Bachelor's thesis UAS

Business Information Technology

2025

Juuso Laakso

The Next Generation of Server-Side JavaScript Runtimes

– Node.js, Deno and Bun: Evolution and Future of JavaScript Runtimes

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor’s thesis | Abstract

Turku University of Applied Sciences

Business Information Technology

2025 | Total pages 43

Juuso Laakso

The Next Generation of Server-Side JavaScript Runtimes

- Evolution and Future of JavaScript Runtimes

This thesis investigates the evolution and future direction of JavaScript runtimes, focusing on Node.js, Deno, and Bun. The motivation for this thesis stems from the growing industry's need to understand how new runtimes are challenging Node.js. This thesis examined differences in these runtimes to offer guidance to developers, educators, and businesses on adoption and long-term significance.

The study combined a literature review with practical benchmark tests, examining the history, design, and ecosystems of each runtime, and comparing their performance. The analysis also covered module systems, built-in tools, and TypeScript support, noting that performance should be evaluated alongside security, compatibility, and ecosystem maturity.

Results indicated that Node.js remains the most established and widely adopted runtime, with proven reliability and a mature ecosystem. Deno provided a more secure, modern development experience, while Bun demonstrated the best overall performance and strong compatibility with existing Node.js applications. This thesis concludes that there is no single best choice among these runtimes, as each can fit different priorities and use cases.

Keywords:

JavaScript, Node.js, Deno, Bun, Server-Side JavaScript, web development, programming

Opinnäytetyö AMK | Tiivistelmä

Turun ammattikorkeakoulu

Tietojenkäsittely

2025 | 43 sivua

Juuso Laakso

Seuraavan sukupolven JavaScript-ajoympäristöt

- JavaScript-ajoympäristöjen kehitys ja tulevaisuus

Tämän opinnäytetyön tarkoituksena oli tutkia JavaScript-ajoympäristöjen kehitystä ja tulevaisuuden suuntaa keskittyen Node.js-, Deno- ja Bun-ympäristöihin. Opinnäytetyön motivaationa oli alan kasvava tarve ymmärtää, miten uudet ajoympäristöt haastavat Node.js:n pitkään jatkuneen aseman verkkokehityksen vakiintuneena ratkaisuna. Opinnäytetyössä analysoidaan näiden ajoympäristöjen eroja ja pohditaan käytännönläheisiä näkökulmia niiden soveltuvuudesta eri sidosryhmille, kuten kehittäjille, kouluttajille ja yrityksille.

Tutkimus toteutettiin kirjallisuuskatsauksen ja käytännön suorituskykytestien yhdistelmänä, jossa tarkastellaan kunkin ajoympäristön taustaa, keskeisiä suunnitteluperiaatteita ja ekosysteemin kehitystä, kun taas paikallisesti suoritetuissa testeissä verrattiin suorituskykyä. Lisäksi vertailtiin moduulijärjestelmiä, sisäänrakennettuja työkaluja ja TypeScript-tukea.

Tulokset osoittivat, että Node.js on edelleen vakiintunein ja laajimmin tuettu ajoympäristö. Deno määrittää turvallisemman ja modernimman kehitysympäristön, kun taas Bun osoittautui suorituskykyisimmäksi ja erittäin yhteensopivaksi olemassa olevan Node.js-koodin kanssa. Opinnäytetyön johtopäätöksenä todettiin, että ajoympäristöjen joukosta ei voi valita yhtä parasta vaihtoehtoa, sillä sopivin ajoympäristö riippuu eri vaatimuksista ja käyttötarkoituksesta.

Asiasanat:

JavaScript, Node.js, Deno, Bun, palvelinpuolen JavaScript, verkkokehitys, ohjelmistokehitys

Contents

Figures

Tables

Usage of AI in the thesis

This thesis work utilized Grammarly and Claude Opus 4 for language refinement, spelling correction, and structural suggestions. All text has been carefully reviewed before inclusion.

List of abbreviations

API	Application Programming Interface
ES	ECMAScript
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I/O	Input / Output
JIT	Just-in-Time
JS	JavaScript
JSR	JavaScript Registry
NPM	Node Package Manager
PNPM	Performant Node Package Manager
TTFB	Time To First Byte
URL	Uniform Resource Locator
V8	Google’s open-source JavaScript engine
YARN	Yet Another Resource Negotiatorß

1 Introduction

The landscape of server-side JavaScript development has undergone a remarkable transformation since Ryan Dahl introduced Node.js in 2009, changing how developers approach backend programming (Dahl et al., 2020). Node.js leveraged Google's V8 JavaScript engine to bring JavaScript beyond the browser, enabling developers to build scalable server applications using a single programming language across their entire technology stack. This innovation quickly revolutionized web development, with Node.js now powering hundreds of thousands of websites and becoming integral to many companies' technology infrastructure (W3Techs, 2025).

According to the State of JavaScript 2024 multiple-choice questionnaire survey with 11 576 respondents, Node.js maintains its position as the dominant JavaScript runtime, as represented in Figure 1, with the majority of developers expressing satisfaction with its capabilities and ecosystem (Greif & Burel, 2024). This widespread adoption has been driven by Node.js's robust ecosystem, extensive NPM package repository, and proven ability to handle concurrent connections efficiently through its event-driven, non-blocking I/O model.

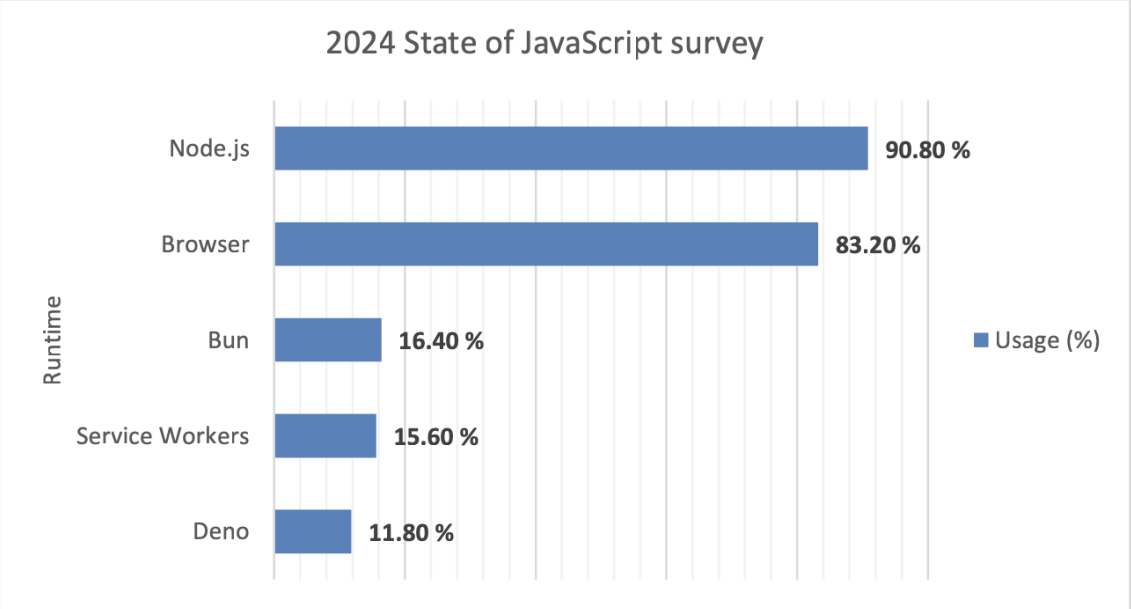

Figure 1. Usage statistics from State of JavaScript 2024 survey.

However, as web applications have grown increasingly complex and performance requirements have intensified, certain limitations of Node.js have become more apparent. Ryan Dahl himself mentions several design choices he now conceives as problematic, including security concerns, module system complexities, and

performance bottlenecks (Dahl, 2018). These acknowledgments have prompted the development community to explore alternatives that could address these shortcomings while building upon Node.js's strengths.

