WASM, Let's GO!

An Incomplete Guide to WebAssembly, the Go Programming Language, and Building WebAssembly Apps with TinyGo

Tommy TIAN
tommy.tian@credit-suisse.com
GitHub: txaty

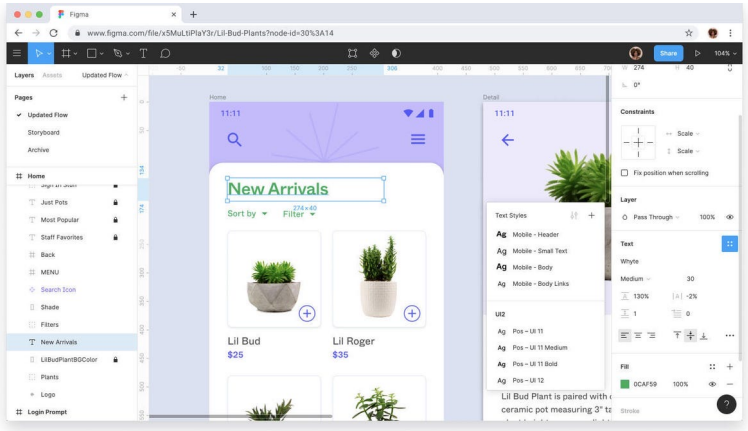
Case Study: Figma

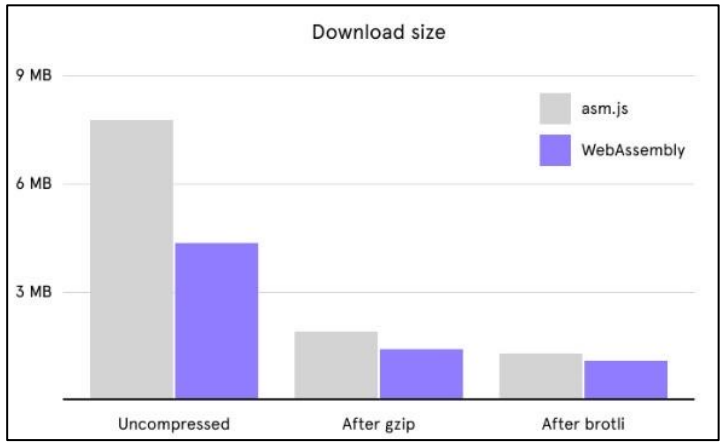
Figma is a collaborative and web-based design and prototyping tool used to create user interfaces, mobile app designs, web pages, and other digital products.

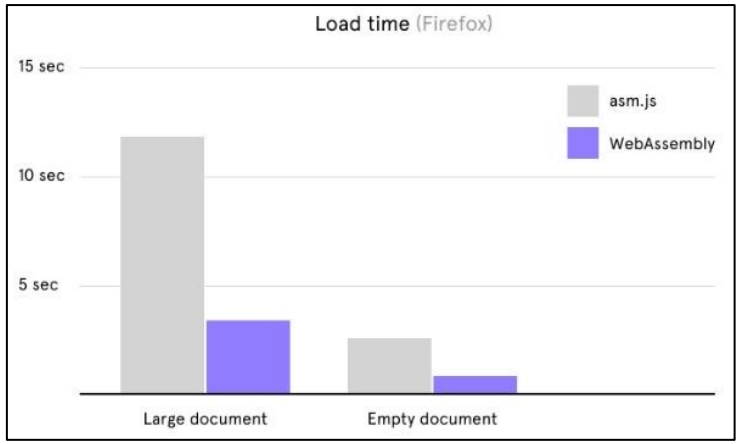
Before → 3x faster → **After**

asm.js → *WebAssembly*

Figma's codebase is in C++. Before WebAssembly, C++ code could be run in the browser by cross-compiling it to a subset of JavaScript known as asm.js. The asm.js subset is basically JavaScript where you can only use numbers (no strings, objects, etc.).

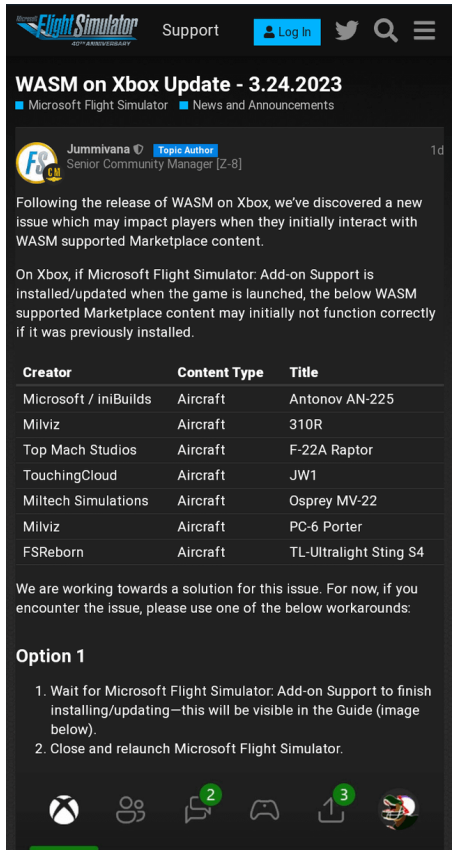






Case Study: Microsoft Flight Simulator

MSFS releases new add-on to prepare for WASM support on Xbox.



