Md Feroj Ahmod

JAVASCRIPT RUNTIME PERFORMANCE ANALYSIS: NODE AND BUN

Faculty of Information Technology and Communication Sciences (ITC)
Master's thesis
June 2023

Abstract

Md Feroj Ahmod: JavaScript runtime performance analysis: Node and Bun
Master's thesis
Tampere University
Master's Degree Programme in Software Development
June 2023

Online services are seeing a growing demand for various use-cases. Functionality of web applications are at a premium. With every new application, different functionality is being implemented with more and more complicated logic. While there are newer technologies invented for the sake of increasing the computing power, it has also been a necessity to support new inventions and improve on existing machines to create a smoother experience on using those applications.

Node.js, a JavaScript runtime, has been a reliable name in the technical industry. Node.js can generally satisfy the needs of online applications. However, its performance has been found to be uneven in applications that demand the highest levels of performance. There has been few attempts to out-weight the performance of Node.js and the most recent promising one is Bun.

The purpose of this thesis is to compare the performance of Node.js and Bun. The comparison is carried out with different use cases that includes memory usage, execution time, response time, and request throughput. In all cases, multiple sampling has been used to get a precise picture of the factors that affects the performance.

The outcome of the thesis shows that Bun is significantly faster compared to Node.js. However, the method of this thesis gives one sided view of the differences between Node.js and Bun. When considering the implementation of Bun, other factors such as security, compatibility and reliability should be taken into account.

Keywords: JavaScript, Node.js, Deno, programming, web server, back-end, performance, security.

Preface

This thesis allowed me to gain some insights on some sophisticated technologies. Completion of the thesis also marks the end of my Masters degree. I would like to thank the members of my family for their continuous support. One person who deserves a lot more gratitude than I could ever give is my mother.

Special thanks to my manager at my work. His support, motivation, and kindness allowed me to carry on the studies alongside my full-time work.

I would also like to thank the university for allowing me such opportunity. Special thanks to my thesis supervisors, educational specialist and former course coordinator for helping me throughout the journey.

June 10, 2023
Md Feroj Ahmod

Contents

1 Introduction

The world wide web, often simply referred to as the web, has undergone a great deal of changes in terms of every aspect over the course of its relatively short history. The web relies heavily on the use of hypertext markup language (HTML) which is a markup language that allows developers to create web pages that can be viewed in a web browser. JavaScript, that initially started as a simple scripting language for HTML document object model (DOM) manipulation and form validation, has become one of the most popular programming languages.

The amount of JavaScript in web applications has grown significantly due to the responsibility of execution that has increasingly shifted from the server to the client. The rise of the single-page applications (SPAs) and front–end frameworks such as React, Angular, Vue and Svelte relies heavily on JavaScript to create dynamic, responsive and interactive user interfaces. As of today, the usage of JavaScript is no longer limited to the web and has expanded its reach to other domains as well. Popular frameworks like Electron enables developers to create desktop applications using web technologies, for instance, Slack, Discord, Postman and Visual Studio Code are all made using Electron [35]. Web technologies are also an attractive choice in mobile application development. This is true especially for the case when the application deployment target is in multiple platforms. The two most popular platforms, Google's Android and Apple's iOS require separate applications to be deployed in their own ecosystem. Using web technologies in such cases enables the developers to share code components. Popular choices are Web-View, hybrid applications and frameworks such as React Native [36]. The use of JavaScript is also present in game development. Browser-based games as well as mobile and desk-top games are also being developed through frameworks such as Phaser and Three.js. Internet of things (IOT) and robotics also has seen use of JavaScript with the help of frameworks such as NodeRed, IoT.js Cylon.js and Johnny Five [39]. Artificial intelligence and Machine learning has also seen a lot of usage of JavaScript through frameworks like TensorFlow.js and Brain.js. It is also possible to use JavaScript for server-side applications with the help of Node.js, which is currently a very popular technology.

While all the major browsers have chosen JavaScript to support the longest, JavaScript was not the only option for the browsers. Multiple times, other technologies were proposed and often been achieved, what JavaScript achieved, through browser plug-ins. Solutions such as Adobe Flash, Microsoft Silverlight and Java Applet, all had their fair share of support as they helped bring the performance benefits of native code to the web. However, their reputation kept taking hits by frequent security issues. Due to the growing wariness towards plug-in-based solutions, introduction of HTML5 and the rapid growth of the mobile platforms, major players like Apple discontinued support for plug-ins led to a significant decline in their usage. As of today, most of the major plug-ins are deprecated or will be soon. As a result, JavaScript has remained the top programming language on the web.

Node.js, created by Ryan Dahl in 2009, is an open-source, cross-platform, runtime environment for JavaScript that exists to execute JavaScript code outside of a web browser allowing developers to build scalable and high-performance applications that can handle a large number of simultaneous connections with high throughput. Node.js is powered by Google's V8 JavaScript engine, which allows it to run at high speeds by compiling JavaScript code into machine code. Due to its non-blocking event-driven architecture, Node.js is known for performance, efficiency and flexibility. However, its performance in complex scenarios has remained inconsistent. There have been attempts to gain more speed and performance by creating Deno, a JavaScript runtime from the creator of Node.js. While Deno is an improvement in many areas, it is built with V8 engine, same as Node.js, thus having similar limitations of Node.js. The newest, very promising, runtime in the context solves the issues in context while designed to be the drop-in replacement for Node.js, is Bun. This means that all the existing applications that run in Node.js would be running the same.

The topic of this thesis is to delve into Bun, its features, relationships with Node.js, evaluate its current state and future. Furthermore, in this thesis, we implement some sample applications of different computation contexts, evaluate and analyze the results after running them using Bun and Node.js in a test setup. The research was conducted based on few research questions, explained in Chapter 4, mentioned below:

- Is Bun faster than Node.js?
- What affects the performance of runtimes?
- What factors may contribute when selecting a runtime?

Literature review was conducted and Chapter 2 covers related work regarding JavaScript, JavaScriptEngines, Node.js and Bun. In Chapter 3, software quality aspects will be covered in detail, especially performance. Chapter 4 will cover the research methodology including the implementations of the various small test applications of different contexts. In Chapter 5, we will evaluate and discuss the results of our test followed by Chapter 6 that will wrap up with conclusions.

2 Background

Along with all the other platforms, the web, especially during the last 20 years, has been quite the attractive platform for applications. While HTML5 opened a door to natively implement many of the features todays web applications have, those features were mostly handled by plug-ins, such as Adobe flash, previously, particularly the multimedia. Eventually, the necessity of plug-ins had run out and the performance gaps between plug-ins and native solutions were closed by WebAssembly and Web graphics library (WebGL) [1]. While other runtimes have not became obsolete by the introduction of Node.js, it certainly did place a giant footstep in the scenario. Node.js is particularly efficient and well-suited for real-time data transfer and processing applications. It is also better suited for light-weight and fast server-side rendered applications. Since Node.js is single threaded, complex applications that re-quire more processing power can put Node.js in a difficult position. Node.js also uses a large amount of memory for complex scenarios. All these difficulties of Node.js helped with the design and implementation of Bun and its exclusive application programming interfaces (API) [34]. Bun chose JavaScriptCore engine from the Webkit project unlike Node.js which uses Google's V8 engine [50], where both engine uses Just-In-Time compilation.

2.1 JavaScript

Although it shares partial syntactic similarity, JavaScript is largely unrelated to Java, despite its name, they are completely different programming languages. It is a dynamically typed, primarily a web programming, scripting language. It is a high-level programming language that is interpreted and includes features that are commonly associated with both object-oriented programming (OOP) and functional programming [16]. JavaScript was created originally for Netscape Navigator browser by Netscape Communication [31]. However, it was subsequently standardized by ECMA International as ECMAScript [11]. JavaScript is widely regarded as one of the most utilized programming languages in the world [38]. The role of JavaScript in managing various functionalities on a web page is so integral that it has become practically impossible to engage in web development without encountering JavaScript. JavaScript was originally created for web browsers but Node.js, discussed in Section 2.4, helped extend it's reach out of web browsers to web server, desktop, mobile and even embedded scenarios.

Being an interpreted language, JavaScript compiles the code into machine language during runtime [16]. An interpreter executes the code line-by-line as it runs.

This opens the possibility to a faster development cycle since any changes to the code can be made quickly without having to compile the entire program. However, such features being a characteristic of a dynamically typed languages like JavaScript, they can negatively impact program performance compared to compiled code.

2.1.1 History

JavaScript is a programming language that was initially developed by Brendan Eich at Netscape in May 1995. It was originally named as Mocha, and later renamed to LiveScript when it was released as beta software in Netscape Navigator 2.0 [20], a web browser by Netscape, in September 1995 [2]. However, it was ultimately renamed JavaScript on December 4, 1995 [31, 2], upon its deployment in a later beta of Netscape browser version 2.0 [31]. Despite the name sounding similar, JavaScript has very little to do with Java. At the time of the birth of JavaScript, Java was a popular choices of programming language among developers. Mocha was created with syntax inspired by Java [40]. Before being renamed, LiveScript incorporated many features from Java, and the desire to leverage Java's growing popularity and positive associations played a significant role in the decision to ultimately rename it as JavaScript [20].

Server-side implementation was not a new concept when Node.js first appeared. Shortly after the release of JavaScript for web browsers, Netscape introduced a version of JavaScript for server-side scripting (SSJS) in their Netscape Enterprise Server [20]. Fast-forward, since the mid-2000s, there has been a significant increase in the number of server-side JavaScript Implementations available. One notable example of server-side JavaScript implementation being utilized in real-world applications is Node.js which has became increasingly popular in recent years [43].

When JavaScript 1.0 made it to the market, its popularity was increasing and helped Netscape Navigator to hold the leader position. By the time Netscape Navigator 3 was released with JavaScript 1.1, Microsoft, to compete with Netscape, decided to opt-in for scripting technologies in their browser and, in 1996, introduced its proprietary implementation of JavaScript with the release of Internet Explorer 3 [20, 40]. It was called ”JScript” to avoid any possible trademark issues by keeping “Java” off the name [40]. Like the name, although it shared many similarities, JScript was also different in terms of implementation [40].

With the growing competition and from the worry of those implementation straying too far away from JavaScript, concerning compatibility issues [20] and Microsoft's no intention of cooperation, Netscape formally submitted JavaScript to the European Computer Manufacturer's Association (ECMA) for consideration as an industry standard language [20, 23]. ECMA is an organization established in

1961 that is dedicated to promoting the standardization of information and communication systems [40]. As a result of ensuing work, the standard version of the language was developed and named ECMAScript formally and today what is known as “JavaScript” is only the commercial name for ECMAScript [40]. In June of 1997, ECMA International published the initial version of the ECMAScript standard, designated as the ECMA-262 specification [11, 20]. One year later, in June of 1998, certain modifications were made to adapt the specification to the ISO/IEC-16262 standard, leading to the release of the second edition [40, 12]. The third edition of the ECMAScript, which was published in December of 1999 [13], is the version that is base line target for many libraries and most widely supported by current web browsers [40].

JavaScript and the community struggled in the years between ECMAScript 3.1 and 4. In 2003, Brendan Eich proposed ECMAScript 4, a major update, introducing some significant new features. However, some of the community stood against it believing that the changes were too radical and would break backward compatibility. This result-ed with a ling period of debate and negotiations within the ECMAScript committee and splitting the group into two. One, led by Brendan Eich, wanted to move forward with ECMAScript 4 while the other preferred to focus on incremental updates to ECMAScript 3.1 that would ensure backward compatibility.[40]

In 2005, Jesse James Garret, co-founder of Adaptive Path, introduced the term AJAX (Asynchronous JavaScript and XML) along with a suite of technologies. The key goal of this approach was to enable the background loading of data without the need of full page reloads and resulting more responsive and dynamic applications [17]. JavaScript served as the foundation for the development of web application. This ground-breaking development spurred the creation of several other open-source libraries, including but not limited to Prototype, jQuery, Dojo and Mootools. Although it was not part of the ECMAScript 3 standardization, Microsoft implemented similar technology in it’s Internet Explorer 5 browser named XMLHttpRequest. Later, XMLHttpRequest was integrated into standards that are part of the WHATWG and the W3C groups, following its success [40].

Major browser vendors, including Microsoft, Mozilla and Opera, further complicated the debate by implementing their own versions of ECMAScript, in their browsers, which diverged from the proposed standard. Achieving a consensus of the future of the language became much more difficult. Fast forward, in 2008, the ECMAScript committee eventually abandoned ECMAScript 4 and instead focused on incremental updates to ECMAScript 3.1. ECMAScript 4, despite the controversy, did have a lasting impact on JavaScript. Many of its proposed features such as classes, generators and iterators were eventually added to ECMAScript in later versions.[40]

JavaScript has often been characterized as a “toy” language, mainly due to its humble beginning in simple scripting and associated simple use cases [47]. Today, JavaScript has come a long way and evolved from being a “toy” language to something that is almost irreplaceable and most desired [45]. Multiple programming paradigms such as object-oriented, imperative and functional styles are supported by JavaScript. It has expanded it’s reach out of web browsers and extended to various domains such as Server-side, Desktop, Mobile, Robotics and Internet of Things (IOT) [38].

2.1.2 Current state

JavaScript is one of the most widely used programming languages in the world [45], with a long history of evolution and development. Today, JavaScript is used in a wide variety of contexts. It is the primary language used in front-end web development as well as a back-end server-side development with the rise of Node.js [10]. It is also quite commonly used for mobile application development, using frameworks such as React Native and Ionic [27], as well as Robotics and IOT.

Everything that JavaScript does, speed is at the heart of it. JavaScript is an interpreted language, meaning it does not have to be compiled every time it’s run, and reduces the time required for development and debugging to start with. While running as a client script, JavaScript finds more speed in the browser without connecting to the server and saving resources. Client-side operations also saves resources of the service side which reduces potential server load. JavaScript is backed by a huge community which makes getting support quicker, easier and ensures rapid development and faster growth.[10]

Despite its advantages, widespread use and popularity, JavaScript still faces challenges and limitations. Due to the fact that it is less strict on conventions and rules, such as assigning types or ending line with semicolon, often lead to bad habits. By design, JavaScript code can be viewed at the client’s side which may open opportunities of attack for malicious parties looking to gain access to related systems, cause damage or deny valid access. Depending on the browser, JavaScript might be interpreted differently resulting users receiving inconsistent experiences and inconvenience [10]. Working with large amounts of data or when running complex calculation may lead to performance issues since JavaScript, by design, is single threaded and unable to take advantages of multi-core processors. JavaScript has limited support for low-level operations, such as no direct access to resources, and Object-Oriented Programming (OOP) [19].

The road ahead of JavaScript looks bright with continued growth and innovation in the language and its ecosystem. The emergence of new technologies such as machine learning and artificial intelligence is also opening new possibilities for

JavaScript. The current state of JavaScript is one of rapid growth and innovation with vast ecosystem of libraries, frameworks and tools that make it one of the most versatile and widely used programming languages in the world [16, 45]. Despite its challenges and limitations, like any other programming language, JavaScript continues to evolve and adapt to new technologies and uses cases, making it an essential tool for modern development.