1.1 Problem Statement and Motivation

In response to Node.js's limitations, two alternatives have emerged: Deno and Bun. Deno, launched in 2020 by Ryan Dahl, was explicitly designed to address security and module management issues while providing first-class TypeScript support (Dahl et al., 2020). The recent release of Deno 2 has further enhanced its compatibility with the existing Node.js ecosystem (Dahl et al., 2024). Bun, introduced as a performance-focused alternative, promises speed improvements, with recent academic research demonstrating that Bun outperforms Node.js across multiple metrics, including memory usage, execution time, and request handling-capacity (Ahmod, 2023).

Despite these promising developments, comprehensive security research has revealed new challenges in alternative runtimes that should be carefully judged and examined (Alhamdan & Staicu, 2025). The choice of JavaScript runtime has implications that extend beyond technical considerations, affecting application performance, security, and long-term maintainability for businesses. Educational institutions face decisions about which technologies to include in their curricula, balancing current industry standards with emerging technologies.

1.2 Research Objectives and Scope

The objective of this thesis is to examine the current state and future direction of server-side JavaScript runtimes, focusing on Node.js, Deno, and Bun. The study aims to identify the key technical, performance, and ecosystem differences between these platforms and to evaluate their suitability for modern web development.

The scope of the research includes both theoretical and practical analysis. The theoretical part explores the background, evolution, and architectural design of each runtime, while the practical part focuses on performance benchmarking. Other factors such as module systems, TypeScript support, and developer experience are also compared to provide a better understanding of each runtime.

The aim is to give a perspective of where JavaScript runtimes stand today and how they may develop further.

2 Background and Evolution of JavaScript Runtimes

The introduction of Node.js in 2009 revolutionized web development by enabling JavaScript execution outside the browser, quickly establishing itself as the industry standard for server-side JavaScript applications (Ahmod, 2023). However, as Node.js matured, its creator Ryan Dahl had concerns over several design and security issues that motivated the development of new approaches to JavaScript runtime design (Dahl, 2018).

The current landscape shows Node.js maintaining a dominant position in the server-side JavaScript market, with widespread adoption across various industries and development teams (Usage statistics of Node.js, 2025; Greif & Burel, 2024). According to the State of JavaScript 2024 survey, Node.js remains the most widely used JavaScript runtime, though emerging alternatives like Deno and Bun are gaining traction among developers seeking modern solutions (Greif & Burel, 2024).

The emergence of these alternatives has also coincided with broader changes in web development practices, including the adoption of serverless architecture and edge computing paradigms that place different demands on runtime performance and capabilities (Vepsäläinen et al., 2025). This evolving landscape requires careful evaluation of how runtimes address both current needs and future requirements in web development.

This chapter examines the evolution of JavaScript runtimes, beginning with Node.js's rise and the specific limitations that motivated the development of alternatives. This will provide foundational context for understanding the technical, security, and business implications explored in subsequent chapters.

2.1 Rise of JavaScript on the Server

Node.js emerged in 2009 as a revolutionary solution that transformed server-side JavaScript development. Created by Ryan Dahl, Node.js enabled JavaScript execution outside the browser environment by leveraging Google's V8 JavaScript engine and introducing an event-driven, non-blocking I/O model (Ahmod, 2023). This innovation represented a paradigm shift that allowed developers to use JavaScript across their entire application stack, addressing limitations in existing server-side technologies that struggled with high-concurrency scenarios.

The technical foundation of Node.js is built upon the robust V8 JavaScript engine, which employs Just-In-Time (JIT) compilation to optimize performance through sophisticated multi-tier compilation approaches (Ahmod, 2023). The event-driven architecture utilizes a callback queue and event loop mechanism that creates asynchronous execution capabilities despite JavaScript's single-threaded nature. When applications encounter I/O operations, the event loop can continue processing other tasks rather than blocking the entire application, making Node.js particularly well-suited for applications requiring high concurrency (Ahmod, 2023).

Node.js has achieved remarkable adoption since its introduction, establishing itself as a dominant force in server-side JavaScript development. Current usage statistics demonstrate that Node.js maintains a considerable presence across various industries and application types, proving its reliability in production environments (Usage statistics of Node.js, 2025; Greif & Burel, 2024).

A critical factor in Node.js's success has been the development of NPM (Node Package Manager), which became the largest repository of reusable software components. This extensive ecosystem has enabled developers to leverage thousands of pre-built modules, accelerating development time (Ahmod, 2023). The NPM ecosystem has become integral to the modern JavaScript development workflow, providing solutions for virtually every common programming challenge.

Node.js's success established the foundation for the broader JavaScript runtime ecosystem that exists today. Its proven capabilities in production environments, combined with the identification of specific limitations and areas for improvement, created the context in which newer runtimes like Deno and Bun emerged (Gautam, 2021). However, as applications face growing demands for performance and security, some limitations of Node.js have become more apparent, particularly in these areas (Ahmod, 2023; Dahl, 2018).

2.2 Limitations and Motivations for Change

Despite Node.js's remarkable success and widespread adoption, several limitations became apparent as the platform matured, and web development requirements evolved. The biggest critique came straight from Ryan Dahl, the original creator of Node.js himself. In his 2018 JSConf EU presentation titled "10 Things I Regret About Node.js," Dahl provided a retrospective analysis that points out critical design decisions that had proven problematic over time (Dahl, 2018).

One of the key concerns Dahl raised was Node.js's overly permissive security model. Unlike browser environments where JavaScript operates within a restricted sandbox, Node.js applications have unrestricted access to the file system, network, and system resources by default (Dahl, 2018). This design choice, while enabling powerful server-side capabilities, introduces substantial security risks, particularly when applications incorporate third-party modules from the NPM ecosystem.

The security entailment of this permissive model has been demonstrated through real-world incidents. Recent security breaches, such as the compromise of popular NPM packages with cryptocurrency mining malware, highlight the ongoing vulnerability of the Node.js ecosystem to supply chain attacks (Lakshmanan, 2024). These incidents show an increasing need for a more granular permission system that can limit potential damage from compromised or malicious packages.

Dahl also expressed regret about several design decisions related to the Node.js module system. The choice to use CommonJS instead of ES modules, the implementation of centralized package management through NPM, and the complex dependency resolution algorithm all contributed to what he described as unnecessary complexity (Dahl, 2018). The reliance on package.json files and the node_modules directory structure, while functional, created a system that could become unwieldy for large projects with complex dependency trees.

Additionally, responsiveness and latency have been noted as areas where improvement could greatly benefit applications, particularly in serverless and edge computing environments where rapid initialization is crucial (Vepsäläinen et al., 2025).

Comparative performance studies demonstrate that Node.js, while adequate for many use cases, may not satisfy the performance requirements of the more demanding applications (Ahmod, 2023). These performance constraints have become more apparent as web applications have grown in complexity and as developers have sought to optimize for metrics such as “Time to First Byte” (TTFB) and overall application responsiveness.

These limitations provided clear motivations for the development of alternative JavaScript runtimes. The security concerns led to an interest in permission-based systems that could provide fine-grained access control. Performance limitations motivated the exploration of new approaches that could improve overall execution speed and performance. The complexity of the module system and build tooling inspired efforts to create more streamlined, integrated development experiences.

Academic research has shown that alternative approaches to runtime design could address developers’ concerns (Gautam, 2021). The emergence of Deno and Bun can be

understood as direct responses to these described challenges, each taking steps to solve the problems that had become apparent in Node.js's design.

