

Jug Summer Camp

-enjoy it-



Server-side WebAssembly

Speaker: Horacio Gonzalez
@LostInBrittany



SERLI

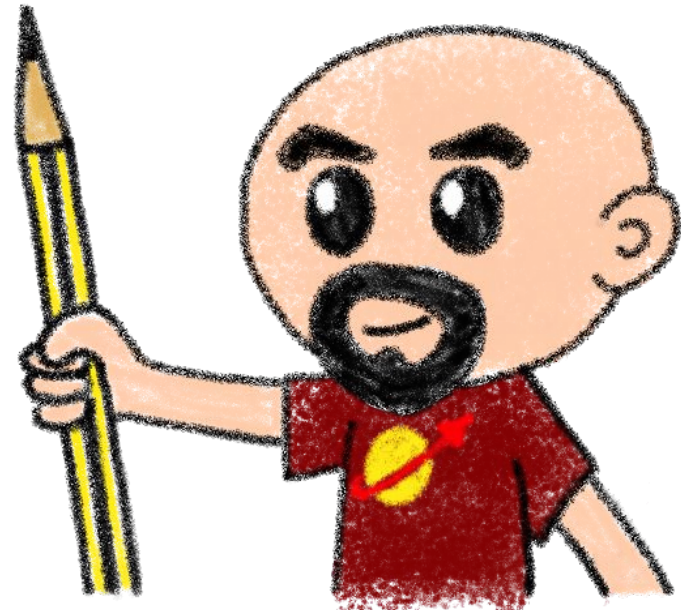


Horacio Gonzalez



@LostInBrittany

Spaniard lost in Brittany,
developer, dreamer and
all-around geek



OVHcloud: A Global Leader



200k Private cloud VMs running



Dedicated IaaS Europe

...
...
...
...
...
...
...
...



30 Datacenters

Hosting capacity :
1.3M Physical Servers

360k
Servers already
deployed

> **1.3M** Customers in **138** Countries





Did I say WebAssembly?

Wasm for the friends...



WebAssembly, what's that?



What's WASM?

Does it replace JS?

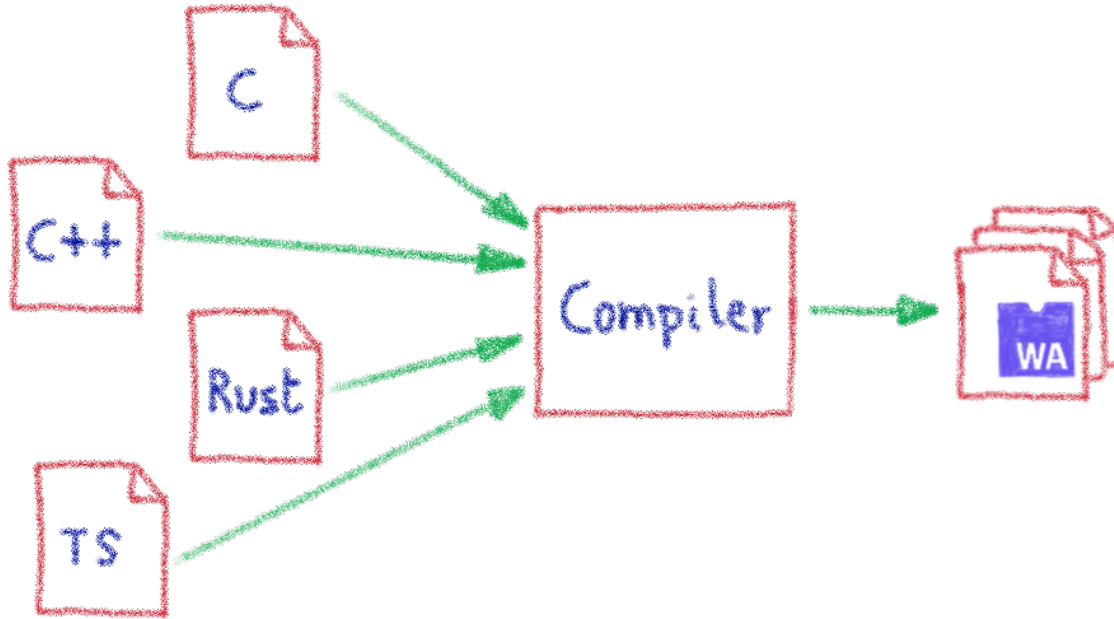
Can I code webapps in Rust?



Is HTML/CSS/JS stack obsolete?



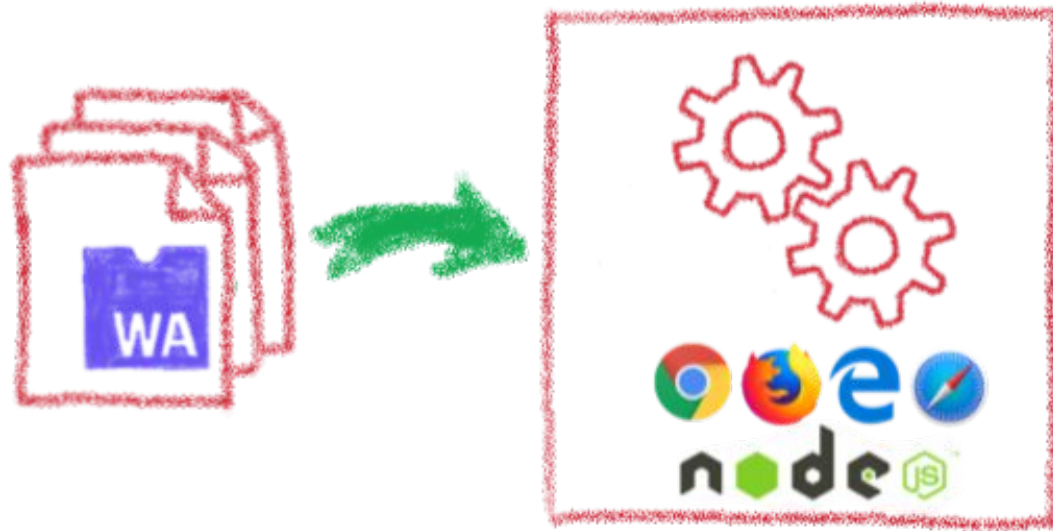
A low-level binary format



Not a programming language
A compilation target



That runs on a stack-based virtual machine



A portable binary format that runs on all modern browsers... but also on NodeJS and elsewhere!!





With several key advantages

Fast & Efficient ⚡

🔒 Memory-safe & Sandboxed

Open & Debuggable 📄

www Part of the Web Platform



But above all...



WebAssembly is not meant to replace JavaScript



Who is using WebAssembly today?



And many more others...





A bit of history

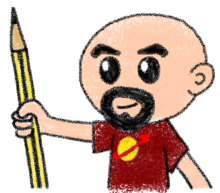
Remembering the past
to better understand the present



Executing other languages in the browser



A long story, with many failures...