2.1.3 Runtimes

A runtime, standardized library and wrapper for a program written in each language, refers to the context within which a programming language is executed. This is needed to execute the compiled programs, applicable regardless of the specific language needs compiling or not [30]. Associated runtime system of a programming language facilitates the functions, variables and the management of memory through the usage of data structure like queues, heaps and stacks. It also provides the necessary tools to interact with the resources [24]. The runtime often provides an abstraction layer in order to make system calls to the operating system and as an OS version update occurs, the runtime is expected to be appropriately updated to match. Figure 1 illustrates a Java Runtime Environment.

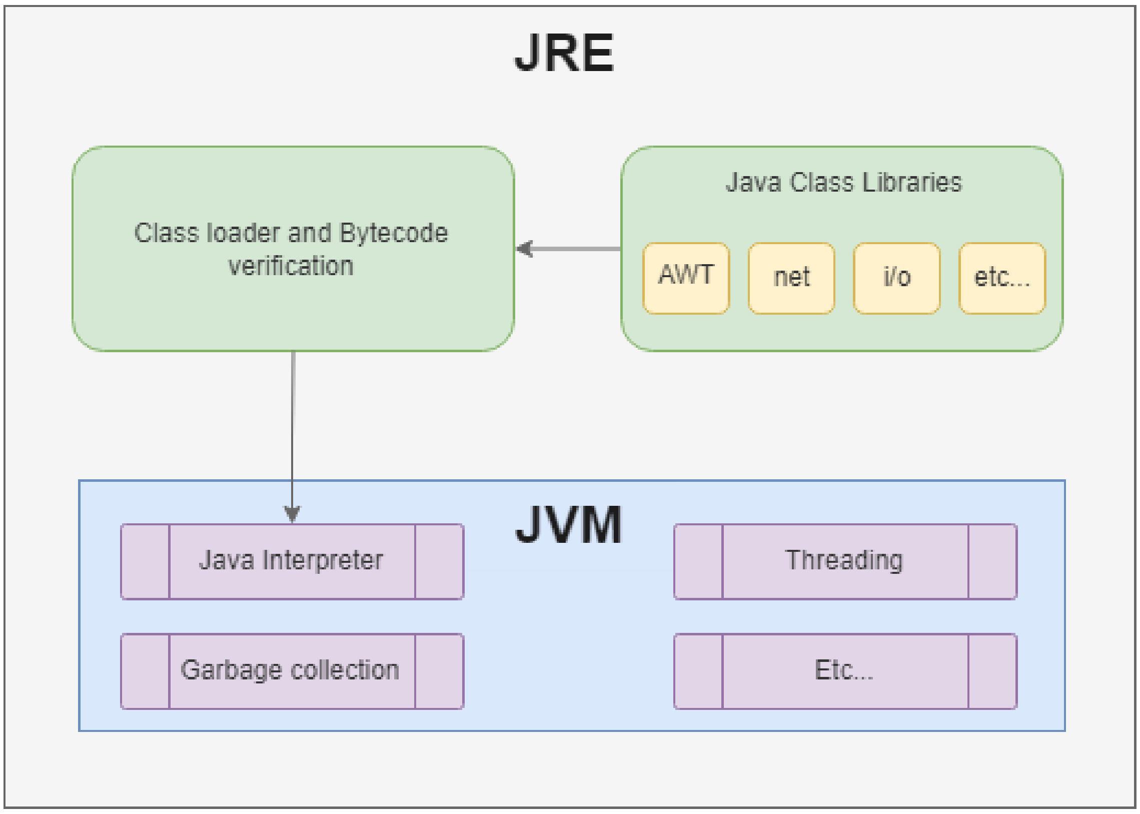

Figure 1 *An over-view of Java Runtime Environment (JRE).*

A JavaScript runtime, illustrated in Figure 2, is a program that extends the functionality of the JavaScript engine, discussed in Section 2.3, and enables it to interact with external entities. Additionally, the JavaScript runtime offers features and APIs to facilitate the development of software that is built on the JavaScript language. This implies that both browsers and JavaScript-based frameworks have their respective runtimes, which vary according to their specific requirements.[9, 24]

JavaScript has primarily two types of runtimes: browser based, and server side. Browser based runtimes execute JavaScript code within the context of a web browser. This is the most common context of JavaScript code execution. Server-side runtimes executes JavaScript on a server or anywhere outside of a web browser. This is entirely a different scenario. While some of the functions may be available, most of the functions, such as window.alert(), can not be used. Instead, the server-side runtime environment gives access to the end users a variety of features that is unavailable in the browser, such as access to the file-system, database and network.[48]

A JavaScript runtime is contextual and can manifest itself in various forms depending on the scenario. For instance, the runtime environment of a browser differs greatly from that of Node.js. However despite these disparities predominantly existing at the implementation level, the fundamental concepts outlined below remain pertinent [9]:

- JavaScript Engine
- Web APIs
- Callback queue
- Event loop

A JavaScript engine, discussed in Section 2.3, is a software program that reads and executes code by translating it into machine code. This translation process is followed by the execution of the resulting machine code, allowing a computer to perform specific tasks defined by the originally written code. JavaScript engines are typically embedded within JavaScript runtime environments, including browsers and Node.js.[24, 9]

The main elements of a JavaScript engine are the heap and the call stack. The heap is often called the ”memory heap”. Heap is a large unstructured data structure that stores all the dynamic data like function definitions, objects, arrays etc. The memory heap is where the memory allocation happens. The memory is occupied until the garbage collector removes it. The call stack is another data structure that stores the execution context/code for each function.[24, 9]

Web APIs, available in modern browsers, are what allows us to do wide variety of operations. The most common ones includes manipulating documents, draw

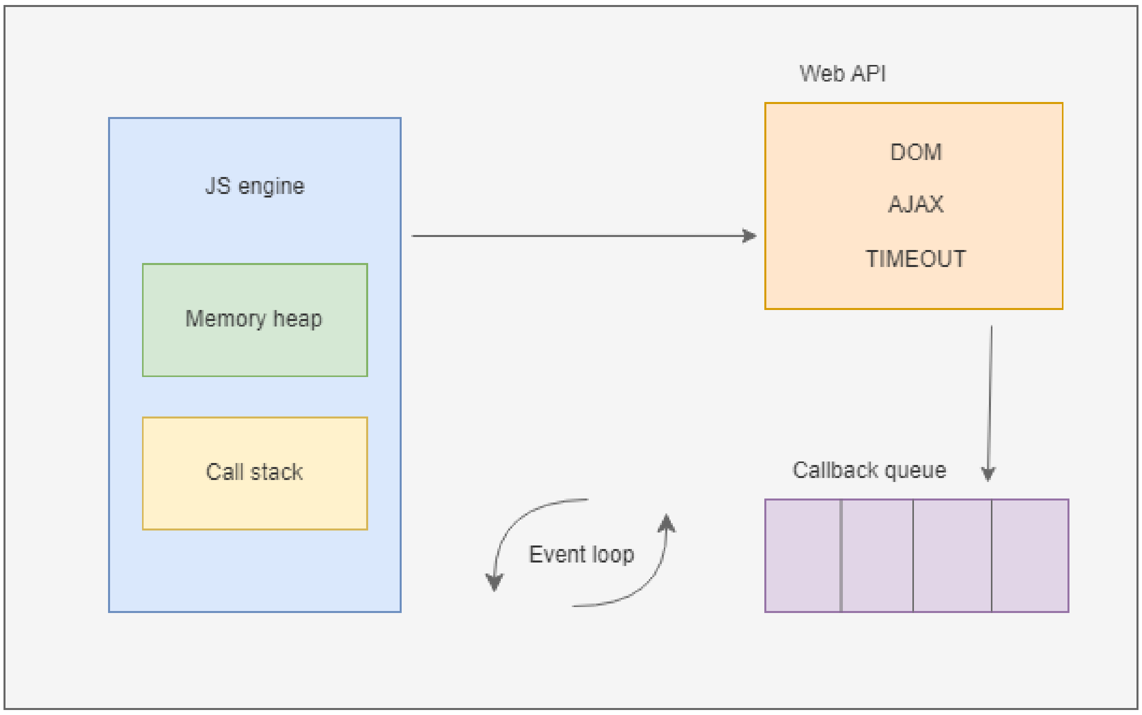

Figure 2 *An over-view of JavaScript runtime.*

and manipulate graphics and fetch data from a server. The Web API container houses various features such as event listeners, timing functions, and AJAX requests, until they are invoked by a particular action. Upon completion of a requests data reception, expiration of a timers set duration, or occurrence of a click event, a corresponding callback function is promptly added to the callback queue.[24, 9]

It is to be noted that not all APIs are inherent to the browser. Numerous notable websites and services such as Twitter, Facebook, Google Maps, PayPal, to name a few, have incorporated APIs in their framework, thereby granting developers the liberty to exploit their data. Contemporary web browsers encompass a multitude of distinct technologies, enabling users to stockpile data associated with websites and fetch it as and when required, thus facilitating the long-term persistence of data, offline site storage, and various other functionalities.[7]

The Callback queue, illustrated by Figure 3, is a storage unit for callback functions that are transmitted from the Web APIs and are organized in the order in which they were appended. This queue follows a First-In-First-Out (FIFO) sequence as a data structure. Callback functions remain in the queue until the call stack is free of any active functions, at which point the event loop comes into the play and directs the queue to move them into the stack for execution.[29, 24, 9]

The Event loop, responsible for for executing the code, continuously monitors the status of both the call stack and the callback queue. In the event that the call stack is devoid of any functions, the Event loop retrieves a callback from the queue

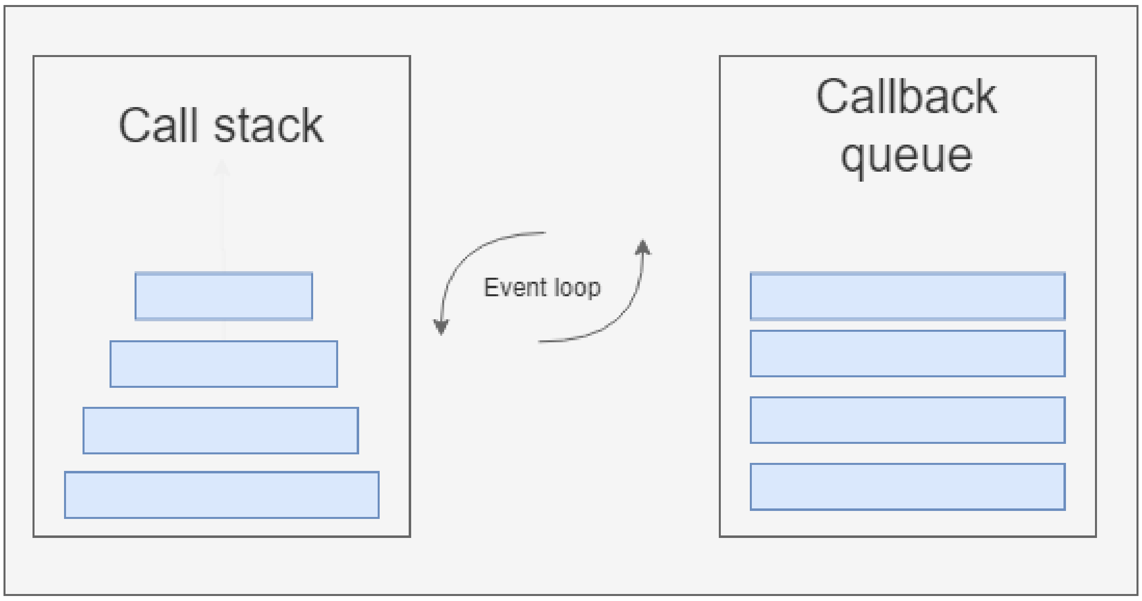

Figure 3 *An over-view of JavaScript callback queue.*

and positions it onto the call stack, thereby scheduling it for immediate execution. As the Web APIs send callbacks to the callback queue, the Event loop continuously adds them to the call stack, thereby inducing the perception of JavaScript's ability to execute asynchronously, despite being single-threaded.[29, 24, 9] As a result, the event loop model of JavaScript has an intriguing characteristic, in contrast to many other programming languages, is non-blocking. The handling of Input/Output (I/O) operations is generally accomplished via events and callbacks. As such, when an application is anticipating the outcome of database query or an XMLHttpRequest request, it can concurrently process other tasks, such as user input [8].

2.2 Just-in-time Compilation

JavaScript is an interpretive language. The interpreter runs the code line by line as it is being created, as opposed to turning it into machine language beforehand. This method permits code modifications without requiring the application to be completely compiled, which shortens the development time. When running the same code repeatedly, such as in a loop, the interpreter must translate each iteration, which might slow down the program's speed compared to compiled code. Browsers began implementing just-in-time (JIT) compilers to improve efficiency and alleviate some of the interpreter's drawbacks. Implementing JIT for browsers and JavaScript was a fresh idea, even though JIT was not a brand-new concept. It had been utilized in LISP and Java before. JIT for browsers was initially introduced by Google's Chrome V8 JavaScript engine, which was motivated by the rising popularity of JavaScript-heavy applications like Google Maps that exposed JavaScript's performance difficulties. As more browsers adopted this strategy, JIT compiler development in browsers advanced quickly due to competition.

Interpreters are convenient, for a dynamically typed programming language, and quick, since they do not need to go through the compilation of the whole code, it is not very efficient since they run the code line by line where the same translation is happening over and over again. Compilers eliminates that but with the cost of execution time since it needs to compile the whole code beforehand. Just-in-time compilers is basically the best of both compiler and interpreter. [6]

Although the exact technology varies slightly between browsers, the fundamental idea is the same. Browsers started employing a profiler to keep track of the types and frequency of code execution. The code is initially reported as cold after being executed through the interpreter by the profiler. The code is labeled as warm if it is performed more than once. Finally, the code will be labeled as hot if it is frequently executed. The baseline compiler will compile and save functions that are designated as warm. The function's internal lines of code are all compiled as stubs that are indexed by line number and variable type. The profiler can determine if the same code and the same variable types are executed repeatedly due to this indexing. If such is the case, it may return the compiled version, which helps to increase performance. [6]

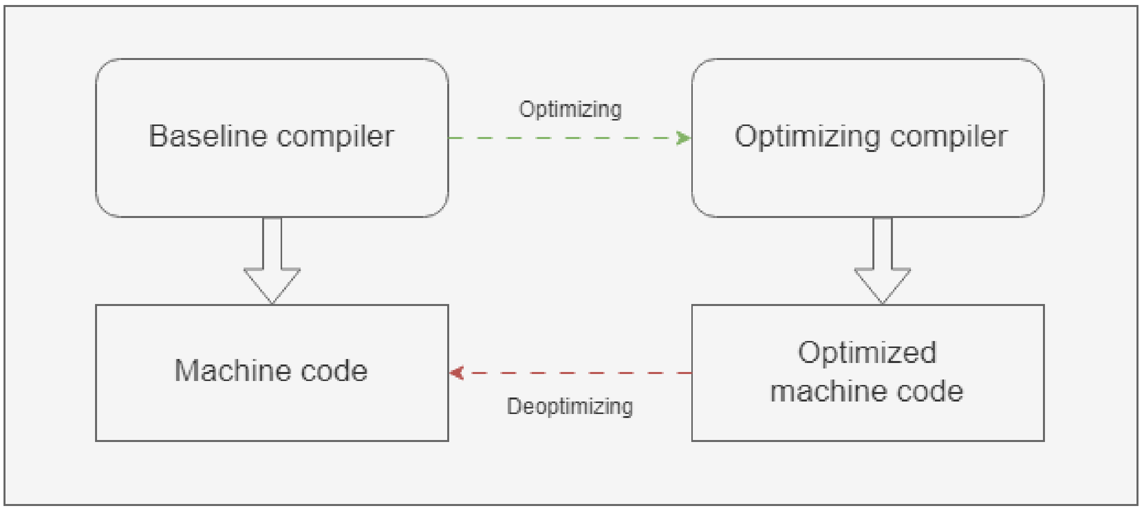

Figure 4 *JIT flow (optimizing and deoptimizing).*

Further optimization can be performed after the profiler recognizes a section of code that is regularly called and marks it as hot. However, the compiler needs make some assumptions based on the information the monitor has gathered in order to optimize the code. Loops can be significantly optimized because of how interpreters work, which causes them to suffer severely. Along with optimization, JIT compiler workflow may include deoptimization if necessary illustrated in Figure 4. A loop is presumed to remain true if it has been true for all prior iterations. JIT will give up on the assumptions and the optimized code if the assumption is wrong, such as when variables entering the loop differ from prior iterations, and will instead revert to the interpreter or the baseline compiled version that was made from warm code. This is called deoptimization. [6]

JIT compilers have enhanced JavaScript's performance [6], especially as optimizations have grown and matured over time. But sometimes this improved performance comes at a price. The JIT compiler must record and store information about code execution statistics, recovery data from bailouts caused by faulty assumptions, and both the baseline and optimized versions of a function. This requires more browser memory. When a section of code is repeatedly optimized and deoptimized, unpredictable performance may occur, leading to slower performance than simply running the baseline version. Fortunately, browsers include restrictions on how frequently this kind of cycle can happen.

