Inclusion Criteria for Third-party Dependencies in Enterprise Software Projects

University of Oulu
Information Processing Science
Master's Thesis
Benjamin Mustonen
2023

Abstract

Third-party libraries are commonly used in software development to save development time, allowing teams to focus on implementing their own business logic. Including third-party dependencies in a project is not without its risks, however. Bugs, vulnerabilities, and license incompatibilities are only some of the potential issues that can arise from third-party dependencies, yet knowing what to look for before including a dependency can be difficult.

This thesis investigates the factors that should be considered when including a third-party dependency through a review of current scientific literature and models a testable set of inclusion criteria through the design science process. The factors found in the literature were validated and assigned importance levels through a developer survey. Based on the survey results, the model was finalised and tested on six different libraries. The model as well as the test results were then evaluated by developers in a small-scale workshop.

The design science process resulted in a proof-of-concept model that was considered quite good by the developers evaluating it, in addition to a synthesis of existing knowledge on third-party dependencies. The model includes 14 factors divided into eight different criteria, with each factor having a clear definition, a way to measure it, as well as the number of points it contributes to the scoring system of the model. The final score of the model can then be used as a reference to aid in the dependency inclusion decision making process. The developers considered the criteria to be usable enough to be implemented as part of their dependency inclusion process with some minor changes. The major limitation with these findings is that the developer data, used in both creating the importance ratings as well as evaluating the model, was acquired through convenience sampling. This means that the findings cannot be generalised to a wider population. Additionally, the survey and the workshop both had low participation rates of 40% and 55% respectively, hurting the credibility of the results. Future research should consider repeating the study with sampling that can be generalised to a larger population to validate and improve upon the results in this thesis.

Keywords
JavaScript, open-source, third-party libraries, dependencies

Supervisor
PhD, Professor Burak Turhan

Foreword

I would like to thank my thesis supervisor Burak Turhan for his excellent guidance and feedback on the thesis. I must also express my deepest gratitude to FCG Finnish Consulting Group for allowing me to spend portions of my work weeks writing the thesis. Special thanks to the Technology Lead Petri Tuomaala for his assistance with the thesis topic and his help with organising the developer survey and workshop, as well as the Team Lead of the TalentRekry team, Mikko Niska, for his patience and understanding with my delayed graduation.

Many thanks to the developers at FCG who took part in the survey and workshop portions of the thesis. Thank you also to those students who gave feedback during the peer review portions of the master's thesis seminar course. Their feedback helped refine the thesis to a higher standard.

Contents

1. Introduction

Third-party libraries are commonly used in software development as a way to save development time, as well as to reduce the amount of code that needs to be maintained by the development team. The JavaScript open-source third-party library ecosystem provides developers with over one million packages containing solutions to common development problems, allowing a development team to focus on their own domain-specific challenges instead of solving the same issues that the wider developer community has faced time and time again. The use of these third-party dependencies is not without its own set of risks, however. Whether it be through incompetence, malice, or straight-up poor luck, even a single library can have dramatic negative consequences for many software projects, as demonstrated by the *left-pad* incident of 2016 (Collins, 2016), which broke thousands of projects worldwide.

The topic of this thesis originates from the concerns of enterprise software development teams at FCG Finnish Consulting Group, where the author works as a software developer. The teams at FCG use third-party dependencies in their web application projects, where the security and quality of dependencies are common considerations. Motivated by the desire to make smarter, business-safe inclusion decisions in real-life software development, this thesis takes a closer look at the various factors that go into the decision-making process of including third-party libraries, framed in the context of an enterprise software project using open-source JavaScript libraries. These factors are then modelled into testable criteria to aid in the decision-making process. The reason for focusing on JavaScript is that it is the client-side programming language of choice across multiple projects at FCG, making it possible to gather data from as many developers as possible within the company. While JavaScript is used in both client- and server-side programming, and thus many of the concepts mentioned here will apply to both, the primary focus of this thesis is on client-side JavaScript dependencies since JavaScript is only used for client-side development at FCG.

Chapter 2 is a look at previous literature on third-party dependencies and libraries, as well as the necessary background information on the subject. Chapter 3 is an explanation of the research method used in the thesis. Chapter 4 summarises the findings of the literature review and a developer survey into individual factors. Chapter 5 covers the finalised design of the model based on literature and the developer survey. Chapter 6 is an evaluation of the model based on a developer workshop. Chapter 7 discusses the results of the research. Chapter 8 presents the conclusions of the thesis.

2. Background

This chapter contains a review of previous literature on third-party dependencies and libraries. The chapter is split into several subchapters, starting with general dependency-related terminology and previously developed software evaluation criteria, followed by more specific properties of dependency inclusion based on the key areas that were discovered when studying literature relating to third-party dependencies. Certain topics found in the literature are expanded upon using official documentation to give them some necessary context. The search query that was used to find the papers was purposefully general, looking for any articles relating to third-party JavaScript dependencies and libraries. This was to ensure that as many dependency-related factors could be discovered without a bias towards the factors that initially motivated the study. The main areas that were identified in the literature were dependencies, security, namespace conflicts, licensing, and tests.

2.1 Terminology

I will be using the terminology definitions by Kikas et al. (2017) as a base for the definitions in this thesis to remain consistent with their usage in previous literature. Packages are reusable code or components that can be included in software projects as dependencies. In general terms, dependencies can be anything external that a project is reliant on, such as people, hardware, and services from other companies, but in the context of third-party libraries and the JavaScript ecosystem, dependencies are packages that a project depends on. Kikas et al. (2017) separate projects into two different categories: packages, which are projects that are published in repositories for everyone to use, and applications, which are projects that are not published as packages, and, therefore, cannot be used as dependencies by other projects. An ecosystem is a set of packages and applications and their dependency relations (Kikas et al., 2017).

Packages are included in projects using dependency management tools, also known as package managers. One of the most popular package managers in the JavaScript ecosystem is npm, short for Node Package Manager. Despite its name, npm is in wide use outside of Node.js projects, and its package hosting repository, the npm Registry, currently contains over one million packages (npm, Inc., n.d.). Other popular package managers include Yarn and pnpm, both of which also use the npm Registry for packages by default (Yarn, n.d., pnpm, n.d.).

There are two types of dependencies: direct dependencies and transitive dependencies. Direct dependencies are those that are directly referenced by a program, while transitive dependencies are those introduced by other dependencies (Kaplan & Qian, 2021) or, in other words, the dependencies of dependencies. Transitive dependencies are also sometimes called indirect dependencies.

2.2 Software evaluation criteria

Jackson et al. (2011) presented a set of assessment criteria to evaluate the quality of software. The criteria were based on the ISO/IEC 9126-1 standard, which is now withdrawn and revised by the ISO/IEC 25010 (ISO, n.d.). The set is divided into two main criteria: “Usability” and “Sustainability and Maintainability.” Under the “Usability” criterion are the sub-criteria Understandability, Documentation, Buildability,

Installability and Learnability. Under the “Sustainability and Maintainability” criterion are the sub-criteria Identity, Copyright, Licencing, Governance, Community, Accessibility, Testability, Portability, Supportability, Analysability, Changeability, Evolvability and Interoperability. It should be noted that the Jackson et al. (2011) definition of Accessibility is not related to ease of use for those with disabilities, but rather to the ability to download the software.

The Jackson at al. (2011) criteria consist of lists of Yes/No questions for each sub-criterion. The criteria are mentioned to not have equal weighting, but the importance of each criterion is not disclosed, and it is thus left to the interpretation of the reader to determine which criteria are the most important. The Jackson et al. (2011) criteria also do not take security into account in their evaluation of software quality.

2.3 Dependency inclusion in JavaScript packages

Dependencies can be included in two ways: they can be cloned and served by the developer, or they can be served by a third-party host (Nakhaei et al., 2020). In the first case, the developer uses a package manager, such as npm, to include the JavaScript resource in their own project, while in the second case, the developer includes the resource using a <*script*> HTML tag that includes the URL of the resource in the tag’s src-attribute.

Both inclusion approaches have their own strengths. Self-hosting gives the developer more control over the resource, such as when to update it, and a guarantee that the resource will always be available to the users of the developer’s project, as it is served from the developer’s server. A third-party host could go down for maintenance or for other reasons, preventing the use of the resource. However, Nakhaei et al. (2020) point out some strengths of using third-party hosting: using a shared host for popular libraries will reduce page loading times due to CDNs and caching, and not having to manually update libraries reduces the cost of maintaining resources. Nakhaei et al. (2020) also note that certain types of resources, such as Google Tag Manager, are not feasible to be self-hosted, as they are based on, and rely on, constant updates. In a study by Nikiforakis et al. (2012), 88.45% of the top 10 000 websites at the time used at least one third-party hosted JavaScript resource.

JavaScript projects using npm have a package.json file that contains information about the project itself, such as the name, version, and description of the project. The file also contains information for three types of dependencies: runtime, development, and optional dependencies. Runtime dependencies are dependencies that are always installed, development dependencies are only installed during development, and optional dependencies are attempted to be installed but won’t cause an error if the installation is not successful. Dependencies are generally specified using a package name and the desired version(s), but URLs, local paths, and the asterisk (*) wildcard – meaning any version – are also accepted in place of the version number. (Jafari et al., 2021).

The versioning in npm uses Semantic Versioning, also known as SemVer. The SemVer versioning scheme is formatted as MAJOR.MINOR.PATCH (SemVer, n.d.). Additional version information, such as pre-release tags or other build metadata, may be appended to the patch number with a hyphen. For example, take version number 1.2.3-alpha.1. In this version number, 1 would be the major release, 2 would be the minor release, 3 would be the patch release, and alpha.1 would be the pre-release version. The SemVer 2.0.0 specification (SemVer, n.d.) states that the major version should be incremented when

incompatible API changes are made, the minor version should be incremented when backwards-compatible functionality is added, and the patch version should be incremented when backwards-compatible bug fixes are made.

npm allows for the use of operators in comparators to determine version ranges. Having no operator is the same as using the equality operator (=). The operators for less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=) behave as expected. For example, >=1.2.3 would match 1.2.3, 1.2.4 and 2.1.1, but not 1.2.2 or 1.1.1. Other operators include tilde (~), which allows patch-level changes, and caret (^), which allows changes that do not modify the left-most non-zero release on the version number. For example, ^1.2.3 allows minor and patch changes, ^0.2.3 only allows patch changes, and ^0.0.3 allows no changes. Version ranges can have multiple comparators joined by whitespace, which creates a comparator set, where the version range is satisfied when it matches all the comparators within the set. Comparator sets can be joined together with double pipes (||). Such a joined version range will be satisfied by a match from either comparator set without needing to satisfy both. (npm, n.d.).

npm also generates a lockfile called *package-lock.json*. This file contains a representation of the package's current dependencies and is generated whenever an operation is run where *package.json* or the *node_modules* tree is modified. The lockfile is committed to source control, and its purpose is to make it so that every install of the package is the same, down to the exact versions of dependencies, without having to commit the large *node_modules* folder into source control. Running the npm command *npm update* will update the dependencies for the package based on the rules set in *package.json*, generating a new *package-lock.json* file in the process. (npm Docs, n.d.).

It is worth noting that *package-lock.json* is not included when publishing a package into the npm Registry and will be ignored if it is found in any other place than the root project. This means that for packages that are meant to be dependencies for other projects, such as libraries, *package-lock.json* only applies to its developers, not its users. However, npm does provide an alternative to *package-lock.json* for cases where such functionality is needed: *npm-shrinkwrap.json*. The file is identical to *package-lock.json*, except it can be included in a published package and will not be ignored even if it is not in the root project. npm does not recommend using *npm-shrinkwrap.json* outside of a few very specific cases, however. (npm Docs, n.d.).

Jafari et al. (2021) identified several dependency-related smells, which they define as recurring violations of dependency management guidelines with negative consequences on projects and the JavaScript ecosystem. *Pinned dependency* is a dependency that is restricted to only one version, or in other words, "pinned" to that version. Pinned dependencies are considered a smell, as they make it so that the package will not be updated to include security and bug fixes. *URL dependency* is a dependency that points to a URL as opposed to a version number. This is considered a smell as, depending on the URL, it will either cause every version to be fetched, including breaking changes, or it will act like a pinned dependency, never updating itself. *Restrictive constraint* is a smell where the dependency constraint is too restrictive, such as not allowing for backward compatible updates. Since backwards-compatible updates are not supposed to break functionality and may include important updates, they should be included. However, this assumes that the package maintainer is following SemVer, and restrictive constraints may be required for packages that are not SemVer compliant. *Permissive constraint* is a smell where dependency constraints are not restrictive enough, such as allowing breaking changes by using the asterisk wildcard, or by specifying a minimum version range without a maximum limit. *No package-lock* is a smell where the developers have opted not to

include the package-lock file in their repository. This is a smell, as without the file, there is no guarantee that installations at different times will yield the same result. *Unused dependency* is a smell where a dependency is included in the list of runtime dependencies without being used in code, taking up unnecessary space and creating potentially further unneeded dependency chains. This smell can also occur when development dependencies are mistakenly included in runtime dependencies. Finally, *missing dependency* is a smell where a dependency is used in code but not specified in the runtime dependencies. Missing dependencies can cause bugs and unexpected behaviour. (Jafari et al., 2021).

The npm ecosystem is heavily interlinked: according to a study by Zimmermann et al. (2019), installing an average npm package introduces an implicit trust in 79 third-party packages. This is a notable increase from a previous study by Kikas et al. (2017), where the average number of dependencies in JavaScript projects was slightly over 34 in 2015 and more than 54 in 2016. Additionally, the most popular packages influence, either directly or indirectly, more than 100 000 other packages. This also means that individual maintainers can have an influence on hundreds of thousands of packages, as was the case in 2016, when a utility package called *left-pad* was removed, causing many packages depending on *left-pad*, or packages that depended on packages depending on *left-pad*, to become unavailable (Zimmermann et al., 2019).

2.4 Security

The cybersecurity community references known vulnerabilities through a system known as the Common Enumeration of Vulnerabilities Program, or the CVE Program. The CVE Program identifies, defines, and catalogues publicly disclosed vulnerabilities in a catalogue called the CVE List. The lifecycle of a CVE Record is as follows: A person or an organisation discovers a vulnerability. The vulnerability is then reported to a CVE Program participant, such as a CVE Numbering Authority, also known as a CNA. The participant requests a CVE ID for the vulnerability. The ID is reserved, allowing the stakeholders to use it for coordination and management before publicly disclosing it. The details are then submitted, such as the affected products, affected product versions, the vulnerability type, the root cause, the impact of the vulnerability, and at least one public reference of the vulnerability. Once the CVE Record contains the minimum required elements, it can be published to the CVE List by the CNA responsible for it. (CVE, n.d.a).

It should be noted that the CVE Program alone is not a comprehensive vulnerability database. Vulnerability databases, such as the U.S. National Vulnerability Database (NVD, https://nvd.nist.gov/) and the Snyk Vulnerability Database (https://security.snyk.io/), build upon the information in CVE Records to provide more detailed and enhanced information on each vulnerability, and sometimes even provide information on vulnerabilities before they have a public CVE Record (CVE, n.d.a, Snyk, n.d.). The Common Vulnerability Scoring System, or CVSS, is used to score the severity of CVEs. The scoring system provides a numerical score that can be translated into a qualitative description, such as “low”, “medium”, “high”, or “critical”. The severity scores for CVE Records are typically provided by the NVD but can also be provided by the CNA that published the record. (CVE, n.d.b).

Cross-Site Scripting, or XSS, is a code-injection vulnerability where the attacker can execute code at the origin of the vulnerable web page. JavaScript’s Same-Origin Policy restricts scripts’ data access to web pages that have the same origin, which means that an attacker exploiting an XSS vulnerability can execute their malicious code with elevated permissions. Client-Side XSS attacks are caused by insecurely processing user-provided

inputs, such as by using *eval()* or *document.write*, which takes strings as parameters that are then executed as code. (Musch et al., 2019).

Musch et al. (2019) investigated Client-Side XSS vulnerabilities in Alexa Top 5000 websites. They found 351 sites vulnerable to Client-Side XSS, with 92 of them being vulnerable purely due to third-party scripts. This means that in the Alexa Top 5000, more than 25% of websites affected by Client-Side XSS are only vulnerable due to flaws in third-party libraries.

Lauinger et al. (2018) discovered that vulnerable JavaScript libraries are in wide use on the Web despite patches being available. One potential cause for this phenomenon is that patches tend to only be supplied for the most recent major versions of libraries, which are usually not backwards compatible with earlier versions. This means that developers must continually update their code to keep up with breaking API changes if they wish for their websites to remain vulnerability-free. Lauinger et al. (2018) found that the median lag between library versions used and their most recent release in Alexa Top 75k websites is 1177 days, concluding that developers rarely update their library dependencies after deploying their sites. However, this is not just a problem with the users of libraries. Zimmermann et al. (2019) found that up to 40% of all packages in the npm ecosystem depend on code with at least one publicly known vulnerability, suggesting that keeping packages up to date may still leave the software vulnerable to known vulnerabilities due to transitive dependencies. One reason for this may be npm's use of the package-lock.json lockfile: the lockfile must be regenerated for vulnerability patches to be applied, leading to technical lag for projects that do not often update (Zimmermann et al., 2019). Decan et al. (2018) came to a similar conclusion in their study of 610 097 npm packages: unmaintained packages and improper dependency constraints are the main causes of dependent packages remaining vulnerable despite patches being available.

The study by Decan et al. (2018) sheds some more light on the discovery and fixing of vulnerabilities in the npm ecosystem. The authors found that a majority of vulnerabilities are discovered more than two years after the release of the version affected by the vulnerability. However, 30.9% of all vulnerabilities are fixed before the reported discovery date, possibly due to maintainers disclosing the vulnerability only after fixing it. 65.7% of all vulnerabilities are fixed after their reported discovery, of which 82% are fixed before the publication of the vulnerability. 12% of all vulnerabilities are fixed after publication, and 3.5% of vulnerabilities are never fixed. Of the vulnerabilities fixed after discovery, the majority are fixed within a month, though there is still a 25% chance that a vulnerability is not fixed after six months and a 17% chance that it is not fixed after 12. When it comes to dependent packages, the authors found that 12% of packages were fixed before the upstream package was fixed, either by removing the dependency or by rolling back to an unaffected version. 44% were fixed at the same time as the upstream package, and 10% after the upstream fix. The remaining 33% of dependent packages were not fixed. (Decan et al., 2018).