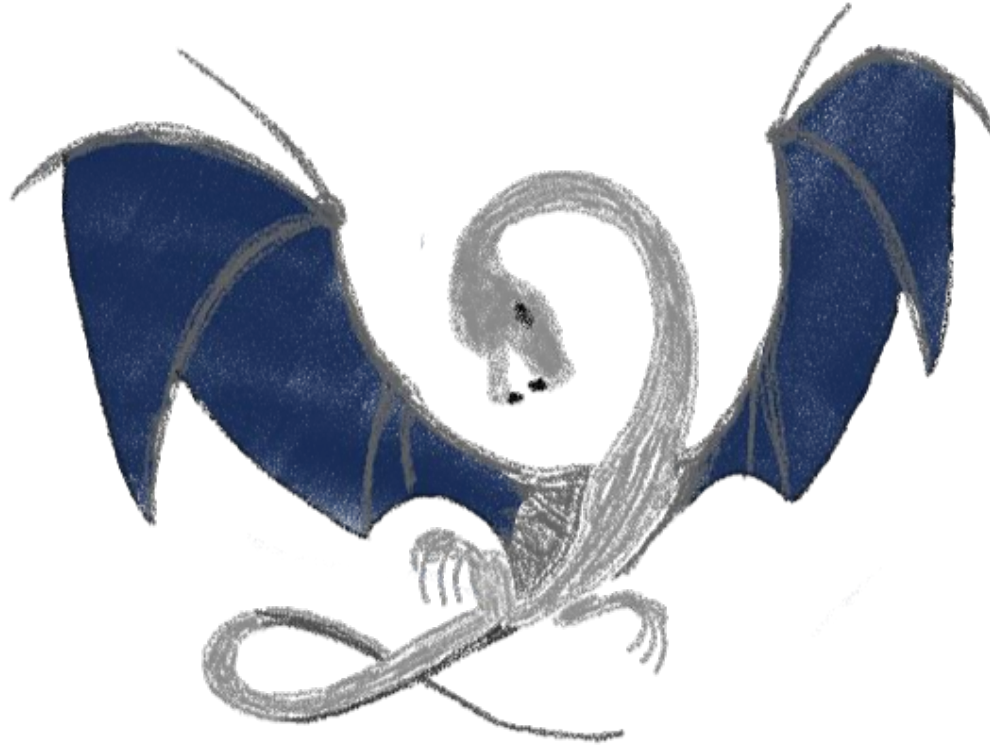
2012 - From C to JS: enter emscripten



Passing by LLVM pivot



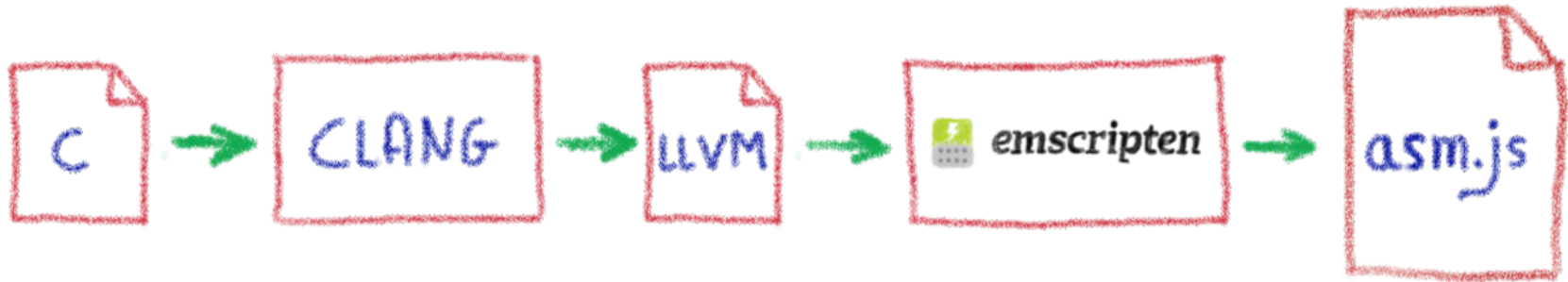
Wait, dude! What's LLVM?



A set of compiler and toolchain technologies



2013 - Generated JS is slow...



Let's use only a strict subset of JS: asm.js

Only features adapted to AOT optimization



WebAssembly project



moz:lla

Google



W3C

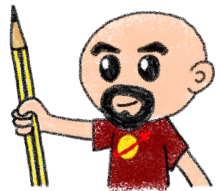
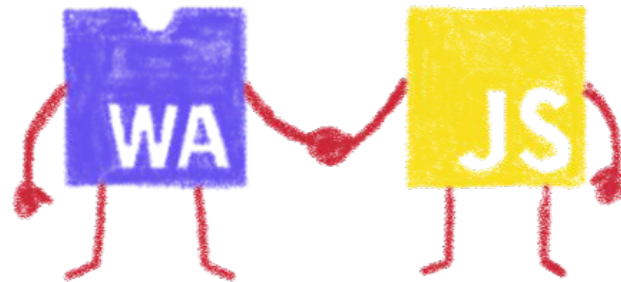
Joint effort





Hello W(asm)orld

My first WebAssembly program



I don't want to install a compiler now...



WebAssembly Explorer v2.18

Please also check out WebAssembly Studio

Options

- Auto Compile
- LLVM x86 Assembly
- Examples
 - C++11
- Optimization Level
 - s
- Fast Math
- No Inline
- No RTTI
- No Exceptions
- Clean WAT
- Baseline JIT

OPEN IN WASMFIDDLE

Console

```
1 Welcome to the WebAssembly Explorer
2 =====
3
4 Here you can translate C/C++ to WebAssembly, and then see the machine code generated by the browser.
5
6 For bugs, comments and suggestions see: https://github.com/mbebenita/WasmExplorer
7 Built with Clang/LLVM, AngularJS, Ace Editor, Emscripten, SpiderMonkey, Binaryen and Capstone.js.
8
9 Service version 3.5 (js: JavaScript-C55.0a1; clang: 5.0.0 (https://chromium.googlesource.com/external/github.com/llvm-mirror
10
11
```