2.3 JavaScript Engine

The inventor of JavaScript himself, Brendan Eich, created the first JavaScript engine for the Netscape Navigator web browser. It was implemented in C++. This engine later evolved to SpiderMonkey, which Firefox uses as their JavaScript engine today [18]. A great feature of modern JavaScript engines is that they are independent of the browsers in which they are hosted [49], enabling a door to execute JavaScript codes out-side of the browsers.

JavaScript is known as a interpreted language. However, many of the modern JavaScript engines no longer just interpret the code, they also compile it using some internally built in JITs [49]. In order to understand JavaScript engines, let us go through a little terminology first. The classification of a 'JavaScript engine' as a subset of virtual machines is widely acknowledged. Virtual machines, by definition, are software-based imitations of computer systems, and their various types are categorized according to their capacity for accurately emulating or replacing physical machines. 'System virtual machines' provide comprehensive emulation of platforms on which operating systems can be executed. For example, Mac users are acquainted with Parallels, a system virtual machine that facilitates the execution of Windows on a Mac. 'Process virtual machines', on the other hand, are less multi-functional and are capable of running a single program or process. Wine is a process virtual machine that enables Windows applications to run on Linux machines, but it does not provide a complete Windows OS on a Linux platform.[28]

A JavaScript engine is a process virtual machine that is specifically designed to interpret and execute JavaScript code [24, 9, 28]. JavaScript engines are responsible for translating JavaScript code written by developers into optimized machine code that can be interpreted by a browser or embedded in an application. JavaScriptCore,

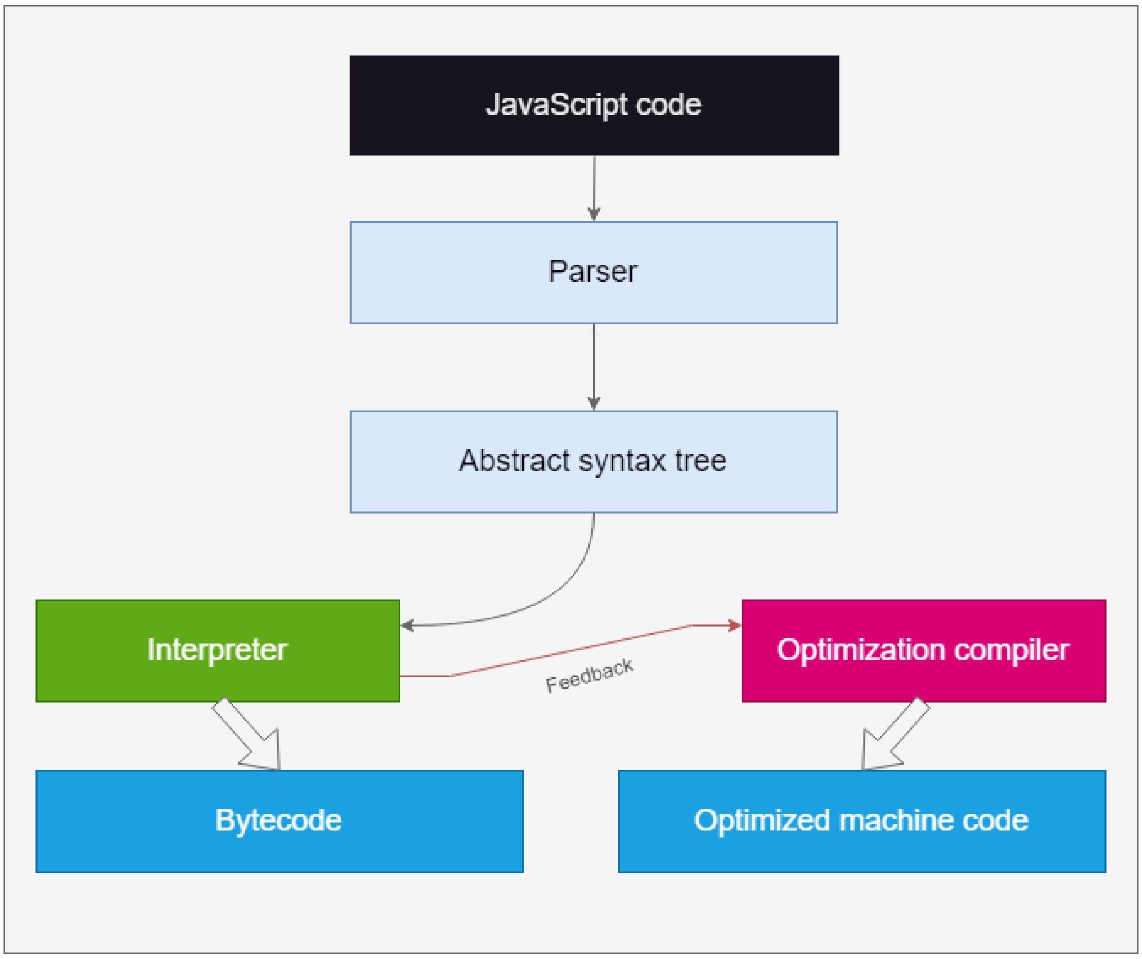

Figure 5 *General workflow of a JavaScript engine.*

a JavaScript engine discussed in Subsection 2.3.1, for instance, is an ”optimizing virtual machine” that focuses on the task of optimizing code for peak performance[28].

Despite different JavaScript being implemented differently, a general workflow, illustrated in Figure 5, is common among them. JavaScript code is first analysed and parsed by lexical analysis and tokenization which is a process of dividing code into pieces such as keywords, identifiers, punctuation, operators etc. An Abstract Syntax Tree (AST), illustrated in Figure 6, is created from the resultant which is sent to the Interpreter. The abstract syntactic structure of a particular source file is represented by the AST as a tree. This implies that it might theoretically be translated into any other language. The Interpreter, in general, interprets the AST and generates byte-code. It also collects feedback from the generated byte-code and send those to the Optimization Compiler, generally JIT, which is responsible for generating optimized machine code. Different engine may have different kind of byte-code compilers and syntax.

Since JavaScript is a dialect of ECMAScript [40], each implementation of a JavaScript engine is an implementation of a version of ECMAScript. The evolution

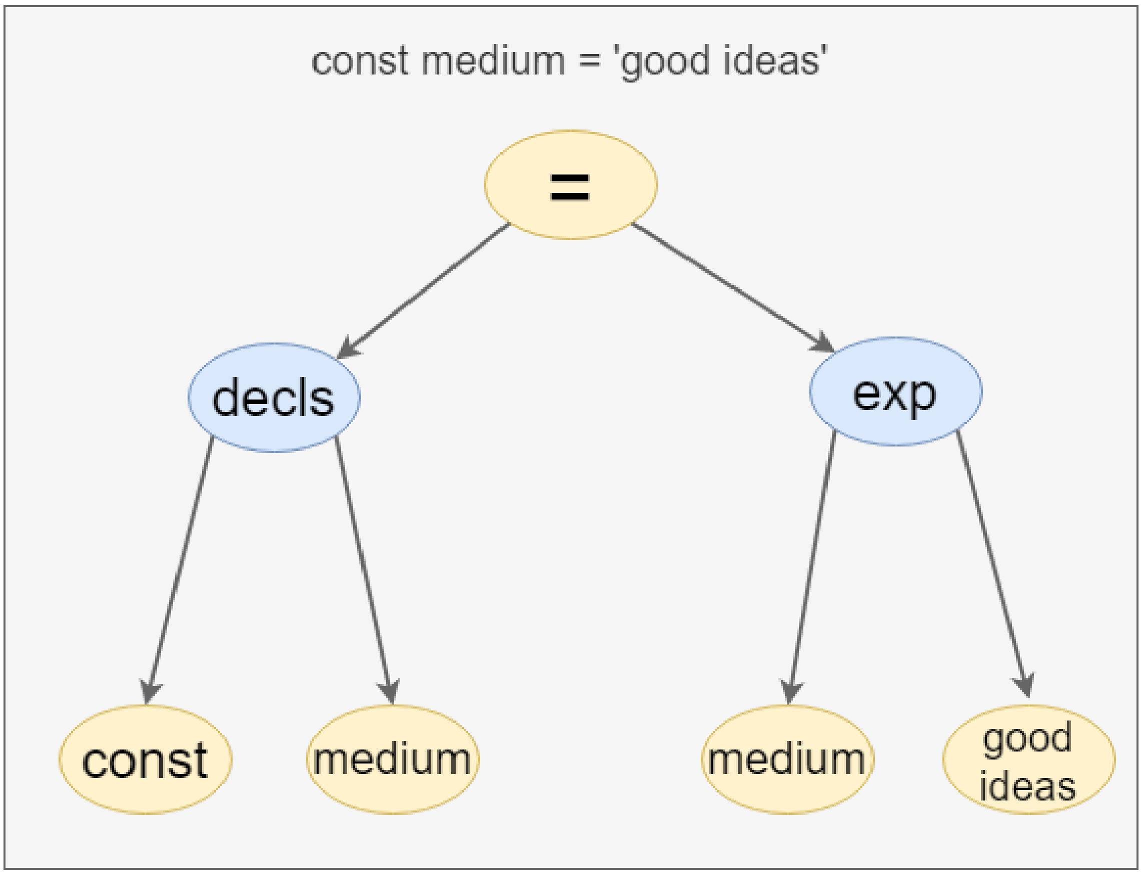

Figure 6 *Visual representation of AST.*

of ECMAScript is matched by the development of JavaScript engines. These engines are designed to function seamlessly with diverse web browsers, headless browsers (browser without user interface, such as PhantomJS), and runtimes such as Node.js, and Bun, which accounts for the plethora of different engines in existence.

A JavaScript engine serves the singular function of reading and compiling JavaScript code. However, this doesn't imply that the engine itself is a simplistic construct. For instance, JavaScriptCore, discussed in Subsection 2.3.1, incorporates six sophisticated 'building blocks' that methodically analyze, interpret, optimize, and manage memory for JavaScript code. [28]

2.3.1 JavaScriptCore

JavaScriptCore is the default JavaScript engine for WebKit, an open source web browser engine used by Safari and other applications. It adheres to the ECMAScript standard, as specified in the ECMA-262 specification. Despite its official name, it is also referred to as SquirrelFish and SquirrelFish Extreme, while within Safari, it is called Nitro and Nitro Extreme. Nevertheless, the official name for this project and its library is always JavaScriptCore. Despite its primary use in Safari and

WebKit, the engine has been made available for use in other applications and runtime environments. [53]

JavaScriptCore functions as an optimizing virtual machine. The JavaScript-Core comprises several constituent components, namely the lexer, parser, start-up interpreter (LLInt), baseline JIT, a low-latency optimizing JIT (DFG), and a high-throughput optimizing JIT (FTL), to analyze, compile, and optimize JavaScript code for better performance. Lexer, hard-coded, performs lexical analysis, such as breaking down the script source into a series of tokens. Parser, another hard-coded, follows up with the syntactic analysis, such as building the AST based on tokens from the Lexer. [53]

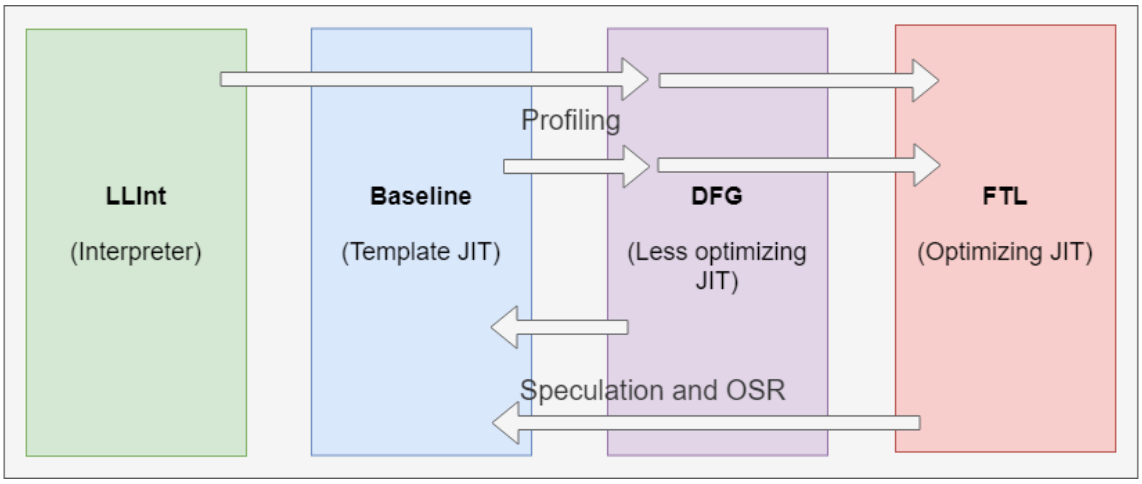

Figure 7 *JavaScriptCore optimisation architecture.*

While JavaScriptCore has similarity with the general JavaScript engine workflow, it does things a little different at this point. Instead of one, JavaScriptCore implements multiple different JIT compilers for code optimization. From this point on, as illustrated in Figure 7, every stage performs optimizations based on their previous stage feedback. Triggering of Baseline JIT, DFG JIT and FTL JIT happens when a certain threshold is reached in terms of invocation or execution. The path is linear meaning for example, FTL JIT will not start without going through Baseline JIT and DFG JIT. It is to note that any kind of threshold mentioned is an approximate and the actual may vary depending on function size and current memory in use. [53]

