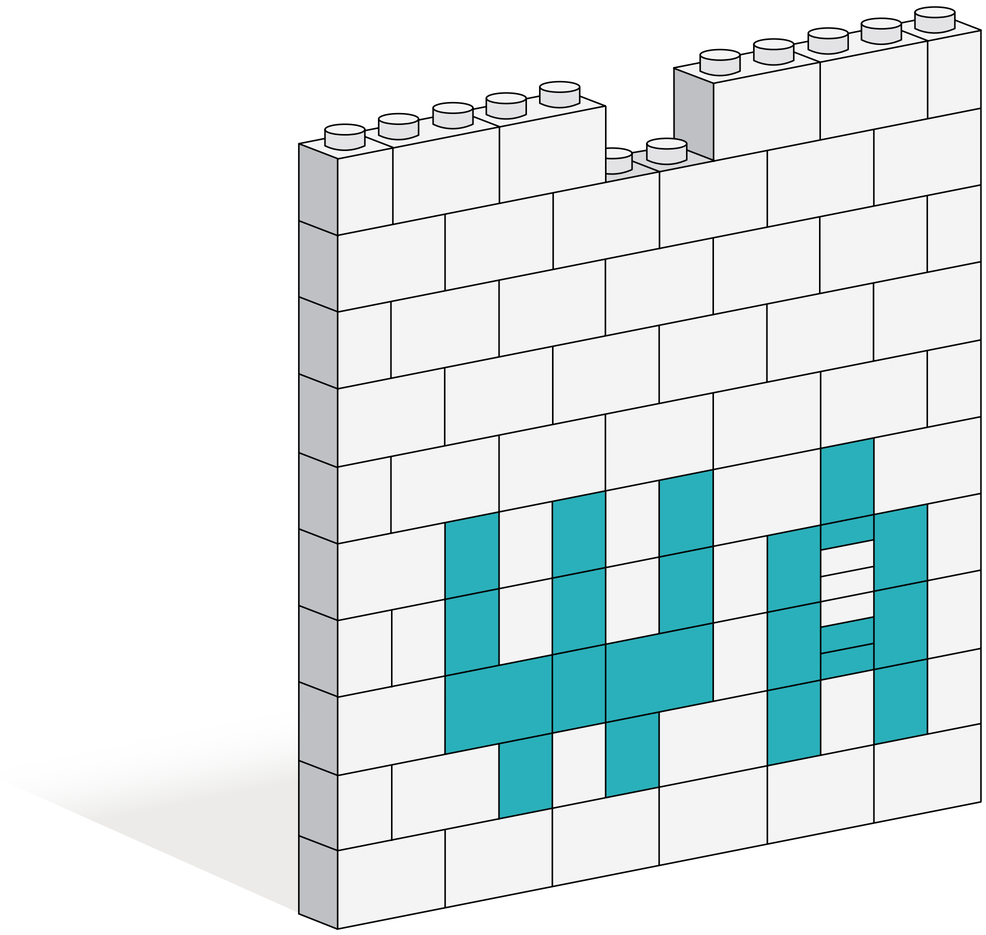
The Web Assembles

WebAssembly and the future of the web

WebAssembly is a new runtime for the web; a fast and efficient compilation target for a wide range of languages that could have a far-reaching impact on the web as we know it. This paper looks at the performance limits of JavaScript and how WebAssembly was designed to tackle them. We then consider the impact of WebAssembly on JavaScript and the wider web platform.

A white paper by Chris Price & Colin Eberhardt



SCOTT LOGIC

ALTOGETHER SMARTER

Contents

Why we need WebAssembly

To understand the rationale behind WebAssembly, and why it works the way it does, it helps to remember just how far JavaScript has come.

"JavaScript was conceived almost a quarter of a century ago when the web was a very different place. A scattered landscape of static documents and simple forms. The gleaming Java applet skyscrapers only just visible on the horizon. Into this world, a modest scripting language was born, with humble dreams of… form validation.