Let's use Wasm Explorer

<https://mbebenita.github.io/WasmExplorer/>



Let's begin with the a simple function

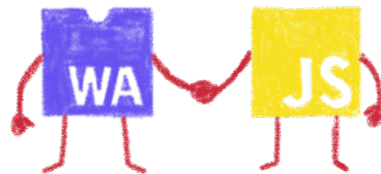
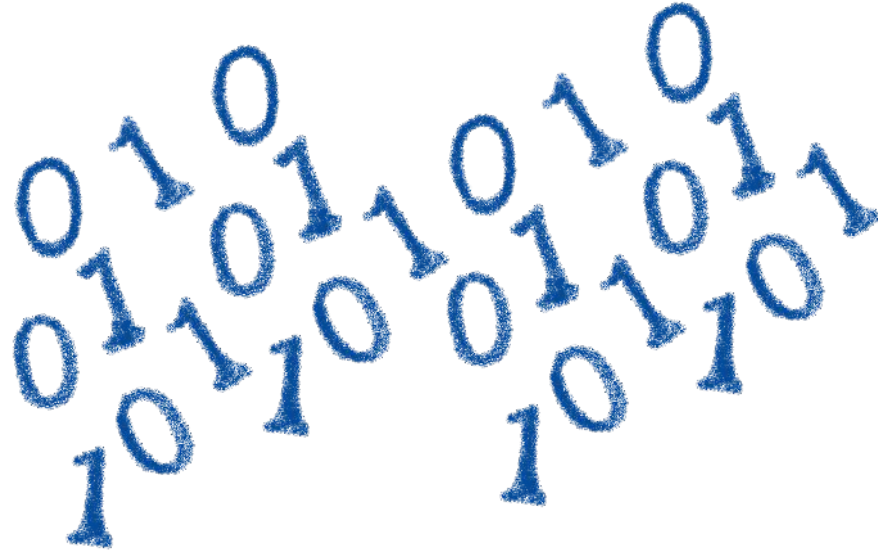


C++11 -Os	COMPILE	Wat	ASSEMBLE	DOWNLOAD	Firefox x86 Assembly
<pre>1 int squarer(int num) { 2 return num * num; 3 }</pre>		<pre>1 (module 2 (type \$type0 (func (param i32) 3 (result i32))) 4 (table 0 anyfunc) 5 (memory 1) 6 (export "memory" memory) 7 (export "_Z7squareri" \$func0) 8 (func \$func0 (param \$var0 i32) 9 (result i32) 10 get_local \$var0 11 get_local \$var0 12 i32.mul 13) 14)</pre>			<pre>- wasm-function[0]: sub rsp, 8 mov edx, edi mov ecx, edx mov eax, edx imul ecx, eax mov eax, ecx nop add rsp, 8 ret</pre>

WAT: WebAssembly Text Format
Human readable version of the .wasm binary



Download the binary .wasm file



Now we need to call it from JS...



Instantiating the Wasm



1. Get the .wasm binary file into an array buffer
2. Compile the bytes into a WebAssembly module
3. Instantiate the WebAssembly module



Instantiating the WASM



```
wasm > squarer > JS squarer.js > ...  
  
3   var importObject = {  
4     imports: {  
5       imported_func: function(arg) {  
6         console.log(arg);  
7       }  
8     }  
9   };  
10  
11  async function loadWebAssembly() {  
12    let response = await fetch('squarer.wasm');  
13    let arrayBuffer = await response.arrayBuffer();  
14    let wasmModule = await WebAssembly.instantiate(arrayBuffer, importObject);  
15    squarer = await wasmModule.instance.exports._Z7squareri;  
16    console.log('Finished compiling! Ready when you are...');  
17  }  
18  
19  loadWebAssembly();  
20
```



Loading the squarer function



```
wasm > squarer > <> squarer.html > ...
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8" />
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <title>WASM Squarer Function</title>
7      <meta name="viewport" content="width=device-width, initial-scale=1">
8  </head>
9  <body>
10
11      <h1>WASM Squarer Function</h1>
12
13      <script src="squarer.js"></script>
14
15      <p>Use the browser console to calculate squares</p>
16  </body>
17  </html>
18
19
```

We instantiate the WASM by loading the wrapping JS



Using it!



WASM Squarer Function

Use the browser console to calculate squares

WA JS

```
Finished compiling! Ready when you are... squares:16
> squarer(3)
< 9
> squarer(11)
< 121
>
```

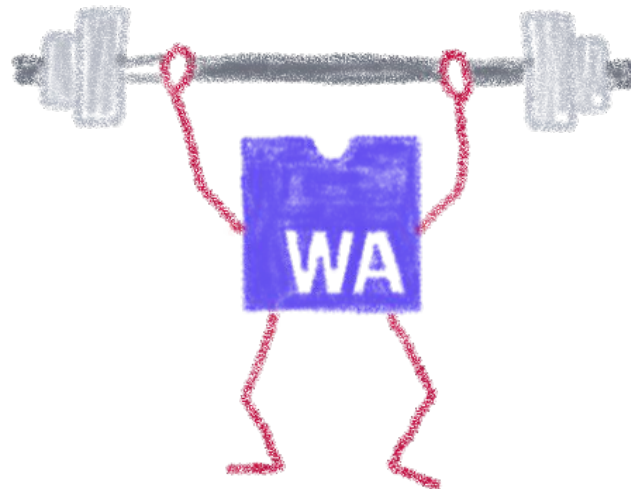
Directly from the browser console
(it's a simple demo...)





Features of Wasm

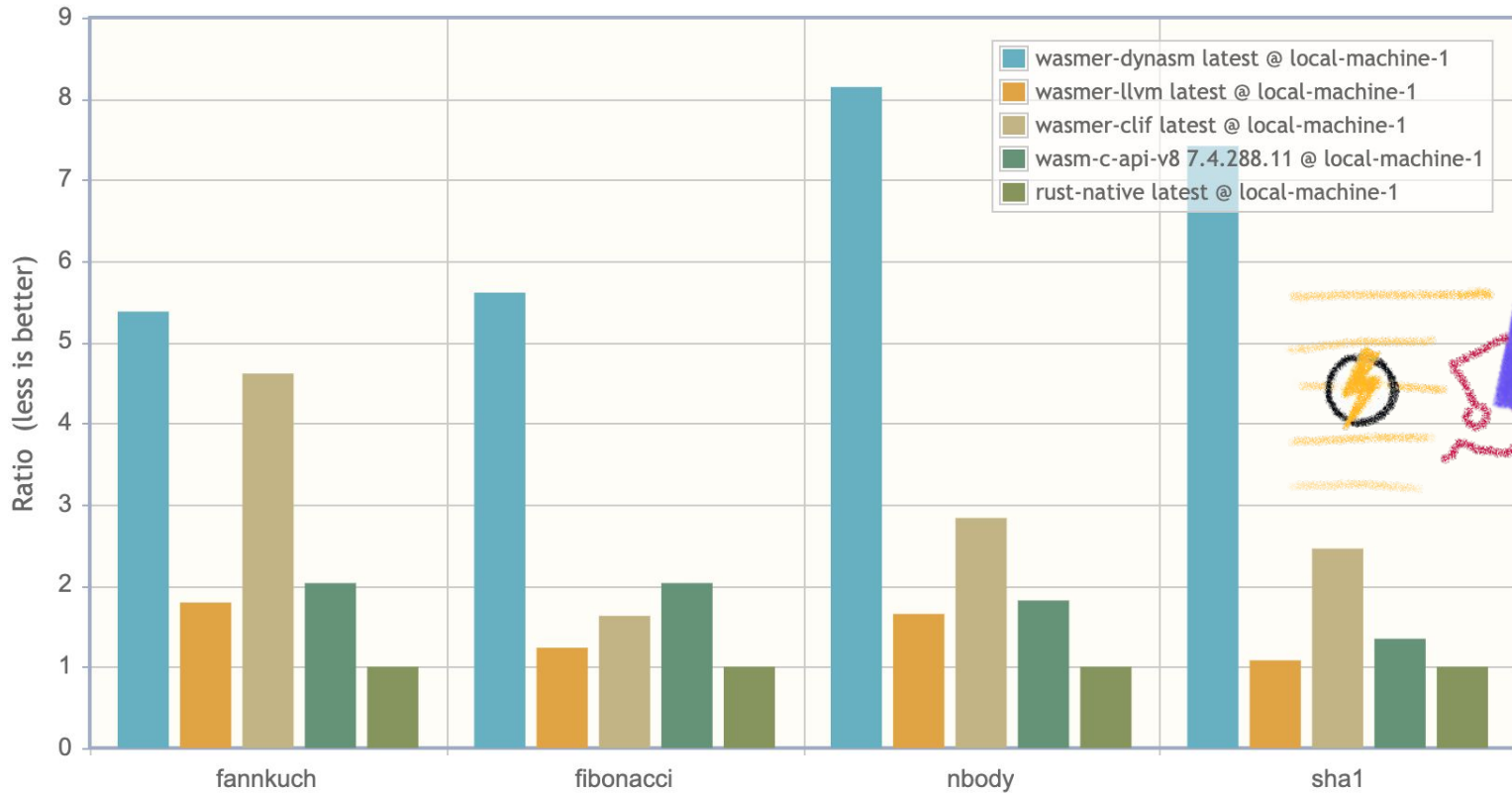
Why is everybody looking at it?



