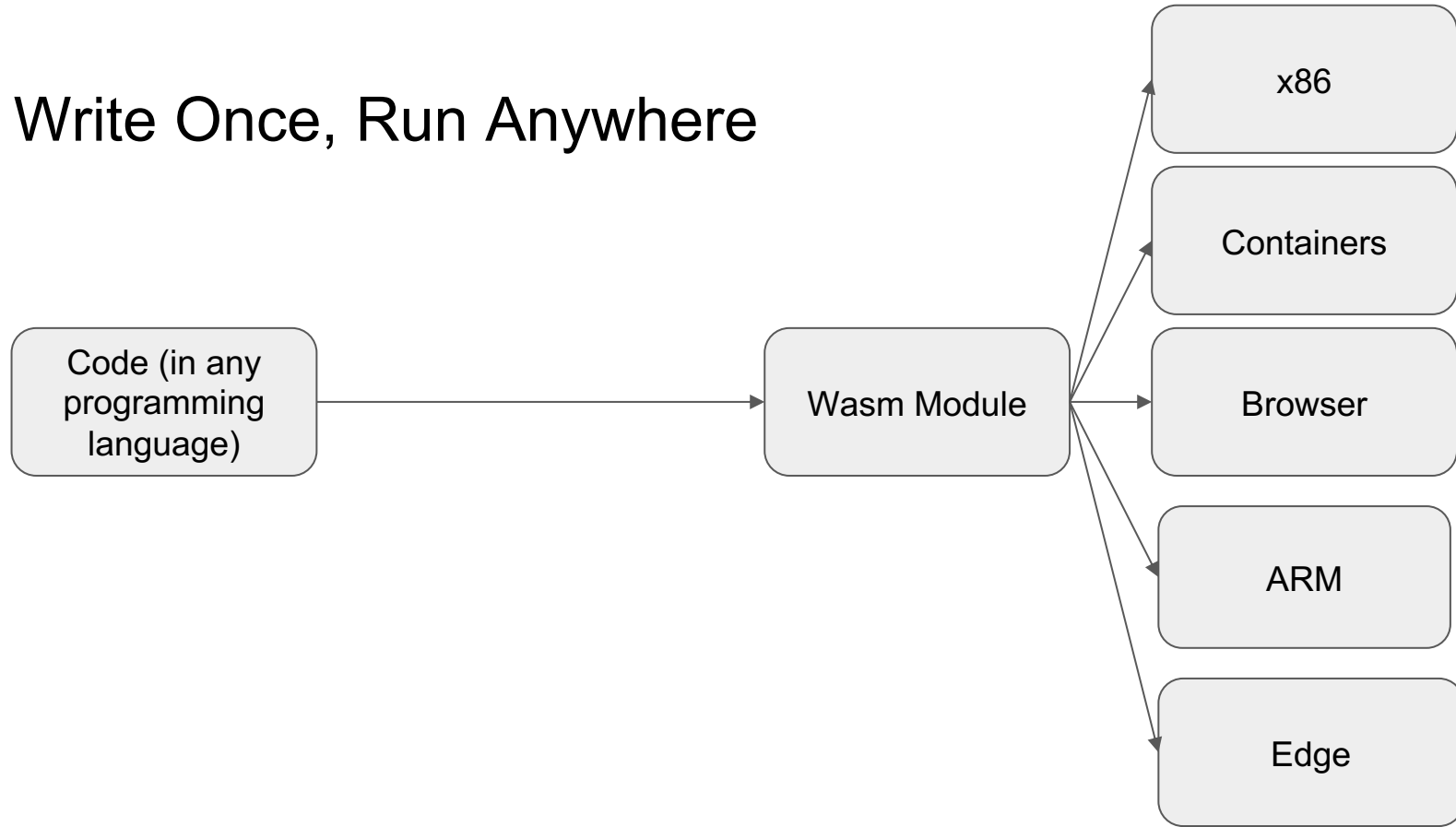
Bringing this back...