Software Composition Analysis (SCA) tools can be used to scan open-source components of software for vulnerabilities (Imtiaz et al., 2021). Imtiaz et al. (2021) compared nine different SCA tools that support analysing npm (JavaScript) and Maven (Java) dependencies and found that the key strength of an SCA tool is its vulnerability database. They discovered that certain databases map vulnerabilities differently: one database reported two vulnerabilities as affecting versions "1.8.0, 1.9.2" and "1.9.2, 1.9.4", while another reported "up to 1.9.1" and "up to 1.9.3". This resulted in the first database reporting 1.9.3 as a vulnerability-free version, while the other one flagged it as vulnerable. Another difference maker found by Imtiaz et al. (2021) is the state of CVEs in a database:

as CVEs can become reserved, disputed, or rejected after their initial publishing, it is important that the databases are kept up to date. In the authors' findings, two databases still reported vulnerabilities for four rejected CVEs.

Vulnerability detection tools such as *npm audit* and *snyk test* operate based on reported vulnerabilities, meaning that they do not actually analyse the code for vulnerabilities. When it comes to analysing code for vulnerabilities, there are two primary forms of analysis: static analysis and dynamic analysis. Static analysis extracts information about a program by analysing its source code, while dynamic analysis extracts information by executing the program with instrumentation, such as logging or profiling (Ntousakis et al., 2021). In addition to the vulnerability scanning tool *snyk test*, Snyk also offers a static code analysis tool, *snyk code*.

Rafnsson et al. (2020) suggest using linters for static code analysis. Linters, such *ESLint* for JavaScript, are tools that scan code for issues such as bugs and programming or stylistic errors, as well as code smells, based on a defined set of rules (Rafnsson et al., 2020). Rafnsson et al. (2020) implemented ESLint rules that find and automatically fix cross-site scripting, security misconfiguration and SQL injection vulnerabilities. However, one would have to manually inspect the code of a third-party library in an IDE with the linter active, making this a less convenient method of static analysis compared to traditional tools. Additionally, ESLint is limited to scanning one file at a time, meaning that it cannot find vulnerabilities caused by the interaction of different modules (Rafnsson et al., 2020).

Third-party dependencies can have security issues caused by factors outside of code quality. Zimmermann et al. (2019) have identified several threat models for the npm ecosystem. The first of these threats is *malicious packages*, where malicious individuals may upload packages that contain malicious code and attempt to trick users into using or depending on these packages. The function of the package itself could be legitimate but paired with a malicious payload executed through a post-installation script. Another threat model is *exploiting unmaintained legacy code*: sometimes developers may abandon their packages and thus their public vulnerabilities may never be fixed. Someone may create a new, fixed version of the package, but the original version will remain in the npm repository as is, and thus every other package depending on the vulnerable package will remain insecure for as long as they use the original package. *Package takeover* is a threat where a malicious individual convinces a current maintainer of a package to add them as a maintainer or manages to get code into the codebase in some other way, such as through a pull request or compromised development tools. *Typosquatting* is an attack strategy similar to package takeover and malicious packages, where the attacker creates a malicious package with a similar name to a legitimate, popular one, hoping to catch users that mistype the legitimate package's name. Another attack similar to typosquatting is *combosquatting*. Some package names consist of multiple words, so a combosquatting attack takes advantage of this by having the malicious package contain the same words as a legitimate package, but in a different order (Kaplan & Qian, 2021). *Account takeover* is a threat where an attacker compromises the account of a maintainer, getting access to deploy malicious code. This is often caused by the maintainer having weak or reused passwords or falling for a social engineering scam. Finally, the *collusion attack* is a threat where multiple maintainers conspire to cause intentional harm, or have their accounts simultaneously taken over by attackers, and attack the npm ecosystem with many of the threats above at once. (Zimmermann et al., 2019).

Zimmermann et al. (2019) mention micropackages, also known as trivial packages, as a security risk in the npm ecosystem. Micropackages are packages that consist of only a

few lines of source code. Despite being small, they offer the same attack vectors as larger packages, and due to their prevalence in the ecosystem, may lead into many transitive dependencies. The larger the number of dependencies in a project, the larger the attack surface.

Choosing to include JavaScript libraries through third-party hosts introduces additional security concerns. While using third-party hosts tends to help keep libraries up to date, they are susceptible to third-party infection threats (Nakhaei et al., 2020). Should a host get infected, every website using a resource from the infected host risks serving their users malicious scripts. This attack vector bypasses the Same-Origin Policy, as third-party hosted scripts are treated as scripts belonging to the website itself (Nakhaei et al., 2020).

2.5 Namespace conflicts

As JavaScript does not have namespaces, everything is part of a global object, often called the global namespace. This means that conflicts can occur from two names colliding, causing one to overwrite the other. Creating global variables, thus increasing the probability of conflicts, is called "global namespace pollution" (Theisen, 2019). Patra et al. (2018) define four classifications of conflicts: inclusion conflicts, which cause exceptions when including multiple libraries; type conflicts, which cause issues when multiple libraries write type-incompatible values to the same globally reachable location; value conflicts, which cause issues when multiple libraries are type-compatible but write differing values to the same globally reachable location; and behaviour conflicts, which cause issues when multiple libraries store functions to the same globally reachable location, but do not provide matching behaviour.

However, there is a way to bring namespace-like functionality to JavaScript libraries. Patra et al. (2018) and Theisen (2019) describe a "single API object" pattern, where the entirety of the library's API is encapsulated into a single object that is part of the global object, minimizing the chance of conflicts. Unfortunately, Patra et al. (2018) found that in a sample of 951 popular JavaScript libraries, 71% did not follow this pattern at the time. This could partially be due to the popularity and widespread browser support for the ECMAScript 5 specification when the study was made. ECMAScript 6, released in 2015, introduced JavaScript modules, an easier way to implement similar behaviour as a standard language feature. Internet Explorer would only partially support ECMAScript 6 before support for the browser officially ended in 2022, and this partial support did not include the modules feature. With libraries in 2018 needing to support IE, many could not afford to use the new language features (Patra et al., 2018).

Theisen (2019) and Paltoglou et al. (2018) recommend using JavaScript modules to reduce global namespace pollution. Like in the single API object pattern, one would encapsulate the contents of a library into a single global variable, which would then be imported as a module wherever it is needed. With Internet Explorer 11 having been sunset on June 15, 2022 (Microsoft, 2022), all current major browsers support JavaScript modules (caniuse, n.d.).

2.6 Licensing

Open-source software has three primary groups of licenses: permissive, copyleft, and weak-copyleft. Permissive licenses do not place many restrictions on the distribution of the software, while copyleft licenses restrict software using the license to be free to use,

run, study, and redistribute. Weak-copyleft is in the middle of these two, allowing software using the license to be proprietary if desired as long as the source code is available. (Moraes et al., 2021).

Gangadharan et al. (2012) define several Free and open-source software (FOSS) licensing clauses. The first of these clauses is composition, determining whether the software can be integrated with other software. Attribution is a clause that determines whether the use of the software requires attribution. The sharealike clause determines whether software using, altering, or transforming the original software must be distributed on the same terms. Copyleft licenses are always sharealike licenses, though some sharealike licenses are only weak-copyleft. The non-commercial use clause determines whether the software can be used commercially.

According to Gangadharan et al. (2008), licensing conflicts generally happen due to three possible scenarios. The first scenario is "conflicts by unacceptable license clauses," where the license contains clauses that the software author does not find agreeable as a matter of opinion, and therefore cannot utilise the library in their project. The second scenario is "conflicts by incompatible license clauses," where certain clauses directly prohibit the distribution of the software. The third scenario is "conflicts by change of licenses over releases," which is born from a future release being licensed under a different license than an earlier version. Licensing conflicts can lead into having to acquire special permission from the license holders, or even removing the offending code and replacing it with a compliant solution (Gangadharan et al., 2012).

Moraes et al. (2021) found that 62% of the open-source JavaScript projects they examined were under more than one license. Furthermore, over a third of these multi-licensed projects contained license incompatibilities with the project's own license, such as the project being licensed under MIT, but using files licensed under a more restrictive license. In a 2011 study by Sojer and Henkel, 24% of survey respondents from a sample size of 732 had no training or knowledge of internet code licenses. Yet in the same survey, only 5% considered themselves not very familiar or below in their familiarity with license obligations, suggesting that developers overestimate their knowledge of licenses. Sojer and Henkel further note that 14% to 21% of developers in their sample have at some point not checked or purposefully ignored license obligations when ad-hoc reusing internet code. However, their results also show that developers with open-source software experience are significantly more knowledgeable about licenses than other types of developers.

Transitive dependencies can cause further license violations. Even if a direct dependency is compatible with a project's license, that dependency may have other dependencies with incompatible licenses (Qiu et al., 2021). However, in a study by Qiu et al. (2021), it was discovered that only 0.644% of packages in their sample size of 419 708 packages in the npm Registry have dependency-related violations. The authors suspect this could be due to over 75% of the studied packages having a permissive license, around 5% having either a copyleft or weak copyleft license, and the rest being either unknown, unlicensed, or in the public domain. Another potential explanation is that, as Sojer and Henkel (2011) point out, open-source software developers are more knowledgeable about licensing than other types of developers and could thus be more careful about not violating license clauses.

2.7 Testing

Automated tests are a way to make sure that a library functions as intended, and that new changes do not break old functionality. Many open-source libraries, such as Shopify Draggable, have tests included in their repositories (Shopify, 2018). Rafi et al. (2012) synthesised the benefits and limitations of automated testing in their systematic literature review. The benefits they discovered include improved product quality, high code coverage, reduced manual testing time, increased reliability of testing, an increase in the confidence of the system, a reduction in human effort, a reduction in cost, and an increase in the system's fault detection. Automation is, however, limited in that it cannot replace manual testing, since not all tasks can be automated. In some cases, automation can also fail to achieve expected goals and can be difficult to maintain. Automated testing also requires an appropriate strategy and skills to utilise its full benefits.

The quality of testing can wary. Code coverage, the percentage of a program that is ran when running tests, is a metric that can be used to assess the quality of testing (Sun et al., 2021). Like in dynamic code analysis, code coverage tools instrument code with line counters to track the lines executed by tests (Istanbul.js, n.d.). Istanbul.js, a JavaScript code coverage tool, measures the following types of coverage: lines, meaning the percentage of lines of code – excluding comments, braces and such – executed in code; statements, meaning the percentage of statements executed in code; functions, meaning the percentage of functions executed in code; and branches, meaning the percentage of branches – such as the *true* and *false* branches of an *if*-statement, or each case of a *switch*-statement – executed in code (Istanbul.js, 2015).

Hemmati (2015) investigated the effectiveness of code coverage criteria. He examined faults in open-source system test cases and applied four control flow coverage criteria as well as one data flow coverage criterion to see whether improving the tests would improve code coverage. The results indicate that even with 100% code coverage, 7% to 35% of faults can still be undetected, with the most common fault being specification related. This is because if a condition is not implemented in code, even 100% coverage of code could not detect the fault.

According to Sun et al. (2021), many packages in the npm registry do not include testing code to decrease the file size of their releases. This means that one must check the package's development repository to find whether the package has tests or not. In a study of 373 JavaScript projects by Fard and Mesbah (2017), 40% of client-side projects had no tests. Additionally, for the client-side projects that did have tests, the tests were of moderate to low quality in terms of code coverage.

Common testing frameworks for client-side JavaScript projects include QUnit, Mocha and Jasmine (Fard & Mesbah, 2017). In Fard and Mesbah's (2017) study, they found that tests written in Mocha tended to have higher coverage than those written in QUnit, with Jasmine being somewhere in the middle. It was also found that tests written without any testing framework generally had the poorest coverage.

3. Research method

This chapter contains the research method used in the thesis, as well as the research questions and methods of data collection and analysis.

According to Stol and Fitzgerald (2020), software engineering research is divided into two modes: knowledge-seeking and solution-seeking research. Knowledge-seeking research contributes to the software engineering knowledge base through making observations, while solution-seeking research results in designs and solutions for software engineering challenges. As the focus of my research is to seek a solution to an already observed challenge in software engineering, solution-seeking research seems like the most fitting approach.

Stol and Fitzgerald (2020) link Design Science to solution-seeking research. Runeson, Engström and Storey (2020) provide further arguments for its application in Software Engineering Research: The Design Science Paradigm can help in assessing the contributions of research, and it can be used to build and synthesise knowledge in the field of software engineering while finding solutions to problems at the same time. The paradigm can also help communicate research across the research community as well as the industry. For these reasons, I have chosen to use Design Science as the research method for this thesis.

Peffers et al. (2006) describe the Design Science Research process as consisting of six activities:

1. Problem identification and motivation
2. Objectives of a solution
3. Design and development
4. Demonstration
5. Evaluation
6. Communication

In this thesis, the problem identification and motivation are covered by the introduction as well as the research questions chapters. The objectives of the solution are covered in research methods. The design and development of the solution can be found in chapters 4 and 5. The demonstration of the solution can be found in chapter 6.2. Chapter 6 itself contains the evaluation of the solution. The communication activity is the act of writing this thesis.

Runeson et al. (2020) recommend the use of a “visual abstract template” that they have designed to help researchers assess research contribution, build knowledge, and communicate research to practitioners. The template covers the main constructs of design science research: the theoretical contribution in terms of a technological rule; the instantiation of the contribution in terms of a problem-solution pair; the empirical or theoretical support for problem conceptualization and solution design; and it addresses the relevance, rigour, and novelty of the research (Runeson et al., 2020). Figure 1 shows the visual abstract template as defined in Engström et al. (2020).

The technological rule, the top box of the visual abstract template, is expressed as follows:

> *"To achieve **effect/change** in **situation/context**, apply **solution/intervention**." (Engström, Storey, Runeson, Höst & Baldassarre, 2020, p. 2633)*

The technological rule is a generalised statement to help a researcher identify and communicate the value of the research and is the main takeaway of design science within the research. The problem-solution pair, the middle section of the template, shows the empirical contribution of the study. The section is laid out as two boxes, with one box containing the problem instance and the other the solution. Supporting the problem-solution pairing are three descriptions of knowledge-creating activities: problem conceptualisation, solution design, and validation. The bottom section of the template contains three assessment boxes: relevance, rigor, and novelty. Relevance shows who the technological rule is relevant to, rigor addresses the maturity of the technological rule, and novelty considers whether there are other rules that should be considered when designing a similar solution in another context. It is important to assess the value of the research, as the goal of design science research is to produce general design science knowledge, as opposed to solving unique, single instance problems. (Engström et al., 2020).

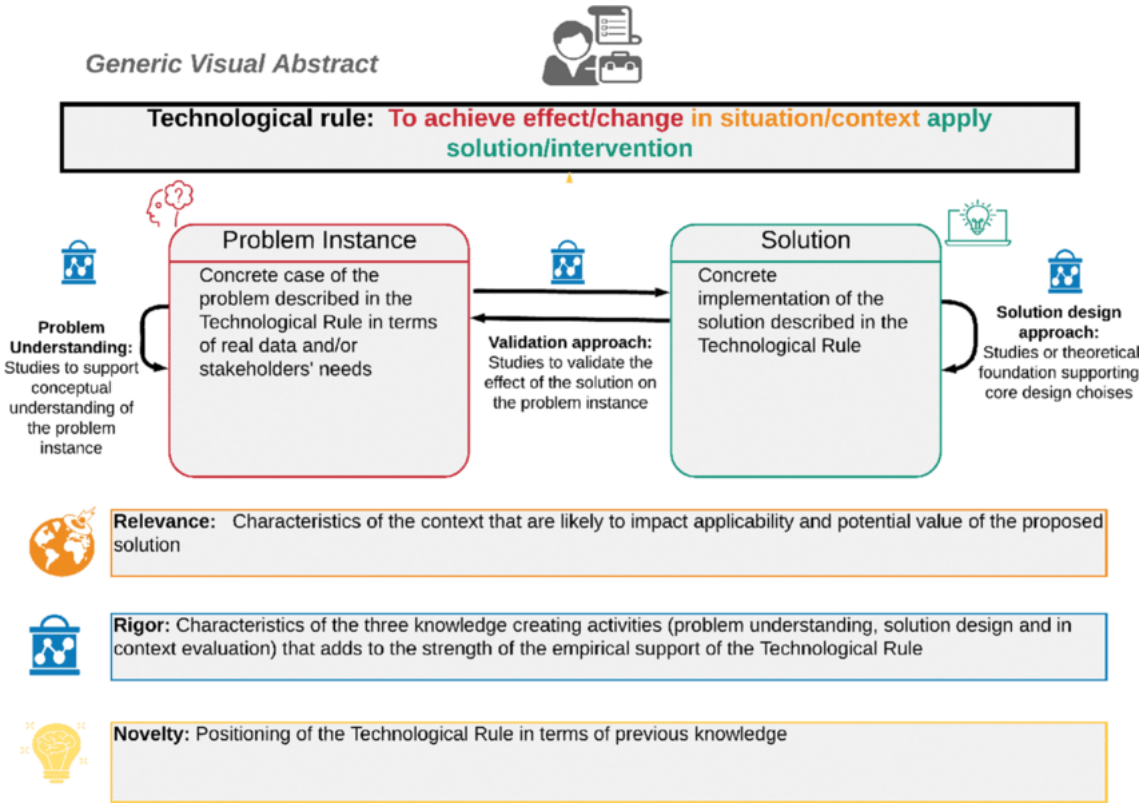


Figure 1. The visual abstract template, showing the technological rule, problem-solution pair and the three assessment criteria. From "How software engineering research aligns with design science: a review," by Engström, E., Storey, M., Runeson, P., Höst, M. & Baldassarre, M. T., *Empirical Software Engineering, 25* (p. 2633), 2020. Licensed under CC BY 4.0.

Due to the benefits of the visual abstract template, I will be using it to frame and communicate the Design Science Research done in this thesis. The contextualised visual abstract can be found in Figure 2 and is explained in further detail in subchapter 3.3.

3.1 Research questions

The research questions for this thesis were selected based on the technology stack and challenges faced in the software project "TalentRekry," a web application where recruiters create and manage their recruitments. The project in question uses several third-party open-source JavaScript libraries, the security and quality of which are common concerns within the development team. The problem that was identified is that there is no easy way to know what factors should be considered when making the decision to include a library, and how these factors should be measured. The primary research question in this thesis is therefore related to the development of a model that aids in the library inclusion decision-making process:

RQ1: How can we develop a model for third-party JavaScript library inclusion?

This research question will be answered through the design science research process. The model draws from previous research covered in chapter 2, as well as new findings from a developer survey conducted as part of this research.

RQ1 has two sub-questions:

RQ1.1: What factors influence the decisions to include third-party libraries?