Near native speed



Time normalized to rust-native latest



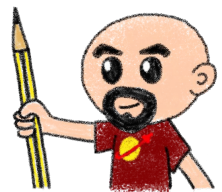
<https://medium.com/wasmer/benchmarking-webassembly-runtimes-18497ce0d76e>



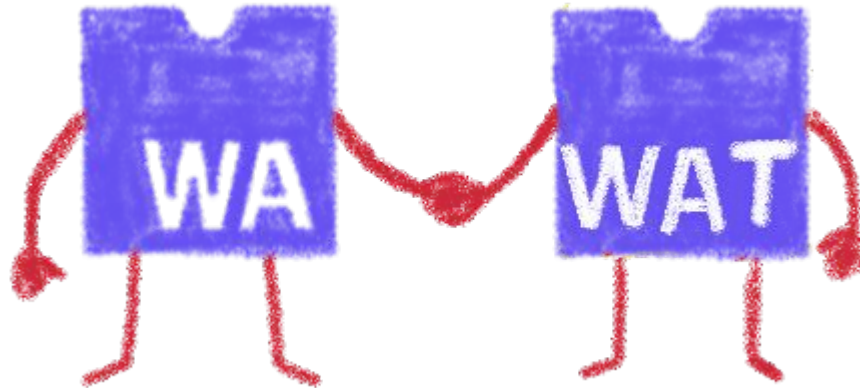
Highly portable



It can be run almost everywhere...



Readable and debuggable



Each `.wasm` file with its `.wat` companion file



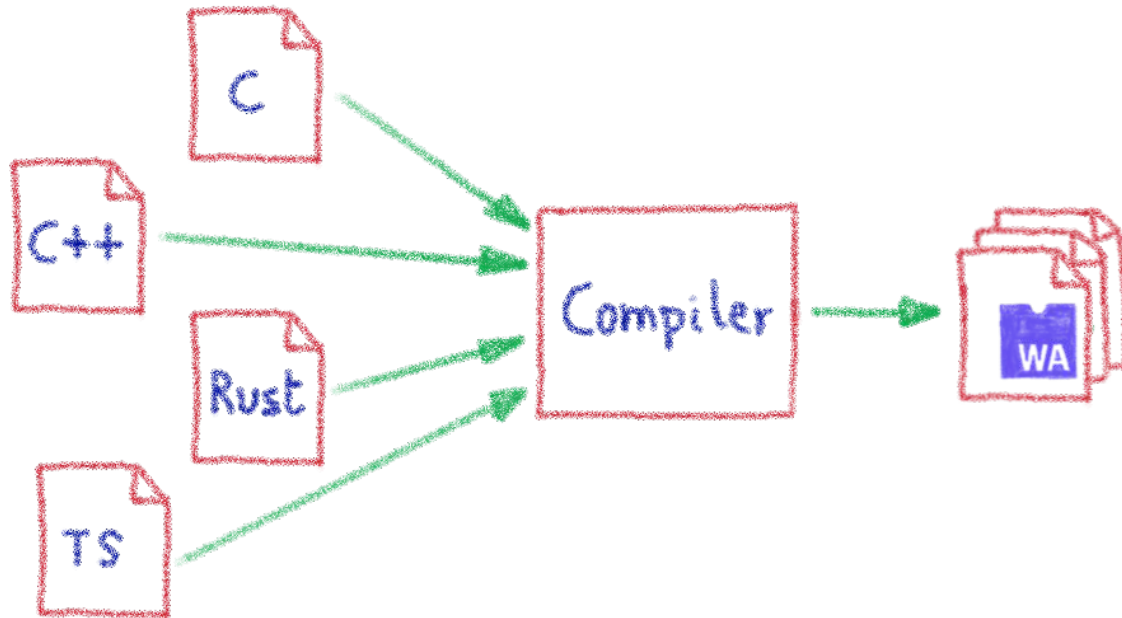
Memory safe & secure



Running in a fully sandboxed environment



Accepting many source languages



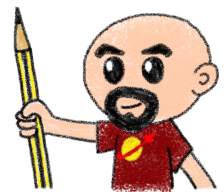
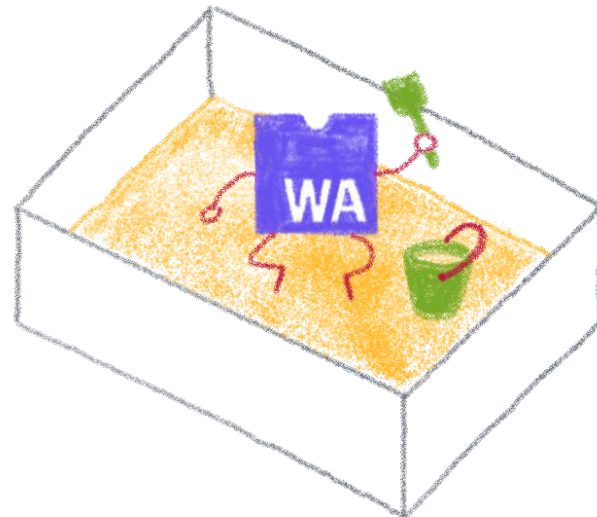
And more and more...





