TARTU UNIVERSITY

FACULTY OF SOCIAL SCIENCES

NARVA COLLEGE

STUDY PROGRAM “INFORMATION TECHNOLOGY SYSTEMS DEVELOPMENT”

Pavel Losev

COMMAND-LINE APPLICATION FOR RESOLVING COMPATIBILITY ISSUES IN JAVASCRIPT LIBRARIES THROUGH AUTOMATION TOOLS.

Bachelor’s thesis

Supervisor: Andre Sääsk, M.Sc.

Narva 2025

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Pavel Losev,

Annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose: “Command-line application for resolving compatibility issues in JavaScript libraries through automation tools”, mille juhendaja on Andre Sääsk, reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commons litsentsiga CC BY NC ND 4.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.

Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.

Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Pavel Losev

17.05.2025

Contents

INTRODUCTION

JavaScript is the primary programming language for the web. The main package manager and registry for JavaScript is called npm. The npm registry has been essential for developers, hosting over 3 million packages[1], making it the largest repository in the world[2]. However, npm, despite it having a critical role in the whole web infrastructure, faces significant challenges, such as compatibility issues between different libraries, major risks of supply chain attacks and bloated dependency graphs[3].

All of this makes it especially difficult to maintain large-scale projects and perform detailed security audits. This thesis dives deeper into the JavaScript ecosystem and proposes a solution prototype in a form of a command-line application. It would help developers save a significant amount of time by automating the resolution of compatibility issues among npm packages.

Problem

Due to the rapid evolution of JavaScript and lack of any conventions when it comes to structuring modules, the library landscape became very diverse and difficult to stay compatible between each other. Different libraries use different module systems[4], rely on different language features, certain libraries don't work in old browsers, others fail to install on newer versions of Node.js, and so forth. The problem multiplies as more dependencies are introduced to a project. It works the other way around as well when a library gets popular, and it becomes difficult to sustain compatibility across different environments and downstream projects. Manual compatibility maintenance takes away development time, which could instead be spent developing business logic and implementing new features.

Solution

A prototype of a specialized command-line application for automation of compatibility resolution will be developed. The prototype will implement a new approach to the JavaScript library compatibility problem that is different to existing solutions.

Goal

The final project will be in the form of a command-line application that would analyse the source code of a library as well as it's dependencies, and based on gathered information, automatically configure project's manifest and runtime requirements.

Tasks

- Dive into the history and the overall landscape of the JavaScript package ecosystem to better understand the origins of struggles web development is facing today.
- Review existing solutions to the problem and compare their flaws and advantages.
- Describe an alternative approach to the issue.
- Develop a basic prototype of the command-line application and test it against real world projects.
- Evaluate the effectiveness of the application.

1. RESEARCH AND DESIGN

1.1. Brief history of JavaScript and npm

According to GitHub, JavaScript stays as one of the most used programming languages in the world[5].

JavaScript was developed by Brendan Eich in 1995 for a browser called Netscape Navigator. The language meant to provide interactivity to websites, as well as to be accessible for people with no prior programming experience. Fast forward to the modern times, it became one. According to a W3Techs survey, the overwhelming 98.9% of all websites in the world use JavaScript[6].

For a long time, JavaScript had no standard way of packaging and delivering libraries and other third-party code. This changed in 2010, when npm appeared, just one year after the first release of Node.js, a server runtime for JavaScript. npm has become the de-facto standard for hosting and publishing JavaScript packages. In March 2020, npm was acquired by GitHub which is a subsidiary of Microsoft. After the acquisition, the registry has received numerous improvements such as support for 2-factor authentication and code signing, as well improved vulnerability monitoring.

1.2. The state of npm

Npm, being the largest package registry in the world, is hosting over 3 million packages and the number only keeps growing. This makes it the largest package registry to ever exist.

The registry has 3 main competing package managers: npm (the client), yarn and pnpm. All three are being actively used, while npm continues to be the most downloaded since it is shipped together with Node.js distributions and pnpm is rising in usage due to its efficient storage mechanisms.

JavaScript modules on the registry are packaged in multiple ways, the most common being CommonJS, a module system that appeared before JavaScript got its own standard module system, which is called ES Modules (ESM). Many popular libraries now are distributed as both CommonJS and ESM for maximum compatibility and interoperability. A rather small, but significant amount of npm packages uses native C++ bindings to extend Node.js core APIs.

1.3. Core problems

The architecture of the registry has been heavily criticized due to its wastefulness of storage space, massive complication of the module resolution algorithm and deviation from browser semantics. One of the creators of Node.js, Ryan Dahl in his JSConf EU 2018 talk[7] admitted that those design choices were a mistake.

As a derivative from the faulty "node_modules" design, module duplication became a large problem as well. Different packages might depend on the same module of different versions, which loads duplicate module instances in memory, wastes disk space and slows down installation time. Semantic versioning ranges and package manager functionality for deduplication partially address the situation, but don't fully solve it, since libraries might depend on semver-incompatible versions of the same library.

Another major issue is security. npm does not put any kind of barriers or compliance checks that would prevent malicious actors from publishing malware on the registry. It resulted in several incidents, including adding malicious code and RCE exploits as dependencies of popular packages[8]. This was partially fixed after GitHub's acquisition by implementing automatic vulnerability scanning[9], as well as security-enhancing features such as code signing. It remains a massive problem that can't be fully resolved due to npm's design and ecosystem complexity.

One of the day-to-day struggles that JavaScript developers face is compatibility and interoperability. Unfortunately, npm offers no methods of verifying runtime support, leaving that to a developer to figure out. In newer versions of Node.js certain libraries break due to usage of outdated native C++ modules[10][11], thus preventing developers from upgrading their infrastructure and lagging new features and security patches. A vast amount of massively used libraries with millions weekly downloads are abandoned and have not been updated for years, leaving developers with unfixed bugs that stay unpatched for long periods of time[12].

1.4. Community feedback

To validate the impact and importance of presented issues, several JavaScript library and tooling maintainers and authors have been interviewed. The libraries vary in their usage and popularity as well as complexity, so that there is a a less biased and more

diverse overview of actual current struggles related to JavaScript library development and maintenance.

In an informal chat with Lars Kappert, the creator and lead maintainer of Knip, a tool for decluttering JavaScript projects with over 10 million monthly downloads[13], highlighted a few major struggles when dealing with maintenance of Knip. In his experience, it repeatedly happened that the same concepts had to be explained and that was also caused by the users of the tool not having a clear understanding of how the tool works and what it does. Issue reproduction is another large pain point, as most of the time when there is a problem or a bug with Knip, either it does not get reported at all, or if it is reported, a clear step by step reproduction is missing, or the steps provided are not actually reproducing the reported error.

In Lars opinion, one of the issues that npm ecosystem is struggling with is ecosystem fragmentation and it being very dynamic. Due to the lack of basic developer tools inside Node.js itself, many mutually incompatible or partially compatible tools emerge, while in other languages or runtimes they are shipped together with the rest of the tools. While the situation is slowly improving in the past few years with things like Node.js including a test runner and implementing basic utilities inside Node.js itself, community is usually not involved and as a result, certain API design deviates too much from industry standard tools which adds additional friction.

When approaching compatibility testing, he prefers to do it manually. He avoids the usage of polyfills, emulators or transpilers. Lars states that “It's not ready if the actual code didn't run in the supported environments”.

Bjorn Lu, the creator of publint and a former Astro maintainer, expressed similar concerns. Bjorn provides an example of a series of radical changes in the ecosystem, such as React introducing RSC and all the libraries having to adjust to this new feature, new language features pop up every year while runtimes and browsers barely keep up with implementing them, and many other similar situations. At the same time, JavaScript ecosystem is the largest and offers a solution for almost every problem and gives you freedom of choice over tools.

Both Lars and Bjorn think that it would be nice to have an environment-aware tool that understands the supported globals and modules. The project that is being built to

demonstrate a concept of a solution for the compatibility issues falls within this category of tools. This means that there is a real need for a tool like this, and the demand is validated by actual library authors and maintainers.

James, a maintainer of highly popular npm packages such as Chai, Chokidar and parse5, claims that issue management takes a lot of time, and at certain moments as a library author issue triaging consumes more time than actual maintenance, such as fixing bugs. Unlike the previous JavaScript developers interviewed, James highlights dependency bloat as the core plague of npm. Most popular libraries are written by the same number of authors, and they are not interested in aiding the situation by shrinking their dependency graphs and forcing everyone to use them or providing a lighter minimal alternative to a specific package. Unlike Lars, James prefers using transpilers to compile source code down to the most compatible version, to allow support for a wider range of environments while not changing the code. He also enjoys using esbuild for transpilation because it is way faster than tools like Babel.

The maintainer wishes there would be more projects using profilers, and more tools around performance profiling. Tools that produce flame graphs help detect waterfalls and memory leaks.

Overall, while most of the interviewed JavaScript maintainers have a shared problem with ensuring compatibility, they use and implement different approaches. 2 of the 3 maintainers validated the demand for a tool that statically analyzes the code and dependencies and ensures runtime support.

