Augmenting Library Development by Mining Usage Data from Downstream Dependencies

Christoph Thiede, Willy Scheibel, Daniel Limberger, and Jürgen Döllner
Hasso Plattner Institute, Digital Engineering Faculty, University of Potsdam
christoph.thiede@student.hpi.uni-potsdam.de, {willy.scheibel, daniel.limberger, juergen.doellner}@hpi.uni-potsdam.de

Keywords: Mining Software Repositories, Downstream Dependencies, API Usage.

Abstract: Software and its dependencies build up a graph where edges connect packages according to their dependencies. In this graph, downstream dependencies are all the nodes that depend on a package of interest. Although gathering and mining such downstream dependencies allow for informed decision-making for a package developer, there is much room for improvement, such as automation and integration of this approach into their development process. This paper makes two contributions: (i) We propose an approach for efficiently gathering downstream dependencies of a single package and extracting usage samples from them using a static type analyzer. (ii) We present a tool that allows npm package developers to survey the aggregated usage data directly in their IDE in an interactive and context-sensitive way. We evaluate the approach and the tool on a selection of open source projects and specific development-related questions with respect to found downstream dependencies, gathering speed, and required storage. Our methods return over 8 000 downstream dependencies for popular packages and process about 12 dependencies per minute. The usage sample extraction offers high precision for basic use cases. The main limitations are the exclusion of unpopular and closed-source downstream dependencies as well as failing analysis when encountering complex build configurations or metaprogramming patterns. To summarize, we show that the tool supports package developers in gathering usage data and surveying the use of their packages and APIs within their downstream dependencies, allowing for informed decision-making and improving the resdesign of APIs.

1 Introduction

The continued proliferation and improvement of open-source software (OSS) platforms such as *GitHub* or *GitLab* and package deployment ecosystems such as *npm*, *pip*, or *NuGet* support and facilitate software development on a global scale. Open-source development offers many advantages over traditional closed-source development, including voluntary contributions from the open-source community, greater transparency in security-related areas, and a high potential for solution reuse (Saied et al., 2018). These solutions are typically organized as packages, each solving an isolated problem and providing a higher-level interface. Packages can *depend* on existing packages by including them in their manifest, i.e., the `package.json` file of npm packages or the `requirements.txt` file of Python projects. These dependency relations form a large directed graph connecting significant parts of the software world for today's programming language ecosystems. We define a software ecosystem as a collection of software projects, which are developed and which co-evolve in the same environment.

Despite this connectedness by design, however, the development process of many packages is still characterized by an isolated approach: While OSS developers commonly submit tickets to or contribute patches against *upstream repositories* that they depend on to solve subproblems, the reverse direction of these edges – called *downstream dependencies* and in the following referred to as *dependencies* – is often neglected by package developers. This deficit can cause a wide range of alignment issues, including poorly suited interfaces (Piccioni et al., 2013), unidentified defects (Wong et al., 2017), and compatibility problems (Bogart et al., 2015). Eventually, all these issues impair the capabilities of the global OSS community to build and support high-quality packages.

To tackle these concerns, we propose an approach to extract API usage samples from downstream-dependency repositories of individual packages to support package developers in exploring usages of their packages. We make these data directly available to package developers by integrating them into an inte-

grated development environment (IDE).

Our approach consists of three steps: (i) collect downstream dependency repositories from public source code repositories, (ii) mine package usage samples from these dependencies using a type analyzer, and (iii) aggregate and present these usage data to the package developer in its development environment. The approach is implemented as a Visual Studio Code extension[1] and evaluated on a selection of open source projects and specific development-centric questions.

The remainder of this paper is structured as follows: In section 2, we summarize existing approaches and in section 3, we outline the overall conditions for our solution and detail the collection of downstream dependency repositories and subsequent mining of usage data. In section 4, we present an implementation of our approach and outline design decisions for displaying usage samples to the user. Finally, in section 5, we examine the fitness of our data mining approach and the usability of the display solution. Section 6 concludes this paper.