RQ1.2: How effective is the model in aiding in the decision making of library inclusion?

Research question 1.1 will be answered through a combination of previous research and a developer survey and contributes to the knowledge required to develop the model. Research question 1.2 will be answered by testing the model on existing libraries that are in use in the "TalentRekry" project, as well as some new libraries that are not used in the project. The results of these tests are then evaluated in a workshop by software developers to see if the model holds up with existing libraries, and to see if the model helps when it comes to deciding on including new libraries.

The model is intended to be a useful tool for practitioners who are planning on including third-party libraries in their projects while also serving as a synthesis of existing knowledge with some new perspectives for researchers.

3.2 Data collection and analysis

The model developed in this thesis was based on factors that were initially discovered in previous literature. Additional data was collected through an anonymous questionnaire sent to developers at FCG. The intent behind making the questionnaire anonymous was to minimise any potential barriers participants could have when it comes to taking part in the questionnaire, and to ensure that the answers were as honest as possible. The aim of this questionnaire was to gather the opinions of software developers on the importance of the identified factors, as well as to discover potential new factors that were not found in literature. The importance ratings were used to create weightings for each factor in the model.

The questionnaire asked the respondents to rate each factor from literature in terms of its importance on a 1-5 unipolar Likert-scale, where 1 signifies "not important at all" and 5 signifies "extremely important". A standard Likert-scale uses a bipolar format, which measures two opposites of an attribute, with a neutral option in the middle (Chyung, 2018). A unipolar Likert-scale on the other hand measures a single attribute from nothing

to an extreme. Some literature reviews consider unipolar scales more reliable than bipolar, and that bipolar scales suffer from some respondents being reluctant to choose negative responses (DeCastellarnau, 2018, Schaeffer & Dykema, 2020). The general consensus among the reviews, however, is that more evidence is required to confirm their findings. Since the aim of the questionnaire is to measure importance – a single attribute – using a unipolar scale made the most sense to me. In addition to the Likert-scale, each question also included an optional text field that allowed respondents to give comments on the question.

The survey had a final portion where the respondents were allowed to suggest any additional factors that they think are important, along with a comment and a rating scale like in the literature factors. This part was included to find potential factors that were missing from literature. Due to a lack of literature support and a small sample size, the one factor from this portion would not be included in the final model but is a good candidate for future research. The questionnaire was conducted using Webropol's survey and reporting tools. A textual version of the questionnaire can be found in Appendix A.

The respondents of the questionnaire were chosen using convenience sampling. Convenience sampling is a non-probability form of sampling where the researcher announces their study, and the participants choose whether they wish to participate (Stratton, 2021). Stratton (2021) describes some of the downsides of the approach. The results of convenience sampling cannot be generalised, as the results only apply to the participants, and sampling errors cannot be determined. Convenience sampling may also introduce motivation bias to the survey, as participation in the study depends on the motivation of the participants. Additionally, convenience sampling runs the risk of poor participation rates.

The software developers at FCG were in the target population of this study and were readily available. This meant that surveying them was a cost-efficient and fast way of building on the data found in literature, allowing for more focus to be spent on the development of the model while retaining perspectives from both researchers and practitioners. Due to the timescale of a master's thesis, convenience sampling seemed like the most reasonable option, despite its limitations.

According to Stratton (2021), there are certain steps that can be taken to improve the credibility of convenience sampling. The first of these is to avoid complex and vague study objectives, focusing instead on precisely defined input and outcome variables. This is addressed by the survey only having two objectives: determining the importance of the factors found in literature and finding potential factors that were missed in the literature review. The outcome variable is the ordinal Likert-scale data, with the optional text fields providing context for the results from a practitioner perspective.

The second step described by Stratton (2021) is to determine how participants will be recruited for the research, and to establish inclusion and exclusion criteria. The questionnaire was posted to a programming-themed Slack channel within FCG, consisting of 21 developers, including the author of this thesis. Most of the developers at FCG at are full-stack developers, meaning that they work on both server and client-side code. However, some developers may not be knowledgeable about the dependencies used in their projects, or dependency management in general, so the survey invitation was sent with an additional qualifier of "those with knowledge of front-end dependencies." Since the survey was anonymous, the actual knowledge of the respondents cannot be verified.

The third step Stratton (2021) recommends taking is recruiting as many participants as possible. A total of 8 respondents, or 40%, of the 20 potential respondents from the Slack channel responded to the survey, a far cry from the desirable 80% participation rate described by Stratton (2021). This small sample size is a major limitation to this portion of the study.

The fourth step is to describe the characteristics of the demographics of the participants (Stratton, 2021). The demographic consisted primarily of full-stack developers with most of them having at least four years of development experience. There were also two purely front-end developers in the potential sample. 85% of the potential respondents were male and 15% were female, though due to the anonymity of the survey, it is uncertain what the gender composition of the actual respondents is. Stratton (2021) also recommends not overstating the findings of the study, as convenience sampling only applies to the participant group. This is addressed in the limitations in chapter 7.3.

The sixth recommendation is to collect data in a diversified manner (Stratton, 2021). Due to time constraints, Slack was found to be the simplest and easiest manner of data collection, and as such this recommendation was not followed. The seventh recommendation, validating the questionnaire, was also missed due to time constraints. Additionally, the since the potential pool of respondents was already so small, pilot questionnaires and questionnaire reviews by experts from the company could have shrunk the final sample to an extremely small number, making validating the questionnaire a less appealing option.

The eighth recommendation is to avoid inappropriate use of statistical methods (Stratton, 2021). The Likert data was analysed as ordinal, using medians as opposed to means, as the data items are not evenly distributed.

The ninth recommendation is to identify possible external biases that may affect the participants, such as media reports, political conflict, or economic stresses (Stratton, 2021). Such external biases were not likely to have affected the respondents' responses, as third-party libraries were not any more prominent than usual in media at the time, and political conflict or economic stress are unlikely to affect the participants' biases towards third-party library inclusion factors.

The final recommendation is to include questions that relate to personal experience, and to be mindful of the length of the questionnaire, as lengthy questionnaires tend to be unreliable. The questionnaire did not have any questions relating to personal experience, but several respondents used the optional text fields to bring out their own experiences regarding the questions. The length of the questionnaire was a potential issue, as Webropol reported that one participant left the questionnaire half-finished for a long time before returning and finalising their answers.

Once the model was developed, a small-scale workshop was conducted with participants from FCG to gather data on practitioners' thoughts on the usefulness and effectiveness of the model. The workshop contained a showcase of the model, followed by the results of applying the model to a selection of libraries already in use in the "TalentRekry" project at FCG, as well as some libraries that were not in use. This was done to see whether the results of the model are valid on already included libraries, such as whether this information would have changed the decision to include a particular library, or whether the results would suggest not including something that should be included. The reason for testing the model on libraries not in use is to see whether the model would aid in the

decision making of including a new library. The discussion resulting from the workshop was collected and analysed, which can be found in chapter 6.

3.3 Contribution in the context of the visual abstract template

Following the recommendation of Runeson et al. (2020), this subchapter covers the contribution of the thesis in the context of the visual abstract template. The contextualised visual abstract can be seen in Figure 2.

The problem that this thesis is aiming to solve is that third-party libraries can cause various issues, and there is no easy way to know what these issues are, what their impact would be, and whether they are present on a library-by-library basis. To better understand the problem, a literature review of library-related issues was conducted, followed by a survey on practitioners' views on library inclusion. The problem solution, a set of criteria that can be tested against to determine whether a library is fit for inclusion, was designed based on solutions mentioned in literature. To validate the solution, the results were presented to software developers in a small-scale workshop. These results included a showcase of the model, as well as the results of applying the model to a number of real libraries, some that were already included in software projects at FCG, and some that were entirely new.

The main takeaway, the technological rule, is as follows: "To determine whether to include a JavaScript library in an enterprise software project, apply the dependency inclusion criteria." This is abstracted at its current level to highlight both the platform-specificity of the inclusion criteria, that being JavaScript libraries, as well as to whom the research is relevant to, being those working on enterprise software projects.

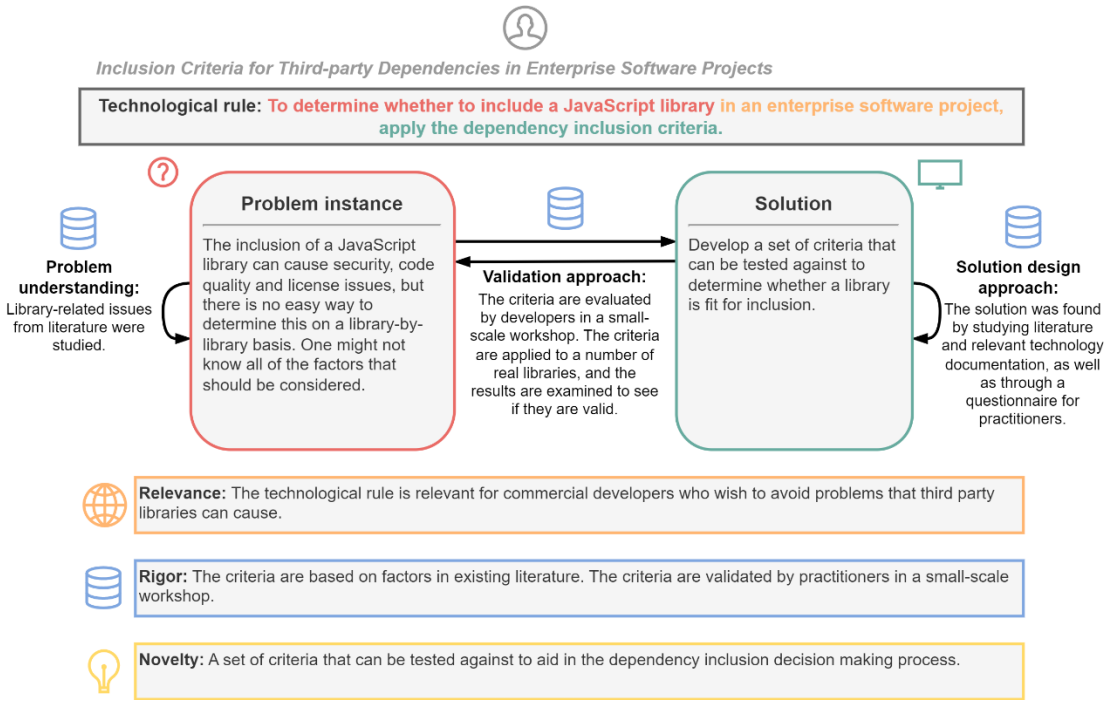


Figure 2. The contextualised visual abstract for this thesis.

To assess the value of the technological rule, we must assess it in the context of its relevance, rigour, and novelty. The research in this thesis is relevant for practitioners who are commercial developers that use third-party libraries in their projects, as the impact of

the problems third-party dependencies can cause is much greater in commercial projects compared to personal or free projects. Additionally, the research is based on a real problem instance. Factors that contribute to the rigour of the research are that the criteria are based on existing peer-reviewed literature, and that the criteria are validated by practitioners. The research is novel in that a testable set of criteria for third-party dependencies has yet to be designed prior to this thesis.

Since creating a testable model requires defining tests, the logical next step would be to automate these tests in the form of a tool that takes a third-party dependency as input and provides the results of the tests as output, perhaps with the addition of a recommendation on whether the library is fit for inclusion based on the importance weightings that were defined as part of this study. The development and evaluation of such a tool is outside the scope of this thesis but would be a prime candidate for future research.

4. Inclusion factor identification

This chapter defines the inclusion factors that are used in the inclusion model. The first subchapter includes all the factors that were identified in the literature review from chapter 2. The second subchapter includes the additional factors that were identified as part of the practitioner survey. The third subchapter covers the survey results in more detail, with a focus on the importance ratings of the survey.

4.1 Factors from literature

Inspired by the Criterion and Sub-criterion composition of the Jackson et al. (2011) software assessment criteria, these factors are separated into subcategories based on their common properties.

4.1.1 Transitive dependencies

The dependency smells identified by Jafari et al. (2021) are factors that should be considered when examining the transitive dependencies of a library. These factors are:

Pinned dependency: A dependency that is restricted to only one version. This could be done by omitting any comparator operators from the version number, by using the equality operator, or by using a restrictive caret.

URL dependency: A dependency that points to either a URL or a local path.

Restrictive constraint: A dependency that does not permit backward compatible updates. This could be done by having a patch constrained range, by using the patch level operator, or by using a restrictive caret.

Permissive constraint: A dependency that permits breaking changes. This could be done by having a minimum version number without defining a maximum version, or by using the asterisk wildcard as a version number.

No lockfile: The library is missing its package-lock or equivalent lockfile. While this does not affect the users of the library, it could be argued that a missing package-lock file makes the development environment of collaborators working on the library less stable.

Unused dependency: A dependency that is listed in package.json but not used. This includes any dependencies that are listed as runtime dependencies but are only used with a development flag. Such dependencies should be listed as development dependencies.

Missing dependency: A dependency that is used in code but is not found in package.json. This includes any dependencies that are always used in code but are only included with a development flag. Such dependencies should be listed as runtime dependencies.

In addition to the smells identified by Jafari et al. (2021), Zimmermann et al. (2019) noted that a high number of dependencies can cause issues in the form of multiple potential attack vectors for malicious parties, as well as making it possible that a dependency far up the chain causes seemingly unrelated libraries to break. This is therefore also a factor to be considered:

High dependency count: A library that has multiple dependencies, including transitive ones. Development dependencies do not affect users and as such are not counted.

4.1.2 Security

Various authors have identified security-related factors that should be considered when deciding to include a library (Much et al., 2019; Lauinger et al., 2018; Zimmermann et al., 2019; Imtiaz et al., 2021; Ntousakis et al., 2021; Rafnsson et al., 2020). These are:

Known vulnerabilities: The vulnerabilities reported by auditing solutions such as npm audit or snyk test. These should be run in "production" mode (--production flag in npm audit, snyk test runs in production mode by default). This is because development dependencies do not affect production, and as such will not compromise the security of our software.

Static code analysis: The vulnerabilities reported by static source code analysis solutions such as snyk code.

In addition to existing library vulnerabilities, Decan et al. (2018) and Zimmermann et al. (2019) show that a key part in ensuring the continued security of a library is its activity. More active packages tend to patch their vulnerabilities faster than less active ones, so activity is also a factor to be considered:

Maintainer activity: The last time a commit was made, or the repository was interacted with by a maintainer, such as by responding to an issue or pull request.

This factor is a lot more context-dependent than the previous ones. Certain types of libraries, particularly micropackages, may not require much maintenance at all since they are so simple.

4.1.3 Code quality

These factors affect the code quality of libraries. They do not necessarily cause issues on their own, but they affect the reliability and maintainability of the libraries and may introduce bugs later down the line. The factors are based on Theisen (2019), Patra et al. (2018), Sun et al. (2021) and Fard & Mesbah (2017).

Use of the global namespace: Not encapsulating the package into a single, library-specific object or module.

Code coverage: The percentage of lines of code covered by the tests of the library.

4.1.4 Licensing

This category only contains one factor derived from Gangadharan et al. (2008), Sojer & Henkel (2011), Gangadharan et al. (2012), Moraes et al. (2021) and Qiu et al. (2021). Licensing conflicts can lead to license disputes that can be expensive to resolve.

License conflicts: The degree of less permissive the license of the library is compared to the license of the target project.

4.2 Factors from practitioner survey

The practitioner survey contained a field where the respondents could write any factors that they thought were missing from the survey. There was one factor that came up in the practitioner survey that was not found in the literature review.

4.2.1 Accessibility

One respondent mentioned "Accessibility" as a factor to consider when including libraries, with the following reasoning:

> *"A project might have accessibility requirements that must be met. If the library is inaccessible, then that can cause a lot of additional effort from the development team, or potentially the need to abandon the library."*

The World Wide Web Consortium Web Accessibility Initiative, or W3C WAI, defines web accessibility to mean that websites, tools, and technologies are designed and developed so that people with disabilities can use them (W3C WAI, 2022). The W3C WAI develops a set of accessibility standards known as the Web Content Accessibility Guidelines (WCAG), which are intended to provide a technical standard for web developers (W3C WAI, 2023). A paraphrased summary of the WCAG 2.1 guidelines by W3C WAI (2018) can be found in Table 1.

Table 1. WCAG 2.1 at a Glance (W3C WAI, 2018).

Principle	Summarised guideline
Perceivable	"Provide text alternatives for non-text content."
	"Provide captions and other alternatives for multimedia."
	"Create content that can be presented in different ways, including by assistive technologies, without losing meaning."
	"Make it easier for users to see and hear content."
Operable	"Make all functionality available from a keyboard."
	"Give users enough time to read and use content."
	"Do not use content that causes seizures or physical reactions."
	"Help users navigate and find content."
	"Make it easier to use inputs other than a keyboard."
Understandable	"Make text readable and understandable."
	"Make content appear and operate in predictable ways."
	"Help users avoid and correct mistakes."
Robust	"Maximize compatibility with current and future user tools."

In a study by Bi et al. (2022), it was found that only 30% of participants in their data sample had direct accessibility-related work experience. They also noted that of the 30% that did have accessibility-related work experience, accessibility was adopted in a light-weight way, being confined to colours and layout as opposed to things like screen reader compatibility.

However, very little, if any, literature can be found discussing accessibility specifically in the context of third-party libraries. The lack of literature on the topic may be due to some libraries having nothing to do with the user experience, and thus not having to concern themselves with accessibility. Since accessibility was only mentioned by one respondent with an importance rating of "Somewhat important" (3), determining its

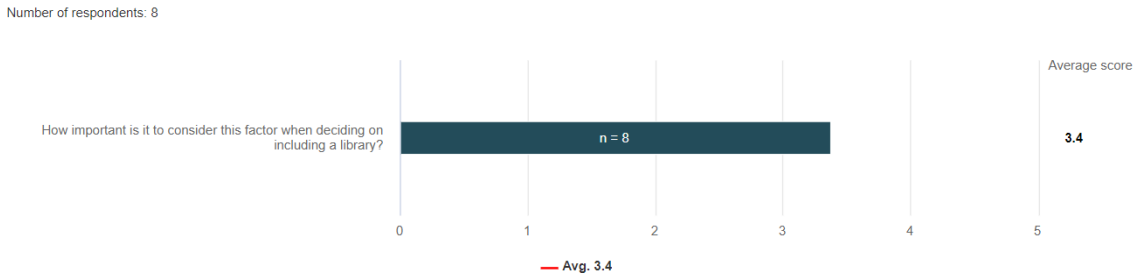
importance is not reliable. For these reasons, accessibility will not be considered for the final model, but studying accessibility in third-party libraries could be a potential topic for future research. It would be interesting to see how third-party libraries, if applicable to the library, conform to various levels of WCAG 2.1 requirements, seeing as the Bi et al. (2022) sample had a number as low as 30% for developer accessibility experience.