1.5. Solutions, approaches and various other efforts

There is a diverse set of issues with the npm registry and JavaScript ecosystem as a whole. This thesis focuses on the compatibility problem in the context of JavaScript library development.

1.6. Babel

Babel is a JavaScript transpiler that is used to convert modern ECMAScript code to a backwards-compatible version of JavaScript[14]. It transforms new or non-standard language features to a more widely supported syntax and injects polyfills for APIs that are

not universally available yet. Babel has been widely relied on in the JavaScript ecosystem as a build tool that helps produce JavaScript code that is compatible among the most browsers, as well as being able to use cutting edge or non-standard syntax that is not present in JavaScript by default, such as JSX.

A few popular JavaScript libraries depend on Babel runtime to ensure compatibility with older browsers and Node.js versions[15], while not duplicating output code with multiple identical implementations of language features. Despite the fact that modern libraries no longer use Babel runtime as a direct dependency, its download count only keeps increasing, due to a few large projects still depending on it.

Over the past few years, Babel has been criticized due to performance downgrades compared to pure native code, as well as inflating the bundle size with polyfills and syntax transformations that are no longer needed, since modern browsers support the majority of JavaScript language features already.

To validate this claim and due to a lack of up-to-date benchmarks on modern JavaScript build tools that contain all the necessary data, a performance benchmark across 3 different transpiler tools was conducted. The two other transpilers other than Babel are SWC, which aims to be a drop-in replacement of Babel written in Rust, and esbuild, which is a general-purpose bundler for the web. In order to properly compare all of the three tools, a relatively new and widely used library was chosen as a test subject – neotraverse[16]. The library makes use of the latest JavaScript features such as private identifiers and optional chaining, which makes it a decent sample for gathering relevant metrics. Before conducting the benchmark, the original source code written in TypeScript was compiled to JavaScript. It was done to exclude TypeScript-specific compilation from performance estimates and to improve benchmark accuracy. All 3 transpilers target the ES2015 version of JavaScript.

Three different metrics were measured: code output size, the time it took to transpile the library, and how well the transpiled code performs when it is consumed as a library. A sample code to measure transpiled code performance is a simple implementation of serializing a JavaScript object, similar to how JSON.stringify works.

Hyperfine[17] was used to measure build time, and **mitata**[18] was used to measure runtime code performance. **Hyperfine** was used with 3 warmups prior to improving

accuracy, since the operating system caches filesystem calls. Tests were run on a Macbook Air 2022 with an M2 Apple Silicon CPU. All tests are run with Node.js 22.13.1.

Table 1. JavaScript transpilers benchmark

Transpiler	Output size	Transpilation time	Transpiled code performance
Babel	16KB	449.3ms ± 9.9ms	2.53 µs/iter
esbuild	12KB	196.2ms ± 13.3ms	4.87 µs/iter
SWC	24KB	261.5ms ± 1.7ms	4.73 µs/iter
Source code	12KB	-	2.03 µs/iter

Observed results reflected in Table 1**Error! Reference source not found.** demonstrate that Babel takes around twice as much on average as the alternatives. One of the unexpected outcomes is the fact that Babel's transpiled code performs better than other transpilers' code and is only slightly slower than native. However, all of the build tools needlessly increase code output size, since the default target of babel, ES2015, is out of date from the version that modern browsers implement. The transpiled features such as private identifiers have been available in all runtimes and browsers for more than 5 years.

Babel environment preset automatically only targets browsers with more than 0.2% global usage by default. Certain browsers with global usage less than one percent lag behind despite major browsers having those features implemented for many years. As a result, unnecessary polyfills and syntax transforms get included in the bundle anyway.

As the code base grows, transpiler's code transforms accumulate, resulting in larger bundle sizes and longer code transpilation time. Libraries use Babel to preserve compatibility with older runtime versions, although in certain cases there is no need to preserve compatibility with versions older than half a decade, when only recent and most used browser or runtime versions are required to be supported.

1.7. esbuild

Esbuild is one of the fastest bundlers for the web, written in Go. Even though it started as proof of concept, it later became of the most used bundling tools.

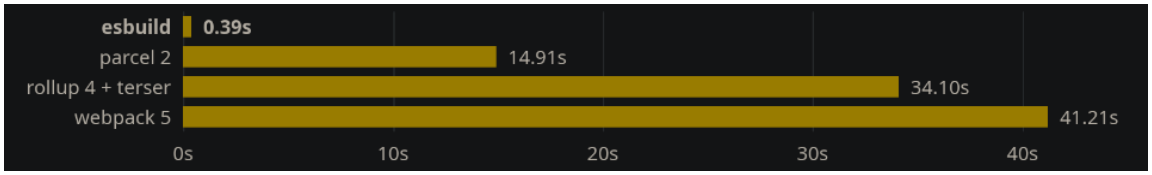

Figure 1. esbuild JavaScript bundling performance

The bundler supports various compilation targets and implements manual polyfill injection out of the box[19], unlike Babel where a preset needs to be installed and configured first. esbuild targets various browsers and JavaScript engines, most of which are also supported by the Babel environment preset. It also allows targeting a specific version of ECMAScript, as well as combining multiple targets without needing to install any additional presets or plugins. While esbuild does not integrate with Browserslist out of the box like Babel does, there is a plugin[20] that provides limited support for it.

Higher level build tools use esbuild under the hood due to its extremely high performance. Vite, a front-end build tool, uses esbuild to accelerate development mode. AWS CDK uses it under the hood to automatically transpile and bundle Node.js lambda functions on the fly[21]. Phoenix, a web framework for the Elixir programming language, utilizes the bundler for web asset management[22].

While much faster than Babel and other JavaScript-based build tools, esbuild plugin ecosystem is still not as large and mature. esbuild also lacks certain features from Babel, such as certain early-stage TC 39 standards which Babel provides as official plugins[23]. Unlike Babel, esbuild is mainly developed by one person and is not publicly funded, which might impose theoretical risks of it becoming abandoned. The JavaScript ecosystem is notorious for projects becoming deprecated and abandoned[24][25][26], so such a concern could be considered valid.

1.8. Browserslist

Browserslist is a shared compatibility configuration that is meant to be consumed by developer tools. It is described in a format of queries with browser versions, global usage percentages, Node.js versions, and different other query formats. In context of JavaScript tools, Browserslist configuration is consumed by Babel and Webpack. Babel uses Browserslist to determine which APIs need to be polyfilled and language features to be transpiled[27]. This ensures complete compatibility with target platforms, allows

flexible control over the code bundle and avoids loading any unnecessary polyfills, thus ensuring that only what is needed for compatibility is included in the bundle.

Browserslist uses a subset of the "Can I Use" database[28] to keep track of browser support information. Apart from its own data, "Can I Use" additionally pulls information from Mozilla's browser compatibility data (BCD) that is updated every week. BCD documents browser support for over 17,000 web platform features.

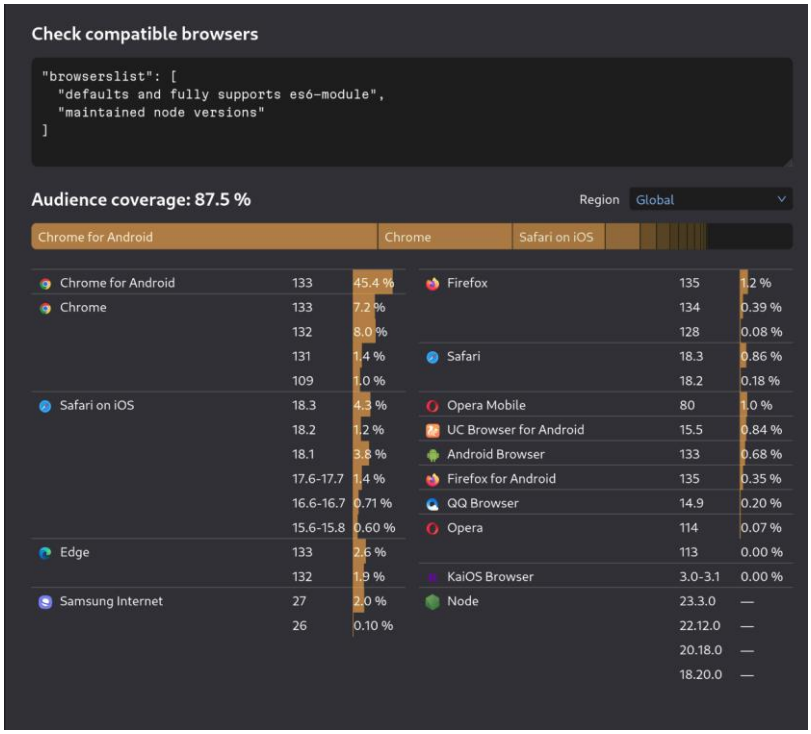

Figure 2. Browserslist configuration preview: https://browsersl.ist

While this tool is very useful for application developers, it is not applicable for library authors. As a rule, libraries do not bundle their dependencies and therefore are not able to modify the source code of the third-party packages. Because of that, library authors need to manually verify compatibility. Browserslist is also limited in server-side runtime support, as it only supports Node.js and no other runtimes such as Deno, Cloudflare Workers, and others.