2 Related Work

Proper tooling design for improving package developers' knowledge about interface usages in dependencies has not yet attracted significant attention in the scientific community. Nevertheless, programmatic analysis of dependency graphs and API usage mining are already standard techniques in the field of *mining software repositories* (MSR). Chaturvedi et al. provide a broad overview of existing achievements and ongoing research topics in this field (Chaturvedi et al., 2013). Besides source code repositories, they describe data sources worthwhile to examine, including telemetry data from IDEs, issue trackers, and discussion platforms, and propose different directions for evaluating the retrieved data, e.g., classifying or ranking repositories, analyzing the evolution of projects, studying development communities, but also inspecting the relationships and dependencies between projects. We further consider related work in the areas of dependency graphs, usage sample extraction, and tooling support.

Searching dependency graphs. A common purpose for *analyzing dependency graphs* is to discover transitive upstream dependencies of a project and assess their impact on the stability and vulnerability of the project (Kikas et al., 2017). Similarly, the spreading of security vulnerabilities along the chain of downstream dependencies can be measured (Decan et al., 2018).

However, existing research on dependency graphs takes a broad statistical perspective and relies on a large corpus of downloaded software repositories (Abdalkareem et al., 2017; Kikas et al., 2017; Katz, 2020). As opposed to this approach, these capacities are not suitable for scenarios that developers of a single package occasionally perform. In this case, developers will often use a public *code search* service instead, such as *GitHub* or *Sourcegraph*. Liu et al. provide a survey of methods and trends in code search tools that include newer approaches such as structural or semantic search queries, code similarity metrics, or machine learning methods (Liu et al., 2020).

Extracting usage samples. Dependency graphs do not provide information about package usage at sufficient granularity. For example, insights about referencing package classes or methods are missing. The process of extracting fine-grained information about all references to individual elements of an interface is referred to as *API usage analysis* (Lämmel et al., 2011). A simple approach is to perform a string search for package names or members in the dependencies (Mileva et al., 2010), or to operate on the *abstract syntax tree* (AST) of each parsed dependency to avoid false positive caused by ambivalent identifier names (Qiu et al., 2016). In addition, one can collect usage data to analyze the historical importance of certain features supported by an API (Sawant and Bacchelli, 2017).

However, the precision of ASTs is limited for *dynamically typed* languages such as JavaScript or Python. As opposed to static typing, the value type of an expression in a dynamically typed language may be unknown at the compile time of the program. For this reason, a *type analysis* can help to predict runtime types of all expressions in the AST (Jensen et al., 2009). This is usually accomplished by analyzing the control flow of a program and inferring possible types of every variable or function. Another source for enhancing type information is *call graphs* that are either gathered *statically*, i.e., as a result of a theoretical analysis of the source code, or *dynamically*, i.e., sampled during actual program execution. While the latter method is better suited to include more contextual information, the former method has the advantage that it can be applied to a more considerable amount of source code without requiring concrete entry points, parameterization, and run-time data. Many solutions exist that analyze the structure of programs and extract relevant information to build call graphs:

Collard et al. propose an infrastructure called

[1] All presented artifacts are available on GitHub: https://github.com/LinqLover/downstream-repository-mining (Thiede et al., 2022)

SRCML that aims to create a unified representation of multilingual source code snippets for arbitrary analysis purposes, including the construction of call graphs (Collard et al., 2013). Another solution utilizes an island parser to build polyglot call graphs (Bogar. et al., 2018). Furthermore, Antal et al. compare several call graph generators for JavaScript and emphasize that static call graph generators have limited precision by design for languages supporting dynamic typing or meta-programming mechanisms (Antal et al., 2018).

To merge dependency graphs and call graphs, ecosystem call graphs can be constructed by applying call graphs to entire ecosystems that cross repository boundaries. Many approaches apply this concept to different ecosystems, often aiming to conduct fine-grained impact analysis for security vulnerabilities (Hejderup et al., 2018; Boldi and Gousios, 2020; Wang et al., 2020; Hejderup et al., 2021; Keshani, 2021; Nielsen et al., 2021). Another approach is the *precise code intelligence* feature of Sourcegraph, which makes code references explorable across repository boundaries for repositories that provide a metadata file in the *Language Server Index Format*.[2]

After collecting these usage data, additional processing is possible to extract general usage information from the extensive raw data. Next to basic grouping and counting operations, such aggregations can be built by detecting popular *usage patterns* of API features in the downstream dependencies. Zhong et al. propose such a framework, that, for instance, finds sequences of API members that are invoked frequently and even uses these patterns for guiding API users by giving them recommendations (Zhong et al., 2009). Hanam et al. pursue a different goal in their tool that helps package developers assess the impact of breaking API changes on the functionality of downstream dependencies (Hanam et al., 2019).