Some constraints

Still a young platform



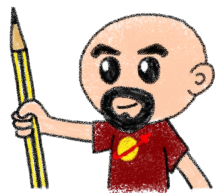
Native WASM types are limited



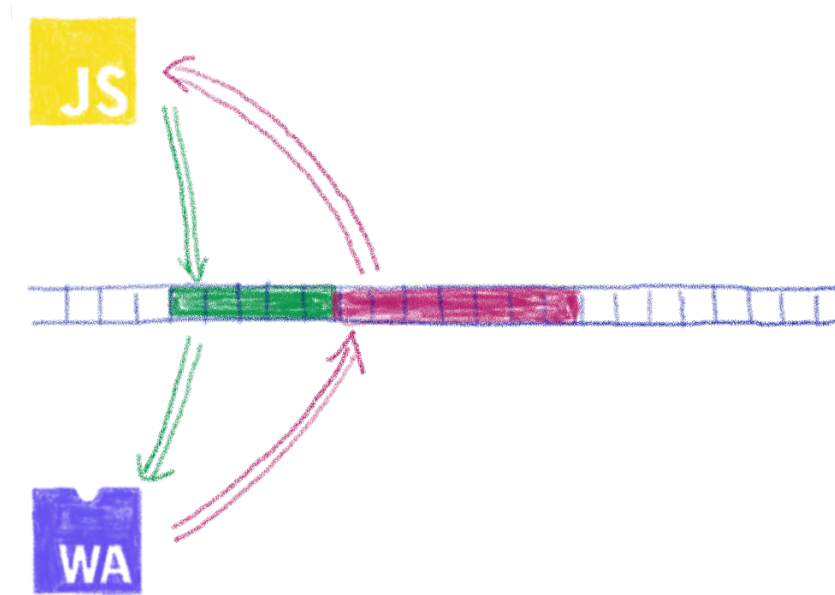
WASM currently has four available types:

- **i32**: 32-bit integer
- **i64**: 64-bit integer
- **f32**: 32-bit float
- **f64**: 64-bit float

Types from languages compiled to WASM are mapped to these types



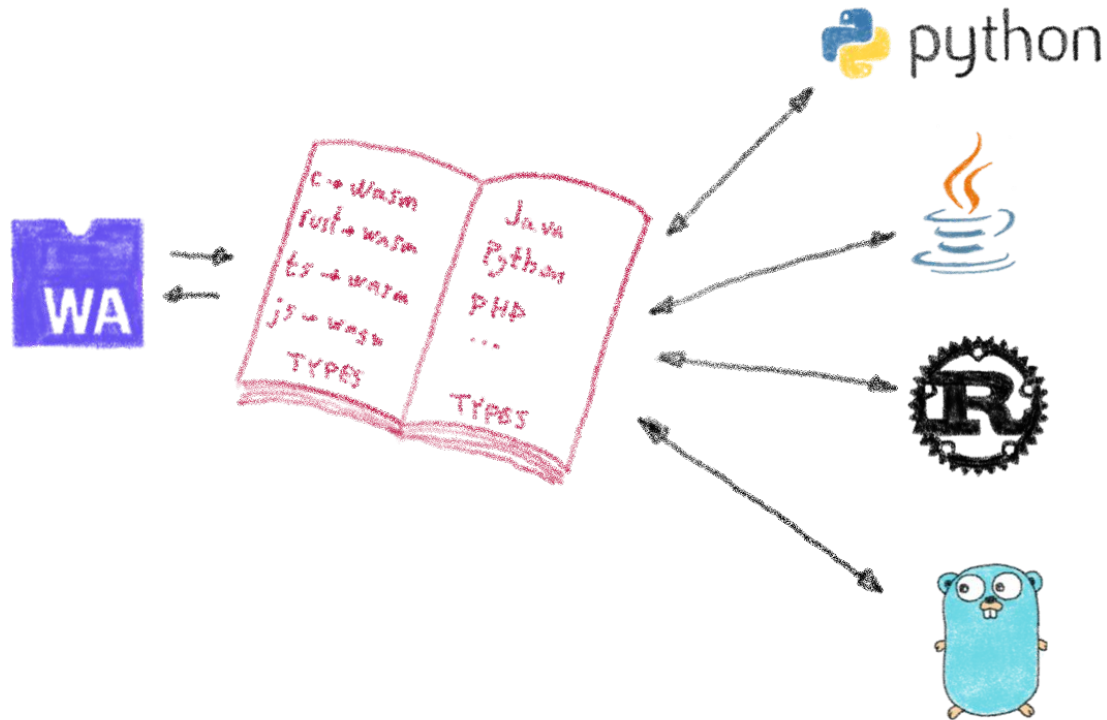
How can we share data?



Using the same data in WASM and JS?
Shared linear memory between them,
and serializing the data to one Wasm types

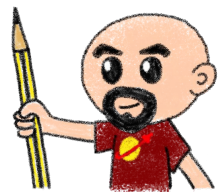


Solution is coming: Interface types



Beautiful description at:

<https://hacks.mozilla.org/2019/08/webassembly-interface-types>



No outside access



By design, communication is done using the shared linear memory only



Solution exists: WASI



WASI

WASI

The WebAssembly System Interface

WASI is a modular system interface for WebAssembly. As described in [the initial announcement](#), it's focused on security and portability.

WASI is being standardized in [a subgroup of the WebAssembly CG](#). Discussions happen in [GitHub issues](#), [pull requests](#), and [bi-weekly Zoom meetings](#).

For a quick intro to WASI, including getting started using it, see [the intro document](#).

The Wasmtime runtime's [tutorial](#) contains [examples](#) for how to target WASI from [C](#) and [Rust](#). The resulting .wasm modules can be run in any WASI-compliant runtime.

For more documentation, see [the documents guide](#).

Already available
in  Wasmer



Mono-thread and scalar operations only



Multiple scalar
operations

$$A1 + B1 = C1$$

$$A2 + B2 = C2$$

$$A3 + B3 = C3$$

Not the most efficient way...