1.9. eslint-plugin-compat

eslint-plugin-compat is an ESLint plugin that analyzes source code and checks for JavaScript language features and web APIs and reports errors if target browsers lack support for certain features. Under the hood it utilizes Browserslist, however it works dif-

ferently than the latter. The plugin bans usage of language features that are not supported on target platforms, instead of instructing tools to polyfill them. By doing so, none of the polyfills, except those that are explicitly loaded by a build tool, end up in the code bundle.

The code becomes more portable for projects that use build tools that do not support Browserslist, as it is not tied to specific configurations. It also provides stricter control over polyfills that are included in the final code bundle, since they must be manually omitted from the plugin's configuration.

Figure 3. eslint-plugin-compat in action

eslint-plugin-compat has one major drawback compared to Browserslist. Since the plugin does not scan imported libraries for issues, the final bundle might end up being incompatible with target platforms through those libraries that potentially rely on less supported language features. This plugin helps manage compatibility of the source code but does not take third-party dependencies into account. Like Browserslist, the plugin is also unsuitable for library authors that use external dependencies since they do not have control over the source code of dependent packages.

1.10. unenv

unenv[29] is a set of polyfills to add Node.js compatibility to other runtimes, including browsers and edge runtimes such as Cloudflare Workers. While most compatibility tools aim for browsers and Node.js, unenv covers a whole range of runtimes. With unenv it becomes easier to write cross-platform JavaScript code, which appeals to full-stack web framework authors.

Besides polyfilling Node.js utilities, it also implements support for certain browser-only APIs, such as **PerformanceEntry**, that are not yet available in Cloudflare Workers and a few other runtimes.

unenv additionally shims certain packages that are mostly polyfills for Node.js (such as node-fetch and whatwg-url) and are not needed on other runtimes where those APIs are already implemented natively.

The way unenv works is that it re-implements every single Node.js native module with runtime-agnostic code. What this means is that there are no external dependencies or Node.js-specific utilities. With unenv under the hood, no matter what the underlying runtime or platform is, the code will still run the same.

Not all of the Node.js utilities are implemented in unenv. Certain Node-only APIs, such as various cryptography functions, are explicitly not implemented, since they are bound to OpenSSL rather than being implemented in Node.js itself.

```
export const createDiffieHellman = /*@__PURE__*/ notImplemented<
  typeof nodeCrypto.createDiffieHellman
>("crypto.createDiffieHellman");

export const createDiffieHellmanGroup = /*@__PURE__*/ notImplemented<
  typeof nodeCrypto.createDiffieHellmanGroup
>("crypto.createDiffieHellmanGroup");

export const createECDH =
  /*@__PURE__*/ notImplemented<typeof nodeCrypto.createECDH>(
    "crypto.createECDH",
  );

export const createHash =
  /*@__PURE__*/ notImplemented<typeof nodeCrypto.createHash>(
    "crypto.createHash",
  );

export const createHmac =
  /*@__PURE__*/ notImplemented<typeof nodeCrypto.createHmac>(
    "crypto.createHmac",
  );
```

Figure 4 - Unimplemented cryptography functions in unenv

Nuxt.js, the most popular web framework for Vue, uses unenv under the hood[30]. Because of that, it is possible to deploy it to any server environment, from Edge Workers to a standalone server. unenv is also used by large enterprises such as Cloudflare for their Node.js compatibility layer[31].

While unenv is perfect for web frameworks that are responsible for the final bundle and thus have complete control over it, this doesn't solve the issue of compatibility for libraries. As a rule, libraries do not explicitly depend on polyfills and expect JavaScript APIs to exist in the global namespace.

1.11. Matrix strategy

One of the most common approaches to cross-version testing is called "matrix strategy". The essence of it is to run tests and various other tasks on multiple platforms in parallel, for example multiple Node.js versions. This is being done to ensure compatibility across a wider range of runtime versions, as well as operating systems. Major continuous integration services such as GitHub Actions, GitLab CI and others support matrix strategy testing or something equivalent[32].

```yaml
test:
  strategy:
    fail-fast: false
    matrix:
      os: [ubuntu-latest, windows-latest]
      node-version: [18, 19, 20, 21, 22, 23]
      # Node.js release schedule: https://nodejs.org/en/about/releases/

  name: Node.js ${{ matrix.node-version }} - ${{matrix.os}}
```

Figure 5 - Matrix strategy testing GitHub Action in the Express.js repository

As the example demonstrated in Figure 3, there is parallel testing happening on two operating systems: Windows and Ubuntu (Linux). For each of the platforms, tests are run across current officially supported Node.js versions.

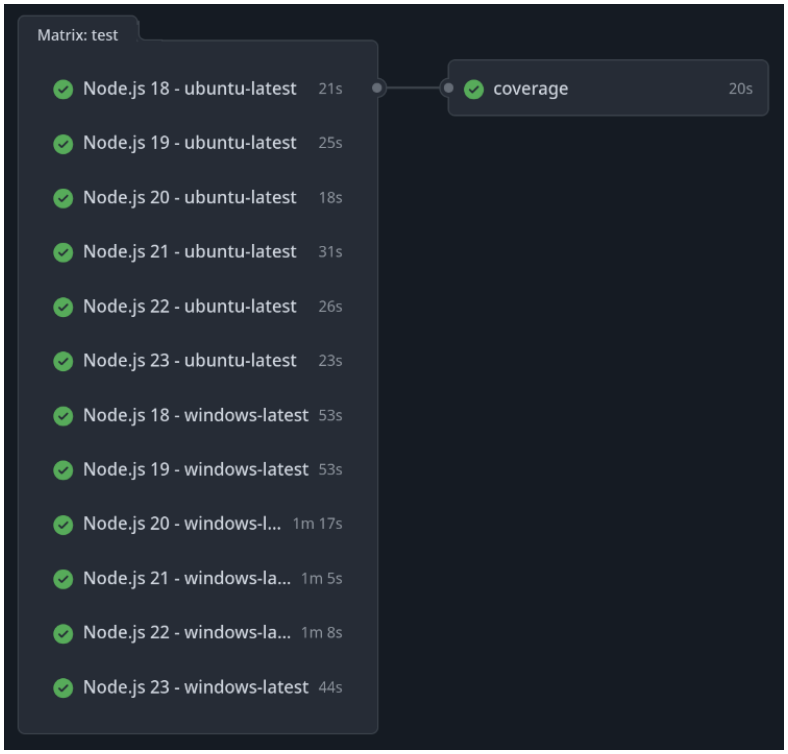

Figure 6 - Express.js GitHub Actions graph

Using matrix strategy for tests might help catch version or platform-specific errors and bugs. Critically important libraries and frameworks take advantage of matrix tests.

Such an approach has been proven to be successful, and it remains one of the most common ways to monitor compatibility issues. It addresses the problem of cross-version and cross-platform testing for libraries.

Unfortunately, this is mainly limited to one specific JavaScript runtime, for example Node.js, as matrix strategy becomes more complicated to configure with a wider range of runtimes that differ substantially. Matrix testing often omits browser testing, as well as alternative JavaScript runtimes. One of the solutions to such limitation is to implement multiple separate matrix testing strategies for each runtime and platform.

1.12. Ecosystem integration testing

Critically important JavaScript libraries and frameworks implement integration tests to ensure compatibility with downstream projects. Such a testing method makes it easier to monitor the library for any new potential breakages before they affect the library's dependents.

Vue, one of the most popular front-end UI frameworks for JavaScript, has a separate GitHub repository for ecosystem integration tests to verify that dependent projects do not suddenly break due to a bug surfaced in Vue itself[33]. The way it is set up is that there is a set of scripts that clones a project, builds and runs its own tests. In case any of projects fails, the CI/CD pipeline fails as well. Such a setup helps with finding and catching edge cases and specific bugs that otherwise wouldn't be caught if tested directly.

```
import { runInRepo } from '../utils.ts'
import { RunOptions } from '../types.ts'

export async function test(options: RunOptions) {
  await runInRepo({
    ...options,
    repo: 'vueuse/vueuse',
    branch: 'main',
    build: 'build',
    test: ['typecheck', 'test'],
  })
}
```

Figure 7 - Vue ecosystem integration test example

SWC, a JavaScript transpiler written in Rust, is another example. It has a configuration like Vue ecosystem tests. SWC integration tests also include cases for verifying if the compilation of certain libraries is done correctly, as well as the transpiler's plugins and projects that use developer tooling that is powered by SWC[34].

Solid.js, a new JavaScript front-end framework, has a single integration test that simultaneously runs checks a third-party library, Solid.js itself and a Babel plugin that transpiles the library's custom language syntax[35]. With such a setup, the whole development tool chain is tested directly end-to-end without needing to implement separate test cases for each of the packages.