Presentation of results. To establish an adequate developer experience, any collected usage data still need to be presented suitably. Due to the hierarchical structure of source code, a common approach for this is a hierarchical and navigatable representation of the collected call tree. An early form of this representation has been invented as the interactive "message set" tool for browsing senders and implementors in an object-oriented system for the Smalltalk programming environment (Goldberg, 1984). Modern alternatives include the *Stacksplorer* (Karrer et al., 2011) or *Blaze* (Krämer et al., 2012) that list next to the currently focused method other methods which are adjacent to this method in the call graph. With a focus on exploring references to API members, De Roover et al. propose a set of additional views, including hierarchical lists and word clouds for highly referenced identifiers (De Roover et al., 2013). Focusing on exploring APIs in a non-code-centric way, Hora and Valente propose a dashboard that allows developers to browse existing APIs and track possibly breaking changes in their interfaces (Hora and Valente, 2015).

[2]https://docs.sourcegraph.com/code_intelligence/explanations/precise_code_intelligence

3 Downstream Dependency Mining

Our approach to improve developers' knowledge about the usage of their packages is based on the automatic gathering of downstream dependency information and integration into their development environment. This approach assumes a target package and focuses on showing a developer actual usage of parts of their package and, in particular, answering the following questions: In doing so, we identified three questions from package developers about the usage of their packages:

Q1 Which dependencies are using the target package, and what problems do they try to solve with them?

Q2 By how many dependencies is a particular package member being used?

Q3 In which contexts and constellations is a particular package member being used?

For an effective integration into the development process, we pose three non-functional requirements:

R1 The tool is ready to use out of the box and requires little configuration (less than 5 minutes for a new user).

R2 The tool consumes sufficiently few computational resources to run on a single machine but still delivers interactive response times according to Shneiderman's requirement to frequent tasks (Shneiderman and Plaisant, 2010).

R3 The tool blends in with the usual workflow of package developers at the best possible rate.

On a conceptual level, we separate the approach into the collection of downstream dependency projects and mining of usage samples based on a target package.

3.1 Downstream Dependency Collection

In the first step, a list of repositories that depend on the target package has to be assembled. As mentioned in section 2, many approaches start with a large downloaded corpus of unspecific repositories from the ecosystem, which they then iterate over to filter the

```
Input:
    package: target package
    dependencies: downstream dependencies
Output: usage samples (set of strings)

for dependency ∈ dependencies:
    asf ← parse(dependency ∪ package)
    annotate_types(asf)
    for ast ∈ asf:
        for node ∈ dfs(ast):
            for pattern ∈ patterns:
                if pattern.matches(node) ∧ package.declares(pattern.getType(node)):
                    yield node.text
```

Algorithm 1: Extraction of usage samples.

relevant repositories. However, a major drawback of this approach is the high resource demand for both creating and traversing this corpus. This approach is suited for analyzing repositories on a large scale but is not conform to requirement R1 because of its large footprint in terms of computational power, memory, and time. As an alternative, we have decided to apply filtering to the set of repositories before downloading a subset of it to the developers' machine. For that, we use two types of data sources that are available as public web platforms:

(i) Package repositories that maintain a doubly-connected edge list of interdependent packages in an ecosystem.

(ii) Code search engines that index the source code of many repositories from OSS platforms. Using these search engines, we can query all repositories that declare a dependency on the target package in their package manifest file.

3.2 Usage Sample Mining

Having downloaded the selected downstream dependency repositories, the next step is to extract usage samples for the target package from each dependency repository. Our goal is to gather fine-grained source code excerpts that reference individual identifiers exposed by the target package. The complete procedure is displayed in algorithm 1.