4.3 Importance and impact of factors

As part of the survey, the respondents were asked to rate the importance of each factor. Additionally, each factor had an optional text field where the respondents could give their reasoning for the chosen rating. This subchapter covers these survey results in detail and contains a final importance rating for each factor based on the data. A summary of the importance ratings can be found in Table 2.

4.3.1 Pinned dependency

The median importance for Pinned dependency was 3, meaning "Somewhat important." The full results for this question can be seen in Figure 3.



	Not important at all	Not very important	Somewhat important	Very important	Extremely important	Total	Average	Median
How important is it to consider this factor when deciding on including a library?	0	1	4	2	1	8	3.4	3.0
	0.0%	12.5%	50.0%	25.0%	12.5%			
Total	**0**	**1**	**4**	**2**	**1**	**8**	**3.4**	**3.0**

Figure 3. Questionnaire results for "Pinned dependency."

Two respondents think that pinned dependencies should be avoided to allow minor and patch updates:

> *"Restricting to exact version might lead to bugs or vulnerabilities not being fixed in patch updates."*

> *"Not good for security. Always allow minor and patch as these should not break anything."*

One respondent liked pinned dependencies, but acknowledged that it can cause problems in libraries:

> *"While I like pinned dependencies, I think it gets hard with eg. diamond dependency problem as having two different versions of the same dependency can be catastrophic."*

Allowing minor and patch updates reduces the likelihood of requiring two differing versions of the same dependency, though it will not eliminate the probability altogether.

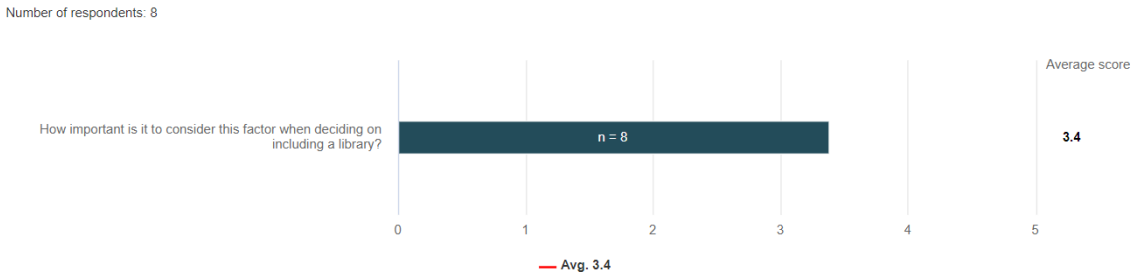
One respondent was sceptical about trusting library authors with SemVer compliance:

> *"Depends on project size in my opinion. Bigger the project, bigger the possibilities of something breaking when updating dependency. Still I would rather update the library's dependency when possible."*

Overall, pinned dependencies will not cause an issue most of the time, but allowing minor and patch updates should help the libraries stay up to date in terms of security and bug fixes with little to no downsides, so an importance rating of "Somewhat important" seems accurate.

4.3.2 URL dependency

The median importance for URL dependency was 3, meaning "Somewhat important." The full results for this question can be seen in Figure 4.



	Not important at all	Not very important	Somewhat important	Very important	Extremely important	Total	Average	Median
How important is it to consider this factor when deciding on including a library?	0	1	4	2	1	8	3.4	3.0
	0.0%	12.5%	50.0%	25.0%	12.5%			
Total	**0**	**1**	**4**	**2**	**1**	**8**	**3.4**	**3.0**

Figure 4. Questionnaire results for "URL dependency."

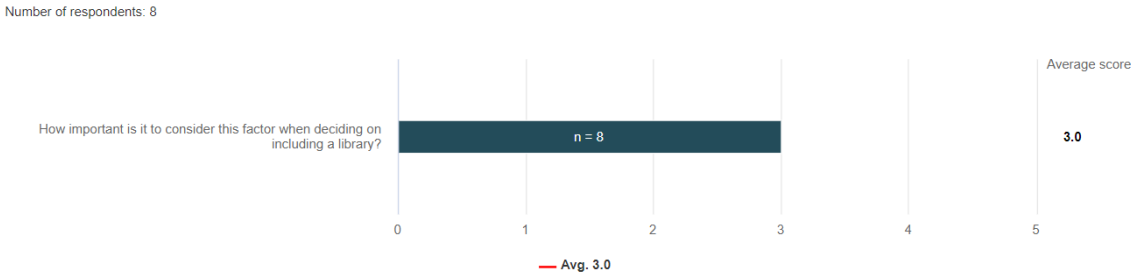
One respondent finds URL dependencies "hard to track." Another respondent prefers not hard coding locations:

> *"I'd prefer not hard coding the location of the dependency, if it is resolvable with the package manager."*

One respondent also brings up that a URL dependency can have the same issues as a pinned dependency. For these reasons, having the same answers and importance rating as pinned dependency makes sense.

4.3.3 Restrictive constraint

The median importance for Restrictive constraint was 3, meaning "Somewhat important." The full results for this question can be seen in Figure 5.



	Not important at all	Not very important	Somewhat important	Very important	Extremely important	Total	Average	Median
How important is it to consider this factor when deciding on including a library?	0	1	6	1	0	8	3.0	3.0
	0.0%	12.5%	75.0%	12.5%	0.0%			
Total	**0**	**1**	**6**	**1**	**0**	**8**	**3.0**	**3.0**

Figure 5. Questionnaire results for “Restrictive constraint.”

One respondent pointed out that bug fixes may not get backported to old versions:

> *“Dependency author might not backport bug fixes to older versions, so it could be useful to allow backward compatible updates.”*

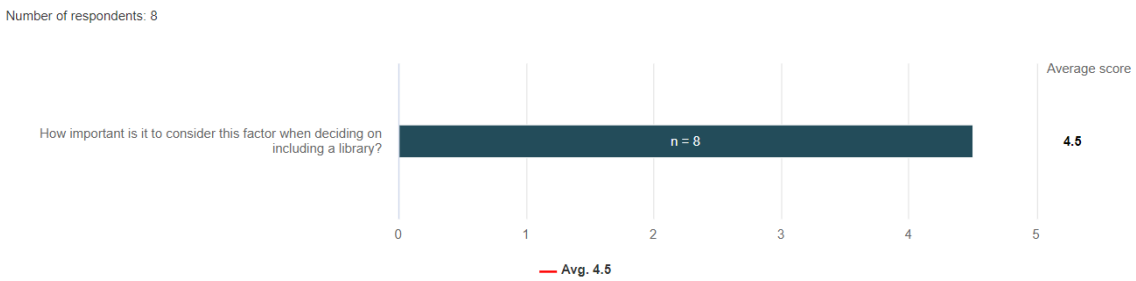
Only permitting bugfixes and not non-breaking updates can also have problems with having multiple versions of the same dependency, though to a lesser degree than a pinned dependency:

> *“I'm usually okay with some restrictions, but again the diamond dependency could be a problem.”*

Restrictive constraint is yet another “Somewhat important” factor to consider, offering similar issues to a pinned dependency, though to a lesser extent.

4.3.4 Permissive constraint

The median importance for Permissive constraint was 5, meaning “Extremely important.” The full results for this question can be seen in Figure 6.



	Not important at all	Not very important	Somewhat important	Very important	Extremely important	Total	Average	Median
How important is it to consider this factor when deciding on including a library?	0	0	1	2	5	8	4.5	5.0
	0.0%	0.0%	12.5%	25.0%	62.5%			
Total	**0**	**0**	**1**	**2**	**5**	**8**	**4.5**	**5.0**

Figure 6. Questionnaire results for “Permissive constraint.”

Of the respondents that left a comment, all of them agreed that breaking changes should not be automatically applied:

> *"Breaking changes should be handled manually."*

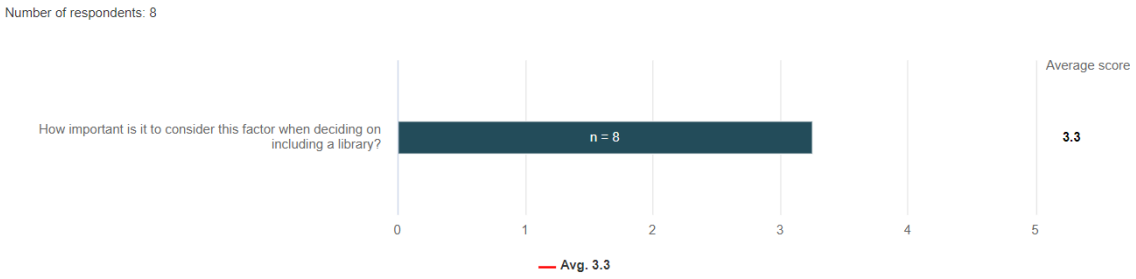
> *"I think allowing breaking changes will bite you at some point."*

> *"Whole project could be on the line so I think it is important to take into account possible breaking changes."*

Ensuring that dependencies do not break your software is understandably an extremely important thing to consider.

4.3.5 No lockfile

The median importance for No lockfile was 4, meaning "Very important." The full results for this question can be seen in Figure 7.



	Not important at all	Not very important	Somewhat important	Very important	Extremely important	Total	Average	Median
How important is it to consider this factor when deciding on including a library?	1	2	0	4	1	8	3.3	4.0
	12.5%	25.0%	0.0%	50.0%	12.5%			
Total	**1**	**2**	**0**	**4**	**1**	**8**	**3.3**	**4.0**

Figure 7. Questionnaire results for "No lockfile."

The importance of a missing lockfile was very divisive among respondents, with the question receiving answers from both ends of the spectrum. The comments reveal multiple differing perspectives. One respondent is of the opinion that a lockfile is not needed if the package.json is correctly versioned:

> *"package.json with correct versioning should suffice."*

Two respondents considered a stable library development environment, as enabled by the lockfile, an important thing to consider when it comes to including a library:

> *"A stable development environment is the way to go."*

> *"Stable environment is needed. Also for CI builds."*

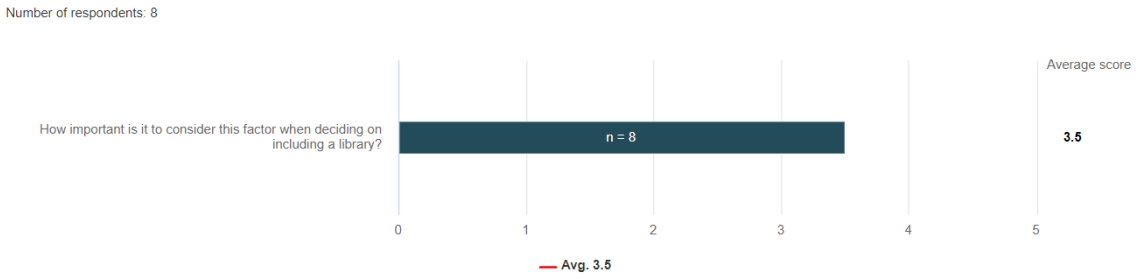
One respondent considered the lack of a lockfile to be a mere annoyance for library developers, seeing as it does not affect its users directly:

> *"While annoying if I have to improve the library, it does not affect me that much."*

I suspect that the respondents who did not see the lack of a lockfile as important viewed the question from the viewpoint of a library consumer, seeing as lockfiles only affect the library developers themselves. On the other hand, the respondents who consider the lack of a lockfile important viewed the question from the viewpoint of library quality, possibly relating a stable development environment to a higher quality library. While both perspectives are valid, the prevailing sentiment leans towards the lack of a lockfile being a very important thing to consider.

4.3.6 Unused dependency

The median importance for Unused dependency was 3, meaning "Somewhat important." The full results for this question can be seen in Figure 8.



	Not important at all	Not very important	Somewhat important	Very important	Extremely important	Total	Average	Median
How important is it to consider this factor when deciding on including a library?	0	0	5	2	1	8	3.5	3.0
	0.0%	0.0%	62.5%	25.0%	12.5%			
Total	**0**	**0**	**5**	**2**	**1**	**8**	**3.5**	**3.0**

Figure 8. Questionnaire results for "Unused dependency."

Respondents were unanimously in agreement that unused dependencies should be removed:

> *"Unnecessary dependency clutter."*

> *"Clean up everything."*

> *"Unused dependency should be deleted."*

> *"While it's hard to know which dependencies are unused without diving into the library, I'd prefer not pulling unused stuff."*

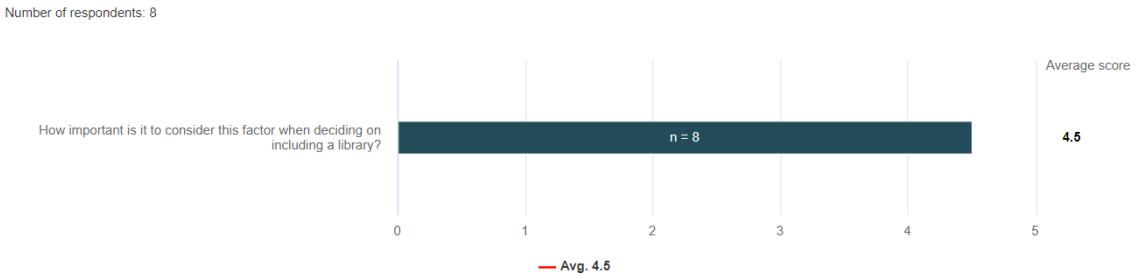
One respondent also looked at the question from a quality angle:

> *"Could be a sign of poor quality."*

While the risks associated with unused dependencies are quite low, they add unnecessary clutter and do not give a good impression of the quality of the library, so it is somewhat important to take this factor into consideration.

4.3.7 Missing dependency

The median importance for Missing dependency was 5, meaning “Extremely important.” The full results for this question can be seen in Figure 9.



	Not important at all	Not very important	Somewhat important	Very important	Extremely important	Total	Average	Median
How important is it to consider this factor when deciding on including a library?	0	0	1	2	5	8	4.5	5.0
	0.0%	0.0%	12.5%	25.0%	62.5%			
Total	**0**	**0**	**1**	**2**	**5**	**8**	**4.5**	**5.0**

Figure 9. Questionnaire results for “Missing dependency.”

The comments from two respondents concern the unpredictable problems that may arise from missing dependencies:

> *“Can cause unforeseen problems.”*

> *“List everything used, always.”*

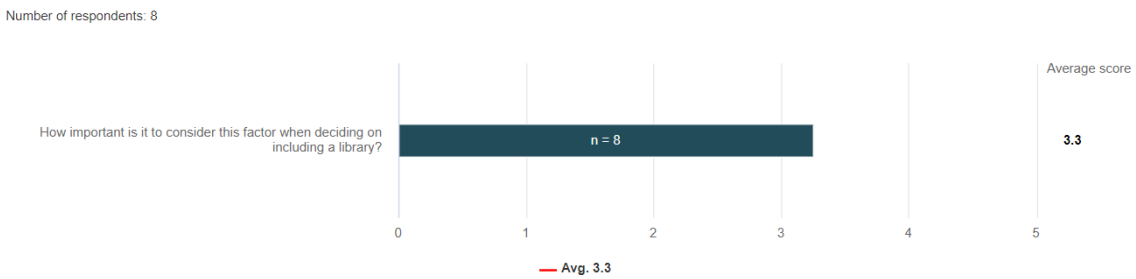
The matter of quality also arises once again:

> *“I think this is one of the extreme ways to demonstrate poor quality.”*

At worst, a missing dependency will not work and break its users’ programs, and at best it shows an extreme lack of care from the library authors. Due to this it is extremely important to consider when deciding to include a library.

4.3.8 Number of dependencies

The median importance for Number of dependencies was 3.5, putting it between “Somewhat important” and “Very important.” The full results for this question can be seen in Figure 10.



	Not important at all	Not very important	Somewhat important	Very important	Extremely important	Total	Average	Median
How important is it to consider this factor when deciding on including a library?	1	1	2	3	1	8	3.3	3.5
	12.5%	12.5%	25.0%	37.5%	12.5%			
Total	**1**	**1**	**2**	**3**	**1**	**8**	**3.3**	**3.5**

Figure 10. Questionnaire results for “Number of dependencies.”

The number of dependencies is another divisive factor among the respondents. One respondent highlights the security and bugs side of dependencies:

> *“A large number of transitive dependencies is harder to manage in terms of security and bugs.”*

Another respondent thinks that a large number of dependencies could be a sign of poor quality:

> *“When all other factors are controlled this has a lower priority. Less is of course less but...”*

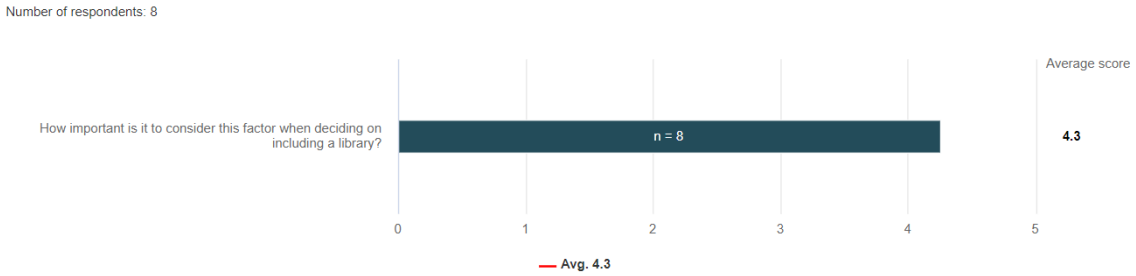
One of the respondents, however, is not so concerned:

> *“I do look for focused packages doing one thing (group of tightly related things) well instead of packages including the kitchen sink, but not in the "left pad" sense as tree shaking is a thing, so I'm not worried about that.”*

A large number of transitive dependencies can become hard to manage, more prone to bugs, as well as providing malicious actors more attack vectors. On the other hand, features such as tree shaking, which removes unused code, can help manage the size of one’s bundles. While issues from having many dependencies are manageable with enough effort, it is still important to consider how many new dependencies a library will add to your dependency tree upon inclusion.

4.3.9 Known vulnerabilities

The median importance for Known vulnerabilities was 4, meaning “Very important.” The full results for this question can be seen in Figure 11.