Ecosystem integration tests help with catching niche and specific bugs before they make into the production release. Such tests additionally save a significant amount of time since a breakage is automatically detected. Without those, such breakages would have to be detected and reported manually by a consumer of the library, while already seeding the error into an unknown number of projects in cascade. Making projects upgrade to a patched version might take an indefinite amount of time, since they all have a different update strategy.

The main difficulty with such an approach is high setup complexity and keeping track of all the dependents. Writing tests for dozens or even hundreds of downstream libraries might be a cumbersome task, and some bugs surfaced in a lesser used third-party library might not end up getting detected anyway.

1.13. installed-check

installed-check[36] is a tool for Node.js that scans third party modules and verifies their compliance with the requirements listed in a package's manifest (package.json). The main use case of the tool is to ensure that the package's dependencies support the specified Node.js version as well as peer dependencies having correct semantic versioning ranges.

```
{
  "name": "example-lib",
  "type": "module",
  "engines": {
    "node": ">=20"
  },
  "dependencies": {
    "react-transition-state": "^2.1.3",
    "ts-pattern": "^5.6.0"
  },
  "devDependencies": {
    "@types/react": "^18.3.12",
    "@types/react-dom": "^18.3.1",
    "react": "^18.3.1",
    "react-dom": "^18.3.1",
    "vite": "^6.1.0",
    "vitest": "3.0.6"
  },
  "peerDependencies": {
    "react": "^18",
    "react-dom": "^18"
  }
}
```

Figure 8 - Sample package manifest (package.json)

A classic example of a problem with incorrect peer dependency ranges is when a library expects React of one version range, but the project specifies React in peer dependencies of another wider range that the library does not cover.

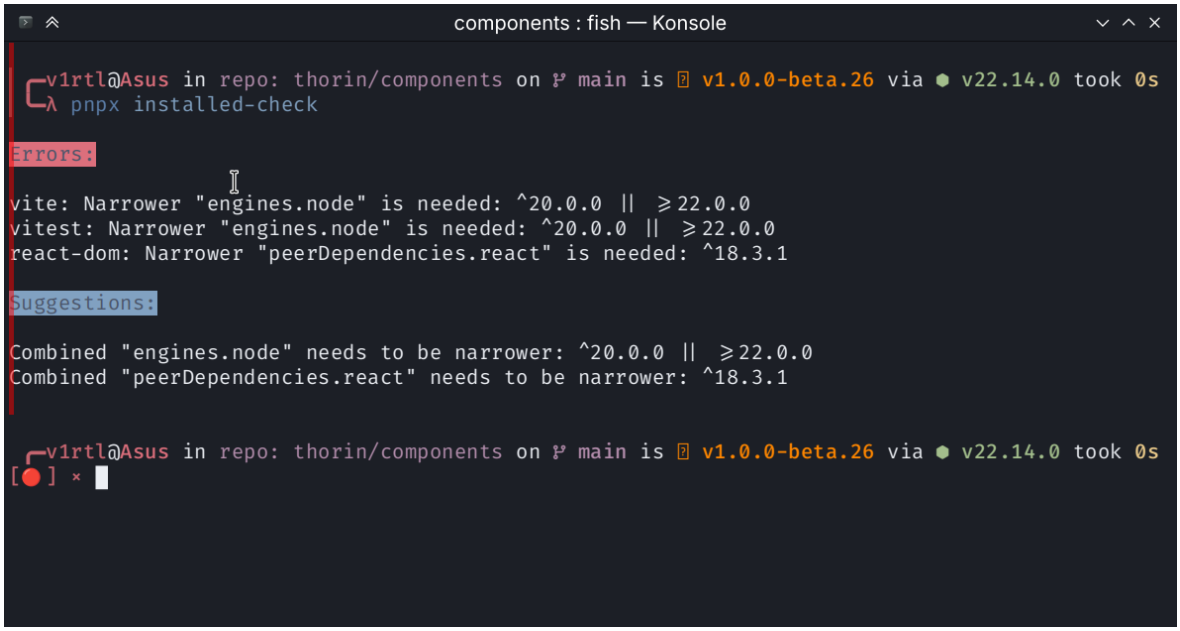

Figure 9 - installed-check sample output

As demonstrated in Figure 9, installed-check identifies incompatible Node.js version ranges, as well as incorrect peer dependency version ranges. In the package manifest, Node.js version range is defined as anything equal or higher than version 20, while dependencies "vite" and "vitest" do not follow the same range. Those dependencies omit version 21 from the range, since they do not officially support it, while the library still technically targets it. In case Node.js version 21 is used, unexpected errors and bugs specific to that version in those two packages may appear. Another potential issue might be related to the "react-dom" package. It expects the dependency "react" to be at least version 18.3.1, while the package manifest allows any version if its major is 18. The reason for narrowing the range to start from 18.3.1 is because React's developer team is removing certain deprecated APIs in React 19[37]. Version 18.3 of React, which is the last release in the 18 major version range. was published with deprecation warnings to make the transition to React 19 easier. This is done to prevent the usage of incompatible React and React DOM versions. As a result, React DOM explicitly expects React 18.3+ versions, since React DOM 18 of the latest version works with React 19 but not React with versions in the "<18.3.1" range.

installed-check is mainly intended to be used by library authors to ensure compatibility across third-party dependencies of a library by taking advantage of the package.json manifest file. With a growing count of dependencies in a project, tracking and ensuring correct version ranges becomes difficult, so the tool makes the process automatic.

1.14. WebAssembly binary distribution

A completely different direction to solving portability and compatibility issues among libraries is replacing JavaScript with something else. Multiple language-agnostic and runtime-agnostic technologies and standards for compatible libraries exist, such as FFI (Foreign Function Interface) and WebAssembly.

Node.js implements FFI as dynamically linked shared objects with native C/C++ add-ons[38]. Building native modules is mainly done via an officially supported tool called node-gyp[39]. Bun, a drop-in replacement for Node.js, provides an experimental FFI module that can load dynamic libraries implementing the C ABI (Application Binary Interface)[40]. Deno, an alternative runtime for JavaScript, provides an FFI module that can load native dynamic libraries written in compiled languages such as C, C++ and Rust.

While FFI is an industry standard for loading foreign libraries, it comes with significant limitations, such as losing type safety. That makes FFI prone to type errors that otherwise would be caught at the compilation/build stage. Dynamic libraries also exit the security sandbox which JavaScript runtimes such as Deno provide, thus lowering the security and making it possible for the FFI to make arbitrary changes to the machine on which that code is running on[41]. FFI does also does not work in browsers and is limited to server-side runtimes that support FFI. That limitation is a critical obstacle to implementing libraries for JavaScript that work everywhere, including both server and browser runtimes.

WebAssembly is another technology which provides an architecture-agnostic and platform-agnostic binary format that is also able to run in browsers. According to the official website, “WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications.”[42]

Unlike FFI, WebAssembly runs in a proper security sandboxing environment and inherits browsers’ security properties when executed client-side, thus lowering the attack surface by not exposing the whole system to WASM binaries. More than 40 programming languages such as Rust, C/C++ and Go support WebAssembly as a compilation target. Multiple runtimes for WebAssembly exist, such as Wasmer[43].

WebAssembly has been used by different JavaScript libraries to embed binaries in a secure and portable way without creating compatibility issues. WASM binaries are superior to Node.js C++ addons in this sense, since there is no need for compiling native code on a current machine, due WebAssembly binary format being universal and not being bound to a specific CPU architecture. No additional tools rather than the language compiler that already supports WebAssembly is required, while with C++ addons node-gyp must be used, which itself also depends on Node.js and Python.

"The primary benefit of using Wasm is that code can be re-used. This means that some code that performs a given task that is needed in N different systems can be compiled to the Wasm target, regardless of the (supported) language it is written in, and then re-used in an arbitrary number of other systems (in runtimes that support loading Wasm, either natively, or through a library)." (Morys-Magiera et al., 2024, p. 472)[44]

Various JavaScript libraries, such as Web API polyfills[45] and SQLite bindings[46], use compiled Wasm binaries to avoid pulling additional build dependencies and to minimize friction that comes from installing C++ add-ons. WebAssembly runs in all major browsers and various server runtimes, including Node.js, Deno, Bun and Cloudflare Workers.

WebAssembly essentially solved the problem of cross-runtime (including different runtime versions) and cross-platform compatibility for libraries but introduced a few new issues. While JavaScript code is tree-shakable by JavaScript bundlers and build tools, WebAssembly is a whole binary blob that is not possible to split so that only the parts that are used in the code remain in the bundle. In case an error is thrown in a WebAssembly binary, the stack trace leads to compiled function signatures rather than the source, which makes it more difficult to debug in situations where an error is thrown in the binary. Projects such as wasm2map[47] introduce source maps to WASM binaries; At the same time, wasm2map aims to be used by developers of libraries that compile WASM binaries themselves, rather than library consumers.

While WebAssembly does not aim to be a band-aid for all the compatibility issues in the JavaScript space, it took its niche of environment-agnostic compatibility and is widely used today. It is especially useful for bringing libraries written in C/C++, Rust and other languages to be the web and edge computing environments.