The first execution of any function always starts in the interpreter tier [42]. The Low Level Interpreter AKA LLInt, written in a portable assembly called offlineasm, executes the resultant byte-codes from the parser. Beside lexical analysis and parsing, the LLInt intends to have minimum start-up cost and includes optimizations such as inline caching to ensure fast property access. [53]

While the LLInt is optimized for low latency start-up, the Baseline JIT is optimized for high throughput [42]. This component is responsible for optimizing functions that are called at least six times or run through a loop at-least 100 times. When these conditions are met, the Baseline JIT will optimize the code for faster execution. Baseline JIT acts as a fallback for the functions, that were compiled by the the Baseline JIT, if they encountered an unhandled case. A sophisticated polymorphic inline caching is also performed by the Baseline JIT for almost all heap accesses. Baseline JIT and LLInt collects light-weight profiling information, such as values of argument, heap or a call return, to enable speculative execution by the next tier of execution, the DFG JIT. [53]

The DFG JIT starts to act to optimize functions when their invocation reaches 60 times or runs through a loop at-least 1000 times. It is a just-in-time compiler in JavaScript engines that optimizes functions based on profiling information. It performs aggressive type speculation to forward-propagate type information and sometimes speculates on values themselves to enable in-lining. De-optimization (called OSR exit by the authors) is used to handle cases where speculation fails, and the Baseline JIT and the DFG JIT share a two-way OSR relationship. Repeated OSR exits from the DFG serve as a profiling hint for re-optimization, which uses exponential back-off to defend against pathological code. [53]

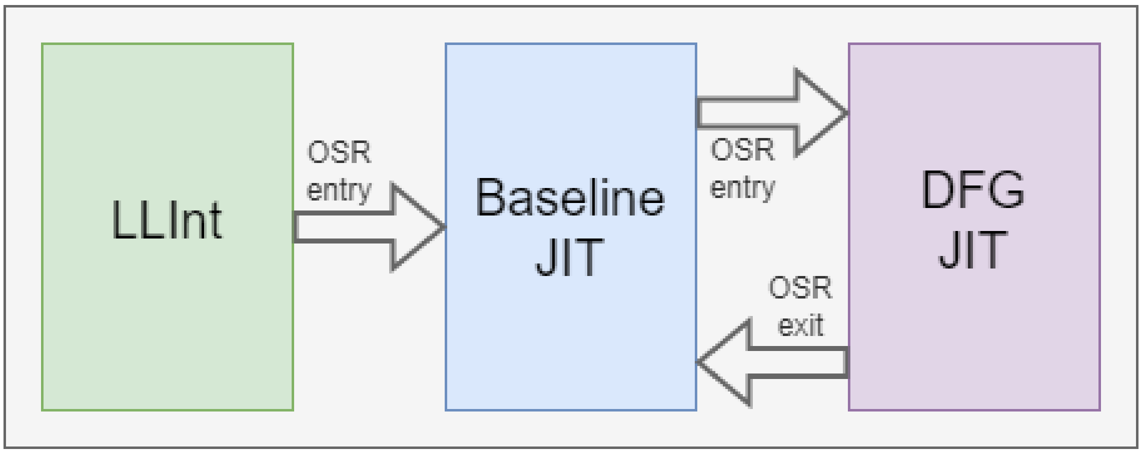

Figure 8 *JavaScriptCore three-tier architecture prior to the FTL JIT.*

When functions are invoked thousands of times or loop tens of thousand of times, the FTL JIT, enabled by default for Mac and iOS ports, takes over and starts to optimize code. It is a top-tier optimizing compiler in JavaScriptCore engine and is a combination of DFG JIT compiler and a lowering phase which is responsible for high-level optimization and type inference respectively. Despite the FTL JIT largely reuses existing DFG functionality they are not the same in terms of output. While DFG focuses on optimizing code that are frequently executed. The FTL JIT primarily concerns about generating code that is as fast as possible, by generating native machine code that is specialized to the program's execution context, which can lead to significant performance improvements specially in-terms of memory management. [42, 52]

Initially, as illustrated in Figure 8, JavaScriptCore used a three-tier strategy for optimizing code that included the LLInt, Baseline JIT and DFG JIT. The FTL JIT was later introduced as a resultant of work on a bug and designed to bring C-like optimizations to JavaScript [42, 52]. Although FTL JIT started with a combination of DFG and LLVM back-end, it was later updated and replaced LLVM with B3 back-end [41].

As of right now, it is clear that JavaScriptCore prioritizes both quick code execution and minimal memory utilization. It accomplishes that with the aid of its three JIT compilers, which are somewhat complex but performant.

2.3.2 V8

Google first created the Google V8 JavaScript engine as an open-source project to power Google Chrome, the company's primary online browser. Nowadays, It is utilized in many different applications such as Node.js and several other web browsers. For current online applications that extensively rely on client-side scripting, V8 is built to swiftly execute JavaScript code. The engine combines many methods, including just-in-time (JIT) compilation, concealed class transitions, and aggressive memory management, to accomplish this speed. V8 is a great option for contemporary web development because, like JavaScriptCore, it also supports the most recent iterations of ECMAScript. It is also an engine for WebAssembly and written in C++. [49, 51]

V8 engine compiles and executes JavaScript code into native machine code just like any other JavaScript engine. To do so, it consists of few main components namely the parser, Ignition and TurboFan. Ignition and TurboFan is just a different name of the interpreter and optimisation compiler[15, 14], similar to what we have seen in Figure 5, respectively. It also incorporates a profiler to gather information about different aspects during execution that is used by the compilers to optimize codes further. [4]

On the road to generate byte-code and machine code, V8 starts by parsing the given JavaScript code using it's own parser. The parser takes the code as input and parses it into an abstract syntax tree (AST). The AST, similar to what we have seen in Figure 6, is a syntactic tree structure that represents the JavaScript code. Each line of the source code is transformed to AST and the AST is passed to the optimizing architecture starting with the interpreter AKA Ignition [4]. Figure 9 illustrates an overview of the architecture.

The interpreter in V8 is called Ignition, As we already know, is in charge of converting JavaScript source code into byte-code. The machine-code, that has been

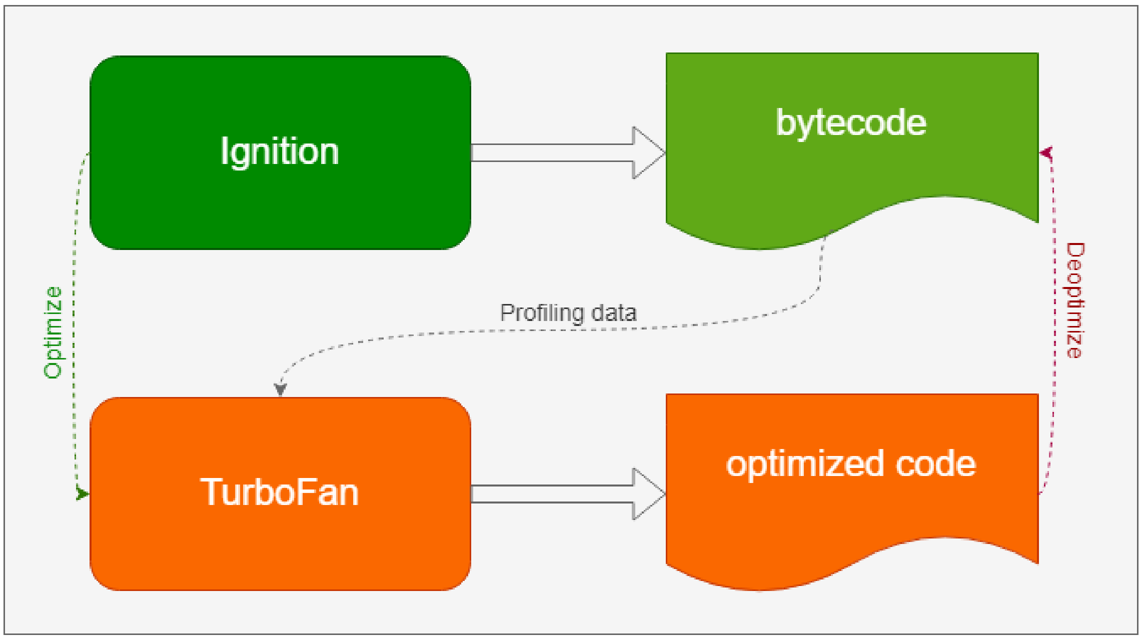

Figure 9 *V8 optimisation architecture.*

outcome of a JIT compiler, could occupy a significant portion of the available memory. Ignition was designed to mitigate, to some extent, this memory consumption and the resultant machine-code is now about 25-50 percent. Ignition reads and processes the JavaScript code, which is at this point a representation of AST, loads it into the V8 engine to produce a byte-code representation. This byte-code is then saved in memory for later use and optimized for engine execution. The byte-code is executed by a high performance interpreter which allows faster execution speed. Ignition interprets the byte-code during execution and transpiles it into machine code that the central processing unit (CPU) may use and along the path Ignition collects feedback, by profiling, which contributes to optimizations later on. [4, 15]

TurboFan, the JIT compiler in V8, optimizes code further when necessary. The optimizations are performed based on feedback collected by the interpreter, Ignition. TurboFan accepts these as input along with the profiling information and feedback produced by Ignition for code that has been executed several times in the same context. Code created by TurboFan is at the machine-level and executed by the CPU. TurboFan works based on the feedback and has to make assumptions based on the collected data from previous steps in order to carry out the optimization. If it turns out that some assumptions were incorrect for some specific portion of code, it is then sent back to the interpreter and executions resumes from there. Similar to what we have seen in Figure 4, This process is called deoptimization. This often costs memory and sometimes may also lead to slower execution speed. [4, 14]

This chapter provided us with a good overview, illustrated in Figure 10, of how V8 works under the hood. It is transparent that V8 utilizes Ignition and TurboFan

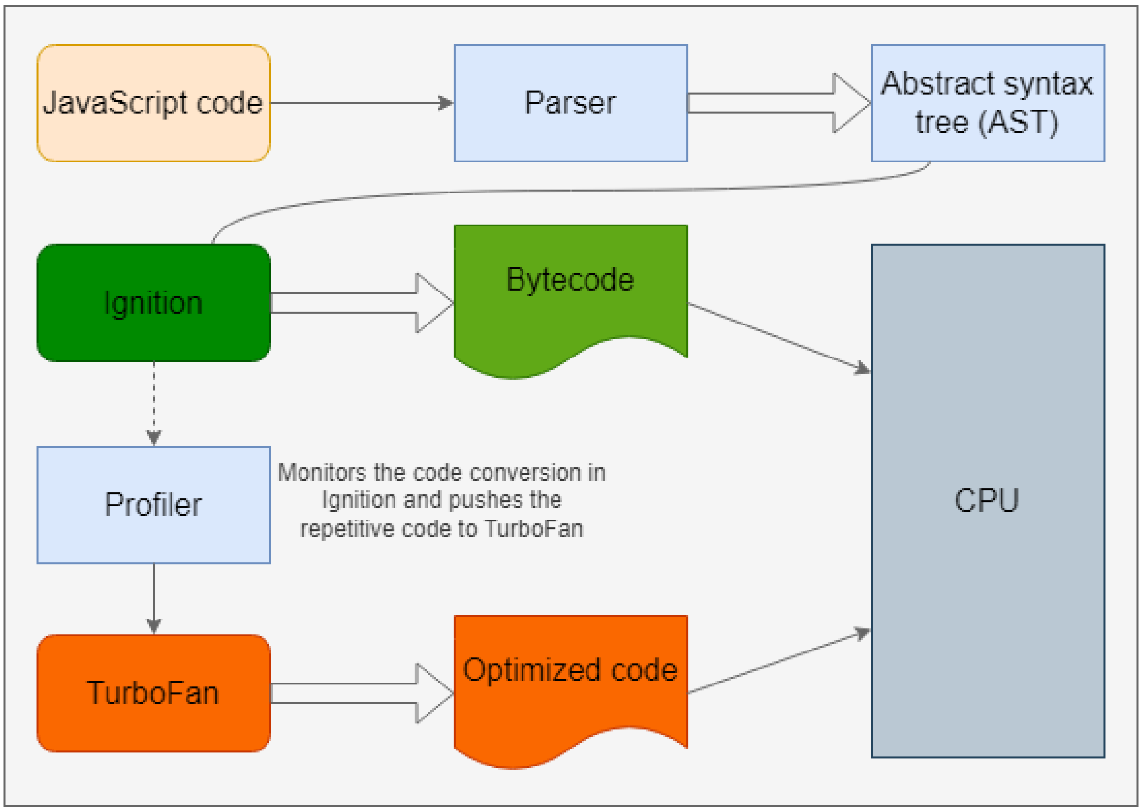
JavaScript code
Parser
Abstract syntax tree (AST)
Ignition
Bytecode
CPU
Profiler
Monitors the code conversion in Ignition and pushes the repetitive code to TurboFan
TurboFan
Optimized code

Figure 10 *V8 architecture overview.*

to achieve a quicker execution time. However, such performance is achieved with a memory cost.

2.4 Node.js

Node.js is a popular tool for a variety of projects since it is an open-source, cross-platform JavaScript runtime environment that allows writing JavaScript code outside of the browsers. It utilizes the same V8 JavaScript engine that powers Google Chrome. Node.js can attain excellent performance levels as a result [25]. Npm, the Node.js packaged ecosystem, hosts the largest collection of open source libraries in the world. Large companies such as Netflix, Uber and Walmart has chosen Node.js into one of their technological stack due to it's unrivaled performance [29].

When utilized in a browser environment, JavaScript used to have a constrained feature set. It was primarily employed for simple tasks like editing a web page's URL, adding click events, or altering its design. With the introduction of Node.js, programmers can use JavaScript for more difficult tasks that were previously only possible with Java, Python, or PHP. JavaScript can now be used by developers to establish web servers, query databases, and manage the file system with Node. These features, that were not previously possible, have significantly increased JavaScript's potential as a programming language due to Node. [29]

Despite the fact that JavaScript is single-threaded, and Node.js constrained in it, Node.js is known for its asynchronous nature. Node.js achieves that with its event-driven, non-blocking I/O model which also makes it lightweight and efficient. This allows it to continue running when responses from I/O operations, such as reading from the network, contacting databases, or accessing the file system, are available. This prevents thread blocking and the wastage of CPU cycles and enables it to handle thousands of concurrent connections without adding the complexity of managing thread concurrency that may lead to bugs [25, 29]. Figure 11 illustrates an overview of the event-driven architecture of Node.js.