	Not important at all	Not very important	Somewhat important	Very important	Extremely important	Total	Average	Median
How important is it to consider this factor when deciding on including a library?	0	0	1	4	3	8	4.3	4.0
	0.0%	0.0%	12.5%	50.0%	37.5%			
Total	**0**	**0**	**1**	**4**	**3**	**8**	**4.3**	**4.0**

Figure 11. Questionnaire results for “Known vulnerabilities.”

When it comes to known vulnerabilities, the use case of the library seems to determine how important it is to consider:

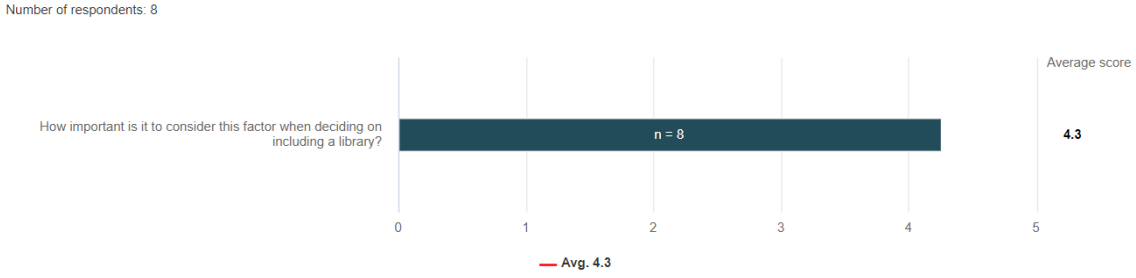
> *“Depends on how well maintained library it is and what is the status of the project. Bad for production. Less bad for dev tools and early stage products.”*

> *“It depends on the library and its use case, if this is extremely important, not that important.”*

For any libraries intended to be used in production code, it is very important that a library has as few vulnerabilities as possible. If the library is used only as a development dependency, then vulnerabilities are less important, as malicious actors would not be able to use them to attack the system in production, and thus will not compromise the security of the software using that library.

4.3.10 Source code security

The median importance for Source code security was 5, meaning “Extremely important.” The full results for this question can be seen in Figure 12.



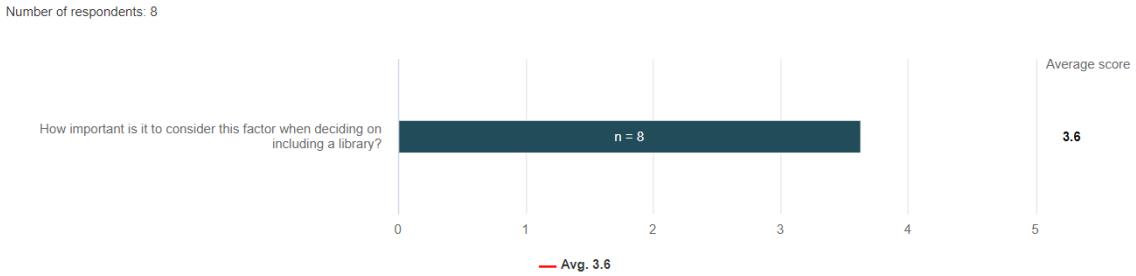
	Not important at all	Not very important	Somewhat important	Very important	Extremely important	Total	Average	Median
How important is it to consider this factor when deciding on including a library?	0	1	1	1	5	8	4.3	5.0
	0.0%	12.5%	12.5%	12.5%	62.5%			
Total	**0**	**1**	**1**	**1**	**5**	**8**	**4.3**	**5.0**

Figure 12. Questionnaire results for “Source code security.”

Static code analysis is great at finding issues such as Cross-Site Scripting vulnerabilities, which are very severe and should not be included in production code. As such, it is extremely important to consider source code security when including a library, if possible.

4.3.11 Maintainer activity

The median importance for Maintainer activity was 4, meaning “Very important.” The full results for this question can be seen in Figure 13.



	Not important at all	Not very important	Somewhat important	Very important	Extremely important	Total	Average	Median
How important is it to consider this factor when deciding on including a library?	0	1	2	4	1	8	3.6	4.0
	0.0%	12.5%	25.0%	50.0%	12.5%			
Total	**0**	**1**	**2**	**4**	**1**	**8**	**3.6**	**4.0**

Figure 13. Questionnaire results for “Maintainer activity.”

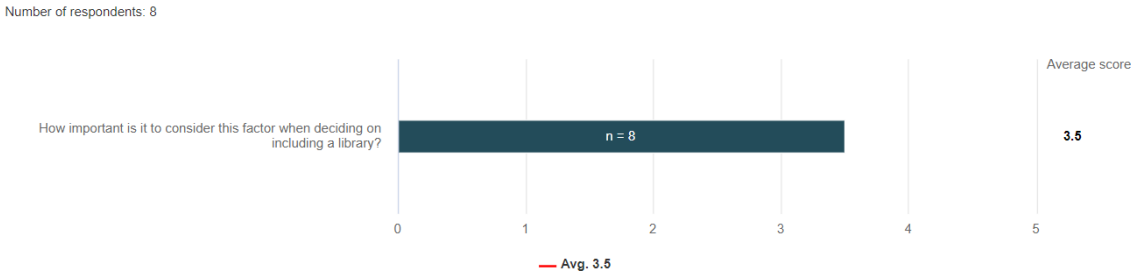
The lifetime of a library is a common concern among respondents:

> *“I think this indicates how "alive" the library is and thus hopefully correlating with its expected lifetime.”*
>
> *“It's always a commitment to a library. It can be hard to replace later.”*
>
> *“Again, if library is not updated, more security threats may be present in the near future.”*
>
> *“Active maintainers mean that bugs and vulnerabilities are more likely to get fixed faster.”*

Indeed, if a library gets abandoned, it is only a matter of time before bugs and security issues start to appear. If there is no one to fix these, it is up to the library user to solve the problems themselves or to migrate to another library, which can be quite the hassle. Because of this, it is very important to consider how active the maintainers of a library are before choosing to include it.

4.3.12 Use of the global namespace

The median importance for Use of the global namespace was 4, meaning “Very important.” The full results for this question can be seen in Figure 14.



	Not important at all	Not very important	Somewhat important	Very important	Extremely important	Total	Average	Median
How important is it to consider this factor when deciding on including a library?	0	1	2	5	0	8	3.5	4.0
	0.0%	12.5%	25.0%	62.5%	0.0%			
Total	**0**	**1**	**2**	**5**	**0**	**8**	**3.5**	**4.0**

Figure 14. Questionnaire results for “Use of the global namespace.”

Polluting the global namespace, as expressed quite aptly by one respondent, is not good:

> *“Pollution of global namespace = no good.”*

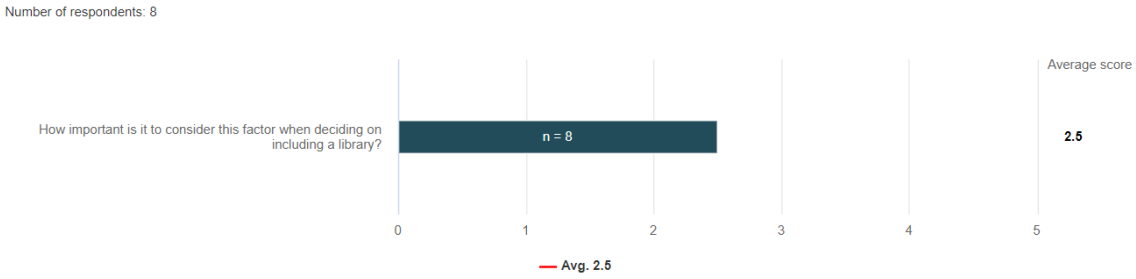
Use of the global namespace could also be a sign of poor quality:

> *“I think this might be a good indicator of the quality of the library.”*

While naming conflicts are reasonably easy to avoid, global namespace pollution also increases memory usage. This happens as variables never go out of scope, and thus are never garbage collected. Because of this it is very important to consider whether a library uses the global namespace when deciding on including it.

4.3.13 Code coverage

The median importance for Code coverage was 3, meaning “Somewhat important.” The full results for this question can be seen in Figure 15.



	Not important at all	Not very important	Somewhat important	Very important	Extremely important	Total	Average	Median
How important is it to consider this factor when deciding on including a library?	1	2	5	0	0	8	2.5	3.0
	12.5%	25.0%	62.5%	0.0%	0.0%			
Total	**1**	**2**	**5**	**0**	**0**	**8**	**2.5**	**3.0**

Figure 15. Questionnaire results for “Code coverage.”

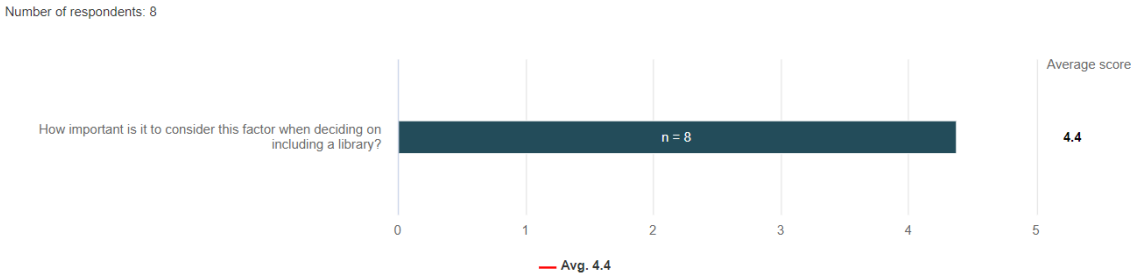
One respondent thinks that the higher the coverage, the better:

> *"Not enough "hands-on" knowledge on this subject, but more tests, the better in my opinion."*

Automated testing could be seen as a sign of good quality. Tests ensure that changes do not break functionality, which also makes SemVer compliance easier: if tests break, then that is a sign of a breaking change, which requires incrementing the library's major version. However, libraries without tests, especially simple libraries, can also be of good quality if they are manually tested. Code coverage does not guarantee that the tests are good, only that the code is being tested. Additionally, code coverage only accounts for functionality that is in the code, meaning that problems caused by faulty specification can still occur even with 100% code coverage. For these reasons, code coverage is a somewhat important factor to consider, but should not be the main reason to include or exclude a library.

4.3.14 License conflicts

The median importance for License conflicts was 5, meaning "Extremely important." The full results for this question can be seen in Figure 16.



	Not important at all	Not very important	Somewhat important	Very important	Extremely important	Total	Average	Median
How important is it to consider this factor when deciding on including a library?	0	0	2	1	5	8	4.4	5.0
	0.0%	0.0%	25.0%	12.5%	62.5%			
Total	**0**	**0**	**2**	**1**	**5**	**8**	**4.4**	**5.0**

Figure 16. Questionnaire results for "License conflicts."

One respondent highlighted the key problem with license conflicts:

> *"License conflicts can cause legal issues."*

Another respondent commented on the difficulties of keeping track of licenses in transitive dependencies:

> *"I do not want to break licenses, especially direct dependencies. However, keeping track of transitive licenses gets a little hard, but I try to manage them as well."*

Legal issues can result in expensive fees or the need to remove the offending piece of software entirely, forcing the development team to create their own replacement or to seek an alternative library. It is therefore extremely important to consider whether any potential licensing conflicts are present before choosing to include a library.

4.3.15 Summary of the importance ratings

The importance ratings from the questionnaire are summarised in Table 2.

Table 2. The importance ratings based on the questionnaire results.

Factor	Importance rating[1] [2]
Pinned dependency	3
URL dependency	3
Restrictive constraint	3
Permissive constraint	5
No lockfile	4
Unused dependency	3
Missing dependency	5
Number of dependencies	3.5
Known vulnerabilities	4
Source code security	5
Maintainer activity	4
Use of the global namespace	4
Code coverage	3
License conflicts	5

Notably, no factor had an importance rating lower than 3. This suggests that all the factors identified in literature are at least somewhat important to consider. The most critical of factors to consider are ones that have the potential to break the software, such as "permissive constraint" or "missing dependency," to introduce severe security issues, as is the case with "source code security," or to cause legal issues, such as with "license conflicts." Due to their severe effects, the existence of these factors should probably prevent library inclusion.

The very important factors to consider are related to quality, such as "no lockfile" or "the use of the global namespace," the longevity of the library in "maintainer activity," as well as the security of the library with "known vulnerabilities." These factors have more nuance to them, with library inclusion depending on the level of severity when it comes to the factor. For example, a library with a few low-severity vulnerabilities might be acceptable to some, while a library with multiple critical vulnerabilities should certainly not be included. Whether or not the library is intended to be used exclusively for development also plays a role in the decision-making process.

"The number of dependencies" is a unique factor because it is the only one to rank between two tiers of importance. The rest of the factors lie in the "somewhat important" tier of importance, being related to various quality related issues, such as "code coverage" or "unused dependency," or dependency constraints, such as "pinned dependency," "url dependency" or "restrictive constraint." These on their own are unlikely to prevent one from including a library, but too many of them at once is a sign of a poor-quality library.

---

[1] median of responses

[2] 1 = not important at all, 2 = not very important, 3 = somewhat important, 4 = very important, 5 = extremely important

5. Model for inclusion criteria

In this chapter, the final model for inclusion criteria is defined based on the factors and weightings identified in chapter 4, with measures for testing the criteria being based on literature from chapter 2. Much like the factors in chapter 4, the model is divided into categories of criteria based on the properties of each criterion. Some criteria only contain one factor, while others consist of multiple factors. The name of each factor is presented in bold. The full, formatted model used for testing libraries can be found in Appendix B., while a shorter, summarised version can be found in Table 3.

5.1 The model scoring system

For the model to be more than a checklist of things to examine about a library before inclusion, the model includes a proposed scoring system intended to provide assistance on deciding whether the library is fit for inclusion. Each identified factor is given a score based on the measurement of the library, and each score is then added together for a final score. If the final score is low enough, the library could be considered fit for inclusion.

The developer questionnaire results are a great starting point for determining weightings for the inclusion criteria. However, it could be argued that most of the factors included here are not binary – for example, when it comes to known vulnerabilities, the vulnerabilities can be either low, medium, high, or critical in their severity. It would not be useful to treat low and critical severity vulnerabilities the same when it comes to deciding on whether we should include a specific library. Because of this, I propose giving each factor a score based on its measurement, which is then multiplied by the impact rating. This way we can have a difference between a library with two low-severity vulnerabilities and one with a critical severity vulnerability. When examining other criteria found in previous literature, the lack of weights in the assessment criteria defined by Jackson et al. (2011) made it difficult for those not intimately familiar with the criteria to know which criteria are the most important, which is why this model includes the importance ratings.

As each factor is different, they will also have to be measured and scored differently. The way of measuring each factor is based on the literature from chapter 2, while the scoring is based on the attributes of the factor being measured. Binary factors, such as whether a lockfile exists or not, will be worth four points if their condition is fulfilled. Non-binary factors that do not have a scale of severity, such as the number of dependencies, will be worth two points for each fulfilled condition. Non-binary factors that do have a scale of severity, such as known vulnerabilities, will be worth one to four points, depending on the severity of each fulfilled condition. Non-binary factors start at half the points of binary factors as there could be multiple of them. For factors with a scale, medium severity is considered equivalent to a non-scalar factor. Maintainer activity and code coverage are factors that must be scored in a unique way. The scoring specifics of each factor is explained in their respective subchapters, as well as how to measure them.

5.2 Criteria based on transitive dependencies

The transitive dependency factors are divided into three criteria: Dependency constraints, dependency use and dependency count.

5.2.1 Dependency constraints

This criterion is based on the dependency constraints of the transitive dependencies in a specific library. The criterion defines whether the transitive dependencies of the library have been properly configured. As development dependencies are not transitively included, this criterion only concerns production dependencies.

Pinned dependency: Pinned dependencies make it so that the dependencies will not be updated to include security and bug fixes. This is measured by analysing the package.json of the library and checking for dependencies that are, for example, in the following format: 1.2.3 or =1.2.3 or ^0.0.3. Each pinned dependency is worth 2 points, as a library could have multiple dependencies. Pinned dependency has a score multiplier of 3, as it was determined to be somewhat important to consider.

URL dependency: Depending on the URL, this may cause every version to be fetched, including breaking changes, or it may act like a pinned dependency where it never updates. This is measured by analysing the package.json for URLs or file paths in place of version numbers. Each URL dependency is worth 2 points, as a library could have multiple dependencies. URL dependency has a score multiplier of 3, as it was determined to be somewhat important to consider.

Restrictive constraint: A restrictive constraint disallows backward compatible updates. Library developers may not backport their bugfixes to earlier minor versions, so allowing minor updates ensures most up to date bugfixes without breaking changes. This is measured by analysing the package.json for version ranges or numbers such as the following: >=1.2.0 <1.3.0 or ~1.2.3 or ^0.2.3. Each dependency with a restrictive constraint is worth 2 points, as a library could have multiple dependencies. Restrictive constraint has a score multiplier of 3, as it was determined to be somewhat important to consider. A pinned dependency is technically also a restrictive constraint, but as they go into their own factor, they are not counted here to prevent duplicate entries.

Permissive constraint: Permissive constraints allow breaking changes. This is measured by analysing the package.json for version ranges such as >=1.2.0, or for the wildcard asterisk (*). Each dependency with a permissive constraint is worth 2 points, as a library could have multiple dependencies. Permissive constraint has a score multiplier of 5, as it was determined to be extremely important to consider.

5.2.2 Dependency definitions

This criterion is based on the library's definitions of dependencies. The criterion defines whether the library has properly defined its dependencies.

No lockfile: Without a lockfile (such as the package-lock.json file), there is no guarantee that installations at different times by the library developers will yield the same results, meaning that the development environment is not stable. This is measured by analysing the repository file structure. If the repository has a package.json file without a package-lock.json, yarn.lock, pnpm-lock.yaml or equivalent file, then the library has failed in this factor. A missing lockfile is worth 4 points, as this is a binary factor. No lockfile has a score multiplier of 4, as it was determined to be very important to consider.

Unused dependency: Unused dependencies take up unnecessary space and may create unneeded dependency chains. This is measured by analysing the package.json file and

parsing the source code files for usages of the listed dependencies. Alternatively, one could use a tool such as the npm package “depcheck,” which lists unused and missing dependencies. Each unused dependency is worth 2 points, as a library could have multiple dependencies. Unused dependency has a score multiplier of 3, as it was determined to be somewhat important to consider.

Missing dependency: Missing dependencies can cause bugs and unexpected behaviour. This is measured by analysing the package.json file and parsing the source code files for usages of the listed dependencies. Alternatively, one could use a tool such as the npm package “depcheck,” which lists unused and missing dependencies. Each missing dependency is worth 2 points, as a library could have multiple dependencies. Missing dependency has a score multiplier of 5, as it was determined to be extremely important to consider.