Solution exists: SIMD

Multiple scalar operations

$$\begin{array}{l} A1 + B1 = C1 \\ A2 + B2 = C2 \\ A3 + B3 = C3 \end{array}$$

Single vectorial operation

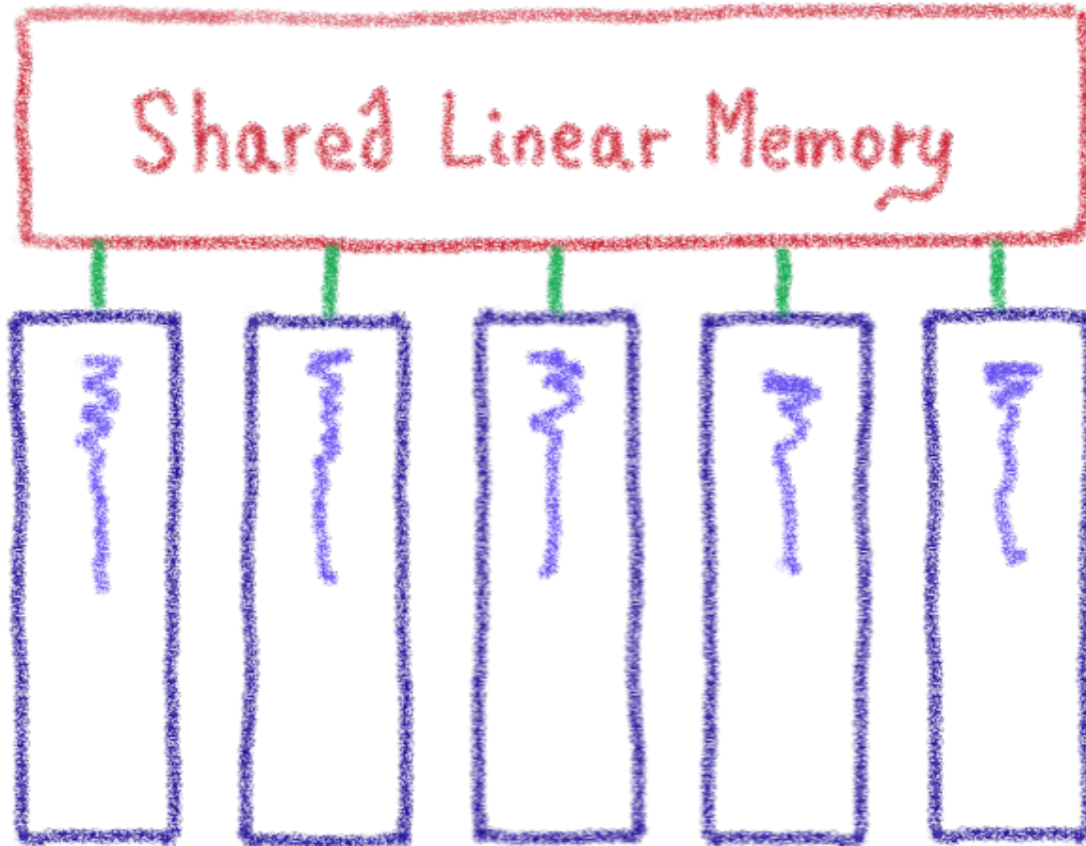
$$\begin{array}{l} A1 \\ A2 \\ A3 \end{array} + \begin{array}{l} B1 \\ B2 \\ B3 \end{array} = \begin{array}{l} C1 \\ C2 \\ C3 \end{array}$$

Single Instruction, Multiple Data

Already available
in  Wasmer



Solutions are coming too: Wasm Threads



Threads on Web Workers with shared linear memory



Incoming proposals: Garbage collector



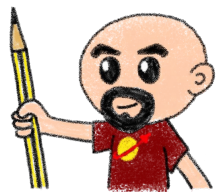
And exception handling





Server-side WebAssembly

Too good to not to use it



Solomon on Web Assembly



Solomon Hykes

@solomonstre




If WASM+WASI existed in 2008, we wouldn't have needed to create Docker. That's how important it is. Webassembly on the server is the future of computing. A standardized system interface was the missing link. Let's hope WASI is up to the task!



Lin Clark @linclark

WebAssembly running outside the web has a huge future. And that future gets one giant leap closer today with...

 Announcing WASI: A system interface for running WebAssembly outside the web (and inside it too)

hacks.mozilla.org/2019/03/standa...

9:39 PM · Mar 27, 2019



1.9K



792 people are Tweeting about this



Solomon on Web Assembly



Solomon Hykes

@solomonstre



“So will wasm replace Docker?” No, but imagine a future where Docker runs linux containers, windows containers and wasm containers side by side. Over time wasm might become the most popular container type. Docker will love them all equally, and run it all :)



Solomon Hykes @solomonstre

If WASM+WASI existed in 2008, we wouldn't have needed to created Docker. That's how important it is. Webassembly on the server is the future of computing. A standardized system interface was the missing link. Let's hope WASI is up to the task! [twitter.com/linclark/statu...](https://twitter.com/linclark/status...)

4:50 AM · Mar 28, 2019



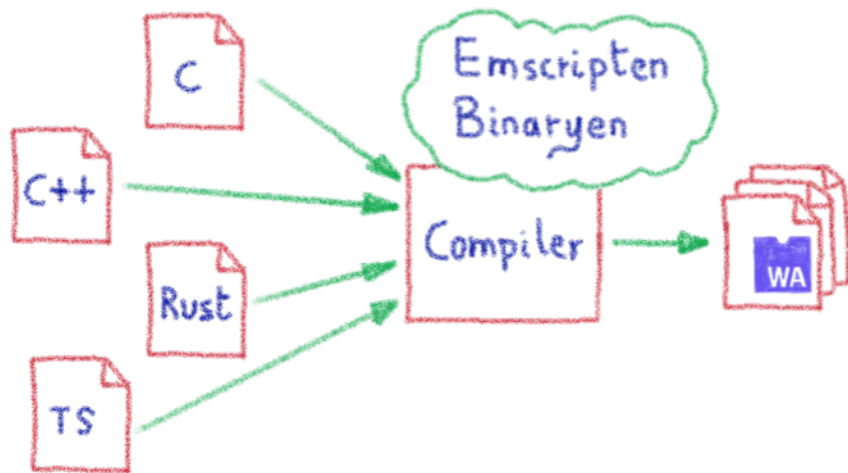
142



52 people are Tweeting about this



A very interesting feature set



Fast & Efficient ⚡

🔒 Memory-safe & Sandboxed

Open & Debuggable 📄

WWW Part of the Web Platform

WHY ONLY FOR THE WEB??





The Bytecode Alliance

Taking WASM out of the browser



The Bytecode Alliance



moz://a

fastly



Bytecode Alliance projects



Wasmtime

Cranelift

WebAssembly
Micro Runtime
(WAMR)

Lucet

Enarx



Wasmtime



wasmtime

A standalone runtime for [WebAssembly](#)

A [Bytecode Alliance](#) project

[Guide](#) | [Contributing](#) | [Website](#) | [Chat](#)

Installation

The Wasmtime CLI can be installed on Linux and macOS with a small install script:

```
$ curl https://wasmtime.dev/install.sh -sSf | bash
```

Windows or otherwise interested users can download installers and binaries directly from the [GitHub Releases](#) page.

Example

If you've got the [Rust compiler installed](#) then you can take some Rust source code:

```
fn main() {  
    println!("Hello, world!");  
}
```

and compile/run it with:

```
$ rustup target add wasm32-wasi  
$ rustc hello.rs --target wasm32-wasi  
$ wasmtime hello.wasm  
Hello, world!
```

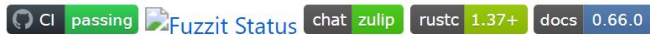




Cranelift Code Generator

A [Bytecode Alliance](#) project

Cranelift is a low-level retargetable code generator. It translates a [target-independent intermediate representation](#) into executable machine code.