Microsoft Flight Simulator Support Log In

WASM on Xbox Update - 3.24.2023

Microsoft Flight Simulator ■ News and Announcements

Jummivana Topic Author
Senior Community Manager [Z-8] 1d

Following the release of WASM on Xbox, we've discovered a new issue which may impact players when they initially interact with WASM supported Marketplace content.

On Xbox, if Microsoft Flight Simulator: Add-on Support is installed/updated when the game is launched, the below WASM supported Marketplace content may initially not function correctly if it was previously installed.

Creator	Content Type	Title
Microsoft / iniBuilds	Aircraft	Antonov AN-225
Milviz	Aircraft	310R
Top Mach Studios	Aircraft	F-22A Raptor
TouchingCloud	Aircraft	JW1
Miltech Simulations	Aircraft	Osprey MV-22
Milviz	Aircraft	PC-6 Porter
FSReborn	Aircraft	TL-Ultralight Sting S4

We are working towards a solution for this issue. For now, if you encounter the issue, please use one of the below workarounds:

Option 1

1. Wait for Microsoft Flight Simulator: Add-on Support to finish installing/updating—this will be visible in the Guide (image below).
2. Close and relaunch Microsoft Flight Simulator.

WebAssembly

- Safe, portable, low-level code format designed for efficient and compact representation.
- A virtual Instruction Set Architecture (ISA)[1] for a virtual stack machine.
- Applications: video and audio codecs, graphics and 3D, multi-media and games, cryptographic computations, portable language implementations, etc.
- The fourth official language of the web, backed by World Wide Web Consortium (W3C).

HTML

CSS

JavaScript

WebAssembly

[1] Generally, an ISA is a binary format designed to execute on a specific machine.

Portable

- WebAssembly is an assembly language for a conceptual machine, not a physical one. It needs a runtime on top of a real machine to execute.
- It makes no architectural assumptions that are not broadly supported across modern hardware.
- It runs on: all major web browers, V8 runtimes like Node.js, and independent WASM runtime like Wasmtime, Lucet, and Wasmer.

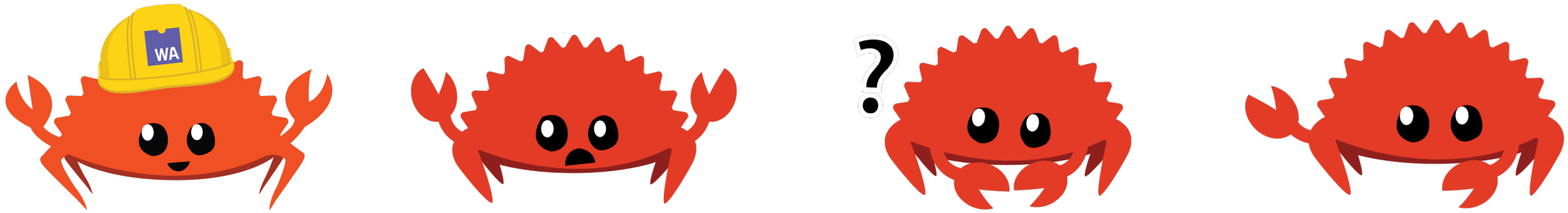
	Your browser	Chrome	Firefox	Safari	Wasmtime	Wasmer	Node.js	Deno	wasm2c
Standardized features									
JS BigInt to Wasm i64 integration	✓	85	78	14.1[d]	N/A	N/A	15.0	1.1.2	N/A
Bulk memory operations	✓	75	79	15	0.20	1.0	12.5	0.4	1.0.30
Extended constant expressions	✓	114	⚑[c]	×	×	×	⚑[g]	⚑[l]	⚑[r]
Multi-value	✓	85	78	✓	0.17	1.0	15.0	1.3.2	1.0.24
Mutable globals	✓	74	61	✓	✓	0.7	12.0	0.1	1.0.1
Reference types	✓	96	79	15	0.20	2.0	17.2	1.16	1.0.31
Non-trapping float-to-int conversions	✓	75	64	15	✓	✓	12.5	0.4	1.0.24
Sign-extension operations	✓	74	62	14.1[d]	✓	✓	12.0	0.1	1.0.24
Fixed-width SIMD	✓	91	89	16.4	0.33	2.0	16.4	1.9	1.0.33
Tail calls	✓	112	×	×	×	×	⚑[j]	⚑[o]	×
In-progress proposals									
Exception handling	✓	95	100	15.2	×	×	17.0	1.16	⚑[q]
Garbage collection	×	⚑[a]	×	×	×	×	×	×	×
Memory64	×	⚑[b]	⚑[c]	×	⚑[e]	×	⚑[h]	⚑[m]	⚑[s]
Multiple memories	?	×	×	×	⚑[f]	×	×	×	⚑[t]
Relaxed SIMD	✓	⚑[b]	⚑[c]	×	×	×	⚑[i]	⚑[n]	×
Threads and atomics	✓	74	79	14.1[d]	N/A	N/A	16.4	1.9	×
Type reflection	?	⚑[b]	⚑[c]	×	×	2.0	⚑[k]	⚑[p]	×

From https://webassembly.org/roadmap/

Safety

WASM code is validated and executed in a memory-safe, sandboxed environment preventing data corruption or security breaches.

