FROM NODE.JS TO DENO: A COMPARATIVE ANALYSIS AND CASE STUDY OF MIGRATION

Lappeenranta–Lahti University of Technology LUT

Bachelor’s thesis, Software Engineering

2025

Author: Rikhard Badorek

Examiner: University Lecturer Erno Vanhala, D.Sc. (Tech.)

ABSTRACT

Lappeenranta–Lahti University of Technology LUT

LUT School of Engineering Sciences

Software Engineering

Rikhard Badorek

From Node.js to Deno: A comparative analysis and case study of migration

Bachelor's thesis

2025

40 pages, 8 figures, 1 table

Examiner: University Lecturer Erno Vanhala (D.Sc.)

Keywords: Node.js, Deno, web development, web programming

This thesis conducts a comparative analysis and a practical case study on migration from Node.js to Deno. It aims to evaluate whether migration between the runtimes is a feasible option. Node.js has been a dominant runtime for server-side applications, but over the years some flaws and limitations have led to a new runtime Deno. Deno was created by Ryan Dahl in 2020 to address some of the shortcomings of Node.js

The study employs a mixed-method approach for the research. A comprehensive literature review is combined with a practical case study that involves migrating a full-stack application's backend from Node.js to Deno.

Results indicate that Deno is a very viable option in many development cases. It generally outperforms Node.js in handling high-concurrency and in I/O-bound tasks. Additionally, Deno offers better security, streamlined configuration and package management and built-in tools. However, migration should be considered case-by-case due to challenges such as the increased CPU usage and a less mature community.

Based on the findings, this study proposes a set of guidelines for developers who are considering migration from Node.js to Deno. The study concludes that Deno can provide significant improvements for web development but the decision to migrate should be carefully evaluated and considered to find if the migration could bring real value and be a feasible option for the case.

TIIVISTELMÄ

Lappeenrannan–Lahden teknillinen yliopisto LUT

LUTin insinööritieteiden tiedekunta

Tietotekniikka

Rikhard Badorek

Nodesta Denoon: Vertaileva analyysi sekä tapaustutkimus siirtymisestä

Tietotekniikan kandidaatintyö

2025

40 sivua, 8 kuvaa, 1 taulukko

Tarkastaja: Yliopisto-opettaja Erno Vanhala (TkT)

Avainsanat: Node.js, Deno, web-kehitys, web-ohjelmointi

Tässä kandidaatintyössä suoritetaan vertaileva analyysi sekä käytännön tapaustutkimus siirtymisestä Node.js:stä Denoon. Tavoitteena on arvioida, onko siirtyminen näiden ympäristöjen välillä suotava ratkaisu. Node.js on ollut hallitseva palvelinpuolen sovellusten kehittämisessä, mutta sen heikkoudet ja rajoitteet ovat vuosien varrella johtaneet uuden ympäristön, Denon, kehittämiseen. Ryan Dahl loi Denon vuonna 2020 korjatakseen Node.js:n puutteita ja rajoitteita.

Tutkimuksessa hyödynnettiin menetelmää, jossa suoritettiin sekä kattava kirjallisuuskatsaus että käytännön tapaustutkimus, joka sisältää full-stack sovelluksen palvelinpuolen siirtämisen Node.js:stä Denoon.

Tulokset osoittavat, että Deno on monissa tapauksissa erittäin toimiva vaihtoehto. Se on parempi etenkin sovelluksissa, jotka vaativat suuren määrän samanaikaisia yhteyksiä tai I/O-tehtäviä. Lisäksi Deno tarjoaa parempaa tietoturvaa, yksinkertaisemman konfiguroinnin sekä pakettien hallinnan ja sisäänrakennetut kehitystyökalut. Denon ongelmiksi näyttäytyi suurempi CPU-kuormitus sekä vähemmän kehittynyt yhteisö, jonka takia siirtymistä alustojen välillä tulisi harkita tilanteen mukaan.

Tutkimuksen perusteella esitetään ohjeistus kehittäjille, jotka harkitsevat siirtymistä Node.js:stä Denoon. Deno voi tuoda huomattavia hyötyjä ja parannuksia web-kehitykseen, mutta siirtymistä tulisi harkita tapauskohtaisesti, jotta voidaan varmistaa, että siirtyminen tuo todellista lisäarvoa ja hyötyä kyseisessä tilanteessa.

ABBREVIATIONS

NPM	Node Package Manager
I/O	Input Output
CPU	Central Processing Unit
ECMA	European Computer Manufacturers Association
API	Application Programming Interface
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URL	Uniform Resource Locator
JWT	JSON Web Token
JSON	JavaScript Object Notation
IDE	Integrated Development Environment
CSS	Cascading Style Sheets
TSC	TypeScript Compiler
CLOC	Count Lines of Code
KB	Kilobyte

Table of contents

Abstract

Abbreviations

1 Introduction ... 7
1.1 Objectives and goals ... 8
1.2 Thesis structure ... 8
2 Related research and background ... 10
2.1 History of Node.js ... 10
2.2 Node.js features, architecture and limitations ... 11
2.2.1 Node.js features and architecture ... 11
2.2.2 Node.js limitations ... 12
2.3 Deno ... 13
2.3.1 History and motivation for Deno ... 13
2.3.2 Features and architecture ... 14
2.3.3 Possible fallbacks ... 14
3 Research methodology ... 16
4 Comparative analysis ... 17
4.1 Summary of findings in Chapter 2 ... 17
4.2 Architecture comparison ... 17
4.3 Security comparison ... 18
4.4 Performance comparison ... 19
4.5 Developer experience comparison ... 20
4.6 Summary of findings ... 21
5 Case Study ... 23
5.1 Review of the original project ... 23
5.2 Migration process ... 24
5.2.1 Setup ... 24
5.2.2 Security and running options ... 26
5.2.3 Modules and imports ... 26
5.2.4 Deno’s built in tools ... 27
5.2.5 Coding differences ... 27

5.2.6 Deno’s challenges 30
5.3 Developer experience 30
6 Results and Discussion 32
6.1 Outcomes of comparative analysis 32
6.2 Outcomes of the case study 33
6.3 Discussion 34
6.4 Artifact: Migration guidelines 35
7 Conclusion 37
References 38

1 Introduction

For more than a decade, Node.js has been the dominant runtime for creating reliable server-side applications. Node.js was created back in 2009 by Ryan Dahl. It revolutionized backend development by introducing built-in asynchronous and I/O capabilities and by taking advantage of event loops for efficient performance. The way developers could achieve great performance with minimal effort and resources allowed for the wide adoption of the runtime. Node became the go-to option for many developers and is still very widely used. Over the years a vast ecosystem of tools and libraries and strong community has been developed around Node.js. [1], [2]

Ryan Dahl, the creator of Node.js presented a new JavaScript runtime Deno in May 2020. While Node.js and Deno share a fair amount of similarities, they have a significant amount of differences, and Deno was specifically aimed to address some of Node's shortcomings. JavaScript has evolved a lot since Node.js was originally released and a goal of Deno was to embrace the modern JavaScript standards and to address the long-standing security concerns of Node.js. [1], [2]

Deno is widely discussed on different online articles and blogs. Many developers are considering whether migrating from Node.js to Deno is a viable option [15]. There is a lot of academic research on Node.js and a growing amount of research on Deno. However, there is limited research done on migrating from Node.js to Deno, and most existing research focuses separately on Node.js or Deno.