2.3 Deno Runtime

Deno emerged in 2020 as a direct response to the downfalls identified in Node.js, representing a rethinking of JavaScript runtime design with security and standardization as primary concerns. Created by Ryan Dahl and his team, Deno was explicitly designed to address security vulnerabilities, module system complexities, and architectural decisions that had proven problematic in Node.js (Dahl et al., 2020). Rather than attempting to evolve Node.js incrementally, Deno represented a clean slate approach that prioritized secure defaults and modern JavaScript standards.

The architectural difference between Deno and Node.js lies in the choice of implementation language. While Node.js was built using C++ and C, Deno was implemented in Rust, a systems programming language that emphasizes memory safety and performance. Deno's architecture combines the V8 JavaScript engine with Rust-based runtime services and utilizes the Tokio asynchronous runtime library for handling concurrent operations (Dahl et al., 2020).

The most distinctive feature of Deno is its permission-based security model, which differs from Node.js's permissive approach. By default, Deno applications run in a secure sandbox with no access to file systems, networks, or environment variables unless explicitly granted through command-line flags (Dahl et al., 2020). This security-first design addresses one of the primary concerns raised about Node.js, where applications and their dependencies have unrestricted access to system resources.

The permission system allows developers to grant specific capabilities such as *–allow-read*, *--allow-write*, *--allow-net*, and *–allow-env* on a granular basis, enabling fine-grained control over what resources an application can access (Xuân, 2025). This approach reduces the potential attack surface and limits the damage that compromised packages or malicious code can inflict, addressing the supply chain security concerns that have plagued the Node.js ecosystem.

Deno was designed with first-class support for modern JavaScript standards, including native ES modules, TypeScript, and web APIs. Unlike Node.js, which requires additional tooling for TypeScript compilation, Deno includes a built-in TypeScript compiler and can execute TypeScript files directly without additional configuration (Dahl et al., 2020). This integration eliminates many of the build process complexities that developers face in Node.js environments.

The runtime also provides built-in support for modern web standards, making it easier for developers to write code that works consistently across many environments (Xuân, 2025). This standard's compliance reduces the need for polyfills and compatibility layers that could be required in older Node.js applications in order to support new modern features in older environments.

Deno abandons the NPM ecosystem in favor of URL-based imports, allowing modules to be imported directly from any HTTP or HTTPS URL. This approach eliminates the need for package.json files, node_modules directories, and centralized package registries (Dahl et al., 2020). While this design provides greater flexibility and reduces some of the complexity associated with NPM's dependency resolution, it has also introduced new challenges related to version management and dependency reliability.

Recognizing some limitations in the URL-based approach, the Deno team introduced JSR (JavaScript Registry) in 2025, a modern package registry designed to support both Deno and other JavaScript runtimes while maintaining improved security and standards compliance (Dahl et al., 2025). JSR represents an evolution in Deno's approach to package management, attempting to balance the flexibility of URL imports with the convenience of centralized package management.

The release of Deno 2 in 2024 marked a considerable milestone in the runtime's evolution, with enhanced compatibility with the Node.js ecosystem (Dahl et al., 2024). This improved compatibility addresses one of the primary barriers to Deno adoption: the inability to leverage existing NPM packages. Deno 2 includes NPM compatibility features that allow developers to use many existing Node.js packages while maintaining Deno's security model and modern features.

The formation of the Deno Company in 2021 provided commercial backing for the runtime's development and signaled a commitment to long-term sustainability (Dahl & Belder, 2021). This commercial foundation has enabled more aggressive growth and marketing efforts, though it has also raised questions about the runtime's governance and future direction (Anderson, 2021).

While Deno's security-first design addresses many of Node.js's vulnerabilities, recent academic research has pointed out new security challenges specific to Deno's architecture and ecosystem. A comprehensive security study revealed various attack vectors and vulnerabilities in Deno's permission system and ecosystem, suggesting that the runtime faces its own unique security considerations (Alhamdan & Staicu, 2025). These findings highlight that while Deno's approach improves security, it does not eliminate all security concerns and requires ongoing vigilance.

Despite its technical merits, Deno has faced adoption challenges that have limited its impact in production environments. Some developers have expressed disillusionment with Deno's ecosystem maturity, compatibility limitations, and the practical difficulties of migrating from established Node.js workflows (Bjarnason, 2024). The runtime's departure from established patterns in the JavaScript ecosystem has created a learning curve that some developers find challenging to justify, given the benefits.

The comparison between Node.js and Deno often reveals trade-offs between innovation and compatibility, with developers weighing Deno's modern features and improved security against Node.js's mature ecosystem and proven reliability in production environments (Chazoule, 2023). These considerations can be important for businesses evaluating whether to adopt Deno for new projects or consider staying in the current Node.js ecosystem.

2.4 Bun Runtime

Bun emerged in 2021 as the newest entrant in the JavaScript runtime landscape, distinguishing itself through an aggressive focus on performance optimization and developer experience. Announced with its 1.0 release in August 2023, Bun positioned itself not as a complete reimagining of JavaScript runtime design like Deno, but rather as a faster, more integrated alternative that maintains compatibility with the existing Node.js ecosystem (Sumner et al., 2023). This compatibility-first approach addresses one of the primary barriers that alternative runtimes face: developers’ and organizations’ reluctance to abandon the vast NPM ecosystem and established Node.js workflows.

The technical architecture of Bun represents a substantial departure from both Node.js and Deno in its choice of JavaScript engine. While Node.js and Deno both utilize Google's V8 engine, Bun is built upon JavaScriptCore, the JavaScript engine originally developed by Apple for the Safari browser (Sumner et al., 2023). This architectural deviation provides Bun with distinct performance characteristics and optimization opportunities compared to V8-based runtimes.

Bun is written in Zig, a low-level systems programming language that emphasizes performance and manual memory management while maintaining memory safety. The combination of JavaScriptCore and Zig creates a runtime stack optimized for minimal overhead and maximum execution speed.

Bun includes a built-in package manager, a native bundler for production deployments, a test runner with Jest-compatible APIs, and a transpiler that handles TypeScript, JSX,

and modern JavaScript syntax without additional configuration (Sumner et al., 2023). This level of integration reduces the need for some external dependencies and configuration files, streamlining the development workflow.

Ahmod's (2023) comparative analysis demonstrated that Bun consistently outperformed Node.js across multiple metrics, including startup time, memory usage, execution speed, and HTTP request handling capacity. The benchmarks showed that Bun could achieve startup times approximately three times faster than Node.js. These performance gains stem from multiple factors, including the JavaScriptCore engine's characteristics, optimized native implementations of common operations, and architectural decisions that minimize overhead.

The runtime also includes native implementations of common functionalities that typically require external packages in Node.js. For example, Bun provides built-in support for reading environment variables from .env files, native SQLite database access, and optimized implementations of frequently used Node.js modules (Partovi, 2025).

Bun's compatibility strategy represents a pragmatic approach to the challenge of ecosystem adoption. Rather than requiring developers to rewrite code or abandon familiar patterns, Bun aims to serve as a drop-in replacement for Node.js (Sumner et al., 2023). The runtime implements Node.js APIs and supports the package.json format, NPM packages, and common Node.js conventions. By version 1.2, released in January 2025, Bun had achieved ninety-nine percent compatibility with Node.js built-in modules, meaning that the vast majority of existing Node.js applications can run on Bun with minimal or no modifications (Partovi, 2025). This profound compatibility reduces the risk and effort required from developers to experiment with Bun.

The adoption patterns revealed in the State of JavaScript 2024 survey show that Bun has achieved notable early traction, with 16% of respondents reporting regular use despite being the newest of the three runtimes examined in this study (Greif & Burel, 2024). This adoption rate suggests that Bun's performance benefits and compatibility approach resonate with a segment of the developer community.

However, Bun's relative newness also means it has less production history than Node.js's 15 years of testing. While the runtime has proven its performance in benchmarks and smaller applications, questions remain about its behavior in large-scale, complex production environments over an extended period of time. The ecosystem of Bun-specific packages and tools is still developing, and edge cases in Node.js compatibility continue to be discovered and addressed with each release (Partovi, 2025).