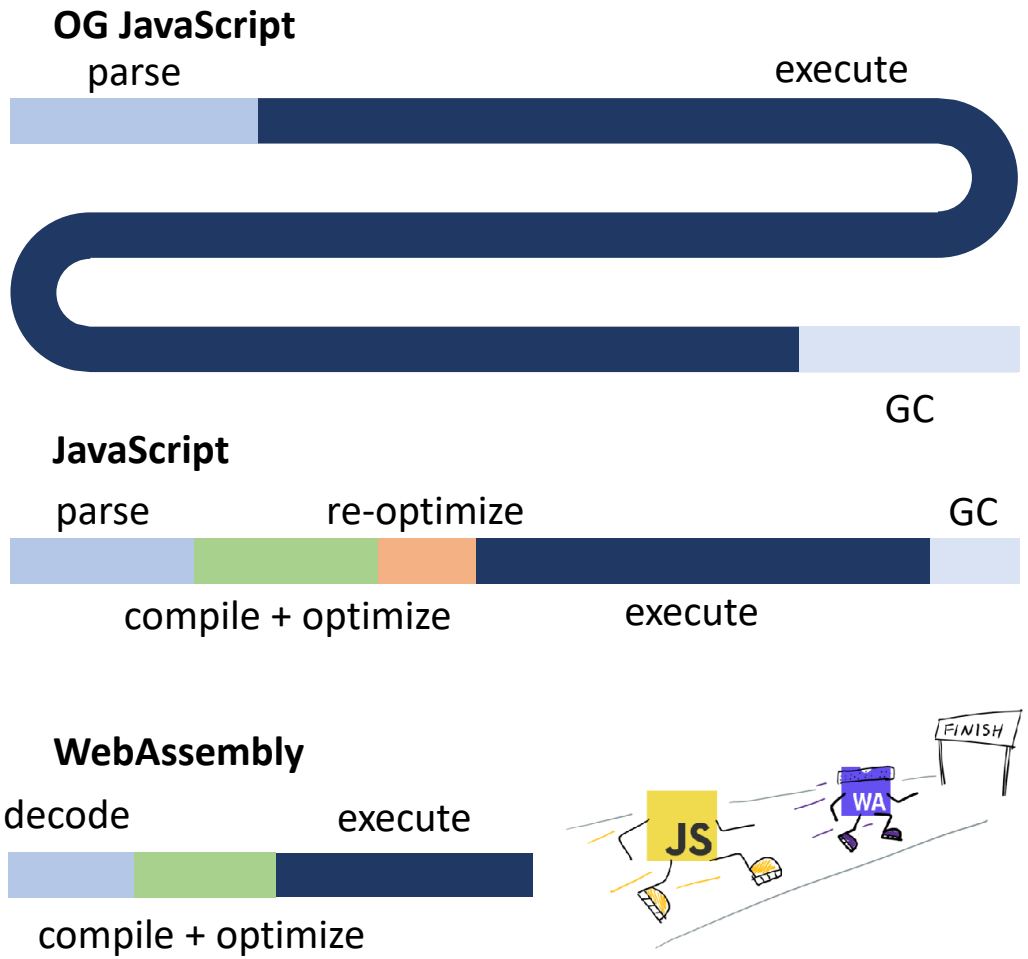
- **Memory safety**: WebAssembly programs have a linear memory that can't be accessed outside of its bounds.
- **Undefined behaviour**: is impossible thanks to the well-defined semantics of WebAssembly.
- **Code injection**: is literally impossible. All the functions used will needed to be declared at load time, and no functions can be generated at runtime (this also makes impossible having JITs in pure WASM).
- **Host functions**: by default, WebAssembly have no access to the host.



Fast

Executes with near native code performance, taking advantage of capabilities common to all contemporary hardware.

- **Compact**: has a binary format that is fast to transmit by being smaller than typical text or native code formats.
- **Efficient**: can be decoded, validated, and compiled in a fast single pass.
- **Streamable**: allows decoding, validation, and compilation to begin ASAP, before all data has been sent.
- **Parallelization**: decoding, validating, and compilation can be split into many independent parallel tasks.