To do so, the source code of every dependency repository as well as the source code of the target package are parsed, each into a separate *abstract syntax forest* (ASF, i.e., a set of ASTs). In a second step, static type analysis against each dependency ASF is performed together with the target package's ASF. The results of the type analysis are attached to the ASF so that every identifier is annotated with a type symbol that links to the declaration of the type for the identifier. For instance, after this step, every variable node contains a link to the assignment node of this variable, and every function call expression contains a link to the definition of this function. The operating principle of this type analysis depends on the kind of programming language. In statically typed languages, type symbols can be retrieved from declarations, whereas in dynamically typed languages, a control flow analysis will be required to identify the origin of every identifier's type.

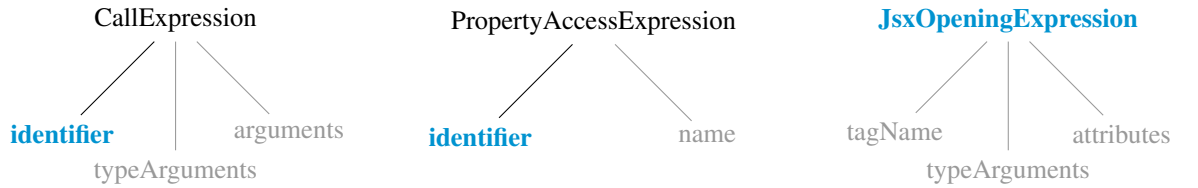
In the final step, all nodes whose type is declared in the target package are collected from each ASF. To identify these links, we use a set of language-specific patterns for AST subtrees that constitute a usage expression. In particular, each of these patterns expects a node containing a link to an identifier declaration (see fig. 1).

4 Prototype

In our prototype, we focus on analyzing JavaScript projects for node.js. This decision is motivated by the large prevalence of JavaScript[3] as well as the first-rate extent of the npm ecosystem which has been unsurpassed since 2015[4]. The prototype instantiates the downstream dependency collection and the usage sample mining and allows for the integration of the results into the development environment.

[3] GitHut 2.0: A small place to discover languages in GitHub. https://madnight.github.io/githut/#/pushes

[4] Modulecounts: http://www.modulecounts.com/



(a) Node pattern for a TypeScript function call, such as in: `result = fun<T1, T2>(arg1, arg2);`

(b) Node pattern for a JavaScript property access, such as in: `return obj.prop;`

(c) Node pattern for a JSX opening element (as supported in React or TypeScript), such as in: or TypeScript `elem = <Button color="blue">Submit</Button>;`

Figure 1: AST patterns for example JavaScript/TypeScript expressions. The **highlighted** node contains the link to the declaration of the referenced identifier.

Listing 1: Example `package.json` file for the `ripple-lib` package specifying multiple dependencies (truncated). The list of dependencies that are extracted from this example contains `bignumber.js`, `https-proxy-agent`, and `jsonschema`.

```
{
  "name": "ripple-lib",
  "version": "1.10.0",
  "description": "An API for the XRP
    Ledger",
  "dependencies": {
    "bignumber.js": "^9.0.0",
    "https-proxy-agent": "^5.0.0",
    "jsonschema": "1.2.2"
  }
}
```

4.1 Downstream Dependency Collection

The de-facto standard package repository manager for node.js is npm. To fetch downstream dependency packages from the npm registry, we use the npm package `npm-dependants` which scrapes the dependents list from npmjs.com as there does not exist a publicly available API for these data. To keep the footprint of our implementation small, only a part of the dependents list is scraped. Then, basic metadata for each found package is requested from the npm registry, including the *tarball* URL of the latest package version, using the package `package-metadata`. Finally, each tarball is downloaded and extracted using the package `download-package-tarball`.

For the second method – based on a code search engine –, we have identified Sourcegraph[5] as a promising platform. npm packages specify their dependencies in a `package.json` metadata file (see listing 1). To find all packages depending on a certain package, we can search all available `package.json` files for a reference to this package identifier. We refine our search query with some additional constraints to filter out package copies from `node_modules` folders that downstream developers have pushed accidentally, and to limit the number of results retrieved. We submit this search query to the publicly available GraphQL API of Sourcegraph[6] by using the npm package `graphql-request`. After retrieving a search result, we download a snapshot of the full repository by using the package `download-git-repo`.