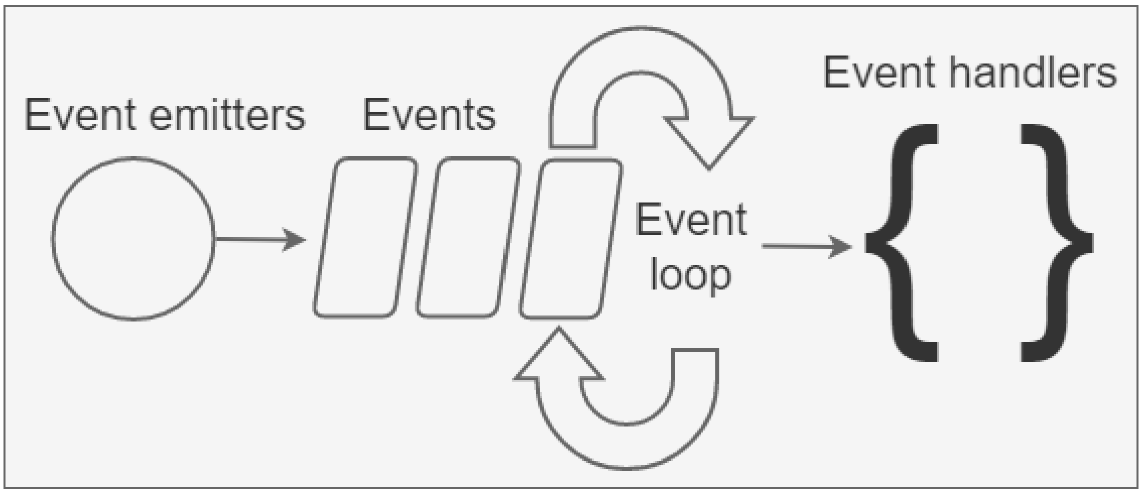

Figure 11 *Event-driven architecture of Node.js.*

Events are emitted and listeners are registered for them in Node.js' event-driven architecture. All of the registered listeners are alerted when an event happens, and they can then carry out the relevant callback procedures. This enables the efficient handling of I/O activities as well as asynchronous, non-blocking code execution. Using a non-blocking I/O architecture, Node.js registers a callback function and keeps running other code until it receives a response from an I/O operation, as opposed to blocking a thread while doing so. As a result, Node.js can manage numerous connections at once without having to create threads for each one. Code execution is not halted while waiting for I/O operations to finish because Node.js is built with an asynchronous architecture. Instead, Node.js manages asynchronous code and makes sure that the right code is performed at the right time by using callbacks, promises, and async/await. Node.js may achieve excellent performance and scalability thanks to this asynchronous architecture while avoiding common issues brought on by blocking I/O. [22]

2.5 Bun

Bun.js is a, open-source, complete toolkit for JavaScript that smoothly combines a number of important server-side JavaScript components to offer a high-performance solution. Bun integrates the features of several tools, including a runtime, at its heart, like Node or Deno, a package manager like NPM or pnpm, and a build tool like Webpack or Vite, despite its humorous moniker. Initially a one-person side project, Bun has swiftly established itself as a competitive alternative to conventional web development techniques. [50, 34]

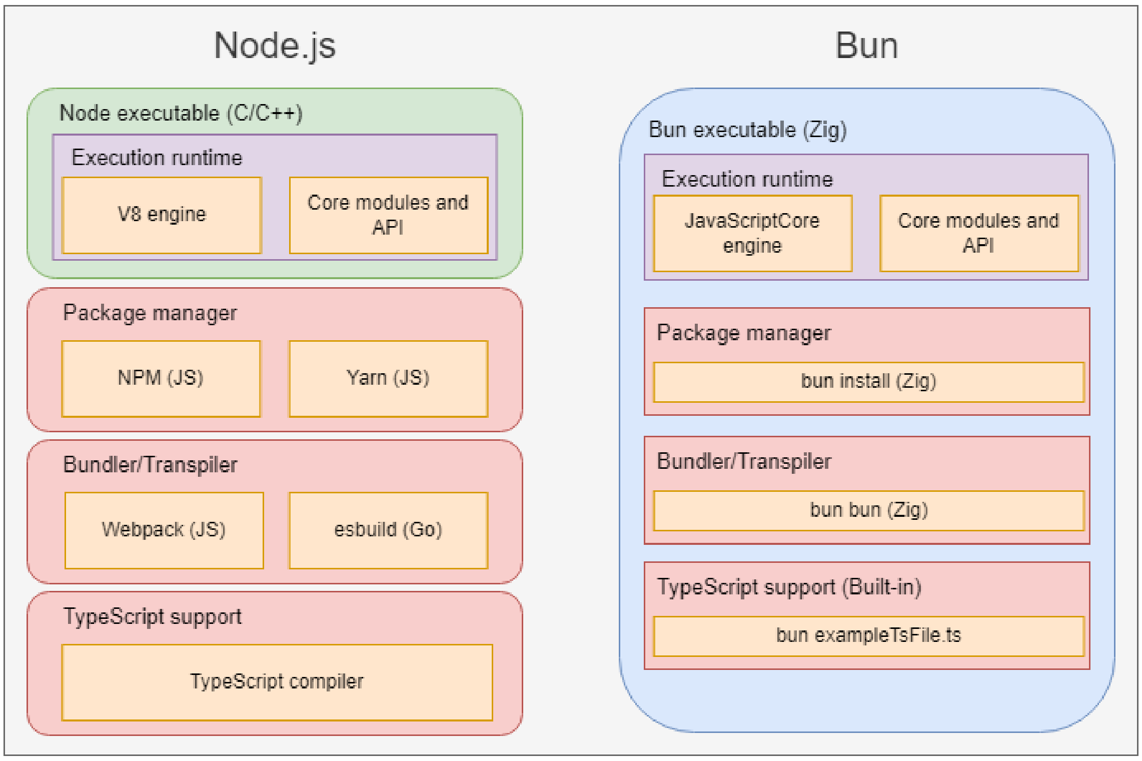

Figure 12 *Bun ecosystem compared to an ecosystem with Node.js.*

Bun is the newest player in the context of JavaScript runtimes and still under development. The aim of Bun is ambitious which is to become the drop-in replacement of the current JavaScript ecosystem including runtime. Node.js, discussed in Section 2.4, is a JavaScript runtime that is currently widely popular for writing JavaScript code outside of browsers. Bun has already implemented optimized version many of Node.js's native modules and standard web APIs such as fetch, WebSocket and ReadableStream. With support for Node-style module resolution and integrated Node.js globals and modules like process, Buffer, path, fs, and http, Bun.js aspires for seamless integration with Node.js. Bun aims towards full compatibility with Node.js, despite the fact that this is still an active project and compatibility is not yet complete. To achieve execution of TypeScript and JSX, today, separate tools/transpilers are needed. The aim of Bun is to be the all-in-one tool so that the support for such features will be available natively, reducing dependencies a software. With little to no changes, existing Node.js projects can use the quicker test runner, script runner, and package management offered by the bun CLI tool. Figure 12 illustrates a side by side comparison between ecosystem of Node.js and Bun. [34]

Designed with the today's JavaScript ecosystem in mind and written using Zig programming language, which is not only a C/C++ like lower level language but also a package manager and build tool, that provides excellent memory management APIs, Bun is fast and memory efficient. Unlike Node.js and Deno, Bun incorporates Webkits JavaScriptCore engine, discussed in Subsection 2.3.1, which is complex but fast in executing code along with minimal memory usage. So, naturally Bun is very quick at startup with some additional optimizations in its own design. This is extremely useful in serverless deployments where it helps achieve scalability by spinning up nodes quickly. This ability denotes that Bun would be well-suited for edge and serverless computing. While implementing, extensive profiling and optimizations were done in order to improve the efficiency of the performance sensitive APIs such as buffer, fetch, and response APIs. Bun claims to be 4 times faster than Node.js [50, 34], which we will evaluate in Section 5.1.

2.6 Influencing factors

One of the purpose of the research topic is to investigate and identify the factors that affects performance of runtimes. In this case, the comparison is between Bun, powered by JavaScriptCore, and Node.js, powered by V8, both of which have sophisticated optimization architectures. The goal is to pinpoint the essential components that Bun's better performance and memory efficiency are due to.

The JavaScript engine used at runtime is one important component that has an impact on performance [34]. The use of the JavaScriptCore engine in Bun is essential in this situation. Bun can produce highly efficient bytecode and machine code due to the sophisticated optimization architecture of the JavaScriptCore engine, which consists of three just-in-time (JIT) compilers and an interpreter. Multiple JIT compilers enable substantial optimization, enhancing the Bun's overall performance and speed of execution. Critical code segments are optimized by the speedy JavaScriptCore compilation process, allowing the runtime to parse JavaScript code more quickly than Node.js.

Additionally, the choice of Zig as the programming language for the development of Bun adds to its performance advantage. A lower-level programming language similar to C/C++, Zig offers effective memory management APIs. Bun may implement effective memory management techniques, such as manual memory allocation and de-allocation [34, 37], which can greatly improve memory efficiency and reduce overhead, due to the use of Zig. Furthermore, Bun's exceptional performance was greatly aided by the extensive profiling, benchmarking, and optimizations that went into it. The resultant data allowed the development team to make Bun achieve such performance with Zig that has lack of hidden control flow with the excellent control over memory management. [34]

3 Performance of JavaScript runtimes

The performance of any runtime is a critical aspect when developing software applications. This chapter examines numerous traits and measurements to understand the effectiveness and responsiveness of JavaScript runtimes as it delves into the complexities of performance as a software quality attribute. Developers may create reliable runtimes by taking performance, security, dependability, and stability into account. Runtime behavior can be understood through the analysis of timing, resource use, latency, throughput, and capacity. In order to locate bottlenecks and improve runtime performance, quantifiable performance indicators like response time, processing time, throughput rate, and resource consumption are useful. This chapter gives us the information and resources needed to test and measure JavaScript runtime performance.

3.1 Software Quality Attributes

In the realm of software development, it is the responsibility of the developers to allot suitable resources, such as processors and communication networks, in addition to identifying and implementing the software required to satisfy the application's needs. For critical systems, only meeting functional requirements is deemed insufficient because they are required to adhere to strict criteria of software quality attributes such as performance, security, safety, dependability, and other related aspects. Thus, the comprehensive fulfillment of these complex requirements becomes crucial for the creation of essential systems. [5]

To understand what software quality means, let us understand what quality itself refers to. Quality can be defined in different ways and the definition of quality may differ depending on perspectives. Quality can generally be described by a variety of characteristics, including dependability, usability, efficiency, safety, and maintainability, among others. Software's quality can be defined as its capacity to satisfy both functional and non-functional requirements, as well as the demands and expectations of its users. Non-functional requirements, commonly referred to as quality attributes, outline the standards for assessing the system's overall performance rather than any particular actions [3].

A software product's quality can be assessed using the quality model as a base. It lists the qualities that will be taken into account while making the assessment. Software quality describes how well a software product satisfies the needs and expectations of its users and adds value. Functionality, performance, security, maintainability, and other factors are among these needs. The quality model organizes these requirements into distinct characteristics and sub-characteristics, enabling a thorough assessment of the overall quality of the software product. [26]

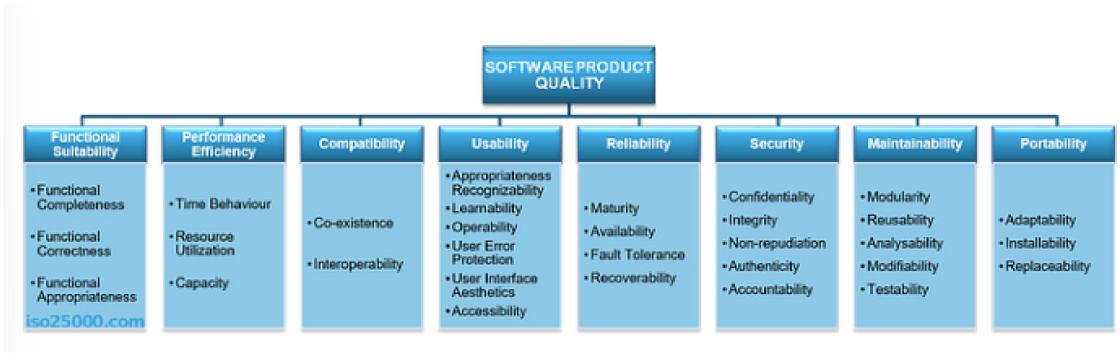

Figure 13 *ISO/IEC 25010 diagram [26].*

The ISO/IEC 25010 standard establishes a benchmark for the quality of software products. Based on eight quality characteristics functional suitability, performance efficiency, compatibility, usability, dependability, security, maintainability, and portability, it offers a thorough framework for assessing the quality of software products. The standard intends to assist businesses in assessing and enhancing the quality of their software products as well as in fostering interaction and cooperation between the many parties engaged in the software development process [26].

3.2 Performance efficiency

Similar to software quality attributes, discussed in Section 3.1, performance can also be defined in different ways. The definition given in the IEEE Standard Glossary of Software Engineering Terminology is [21]:

> The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage.

Performance, as a software quality attribute, refers to the timeliness aspects of how software systems behave. Smith's definition of performance is [44]:

> "Performance refers to responsiveness: either the time required to respond to specific events or the number of events processed in a given interval of time

Performance is that attribute of a computer system that characterizes the timeliness of the service delivered by the system.

A general misconception about performance is that it equates to speed in software engineering, believing that improving hardware elements like CPUs or communication links will result in a performance increase. This, however, ignores the fact that matching the precise timing needs of each service is necessary to achieve timeliness in many systems, notably real-time systems [46]. Hardware features like caching and pipe-lining can lower response times on average, but they can also provide erratic worst-case response times. As a result, enhancing performance necessitates a more complex strategy that takes into account the unique temporal restrictions of the system being designed.

With all that, we can say that performance shows the response of the system to performing certain action for a certain period. The ISO/IEC 25010 provides a characteristic model to determine performance efficiency. It consists of three different ways, that opens up possibilities to assess performance of a system, including Time behaviour, Resource utilization, and Capacity. The degree to which a software product or system's response and processing times, as well as its throughput rates, satisfy the necessary specifications when carrying out its activities is referred to as the system's or product's time behavior. Resource utilization refers to the extent to which a system or product utilises the proper kinds and quantities of resources when carrying out its functions in line with the given requirements. Capacity describes the extent to which a feature of a system or product can satisfy the criteria within its maximum bounds. [26]

3.3 Performance metrics

Software performance is assessed using performance metrics. Every bit of software is created to serve a particular function. By examining how much time and resources the software is using to provide the service, performance metrics can be used to evaluate if the software is meeting user needs efficiently.

Understanding of the performance characteristics is necessary in order to choose the right performance metrics. The following sub-characteristics make up this characteristic and represents the performance relative to the amount of resources used under stated conditions [26]:

- Time behaviour
- Resource utilization
- Capacity

The degree to which a system complies with the necessary specifications for response and processing times, as well as throughput rates, while carrying out its functions is referred to as time behavior. It focuses on the system's capacity to react quickly and keep its processing speed at its optimal level in order to fulfill the intended performance targets [26, 32]. In this context, performance concerns or requirements can be further specified to evaluate the system's performance by latency and throughput. The amount of time it takes for a system to respond to an event is referred to as latency. It establishes a response window with minimum and maximum latency values that can be either absolute times or offsets from a given event. The quantity of event replies that are finished during a specific observation interval is known as throughput. It is measured as the number of events processed per unit of time and shows the system's processing rate. Key metrics of time behaviour includes response times, processing time, and throughput rate [5, 32].