For more information, see [the documentation](#).

For an example of how to use the JIT, see the [SimpleJIT Demo](#), which implements a toy language.

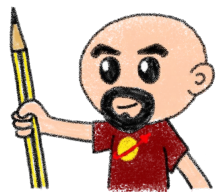
For an example of how to use Cranelift to run WebAssembly code, see [Wasmtime](#), which implements a standalone, embeddable, VM using Cranelift.

Status

Cranelift currently supports enough functionality to run a wide variety of programs, including all the functionality needed to execute WebAssembly MVP functions, although it needs to be used within an external WebAssembly embedding to be part of a complete WebAssembly implementation.

The x86-64 backend is currently the most complete and stable; other architectures are in various stages of development. Cranelift currently supports both the System V AMD64 ABI calling convention used on many platforms and the Windows x64 calling convention. The performance of code produced by Cranelift is not yet impressive, though we have plans to fix that.

The core codegen crates have minimal dependencies, support `no_std` mode (see below), and do not require any host floating-point support, and do not use callstack recursion.



WebAssembly Micro Runtime



WebAssembly Micro Runtime

[Build WAMR VM core](#) | [Embed WAMR](#) | [Export native function](#) | [Build WASM applications](#) | [Samples](#)

A [Bytecode Alliance](#) project

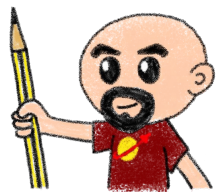
WebAssembly Micro Runtime (WAMR) is a standalone WebAssembly (WASM) runtime with a small footprint. It includes a few parts as below:

- The "iwasm" VM core, supporting WebAssembly interpreter, ahead of time compilation (AoT) and Just-in-Time compilation (JIT)
- The **application framework** and the supporting API's for the WASM applications
- The **dynamic management** of the WASM applications

iwasm VM core

key features

- 100% compliant to the W3C WASM MVP
- Small runtime binary size (85K for interpreter and 50K for AoT) and low memory usage
- Near to native speed by AoT
- Self-implemented module loader enables AoT working cross Linux, SGX and MCU systems
- Choices of WASM application libc support: the built-in libc subset for the embedded environment or [WASI](#) for standard libc
- [Embeddable with the supporting C API's](#)
- [The mechanism for exporting native API's to WASM applications](#)
- [Multiple modules as dependencies](#)
- [Thread management and pthread library](#)



Lucet



Lucet 

A [Bytecode Alliance](#) project

Lucet is a native WebAssembly compiler and runtime. It is designed to safely execute untrusted WebAssembly programs inside your application.

Check out our [announcement post on the Fastly blog](#).

Lucet uses, and is developed in collaboration with, the Bytecode Alliance's [Cranelift](#) code generator. It powers Fastly's [Terrarium](#) platform.

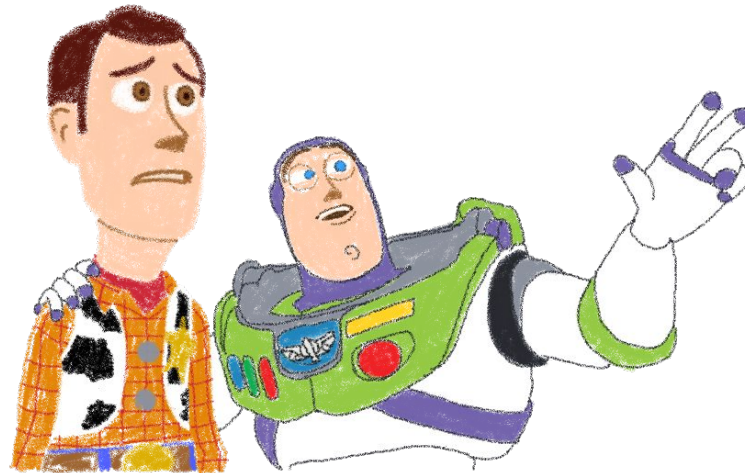
```
(func $_start (type 7)
  (local i32 i32 i32 i32)
  global.get 0
  i32.const 16
  i32.sub
  local.tee 0
  global.set 0
  call $__wasilibc_init_preopen
  i32.const 3
  local.set 1
  block ;; label = @1
    block ;; label = @2
      block ;; label = @3
        block ;; label = @4
          loop ;; label = @5
            local.get 1
            local.get 0
```



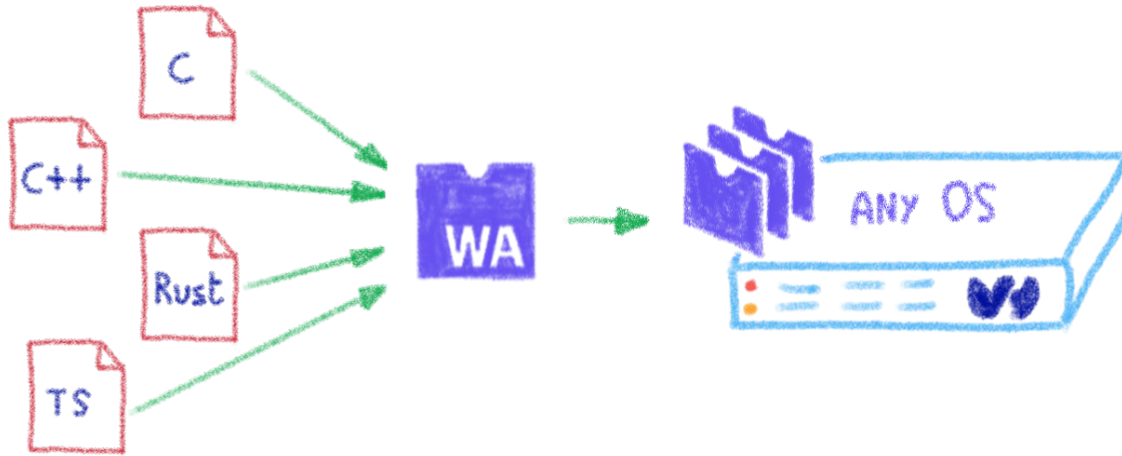


Other runtimes

Runtimes, runtimes everywhere



Wasmer



Run any code* on any client... almost

* Languages compiling to WASM





Wasmer

Run or embed anywhere super lightweight containers
based on Wasm

 FAST

 WASI

 Sandboxed

A1	B1	C1
A2	B2	C2
A3	B3	C3

 + = SIMD

Multi platform



 Pluggable

LLVM Cranelift
Singlepass


 @wasmerio


 wasmerio/wasmer





Wasmer ecosystem

 Wasmer ecosystem

 Wasmer.js



 Wasienv

Bringing languages to WASI





Wasm3



wasm v0.4.7 issues 30 tests failing license MIT

A high performance WebAssembly interpreter written in C.