This thesis aims to fill some of the gap in the current research by providing a comparative analysis and a practical case study of migrating a project from Node.js to Deno. By this approach this thesis seeks to help developers get information whether migrating from Node to Deno is a viable or sensible option, and to give developers insight into the differences of these runtimes.

1.1 Objectives and goals

This thesis aims to provide a comparative analysis of migrating from Node.js to Deno, both in terms of comparing their architecture, performance, security and developer experience, as well as conducting a practical case study of implementing a backend for a project in both runtimes. Through this approach, this research provides both theoretical and practical insight for developers. The main research question of this thesis is evaluating whether migrating from Node.js to Deno is a viable option and for whom?

To achieve this, the study focuses on these objectives:

1. To compare the architecture, performance, security and developer experience of Node.js and Deno to identify their differences.
2. To implement a backend project with both Node.js and Deno and compare the development experience and evaluate the process of migrating from Node.js to Deno.
3. To provide recommendations for developers who are considering transition from Node.js to Deno and assess the challenges and benefits of migrating.

1.2 Thesis structure

This thesis is structured in a way that allows the reader to go through the subject in a logical manner. The first chapter is the introduction for the thesis. This chapter provides some background on the topics of this thesis. It also covers the research objectives and goals.

In the second chapter an overview of Node.js and Deno is provided with history and details of their differences, key features and architecture. In addition, existing research is reviewed, and both runtimes are discussed particularly in the context of migration between them. The third chapter covers research methodology.

The fourth chapter presents the comparative analysis between Node.js and Deno. The comparison is based on various factors such as architecture, security, performance and developer experience. Both theoretical analysis and existing benchmarks are used to support this comparison.

In the fifth chapter, the case study of migrating a project's backend from Node.js to Deno is conducted. It covers the documentation of the process, challenges encountered and the overall developer experience.

The sixth chapter includes the results and discussion about both the comparative analysis and the practical case study. The feasibility and sensibility of migrating from Node.js to Deno is evaluated. In addition, advantages and disadvantages of migrating are discussed.

The final chapter concludes with the thesis summarizing the key findings from the research and the case study. Additionally in this chapter recommendations for developers considering the transition to Deno are provided including suggestions for areas for potential future research.

2 Related research and background

This chapter covers the background of Node.js and Deno more thoroughly. It examines the history, features and limitations of Node.js that led to the development of the new runtime Deno. In addition, related research and studies are reviewed to set the stage for the theoretical and practical comparison between Node.js and Deno.

2.1 History of Node.js

Node.js was developed by Ryan Dahl in 2009. Before Node.js, many server-side environments used synchronous, blocking I/O models. When dealing with high concurrency, this becomes a problem. Modern web applications can require high concurrency, and this was one of the things that Ryan Dahl recognized. The solution was to use an asynchronous, non-blocking I/O model that could allow servers to efficiently handle many simultaneous connections. [2]

Many server-side environments before Node.js such as Apache used multi-threaded blocking architecture that used a lot of memory and wasn't effective with high concurrency. Node.js took advantage of single-threaded event-based architecture that would decrease memory usage and allow for efficient performance with high concurrency [3]. Multi-threaded programming can be done successfully, but many developers agree that it is not easy and there are a lot of possible problems such as deadlocks and failure to protect resources shared among threads [4].

It seems that Node.js addressed the right issues in the web development world, since it became very popular, and many large companies adopted it. These large companies include PayPal, Walmart, LinkedIn and eBay [5]. According to Stack Overflow's annual developer survey, Node.js was the most used web framework among all respondents. Node.js peaked in popularity in 2020 but remains at the top [6].

2.2 Node.js features, architecture and limitations

Node.js has a lot of features that made it what it is today. It's important to have the needed information about Node.js features and limitations to fully understand why Deno was developed and what it was designed to improve.

2.2.1 Node.js features and architecture

As discussed, Node.js uses architecture that relies on event loops and a single thread. This simplifies the development process and takes out a lot of problems such as deadlocks, file and resource locking and thread communication issues. By using this single threaded event-based architecture, events are sent to the event loop, and the asynchronous queue executes the operations when resources are available. For this to work, all operations need to be non-blocking. This eliminates the need to wait for the completion of the operation. Event is sent and response is gathered later [2].

Node.js uses Google's V8 engine. In which JavaScript code is compiled directly into machine code making code easy to implement [7]. The V8 engine is constantly improved because Google has a lot of motivation for making the engine as fast as possible, to make Google Chrome more popular. This work on the V8 engine has allowed Node.js to quickly adopt the latest JavaScript features and get all the performance benefits [5].

Node.js has a lot of other features and strengths that have made it so broadly adopted among the developers. It's easy to learn and implement, and Full-stack JavaScript sounds compelling to many developers. Both server-side and client-side can be made by a developer who knows how to work with JavaScript. This can also be an advantage for companies not having to hire separate developers for front-end and back-end development [7].

Npm is the standard package manager used in Node.js. It is used to download and manage Node.js package dependencies. Packages and modules are the building block of Node.js applications and they provide a lot of functionality for developers [5].

2.2.2 Node.js limitations

Ryan Dahl saw several shortcomings in Node.js that became permanent technical debt over time because they were not addressed in time. In his mind Node.js was an insecure platform. There is no inherent security in Node.js and that could lead to unaware developers facing security issues because there is nothing stopping the use of their host computers network services [1]. In other words, there are no real restrictions with network access using Node.js.

The architectural choices of Node.js introduce a range of other security concerns as well. The way that Node.js is designed to take advantage of the event-based single threaded architecture is one of the main features that made Node.js so popular and widely used. On the other hand, it also brings the issue that any unhalted exception or blocking operation can halt the entire application. One mistake that disrupts the event loop can crash the whole server [8].

Server-side JavaScript lacks the sandbox mechanisms that are found on the client-side. Developers should be very cautious developing server-side applications but unfortunately, it's very easy to slip and introduce serious security vulnerabilities with some JavaScript features. This lack of sandbox mechanisms also introduces the added threat of malicious actors trying to manipulate the global state of an application [8].

The ecosystem of Node.js itself brings security concerns as well. The functionality of the platform can be extended with the use of third-party modules accessible through the npm repository. There is always the possibility of malicious installation scripts being embedded in the third-party modules. These installation scripts can execute unwanted things in the environment and usually npm is ran in the root which makes this that more concerning [8]. There are other factors that make the module system problematic. Npm being the de facto standard for package management in the Node.js ecosystem and the fact that it's a centralized, and privately controlled repository wasn't something that Ryan Dahl was fond of. The way manifesto files such as "package.json" files are handled, and the way modules are saved in the large "node_modules" folder are some features Ryan wished to get rid of as well [1].