Resource utilization refers to how efficiently a system makes use of the various kinds and amounts of resources needed to run its activities. It entails effectively managing resources, including CPU, memory, disk space, network bandwidth, and others, to make sure they satisfy the needs while minimizing waste or under-utilization. Key metrics includes CPU utilization, memory usage and disk space utilization. [5, 26]

The system's capacity is measured by how well it can satisfy the upper bounds of particular parameters as specified by the criteria. It determines if the system can manage rising workloads, more data sets, more users, or more transaction volumes without going over set limits. Assessing capacity makes it easier to spot potential restrictions and guarantees that the system can scale and efficiently meet growing demand. Key metrics includes scalability, maximum concurrent user and transaction processing capacity. [5, 26]

4 Research methodology

In this chapter, the research approach used to compare the performance features of two different JavaScript runtimes, known as Bun and Node.js, is presented. This chapter serves as a crucial component in addressing the research questions and provides a clear road-map for conducting the performance tests on Bun and Node.js. The following research questions are the focus of this study's aim:

- Is Bun faster than Node.js?
- What affects the performance of runtimes?
- What factors may contribute when selecting a runtime?

Empirical evidence and insights into the relative performance of the two runtimes and identify factors influencing their selection and performance characteristics was gathered. Appropriate tests were performed on Bun and Node.js, using specific JavaScript scripts made to assess various performance attributes in order to address these research objectives. Memory utilization, response time, and overall execution time are the performance characteristics that was evaluated.

The research methodology consists of three main sub-chapters: Description, Implementation, and Test setup. The Description sub-chapter provides an overview of the process of test that was carried out. The Implementation sub-chapter explains the code used in the research. The Test setup sub-chapter provides details about the machine on which the tests was performed. We will go into greater detail about each step of the research technique in the next sub-chapters. By adhering to this systematic approach, we can effectively address the research questions and contribute to the understanding of the performance aspects of different JavaScript runtimes.

4.1 Description

In order to compare their performance characteristics, Bun and Node.js was tested using specialized programs made to assess various performance attributes. These tests had the explicit objectives of calculating the memory consumption, response times, and overall execution time for both runtimes. To test network request capabilities, two different code for both runtimes, node-http.js (Program 2) and bun-http.js (Program 1), were used for the experiments. Although, a single JavaScript code can be run by both runtimes, the reason for being two different code is that Bun has it's own implementation for some specific cases [50], in this case serving a server. So, two different code was used with the runtimes own implementations to get the best out of them. To test how the runtimes handles a standalone script, a JavaScript script, find getFibNum.js in Program 8, was implemented.

Bombardier, http benchmark tool, was used for node-http.js (Program 2) and bun-http.js (Program 1). Using this tool, we could assess how well Bun's and Node.js's HTTP servers performed. In three separate scenarios, 10 million requests each with 10 concurrent connections, 100 concurrent connections, and 500 concurrent connections were made with the intention of overwhelming the servers. We may evaluate the servers' performance in terms of response time and throughput by putting them under these load circumstances. The amount of time spent, the response time (median, 95^{th} percentile, average, and maximum), and the average and maximum number of requests per second were all reported by Bombardier. Additionally, the peak memory usage of the programs was measured during execution with each runtime using the built-in ”time” command-line tool accessible in the zsh shell. As a result, we were able to learn more about how well Bun and Node.js used memory to handle heavy request loads. The highest memory utilization of the applications during the testing was determined using the ”time” command-line tool, which is built into the zsh shell. We were able to learn more about Bun's and Node.js's memory efficiency by running the applications with corresponding runtimes through the ”time” command.

Hyperfine, command-line process benchmark tool, was used for the standalone script, find getFibNum.js in Program 8. The goal of this test was to establish the typical execution time for a script that computes Fibonacci numbers using Bun and Node.js. Hyperfine was used to run the script 10 times for each runtime, which delivered accurate statistical data on the typical completion time. We could learn more about the two runtimes' relative performance in carrying out computationally expensive jobs by comparing the average completion times between the two.

Choosing a right tool is essential for testing such scenarios. Node.js based tools like autocannon is not fast enough for Bun's functions such as Bun.serve() [34]. Bombardier is written with Go programming language and it incorporates ”fasthttp” package instead of the default builtin http library of Go. This allows bombardier to perform faster [33]. Hyperfine is written in Rust programming language and also has great performance. Using these tools, empirical data on the performance characteristics of Bun and Node.js was gathered by running the customized programs. The outcomes of these tests will assist in addressing the research questions and shed light on the relative effectiveness and memory usage of the two runtimes.

4.2 Implementations

This section includes information on the benchmark tool configuration and the code snippets used to implement the performance tests. For network request performance testing, two different program was implemented with simple structure. The following are the code snippets for Bun (Program 1) and Node.js (Program 2):

```
Bun.serve({
    port: 3000,
    fetch(_) {
        return new Response('Test: Bun');
    },
});
```

Program 1 *bun-http.js*

```
const http = require('node:http');

http.createServer((_, resp) => {
    resp.writeHead(200, {
        'content-type': 'text/plain',
    });
    resp.end('Test: Node');
}).listen(3000);
```

Program 2 *node-http.js*

In order to measure the performance attributes of these programs, Bombardier http benchmark tool was used. The tool was configured according to the test specifications which included 10Millions of total requests with 10, 100 and 500 concurrent connections.

```
bombardier -c 10 -n 10000000 -l http://localhost:3000
```

Program 3 *Bombardier command for 10M request and 10 concurrent connections*

```
bombardier -c 100 -n 10000000 -l http://localhost:3000
```

Program 4 *Bombardier command for 10M request and 100 concurrent connections*

```
bombardier -c 500 -n 10000000 -l http://localhost:3000
```

Program 5 *Bombardier command for 10M request and 500 concurrent connections*

Simulation of high request loads with different concurrent connection was possible due to these configuration. This allowed gathering of the statistical data on response times, throughput, and total time taken. In order to measure peak memory usage during the high load simulation of network request, ”time”, built in command line tool in zsh shell, was utilized. It requires simply to run the process through the tool.

```
time bun bun-http.js
```

Program 6 *time command for Bun*

```
time node node-http.js
```

Program 7 *time command for Node.js*

In order to measure the performance of a standalone script for both Bun and Node.js, a simple script was implemented. The script calculates the 40th Fibonacci number. The script was executed by both Bun and Node.js to evaluate their performance in executing computationally intensive tasks.

```
function fib(n) {
    if (n <= 0) return 0;
    if (n <= 1) return 1;
    if (n <= 2) return 2;

    return fib(n - 1) + fib(n - 2);
}

console.log(fib(40))
```

Program 8 *getfibnum.js*

Hyperfine, command line benchmark tool, was employed for this task. The script was executed 10 times and average completion time was measured. We were able to assess the effectiveness of Bun and Node.js in carrying out computationally complex tasks thanks to the use of Hyperfine, which provided us with trustworthy statistical data on the average time of completion.

```
hyperfine 'bun findFibNum.js' 'node findFibNum.js'
```

Program 9 *Hyperfine command for Bun and Node.js running the Fibonacci script*

Using these tools, we were able to gather information on the two runtimes' performance characteristics and respond to the study questions about their relative performance and memory utilization thanks to their implementations and setups.

4.3 Test setup

In order to ensure a consistent and controlled testing environment a specific set of hardware and software was chosen. This allowed for reliable comparisons between Bun and Node.js in terms of performance attributes such as memory usage, response time and execution time. The tests were conducted on a Macbook Pro 13" with the following specifications:

- Processor: Intel i7, 2.8GHz Quad-Core
- Memory: 16GB 2133MHz LPDDR3
- OS: macOS Ventura 13.3.1 (a)

The iTerm2 terminal emulator's zsh shell was used to run the tests. The shell offered a dependable setting for carrying out the performance testing and acquiring the required information. The benchmark tools, with versions, utilized for the tests are follows:

- Bombardier v1.2.6
- Hyperfine v1.16.1

In addition to these benchmarking tools, the built-in "time" command-line tool was used to measure the peak memory usage of the programs during the tests. The specific version of both of the runtimes used were as follows:

- Bun: v0.6.2
- Node.js: v18.16.0 LTS

The selected test configuration made it easier to quantify the runtimes' performance traits precisely and gave useful information about how well they performed in comparison. The outcomes of this configuration help us better understand the performance characteristics of various JavaScript runtimes and address the research questions.

5 Results

This chapter presents the findings of the performance tests carried out to contrast Bun and Node.js's performance features. The outcomes were obtained by the use of specific programs, benchmark tools like Bombardier and Hyperfine, as well as the built-in ”time” command-line tool from zsh. With the aid of these tools, we were able to evaluate the two runtimes' performance in terms of memory utilization, response time, and overall execution time.

Addressing the research questions and develop a thorough grasp of the performance facets of Bun and Node.js was possible by evaluating and interpreting the test data. The next section, ”Performance Evaluation,” provides a thorough examination of the outcomes collected, and the one after that, ”Discussion,” adds more information and examines the consequences of these findings.

5.1 Performance evaluation

The performance evaluation provides a comprehensive analysis of the performance attributes of Bun and Node.js based on the conducted tests. The test results are shown in the next section, which also compares the two runtimes' efficiency in terms of peak memory utilization, processing time, response time, and requests per second.

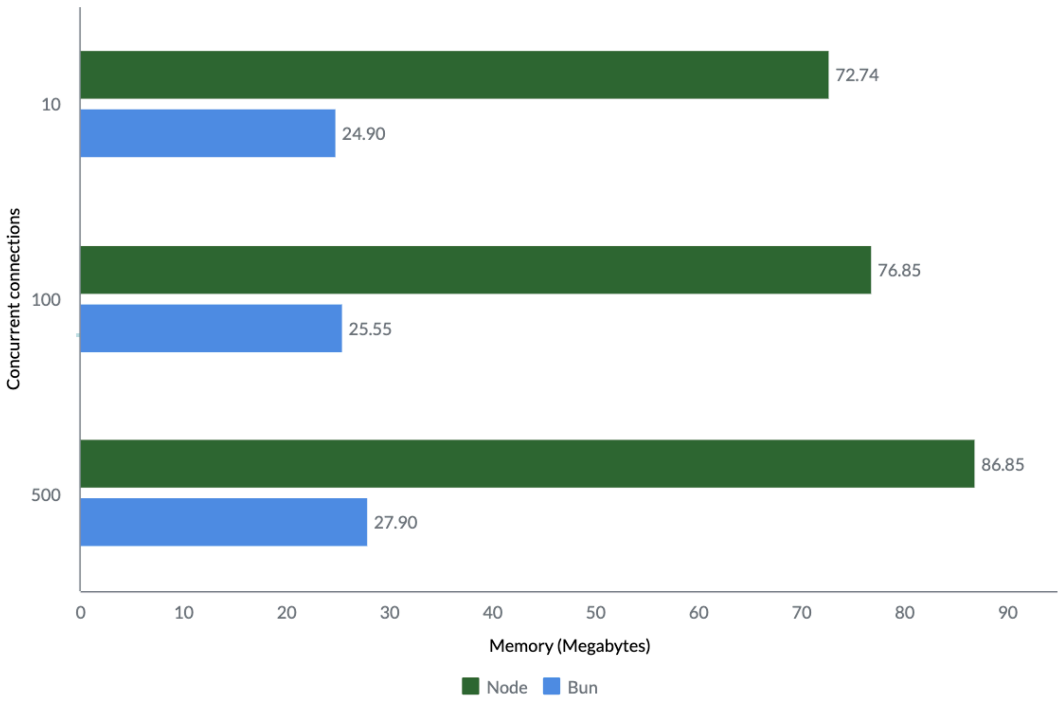

Figure 14 *Peak memory usages.*

Let us, first, take a look into the peak memory usage by the scripts when ran through both runtimes per test requirements. Peak memory use is a crucial indicator that shows how effectively the runtimes use memory. Bun consistently displayed lower peak memory consumption than Node.js in all three different load circumstances. Bun used only 24.90MB of memory for the scenario with 10 concurrent connections, but Node.js used 72.74MB, a substantial amount more. Similar to this, Bun maintained a memory advantage in the 100 and 500 concurrent connection situations, with peak usages of 25.55MB and 27.90MB, respectively, as opposed to Node.js's greater memory consumption of 76.85MB and 86.85MB. These results imply that Bun exhibits greater memory management and efficiency, which may contribute to result in enhanced performance.

This significant difference in memory usage implies that Bun's use of Zig, a lower-level programming language similar to C known for its effectiveness and great memory management API, as well as the way it was designed, helped to optimize memory usage. While using V8, Node.js, which includes a JIT compiler and an interpreter, needs more memory to handle the same workload. The improved memory economy of Bun indicates possible advantages in terms of scalability and cost-effectiveness in addition to reflecting its capacity to manage heavier workloads with constrained resources.

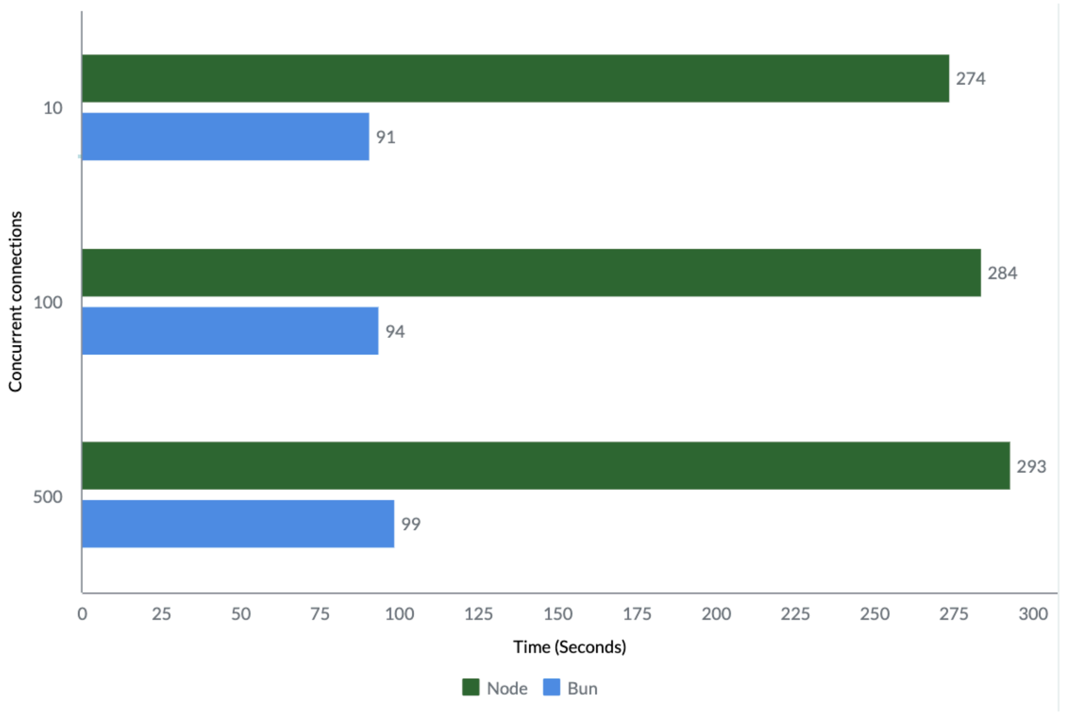

Figure 15 *Total time taken - network requests.*

Moving on to the total time taken to complete the tests, another important factor in performance evaluation, Bun consistently outperformed Node.js. This was seen across all three different load circumstances. Bun finished the test in 91 seconds while Node.js needed 274 seconds in the scenario with 10 concurrent connections. Similar to this, Bun performed better than Node.js in the case with 100 concurrent connections, finishing in 94 seconds as opposed to 284 seconds. In the case of 500 concurrent connections, Bun continued to perform better than Node.js, finishing the test in 99 seconds as opposed to 293 seconds. These outcomes illustrate Bun's capacity for speedier execution, demonstrating its effectiveness and responsiveness in managing multiple concurrent requests.