1.15. Conclusion

The complex issue of JavaScript library compatibility contains a diverse range of different problems. Developers approached it from different angles, including build tooling, linting, compatibility layers and different testing strategies. These solutions solve different problems in context of compatibility: runtime, platform compatibility and third-party dependents.

There is no silver bullet that would solve all of those at the same time, and because of that multiple different tools and practices exist. Chosen methods are also influenced by the nature of the library, for instance one that provides bindings to a library from another language would most likely be using WebAssembly and have a JS wrapper, while a web framework that is intended to be deployed to any platform would take advantage of something like unenv. In certain scenarios, different combinations of such practices and tools help achieve maximum control over compatibility checking, for example ecosystem integration testing and matrix testing together.

Ultimately, JavaScript library compatibility is quite a multi-layered topic and each of the solutions that exist today have their own trade-offs which should be considered.

2. PROJECT IMPLEMENTATION

2.1. Technology stack

The command-line application is written in TypeScript and runs with Node.js. Oxc parser is used to generate abstract syntax trees (AST). The reason why this parser was chosen for the project and no other AST parsers like Acorn is because Oxc is extremely fast, due to being written in Rust, and it can parse both CommonJS and ESM modules correctly, which usually requires using something like cjs-module-lexer. Node.js native modules for filesystem and path utilities are used to inspect the dependencies. tinyglobby is used to query files with specific file extensions within package directories. The tool depends on the MDN compatibility database to obtain language feature support information. The database is not consumed directly but is instead queried through the "ast-metadata-inferer" npm package. The library converts the data into a more convenient structure which is easier to query for specific APIs and syntax expressions. Spektr[48] CLI framework is used for command-line argument parsing. "semver"[49] npm package is used to properly compare semantic versions of Node.js versions.

2.2. General overview

The thesis project suggests an alternative approach to solving the compatibility problem in the form of a command line tool. While tools such as **eslint-plugin-depend** mainly scan source code exclusively, the prototype would take third-party dependencies into account and consider them equally important to the source code during the assessment of compatibility requirements. Such a method can reduce maintenance workload for library authors. Instead of manually verifying runtime compatibility with each of the external packages, running a tool in console would automatically perform all the necessary checks and configure tooling as well as a package manifest to correspond to the actual supported Node.js version.

2.3. Architecture

The primary focus of the tool is dependencies. As a rule, a library or any other JavaScript project imports a set of libraries which themselves also might import other libraries. Based on that it is possible to build a dependency tree as demonstrated in Figure 10.

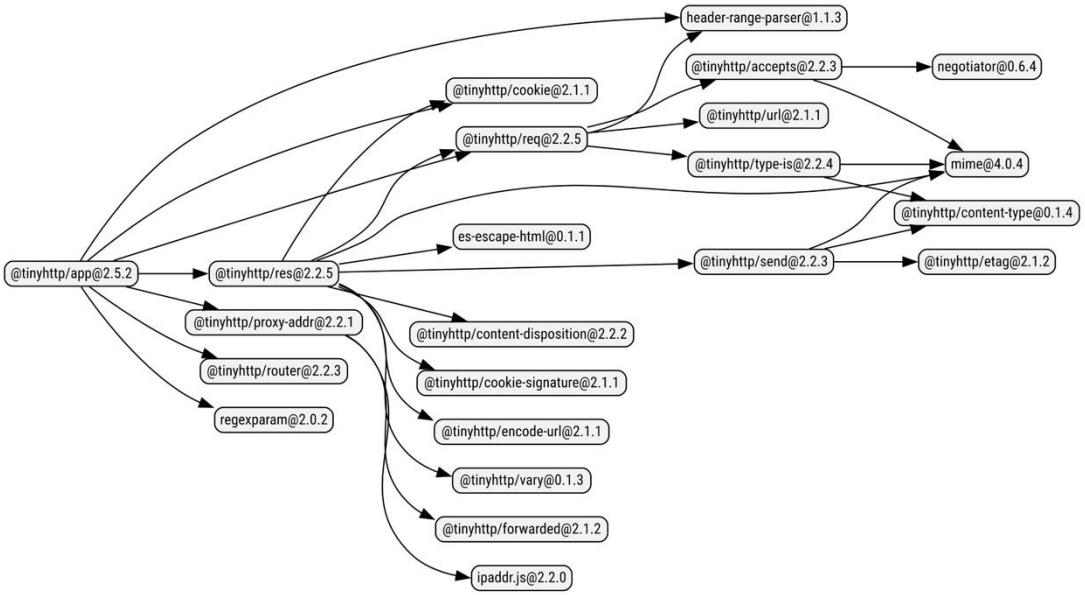

Figure 10. Example JavaScript library dependency tree

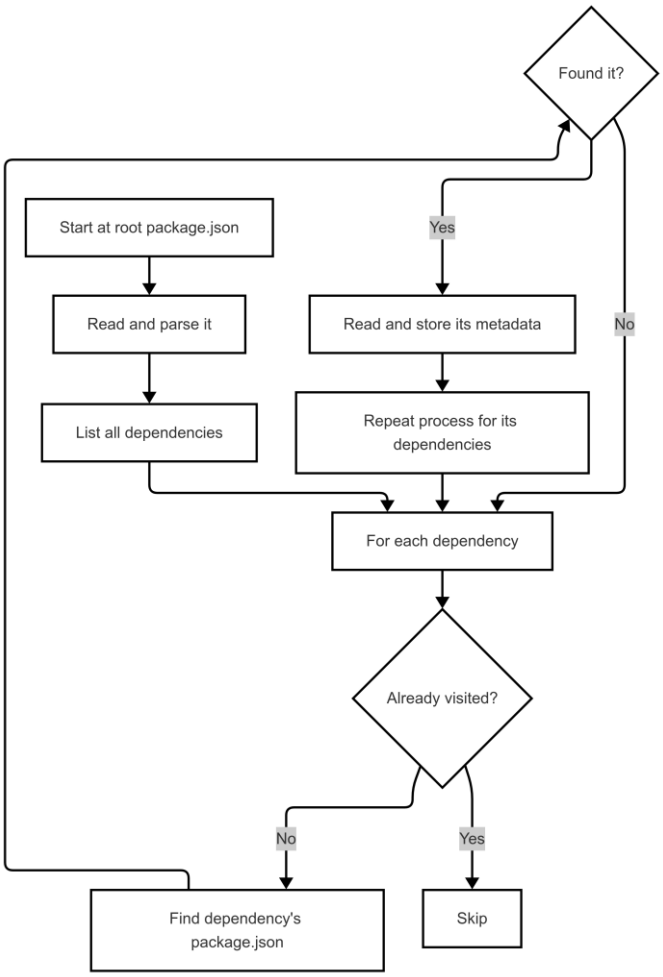

Figure 11. node_modules crawling algorithm

First, the tool would scan the project's source code and traverse the "node_modules" directory, while looking for JavaScript files and scanning their contents in the meantime. The way this process works for "node_modules" is described in Figure 11. The algorithm starts with reading a project's root package.json file which contains a list of dependencies, as well as includes a list of source files. Next, for each of the dependencies, their package.json file is read and parsed. It contains the same data – dependency list and a file list. This process repeats recursively until a total graph of dependencies and their metadata objects are built.

After the file scan is complete, each of the code files is mapped and parsed into an abstract syntax tree. Next, the AST is traversed to identify language globals, such as "Intl", "String" and others, as well as specific syntax expressions, such as optional chaining and nullish coalescing, which are available only starting specific ECMAScript versions. To prevent excessive computational work, all the AST tokens are automatically deduplicated with JavaScript Maps.