Resource management in Node.js also poses security concerns [8]. Working with event loops means that when a heavy request comes to the queue, Node.js will set all the available CPU

to process it. Other pending requests will be thus bottlenecked, and this is why it's not very suitable for CPU-intensive tasks [9]. This also makes applications made with Node.js vulnerable to memory exhaustion and denial-of-service conditions with the slightest mistakes in handling files or other data. If attackers can cause prolonged processing times the event loop will get blocked and disrupt the service availability [8].

Node.js handles asynchronous operations mainly using callback functions. This is a way to manage asynchronous code but can also lead to very complex and nested code structures. This is often referred to as "callback hell". The use of async/await and promises have been the answer to this issue, but many legacy codebases still heavily rely on callbacks [7]. Heavy use of callbacks can lead to difficulties in error handling and make it difficult to trace and mitigate vulnerabilities [8].

2.3 Deno

Now that we better understand Node.js and its limitations, Deno can be covered in depth to understand its development, and to see how it answers to the problems and limitations of Node.js.

2.3.1 History and motivation for Deno

Ryan Dahl mentioned a lot of things he was not happy with Node.js in his famous talk, *10 things I regret about Node.js.* He covered issues related to the following aspects of Node.js: not sticking with promises, security, the build system, package.json, node_modules, requiring("module") without the extension ".js" and index.js [10]. From the issues that Ryan Dahl listed in his speech, it's easy to start understanding the motivation for him to start working on a new and improved runtime.

In the same speech he introduced the idea of Deno. He talked about what Deno aims to achieve and what type of technologies will be used. Security was the first thing that Ryan talked about. Deno aimed to make security better by taking advantage of JavaScript's sandboxing. Simplifying the module system was the next goal. He talked about TypeScript and how it would be used with built in support in Deno. Lastly, he talked about how he will

take advantage of the modern possibilities of software development to make Deno simpler than Node.js was [10].

2.3.2 Features and architecture

Deno was originally released in May 2020 with its first stable version 1.0.0. It is secure by default meaning that it uses the V8 sandbox and allows developers to handle the permissions in order to control the access of the code. TypeScript support is built in. Developers can use TypeScript without needing to do any extra configuration. Deno works with a single small executable file that has all the tools needed for writing applications. Deno also provides a set of developer tools, for example a linter, a formatter and a test runner. The issues with Node.js package management have been solved with Deno's standard library which aims to diminish the use of third-party packages. Additionally, it ensures compatibility with ECMAScript and browser environments [2]. All these features and design decisions seem promising and a great foundation for Deno to be a great improvement from Node.js.

The four main architectural choices and technologies that make Deno possible are the V8 engine, TypeScript, Tokio and Rust. Deno uses the same Google V8 high performance engine as Node.js. The TypeScript support makes development better and includes optional static typing to the language. Rust is a server-side language that focuses on performance and safety [2]. Tokio is an asynchronous runtime for Rust language. It is designed for writing stable network applications with including asynchronous APIs for TCP and UDP, timers and a multithreaded work-stealing scheduler [11].

2.3.3 Possible fallbacks

Deno improved security by making the runtime secure by default and allowing the developers to control the code permissions. This makes using Deno more secure but also the user experience can feel complicated and not so user-friendly. Although the standard library of Deno aimed to diminish third-party package use among developers, it must be stated that it's still very much in use depending on the developer. In other words, some of the Node.js security problems are still present in Deno, but importantly improvements have been made [12].

Deno is a fairly new runtime which obviously means that in some ways it is not as stable as for example Node.js that has been developed for over a decade. As time goes on and Deno keeps improving, it will get more stable. Deno is also not compatible with existing JavaScript packages and tooling due to the many changes that have been introduced in the making of it. Deno’s TypeScript compiler is one of the slowest things regarding the runtime. This is aimed to be fixed by migrating the compiler to Rust in the future [2].

All runtimes have limitations and concerns and ultimately it comes down to how they are handled by architectural choices and the development decisions of the runtime. Some of it comes down to pure preference as well.

3 Research methodology

In this chapter, the research methodology used in this thesis is covered. This thesis evaluates the viability of migrating from Node.js to Deno. To achieve this, the study employs a mixed-method approach that combines a detailed theoretical analysis and a practical case study. A thorough foundation of knowledge is produced with analysis of existing literature, benchmarks and documented experiences. The case study is focused on capturing hands on developer experience during the migration process.

Design science research methodology is the research methodology used in this thesis. Design science is designing and investigating artifacts within a specific context. The goal of design science is to address practical problems and to improve existing practices. In design science research, there are two kinds of research problems. Design problems are problems that call for a real change for improvement. The solution is a design, and many different solutions may be a possibility. The second kind of research problem is a knowledge question. When addressing a knowledge question, it is assumed that only one correct answer exists [13]. Distinguishing these problem types is fundamental for framing the research.

In design science research it is crucial to identify the research problem and thus gain understanding for the motivation of the research. Defining the objectives for a solution is also important. It is important to understand how the created artifact is expected to support the solution of the research. The artifact serves and contributes to the research and helps to ground the solution [14].

The artifact created in this study is a guideline for developers who are figuring out if migrating from Node.js to Deno is a viable option. The comparative analysis will give a foundation for understanding the ways Node.js and Deno compare in several ways. The case study of developing a backend for an application will give good insights and real-world indicators of the development process and experience. The evaluation of the artifact is used as the basis for answering the main research question of this study.

4 Comparative analysis

In this chapter, a comprehensive comparative analysis is conducted of Node.js and Deno. The goal of this chapter is to analyze key factors such as architecture, security, performance and developer experience. This analysis is done to further understand the potential differences between the runtimes and therefore get more insight into whether migrating from Node.js to Deno is a viable option. Both theoretical analysis and existing benchmarks are used to support this comparison. Additionally, this chapter will set the stage for the case study conducted in the next chapter.

4.1 Summary of findings in Chapter 2

The second chapter of this thesis offered an in-depth background for Node.js and Deno. Understanding the history of both runtimes and the motivation for Deno gives a basis for comparing the two head-to-head in this chapter. In the second chapter, key features, some architectural differences and possible fallbacks were discovered. In the following chapter, the two runtimes are compared more deeply.

4.2 Architecture comparison

The design philosophy of both runtimes is discussed broadly in the second chapter of this thesis. In this chapter the goal is to try and assess the cases where the differences in the architecture of these runtimes could affect the decision of migrating from Node.js to Deno.

In Node.js the event loop is implemented by C library “Libuv”. The event loop in Libuv runs constantly and when an event occurs, the corresponding callback is triggered by the loop. After the callback, it returns to waiting for new events. With this design Node.js is great for handling many concurrent connections. Additionally, the main event loop is kept responsive, even when handling potentially blocking tasks. This is done by managing a thread pool to offload operations that cannot be performed asynchronously. This way of implementing the event loops in Node.js is the way that applications can be highly scalable and responsive [16].

Deno does event loops a bit differently. Deno uses a third-party library Tokio for its event loops. With heavily relying on asynchronous programming and Tokio, Deno aims to minimize callbacks. Tokio is built with Rust and takes advantage of a multi-threaded scheduler that utilizes a work-stealing technique. This allows for applications to handle a large number of requests efficiently. This architectural design is fast and allows for responsive user experience with great possibilities for scalability [17].