- ~ 8x faster than other known wasm interpreters
- ~ 4-5x slower than state of the art wasm JIT engines
- ~ 12x slower than native execution

* Based on [CoreMark 1.0 benchmark](#). Your mileage may vary.

twitter 513 discord 6 online telegram chat

Status

wasm3 passes the [WebAssembly spec testsuite](#) and is able to run many WASI apps.

Minimum useful system requirements: ~64Kb for code and ~10Kb RAM

wasm3 runs on a wide range of architectures (x86 , x86_64 , ARM , RISC-V , PowerPC , MIPS , Xtensa , ARC32 , ...) and [platforms](#):

- Linux, Windows, OS X, FreeBSD
- Android, iOS
- OpenWRT-enabled routers
- Raspberry Pi, Orange Pi and other SBCs
- MCUs: Arduino, ESP8266, ESP32, Particle, ... [see full list](#)
- Browsers... yes, using WebAssembly itself!
- wasm3 can execute wasm3 (self-hosting)

```
Wasm3
Wasm3 v0.4.6 on iOS (arm64-v8a)
Build Feb  7 2020 21:48:53
Device: iPhone10,4

Loading WebAssembly...
Running fib(40) on WebAssembly...
Result: 102334155
Elapsed: 4844 ms

Running fib(40) on Native C...
Result: 102334155
Elapsed: 611 ms
```

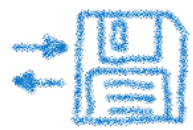


Wasm3



Wasm3

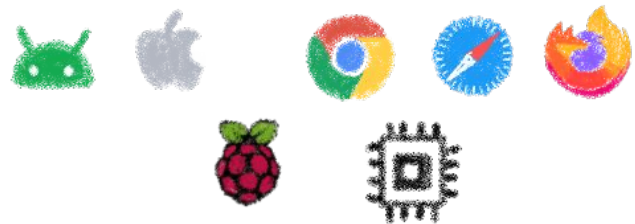
The fastest WebAssembly interpreter



WASI



Multi platform



Wasm inception

Web Assembly
& WASI
self-hosting



@wasm3_engine



wasm3/wasm3





And even Wasm over GraalVM!

GraalVM.

ALL ARTICLES | LEARN MORE

Announcing GraalWasm — a WebAssembly engine in GraalVM



Aleksandar Prokopec [Follow](#)

Dec 2, 2019 · 10 min read



We're happy to announce the initial public work on [GraalWasm](#) — the WebAssembly engine implemented in GraalVM. GraalWasm currently implements the WebAssembly MVP (Minimum Viable Product) specification, and can run WebAssembly programs in the binary format, generated with compiler backends such as Emscripten.

Supporting WebAssembly expands the set of languages GraalVM can execute with a whole other set of [languages](#) to the ones supported by GraalVM and is further step towards making it a universal platform for programming language execution. This feature was also [highly requested](#) by the GraalVM community and we are happy to share our first results.





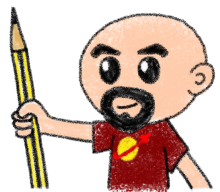
Some examples IRL?

Like companies using these things



clever cloud

fastly



Fastly works on the edge of the cloud

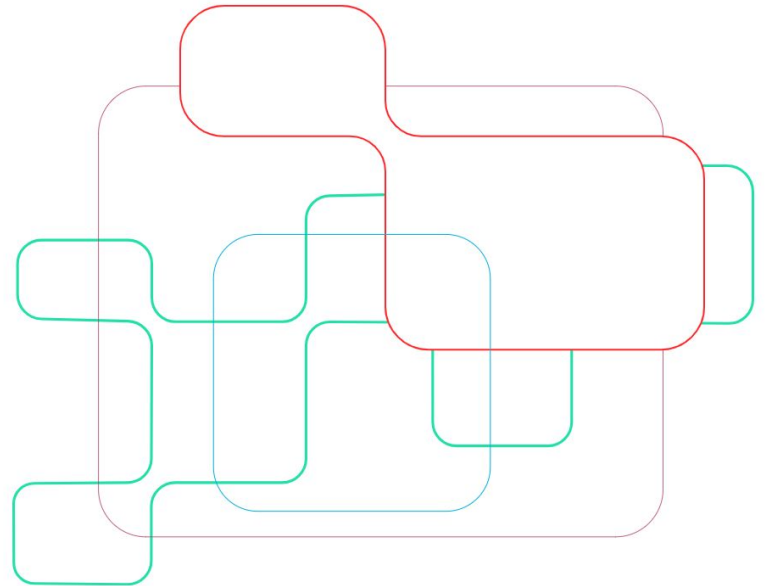


Edge compute technology

Give your developers an edge

Extend the power of your core cloud to the edge, and empower your teams to innovate. By moving data and applications as close to your end users as possible, you can deliver fast, highly personalized experiences to customers around the world. And now you can [join the private beta for Compute@Edge](#), our new serverless compute environment built to take you further at the edge.

Get a demo



fastly



Clever Cloud FaaS infrastructure



WEB ASSEMBLY (WASM)

WA

Made with ohmydeck! Oh!

BLEND WEB MIX ONLY LYON

37:13 / 51:30



clever cloud

<https://www.youtube.com/watch?v=wchehMIIsu80>



Clever Cloud FaaS Infrastructure



clever cloud

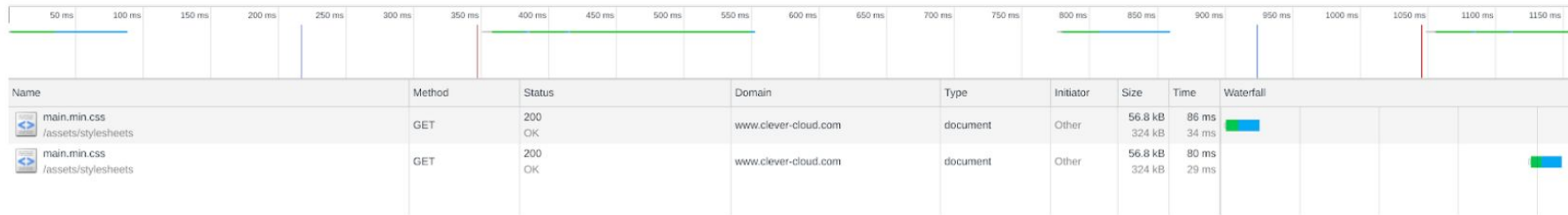
Already used in internal projects
Able to execute complex loads
(neural networks models)



Clever Cloud FaaS Infrastructure



le chargement directement depuis le site original



au travers du faas



on met 130ms à la première requête, parce qu'il faut boot la VM (9ms), la fork (1.5ms), puis établir la connexion TLS au backend



Blazing fast



Clever Cloud FaaS Infrastructure



le site complet:



le site complet au travers du faas:



en gros, 260ms de plus sur le DomContentLoaded, 300ms de plus sur le load sachant que mes VM s'exécutent sur un seul thread, j'ai pas parallélisé



Blazing fast