5.2.3 Dependency count

This criterion is based on the library’s number of dependencies used. The criterion defines whether the library is potentially using too many other dependencies.

Number of dependencies: Each additional dependency multiplies the attack surface for malicious actors. When a library has multiple transitive dependencies, there is a higher chance that one of them becomes abandoned, potentially resulting in vulnerabilities that never get fixed. Additionally, the maintainers of abandoned packages may get their accounts hijacked, giving attackers the ability to deploy malicious code. This is measured by analysing the package.json for the number of dependencies, followed by analysing each transitive dependency in the same way. Alternatively, one could use a tool such as the npm package “howfat,” which counts the dependency tree of a package. As development dependencies are not transitively included with libraries, they are also ignored here. Each dependency is worth 2 points, as the library could have multiple dependencies. Number of dependencies has a score multiplier of 3.5, as it was determined to be between somewhat important and very important to consider.

5.3 Criteria based on security

The security factors are split into two criteria: Vulnerabilities and maintainer activity.

5.3.1 Vulnerabilities

This criterion is based on the vulnerabilities of the library. The criterion defines whether the library is keeping up with security updates and using unsafe coding practices.

Known vulnerabilities: Known vulnerabilities can be exploited to attack software that is using a vulnerable library. This is measured by cloning the library and running either npm audit or snyk test on it in production mode. Each low severity vulnerability is worth 1 point, each medium severity vulnerability is worth 2 points, each high severity vulnerability is worth 3 points, and each critical severity vulnerability is worth 4 points. Known vulnerabilities has a score multiplier of 4, as it was determined to be very important to consider.

Static code analysis: Libraries can have vulnerabilities or other security issues that are yet to be reported which could cause issues in the future. This is measured by cloning the library and using a static code analysis tool such as snyk code. Each low severity issue is worth 1 point, each medium severity issue is worth 2 points, and each high severity issue is worth 3 points. Tools other than snyk code may require a different scoring mechanism depending on how they report their findings. Static code analysis has a score multiplier of 5, as it was determined to be extremely important to consider.

5.3.2 Maintainer activity

This criterion is based on the maintainers' activity in the library's repository. The criterion defines whether the library is being actively maintained.

Maintainer activity: A lack of activity can be a sign of a poorly maintained library. Inactive libraries are less likely to patch their vulnerabilities or update their dependencies. This is measured by analysing the maintainers' account activity for the repository, such as commits or issue activity. Every 10 weeks of inactivity since the day of measurement is worth one point. Maintainer activity has a score multiplier of 4, as it was determined to be very important to consider.

5.4 Criteria based on quality

The quality factors are divided into two criteria: Code coverage and namespacing.

5.4.1 Code coverage

This criterion is based on the code coverage of the tests of the library. The criterion defines whether the library is thoroughly tested, though good coverage does not necessarily mean that the tests themselves are good.

Code coverage: Code coverage can be used as a metric of the quality of testing. While coverage alone does not guarantee good tests, high coverage can be an indicator that changes made to the library are not likely to break existing functionality, and that the library is relatively bug-free. Many libraries report the coverage of their testing themselves, but this can also be measured by cloning the library and running the tests using a code coverage tool such as Istanbul.js. A code coverage of 100% is worth 0 points, 99-75% is worth 1 point, 74-50% is worth 2 points, 49-25% is worth 3 points, and 24-0% is worth 4 points. Code coverage has a score multiplier of 3, as it was determined to be somewhat important.

5.4.2 Namespacing

This criterion is based on how whether the library uses the global namespace. The criterion defines whether the library is safe to use without having to worry about name collision.

Use of the global namespace: Using the global namespace can cause issues with name collision, forcing the users to be mindful of their own naming so as not to break the library. This is measured by analysing the source code of the library and making sure that it is encapsulated in its own classes or modules. Use of the global namespace is worth 4

points, as this is a binary factor. Use of the global namespace has a score multiplier of 4, as it was determined to be very important.

5.5 Criteria based on licensing

The lone licensing factor features in its own criterion: licensing.

5.5.1 Licensing

This criterion is based on the license of the library. The criterion defines whether the library is compatible with one's project.

License conflicts: A library with a restrictive license, such as a copyleft license, is often not compatible with a closed source commercial product. If one is found to be using an incompatible license, it could cause expensive legal issues and extra work to replace the library with a permissive one, or to create matching functionality from scratch. This can be measured using Snyk's license issue checker, or by manually comparing the level of permissiveness of the license. The npm package "howfat" can be used to easily find out the licenses of transitive dependencies. An incompatible license somewhere in the dependency tree is worth 4 points, as this is a binary factor. License conflicts has a score multiplier of 5, as it was determined to be extremely important.

5.6 Summary of the model

The model is summarised in Table 3.

Table 3. The dependency inclusion model summarised. See Appendix B. for the full model.

Criterion	Factor	Explanation	Measuring	Import.[3]	Source
Dependency constraints	Pinned dependency	A dependency that is restricted to only one version.	Analysing the package.json file. 2 points per pinned dependency.	3	Jafari et al. (2021)
	URL dependency	A dependency that points to a URL (or local path).	Analysing the package.json file. 2 points per URL dependency.	3	Jafari et al. (2021)
	Restrictive constraint	A dependency that does not permit backward compatible updates.	Analysing the package.json file. 2 points per restrictive constraint.	3	Jafari et al. (2021)
	Permissive constraint	A dependency that permits breaking changes.	Analysing the package.json file. 2 points per permissive constraint.	5	Jafari et al. (2021)
Dependency definitions	No lockfile	The package repository is missing its lockfile.	Analysing the repository file structure. 4 points if the repository is missing its lockfile.	4	Jafari et al. (2021)
	Unused dependency	A dependency that is listed in package.json but not used.	Analysing the package.json file, as well as parsing the source code files or using a tool such as "depcheck." 2 points per unused dependency.	3	Jafari et al. (2021)
	Missing dependency	A dependency that is used in code but not found in package.json.	Analysing the package.json file, as well as parsing the source code files or using a tool such as "depcheck." 2 points per missing dependency.	5	Jafari et al. (2021)
Dependency count	Number of dependencies	The number of dependencies listed in package.json, and the transitive dependencies of those dependencies.	Analysing the package.json file, and the package.json files of each found dependency or using a tool such as "howfat." 2 points per dependency.	3.5	Zimmermann et al. (2019)
Vulnerabilities	Known vulnerabilities	The number and severity of vulnerabilities reported by an audit tool in production mode (ignoring development dependencies).	Running an audit tool such as npm audit --production or snyk test on the target library. 1 point per low, 2 per medium, 3 per high, 4 per critical vulnerability.	4	Musch et al. (2019), Lauinger et al. (2018), Zimmermann et al. (2019), Imtiaz et al. (2021)
	Static code analysis	The issues reported by a static code analysis tool.	Running a static code analysis tool such as snyk code on the target package. 1 point per low, 2 per medium, 3 per high vulnerability.	5	Ntousakis et al. (2021), Rafnsson et al. (2020)
Maintainer activity	Maintainer activity	The last time a commit was made, or the repository was interacted with by a maintainer.	Analysing the maintainers' account activity in the repository. 1 point per 10 weeks of inactivity.	4	Decan et al. (2018), Zimmermann et al. (2019)
Code coverage	Code coverage	The percentage of lines of code covered by the tests of the library, if it has any.	Using the code coverage reported by the repository, or by running the tests using a code coverage tool such as Istanbul.js. 0 points for 100%, 1 for 99-75%, 2 for 74-50%, 3 for 49-25%, 4 for 24-0%.	3	Sun et al. (2021), Fard & Mesbah (2017), Hemmati (2015)
Namespacing	Use of the global namespace	Not encapsulating the package into a single, library-specific object or module.	Analysing the source code of the library. 4 points for using the global namespace.	4	Theisen (2019), Patra et al. (2018)
Licensing	License conflicts	The degree of less permissive the license of the library is compared to the license of the target project.	Using a license issue checker such as Snyk License Compliance Management, or by manually comparing the licenses. "howfat" can be used to find out the licenses of transitive dependencies. 4 points for an incompatible license.	5	Gangadharan et al. (2008), Sojer & Henkel (2011), Gangadharan et al. (2012), Moraes et al. (2021), Qiu et al. (2021)

[3] 1 = not important at all, 2 = not very important, 3 = somewhat important, 4 = very important, 5 = extremely important

6. Model evaluation

In this chapter, the model is evaluated in a developer workshop. The strengths and weaknesses of the model are identified based on the discussions in the workshop, and some suggestions for improvement are presented.

6.1 Evaluation approach

According to a review of Design Science Research articles by Peffers et al. (2012), "Illustrative Scenario" is the second most common method of evaluation for Model-type artifacts in their sample of 21 papers of said artifact type. In their coding, "Illustrative Scenario" is the "application of an artifact to a synthetic or real-world situation aimed at illustrating suitability or utility of the artifact" (Peffers et al., 2012). Since the model is intended to help practitioners decide whether to include third-party libraries, using a real-world situation to illustrate its suitability and utility makes the most sense to me.

The most common method of evaluating Model-type artifacts in the Peffers et al. (2012) review is the "Technical Experiment," which is a performance evaluation of an algorithm implementation designed to evaluate technical performance, rather than real world performance. As the model developed in this thesis is concerned with helping practitioners in their decision making, the real-world performance of the model, that is, its suitability and utility, is more important than its technical performance.

To illustrate the suitability and utility of the model, the model is applied to six different libraries: three libraries that are already in use in TalentRekry, and three libraries that are not used in TalentRekry. The model is applied to already selected libraries to evaluate whether its output is agreeable on libraries that have already been considered fit for inclusion by the developers of TalentRekry. The model is also applied to new libraries to evaluate whether its output would help in determining whether one should include a new library. After applying the model to the libraries, a small-scale workshop is held with developers from FCG to illustrate its utility to practitioners. The number six was thought to be a convenient number of libraries to present in a single workshop session, as it allows for some variation in the model results without being too overwhelming to present in a small-scale workshop.

6.2 Test data

To evaluate the model, six libraries were chosen to be tested: three that are already in use in TalentRekry, and three that were not in use. The three TalentRekry libraries were:

1. Draggable
2. Autosize
3. Bowser

Draggable is a drag and drop library (Shopify, 2018). Autosize is a library that automatically adjusts textarea height to fit text (Moore, 2015). Bowser is a library for detecting what browsers users have (Diaz, 2015).

The three libraries not already in use were:

1. Sortable
2. Dragula
3. Chart.js

Sortable and Dragula are drag and drop libraries (Sortable Contributors, 2019; Bevacqua, 2016), while Chart.js is a charting library (Chart.js Contributors, 2022). To make the tests as realistic as possible, the first two libraries are drag and drop libraries, with the idea that these could potentially replace the existing Draggable library. Chart.js was selected as similar charting functionality has been desired in the past for TalentRekry.

The tests were conducted according to the measuring and scoring defined in this thesis. A full example of applying the model to a library can be found in Appendix C, and a summary of the test results can be found in Table 4.

6.3 The small-scale workshop

The workshop was conducted as a one-hour long Microsoft Teams meeting. The thesis author presented a short background into the topic of the thesis, followed by a deeper dive into the model, including an explanation of each factor, how to measure them as well as their scoring and importance. The presentation concluded with the results from applying the model to the libraries outlined before. This was followed by a discussion session where the participants discussed the model in general as well as how suitable it was for day-to-day developer use and the utility it would provide to developers. The meeting was recorded and transcribed for use as data in this thesis.

6.3.1 Participant demographics

The participants were once again invited to participate through an open Slack message on the same programming-themed channel that was used to share the questionnaire. The invitation let potential participants know that the meeting would be recorded, but that the participants would be anonymized in the thesis. As was the case with the questionnaire, anonymity was used to ensure as low a barrier to participate as possible. Participants were also free to merely observe if they wished to not partake in the discussion but were interested in the topic.

The meeting had a total of 11 participants, excluding the thesis author, but only five took part in the discussion. The demographics for the five active participants can be found in Table 5. Developer 4 only asked some questions about the details of the criteria, so his participation in the discussion will not be apparent in the next section. The remaining six observers will not be elaborated on.

Table 4. The results of applying the model to six libraries.

	Activity	Code coverage	Lockfile	Dependencies	Vulnerabilities	License	Namespace	Total score
The libraries already in use								
Draggable	Latest activity is a commit from 39 weeks ago. (39/10)*4=15,6	Reported code coverage is 77%. 1*3=3	Lockfile is included (yarn.lock). 0	No dependencies. 0	No vulnerabilities. 0	MIT license. 0	Uses ES6 modules. 0	18,6
Autosize	Latest activity is a commit from 7 weeks ago. (7/10)*4=2,8	No tests. 4*3=12	Lockfile is included (package.lock). 0	No dependencies. 0	No vulnerabilities. 0	MIT license. 0	Uses ES6 modules. 0	14,8
Bowser	Latest activity is accepting a pull request 129 weeks ago. (129/10)*4=51,6	Reported code coverage is 91%. 1*3=3	Lockfile is included (package.lock). 0	No dependencies. 0	No vulnerabilities. 0	MIT license. 0	Uses ES6 modules. 0	54,6
The libraries not in use								
Sortable	Latest activity is a commit from 55 weeks ago. (55/10)*4=22	Running the tests instrumented with Istanbul.js returns 100% coverage. 0	Lockfile is included (package.lock). 0	No dependencies. 0	No vulnerabilities. 0	MIT license. 0	Uses ES6 modules. 0	22
Dragula	Latest activity is a commit from 132 weeks ago. (132/10)*4=52,8	Tests do not run. 4*3=12	Lockfile is included (yarn.lock). 0	2 pinned dependencies. 2*2*3=12 2 direct dependencies, 3 transitive dependencies (5 total). 2*5*3,5=35	No vulnerabilities. 0	MIT license. 0	Uses CommonJS modules. 0	111,8
Chart.js	Latest activity is a response from 4 days ago. 0	Reported code coverage is 97%. 1*3=3	Lockfile is included (pnpm-lock.yaml). 0	1 restrictive constraint. 2*3=6 1 direct dependency. 2*3,5=7	No vulnerabilities. 0	MIT license. 0	Uses ES6 modules. 0	18,6

Table 5. Demographics of active workshop participants.

Participant	Gender	Developer type	Development experience
Developer 1	Male	Full-stack developer	5+ years
Developer 2	Male	Full-stack developer	20+ years
Developer 3	Male	Full-stack developer	20+ years
Developer 4	Male	Full-stack developer	5+ years
Developer 5	Male	Front-end developer	10+ years

6.3.2 Workshop discussion

The general feeling about the model was that the issues with third-party dependencies are very central to what is currently being developed at FCG, and that the criteria are "mostly good," according to Developer 2. However, a major concern from Developer 2 was that the maintainer activity criterion might play too big of a role in the final score, and that it could be misleading:

> *"If there is a library that does something small very well, then it could suffer from simply not receiving changes afterwards." (Developer 2)*

A suggestion from Developer 2 to amend this would be to combine other activity metrics, for example the download activity as reported by the npm Registry, to show that even if the library is not regularly changing, it could still be active and of high quality. This is a very good point, as a major factor in the score differentials in the test data was maintainer activity. Other than the maintainer activity criterion, Developer 2 sees this as a sensible model and one that could be implemented into the company dependency inclusion process as something that helps guide developers in their decision making.

Developer 5 commented on the model being a good idea but found that performing such a thorough manual investigation may not be realistic for someone regularly including new dependencies into a project. He thinks that if such a process is not automated somehow it runs the risk of being forgotten, and that inclusion decisions will be made with a "gut feeling." Developer 2 chimed in by saying that testing at least some of the criteria could be automated in a single script with relative ease, providing a lot of added value to the dependency inclusion process, even if not every criterion is automated or even considered straight away.

Developer 1 commented that a scoring mechanism such as this one would have been useful in the past when the team was updating the packaging of the project and going through all its dependencies. The score could have been used as a sort of preliminary investigation on which dependencies need further investigation, potentially saving some time.

Developer 3 raised a question about whether a similar set of criteria existed before this thesis. While the thesis author did not find any scientific literature pertaining to inclusion criteria in third-party dependencies, the Jackson et al. (2011) criteria provide a checklist of things that one can examine about a piece of software. However, the Jackson et al. (2011) criteria is missing many aspects that are important in front-end dependencies, such as security and transitive dependencies. Outside of scientific literature, there are some services out there that provide similar value. Snyk Advisor (*Snyk Open Source Advisor*, n.d.) provides popularity, maintenance, security and community metrics for various open source packages, including npm packages. It also rates each package with its own

"Package Health Score," which is a score from 0 to a 100 based on factors such as security, popularity, maintenance and community activity. Compared to the model developed in this thesis, Advisor is missing information on transitive dependency specifics, such as permissive constraints, being limited to only reporting the number of dependencies that a package has. Advisor also does not take the presence of a lockfile into account, nor does it check the namespacing or code coverage of a package. The npm Registry (*npm*, n.d.b) also provides some key metrics for a package, such as its license, the number of dependencies, latest published release, and how many times a week it is downloaded, but is missing many things from the model developed in this thesis.

During a discussion sparked by a question from Developer 3 regarding whether the points are too low or too high for any specific criterion, Developer 2 mentioned that critical severity vulnerabilities, for example, should probably be so severe that they block inclusion, yet the current model would only give a critical severity vulnerability a score of 16, equivalent to 40 weeks of inactivity. Licensing was also mentioned as a criterion that should probably block inclusion, yet its points may not reflect the severity. One way around this could be to designate these factors as blockers, where rather than giving a score, they would give a "blocker" rating instead. Another suggestion from Developer 2 was to combine the score from vulnerabilities with their CVSS ratings, with a scaling curve as the vulnerability gets more severe. This way critical severity vulnerabilities could get the very high score they deserve while allowing lower severity vulnerabilities to affect the score at a reasonable rate.

6.4 Summary of evaluation

The model was presented as an illustrative scenario where it was applied to six different libraries and the results were examined with a group of developers to see if the model was suitable and usable. The main takeaway from the evaluation workshop was that the model was suitable and somewhat usable, but that it could also be improved.

The purpose of the model was to aid developers in their dependency inclusion decision making process, for which it was considered suitable for. When it comes to suitability, there were two major limitations. The first was that maintainer activity was weighted too heavily, giving smaller packages that do not often update an unfair disadvantage. The second was the lack of a "blocker"-tier rating, which means that an issue that should practically block a developer from including a package receives a score like any other criterion, which could give a developer the wrong idea about the severity of that criterion.