In essence, Tokio built in Rust, features guarantees for memory safety and concurrency, and relies heavily on asynchronous programming. Tokio operates with an event-driven runtime with an integrated I/O driver, a multi-threaded scheduler and a timer. Tasks are lightweight asynchronous threads managed by the runtime, allowing for many concurrent tasks with minimal overhead. In Deno, Tokio is used to make synchronous tasks into asynchronous ones to manage the overall asynchronous runtime. The main strong points of using Tokio over Libuv, are getting rid of the so called “callback hell” of Node.js, the great handling of concurrency and the integrated safety with using Rust [16] [17].

Over time, the Node.js module system has become very complex and hard to maintain. The module system depends heavily on npm which initially was part of Node.js itself but they separated in 2014. Having a centralized package manager such as npm has been one of the issues of Node.js, because millions of applications depend on this single registry to survive. This is a liability and Deno answers to this issue with URL-based imports. With this Deno removes the need for messing with dependencies and installations [2].

Deno offers a wide range of built-in tools. These tools include a dependency inspector, a linter, a test runner, a V8 debugger, a code formatter, a documentation generator, a script bundler and a script installer [18]. These built in tools can provide true value for developers.

4.3 Security comparison

Node.js and Deno handle security differently. Node.js has an open security model that leaves a lot of security handling in the developer’s hands. Using Node.js securely depends on the code written, third-party modules used and awareness of threats [19]. Deno is secure by default. It uses the V8 sandbox and provides a strict permission model that allows for great handling of what the code has access to [2]. As stated in the second chapter of this thesis,

security is one of the main things Deno aimed to fix in Node.js. Added security is always welcomed by developers and this is one of the things that could motivate developers to migrate.

As discussed in Chapter 2, the ecosystem of Node.js itself brings security concerns as well. The way the module system is handled in Node.js can allow for multiple issues related to security. Malicious scripts can be embedded in the third-party modules and these scripts can execute unwanted things [8]. Deno on the other hand uses importing from the web. In other words, in Deno there is no need for depending on third-party modules [21].

The sandboxing in Deno provides a capability-based security model. It requires developers to explicitly grant permissions for different operations. This approach helps mitigate security vulnerabilities in applications. By default, Deno runs with no permissions and operations such as file access, network communication and environment variable access need to be allowed by the developer. The security model using sandboxing and permissions for sensitive operations mitigate risk for breaches and attacks in server-side applications [20].

4.4 Performance comparison

To evaluate the performance of Node.js and Deno it’s important to compare and analyze existing benchmarks. One illustrative benchmark from a Medium article [22] provides a great comprehensive comparison of the performance of these runtimes.

In this benchmark analysis, three core tasks are performed which are JWT authentication, database query performance and PDF generation, assessing CPU efficiency and resource usage [22]. This will nicely evaluate how these runtimes handle I/O intensive work and CPU intensive work. Key benchmarks from this Medium article [22] are illustrated in Figure 1.

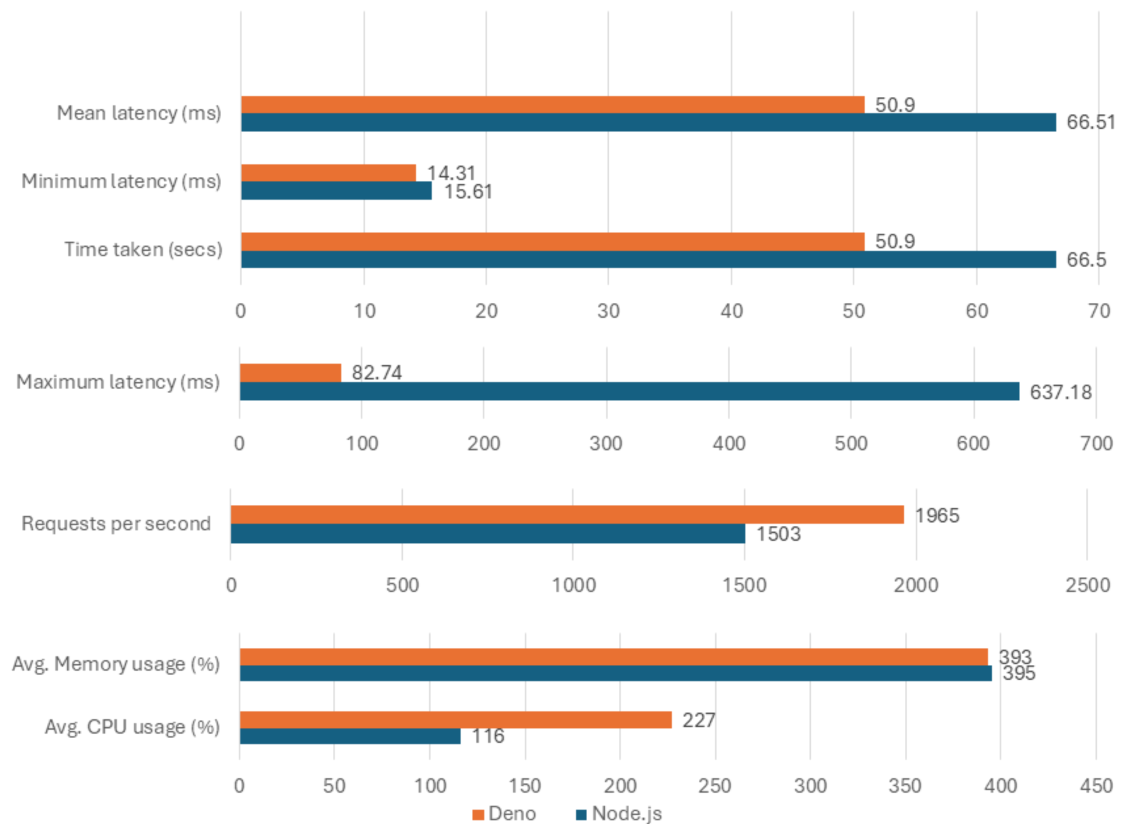

Figure 1: Node.js and Deno benchmarks

In each benchmark above, 100 concurrent connections are executed for 100k requests. The results show that Deno's performance is better than Node.js' in almost every benchmark. Deno is faster, requests per second is greater, latency is less, and memory usage is very much like Node.js. On the other hand, CPU usage is much greater than in Node.js. Based on these benchmarks, it can be said that Deno offers better performance all around, but CPU usage is way greater than in Node.js. 100 concurrent connections test the ability to handle high concurrent connections nicely and Deno seems to handle concurrent connections better than Node.js.

4.5 Developer experience comparison

Node.js has been popular for many years. With the popularity of the runtime and the strong community, it has evolved and improved a lot through the years. It is an open-source project

and while large corporations such as Google have contributed to its development, no one controls it. The ecosystem is strong, and documentation is very broad. JavaScript is a language that most developers have used and the documentation on JavaScript is also very broad [23] [24].

Deno has a strong ecosystem as well. It is an open-source runtime just like Node.js. It already has a fairly strong community and documentation. It prides itself on having native TypeScript support, zero configuration, security out the box and built-in tooling for developers [26].