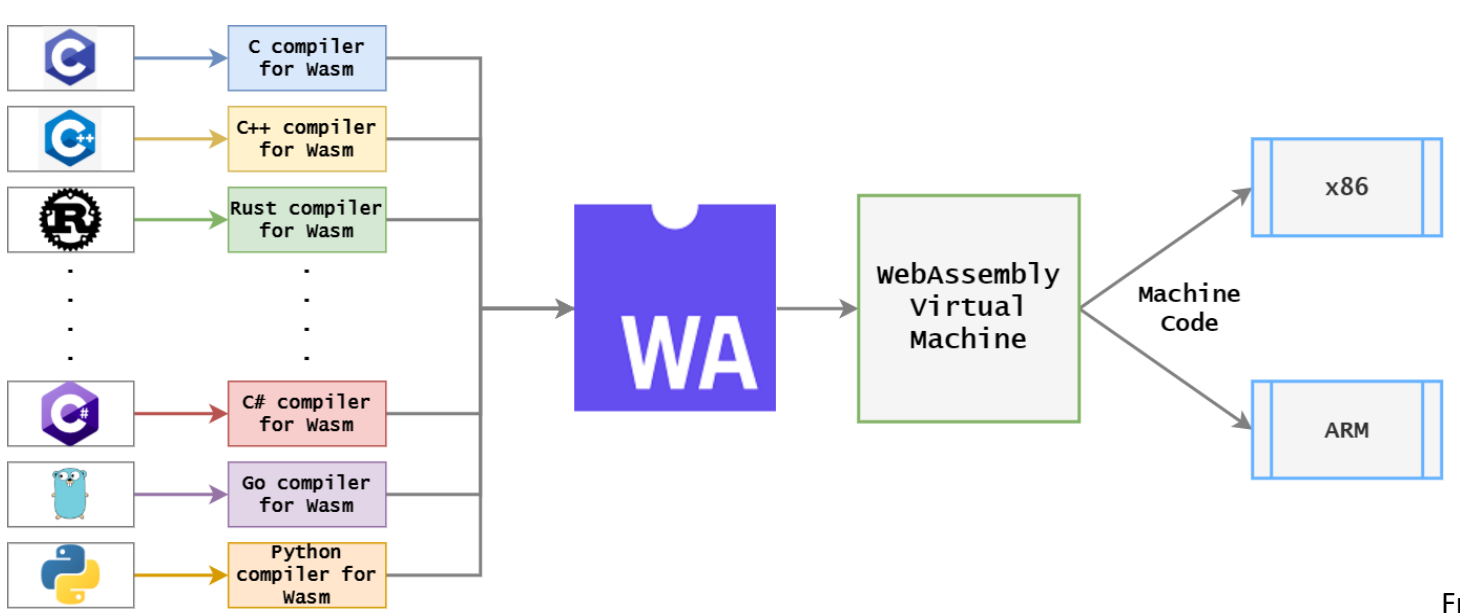


From A Cartoon Intro to WebAssembly by Lin Clark

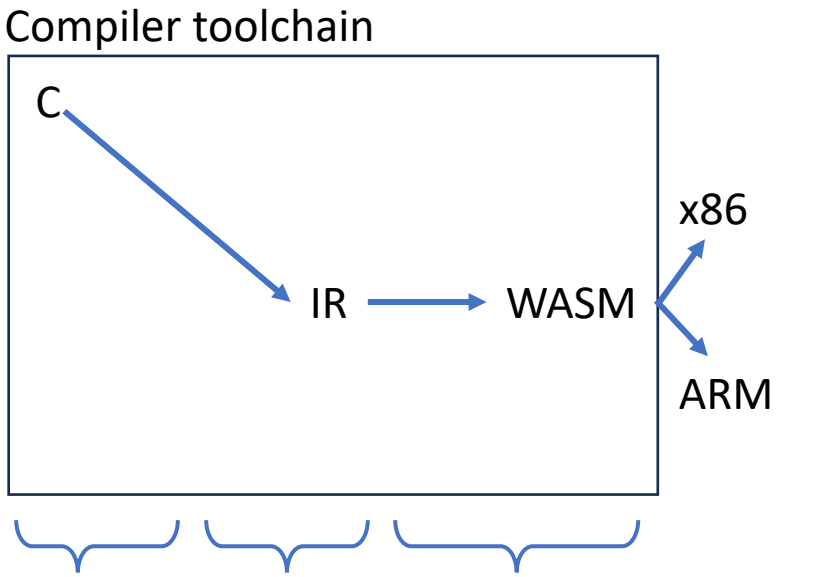
Language-independent

WASM does not privilege any particular language, programming model, or object model.

Developers can write WASM apps in Java, Python, Rust, Golang or C++ so long as they have a compiler to convert the source code into WebAssembly-compatible binaries.



From WebAssembly - What it is & Why is it so important by Arghya C



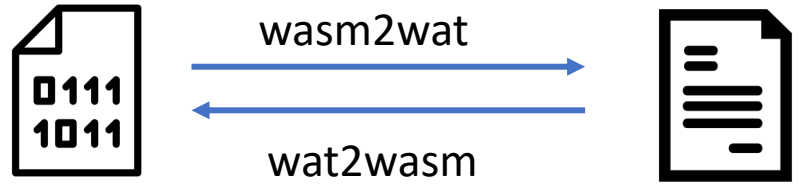
From A Cartoon Intro to WebAssembly by Lin Clark

WAT

WebAssembly Text (WAT), is the textual representation of the WASM binary format.