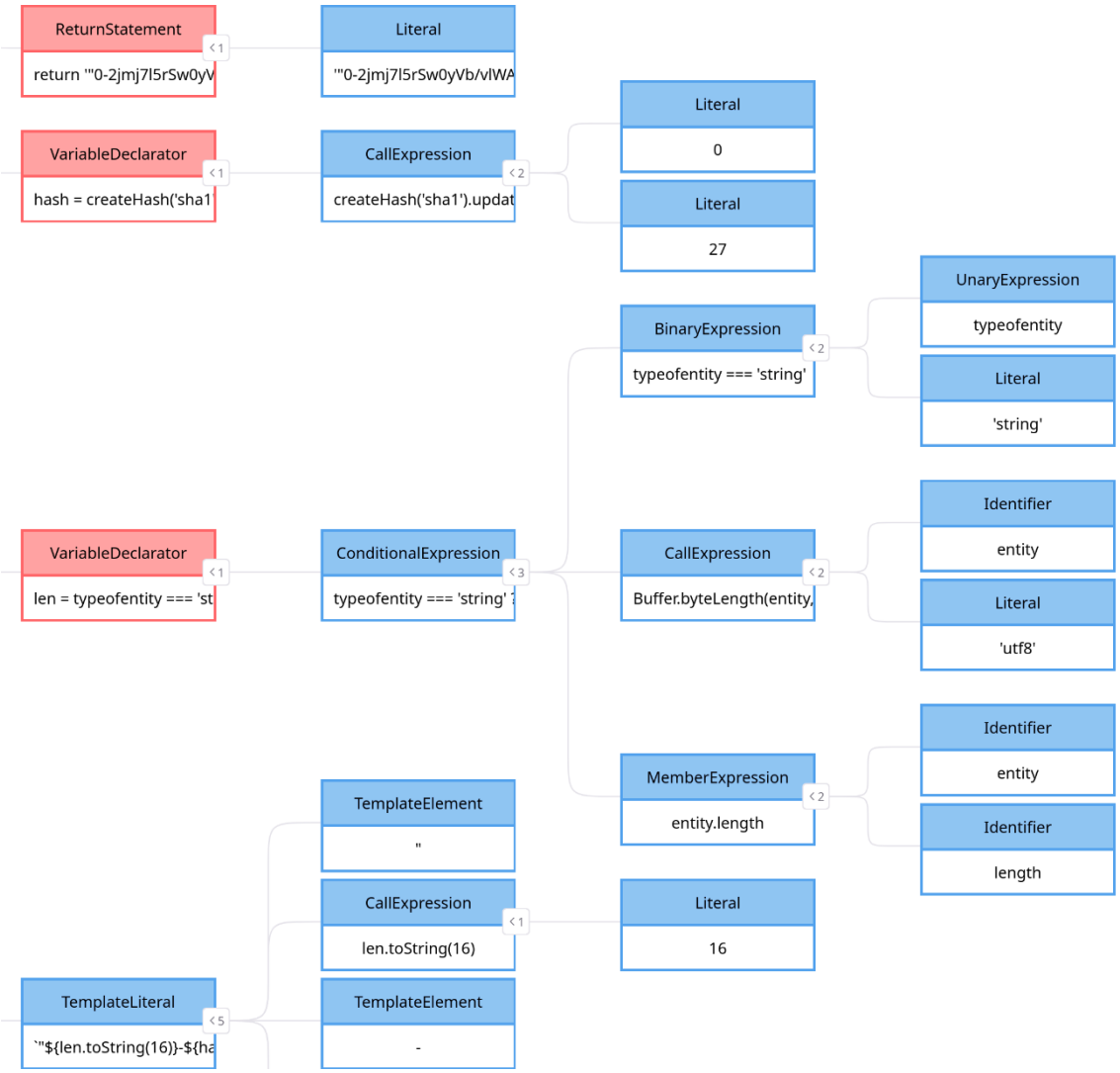

Figure 12. Example AST visualization of a JavaScript library source code

After the tokens are collected into a consistent data structure, they are looked up in the MDN compatibility database, from which the minimum supported Node.js version is extracted. After all the tokens are mapped to their respective Node.js versions, a minimum supported version of Node.js is calculated. The version is determined based on what is the highest identified version. For example, if one library requires at least version 18 and other libraries require at least version 10, the minimum supported version would be 18.

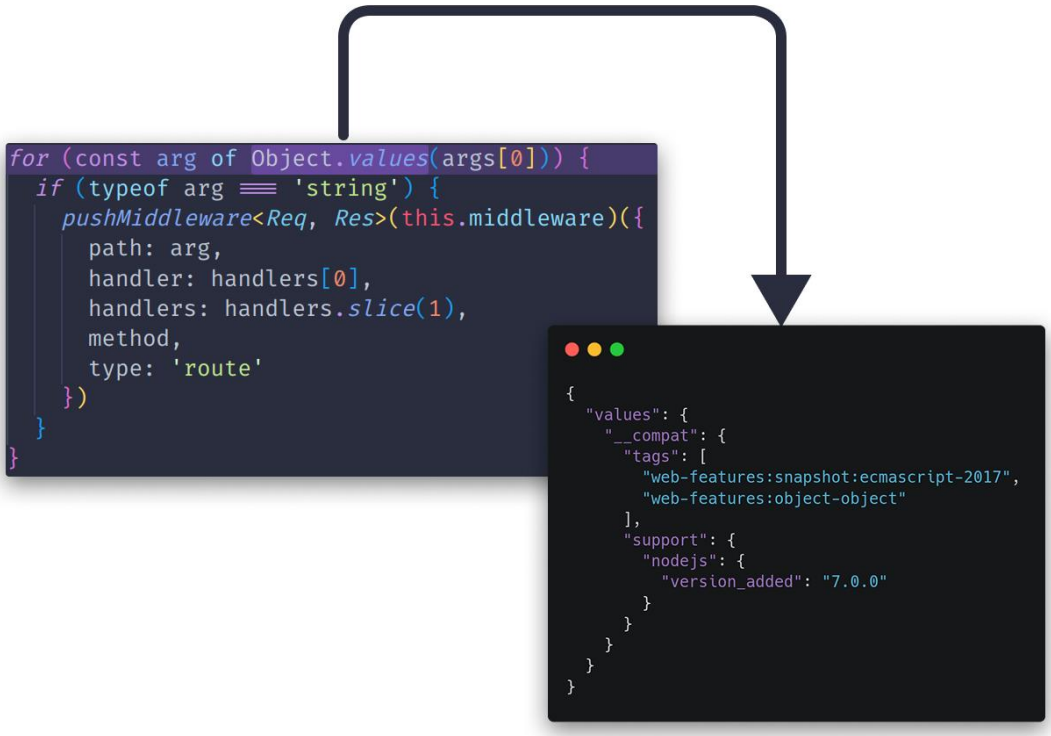

Figure 13. Language feature lookup in the MDN database

After the target Node.js version has been determined, it's now possible to properly update the package manifest (package.json) to include the relevant engine support data. This can be achieved by populating package.json with an "engines" field that would contain target Node.js version.

The prototype does not cover the full set of JavaScript language features and Node.js API support since it is built for demonstrational purposes. The tool can also identify Node-specific features such as "node:" protocol imports, as well as certain modern JavaScript language features such as import attributes.

2.4. Implementation challenges

During development, a few issues started popping up. While it was relatively untrivial to implement source code parsing and feature detection, detecting types became a large obstacle. Since language parsers such as Oxc cannot infer variable and expression types, expressions such as "arr.map" are not being detected. Type inference is a global

issue when it comes to implementing tooling that parses source code. Biome, an extremely fast set of developer tools for JavaScript written in Rust, is putting monumental effort into implementing type detection, and thus improving linting experience, as well as enabling new and more precise rules[50]. At the time of writing, Biome's type inference algorithm is not meant to be used by external libraries and is contextual to linting. Implementing a type inference algorithm on your own implies building a subset of TypeScript. TypeScript itself is an extremely complicated compiler with one of the most advanced type systems, so just using the TS compiler would hinder performance and add unnecessary verbosity. For the sake of simplicity of creating the demonstrational version, type inference was not implemented now until there are more accessible ways to do it rather than implementing it from scratch.

2.5. Practical testing

To assess the effectiveness of the tool, it was run against a web framework called tinyhttp and the CLI itself. tinyhttp was chosen as it has a modern codebase that uses the latest language features while targeting modern Node.js versions. The project receives around 130000 weekly downloads, which makes it a used npm package.

Since tinyhttp is a monorepository, the tool was run from one of the package directories at first. It scanned both the dependencies and the source, while identifying both JavaScript and Node-specific language features.