The tooling of Node.js depends a lot on the development environment and third-party modules. With a good IDE, Node.js applications can be easily created, managed and debugged. Node.js does not have built in developer tools like found in Deno. Deno provides a comprehensive set of built-in tools that many developers can find to be very useful [18] [24]. This built-in approach can make developing applications simpler and additionally the security is better when the developer doesn't need to download multiple third-party tools. It can also be a time saver since with Node.js you must go through a lot of configurations for things like linting, formatting and testing [25].

4.6 Summary of findings

From the comparative analysis it is easy to see that both runtimes aim to achieve similar goals but differ in many important ways. These differences may influence one's decision of migrating. In this chapter, the analysis is summarized.

Both of the runtimes are designed to achieve very similar things and even though their architecture differs with some amount the architecture difference doesn't seem to be a very conclusive factor on deciding between the runtimes. It somewhat seems to come down to the applications' needs, what each runtime supports the best and developer preference. The module system, Denos event-loops and the additional developer tools in Deno can however be said to very conclusively be an improvement from Node.js.

Node.js leaves a lot of responsibility for safe coding to the developer, with its open security model. Deno on the other hand, uses a secure by default approach with sandboxing and a strict permission model to help developers write programs safely. Additionally, this security

model used in Deno addresses a lot of the vulnerabilities caused by Node.js third-party dependencies.

The performance comparison indicated that Deno is generally greater than Node.js in performance and in handling concurrency. It outperformed Node.js, but with one aspect that needs to be considered, which was the greater CPU usage.

Node.js prides itself on a mature ecosystem and extensive documentation. It benefits from multiple years of community development and enterprise help. Deno provides strong community and documentation as well as prides itself on native TypeScript support, zero configuration and built-in developer tooling.

In conclusion, both Node.js and Deno have their own strengths. Migrating from one to another might depend on the developer’s preference or the project requirements. Deno seems to have addressed many aspects of Node.js limitations and weaknesses and many of these improvements might be a deciding factor for a developer to migrate. This analysis sets the stage for the next chapter that presents the practical case study. In the case study migration from Node.js to Deno is further evaluated with a real-world development scenario.

5 Case Study

In this chapter, a case study is conducted of migrating a full-stack applications backend from Node.js to Deno. First the original project is reviewed to understand the starting point for this evaluation. After the original project is reviewed, the differences with implementation using Deno compared to Node.js are conducted. Lastly, the overall development experience is evaluated based on the work done with the project.

5.1 Review of the original project

Lappeenranta-Lahti University of Technology offers a full-stack programming course named *Advanced Web Applications*. In this course the final course project is to conduct a full-stack web application that mimics a Kanban board. In the project work students are encouraged to add a lot of their own functionality and innovative solutions to improve the application. In this case study, the focus will be on the back-end side but to understand what exactly needs to be handled on the backend, it is good to review the front-end functionality as well.

The front-end uses React with Vite as the build tool and CSS for styling. The basic idea of the application is to be a similar application to largely used Kanban boards. Users can create their own boards, in which they can create columns to hold cards. The cards act as notes of different kinds and allow users to track and manage different types of notes or to-dos for example. The functionality includes:

1. User authentication, with secure registration, login and logout processes utilizing JWT authentication.
2. Board management, with ability to create and edit users board and the content of the board.
3. Interactive user interface, including drag-and-drop ordering of the card elements, component editing and visualization of task statuses using checkboxes and colors.

The backend of the original project uses Node.js with Express and utilizes MongoDB as the database. In the upcoming study the idea is to try and implement the exact functionality of

this original project by keeping the front-end as similar as possible to the original one and to implement all the same functionality on the back-end side only now using Deno instead of Node.js.

5.2 Migration process

In this chapter the all-around developer experience of migrating a projects backend from Node.js to Deno is evaluated. The evaluation focuses on simplicity, efficiency, user-friendliness and the difference between the runtimes.

5.2.1 Setup

Setting up a basic Deno application is incredibly simple. The only requirements are downloading Deno, selecting a framework which in this case was chosen to be Oak, and lastly running the application. The fact that only one binary installation is required, and no centralized package manager is needed makes the set up process very clean and straightforward. Additionally, this makes the project structure much cleaner as well. In Figure 2 and Figure 3, the simple setup process of Deno is displayed and the final project structures are compared.

```
irm https://deno.land/install.ps1 | iex
```

TS firstprogram.ts 1 ×

TS firstprogram.ts > ...

```
1  import { Application } from "https://deno.land/x/oak/mod.ts";
2
3  const app = new Application();
4
5  app.use((ctx) => {
6      ctx.response.body = "Hello from Deno!";
7  })
8
9  await app.listen({port: 8000});
10
```

```
deno run --allow-net .\firstprogram.ts
```

localhost:8000

Hello from Deno!

Figure 2: Deno setup process for first test application

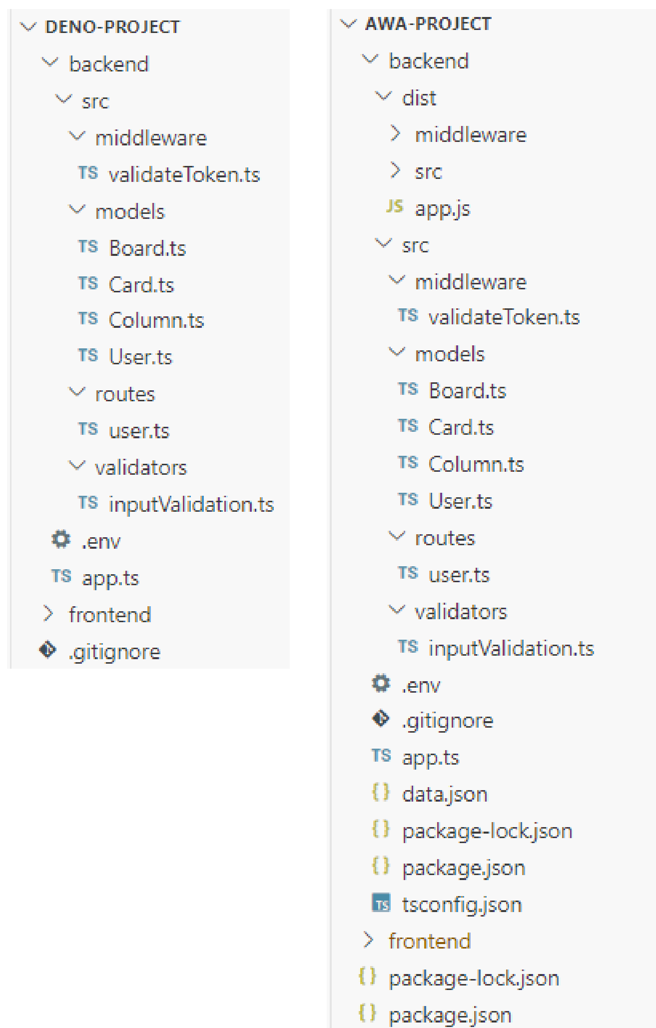

Figure 3: Deno vs Node.js final project structure

The built-in TypeScript support made the workflow even better. Unlike with Node.js there is no need for TypeScript compiling. Additionally, downloading a lot of modules and dependencies when using Node.js can be a hassle. Deno gets rid of all this extra work and all around makes development much more straightforward and simpler. Not having to handle dependencies, TypeScript compilers, module installations and other configurations allows developers to focus more on the actual coding aspect instead.