This considerable performance disparity shows that Bun's faster execution time is a result of its optimized machine code, effective interpreter, and three separate JIT compilers. Due to the interaction of these elements, Bun is able to produce code that is more highly optimized and performant, which shortens execution times and enhances responsiveness.

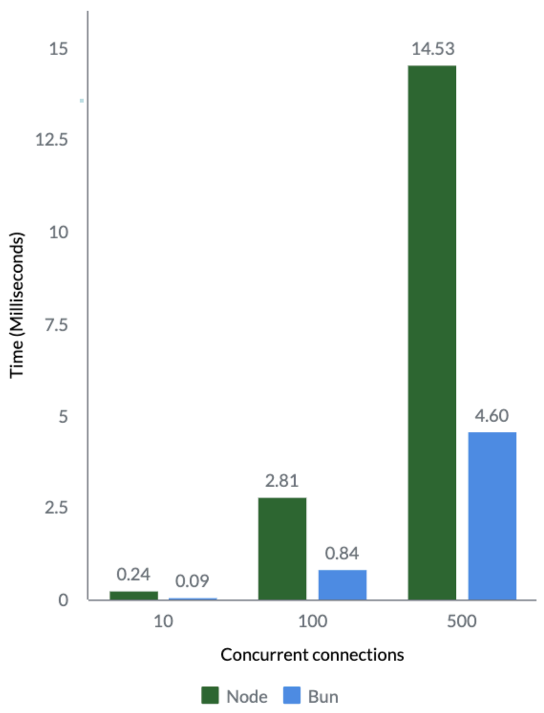

Figure 16 *Median response time.*

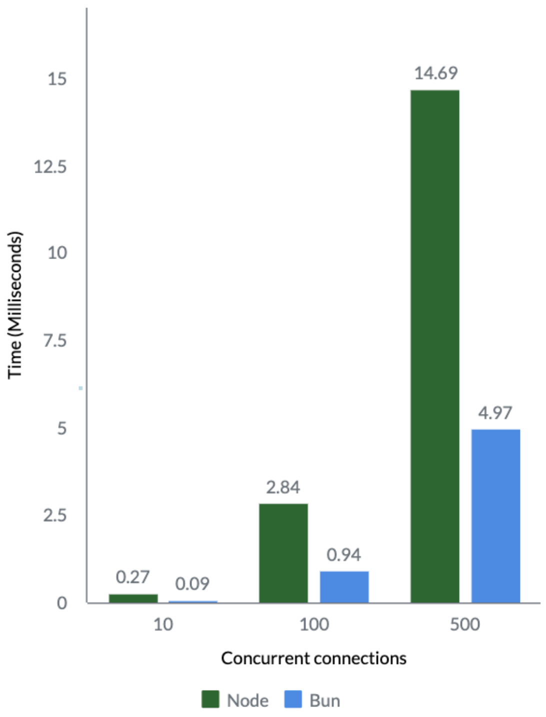

Figure 17 *95th percentile response time.*

The capacity of both runtime, to manage incoming requests and give prompt responses, is revealed by the response time metrics. Across all load conditions, Bun consistently demonstrated faster reaction times than Node.js. Bun showed noticeably lower values for the median reaction time, indicating faster response times. Similarly, Bun consistently displayed reduced 95th percentile and average reaction times, demonstrating its capacity to deliver consistent performance even under heavier loads. Additionally, Bun consistently had a lower maximum response

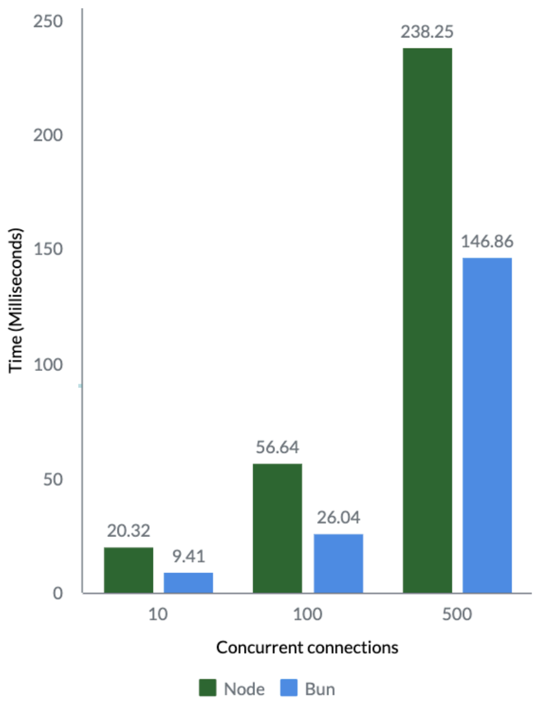

Figure 18 Median response time.

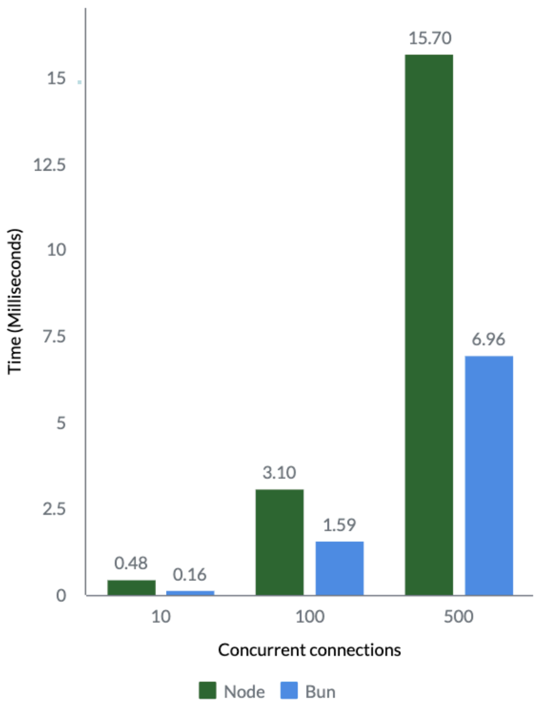

Figure 19 95th percentile response time.

time than Node.js, underscoring the fact that it is more responsive than Node.js. These results indicate that Bun performs better in handling and processing network requests overall.

Bun has faster response times than Node.js due to its effective memory management and optimized machine code, which helps it to handle and process requests quickly. These results show that Bun is more capable of handling real-time and latency-sensitive applications that need quick and reliable response times.

Metrics for requests per second gives an idea of the runtimes' throughput and capacity for large numbers of requests. Across all load circumstances, Bun performed better than Node.js in terms of average and maximum requests per second. Bun achieved better average request per second numbers, demonstrating its capacity to process more requests in a given amount of time. Similar to this, Bun showed greater maximum request per second figures, demonstrating its effectiveness in managing peak loads. These outcomes demonstrate Bun's higher throughput and its efficiency in handling several concurrent queries.

The request per second metrics provide insights into the runtimes' ability to handle a high volume of requests within a given time frame. Bun recorded an average of 98,811 requests per second in the scenario with 10 concurrent connections, while Node.js recorded an average of 36,423 requests per second. Similarly, Bun consistently showed greater average and maximum request per second numbers than Node.js in the 100 and 500 concurrent connection scenarios. These findings highlight

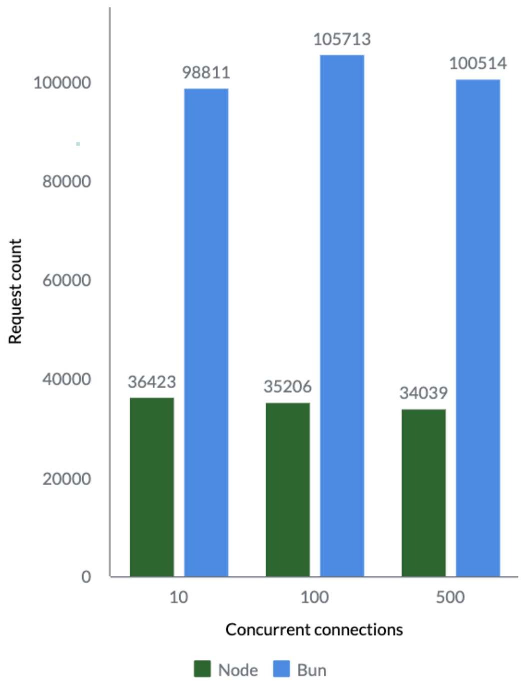

Figure 20 *Median response time.*

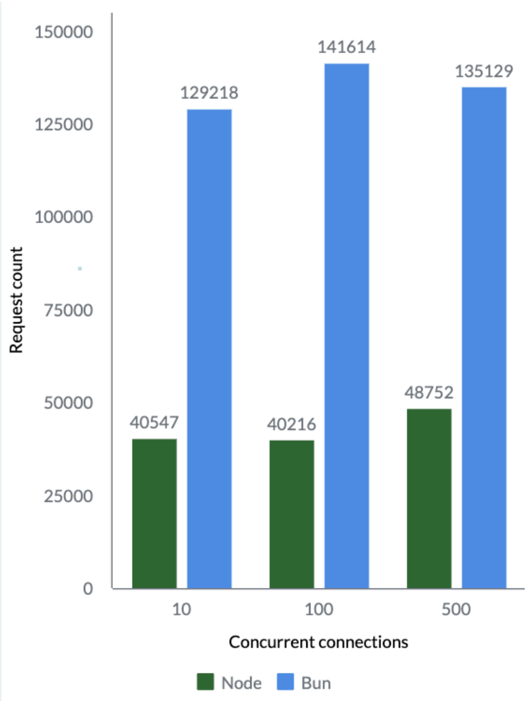

Figure 21 *95th percentile response time.*

Bun's superior speed and scalability in managing high traffic levels and point to its appropriateness for applications that demand rapid request processing.

The evaluation also included a stand-alone script test to gauge how well the 40^{th} Fibonacci number could be calculated, in addition to the network request benchmarks. In this case, Bun displayed quicker execution times, taking just 876.4 milliseconds on average to complete while Bun took 465.6 milliseconds. This result shows that Bun is more effective at completing computational tasks and produces results more quickly than Node.js.

The outcomes of the test of the standalone script emphasizes the significance of taking into account the particular workload and processing needs when choosing a runtime. Applications that rely substantially on CPU-intensive processes, including calculations or sophisticated algorithms, might greatly benefit from selecting a runtime like Bun, which specializes at carrying out such tasks quickly. Bun can offer significant speed advantages and enable quicker completion of CPU-bound processes by utilizing optimized machine code and enhanced execution techniques.

The performance evaluation demonstrates that Bun consistently outperforms Node.js in various performance aspects, including memory usage, execution time, response time, and request throughput. In terms of network requests, Bun was approximately 3 times faster than Node.js according to the test data. When running scripts, Bun exhibited a performance advantage of approximately 1.88 times over Node.js based on the test results. These findings align with the hypothesis that

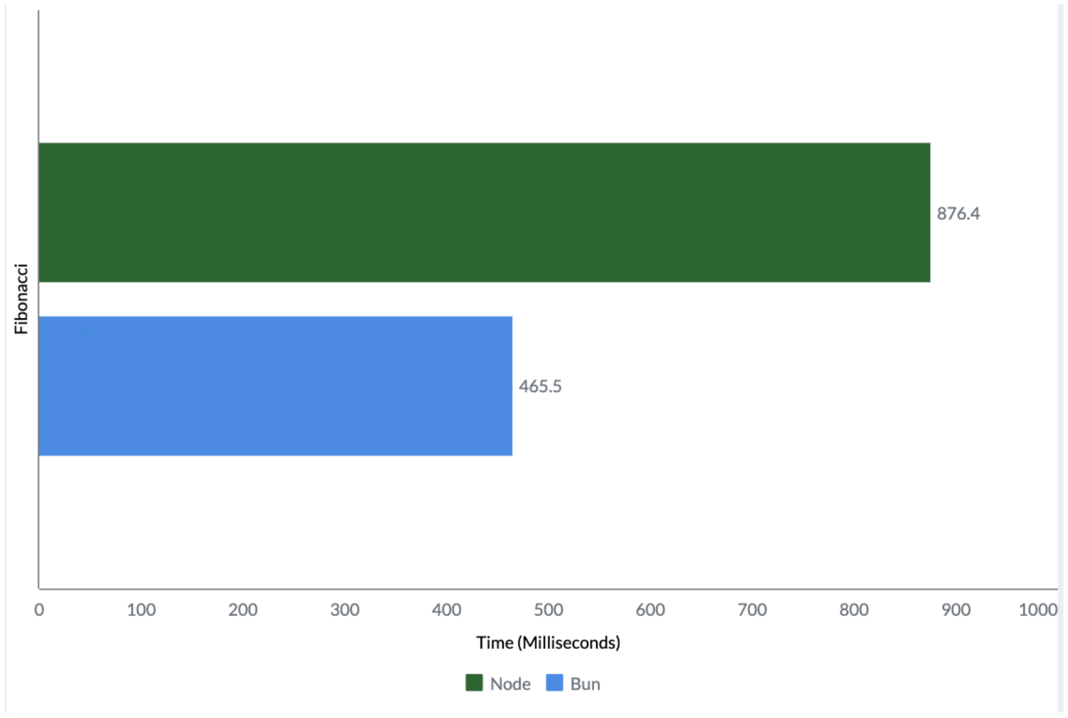

Figure 22 *Total time taken - script.*

the design and implementation choices of Bun, such as its optimized machine code from the engine of choice, specialized modules, and efficient memory management, contribute to its superior performance. The subsequent section, "Discussion," will delve deeper into the implications of these results and provide further insights into the factors influencing the observed performance differences.

5.2 Discussion

This section aims to address the research questions formulated earlier in the thesis by analyzing and interpreting the findings obtained from the performance evaluation tests. The comparison of Bun and Node.js's performance characteristics will be discussed, and conclusions based on the test findings will be drawn. The discussion will also identify topics for further investigation and acknowledge the research's shortcomings.

Based on the outcomes of the performance evaluation, it can be said that Bun performs better than Node.js in terms of performance. Bun regularly beat Node.js in terms of network requests, showing almost three times faster response times and better request throughput. Similarly, while executing scripts, Bun had a speed advantage over Node.js of almost 1.88 times. These results support the claim that Bun is quicker than Node.js in above mentioned scenarios.

The performance evaluation tests brought crucial factors, to light, that should be considered while selecting a runtime. The outcomes highlight the value of efficient interpreters, just-in-time (JIT) compilation approaches, and optimized machine code production in the underlying engine. Due to the engines complex design, which contains a productive interpreter and numerous extremely effective JIT compilers, Bun offers noticeable speed advantages. By choosing a suitable programming language, such as Zig in the case of Bun , performance can also be improved. These findings show that factors like engine design, memory management, and programming language selection have a significant impact on runtime performance. These factors should be taken into account when selecting a runtime for applications with particular performance requirements. However, The choice of runtime cannot be restricted by performance requirements. Security, stability, and reliability are just a few of the qualities of high-quality software that are equally crucial.

Based on the findings of the performance evaluation, it can be said that Bun provides better performance than Node.js in terms of making network requests and running scripts. Bun performs better due to its improved design, effective interpreters, and use of the Zig programming language.