Jump forward 20 years, beyond the bruising browser wars, past the painful plugin wars and hiding behind the hype of HTML5, the web is a very different place. JavaScript can no longer be described as the light dusting of icing sugar on the cake. It's more accurately described as the cake. And the plate the cake is sat on. And the person who carried the plate out.

And the pastry chef who made the cake, the kitchen housing the chef, the building containing the kitchen, the fabric of the building, the design of the building, the construction of the building, the city in which the building was built, the city's critical infrastructure… the list goes on*."

** Don't believe us? Take a look in your node_modules folder…*

So where did all of this JavaScript come from? And why did everyone suddenly decide to use it?

The short answer is that, in many cases, they didn't. For a long time developers adopted Flex (Flash), or to a lesser extent Silverlight or Java (FX), in a bid to paper over their perceived shortcomings of the language, its runtime performance or the functionality gaps in the browsers of the day. However, over time, plugins encountered their own issues - with headline-grabbing concerns about their security and availability.

This was brought to a head when plugins started suffering platform hostility, most notably on iOS. This, coupled with ever-improving browser performance, an ever-growing feature set and significantly improved tooling, meant that for many people JavaScript rapidly became "good enough".

JavaScript is now the only language that can honestly boast the "learn once, use anywhere" mantra popularised in various forms by other contenders in the past. Interestingly, whilst this is true and despite it being an interpreted language, it's actually very rare to find developers directly authoring their production code in it.

The rapid innovation in JavaScript has continued, to the extent that most developers use build toolchains ranging from simple stylistic checks, polyfills and code minification (e.g. Webpack), through transpilation of individual features from future versions of JavaScript (e.g. Babel) or JavaScript-like languages (e.g. TypeScript).

So, if JavaScript is itself now a valid compiler target, why do we need WebAssembly?

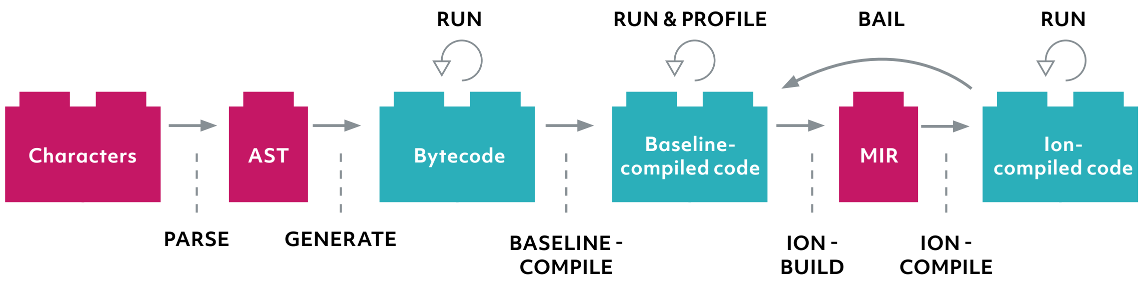
The unbridgeable performance gap

While JavaScript is increasingly being used as a compilation target, it is unfortunately not a very good one. To understand why, we need to consider some of the technical aspects of how the browser's runtime has evolved.

In the very early days JavaScript execution was slow. Browsers used simple interpreters to execute relatively simple scripts. However, as the hype around HTML5 heated up, competition between browser vendors resulted in considerable advances in JavaScript performance. Most of this could be attributed to the introduction of Just-In-Time (JIT) compilation and increasingly sophisticated runtimes.