It act as the assembly language for WebAssembly.

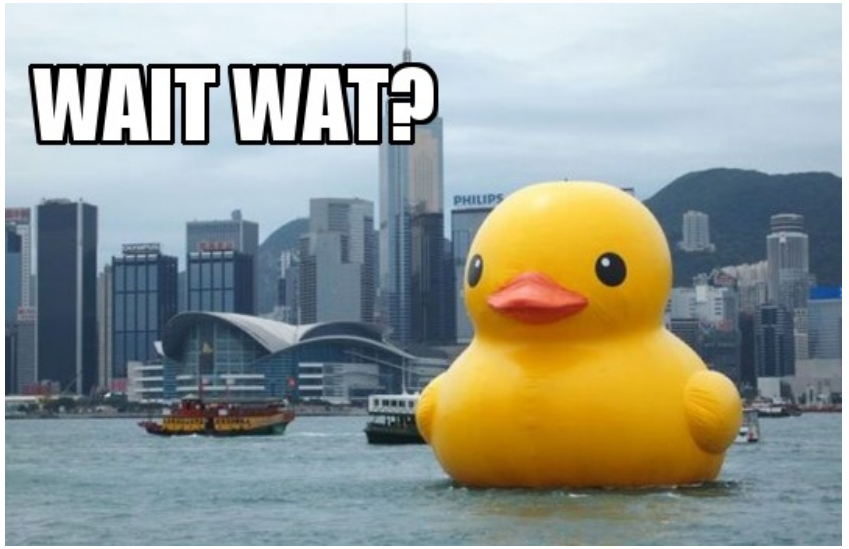
It uses S-Expressions, a nested tree structure coding style used in programming languages such as Lisp.



WASM Binary WAT File

```
(module
  (import "env" "print_string" (func $print_string (param i32)))
  (import "env" "buffer" (memory 1))
  (global $start_string (import "env" "start_string") i32)
  (global $string_len i32 (i32.const 12))
  (data (global.get $start_string) "hello world!")
    (func (export "helloworld")
      (call $print_string (global.get $string_len))
  )
)
```

Hello World in WAT



WebAssembly Everywhere

WebAssembly is **platform-independent**. It can be embedded in browsers, run as a stand-alone VM, or integrated in other environments.

WASM – WebAssembly
A a binary instruction format that provides a build target for a conceptual machine.

WASI – Web Assembly System Interface
A modular system interface for WebAssembly which provides a conceptual operating system.

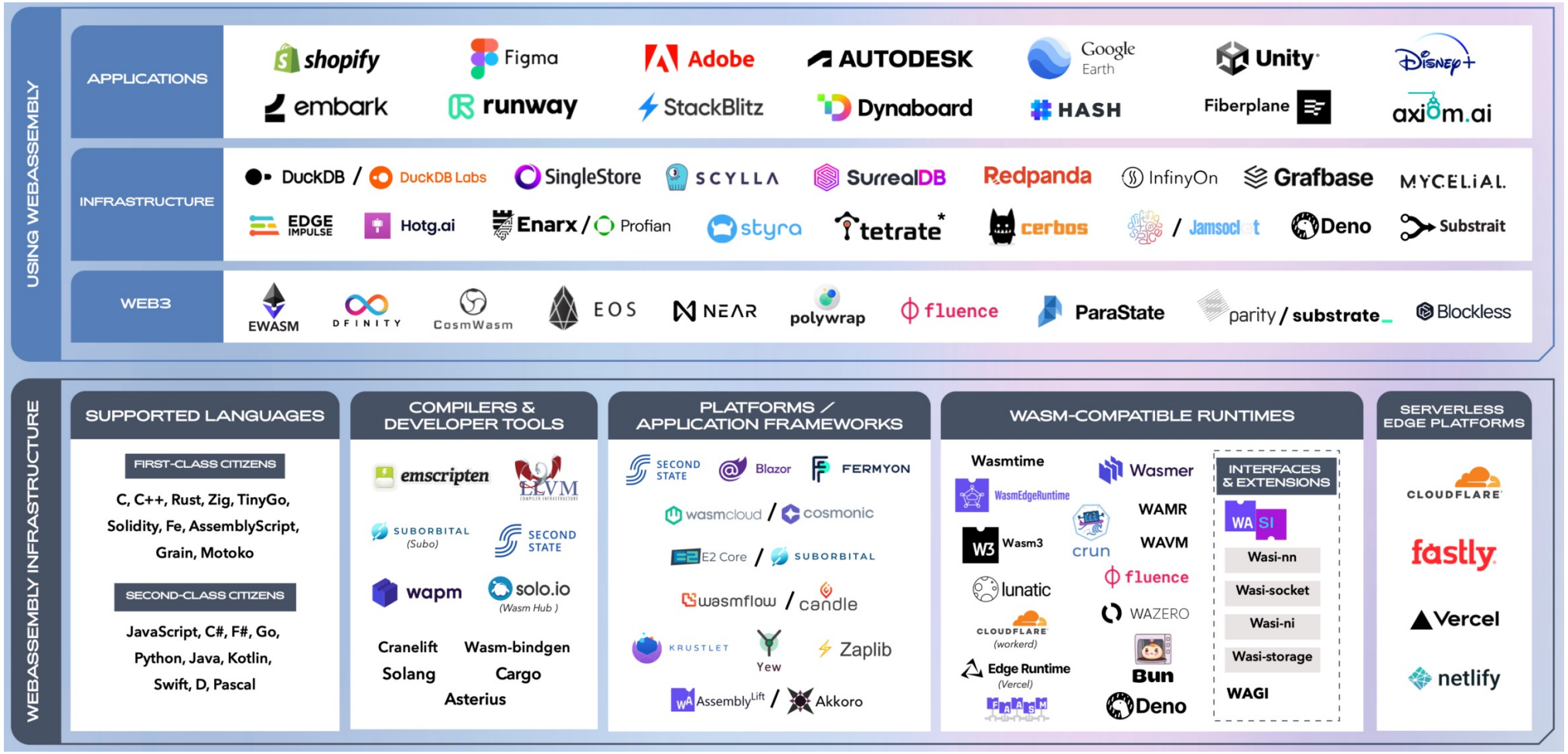
WAGI – Web Assembly Gateway Interface
Like CGI, it can dynamically load WASM code, providing a framework for calling, loading and executing modules.