5.2.2 Security and running options

Deno runs in a sandboxed environment. When running programs on Deno, the developers have to specify permissions that the code has access to. The system is easy and secure, giving the developers more options for running their programs. With the original project the running options where complicated and managed in the configuration files and with tsc-watch configuration to mitigate the need of constantly running the code repeatedly when changes have been made. With Deno the running is made simple and just by adding the necessary flags, the program is easy to run.

```
deno run --allow-net --allow-env --allow-read --allow-write --allow-sys --watch app.ts
```

Figure 4: Deno running options

The system used by Deno to run programs is more secure and simple than with the usual Node.js configurations used, and it is great for beginners because it requires the developers to fully understand what permissions are needed to run the program. Additionally, it doesn't penalize beginners for not knowing what permissions to use since the program will not run before the permissions are granted. With understanding the requirements for permissions, it's easier to mitigate risk for security vulnerabilities. Additionally, with less permissions the program has, the safer it is. In contrast, the way that Node.js runs with full system access in contrast is not a very secure way of handling things.

5.2.3 Modules and imports

Deno uses a modern approach to module management that differs a lot from Node.js. With Deno the imports are made with URL-based references that don't require any additional configuration or hassle. In contrast, Node.js requires a lot more steps to import any modules to the application. With Node.js the developer must install the package that they want to use, download the needed type definitions and then import the module to their code. It is easy to get lost in the modules and one can easily have multiple modules downloaded that are not in use.

The negative side of the URL-based imports in Deno is that the developer needs to keep up with the changes in recommended modules and they must have knowledge of what to import

in order to get the actual functionality that is desired. Deno documentation is very important when developing applications using Deno and to gain the full security benefits of Deno, it is crucial to first gain knowledge of the module system and the runtime before implementing new applications.

5.2.4 Deno's built in tools

The use of the built-in tools of Deno is made very simple. These tools were used during the development of this project. The formatter and linter tools proved to be particularly helpful for the project.

The formatter tool is simple but effective. It provides all the formatting issues for all the files in the project and makes it easy to produce high quality and nicely formatted code. The formatting tool includes the option to automatically format the code or to first check the formatting errors and then fix them. The linter tool provides feedback for potential code quality issues. It is a great tool for finding possible issues before they start affecting the code. It finds issues that are easy to neglect and oversee.

5.2.5 Coding differences

Both the Node.js project and the Deno project include the exact same backend functionality, and they are very similar in many ways regarding the written code. However, there are many differences between the runtimes and how certain functionality is achieved. Obviously, the technology choices change things as well. For example, the different frameworks handle things a bit differently. The focus of this chapter is to analyze the main differences in the code of both projects without concentrating on any other differences except between the implementation in Node.js and Deno.

With Deno the different API calls are handled a bit differently. The Oak framework provides a single context object that includes both the response and request. Additionally, reading the request body is done asynchronously.

Node.js:

```
router.post("/api/user/register", registerValidator, async (req: Request, res: Response) => {

const {email,username,password} = req.body

    res.status(200).json(newUser)

} catch (error: any) {
    console.error(`Error during user register: ${error}`)
    res.status(500).json({ error: 'Internal Server Error' })
}
```

Deno:

```
router.post("/api/user/register", async (ctx: Context) => {

const body = await ctx.request.body({ type: "json" }).value;

const { email, username, password } = body;

  ctx.response.status = 200;
} catch (err) {
  console.error("Error during user registration:", err);
  ctx.response.status = 500;
  ctx.response.body = { error: "Internal Server Error" };
}
```

Figure 5: Node.js and Deno API calls

Deno uses top-level asynchrony out the box. Node.js relies on internal promise handling or callbacks.

Node.js:

```
app.listen(port, () => {
    console.log(`Server running on port: ${port}`)
})
```

Deno:

```
console.log(`Server running on http://localhost:${port}`);
await app.listen({ port });
```

Figure 6: Node.js and Deno application startup

To analyze the code amount used with both runtimes, the CLOC tool was used [27]. This tool quantifies code by counting blank lines, comment lines and source code lines to get a clear size of the codebase. The tool revealed a significant difference in the code volume between the two backend implementations. This difference is displayed in Figures 7 and 8.

```
      29 text files.
      27 unique files.
       2 files ignored.

github.com/AlDanial/cloc v 2.04  T=0.49 s (54.8 files/s, 9126.3 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
JSON                             4              8              0           2904
JavaScript                      15              0              5            881
TypeScript                       8            176              9            514
-------------------------------------------------------------------------------
SUM:                            27            184             14           4299
-------------------------------------------------------------------------------
```

Figure 7: Node.js backend CLOC results

```
       9 text files.
       8 unique files.
       1 file ignored.

github.com/AlDanial/cloc v 2.04  T=0.08 s (103.7 files/s, 9905.6 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
TypeScript                       8             94              5            665
-------------------------------------------------------------------------------
SUM:                             8             94              5            665
-------------------------------------------------------------------------------
```

Figure 8: Deno backend CLOC results

The results show that the Node.js codebase is far more complex. It includes JSON files, JavaScript and TypeScript with a total of 27 files and 4299 lines of code. In contrast, the Deno codebase has only 8 TypeScript files and 665 lines of code while achieving the exact same functionality. However, it is important to note that most of the lines from the JavaScript files in the Node.js implementation come from the compiled TypeScript code, and The JSON files are mostly configuration code. The actual code amount that is written in the implementations are actually very similar to each other. The key takeaway from this analysis is to realize that with Node.js the overall codebase size is significantly larger compared to Deno.

The size of the backend applications is significantly different. The Node.js backend is 165 KB in size. In contrast, the Deno backend is 21 KB in size. This shows how effective the modern approach of Deno is and how the native TypeScript support streamlines the coding.

The database handling is different in the projects as well, but this is not specifically due to the different runtimes. This is more due to the different methods of handling the MongoDB

databases. As stated, the coding is very similar with Deno than it is with Node.js. Some of the coding differences are purely due to the ways in which things are handled differently in the projects and not due to the runtimes. If a developer is used to using Node.js the learning curve to using Deno is not very steep and the difference in the actual coding is probably not going to be the deciding factor for migrating.

5.2.6 Deno's challenges

The maturity of the Deno community is starting to be in good shape. However, it is nowhere near Node.js. This is very understandable, since Node.js has been around for much longer and as stated before in this thesis it is largely more used than Deno. There are many more examples and discussions to be found of Node.js than Deno. Deno's own documentation is great, and it provides most of the information that one needs to develop great applications using Deno. In conclusion, a developer must rely more on Deno's own documentation than the community compared to using Node.js.

From the project done in this study other issues were hard to find from using Deno instead of Node.js. The runtime is very simple and doesn't increase the amount of hassle or problems when using it.