Bun represents an important evolution in JavaScript runtime design, demonstrating that large performance improvements are possible while maintaining compatibility with existing ecosystems. As Bun continues to mature and its ecosystem develops, it may prove to be a viable option for performance-critical applications.

2.5 Current Landscape and Developer Adoption

Node.js maintains a market-leading position relative to its competitors. However, the 2024 State of JavaScript survey also revealed that developers are seeking improvements in several areas, including better organization and maintenance, improved dependency management, higher performance, stronger type-safety, and more advanced build tools. These findings suggest the potential for a shift in market dynamics and may indicate an opening for emerging competitors (Greif & Burel, 2024).

Deno has better built-in security practices compared to Node.js and its NPM ecosystem, which has become more notorious for its supply chain attacks over time, making headline news in many blogs and forums. (Lakshmanan, 2024)
Despite Deno boasting better security by default, it does not mean that Deno is immune to such attacks. It may have a smaller attack surface, but it does not mean that more sophisticated attacks would not work (Alhamdan & Staicu, 2025).

Ultimately, the choice among these runtimes appears to depend on priorities: Bun could be recommended for projects where speed is paramount, Node.js for its large community and extensive support, and Deno for those seeking a balance in between, with modern features and improved runtime security by default.

3 Comparison of Runtime Features

Having established the historical context and evolution of JavaScript runtimes in the previous chapter, it becomes necessary to examine the specific technical features that differentiate Node.js, Deno, and Bun from one another. This chapter provides a comparison of the core features that developers encounter in their day-to-day work with these runtimes.

3.1 Module Systems and Dependency Management

One of the many ways these three runtimes differ lies in how they handle modules and manage dependencies. Node.js originally used the CommonJS module system, which uses the *'require'* function to load modules synchronously. As Ryan Dahl (2018) explained in his reflection on Node.js design decisions, this choice made sense at the time but created long-term complications as JavaScript evolved. Node.js has since added support for ECMAScript modules through the import and export syntax, but this dual system has created complexity for developers who must navigate between two different module formats within the same ecosystem.

The package management in Node.js centers around NPM and the package.json file, which creates a centralized registry for discovering and installing packages. While this system has been incredibly successful in building one of the largest package ecosystems, it has introduced some security concerns. The Rspack NPM packages compromise in December 2024 demonstrated how malicious actors can inject crypto mining malware into popular packages, affecting thousands of developers who trusted the NPM ecosystem (Lakshmanan, 2024). This incident highlighted the vulnerabilities inherent in a centralized package registry where package maintainers have meaningful power over the security of downstream applications.

Deno took a different approach to module management by embracing web standards from the beginning. Instead of relying on a centralized package registry, Deno allows developers to import modules directly from URLs, treating modules as web resources (Dahl et al., 2020). This design decision aligns with how browsers handle JavaScript modules and eliminates the need for a package.json file or a node_modules directory. The runtime caches downloaded modules locally, but the source of truth remains in the URL itself.

However, Deno's package ecosystem has faced criticism from developers accustomed to the Node.js ecosystem. Bjarnason (2024) expressed frustration, noting that requiring

more than just Deno's standard library or tooling is difficult, and that the lack of a centralized discovery mechanism makes it harder to find quality packages. Recognizing these concerns, the Deno team introduced import maps as a way to create aliases for URLs, and more importantly, they launched JSR (JavaScript Registry) in 2025 as a modern alternative to NPM (Dahl et al., 2025). JSR is designed specifically for TypeScript and ECMAScript modules, providing package discovery while maintaining Deno's security-first philosophy. The registry generates documentation automatically from TypeScript types and enforces package quality standards that NPM does not require (Dahl et al., 2025).

Bun represents another approach to dependency management by prioritizing compatibility with the existing Node.js ecosystem while improving performance. Bun uses a package.json file just like Node.js, but it implements its own package manager that is notably faster than NPM, yarn, or PNPM. According to the Bun 1.0 announcement, the Bun package manager can install dependencies up to twenty times faster than NPM in certain scenarios (Sumner et al., 2023).

3.2 Built-in Tooling and Standard Library

The scope and quality of built-in tooling represent another differentiator between these runtimes. Node.js follows a philosophy where the core runtime provides basic functionality, and developers rely on third-party packages for most tools. This means that a typical Node.js project requires installing separate packages for tasks like testing, code formatting, bundling, and transpiling TypeScript. While this approach seems flexible, developers still have to research, select, and configure multiple tools, each with its own documentation, update cycle, and potential security vulnerabilities.

Deno challenged this approach by including many common development tools directly in the runtime. From versions 1.x, Deno shipped with a built-in test runner, code formatter, compiler, and linter. These tools follow consistent command patterns and work together seamlessly. For example, running '*deno fmt*' formats code according to opinionated standards without requiring any configuration files, while '*deno test*' discovers and executes tests without needing additional test frameworks like Jest (Dahl et al., 2020; Dahl & Iwańczuk, 2020). This integrated approach can reduce the number of dependencies a project needs and simplify the development workflow.

The Deno standard library also deserves attention as it provides well-tested, audited modules for common tasks like file system operations, HTTP servers, and data manipulation. Unlike Node.js, where developers often reach for third-party packages for these tasks, Deno provides its standard library modules that are maintained by the

core team and follow semantic versioning (Dahl et al., 2024). This reduces the need for external dependencies and provides more stability.

Bun implements many Node.js APIs natively and performantly, while adding new APIs for common needs. This includes native support for environment variable files, built-in HTTP server capabilities, and native implementations of packages like node:sqlite (Partovi, 2025). By version 1.2, Bun reached 99% compatibility with Node.js APIs while offering better performance.

Table 1 summarizes the key build-in tooling and features available in each runtime.

3.3 TypeScript Support

To get full TypeScript support for all TypeScript-related features, Node.js requires developers to set up a third-party TypeScript package and create a tsconfig.json configuration file (Node.js, 2025).

Deno was designed with TypeScript in mind, executing TypeScript files directly without configuration by including a TypeScript compiler internally (Dahl et al., 2020). Developers simply write TypeScript and run it with Deno. The runtime uses TypeScript for its own standard library, providing type definitions and IDE support.

Bun also provides first-class TypeScript support with direct file execution. The runtime uses a transpiler rather than a full type checker, stripping type annotations and transforming syntax very quickly without validating type correctness (Sumner et al., 2023).

Node.js uses type stripping similar to Bun, removing type annotations without performing type validation, prioritizing execution speed over type safety. In contrast, Deno maintains the option for full type checking when needed. (Dahl & Jiang, 2025).

To wrap up the last sections, Table 1 shows how each runtime compares to the other and what features they provide.

Table 1. Overview of runtime features.

Feature	Node.js	Deno	Bun
Test Runner	External	Built-in	Built-in
Code Formatter	External	Built-in	Built-in
Linter	External	Built-in	External
TypeScript Support	Partial	Built-in	Built-in
Bundler	External	Built-in	Built-in

3.4 Compatability and Migration View

As Node.js is the most adopted and widely supported runtime with the largest ecosystem, both Deno and Bun have invested substantially in Node.js compatibility to lower adoption barriers.

Deno originally positioned itself as a clean break from Node.js, requiring developers to rewrite code for URL imports and Deno's APIs. This created friction as developers were reluctant to abandon existing codebases and the NPM ecosystem. The introduction of NPM specifiers allowed importing NPM packages directly using syntax like ‘*import express from "npm:express"*’, making much of the NPM ecosystem accessible (Dahl et al., 2024).

Despite improvements, some developers remain skeptical. Bjarnason (2024) argued that Deno's initial rejection of the Node.js ecosystem and subsequent backtracking undermined its original vision, creating a runtime that neither fully embraces Node.js compatibility nor cleanly breaks from it. This highlights the challenge of balancing innovation with practical adoption concerns.