Maintainer activity could be improved by changing the criterion to instead measure overall package activity, including in things like weekly downloads and community activity, which could give a clearer sense of the quality of the package. Alternatively, the maintainer activity score could be more lenient in the number of points it gives, or it could receive some sort of a score ceiling after a certain number of weeks of inactivity has passed.

The lack of a blocker-tier rating could be solved by marking certain criteria as blockers, such as an incompatible license or a critical severity vulnerability. Another way this limitation could be addressed is by including the CVSS rating of a vulnerability in the scoring system, perhaps with a logarithmically scaling score the higher the CVSS rating.

The usability of the model also faced some critique. The model as it stands requires its user to go through a series of manual activities to test a dependency. While the model

gives a clear way to measure each criterion, the fact that one must perform each activity manually could limit its usability in real-world situations where speed may be favoured. This could be solved by automating the testing process. Even if it is not feasible to automate everything, steps like checking the dependency tree, checking the license, checking for missing dependencies and such could be automated easily. Bundling the activities into a single script that one can run against a package would improve the usability of the model by a lot for developers.

7. Discussion

In this chapter, the findings of the study are explained, the developed model is contrasted with an existing set of software evaluation criteria, the limitations of the study are covered, and some recommendations for future research are presented.

7.1 Findings

The primary research question in this thesis was "How can we develop a model for third-party JavaScript library inclusion?"

The research question was split into two sub-questions:

RQ1.1: What factors influence the decisions to include third-party libraries?

RQ1.2: How effective is the model in aiding in the decision making of library inclusion?

The factors that influence the decisions to include third-party libraries were discovered through a review of previous research. They were validated as important factors to consider through a developer survey. These factors can be found in Table 6. In addition to the factors found in literature, one respondent to the developer survey also responded with "accessibility" as a factor to consider. However, due to the low sample size of only one respondent considering it a factor, and with no support for it from literature, the factor was not considered in the final model.

The ways of measuring each factor were based on the literature, with the scoring based on the individual attributes of each factor. For example, a binary factor such as no lockfile was worth a static amount of 4 points, while a non-binary factor such as the number of dependencies was worth 2 points per dependency. The developer survey asked the respondents to rate the importance of each identified factor. These ratings were then used to give the model's factors their importance ratings, which were used as a score multiplier. Each identified factor was considered at least somewhat important by the developers.

Table 6. Factors that influence the decisions to include third-party libraries.

Factor	Source
Pinned dependencies	Jafari et al. (2021)
URL dependencies	Jafari et al. (2021)
Restrictive constraints	Jafari et al. (2021)
Permissive constraints	Jafari et al. (2021)
No lockfile	Jafari et al. (2021)
Unused dependencies	Jafari et al. (2021)
Missing dependencies	Jafari et al. (2021)
High dependency count	Zimmermann et al. (2019)
Known vulnerabilities	Musch et al. (2019), Lauinger et al. (2018), Zimmermann et al. (2019), Imtiaz et al. (2021)
Issues found through static code analysis	Ntousakis et al. (2021), Rafnsson et al. (2020)
Maintainer activity	Decan et al. (2018), Zimmermann et al. (2019)
Use of the global namespace	Theisen (2019), Patra et al. (2018)
Code coverage	Sun et al. (2021), Fard & Mesbah (2017), Hemmati (2015)
License conflicts	Gangadharan et al. (2008), Sojer & Henkel (2011), Gangadharan et al. (2012), Moraes et al. (2021), Qiu et al. (2021)

The effectiveness of the model was evaluated in a developer workshop. In the workshop, the model was found to be useful, but it was also found to have some deficiencies when it came to its effectiveness. The main hinderances with the effectiveness of the model were that applying it requires a lot of manual effort, that the maintainer activity factor was weighted too heavily in the final score, and that the model lacks a “blocker”-tier for factors that should block inclusion if their conditions are fulfilled, such as with critical severity vulnerabilities or license conflicts. Despite these hinderances, the reception towards the model was positive, and some developers felt that it, with some changes, could be included in the software development process as a standard set of criteria to aid in the decision making of including new dependencies.

The answer to the research question of “How can we develop a model for third-party JavaScript library inclusion?” was therefore to create a set of testable criteria based on scientific literature and have it validated by practitioners. The model should include ways to measure each criterion so that it can be used as is without having to perform external research on every individual factor. By including the scoring system, the results of applying the model to a library can be compared with the results of applying it to other libraries, making it easier to see which library requires further investigation. The weights for the scoring system should be based on data from a practitioner survey, as certain criteria are going to be more important than others, and practitioners are likely to know which ones should be prioritised. Care should be taken to ensure that the criteria do not favour specific types of libraries over others, such as by putting too much emphasis on maintainer activity, which would favour more complicated libraries over simple ones that do not receive as many changes. Based on the developer reception towards the resulting model, the use of design science research was a good choice when it comes to developing a model for third-party JavaScript library inclusion. The model developed in this thesis can be found in Appendix B.

7.2 The model contrasted to an existing set of software evaluation criteria

In comparison to the set of software evaluation criteria by Jackson et al. (2011), the criteria developed in this thesis are a better fit for dependency inclusion. The Jackson et al. (2011) criteria do not take security or transitive dependencies into account, both of which are important to consider when including a dependency. Their criteria also lack weights, which makes it difficult to discern which criteria are the most important to consider. Additionally, their criteria consist purely of yes/no questions, meaning that it is hard to get a sense of severity when it comes to each individual criterion. For example, compare “Does the library have dependencies?” versus “How many dependencies does the library have?” The second cannot be answered as a yes/no question, yet it gives us more information to aid in the decision-making process.

The Jackson et al. (2011) criteria contain some important sub-criteria that the model in this thesis does not include. One such criterion is documentation, which can be important to consider when it comes to using a library that one is planning to include. However, documentation is a factor that is difficult to measure. One could measure whether documentation exists, but this alone does not tell us whether the documentation is useful or up to date. This is also why the model does not simply measure whether tests exist, but rather their coverage of the codebase. Like with documentation, the existence of tests does not guarantee that they are useful or up to date, but by measuring coverage, we can at the very least get some insight into the quality of the testing. Reviewing documentation

is a subjective activity and the quality of documentation difficult to measure, so it is not included in this model.

There are some criteria that could be adapted from the Jackson et al. (2011) set that the current model does not incorporate. For instance, their sub-criteria "Portability" and "Interoperability" could be considered. In the current web development landscape browser support is not as big of a concern as it was a few years ago, but there may still be some projects out there that need to support old browsers such as IE11, and for these projects, browser support is still a concern. Additionally, some frontend frameworks may benefit from "wrapper" versions of a library, such as when using the Vue.js framework in conjunction with vanilla JavaScript libraries. Having such a wrapper available may be a benefit for the library, but at the same time, if one simply requires a vanilla JavaScript library, counting the lack of a Vue.js wrapper against the library feels counterproductive. If the model was to be expanded, these could be optional criteria for those use cases where they apply.

7.3 Limitations

The primary limitation of the study is the low sample size in both the developer survey as well as the developer workshop. The survey participants were chosen using convenience sampling, with the survey receiving a participation rate of eight respondents from a potential pool of 20, or in other words, a 40% participation rate. Stratton (2021) recommends a participation rate of 80% to ensure credibility for convenience sampling, which this study did not reach. Additionally, while the developer survey also contained a factor that was important to consider that was not found in literature, due to the small sample size of one response, it could not be used in the model. The turnaround for the workshop was slightly better at 11 participants, or a participation rate of 55%. However, only five of the 11 participants took part in the workshop discussion, potentially limiting the validity of the discussion.

The developer survey was conducted anonymously to achieve as high a response rate as possible, but this also meant that the demographics of the respondents were unknown. Only the demographics of the total pool of potential respondents is known, and since only 40% of the pool responded, we cannot equate the demographics of the entire pool to the respondents. The survey invitation also included the inclusion criterion of "knowledge of front-end dependencies," but due to the anonymity of the respondents, this could not actually be verified. The workshop, on the other hand, was not anonymous, and each developer taking part in the discussion was knowledgeable about front-end dependencies.

As the survey and workshop participants were chosen using convenience sampling, the results from this thesis cannot be generalised to a wider population, and any potential sampling errors cannot be determined. Additionally, the study may have suffered from motivation bias, as participation in the study depended on the motivation of the participants. Dependency management has been a topic that has received some discussion at FCG in the past few years, so those who feel strongly about the topic may have had a stronger representation in the data compared to the average developer.

The survey might have been affected by the central tendency bias, a bias where participants avoid the ends of a scale and tend to respond closer to the middle of the scale (Douven, 2018). The results were heavily focused on the middle rating – "Somewhat important" – which would point towards central tendency bias. Notably, no factor had a median below "Somewhat important," despite several being above it. There is a

possibility that the central tendency bias may have muddied the results in terms of the low importance factors without influencing the very important ones. As a result, some of the "Somewhat important" factors may deserve to be "Not very important" instead, for instance.

The survey was not validated beforehand as the author feared it could lower the potential pool of respondents even further. Validating the survey, however, could have caught some of these potential limitations, which could have then been addressed in the final survey. Should a future study implement a similar survey, the survey should be validated first.

7.4 Recommendations for future research

One respondent to the developer survey mentioned "Accessibility" as factor that should be considered when deciding to include a library. However, the thesis author could not find any existing literature on accessibility in third-party libraries. Studying how accessibility is considered in libraries, perhaps by examining how popular libraries conform to the various levels of WCAG 2.1 requirements, would be a great topic for future research.

The model developed in this thesis was found to have some deficiencies. Future research could build upon the model, taking the problems identified in this thesis into account. Applying the model developed in this thesis also requires a not insignificant amount of effort from developers. Future research could study ways to develop the model into an automated tool or a script that developers could run once to inspect all the important details of a library that are needed to make the inclusion decision.

A major limitation when it comes to the validity of the study was its use of convenience sampling in developing the importance ratings as well as model validation. Repeating those portions of the study on a larger scale using a method of sampling that is more generalisable would address the biggest limitation of the study.

8. Conclusion

This study developed a proof-of-concept model for inclusion criteria for third-party dependencies using the design science research process. The factors to be considered when including a library were collected through reviewing existing literature. These factors were then validated and assigned importance ratings through a developer survey. The model was evaluated by developers in a small-scale workshop, and was found to be "mostly good," though one criterion, maintainer activity, was found to be weighted too high in the model's scoring system. The model could have also benefited from having some sort of a "blocker" tier for factors that are so severe that they should stop one from including a library altogether, such as critical severity vulnerabilities or license incompatibilities. Some developers also found the model too cumbersome to go through every time they wanted to include a library, so automating the tests was found to be a potential requirement for real-life application.

The model developed in this thesis is novel in that a set of testable criteria for dependency inclusion has not yet been developed before. Additionally, this study provides a synthesis of existing scientific literature on the factors to consider when including a library. The contributions are therefore relevant to both practitioners and researchers: practitioners can use the criteria to aid them in the dependency inclusion process, and the literature review contributes to the knowledge of the field of software engineering by synthesising existing knowledge into a cohesive summary.

The largest limiting factor to the study was that it used convenience sampling with a small sample size when it came to collecting data from developers. The current importance ratings, as well as the validity of the model, cannot be generalised to the wider software development community. Future research could entail repeating these portions of the study with sampling that is able to be generalised to a larger population, which would improve the validity of the model.

Another topic to consider for future research is studying accessibility in the context of third-party libraries, as one respondent to the survey considered this important, yet no existing literature support could be found on the subject. Studying ways to automate the application of the model in the form of a tool should also be considered, as the time-consuming manual application of the model was a major point of feedback among developers.

References

Bevacqua, N. (2016). Dragula [Source code]. Retrieved May 2, 2023, from https://github.com/bevacqua/dragula

Bi, T., Xia, X., Lo, D., Grundy, J., Zimmermann, T., & Ford, D. (2022). Accessibility in software practice: A practitioner's perspective. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *31*(4), 1-26.

caniuse. (n.d.) Can I use javascript modules? Retrieved February 22, 2023, from https://caniuse.com/?search=javascript%20modules

Chart.js Contributors. (2022). Chart.js [Source code]. Retrieved May 2, 2023, from https://github.com/chartjs/Chart.js

Chyung, S. Y., Swanson, I., Roberts, K., & Hankinson, A. (2018). Evidence-based survey design: The use of continuous rating scales in surveys. *Performance Improvement*, *57(5)*, 38-48.

Collins, K. (2016, March 27). How one programmer broke the internet by deleting a tiny piece of code. *Quartz*. Retrieved April 4, 2023, from https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code

CVE. (n.d.a). Retrieved August 9, 2022, from https://www.cve.org/

CVE. (n.d.b). Related Efforts. Retrieved August 9, 2022, from https://www.cve.org/About/RelatedEfforts#CVSS

Decan, A., Mens, T., & Constantinou, E. (2018, May). On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories* (pp. 181-191).

DeCastellarnau, A. (2018). A classification of response scale characteristics that affect data quality: a literature review. *Quality & quantity*, *52(4)*, 1523-1559.

Diaz, D. (2015). Bowser [Source code]. Retrieved May 2, 2023, from https://github.com/lancedikson/bowser

Douven, I. (2018). A Bayesian perspective on Likert scales and central tendency. *Psychonomic bulletin & review, 25*, 1203-1211.

Engström, E., Storey, M., Runeson, P., Höst, M. & Baldassarre, M. T. (2020). How software engineering research aligns with design science: a review. *Empirical Software Engineering, 25*, 2630–2660. https://doi.org/10.1007/s10664-020-09818-7

Fard, A. M., & Mesbah, A. (2017, March). JavaScript: The (un) covered parts. In *2017 IEEE international conference on software testing, verification and validation (ICST)* (pp. 230-240). IEEE.

Gangadharan, G. R., D'Andrea, V., De Paoli, S., & Weiss, M. (2012). Managing license compliance in free and open source software development. *Information Systems Frontiers*, *14*, 143-154.

Gangadharan, G. R., De Paoli, S., D'Andrea, V., & Weiss, M. (2008). License compliance issues in free and open source software. *MCIS 2008 Proceedings*, *2*.

Hemmati, H. (2015, August). How effective are code coverage criteria?. In *2015 IEEE International Conference on Software Quality, Reliability and Security* (pp. 151-156). IEEE.

Imtiaz, N., Thorn, S., & Williams, L. (2021, October). A comparative study of vulnerability reporting by software composition analysis tools. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 1-11).

ISO. (n.d.) ISO/IEC 9126-1:2001 Software engineering — Product quality — Part 1: Quality model. Retrieved May 9, 2023, from https://www.iso.org/standard/22749.html

Istanbul.js. (2015). nyc [Source code]. Retrieved April 6, 2022, from https://github.com/istanbuljs/nyc

Istanbul.js. (n.d.). Retrieved April 6, 2022, from https://istanbul.js.org/

Jackson, M., Crouch, S., & Baxter, R. (2011). Software evaluation: criteria-based assessment. *Software Sustainability Institute, 1*.

Jafari, A. J., Costa, D. E., Abdalkareem, R., Shihab, E., & Tsantalis, N. (2021). Dependency smells in Javascript projects. *IEEE Transactions on Software Engineering.*

Kaplan, B., & Qian, J. (2021, August). A Survey on Common Threats in npm and PyPi Registries. In *International Workshop on Deployable Machine Learning for Security Defense* (pp. 132-156). Springer, Cham.

Kikas, R., Gousios, G., Dumas, M., & Pfahl, D. (2017, May). Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* (pp. 102-112). IEEE.

Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., & Kirda, E. (2018). Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. *arXiv preprint arXiv:1811.00918.*

Microsoft. (2022). Internet Explorer 11 desktop application ended support for certain operating systems. Retrieved February 22, 2023, from https://learn.microsoft.com/en-us/lifecycle/announcements/internet-explorer-11-end-of-support

Moore, J. (2015). Autosize [Source code]. Retrieved May 2, 2023, from https://github.com/jackmoore/autosize

Moraes, J. P., Polato, I., Wiese, I., Saraiva, F., & Pinto, G. (2021). From one to hundreds: multi-licensing in the JavaScript ecosystem. *Empirical Software Engineering*, *26*(3), 1-29.

Musch, M., Steffens, M., Roth, S., Stock, B., & Johns, M. (2019, July). Scriptprotect: mitigating unsafe third-party javascript practices. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security* (pp. 391-402).

Nakhaei, K., Ansari, F., & Ansari, E. (2020). JSSignature: eliminating third-party-hosted JavaScript infection threats using digital signatures. *SN Applied Sciences*, *2*(1), 1-11.

Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., ... & Vigna, G. (2012, October). You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security* (pp. 736-747).

npm Docs. (n.d.). package-lock.json [Documentation]. Retrieved May 3, 2022, from https://docs.npmjs.com/cli/v8/configuring-npm/package-lock-json

npm, Inc. (n.d.). Retrieved May 10, 2022, from https://www.npmjs.com/

npm. (n.d.a). semver [Source code]. Retrieved April 27, 2022, from https://github.com/npm/node-semver

npm. (n.d.b). npm. Retrieved May 9, 2023, from https://www.npmjs.com/

Ntousakis, G., Ioannidis, S., & Vasilakis, N. (2021, November). Detecting Third-Party Library Problems with Combined Program Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (pp. 2429-2431).

Paltoglou, A., Zafeiris, V. E., Giakoumakis, E. A., & Diamantidis, N. A. (2018, March). Automated refactoring of client-side JavaScript code to ES6 modules. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 402-412). IEEE.

Patra, J., Dixit, P. N., & Pradel, M. (2018, May). Conflictjs: finding and understanding conflicts between javascript libraries. In *Proceedings of the 40th International Conference on Software Engineering* (pp. 741-751).

Peffers, K., Rothenberger, M., Tuunanen, T., & Vaezi, R. (2012). Design science research evaluation. In *Design Science Research in Information Systems. Advances in Theory and Practice: 7th International Conference, DESRIST 2012, Las Vegas, NV, USA, May 14-15, 2012. Proceedings 7* (pp. 398-410). Springer Berlin Heidelberg.

Peffers, K., Tuunanen, T., Gengler, C. E., Rossi, M., Hui, W., Virtanen, V., & Bragge, J. (2006). The design science research process: Amodel for producing and presenting information systems research. In *First International Conference on Design Science Research in Information Systems and Technology* (pp. 83-16).

pnpm. (n.d.) .npmrc. Retrieved February 7, 2023, from https://pnpm.io/npmrc

Qiu, S., German, D. M., & Inoue, K. (2021). Empirical Study on Dependency-related License Violation in the JavaScript Package Ecosystem. *Journal of Information Processing*, *29*, 296-304.