Smart Contract VMs

ewasm – Ethereum WebAssembly, Ethereum smart contract execution layer using a deterministic subset of WASM

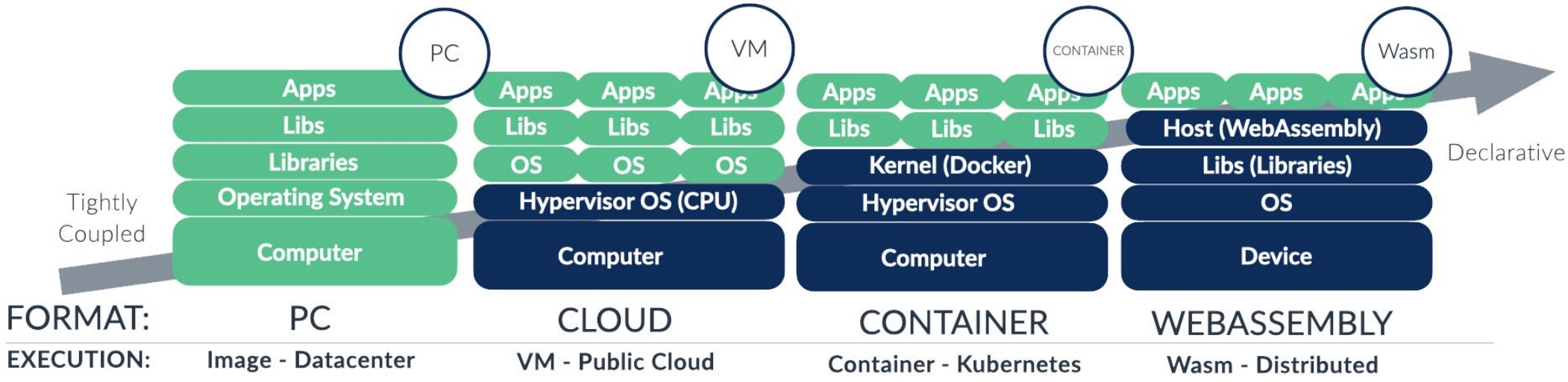
CosmWasm – smart contract platform built for the Cosmos ecosystem.

WASM Application & Infrastructure Landscape



From SAPPHIRE blog post: What's Up With WebAssembly: Compute's Next Paradigm Shift

WASM Ambitions



Developer	Full	OS, App, Lib	App, Lib	Wasm
Shared	-	CPU	Linux Kernel	-
Compatability	All	Most	Most	Most
Size	Large	Med	Small	Tiny
Portability	-	Low	Med (CPU, Linux)	High
Security	System	OS	Process Boundary	Capability
Where		Proprietary Cloud & Edge	Dev, Edge, Cloud, K8s	Dev, App, Edge, Cloud, K8s, Browser, Devices

Legend: Developer Provided / Device / Computer

From Kubecon EU 2021 Keynote: WebAssembly & Cloud Native: Better Together

The Go Programming Language

Go is a statically typed, compiled high-level programming language with memory safety, garbage collection, structural typing, and CSP-style concurrency. It was designed by Google in 2007.

Why it is designed:

- Readability, Expressiveness, and Maintainability: Ensuring Long-term Durability.
- Maximizing Agility and Reducing Time to Market.
- Concurrency: Designed for Modern Computing.