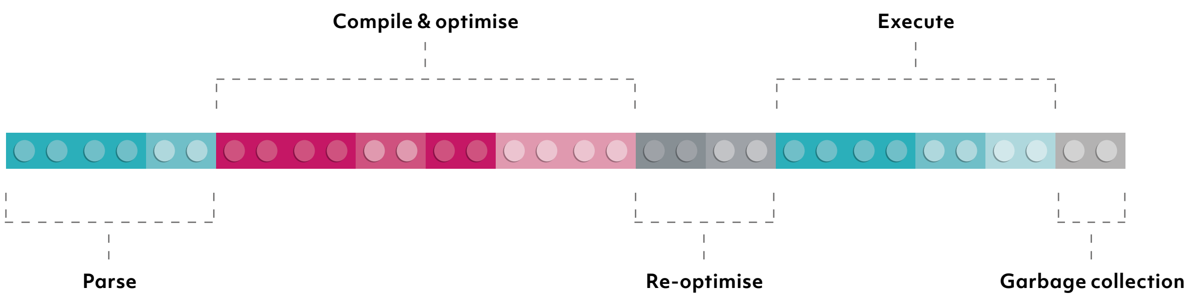
The diagram below gives an example of how JavaScript is executed within a modern browser, with arrows representing algorithms, and blocks representing data structures.

JavaScript is delivered to the browser as text over HTTP. This text is parsed into an Abstract Syntax Tree (AST), which is used to generate bytecode that is initially run by an interpreter.



The JIT profiles this running code and, based on certain assumptions, compiles it into a more efficient form. Through continuous profiling, the browser is able to further optimise, and re-compile sections of the code, resulting in yet-faster execution.

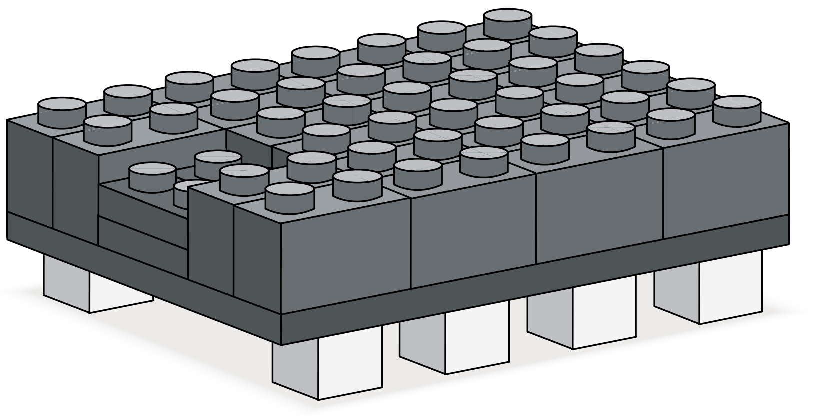
However, at any time, the assumptions that were made can be violated, resulting in the optimised code being rejected, and a less-optimal version being used instead.



This complex runtime, which consists of multiple tiers of execution (Chrome's V8 has two tiers, while Safari's Nitro has three), can achieve near-native performance under optimal conditions, but this level of performance isn't immediately available. There is an inherent start-up cost, as illustrated above.

"Time to Interactive" - the time it takes to load a web page (or app), render the HTML, parse and execute the JavaScript so that an end user can interact - has become a very important metric, especially on mobile devices. Further innovations in browser technology such as code-caching and script streaming help reduce this time, and there are wider initiatives, such as the HTTP/2 standard, that optimise how resources are downloaded by the browser.

Despite all these innovations, we are fundamentally limited by the inefficient way in which our code is sent to the browser. Wouldn't it be better if our build tools could deliver code as a pre-optimised version in a compact form?



Introducing WebAssembly

What is WebAssembly?

The official site has the following to say:

" WebAssembly or wasm is a new portable, size - and load-time - efficient format suitable for compilation to the web. "

Paraphrasing, the standard aims to continue to offer the same ubiquitous platform as JavaScript, as well as offering the same security guarantees, allowing the same code to run safely on a wide variety of devices. Additionally, it aims to go beyond JavaScript performance with a smaller binary format optimised for parallel parsing.

On a technical level, WebAssembly co-exists with JavaScript in a common runtime. They can share memory and fully interoperate, with WebAssembly calling JavaScript and vice versa. This means that, in the long term, they will continue to evolve together, and in the short term JavaScript features will be made available to WebAssembly (e.g. garbage collection or DOM access), so we'll see WebAssembly features being made available to JavaScript (e.g. 64-bit integers or threads).

Where can I use WebAssembly?