Bun took the opposite approach by making Node.js compatibility a core priority from the start, aiming to be a drop-in replacement implementing the same APIs and supporting the same package ecosystem without code changes (Sumner et al., 2023). This lowers adoption risk as developers can experiment on existing projects without committing to full rewrites.

The compatibility story also extends to web standards. Both Deno and Bun implement many Web APIs like fetch, WebSocket, and ReadableStream that are familiar to frontend developers and standardized across browsers (Dahl et al., 2020; Sumner et al., 2023). This alignment makes it easier to write code that works across many environments, from backend services to edge functions to browser applications.

Migrating large Node.js applications to Deno may require more planning and refactoring. In contrast, trying Bun on existing Node.js projects can be as simple as replacing the '*node'* command with '*bun'* in development scripts. Performance improvements may justify this switch even without code changes (Ahmod, 2023; Sumner et al., 2023).

For new projects, compatibility considerations shift from migration to ecosystem benefits. Developers starting fresh with Deno can fully embrace its URL-based imports and modern architecture or use NPM packages through compatibility features. Bun developers get automatic access to the entire NPM ecosystem while benefiting from improved performance and integrated tooling. Node.js remains the standard, established choice with comprehensive documentation and community support (Tal, 2023). The compatibility landscape continues to evolve as all three runtimes actively develop new features and improve interoperability.

4 Performance Comparison and Analysis

While the previous chapter examined the feature sets and architectural differences between Node.js, Deno, and Bun, this chapter focuses on performance from locally run benchmarks and compares it with existing research to provide a basis for comparing these runtimes.

The performance differences between each runtime should always be taken with a slight scepticism. Performance benchmarks don't necessarily give the full picture of what the runtime is like, how it will perform over a longer period of time with real applications, and other potential considerations. (Ratanaworabhan et al., 2010)

4.1 Benchmarking Approach and Test Environment

The benchmark tests were run on a MacBook Air M2 with 16GB of system memory, with no other applications running in the background, running tests that were meant to compare performance from the Node.js ecosystem and what kind of performance could be expected.

Each test was manually run 10 times, taking the best result for each runtime to avoid potential hiccups and bias. The following benchmark code and other material can be reviewed and viewed in GitHub. (Laakso, 2025)

This benchmarking was not meant to be scientific nor perfect, but rather a window to compare results found in other research papers and articles, to get a general understanding of where each runtime lands in terms of performance.

As stated at the beginning of this chapter, these performance metrics should not be treated as an absolute source of truth, since there can be many variables that come into play with performance benchmarking. A good example of this can be observed in Figure 2 and Figure 3, which showcases a performance impact Deno suffers from having to convert Node.js code for compatibility, resulting in some performance overhead.

4.2 Benchmarks

This section showcases performance benchmarks that were conducted for Node.js, Deno, and Bun. The tests cover simple HTTP request handling, file system operations, JSON processing, and CPU computation.

Table 2 shows a simple HTTP server implementation in Node.js that was used for this request-per-second benchmark.

Table 2. Simple HTTP server code that was used for Figure 2 benchmark.

```
import { createServer } from "node:http";

const server = createServer((req, res) => {
  res.writeHead(200, { "Content-Type": "application/json" });
  res.end(JSON.stringify({ message: "Hello World", timestamp:
Date.now() }));
});

server.listen(3000, () => console.log("Server running on port
3000"));
```

Figure 2 shows how well each runtime handles a continuous stream of HTTP Requests, utilizing the autocannon NPM package with 10 clients over 10 seconds, and then counting the average HTTP requests handled per second.

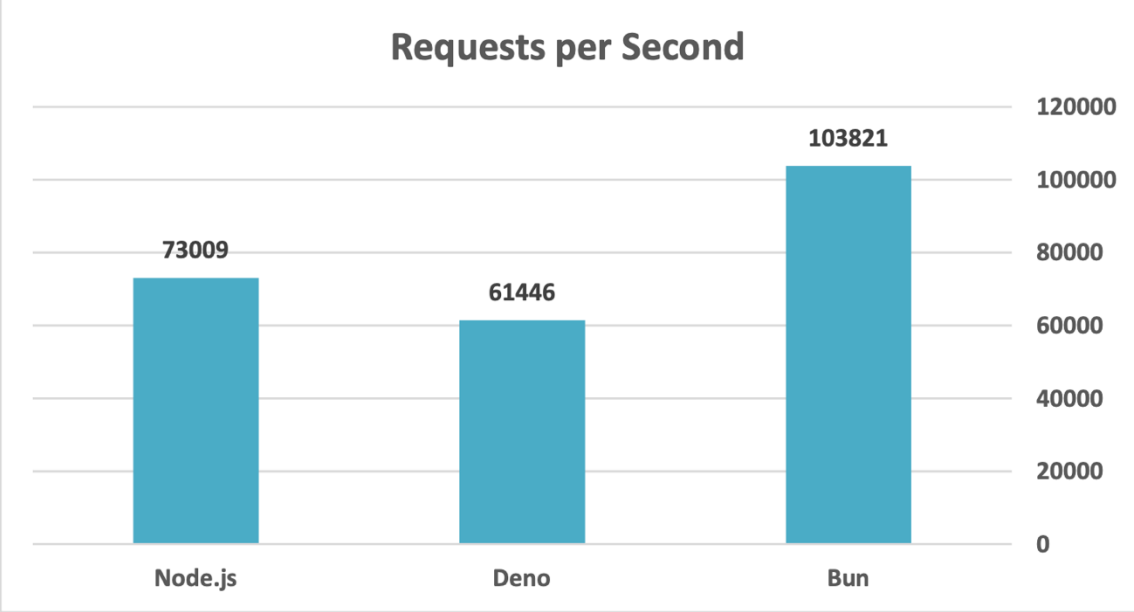

Figure 2. How well each runtime performed in a constant stream of HTTP requests.

In this instance, Bun seems to handle the largest number of requests, reaching as high as 100 000 requests. Node.js is not far behind, with roughly 70 000 requests. Deno is performing the worst in this case, with only 61 446 requests. This is most likely caused

by Deno having to convert Node.js-specific behaviour to something that the Deno runtime is compatible with, which ends up costing performance.

In Figure 3, we have the same test run with the exception of Deno serving an HTTP server with Deno-specific code shown in Table 3.

Table 3. Deno specific code that was used to host a basic HTTP server.

```
Deno.serve({ port: 3000 }, () => {
  return new Response(
    JSON.stringify({ message: "Hello World", timestamp: Date.now()
}),
    {
      headers: { "Content-Type": "application/json" },
    },
  );
});
```

We can observe improved performance for Deno, now performing on par with Node.js.