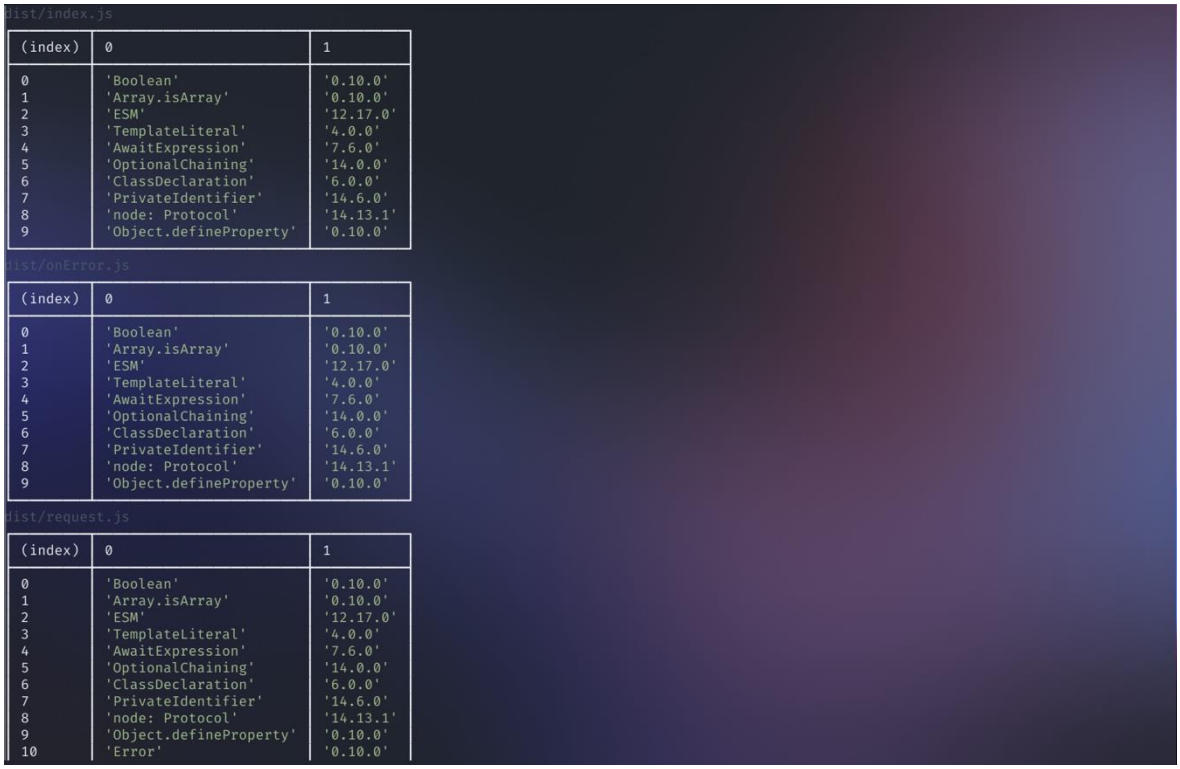

Figure 14. Scan results of source files of tinyhttp

By the end of the execution of the tool, it suggested to update the Node.js version specified in the package.json manifest file, as the one that is already specified does not correspond to the actual supported version. The package supports versions older than the previously specified one.

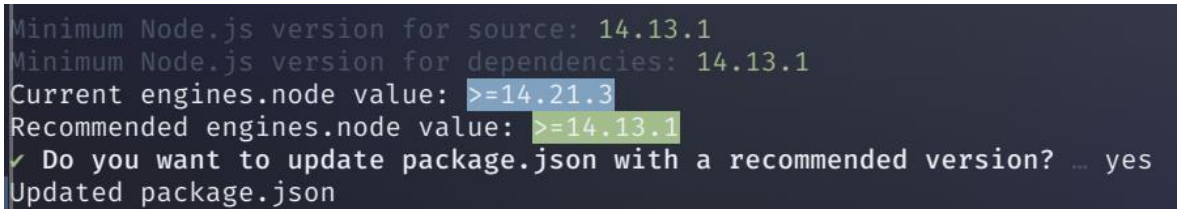

Figure 15. Automatic detection and configuration of a recommended Node.js version

The same process was repeated with other packages in the monorepo.

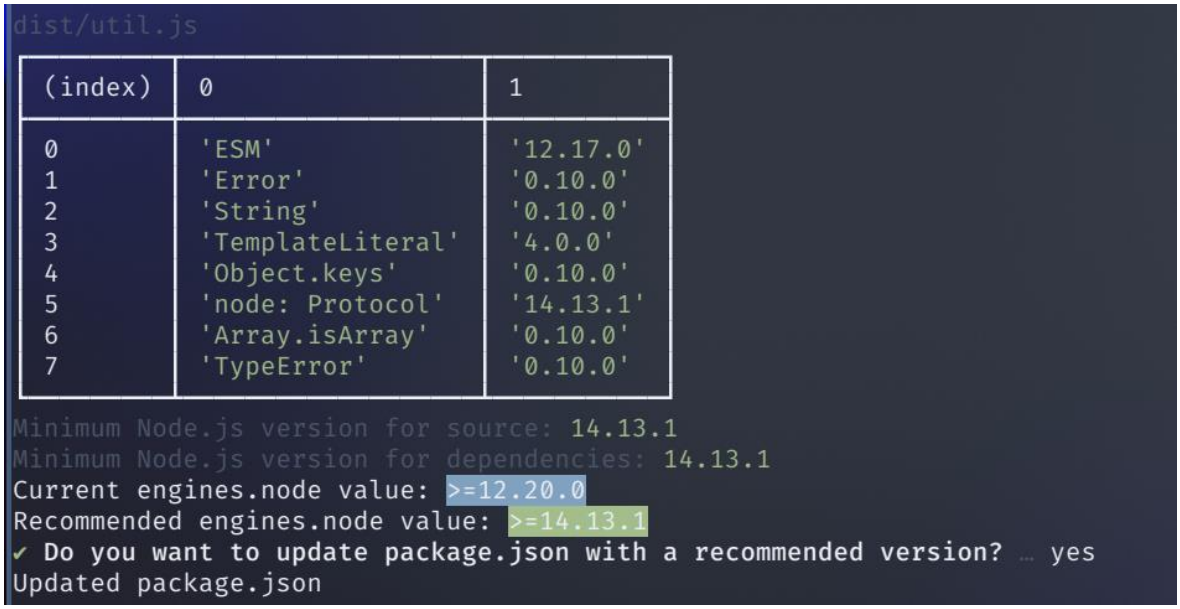

Figure 16. Re-running the tool against other packages

In certain cases, it helped identify incorrect "engines.node" values, where the claimed supported version wouldn't be able to load the module at all.

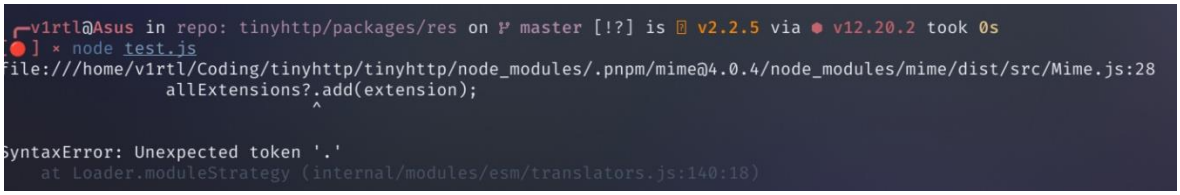

Figure 17. Example of a runtime error caused by an outdated Node.js version

By the end of running the CLI recursively on all monorepository packages, it additionally highlighted packages that did not have ”files” field set in package.json, since it throws an error if the field is missing. The field is recommended to be set to avoid publishing any redundant files not related to the functionality of the package. As a result, the application fixed incorrect "engines.node" values both in situations where the value was too set to high, while nothing prevented those packages from supporting older versions, and where the previously specified version would not be able to load such packages and instead error during runtime.

Next, the CLI was run against its own source code. Since the project is rather fresh, it takes advantage of latest language features.

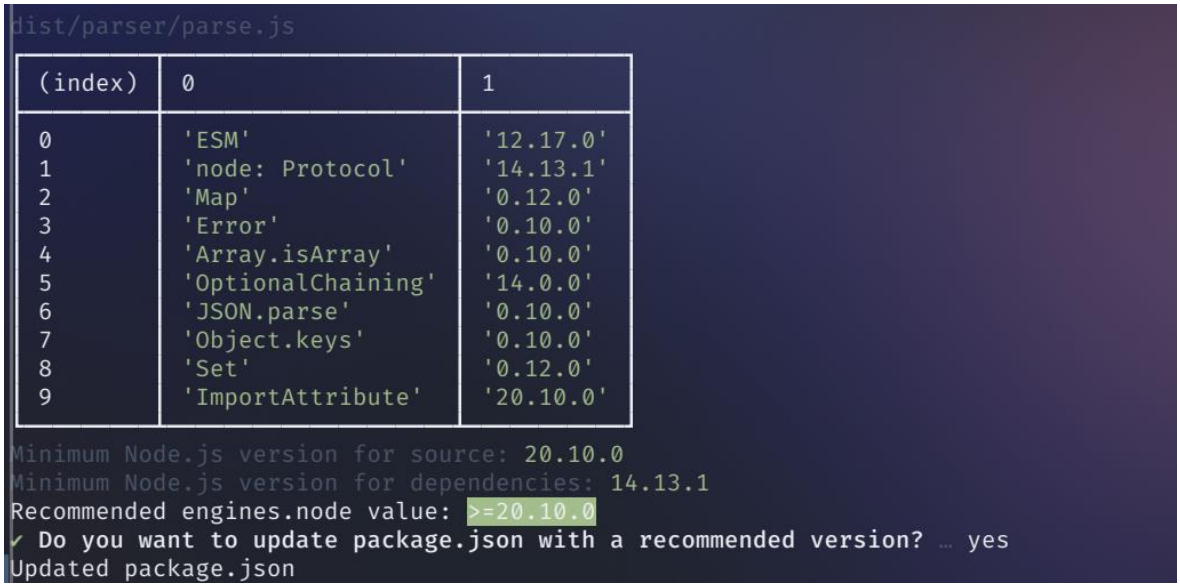

Figure 18. Running the tool against its own source code

It managed to detect even recent JavaScript features, such as import attributes. The recommended version is confirmed to be able to run the tool.

In conclusion, the prototype is capable of what it intended to do, to scan the source code including the dependencies, analyse language features, determine supported Node.js versions, and automatically update package.json with relevant engine requirements.