All of the major JavaScript platform developers (Apple, Microsoft, Google, Mozilla, Node.js) have come together to form a W3C working group and have shipped working MVP implementations of WebAssembly in their respective platforms.This level of engagement for a new format is unprecedented across any previous attempts.

The key to this collective understanding of the problem was asm.js: an attempt to attain the same low-level performance requirements by utilising only an optimisable subset of JavaScript. This had the advantage of running in all existing JavaScript environments but also opening the door to asm.js-specific engine optimisations. It also introduced novel ideas to solve the problems of portable code, such as using the newly introduced ArrayBuffer for memory access rather than buffer-overrun-prone pointers.

Whilst this demonstrated measurable performance improvements, it also highlighted the fundamental limits of the language.

With empirical performance evidence available, and the lessons learned in its implementation, the web community were - for the first time - able to forge a consensus on the need for a new low-level language.

How do I use WebAssembly?

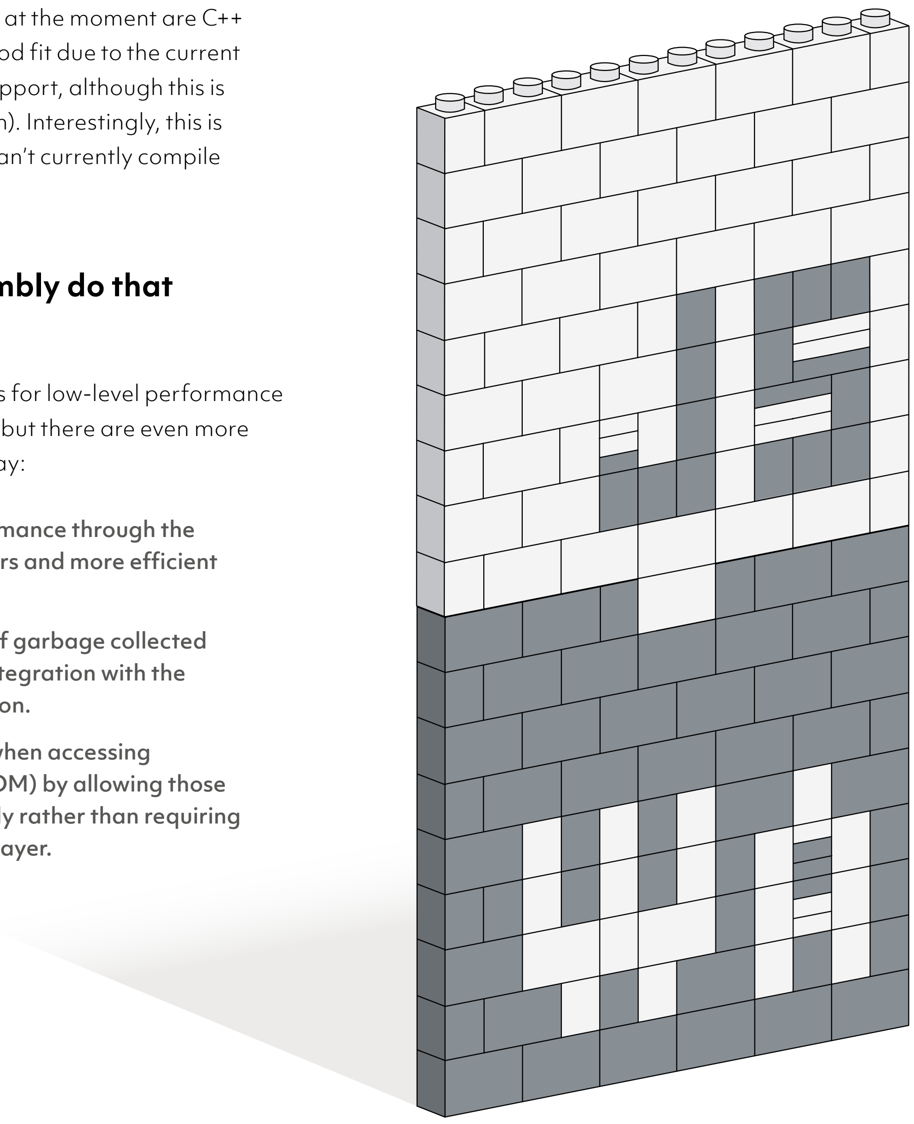
Unlike JavaScript, WebAssembly is intended to be a compilation target rather than a programming language. Its native format is currently a binary encoding of its bytecode and related data structures. Whilst there is a text format, it's only really intended for last-resort debugging. Instead, developers are expected to use higher-level languages which can compile down to WebAssembly.