Additionally, we enrich the metadata of each package with a few metrics indicating its popularity from the associated GitHub repository if specified. The GitHub API is accessed via the official *octokit* library[7].

4.2 Usage Sample Mining

For parsing the dependencies, we use the *TypeScript Compiler API*[8]. TypeScript is an extension of JavaScript that adds strong typing; typing is based on a combination of implicit type inference and explicit type declarations. As a side benefit, our implementation is also able to analyze packages implemented in true TypeScript.

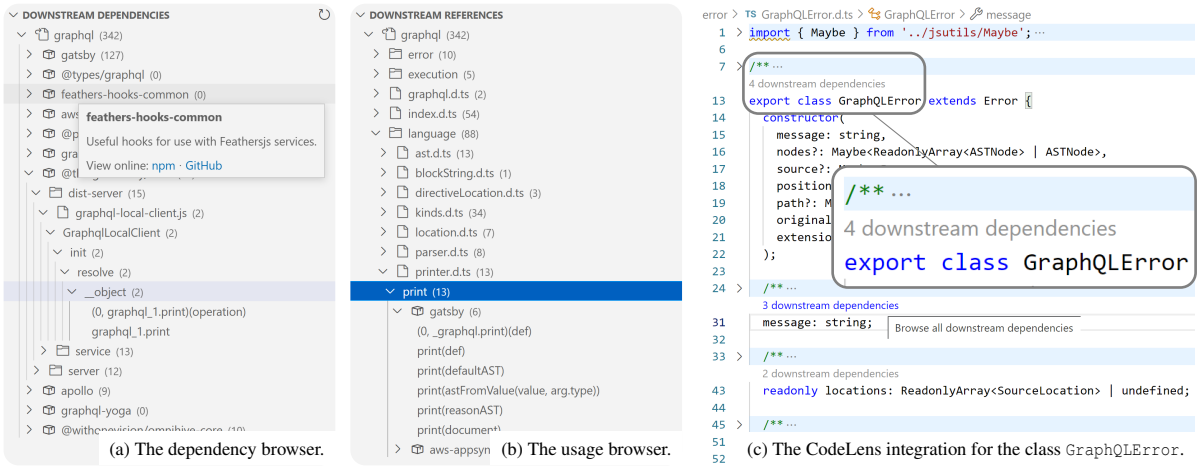
The TypeScript compiler's *binder* component annotates every node of the parsed program with a type symbol if it can resolve one. After performing a type analysis of the dependency by instrumenting the binder, we collect all nodes from the type-annotated ASF that refer to the target package, matching each node against a list of defined patterns (similar to fig. 1). A lookup of the node's type symbol's declaration is performed based on the syntax kinds of the node, its ancestors, and its descendants, and then compared with the source directory of the target package.

[5]https://sourcegraph.com/

[6]https://docs.sourcegraph.com/api/graphql

[7]https://octokit.github.io/

[8]https://github.com/Microsoft/TypeScript/wiki/Using-the-Compiler-API



(a) The dependency browser. (b) The usage browser. (c) The CodeLens integration for the class `GraphQLError`.

Figure 2: Screenshots of our VS Code extension used to explore downstream dependencies of the npm package `graphql`.

Normally, foreign types can only be resolved if they are declared in a dependency module that has been installed into the `node_modules` folder of the repository. Because downloading and installing all dependency modules for every downstream repository of the target package would increase the computational resources of the usage mining drastically and eventually violate requirement R2, we skip this step but instead modify the module resolution strategy of the TypeScript compiler host. To do so, we extend the `paths` parameter from the default compiler options[9] and insert a new entry that remaps accesses to the target package name to the separate source code directory of the target package.

4.3 Presentation of Results

After all references have been collected, a proper user interface (UI) is still required to provide users easy access to these data that fulfills the requirements mentioned above. To satisfy requirement R1 and requirement R3 by making our tool available in the usual working environment of users, we implement it as an extension to the *Visual Studio Code* IDE[10]. It also provides a comprehensive set of APIs for extension developers. To support other researchers in reusing our solution, we also provide a CLI. The user interface is divided into three key views, supporting developers in answering the questions raised in section 3 (see fig. 2):

(i) The *dependency browser* allows to explore all downstream dependencies and, grouped for each dependency, all references to the target package.