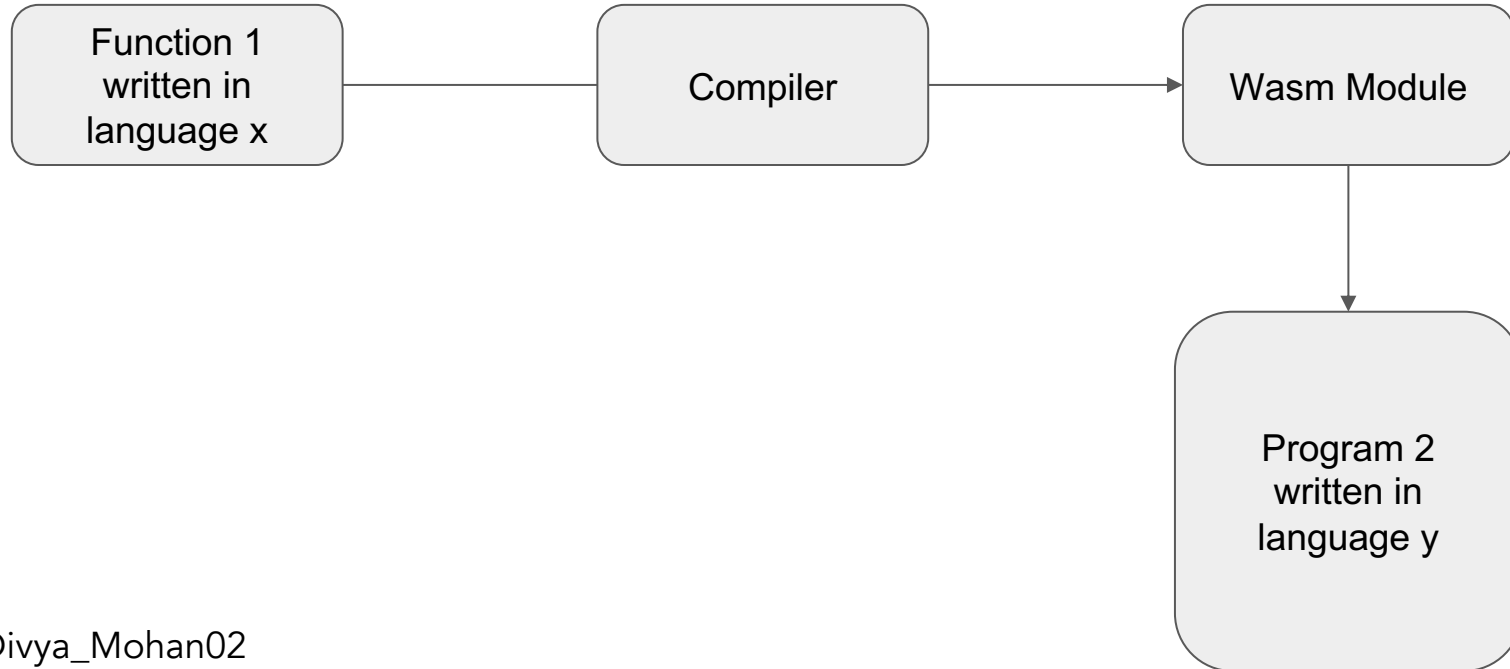
Live demo!

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](#)

Resources

- [Getting started developer's guide](#)
- [Various proposals for Wasm](#)
 - [Garbage collection proposal](#)
- Trainings:
 - <https://training.linuxfoundation.org/training/introduction-to-webassembly-lfd133/>
 - <https://www.linuxfoundation.org/research/webassembly-for-legal-professionals>
 - <https://training.linuxfoundation.org/training/webassembly-actors-from-cloud-to-edge-lfd134x/>
- [Guy Bedford + Peter Huene's session on the future of component tooling](#)

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!