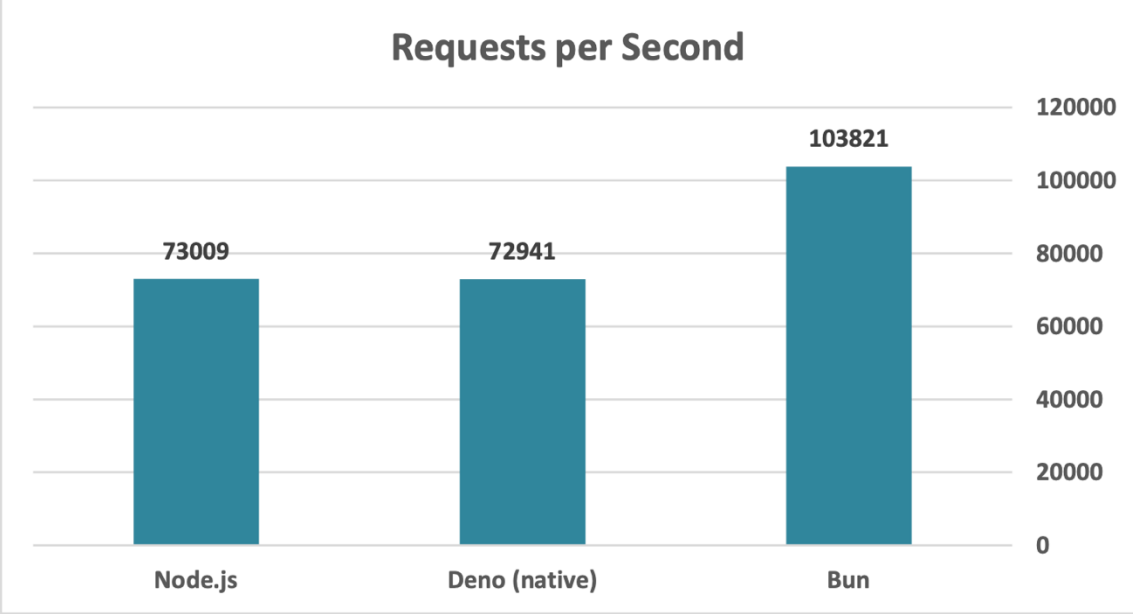

Figure 3. Showing Deno's improved performance when running Deno-specific code.

This is a good example of the matter that benchmarks are not everything, and results can vary depending on the implementation alone. With this in mind, it could be considered that with native implementations, Deno should at the very least perform at the same level as Node.js in many scenarios. Similarities can also be attributed to the

fact that both runtimes use the V8 engine, which could indicate a performance limitation with said engine.

File I/O Performance

In this section, we test the read and write performance of each runtime. Reading and writing 1000 files, each with 1 kilobyte of data.

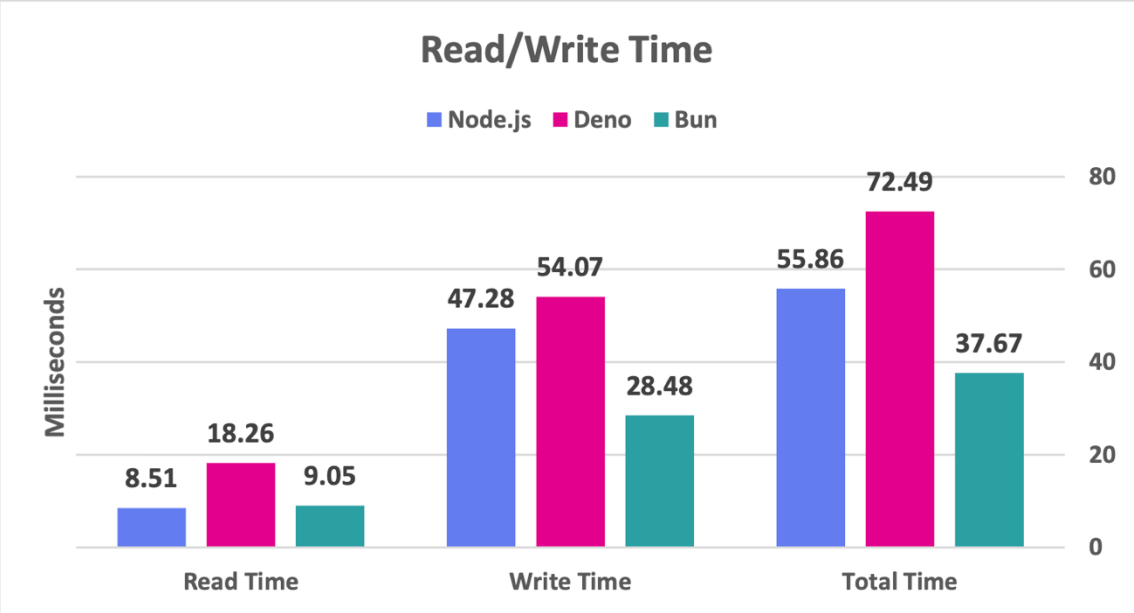

Figure 4. Runtime read and write performance.

From these benchmarks, we can observe that Bun once again comes first, fully completing the task in approximately 38 milliseconds. Followed up by Node.js in roughly 56 milliseconds, and lastly Deno, being the slowest at around 72 milliseconds.

JSON Stringify and Parse Performance

In this section, we will see how performantly each runtime handles JSON, handling userdata objects over multiple iterations.

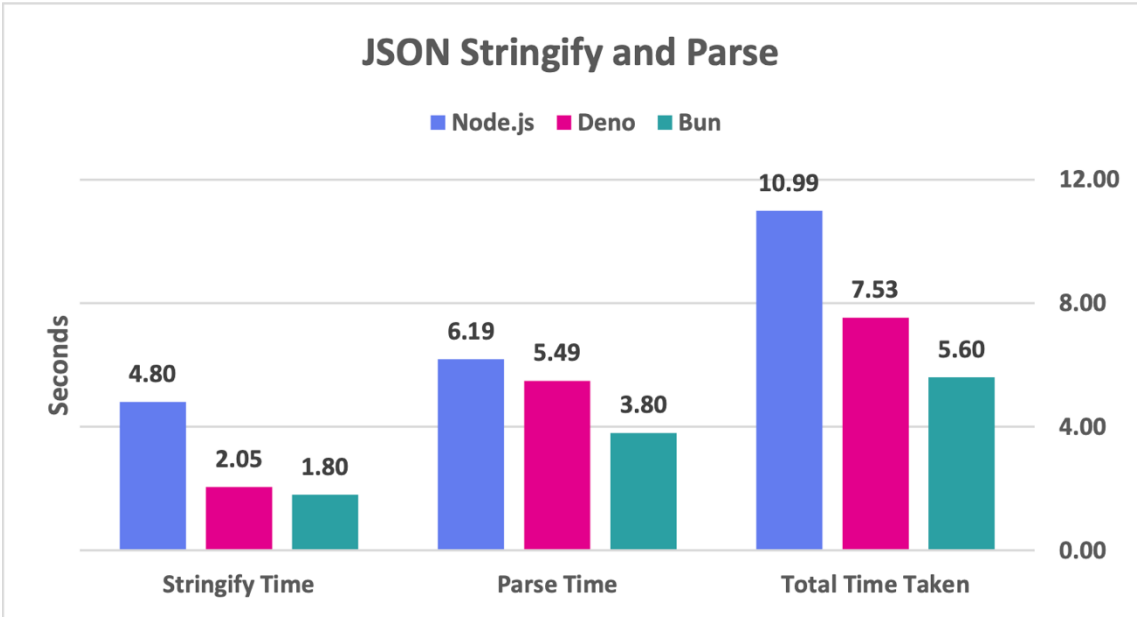

Figure 5. Displaying JSON Stringify and Parse performance between each runtime.

Once again, Bun comes out as the fastest in JSON operations, completing the task in 5,6 seconds. Deno this time clearly performing better than Node.js, fully completing the operation in 7,53 seconds. In this scenario, both Deno and Bun can be observed to be superior for JSON operations over Node.js.

CPU-bound Operations Performance

In this section, we will compare runtime performance in milliseconds, how quickly each runtime runs through a set of Fibonacci calculations and array sorting.