5.3 Developer experience

The migration process from Node.js to Deno involves technical adjustments and brings a new perspective on developer experience. In this section, the development experience is evaluated based on the project work.

The most noticeable difference between the runtimes seen during the project was the simplified setup process and the streamlined configuration with using Deno. Having a single binary installation, built-in TypeScript and minimal configuration, makes development super easy to start. Deno takes out a lot of the hassle that Node.js brings with it.

The built-in tools that Deno include bring only more value for the developers. The tools are user-friendly, efficient and can help developers learn to produce higher quality code. The security model with Deno requires developers to understand the code permissions needed

and helps unaware developers to be more aware of potential vulnerabilities in the application.

When developing a Deno application, using the top-level asynchronous support can mitigate the need for relying on callbacks. Overall, the coding experience is pretty similar to using Node.js, but there are some differences in how certain functionality is achieved. The learning curve is not too immense and a developer that is used to Node.js will be able to learn Deno rather quickly. The biggest things to learn when transitioning are the importing, security model and the documentation of Deno.

Deno's module system, based on URL imports and a decentralized approach to dependency management, takes out the need for a centralized package manager such as found in Node.js. This simplifies the project structure and minimizes the complexity of handling dependencies.

In summary, the transition from Node.js to Deno offers developers several advantages regarding developer experience. The runtime offers tooling, asynchronous patterns and simplified processes to make programming more efficient and better. From a developer perspective Deno seems to be a very appealing option to produce a backend application.

6 Results and Discussion

In this chapter, the outcomes of the comparative analysis and the practical case study are synthesized. The discussion focuses on the feasibility and advisability of migrating from Node.js to Deno. Additionally, this chapter presents a guideline for developers who are considering migration.

The research aimed to assess whether migrating from Node.js to Deno is a viable option for developers. The findings suggest that while both of the runtimes share similarities, they share a lot of differences as well. The differences in architecture, security, performance and developer experience are good indicators for developers to consider the possibility of migration.

6.1 Outcomes of comparative analysis

From the comparative analysis, we can divide the findings into four categories architecture, security, performance and developer experience. The findings are as follows:

Architecture:

- Node.js uses Libuv for its event loop and relies more on callbacks than Deno. Relying on callbacks can lead to more nested complex code structures.
- Deno uses a Rust-based Tokio for its event loop. This offers a more modern, work-steeling scheduler and supports top-level asynchronous operations. Deno's URL-based module system simplifies dependency management.

Security:

- Node.js uses an open security model where the responsibility of security is largely dependent on the developer. An open security model can lead to security vulnerabilities more easily.
- Deno is secure by default and works in a sandboxed environment. The applications made with Deno require the developer to grant code permissions which encourage developers to understand the possible risks and vulnerabilities.

Performance:

- The benchmarks suggest that Deno outperforms Node.js in almost every way in a high concurrency application but requires more CPU usage.
- Deno's modern technological choices and improvements in the way the runtime is designed make this improvement in performance possible.

Developer experience:

- Node.js benefits from a very mature ecosystem and community. The reliance of third-party modules and tools can increase complexity.
- Deno offers built-in TypeScript support and developer tools which reduces the complexity of the configuration and coding process.

6.2 Outcomes of the case study

The practical case study conducted by migrating a full-stack application's backend from Node.js to Deno gave several important insights for this study. The findings can be divided into:

Simplified setup and configuration:

- Deno's single binary installation made starting simple.
- The lack of centralized package manager made project structure simpler, and less configuration was needed.

Built-in tools:

- The built-in tools of Deno helped with the programming process and mitigated issues in the code.
- Especially the linter and formatter where very useful and these tools can bring a lot of value for developers.

Module system and security model:

- The URL imports made the module handling very manageable and simple.

- The security model was brilliant and running the programs with the wanted permissions were easy.

Negative side:

- The less mature community of Deno developers makes a developer more dependent on Deno’s own documentation and less concise coding examples is harder to find compared to Node.js.
- Some functionality is achieved differently with Deno compared to Node.js and thus a developer who is migrating needs to learn the new way of conducting things.
- The module system is simple and effective, but a developer needs to be up to date with versions and possibly update their application with new changes.

6.3 Discussion

The findings support the conclusion that the migration from Node.js to Deno seems to be a very feasible and advantageous option. A project that is dependent on good security and modern development practices can gain real value from migrating.

For applications that require a large use amount of the Node.js legacy libraries or is very CPU-bound may however benefit from staying with Node.js. Deno is better for applications that are heavily I/O-bound and as mentioned where security needs to be strict in the runtime environment.

Deno simplifies a lot of the setup and configuration hassle, but there is a slight learning curve for developers starting to use Deno. However, if a developer is familiar with Node.js or other similar runtimes the learning process is going to be fairly minimal. Deno’s community is growing but there is a significant difference in the maturity of the community support with the runtimes.

6.4 Artifact: Migration guidelines

Based on the findings from the comparative analysis and the case study, the following guideline has been developed. The guideline aims to assist developers who are considering migration from Node.js to Deno.

Before migrating it's crucial to determine the requirements for the application. Migration should bring real value for it to be a reasonable decision. If the use of legacy Node.js libraries or other functionality is used that can't be achieved with Deno naturally one should stick with Node.js.

Understanding the security needs of the application is important. If the knowledge about security is limited, Deno is always a safer option with its built-in security. Deno will also help with mitigating vulnerabilities that even an experienced professional could easily neglect.

Deno's built-in tools can be an aspect that could benefit many from migration. It's important to identify if these tools could bring more value in the development process. These tools can help with mitigating issues and with testing the application effectively.

The documentation and community support are stronger with Node.js. However, Deno is well documented and with the experience gathered in this study the less mature community and documentation do not cause large issues in development. Node.js can be seen stronger in this field but Deno is evolving at a fast rate.

If a developer is not happy with the way modules and dependencies are handled with Node.js, and how it can be messy and bring unnecessary hassle into development, Deno's systems handle this far better. From the initial setup process to all the module management during a project Deno makes things simple.

Largely the preference of a developer is going to be a deciding factor. If one is brilliant at using Node.js and knows it extremely well, there must be real issues that are tried to be solved with Deno in order for them to migrate.

Table 1. Runtime comparison with key migration considerations.

Consideration	Node.js	Deno	Migration benefit
Security	Open security model and all permissions granted by default	Sandboxed environment and security by default	Added security and a better way of mitigating vulnerabilities
Performance	Less CPU-usage	Better performance with high concurrency and with I/O bound applications	Better performance with most cases in web development
Developer experience	Mature community and documentation	More simple and modern environment	Ability to focus more on the actual coding aspect and less configuration hassle
Module management	Relies on centralized package manager such as npm	URL-based imports giving a simpler and cleaner project structure and module management	Simplicity and more modern way of handling modules
Codebase	Larger in terms of lines of code and in data size	Less lines of code and smaller data size	The native TypeScript support and modern configuration handling makes the Deno codebase smaller and simpler

In conclusion, it should be evaluated on a case-by-case basis whether migration from Node.js to Deno is the right decision. This guideline provides a structured framework for developers to decide if migration could bring real added value for them. If there is real value to be gained from migration, Deno can be highly effective runtime for web-developers.