5.3 Limitations

It is important to recognize the limits of the research that has been done. The evaluation may not have included all situations or workload variations because it concentrated on particular performance factors like response times, throughput, and execution speed. The tests were performed on a particular system, thus they might not accurately reflect the performance traits on other hardware setups. Additionally, because only Bun and Node.js were compared, it's possible that the results cannot be applied to other JavaScript runtimes or engines. Specific versions of Bun and Node.js were used for the evaluation. Performance may be affected if new optimizations and enhancements are added to runtime versions over time. It is crucial to remember that the conclusions are particular to the versions that were assessed and might not be directly transferable to upcoming or other runtime versions.

Although the performance tests were carried out in a controlled setting, runtime performance may still be affected by outside influences. The performance that was observed could be impacted by variables like system load, network congestion, or background operations. It is critical to recognize that these outside influences may have an impact on real-world runtime performance.

The performance tests' examined programs were rather straightforward and centered on a few key capabilities. Applications used in the real world frequently have code-bases that are more complex and may display various performance traits. The results might not accurately reflect the runtime behavior in complex code environments.

Addressing these limitations and acknowledging their potential impact on the findings is essential for a comprehensive and accurate understanding of the performance aspects of JavaScript runtimes.

5.4 Future works

The study opens up a number of new avenues for investigation and future development. To gain a wider perspective, it would be beneficial to look into the performance characteristics of different JavaScript runtimes and engines. A more complete knowledge of runtime performance would also benefit from taking into account various workloads and circumstances, such as CPU-bound and I/O-bound processes. Further research could benefit from examining the effects of runtime configurations, optimization methods, and memory management techniques.

It should be possible to learn more about how Bun and Node.js function in practical situations by expanding the evaluation to include benchmarks and real-world applications. Specific considerations and optimizations necessary for various use cases might be discovered by examining their performance in well-known frameworks or large-scale applications.

Executing performance tests in cloud-based settings with different resource allocations and configurations. Insights regarding the performance traits of JavaScript runtimes under various deployment scenarios, such as varied server capacities and network conditions, may be gained from doing so.

Researchers and developers may improve runtime choices, extend their understanding of the performance features of JavaScript runtimes, and contribute to the continued development of runtime technology by pursuing these future works.

6 Conclusions

In conclusion, this thesis aimed to compare the performance of different JavaScript runtimes, namely Bun and Node.js, in the context of network requests and running standalone scripts. We learned important things about how different runtimes compare in terms of performance through a series of tests and evaluations.

Across a range of concurrency levels, Bun consistently performed better than Node.js in terms of network requests. The testing showed that Bun outperformed Node.js in terms of response times, request throughput, and memory use by about three times, respectively. This shows that Bun is more effective at handling network requests and may offer better performance for web applications that depend on network interactions. Bun performed 1.88 times better than Node.js when it came to running independent scripts. The testing revealed that Bun executed computationally intensive jobs more quickly, demonstrating its effectiveness. As a result, it seems that Bun would be more appropriate in situations that call for quick script execution, including data processing or mathematical calculations. Bun could also be valuable in the scenario of serverless architecture where functions would spin up and down based on demand.

Based on these results, it can be said that Bun performs better than Node.js in terms of network requests and the execution of independent scripts. This information can be used by developers and system architects to choose the best JavaScript runtime for their unique use cases while taking the performance needs of their apps into account.

However, it is crucial to recognize the limits of this research. The performance assessments were not done in all situations and application domains because they were done under specific test settings. The workload's characteristics, hardware setups, and other variables may affect the outcomes. There may be more JavaScript runtimes on the market with differing performance characteristics, and the comparison was only between Bun and Node.js.

Future research should broaden the study to take into account more JavaScript runtimes and conduct more thorough benchmark using a variety of application situations including real-world applications. The effects of runtime settings, memory management techniques, and concurrency models on performance could be the subject of further investigation. The compatibility and interoperability of these runtimes with well-known JavaScript libraries and frameworks would also offer developers useful information.

In summary, this research advances knowledge of JavaScript runtime performance and emphasizes Bun's benefits in terms of network requests and script execution. The performance of JavaScript applications can be improved by taking into account the research findings and addressing the identified limits, which will increase user experiences and boost system efficiency in general.

References

[1] “Adobe announces the end of flash; highlights WebGL”. In: (2017). Accessed: 10.06.2023. Available: https://www.khronos.org/blog/adobe-announces-the-end-of-flash-highlights-webgl.

[2] Marc Andreessen. “Innovators of the net: Brendan Eich and JavaScript”. In: (1998). Accessed: 11.03.2023. Available: http://web.archive.org/web/20080208124612/http:/wp.netscape.com/comprod/columns/techvision/innovators_be.html.

[3] Nikolay Ashanin. “Quality attributes in software architecture”. In: *medium.com* (2018). Accessed: 12.04.2023. Available: https://medium.com/@nvashanin/quality-attributes-in-software-architecture-3844ea482732.

[4] Edison Augusthy. “Deep dive into JavaScript engine - (Chrome V8)”. In: *dev.to* (2020). Accessed: 07.04.2023. Available: https://dev.to/edisonpappi/how-javascript-engines-chrome-v8-works-50if.

[5] Mario Barbacci, Mark Klein, Thomas Longstaff, and Charles Weinstock. “Quality Attributes”. In: (1995). Accessed: 04.06.2023. Available: https://www.researchgate.net/publication/242437986_Quality_Attributes.

[6] Lin Clark. “A crash course in just-in-time (JIT) compilers”. In: *hacks.mozilla.org* (2017). Accessed: 08.04.2023. Available: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/.

[7] Mozzila Corporation. *Client-side web APIs*. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs. Accessed: 25.03.2023. 2023.

[8] Mozzila Corporation. *The event loop*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop. Accessed: 25.03.2023. 2023.

[9] Gemma Croad. “Understanding the JavaScript runtime environment”. In: *medium.com* (2020). Accessed: 25.03.2023. Available: https://medium.com/@gemma.croad/understanding-the-javascript-runtime-environment-4dd8f52f6fca.

[10] Ian Deed. “Pros and cons of JavaScript development”. In: *www.pangea.ai* (2023). Accessed: 19.03.2023. Available: https://www.pangea.ai/dev-javascript-resources/best-practices/.

[11] *ECMA-262, 1st edition*. Accessed: 11.03.2023. Available: https://www.ecma-international.org/publications-and-standards/standards/ecma-262/. European Computer Manufacturers Association. 1997.

[12] *ECMA-262, 2nd edition.* Accessed: 12.03.2023. Available: `https://www.ecma-international.org/publications-and-standards/standards/ecma-262/`. European Computer Manufacturers Association. 1998.

[13] *ECMA-262, 3rd edition.* Accessed: 12.03.2023. Available: `https://www.ecma-international.org/publications-and-standards/standards/ecma-262/`. European Computer Manufacturers Association. 1999.

[14] *Firing up the ignition interpreter.* Accessed: 07.04.2023. Available: `https://v8.dev/blog/turbofan-jit`. Google. 2015.

[15] *Firing up the ignition interpreter.* Accessed: 07.04.2023. Available: `https://v8.dev/blog/ignition-interpreter`. Google. 2016.

[16] David Flanagan. *JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language.* Accessed: 11.03.2023. Available: `https://learning.oreilly.com/library/view/javascript-the-definitive/9781491952016/ch01.html`. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc, 2020.

[17] Jesse James Garrett. "AJAX: a new approach to web applications". In: *adaptivepath.org* (2005). Accessed: 12.03.2023. Available: `https://web.archive.org/web/20190226075734/https:/adaptivepath.org/ideas/ajax-new-approach-web-applications/`.

[18] Alireza Hamid. "JavaScript engine, a true story (Part 1)". In: *dev.to* (2022). Accessed: 26.03.2023. Available: `https://dev.to/alirezahamid/javascript-engine-a-true-story-part-1-1hp1`.

[19] Malik Haziq. "The limitations of JavaScript as a programming language". In: *dev.to* (2023). Accessed: 19.03.2023. Available: `https://dev.to/malikhaziq/the-limitations-of-javascript-as-a-programming-language-2fd7`.

[20] *History of JavaScript.* `https://www.javascriptinstitute.org/javascript-tutorial/history-of-javascript/`. Accessed: 11.03.2023. 2015.

[21] IEEE. "610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Standard Glossary of Software Engineering Terminology* (1993). Accessed: 15.04.2023. Available: `https://ieeexplore.ieee.org/document/238572`.

[22] David Ikukoyi. "Understanding the NodeJS architecture". In: *www.turing.com* (2023). Accessed: 08.04.2023. Available: `https://www.turing.com/kb/understanding-the-nodejs-architecture`.

[23] "Industry leaders to advance standardization of Netscape's JavaScript at standards body meeting". In: *Press release* (1996). Accessed: 12.03.2023. Available: `https://web.archive.org/web/19981203070212/http:/cgi.netscape.com/newsref/pr/newsrelease289.html`.

[24] *Introduction to JavaScript runtime environments.* `https://algodaily.com/lessons/introduction-to-js-engines-and-runtimes`. Accessed: 12.03.2023. 2023.

[25] *Introduction to Node.js.* Accessed: 08.04.2023. Available: `https://nodejs.dev/en/learn/introduction-to-nodejs/`. OpenJS Foundation. 2023.

[26] *ISO/IEC 25010.* Accessed: 12.04.2023. Available: `https://iso25000.com/index.php/en/iso-25000-standards/iso-25010`. ISO 25000. 2023.

[27] Kai. "JavaScript is everywhere". In: *dev.to* (2021). Accessed: 19.03.2023. Available: `https://dev.to/kais_blog/javascript-is-everywhere-5fo0`.

[28] Jen Looper. "A guide to JavaScript engines for idiots". In: *developer.telerik.com* (2015). Accessed: 26.03.2023. Available: `https://web.archive.org/web/20181208123231/http://developer.telerik.com/featured/a-guide-to-javascript-engines-for-idiots/`.

[29] Andrew Mead. *Learning Node.js development.* Accessed: 25.03.2023. Available: `https://learning.oreilly.com/library/view/learning-node-js-development/9781788395540/b34c7f1b-8ae4-4974-a24b-6051605af3c7.xhtml`. 35 Livery Street Birmingham B3 2PB: Packt Publishing, 2018. Chap. Basics of Asynchronous Programming in Node.js.

[30] Svetlin Nakov and Team. *Programming Basics with C#.* Accessed: 21.03.2023. Available: `https://csharp-book.softuni.org/Content/Chapter-1-first-steps-in-programming/how-to-write-console-app/runtime-environments.html`. Faber Publishing, Sofia, 2019. Chap. Runtime environments, low-level and high-level languages.

[31] "Netscape and Sun announce JavaScript, the open, cross-platform object scripting language for enterprise networks and the internet". In: *Press release* (1995). Accessed: 11.03.2023. Available: `https://web.archive.org/web/20070916144913/http:/wp.netscape.com/newsref/pr/newsrelease67.html`.

[32] University of New Brunswick. "Understanding quality attributes". In: (2023). Accessed: 04.06.2023. Available: `https://www.cs.unb.ca/~wdu/cs6075w10/sa2.htm`.

[33] *Official documentation of Bombardier.* Accessed: 08.04.2023. Available: `https://pkg.go.dev/github.com/codesenberg/bombardie`. 2023.

[34] *Official documentation of Bun.* Accessed: 08.04.2023. Available: `https://bun.sh/docs`. Oven. 2023.

[35] *Official documentation of Electron.* Accessed: 10.06.2023. Available: `https://www.electronjs.org`. OpenJS. 2023.

[36] *Official documentation of React Native.* Accessed: 10.06.2023. Available: `https://reactnative.dev`. Meta. 2023.

[37] *Official documentation of Zig.* Accessed: 10.06.2023. Available: `https://ziglang.org`. Ziglang. 2023.

[38] Adeyefa Oluwatoba. “JavaScript everywhere — web, mobile and desktop”. In: *medium.com* (2019). Accessed: 12.03.2023. Available: `https://sainttobs.medium.com/javascript-everywhere-web-mobile-and-desktop-68131878d22d`.

[39] Kassandra Perch. *Hands-On Robotics with JavaScript.* Accessed: 10.06.2023. Available: `https://learning.oreilly.com/library/view/hands-on-robotics-with/9781789342055`. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O’Reilly Media, Inc, 2018.

[40] Sebastian Peyrott. “A brief history of JavaScript”. In: *auth0.com* (2017). Accessed: 12.03.2023. Available: `https://auth0.com/blog/a-brief-history-of-javascript/`.

[41] Filip Pizlo. “Introducing the B3 JIT Compiler”. In: *webkit.org* (2016). Accessed: 28.03.2023. Available: `https://webkit.org/blog/5852/introducing-the-b3-jit-compiler`.

[42] Filip Pizlo. “Introducing the WebKit FTL JIT”. In: *webkit.org* (2014). Accessed: 28.03.2023. Available: `https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit`.

[43] *Server-side JavaScript: back with a vengeance.* `https://readwrite.com/server-side_javascript_back_with_a_vengeance/`. Accessed: 11.03.2023. 2009.

[44] C.U. Smith and L.G. Williams. “Software performance engineering: a case study including performance comparison with design alternatives”. In: *IEEE Transactions on Software Engineering* (1993). Accessed: 15.04.2023. Available: `https://ieeexplore.ieee.org/document/238572`.

[45] *Stackoverflow developer survey 2022.* `https://survey.stackoverflow.co/2022/`. Accessed: 12.03.2023. 2022.

[46] J.A. Stankovic. “Misconceptions about real-time computing: a serious problem for next-generation systems”. In: *IEEE Computer* (1998). Accessed: 15.04.2023. Available: `https://ieeexplore-ieee-org.libproxy.tuni.fi/document/7053`.

[47] Antero Taivalsaari, Tommi Mikkonen, Matti Anttonen, and Arto Salminen. “The death of binary software: end user software moves to the web”. In: *2011 Ninth International Conference on Creating, Connecting and Collaborating through Computing*. Accessed: 12.03.2023. Available: `https://ieeexplore-ieee-org.libproxy.tuni.fi/document/5936687`. Kyoto, Japan: IEEE, 2011, Section III(A).

[48] Codecademy Team. “Introduction to JavaScript runtime environments”. In: *codecademy.com* (2023). Accessed: 19.03.2023. Available: `https://www.codecademy.com/article/introduction-to-javascript-runtime-environments`.

[49] *The V8 JavaScript engine*. Accessed: 28.03.2023. Available: `https://nodejs.dev/en/learn/the-v8-javascript-engine`. OpenJS Foundation. 2023.

[50] Matthew Tyson. “Explore Bun.js: The all-in-one JavaScript runtime”. In: *InfoWorld.com* (2023). Accessed: 19.03.2023. Available: `https://www.proquest.com/docview/2779153875`.

[51] *V8 official documentation*. Accessed: 07.04.2023. Available: `https://v8.dev/docs`. Google. 2023.

[52] Webkit. *FTL JIT*. Accessed: 28.03.2023. Available: `https://trac.webkit.org/wiki/FTLJIT`. Webkit. 2023.

[53] Webkit. *JavaScriptCore*. Accessed: 28.03.2023. Available: `https://trac.webkit.org/wiki/JavaScriptCore`. Webkit. 2023.