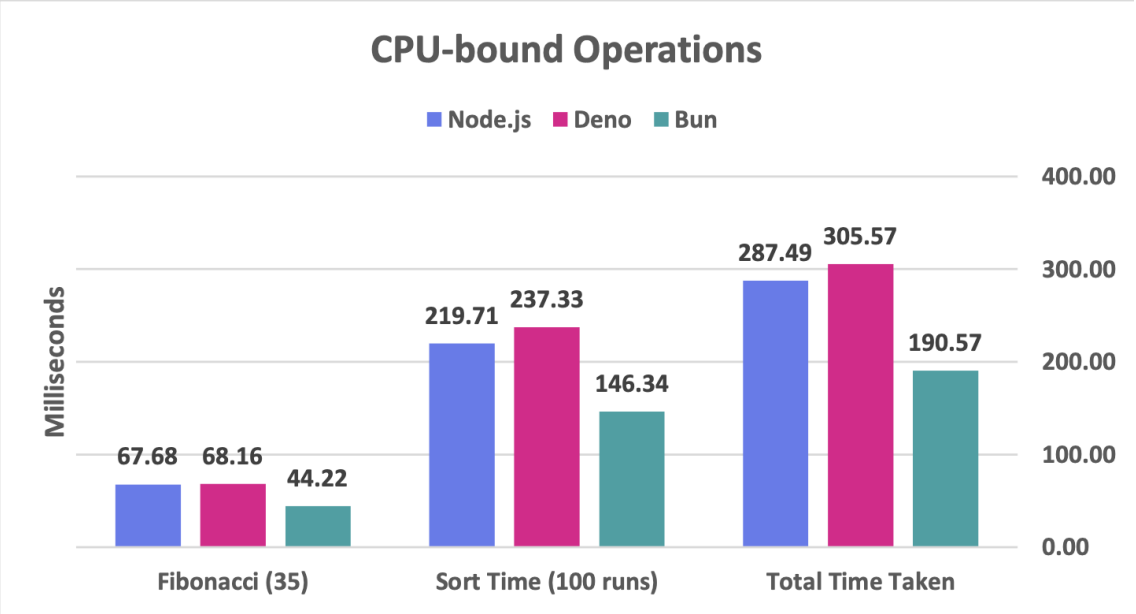

Figure 6. CPU-bound operations, performance displayed in milliseconds.

In this test, Bun outperforms both Node.js and Deno by a noteworthy margin, and once again, both Node.js and Deno perform mostly similarly, within a margin of error. Deno being ever so slightly slower.

4.3 Performance Analysis and Interpretation

These benchmark results showcase consistent patterns across different tests. Bun performing the best across all tested scenarios, likely due to the use of the JavaScriptCore engine. Deno's performance falls somewhere between Node.js and Bun in most benchmarks. Since Deno also uses the V8 engine like Node.js, its raw JavaScript execution speed is comparable to Node.js in many scenarios. Tal (2023) noted that Deno's additional security checks and permission system can introduce some overhead, particularly during file system operations and network requests where permission validation must occur. This security overhead is generally minimal for CPU-bound tasks but becomes more noticeable in I/O-heavy applications where the runtime must repeatedly verify permissions for external resource access.

For web applications and API services, HTTP server performance can be important. Tal (2023) also conducted HTTP server benchmarks, measuring requests per second and response latency. The results demonstrated that Bun outperformed both Node.js and Deno, which is in line with these benchmark results. When Ahmod (2023) tested HTTP

server performance, the findings aligned with Tal's results, showing Bun maintaining higher throughput even as the concurrent connection count increased.

That being said, Node.js HTTP performance is not to be considered bad; if anything, it’s still quite adequate, and the runtime remains highly capable of serving high levels of web traffic. Many of the world's largest web applications run on Node.js, which demonstrates that raw benchmark numbers do not tell the complete story of production readiness. While Bun's built-in server may achieve higher benchmark scores, Node.js servers benefit from extensive battle-testing and tooling built around them.

Deno's HTTP server performance has evolved over the course of its development. The native HTTP server implementation in Deno improved performance to be on par with Node.js. While Deno may not match Bun's benchmark numbers, its HTTP performance is respectable and adequate for most web applications. The runtime's focus on web standards means that developers can use the same fetch API and web server patterns across different environments, which offers architectural benefits even if not pure performance advantages.

4.4 Security and Performance Implications

Node.js operates with an implicit trust model where any executed code has full access to system resources by default. This approach minimizes security-related performance overhead since the runtime does not need to verify permissions for operations. However, this also means that malicious or compromised code can freely access the file system, network, and environment variables without any restriction.

Each time a Deno script attempts to access any system resource, the runtime must verify that the appropriate permission has been granted. Alhamdan and Staicu (2025) conducted an analysis of Deno's security implementation and found that while the permission checks introduce some overhead, the impact on overall application performance is typically minimal for most workloads, which could be observed in earlier benchmark results where Deno mostly matched the performance of Node.js.

Bun, on the other hand, follows Node.js's trust model while focusing on performance optimization. The runtime does not implement a permission system like Deno's, leaving the responsibility to developers. This fact allows Bun to avoid the performance overhead of permission checks, but it also means that Bun applications face similar security considerations as Node.js applications.

The practical meaning of these different security approaches extends beyond raw performance metrics. Organizations evaluating runtimes could consider whether the additional security provided by Deno's permission system justifies any performance overhead and whether it aligns with their security requirements. For educational environments, Deno's security model could be considered a valuable teaching opportunity about least privilege and resource access control. For high-performance scenarios where every millisecond matters, Bun's approach of minimizing overhead might be preferable, with security implementations and considerations left over to the developer.

4.5 Developer Experience Considerations

All runtimes have their own positives and negatives. Node.js, being the oldest and most supported runtime, has many guides and research already built around it, whereas both Deno and Bun are still fairly new and have room to grow. Despite that, both Deno and Bun offer some neat functionality out of the box that can make the developer experience more enjoyable, and potentially save time by not needing to do separate configuration for modern development tooling like code linting and formatting.

Deno, for example, does not need a separate formatter, linter, or TypeScript configuration. These functionalities already exist within the runtime, so there is no need to necessarily configure anything when starting a new project.

Bun also comes with direct TypeScript support out of the box, but Bun doesn’t have the same kind of standard library or tooling as Deno does. Bun more or less relies on what already exists within the Node.js ecosystem.

Currently, both Deno and Bun support existing Node.js code; specifically, support for modules and packages has gotten relatively good over the past year.
On release, Deno did not have backwards compatibility, but since 2023, they have gradually started to add more support. Bun has always from wanted to support Node.js’s existing ecosystem. On release, there was Node.js support, which worked most of the time, but still some issues persisted. Bun has, over the years, polished their Node.js support to the point where Bun can most likely run most existing JavaScript projects out there without any need for big modifications.

5 Education and Business Considerations

Having examined the technical features, performance characteristics, and compatibility considerations of Node.js, Deno, and Bun, it becomes necessary to address the practical factors for educators and businesses. This chapter concludes the research to guide runtime adoption decisions in academic and commercial contexts.

5.1 Runtime Adoption for Education

The decision of which JavaScript runtime to teach carries some consideration for student learning and career preparedness. Current usage statistics demonstrate that Node.js remains the predominant runtime and is considered the current standard. Node.js should remain the standard for education, as it continues to be widely used globally, and the skills acquired are transferable to other runtimes.

Deno's security model provides valuable lessons about the principle of least privilege and secure-by-default design. As Dahl et al. (2020) emphasized, Deno was designed to address issues with Node.js that cannot be fixed without any negative implications. Exposing students to these improved designs may help them understand that current standards are not necessarily optimal as technology and development keep moving forward.

From a more practical standpoint, educators need not choose exclusively between runtimes. Teaching core JavaScript and Web Development concepts using Node.js still gives knowledge towards these alternative runtimes. Demonstrating runtime security with simple permission examples of Node.js compared to Deno could be a fun learning experience for students, seeing firsthand that Node.js will not prevent a malicious use case.

5.2 Business Adoption and Migration

The performance improvements demonstrated in Chapter 4 represent genuine value for businesses where performance directly impacts costs or user experience. Organizations running serverless functions that execute millions of times daily could realize cost savings from Bun's faster execution and lower memory usage. However, performance alone rarely justifies the migration of existing applications.