7 Conclusion

This thesis compared Node.js and Deno as runtimes and examined the differences that could indicate to migration being a feasible option for developers. The objective was to develop a guideline for developers who are considering migration between the runtimes. The aim was to first set a great background for the study by going through the related research from both runtimes, focusing on history and the different implementation of the runtimes.

There has been a lot of research done with Node.js and a fair amount with Deno. However, very limited studies have been conducted about analyzing when a developer should migrate from Node.js to Deno. This study bridges an important gap in the research about the runtimes and provides real value for the developers who are considering migration.

This study focused solely on the runtime differences and the practical case study only on the implementation of a backend application. The frontend implementation and differences should be studied and documented in the future. Additionally, the benchmarks evaluated in this study were derived from a single study and while providing good indication, the performance of the runtimes should be further evaluated.

The artifact was a successful implementation with great findings from both the comparative analysis and the practical case study. It gives good guidance for developers to consider when interested in migration.

From the findings of this study, it can be concluded that Deno can provide a lot of new value for developers with its security model, module handling, modern architecture and built-in tooling. Deno has answered nicely to the outdated technology choices and implementation faults of Node.js.

In the end it’s crucial to evaluate the migration decision case-by-case. To make an informed decision, the specific requirements and constraints need to be evaluated. This study serves as a great starting point for developers to evaluate migration between the runtimes and encourages further research in this evolving field of web-development.

References

[1] F.Doglio, *Introducing Deno: A First Look at the Newest JavaScript Runtime, First Edition*. Apress, 2020.

[2] A. P. dos Santos and M. Loureiro, *Deno Web Development: Write, Test, Maintain, and Deploy JavaScript and TypeScript Web Applications Using Deno, First Edition*. Birmingham, UK: Packt Publishing, 2021.

[3] “Node.js Introduction [Video],” YouTube. [Online]. Available: https://www.youtube.com/watch?v=ztspvPYybIY [Accessed: Feb. 11, 2025].

[4] S.Tilkov and S.Vinoski, “Node.js: Using JavaScript to Build High-Performance Network Programs”, IEEE Internet Computing, vol. 14, no. 6, pp. 80-83, Nov. 2010.

[5] D.Herron, *Node.js Web Development: Server-Side Web Development Made Easy With Node.js 14 Using Practical Examples and Expert Techniques, 5th Edition.* Packt Publishing, 2020.

[6] “Stack Overflow Developer Survey 2024 – Most Popular Technologies,” [Online]. Available: https://survey.stackoverflow.co/2024/technology/#most-popular-technologies [Accessed: Feb. 11, 2025].

[7] “Advantages and Disadvantages of Node.js in 2025,” GraffersID, [Online]. Available: https://graffersid.com/node-js-advantages-and-disadvantages/ [Accessed: Feb. 12, 2025].

[8] A. Ojamaa and K. Düüna, "Assessing the security of Node.js platform," *2012 International Conference for Internet Technology and Secured Transactions*, London, UK, 2012, pp. 348-355.

[9] ”The Pros and Cons of Node.js in Web Development,” GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/the-pros-and-cons-of-node-js-in-web-development/ [Accessed: Feb. 12, 2025].

[10] “10 Things I Regret About Node.js – Ryan Dahl – JSConf EU [video],” YouTube. [Online]. Available: https://www.youtube.com/watch?v=M3BM9TB-8yA [Accessed: Feb. 13, 2025].

[11] P.Krill, “Tokio Rust runtime reaches 1.0 status”, InfoWorld.com, SanMateo, CA, Jan.8, 2021. Available: https://www.infoworld.com/article/2261631/tokio-rust-runtime-reaches-10-status.html [Accessed: Apr. 10, 2025].

[12] A. Alhamdan and C.-A. Staicu, “Welcome to Jurassic Park: A Comprehensive Study of Security Risks in Deno and its Ecosystem,” in Proc. Network and Distributed System Security Symposium (NDSS) 2025, San Diego, CA, USA, Feb. 23-28, 2025, doi: 10.14722/ndss.2025.23284.

[13] R.J.Wieringa, *Design Science Methodology for Information Systems and Software Engineering,* 1st ed. Berlin, Heidelberg, Germany: Springer, 2014.

[14] J. vom Brocke, A.Henver, and A.Maedche, *Design Science Research. Cases*, 1st ed, Cham: Springer International Publishing AG, 2020.

[15] “What are the reasons for using Deno instead of Node.js or other alternatives like Go? Is security a factor in this decision?,” [Online]. Available: https://www.quora.com/What-are-the-reasons-for-using-Deno-instead-of-Node-js-or-other-alternatives-like-Go-Is-security-a-factor-in-this-decision [Accessed: Feb. 24, 2025].

[16] N.Marathe, “An Introduction to libuv, Release 2.0.0,” Dec.01, 2015. [Online]. Available: https://nikhilm.github.io/uvbook/An%20Introduction%20to%20libuv.pdf [Accessed: Feb. 24, 2025].

[17] “Internals of Deno: Tokio,” [Online]. Available: https://choubey.gitbook.io/internals-of-deno/architecture/tokio [Accessed: Feb. 24, 2025].

[18] “How to Use Deno’s Built-in Tools”, [Online]. Available: https://www.sitepoint.com/deno-built-in-tools/ [Accessed: Feb. 25,2025].

[19] “Is Node.js Secure?”, [Online]. Available: https://www.nodejs-security.com/blog/is-nodejs-secure [Accessed: Feb. 26,2025].

[20] “What is Deno and how to use its sandbox?”, [Online]. Available: https://www.zaynetro.com/post/what-is-deno#sandbox [Accessed: Feb. 26,2025].

[21] “Security and permissions”, [Online]. Available: https://docs.deno.com/runtime/fundamentals/security/ [Accessed: Feb. 26,2025].

[22] “Node.js vs Deno vs Bun: Benchmark for a real-world case – JWT, Postgres, PDF gen”, [Online]. Available: https://medium.com/deno-the-complete-reference/node-js-vs-deno-vs-bun-benchmark-for-a-real-world-case-jwt-postgres-pdf-gen-9fbd94bb9a83 [Accessed: Feb. 27,2025].

[23] “Node.js v23.9.0 documentation”, [Online]. Available: https://nodejs.org/docs/latest/api/ [Accessed: Mar. 6,2025].

[24] “Understanding Node.js Ecosystem and Tooling”, [Online]. Available: https://www.profoundlogic.com/understanding-node-js-ecosystem-and-tooling/ [Accessed: Mar. 6,2025].

[25] “All-in-one tooling”, [Online]. Available: https://docs.deno.com/examples/all-in-one_tooling/ [Accessed: Mar. 6,2025].

[26] “Deno”, [Online]. Available: https://deno.com/ [Accessed: Mar. 6,2025].

[27] AlDanial, “Cloc – Count Lines of Code,” GitHub, v2.04, Jan. 31, 2025. [Online]. Available: https://github.com/AlDanial/cloc [Accessed: Apr. 7, 2025].