2.6. GitHub link

The project's source code is available on GitHub: https://github.com/talentlessguy/autocompat. The package is published on npm under the "autocompat" name. It is possible to run the CLI from npm by executing "npx autocompat".

Conclusion

Software compatibility remains to be one of the most recurring issues in IT. It is especially prevalent in the JavaScript world, where everything is connected to each other, libraries depend on other libraries, and projects rely on them. Libraries must ensure compatibility with as many environments as possible, as well as with numerous dependents and dependencies on their own. Such an amount of technical debt is at times overwhelming for developers, and different tools aid this situation by making the lives of library authors and maintainers a little bit easier. At the time of writing, developers can take advantage of cross-platform, cross-environment and integration testing, including various other approaches and their combinations described in this thesis.

As a result of research and practical development efforts, a new concept of a tool came to fruition. It stands out from other similar tools by focusing on third-party dependencies first rather than just the source code. While most tools help maintain compatibility for web applications and other user-facing products, this command line application instead is built to be used by library authors and maintainers.

The tool was tested against a real-world JavaScript web framework with over 7 million downloads. During the assessment phase, the command-line application managed to properly identify runtime support information that was previously defined incorrectly.

All the set goals outlined in the “Goals” section have been successfully achieved through extensive research, practical testing and the command line application’s development process, as well as the tool’s testing results on a target project.

Resume

Tarkvara ühilduvus on endiselt üks kõige sagedamini esinevaid probleeme IT-sektoris. Eriti levinud on see JavaScripti maailmas, kus kõik on omavahel seotud, teegid sõltuvad teistest teekidest ja projektid sõltuvad neist. Teegid peavad tagama ühilduvuse võimalikult paljude keskkondadega, samuti oma arvukate sõltuvustega. Selline hulk tehnilist võlga on arendajatele kohati üle jõu käiv ja erinevad tööriistad aitavad sellele olukorrale kaasa, tehes raamatukogude autorite ja hooldajate elu veidi lihtsamaks. Käesoleva töö kirjutamise ajal saavad arendajad kasutada ära platvormide-, keskkonna- ja integratsioonitestimist, sealhulgas mitmesuguseid muid lähenemisviise ja nende kombinatsioone, mida käesolevas töös kirjeldatakse.

Uurimis- ja praktiliste arendustööde tulemusel sai teoks uue töövahendi kontseptsioon. See eristub teistest sarnastest tööriistadest selle poolest, et keskendub esmalt kolmandate osapoolte sõltuvustele, mitte ainult lähtekoodile. Kui enamik tööriistu aitab säilitada veebirakenduste ja muude kasutajale suunatud toodete ühilduvust, siis see käsurea rakendus on selle asemel loodud raamatukogude autorite ja hooldajate jaoks.

Tööriista testiti reaalse JavaScripti veebiraamistikuga, mida on alla laaditud üle 7 miljoni korra. Hindamisetapis suutis käsurea rakendus õigesti tuvastada varem valesti määratletud tööajalist tugiteavet.

Kõik töö eesmärgid on edukalt saavutatud läbi ulatusliku uurimistöö, praktilise testimise ning käsurea rakenduse arendamise ja selle testimise sihtprojektil.

References

[1] Gobbi, M. F., & Kinder, J. (2024). GENIE: Guarding the npm Ecosystem with Semantic Malware Detection. 2024 IEEE Secure Development Conference (SecDev), Secure Development Conference (SecDev), 2024 IEEE, SECDEV, 117–128.

[2] Zerouali, A., Mens, T., Robles, G., & Gonzalez-Barahona, J. M. (2019). On the Diversity of Software Package Popularity Metrics: An Empirical Study of npm. 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Software Analysis, Evolution and Reengineering (SANER), 2019 IEEE 26th International Conference On, 589–593.

[3] Liu, Y., Tiwari, D., Bogdan, C., & Baudry, B. (2024). Detecting and removing bloated dependencies in CommonJS packages.

[4] Paltoglou, K., Zafeiris, V. E., Diamantidis, N. A., & Giakoumakis, E. A. (2021). Automated refactoring of legacy JavaScript code to ES6 modules. Journal of Systems and Software, 181.

[5] GitHub: The State of open source. https://octoverse.github.com/2022/top-programming-languages

[6] Usage statistics of JavaScript as client-side programming language on websites: https://w3techs.com/technologies/details/cp-javascript

[7] 10 Things I Regret About Node.js: https://www.youtube.com/watch?v=M3BM9TB-8yA

[8] Kerner, S. M. (2018). Node.js Event-Stream Hack Exposes Supply Chain Security Risks. EWeek, N.PAG.

[9] Keep all your packages up to date with Dependabot: https://github.blog/news-insights/product-news/keep-all-your-packages-up-to-date-with-dependabot

[10] Problem Installing SDK 4 – Couchbase forums: https://www.couchbase.com/forums/t/problem-installing-sdk-4/33270

[11] Failing to install (node-pre-gyp) with node version 8.10.0: https://github.com/realm/realm-js/issues/6989

[12] Deceptive Deprecation: The Truth About npm Deprecated Packages: https://www.aquasec.com/blog/deceptive-deprecation-the-truth-about-npm-deprecated-packages/

[13] Knip on npm: https://www.npmjs.com/package/knip

[14] What is Babel? – Babel documentation: https://babeljs.io/docs/

[15] packages depending on @babel/runtime – npm: https://www.npmjs.com/browse/depended/@babel/runtime

[16] neotraverse on Github: https://github.com/PuruVJ/neotraverse

[17] Hyperfine on GitHub: https://github.com/sharkdp/hyperfine

[18] mitata on GitHub: https://github.com/evanwashere/mitata

[19] Inject – esbuild API documentation: https://esbuild.github.io/api/#inject

[20] esbuild-plugin-browserslist - GitHub: https://github.com/nihalgonsalves/esbuild-plugin-browserslist

[21]Amazon Lambda Node.js Library - AWS CDK Reference Documentation: https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_lambda_nodejs-readme.html#nodejs-function

[22] Asset management – Phoenix: https://hexdocs.pm/phoenix/asset_management.html

[23] @babel/plugin-proposal-pipeline-operator – Babel docs: https://babeljs.io/docs/babel-plugin-proposal-pipeline-operator

[24] react-toolbox on GitHub: https://github.com/react-toolbox/react-toolbox

[25] mojito on GitHub: https://github.com/YahooArchive/mojito

[26] express-ws on GitHub: https://github.com/HenningM/express-ws

[27] Browserslist integration – Babel: https://babeljs.io/docs/babel-preset-env#browserslist-integration

[28] About – Can I Use: https://caniuse.com/ciu/about

[29] unenv on GitHub: https://github.com/unjs/unenv

[30] Nuxt on the Edge: https://nuxt.com/blog/nuxt-on-the-edge

[31] Node.js Compatibility - Cloudflare Workers documentation: https://developers.cloudflare.com/workers/runtime-apis/nodejs/#nodejs-api-polyfills

[32] Running variations of jobs in a workflow – GitHub Docs: https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/running-variations-of-jobs-in-a-workflow

[33] Vue Ecosystem CI – GitHub: https://github.com/vuejs/ecosystem-ci

[34] SWC Ecosystem CI – GitHub: https://github.com/swc-project/swc/tree/main/.github/swc-ecosystem-ci

[35] Solid.js integration tests – GitHub: https://github.com/solidjs/solid/tree/main/packages/test-integration

[36] installed-check on GiHub: https://github.com/voxpelli/node-installed-check

[37] React 19 Upgrade Guide - https://react.dev/blog/2024/04/25/react-19-upgrade-guide

[38] OpenJS Foundation, “Node.js C++ addons.” 2021: https://nodejs.org/api/addons.html

[39] node-gyp – GitHub: https://github.com/nodejs/node-gyp

[40] FFI – Bun documentation: https://bun.sh/docs/api/ffi

[41] Papaevripides, M., & Athanasopoulos, E. (2021). Exploiting Mixed Binaries. ACM Transactions on Privacy and Security (TOPS), 24(2), 1–29.

[42] WebAssembly - https://webassembly.org

[43] Wasmer - https://wasmer.io

[44] Morys-Magiera, A., Długosz, M., Skruch, P., & Szelest, M. (2024). Portable native code for JavaScript Runtimes with WebAssembly, Rust and Wasm-Pack. 2024 International Conference on Emerging eLearning Technologies and Applications (ICETA), Emerging eLearning Technologies and Applications (ICETA), 2024 International Conference on,20241024, 471–474. https://doi.org/10.1109/iceta63795.2024.10850794

[45] Intl Segmenter Polyfill – GitHub: https://github.com/surferseo/intl-segmenter-polyfill

[46] sql.js – GitHub: https://github.com/sql-js/sql.js

[47] wasm2map – GitHub: https://github.com/mtolmacs/wasm2map

[48] Spektr – GitHub: https://github.com/StauroDEV/spektr

[49] semver – GitHub: https://github.com/npm/node-semver

[50] Biome Type Inference: A Look Behind The Scenes - https://arendjr.nl/blog/2025/05/biome-type-architecture