The most popular languages at the moment are C++ and Rust. They are both a good fit due to the current lack of garbage collection support, although this is coming (see the next question). Interestingly, this is one of the reasons that you can't currently compile JavaScript to WebAssembly.

What can WebAssembly do that JavaScript can't?

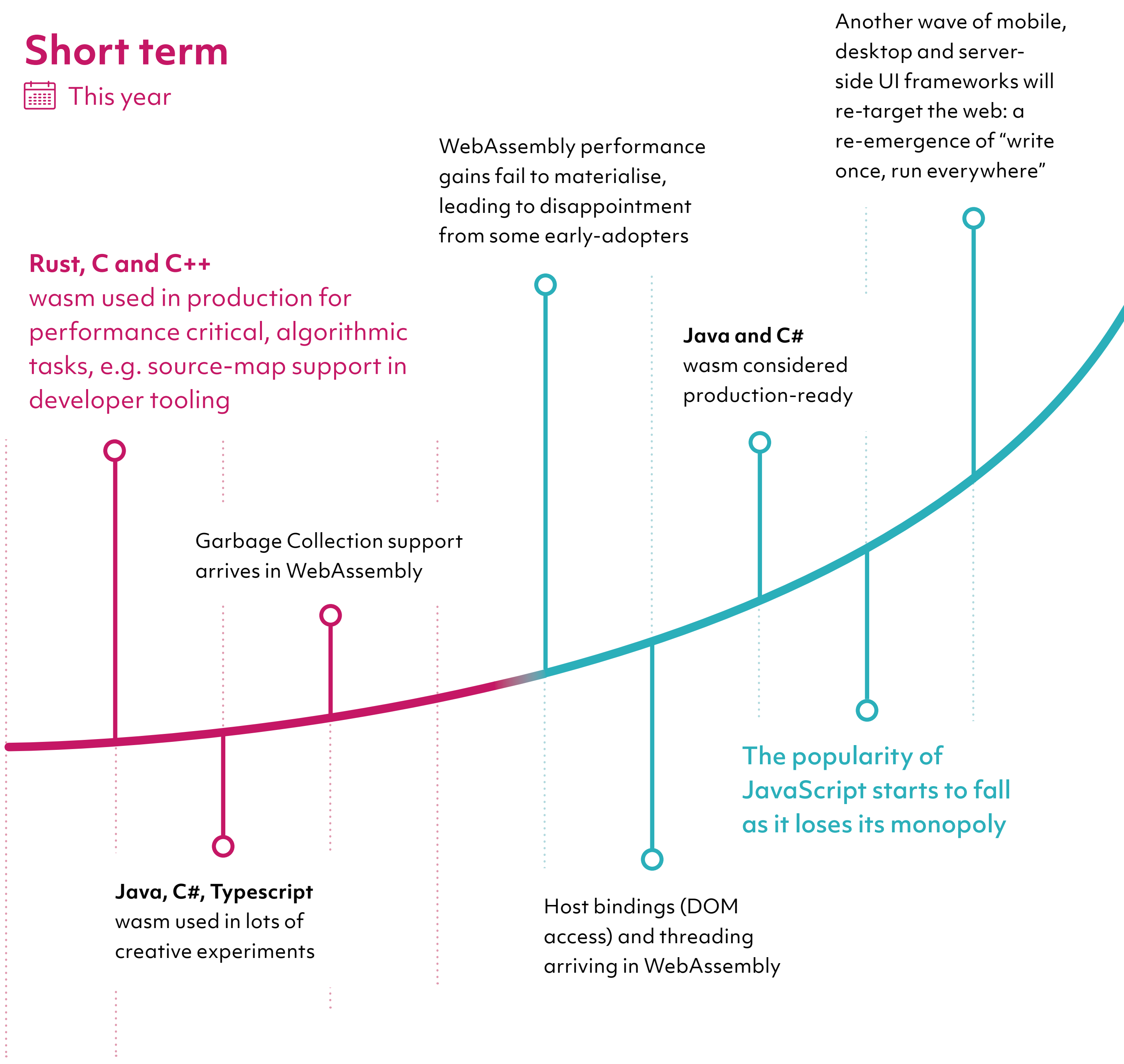
WebAssembly already allows for low-level performance not possible with JavaScript, but there are even more interesting features on the way:

- **Improved loading performance through the use of streaming compilers and more efficient binary encodings.**
- **Improved performance of garbage collected languages by allowing integration with the engine's garbage collection.**
- **Improved performance when accessing browser APIs (e.g. the DOM) by allowing those APIs to be invoked directly rather than requiring a JavaScript translation layer.**

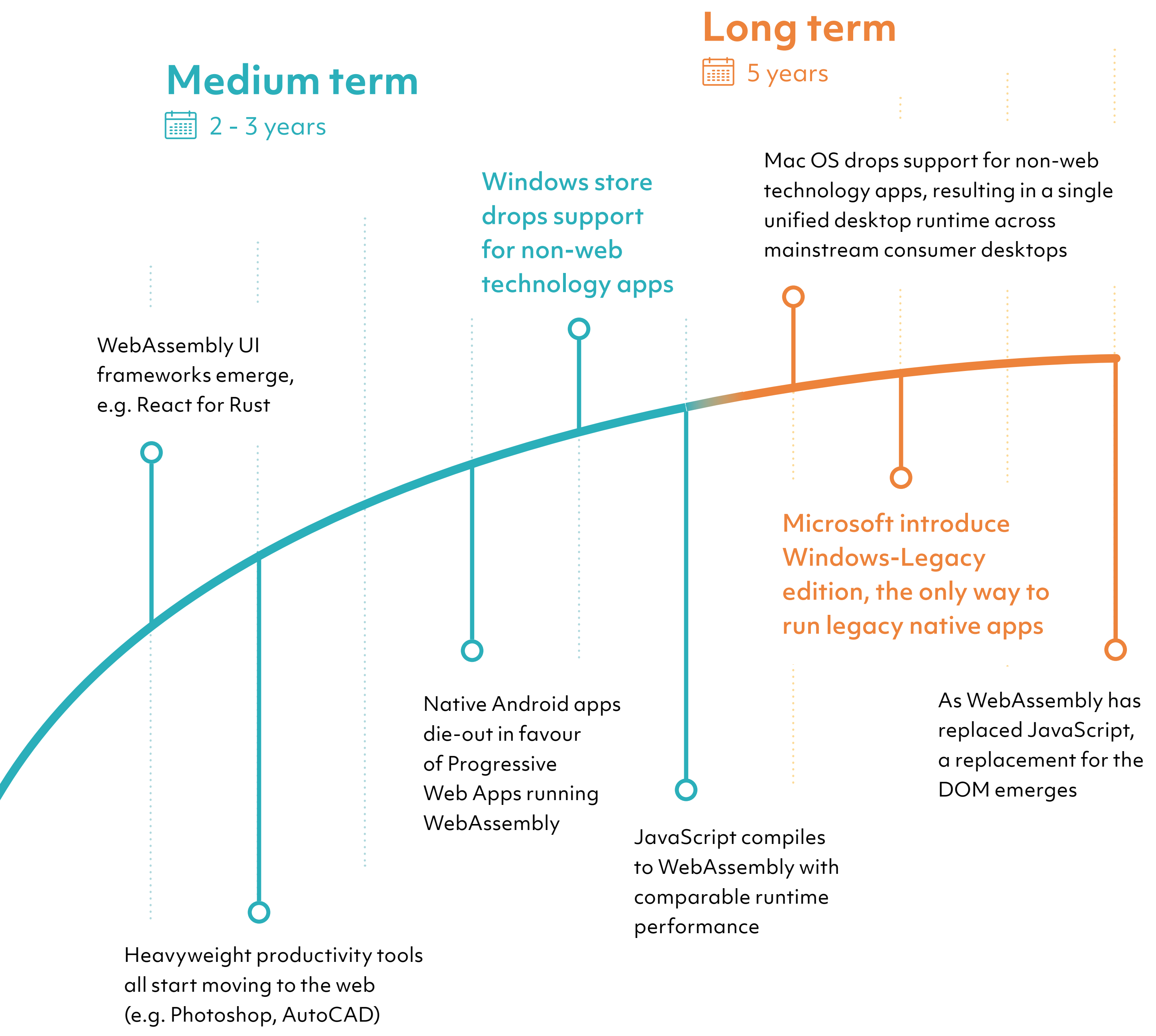


The rise of WebAssembly and fall of JavaScript

In the short term the impact of WebAssembly is likely to be small, with the immature tooling and restrictive nature of the MVP limiting its use to a few specialised use cases. However, as the WebAssembly runtime and tooling evolves, its impact could be quite far-reaching.



Short term
This year
Rust, C and C++
wasm used in production for performance critical, algorithmic tasks, e.g. source-map support in developer tooling
Java, C#, Typescript
wasm used in lots of creative experiments
Garbage Collection support arrives in WebAssembly
WebAssembly performance gains fail to materialise, leading to disappointment from some early-adopters
Host bindings (DOM access) and threading arriving in WebAssembly
Java and C#
wasm considered production-ready
The popularity of JavaScript starts to fall as it loses its monopoly
Another wave of mobile, desktop and server-side UI frameworks will re-target the web: a re-emergence of "write once, run everywhere"



Medium term

2 - 3 years

WebAssembly UI frameworks emerge, e.g. React for Rust

Heavyweight productivity tools all start moving to the web (e.g. Photoshop, AutoCAD)

Native Android apps die-out in favour of Progressive Web Apps running WebAssembly

Windows store drops support for non-web technology apps

JavaScript compiles to WebAssembly with comparable runtime performance

Long term

5 years

Mac OS drops support for non-web technology apps, resulting in a single unified desktop runtime across mainstream consumer desktops

Microsoft introduce Windows-Legacy edition, the only way to run legacy native apps

As WebAssembly has replaced JavaScript, a replacement for the DOM emerges

We predict a slow start for WebAssembly but, as it matures and gains traction, we believe it will become a central part of the web platform. WebAssembly doesn't exist in isolation, and other web technologies will continue to evolve alongside it. Most notably the umbrella of technologies associated with Progressive Web Apps (PWA) which, amongst other features, allow web apps to be installed and used offline - further blurring the line between web and native apps.

Ultimately, our more outrageous predictions see WebAssembly and PWAs pushing out native apps from operating systems, crowning the browser as the modern operating system.