Rafi, D. M., Moses, K. R. K., Petersen, K., & Mäntylä, M. V. (2012, June). Benefits and limitations of automated software testing: Systematic literature review and

practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)* (pp. 36-42). IEEE.

Rafnsson, W., Giustolisi, R., Kragerup, M., & Høyrup, M. (2020, November). Fixing Vulnerabilities Automatically with Linters. In *International Conference on Network and System Security* (pp. 224-244). Springer, Cham.

Runeson, P., Engström, E. & Storey, M. (2020). The Design Science Paradigm as a Frame for Empirical Software Engineering. In M. Felderer & G. H. Travassos (Eds.), *Contemporary Empirical Methods in Software Engineering,* 127-147. Switzerland: Springer Nature Switzerland AG.

Schaeffer, N. C., & Dykema, J. (2020). Advances in the science of asking questions. *Annual Review of Sociology*, *46*, 37-60.

SemVer. (n.d.). Semantic Versioning 2.0.0. Retrieved April 27, 2022, from https://semver.org/

Shopify. (2018). Draggable [Source code]. Retrieved April 6, 2022, from https://github.com/Shopify/draggable

Snyk Open Source Advisor. (n.d.). Snyk. Retrieved May 9, 2023, from https://snyk.io/advisor/

Snyk. (n.d.). What is CVE? Keeping Track of Vulnerabilities with CVE Vulnerability Database. Retrieved August 9, 2022, from https://snyk.io/learn/what-is-cve-vulnerablity/

Sojer, M., & Henkel, J. (2011). License risks from ad hoc reuse of code from the internet. *Communications of the ACM*, *54(12)*, 74-81.

Sortable Contributors. (2019). Sortable [Source code]. Retrieved May 2, 2023, from https://github.com/SortableJS/Sortable

Stol, K. & Fitzgerald, B. (2020). Guidelines for Conducting Software Engineering Research. In M. Felderer & G. H. Travassos (Eds.), *Contemporary Empirical Methods in Software Engineering,* 27-62. Switzerland: Springer Nature Switzerland AG.

Stratton, S. J. (2021). Population research: convenience sampling strategies. *Prehospital and disaster Medicine*, *36(4)*, 373-374.

Sun, H., Rosà, A., Bonetta, D., & Binder, W. (2021, May). Automatically Assessing and Extending Code Coverage for NPM Packages. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)* (pp. 40-49). IEEE.

Theisen, K. J. (2019). Programming languages in chemistry: a review of HTML5/JavaScript. *Journal of Cheminformatics*, *11*(1), 11.

W3C WAI. (2018). WCAG 2.1 at a Glance. Retrieved January 31, 2023, from https://www.w3.org/WAI/standards-guidelines/wcag/glance/

W3C WAI. (2022). Introduction to Web Accessibility. Retrieved January 31, 2023, from https://www.w3.org/WAI/fundamentals/accessibility-intro/

W3C WAI. (2023). WCAG 2 Overview. Retrieved January 31, 2023, from https://www.w3.org/WAI/standards-guidelines/wcag/

Yarn. (n.d.) Questions & Answers. Retrieved February 7, 2023, from https://yarnpkg.com/getting-started/

Zimmermann, M., Staicu, C. A., Tenny, C., & Pradel, M. (2019). Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)* (pp. 995-1010).

Appendix A. The developer questionnaire

Factors to consider when including a third-party JavaScript library

Mandatory questions are marked with a star (*)

Transitive dependencies

These are factors that relate to the dependencies of the library. In other words, the dependencies that we would transitively add to our project should we decide to include the library. These assume that packages are following npm's standard versioning scheme SemVer 2.0.0 specification, which is formatted as MAJOR.MINOR.PATCH. According to the specification, the major version should be incremented when incompatible API changes are made, the minor version when backwards compatible functionality is added, and the patch version when backwards compatible bug fixes are added.

1. Pinned dependency *

A dependency in the library's package.json that is restricted to only one version.
E.g. a version number of:
1.2.3
=1.2.3

	Not important at all	Not very important	Somewhat important	Very important	Extremely important
How important is it to consider this factor when deciding on including a library?	O	O	O	O	O

Reasoning or comment (optional)

2. URL dependency *

A dependency in the library's package.json that points to a URL or a local path instead of specifying a version number.
E.g.
https://registry.npmjs.org/foo/-/foo-1.0.0.tgz
../foo/bar

	Not important at all	Not very important	Somewhat important	Very important	Extremely important
How important is it to consider this factor when deciding on including a library?	O	O	O	O	O

Reasoning or comment (optional)

3. Restrictive constraint *

A dependency in the library's package.json that does not permit backward compatible updates, but allows patches (bug fixes).
E.g. a version number or version range of:
>=1.2.0 <1.3.0
~1.2.3

	Not important at all	Not very important	Somewhat important	Very important	Extremely important
How important is it to consider this factor when deciding on including a library?	O	O	O	O	O

Reasoning or comment (optional)

4. Permissive constraint *

A dependency in the library's package.json that permits all (including breaking) changes.
E.g. a version number or version range of:
>=1.2.0
*

	Not important at all	Not very important	Somewhat important	Very important	Extremely important
How important is it to consider this factor when deciding on including a library?	O	O	O	O	O

Reasoning or comment (optional)

5. No lockfile *

The library is missing its lockfile.
E.g. package-lock.json or yarn.lock

	Not important at all	Not very important	Somewhat important	Very important	Extremely important
How important is it to consider this factor when deciding on including a library?	O	O	O	O	O

Reasoning or comment (optional)

6. Unused dependency *

The library lists a dependency in its package.json that is not used.

	Not important at all	Not very important	Somewhat important	Very important	Extremely important
How important is it to consider this factor when deciding on including a library?	O	O	O	O	O

Reasoning or comment (optional)

7. Missing dependency *

The library uses a dependency in code that is not in its package.json.

	Not important at all	Not very important	Somewhat important	Very important	Extremely important
How important is it to consider this factor when deciding on including a library?	O	O	O	O	O

Reasoning or comment (optional)

8. Number of dependencies *

The number of dependencies that the library has. This includes transitive dependencies.

	Not important at all	Not very important	Somewhat important	Very important	Extremely important
How important is it to consider this factor when deciding on including a library?	O	O	O	O	O

Reasoning or comment (optional)

Security

These are factors that relate to the security of the library.

9. Known vulnerabilities *

The number of known vulnerabilities in the library, as reported by e.g. npm audit.

	Not important at all	Not very important	Somewhat important	Very important	Extremely important
How important is it to consider this factor when deciding on including a library?	O	O	O	O	O

Reasoning or comment (optional)

10. Source code security *

The number of known security issues found in the library through static code analysis, e.g. by running snyk code on the library.

	Not important at all	Not very important	Somewhat important	Very important	Extremely important
How important is it to consider this factor when deciding on including a library?	O	O	O	O	O

Reasoning or comment (optional)

11. Maintainer activity *

The activity of the maintainers of the library.
E.g. how frequently the repository gets new commits, how often the maintainers respond to issues and pull requests.

	Not important at all	Not very important	Somewhat important	Very important	Extremely important
How important is it to consider this factor when deciding on including a library?	O	O	O	O	O

Reasoning or comment (optional)

Quality

These are factors that relate to the quality of the library.

12. Use of the global namespace *

Not encapsulating the library into a single, library-specific object.

	Not important at all	Not very important	Somewhat important	Very important	Extremely important
How important is it to consider this factor when deciding on including a library?	O	O	O	O	O

Reasoning or comment (optional)

13. Code coverage *

The percentage of lines of code covered by the tests of the library.

	Not important at all	Not very important	Somewhat important	Very important	Extremely important
How important is it to consider this factor when deciding on including a library?	O	O	O	O	O

Reasoning or comment (optional)

Other

Factors that did not fit the previous categories.

14. License conflicts *

The degree of less permissive the license of the library is compared to the license of the project that is going to use the library.
E.g. using a library with a copyleft license (such as GPL) in a proprietary software project.

	Not important at all	Not very important	Somewhat important	Very important	Extremely important
How important is it to consider this factor when deciding on including a library?	O	O	O	O	O

Reasoning or comment (optional)

15. Are there any factors that you think are missing from this survey?

Factor

Why should it be considered?

	Not important at all	Not very important	Somewhat important	Very important	Extremely important
How important is it to consider this factor when deciding on including a library?	O	O	O	O	O

Appendix B. The full model for dependency inclusion criteria

Inclusion Criteria for Including a Third-party JavaScript Library

Library: -

Final score: -

Verdict: -

Criterion: Dependency constraints

Factor: Pinned dependency

Explanation	Measuring	Scoring	Score multiplier	Findings
A dependency that is restricted to only one version.	Analysing the package.json file.	2 points for each pinned dependency found.	3 (somewhat important)	-

Score: -

Criterion: Dependency constraints

Factor: URL dependency

Explanation	Measuring	Scoring	Score multiplier	Findings
A dependency that points to a URL (or local path).	Analysing the package.json file.	2 points for each URL dependency found.	3 (somewhat important)	-

Score: -

Criterion: Dependency constraints

Factor: Restrictive constraint

Explanation	Measuring	Scoring	Score multiplier	Findings
A dependency that does not permit backward compatible updates.	Analysing the package.json file.	2 points for each restrictive constraint found.	3 (somewhat important)	-

Score: -

Criterion: Dependency constraints

Factor: Permissive constraint

Explanation	Measuring	Scoring	Score multiplier	Findings
A dependency that permits breaking changes.	Analysing the package.json file.	2 points for each permissive constraint found.	5 (extremely important)	-

Score: -

Criterion: Dependency definitions

Factor: No lockfile

Explanation	Measuring	Scoring	Score multiplier	Findings
The package repository is missing its lockfile.	Analysing the repository file structure.	4 points if the library repository is missing its lockfile.	4 (very important)	-

Score: -

Criterion: Dependency definitions

Factor: Unused dependency

Explanation	Measuring	Scoring	Score multiplier	Findings
A dependency that is listed in package.json but not used.	Analysing the package.json file, as well as parsing the source code files for usages of listed dependencies or using a tool such as “depcheck.”.	2 points for each unused dependency found.	3 (somewhat important)	-

Score: -

Criterion: Dependency definitions

Factor: Missing dependency

Explanation	Measuring	Scoring	Score multiplier	Findings
A dependency that is used in code but not found in package.json.	Analysing the package.json file, as well as parsing the source code files for usages of listed dependencies or using a tool such as “depcheck.”	2 points for each missing dependency found.	5 (extremely important)	-

Score: -

Criterion: Dependency count

Factor: Number of dependencies

Explanation	Measuring	Scoring	Score multiplier	Findings
The number of dependencies listed in package.json, and the transitive dependencies of those dependencies.	Analysing the package.json file, and the package.json files of each found dependency or using a tool such as “howfat.”	2 points for each dependency found.	3.5 (somewhat to very important)	-

Score: -

Criterion: Vulnerabilities

Factor: Known vulnerabilities

Explanation	Measuring	Scoring	Score multiplier	Findings
The number and severity of vulnerabilities reported by an audit tool in production mode (ignoring development dependencies).	Running an audit tool such as npm audit --production or snyk test on the target library.	Points depend on the severity of the discovered vulnerabilities: 1 (low) 2 (medium) 3 (high) 4 (critical)	4 (very important)	-

Score: -

Criterion: Vulnerabilities

Factor: Static code analysis

Explanation	Measuring	Scoring	Score multiplier	Findings
The issues reported by a static code analysis tool.	Running a static code analysis tool such as snyk code on the target package.	Points depend on the severity of the discovered vulnerabilities: 1 (low) 2 (medium) 3 (high) For tools other than snyk code, a different scoring mechanism may be required.	5 (extremely important)	-

Score: -

Criterion: Maintainer activity

Factor: Maintainer activity

Explanation	Measuring	Scoring	Score multiplier	Findings
The last time a commit was made, or the repository was interacted with by a maintainer.	Analysing the maintainers' account activity in the repository.	1 point for every 10 weeks of inactivity	4 (very important)	-

Score: -

Criterion: Code coverage

Factor: Code coverage

Explanation	Measuring	Scoring	Score multiplier	Findings
The percentage of lines of code covered by the tests of the library, if it has any.	Using the code coverage reported by the repository, or by running the tests using a code coverage tool such as Istanbul.js.	0 points for 100% coverage, 1 point for 99-75% coverage, 2 points for 74-50% coverage, 3 points for 49-25% coverage, 4 points for 24-0% coverage.	3 (somewhat important)	-

Score: -

Criterion: Namespacing

Factor: Use of the global namespace

Explanation	Measuring	Scoring	Score multiplier	Findings
Not encapsulating the package into a single, library-specific object or module.	Analysing the source code of the library.	4 points if found to be using the global namespace.	4 (very important)	-

Score: -

Criterion: Licensing

Factor: License conflicts

Explanation	Measuring	Scoring	Score multiplier	Findings
The degree of less permissive the license of the library is compared to the license of the target project.	Using a license issue checker such as Snyk License Compliance Management, or by manually comparing the licenses.	4 points if the library is found to be using an incompatible license.	5 (extremely important)	-

Score: -

Appendix C. An example of applying the model to a library

Inclusion Criteria for Including a Third-party JavaScript Library

Library: Draggable (https://github.com/Shopify/draggable)

Final score: 18,6

Verdict: The final score of 18,6 is low enough for the library to be considered fit for inclusion.

Criterion: Dependency constraints

Factor: Pinned dependency

Explanation	Measuring	Scoring	Score multiplier	Findings
A dependency that is restricted to only one version.	Analysing the package.json file.	2 points for each pinned dependency found.	3 (somewhat important)	No pinned dependencies. Score is 0.

Score: 0

Criterion: Dependency constraints

Factor: URL dependency

Explanation	Measuring	Scoring	Score multiplier	Findings
A dependency that points to a URL (or local path).	Analysing the package.json file.	2 points for each URL dependency found.	3 (somewhat important)	No URL dependencies. Score is 0.

Score: 0

Criterion: Dependency constraints

Factor: Restrictive constraint

Explanation	Measuring	Scoring	Score multiplier	Findings
A dependency that does not permit backward compatible updates.	Analysing the package.json file.	2 points for each restrictive constraint found.	3 (somewhat important)	No restrictive constraints. Score is 0.

Score: 0

Criterion: Dependency constraints

Factor: Permissive constraint

Explanation	Measuring	Scoring	Score multiplier	Findings
A dependency that permits breaking changes.	Analysing the package.json file.	2 points for each permissive constraint found.	5 (extremely important)	No permissive constraints. Score is 0.

Score: 0

Criterion: Dependency definitions

Factor: No lockfile

Explanation	Measuring	Scoring	Score multiplier	Findings
The package repository is missing its lockfile.	Analysing the repository file structure.	4 points if the library repository is missing its lockfile.	4 (very important)	Repository contains yarn.lock. Score is 0.

Score: 0

Criterion: Dependency definitions

Factor: Unused dependency

Explanation	Measuring	Scoring	Score multiplier	Findings
A dependency that is listed in package.json but not used.	Analysing the package.json file, as well as parsing the source code files for usages of listed dependencies or using a tool such as “depcheck.”	2 points for each unused dependency found.	3 (somewhat important)	No unused dependencies. Score is 0.

Score: 0

Criterion: Dependency definitions

Factor: Missing dependency

Explanation	Measuring	Scoring	Score multiplier	Findings
A dependency that is used in code but not found in package.json.	Analysing the package.json file, as well as parsing the source code files for usages of listed dependencies or using a tool such as “depcheck.”	2 points for each missing dependency found.	5 (extremely important)	No missing dependencies. Score is 0.

Score: 0

Criterion: Dependency count

Factor: Number of dependencies

Explanation	Measuring	Scoring	Score multiplier	Findings
The number of dependencies listed in package.json, and the transitive dependencies of those dependencies.	Analysing the package.json file, and the package.json files of each found dependency or using a tool such as “howfat.”	2 points for each dependency found.	3.5 (somewhat to very important)	No dependencies. Score is 0.

Score: 0

Criterion: Vulnerabilities

Factor: Known vulnerabilities

Explanation	Measuring	Scoring	Score multiplier	Findings
The number and severity of vulnerabilities reported by an audit tool in production mode (ignoring development dependencies).	Running an audit tool such as npm audit --production or snyk test on the target library.	Points depend on the severity of the discovered vulnerabilities: 1 (low) 2 (medium) 3 (high) 4 (critical)	4 (very important)	npm audit --production: No known production vulnerabilities. Score is 0.

Score: 0

Criterion: Vulnerabilities

Factor: Static code analysis

Explanation	Measuring	Scoring	Score multiplier	Findings
The issues reported by a static code analysis tool.	Running a static code analysis tool such as snyk code on the target package.	Points depend on the severity of the discovered vulnerabilities: 1 (low) 2 (medium) 3 (high) For tools other than snyk code, a different scoring mechanism may be required.	5 (extremely important)	snyk code: No issues were found. Score is 0.

Score: 0

Criterion: Maintainer activity

Factor: Maintainer activity

Explanation	Measuring	Scoring	Score multiplier	Findings
The last time a commit was made, or the repository was interacted with by a maintainer.	Analysing the maintainers' account activity in the repository.	1 point for every 10 weeks of inactivity	4 (very important)	Latest activity is a commit from 39 weeks ago. Score is (39 / 10) * 4 = 15,6.

Score: 15,6

Criterion: Code coverage

Factor: Code coverage

Explanation	Measuring	Scoring	Score multiplier	Findings
The percentage of lines of code covered by the tests of the library, if it has any.	Using the code coverage reported by the repository, or by running the tests using a code coverage tool such as Istanbul.js.	0 points for 100% coverage, 1 point for 99-75% coverage, 2 points for 74-50% coverage, 3 points for 49-25% coverage, 4 points for 24-0% coverage.	3 (somewhat important)	Reported code coverage is 77%. Score is 1 * 3 = 3.

Score: 3

Criterion: Namespacing

Factor: Use of the global namespace

Explanation	Measuring	Scoring	Score multiplier	Findings
Not encapsulating the package into a single, library-specific object or module.	Analysing the source code of the library.	4 points if found to be using the global namespace.	4 (very important)	The library uses ES6 modules. Score is 0.

Score: 0

Criterion: Licensing

Factor: License conflicts

Explanation	Measuring	Scoring	Score multiplier	Findings
The degree of less permissive the license of the library is compared to the license of the target project.	Using a license issue checker such as Snyk License Compliance Management, or by manually comparing the licenses.	4 points if the library is found to be using an incompatible license.	5 (extremely important)	The library uses the MIT license, which is compatible with a proprietary project. Score is 0.

Score: 0