The compatibility analysis in Chapter 3 revealed that while Bun aims for drop-in Node.js compatibility and achieves high compatibility rates, subtle differences may still exist that require testing and potential code changes. Migrating a large Node.js application to Bun requires thorough testing to ensure that all dependencies work correctly and that no edge cases cause failures in production. The risk of introducing bugs or unexpected behavior during migration must be weighed against the performance benefits.

Deno presents a more complex migration story due to its different approach to modules and security. Organizations considering Deno for existing applications face more extensive refactoring work compared to trying Bun. However, Deno's improved Node.js compatibility in version 2.0 has reduced this barrier to entry (Dahl et al., 2024). The NPM specifier support means that organizations can gradually migrate applications while still using NPM packages, though this hybrid approach may not fully leverage Deno's architectural improvements.

Starting a new project with Bun or Deno involves less risk than migrating existing code because there is no legacy codebase to maintain compatibility with. Organizations can evaluate whether the performance benefits, integrated tooling, or security model align with project requirements. A new API service with high performance requirements might benefit substantially from Bun's speed, while a new application with more security requirements might value Deno's permission system.

6 Conclusion and Future Work

This thesis embarked on an investigation into the evolving landscape of JavaScript runtimes, motivated by the emergence of Deno and Bun as alternatives to Node.js. The primary objective was to conduct a comparative analysis of these three technologies to guide developers, educators, and businesses. By examining their historical context, technical features, and performance characteristics, this work sought to clarify the trade-offs inherent in choosing a runtime in the modern web development ecosystem.

Through this investigation, it can be concluded that Node.js still remains the established industry standard, its position earned through a long track record of use, a vast and mature ecosystem, and a large global talent pool. While it may lack in raw performance, its reliability and extensive community support make it the default choice for a wide range of applications.

Deno started from a direct critique of Node.js's original design, successfully delivering on its promise of a more secure by default and modern development experience. While its performance is generally on par with Node.js, its primary appeal lies in its architectural coherence, standard library, web standards, and tooling, which make it an excellent choice for new projects where a level of security and developer experience matter.

Bun has clearly established itself as the clear leader in performance. Consistently outperforming both Node.js and Deno across benchmarks. By prioritizing high compatibility with the Node.js ecosystem from the beginning, Bun offers a compelling, low-friction migration path for existing applications, with promising performance gains.

Ultimately, this thesis concludes that there is no single correct choice when it comes to JavaScript runtimes. The choice may depend on multiple factors such as project requirements, performance, and long-term support. Node.js has enjoyed an era of unchallenged dominance, but we are finally starting to see competition in the server-side JavaScript community.

On a personal level, this thesis research achieved the goals I set out to accomplish. My primary motivation was to deepen my understanding of JavaScript runtimes beyond surface-level knowledge driven by my own interests and to develop my academic research skills. I also identified areas for improvement in my own work and thinking for conveying complex information to others more simply.

6.1 Future Work

While this thesis provides a snapshot of the current JavaScript landscape at the end of 2025, the rapid pace of development in this area leaves questions open for future investigation and research. The performance benchmarks conducted, while necessary for comparison, represent specific workload patterns that don't reflect all scenarios. Long-term stability and ecosystem evolution of newer runtimes remain slightly uncertain, requiring continued observation as these platforms keep maturing.

Future research would benefit from real-world scenarios in which runtime performance differences genuinely impact business outcomes or user experience meaningfully, moving beyond synthetic benchmarks to production case studies. The longevity of these platforms also warrants investigation. Which runtimes will continue to see active development in the coming years, and what factors determine their staying power?

Technical questions also remain open. Can runtime-level performance continue improving, or are gains primarily limited by the underlying JavaScript engines (V8, JavaScriptCore)? Standardized benchmarking methodologies would benefit the entire ecosystem, enabling more reliable comparisons across runtime versions and research studies.

Finally, ecosystem evolution presents questions about the future of JavaScript package management. Will npm maintain its dominance, or could JSR gradually capture significant market share?

References

Ahmod, M. F. (2023). *Javascript runtime performance analysis: Node and Bun*. Trepo. https://trepo.tuni.fi/handle/10024/149672

Alhamdan, A., & Staicu, C.-A. (2025). Welcome to Jurassic Park: A comprehensive study of security risks in Deno and its ecosystem. *Network and Distributed Systems Security Symposium*, 1-17. https://doi.org/10.14722/ndss.2025.23284

Anderson, T. (2021, April). The JavaScript ecosystem is 'hopelessly fragmented'... so here is another runtime: Deno is now a company. *The Register*. https://web.archive.org/web/20240625220804/https://www.theregister.com/2021/04/06/deno_is_now_a_company/

Bjarnason, B. (2024, January). Disillusioned with Deno. https://www.baldurbjarnason.com/2024/disillusioned-with-deno/

Chazoule, D. (2023, December). Node or Deno, that is the question!? https://dev.to/dmnchzl/node-or-deno-that-is-the-question-16h9

Dahl, D., & Jiang, A. (2025, March). Node just added TypeScript support. What does that mean for Deno? https://Deno.Com/Blog/Typescript-in-Node-vs-Deno.

Dahl, R. (2018). 10 Things I regret about Node.js. JSConf EU. https://www.youtube.com/watch?v=M3BM9TB-8yA

Dahl, R., & Belder, B. (2021, March). Announcing the Deno Company. https://deno.com/blog/the-deno-company

Dahl, R., Belder, B., & Iwańczuk, B. (2020, May). Deno 1.0. https://deno.com/blog/v1

Dahl, R., Belder, B., Iwańczuk, B., & Jiang, A. (2024, October). Announcing Deno 2. https://deno.com/blog/v2.0

Dahl, R., Casonato, L., & Whinnery, K. (2025, May). Introducing JSR - The JavaScript Registry. https://deno.com/blog/jsr_open_beta

Dahl, R., & Iwańczuk, B. (2020). Deno in 2020. https://deno.com/blog/deno-in-2020.

Gautam, S. (2021). *Deno - A new Node.js?* [Bachelor's thesis, Haaga-Helia, University of Applied Sciences]. Theseus. https://www.theseus.fi/handle/10024/497744

Greif, S., & Burel, E. (2024, December). State of Javascript: 2024. https://2024.stateofjs.com/en-US/other-tools/#runtimes

Laakso, J. (2025, November). JavaScript Runtime Benchmarks. https://github.com/Ducky07/js-runtime-snapshot/tree/11-2025

Lakshmanan, R. (2024, December). Rspack npm Packages Compromised with Crypto Mining Malware. The Hacker News. https://thehackernews.com/2024/12/rspack-npm-packages-compromised-with.html

Node.js. (2025, August). Modules: TypeScript | Node.js. https://web.archive.org/web/20250812125702/https://nodejs.org/api/typescript.html

Partovi, A. (2025, January 22). Bun 1.2. *Bun Blog*. https://bun.sh/blog/bun-v1.2

Ratanaworabhan, P., Livshits, B., & Goth Zorn, B. (2010). JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. https://www.usenix.org/legacy/events/webapps10/tech/full_papers/Ratanaworabhan.pdf

Sumner, J., Partovi, A., & McDonnell, C. (2023, August 8). Bun 1.0. *Bun Blog*. https://bun.sh/blog/bun-v1.0

Tal, L. (2023, September). Node.js vs. Deno vs. Bun: Performance & JavaScript runtime comparison. Snyk Blog. https://snyk.io/blog/javascript-runtime-compare-node-deno-bun/

Usage statistics of Node.js. (2025). W3Techs. https://w3techs.com/technologies/details/ws-nodejs

Vepsäläinen, J., Vuorimaa, P., & Hellas, A. (2025). The potential of serverless edge-powered islands for web development. *Journal of Web Engineering*, 24(1), 1-38. https://doi.org/10.13052/jwe1540-9589.2411

Xuân, H. (2025). Things I like about Deno. https://2coffee.dev/en/articles/5-things-i-like-about-deno