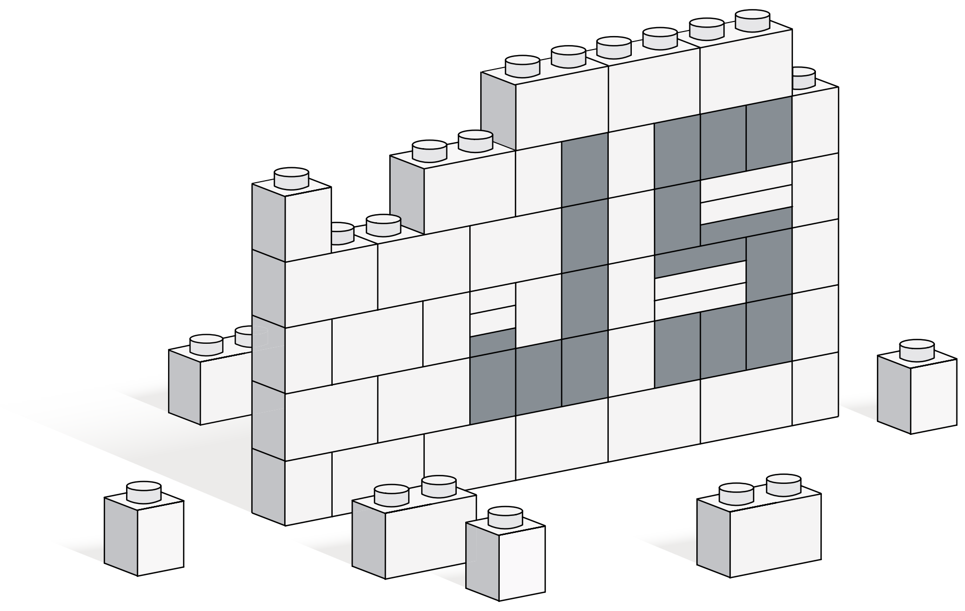
Summary

JavaScript has enjoyed a twenty three year monopoly as the only widely-supported language on the web. As of 2017, with the release of WebAssembly, the tide has started to turn.

It's going to take a while for the impact of this to be felt, but, when it does, it will be far-reaching, further cementing the ubiquity of the web, and extending its reach beyond the restrictions that JavaScript has imposed.

"The Web has become the most ubiquitous application platform ever, and yet by historical accident the only natively supported programming language for that platform is JavaScript."

Haas et al., PLDI 2017



White papers by Scott Logic

You'll find more white papers, practical guides, and technical articles on our website, please visit **blog.scottlogic.com**

UNSTOPPABLE HTML5

2016

A WHITE PAPER BY COLIN EBERHARDT

SCOTT LOGIC

ALTOGETHER SMARTER

Unstoppable HTML5

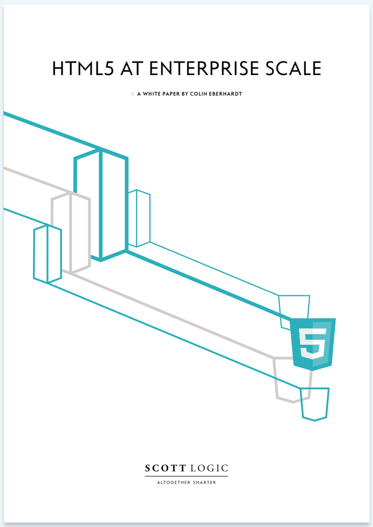
This paper looks at the many opportunities HTML5 offers for desktop development, including innovative UI technologies, accelerated application development, evergreen deployment and the prospect of a single technology for app development, regardless of the platform (be it form factor or operating system). It also looks at how the collaborative nature of HTML5 is pushing the technology forward in a way that makes it hard, if not impossible, for others to compete.

→

HTML5 at enterprise scale

Moving from desktop or plugin technologies (Flex, Silverlight, Java Applets) to HTML5 is a challenge for developers of large-scale enterprise applications. This white paper discusses the challenges and offers potential solutions.

→


HTML5 AT ENTERPRISE SCALE

A WHITE PAPER BY COLIN EBERHARDT

SCOTT LOGIC

ALTOGETHER SMARTER

If you'd like a copy of these sent to your inbox, please email **colin@scottlogic.com**

Want to discuss the impact of WebAssembly on your organisation?

At Scott Logic, we design and build software that transforms the performance of some of the world's biggest and most demanding organisations. This means truly understanding current and emerging technologies, and helping our clients make the right choices.

If you'd like to discuss the impact of WebAssembly, or any other technology challenges that face your organisation right now, we're always happy to chat.

To arrange a free consultation contact Colin Eberhardt on:

+44 333 101 0020

colin@scottlogic.com

SCOTT LOGIC / ALTOGETHER SMARTER

3rd Floor, 1 St James' Gate
Newcastle upon Tyne
NE1 4AD

+44 333 101 0020

scottlogic.com