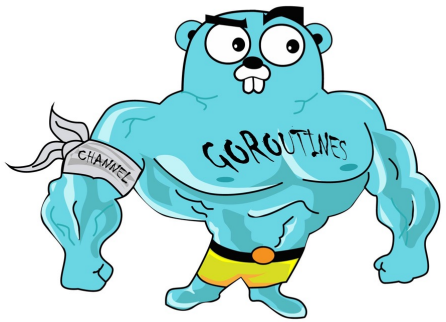
Essential characteristics:

- **Stability**: Dependable for ongoing projects despite regular updates.
- **Expressivity**: Minimalist design ideal for extensive codebases.
- **Efficient Compilation**: Rapid compilation enhances developer productivity.
- **Safety**: Strong, static typing reduces runtime errors, boosting reliability.

Go vs. TinyGo

TinyGo: A compact Go compiler tailored for microcontrollers, WebAssembly, and command-line tools.

It reuses libraries used by the Go language tools alongside LLVM to provide an alternative way to compile programs written in the Go programming language.



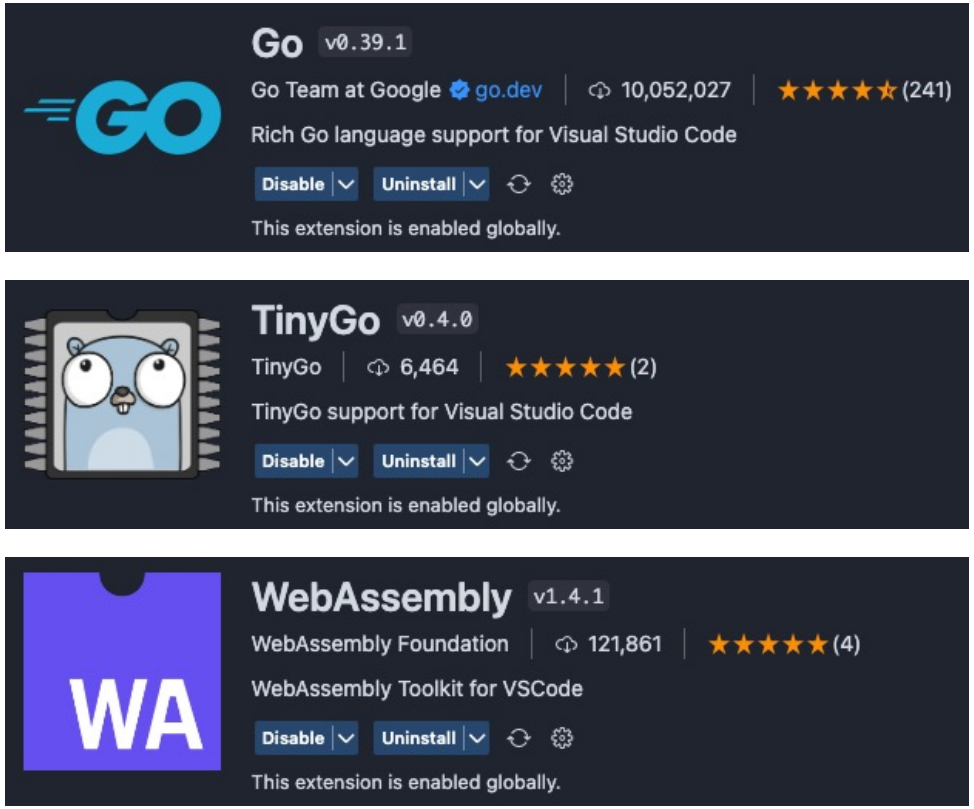
Feature	Go	TinyGo
Footprint	Larger binary size	Smaller binary size, ideal for small places
Concurrency	Full support with goroutines	Limited support
Standard Library	Comprehensive	Limited subset due to size constraints
Compiler	Use its own compiler	Uses LLVM compiler infrastructure
Garbage Collection	Complex, resource-intensive	Simpler and less resource-intensive
Interoperability	Full support for C with cgo	Limited support for cgo features

WASM with TinyGo: Prerequisites

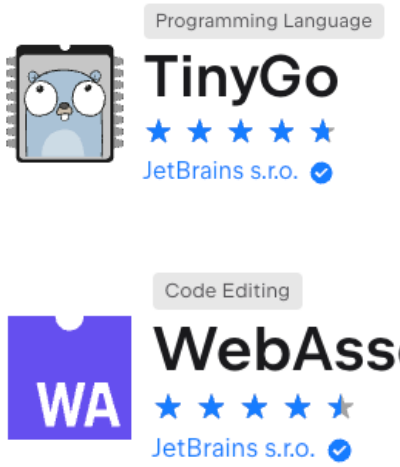
Install Go: https://go.dev/doc/install, and TinyGo: https://tinygo.org/getting-started/install

Plugins:

On VS Code



On JetBrains Goland



WASM with TinyGo: Go File

Write main.go:

```go
package main

// We don't use the main function but we still need it here.
func main() {}

// Add `export add` flag to export this function,
// so that it is callable from JavaScript.
//export add
func add(x int, y int) int {
  return x + y;
}
```

- Export the functions with export flag, so that they are callable from JavaScript.
- Sometimes we need to keep application alive after the WASM module instantiation. We can achieve this by creating a channel and reading from it which will block the application from exiting.

Compile to WASM:

TinyGo:

```
tinygo build -o main.wasm -target wasm ./main.go
```

Go:

```
GOOS=js GOARCH=wasm go build -o main.wasm
```

To shrink the size of the compiled binary, we can use ldflags with two additional flags:

- -w: turn off DWARF debugging info
- -s: turn off generation of Go Symbol table

```
GOOS=js GOARCH=wasm go build -ldflags="-s -w" -o main.wasm
```

WASM with TinyGo: HTML File

Write index.html:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello World - Go</title>
  </head>
  <body>
    <script src="./wasm_exec.js"></script>
    <script type="module" src="./index.js"></script>
  </body>
</html>
```

Copy WASM wrapper:

Both TinyGo and Go provide implement their own wrappers wasm_exec.js for WebAssembly.

TinyGo:

```
cp $(tinygo env TINYGOROOT)/targets/wasm_exec.js .
```

Go:

```
cp "$(go env GOROOT)/misc/wasm/wasm_exec.js" .
```

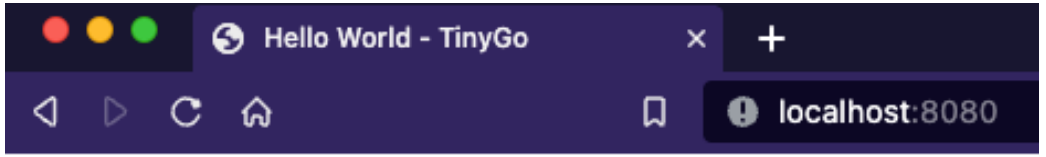
The wasm_exec.js must come before our Javascript (index.js), as it defines some global objects that drives Go WASM functionalities.

WASM with TinyGo: JS File

```
const go = new Go(); // Defined in wasm_exec.js.
const WASM_URL = 'main.wasm';

let importObject = go.importObject;

const init = async () => {
    let response;
    if (!importObject) {
        importObject = {
            env: {
                abort: () => console.log("Abort!")
            }
        };
    }
    if (WebAssembly.instantiateStreaming) {
        response = await WebAssembly.instantiateStreaming(
            fetch(WASM_URL),
            importObject
        );
    } else {
        const fetchAndInstantiateTask = async () => {
            const wasmArrayBuffer = await fetch(WASM_URL).then(response =>
                response.arrayBuffer()
            );
            return WebAssembly.instantiate(wasmArrayBuffer, importObject);
        };
        response = await fetchAndInstantiateTask();
    }
    return response;
};

const wasmModule = await init();

// Call the Add function export from wasm, save the result
const addResult = wasmModule.instance.exports.add(24, 24);

// Set the result onto the body
document.body.textContent = `Hello World! addResult: ${addResult}`;
```

1. Loading WASM moduels with WebAssembly Web APIs.
2. Use the exported function from the WASM module.

Expected Result:



Hello World! addResult: 48

More examples:

https://github.com/txaty/wasm-lets-go

WASM Optimization

Twiggy

Twiggy is a code size profiler for Wasm.
It analyzes a binary's call graph.

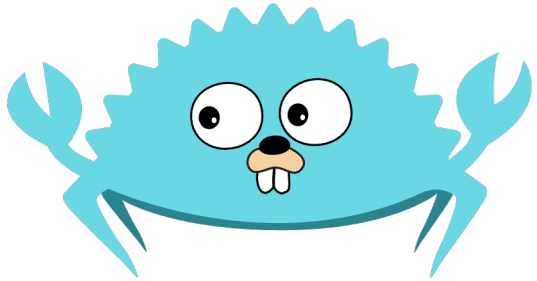
```
tommytian$ twiggy top -n 10 main.wasm
 Shallow Bytes | Shallow % | Item
---------------+-----------+-----------------------------
         92874 |     7.05% | data[15417]
         32843 |     2.49% | data[15415]
         21790 |     1.65% | data[635]
         21310 |     1.62% | data[611]
         15865 |     1.20% | code[781]
         12561 |     0.95% | data[634]
          9607 |     0.73% | code[816]
          9322 |     0.71% | code[230]
          9020 |     0.68% | code[693]
          8797 |     0.67% | code[333]
       1083210 |    82.24% | ... and 16906 more.
       1317199 |   100.00% | Σ [16916 Total Rows]
```

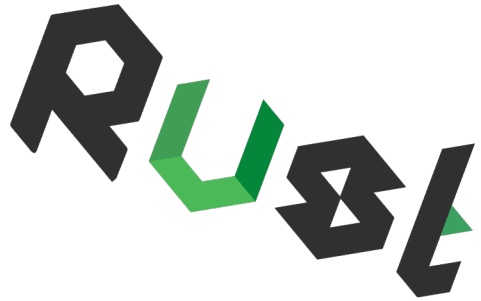
wasm-opt

A tool in the binaryen toolkit which is a wasm-to-wasm transformation that optimizes the input WASM module.


REACTJS


JavaScript




RUBY


Vscode

Q & A


ubuntu


GitHub


RubyOnRails