(ii) The *usage browser* displays all public package members and, grouped for each member, all dependencies and their references to this member.

(iii) The *CodeLens integration* provides quick access to a slice of usage samples and is attached to the definition of each package member in the source code editor.

Both references and members are organized each in a tree view that reflects the hierarchical structure of the original software repository. In addition, we recognize the effort of browsing large lists of dependency data and encounter it by providing an "I'm feeling lucky" button for every view that redirects the user to a random dependency or reference, respectively, to gain a faster, unbiased impression of usage samples.

To implement each of these views in the VS Code Extension API (VSCE)[11], we use the *Tree View API* and the *CodeLens API*. One challenge has been to deliver fast results and not to block the user interface because even despite the lightweight mining methods we chose to satisfy requirement R2 from above, querying, downloading, and analyzing each dependency takes a few seconds (see section 5). To overcome this issue, we use pagination wherever possible and push incremental UI updates. As the VSCE API per se does not provide for multithreading or multiprocessing operations but suggests a Promise/A+-driven dataflow, we adopt this style for our object model by using the JavaScript concepts asynchronous functions, `AsyncIterator`s, and `Promise.all` to avoid busy waiting during the data collection process.

[9] https://www.typescriptlang.org/tsconfig#paths
[10] https://code.visualstudio.com/
[11] https://code.visualstudio.com/api

Table 1: Quantity and false-positive rates (FPR) of downstream dependencies found by the presented methods (using npm and Sourcegraph) for selected packages.

		npm		Sourcegraph		
Package	GitHub stars	Count	FPR	Count	FPR	Intersection in %
base64id	16	27	0.20	45	1.00	8
nemo	38	1	0.00	1	1.00	0
random-js	556	219	0.14	193	0.36	15
kubernetes-client	902	36	0.13	79	0.21	16
jsonschema	1 547	394	0.00	517	0.18	2
graphql	18 005	396	0.17	8 863	0.68	2
cheerio	24 228	396	0.07	6 779	0.07	0

5 Evaluation

To evaluate our approach, we formulate three research questions:

RQ1 What is the quality and quantity of the proposed methods for dependency collection?

RQ2 What is the quality and quantity of the proposed method for mining usage samples?

RQ3 How well is the proposed tool applicable with regard to the questions and requirements described in section 3?

In the following, we will investigate each question.

5.1 RQ1: Dependency collection

To assess the quality and quantity of dependency collection, we analyze it with respect to extent, precision, recall, and performance. Table 1 shows the number of dependencies collected to build an experimental dataset from each data source following the proposed methods for a set of manually selected npm packages. For each package, we have annotated a subset of the collected dependencies (max. 20 dependencies per package) to identify and classify false positive hits.

Extent. Regarding the total number of dependencies collected by each method, no clear trend in favor of any method can be ascertained for small or medium packages, i.e., having less than 10,000 stars on GitHub. For larger packages, i.e., having at least 10,000 stars, however, the number of dependencies collected from the npm registry stagnates near to 400 hits. Beyond this limit, the npm registry returned internal server errors for our requests. For Sourcegraph, we have not hit any limitations so far. Currently, they do not provide any official documentation for the exact rate limits. As the disjunct proportion of results from both methods is very small (on average about 6 %), we consider a combination of both data sources useful for maximizing the extent and diversity of the gained dataset.

Precision. The precision of collected dependencies is drastically lower for dependencies found on Sourcegraph than for such found on npm. Figure 3 collates the causes we have identified for false positives; most frequently, repositories specify a dependency on a target package in their package manifest file but do not import this package at any place. In some situations, this may happen if the package is a plugin for another package or if it is invoked as a CLI from a build script, but in the majority of repositories, developers trivially appear to have specified the upstream dependency by accident (e.g., while copying a `package.json` file over from another package), or to have forgotten to remove the upstream dependency after switching away from using the package. Less common causes are repositories that only add type definitions to a package but do not actually import it, as well as packages required by a repository that declares a peer dependency on the target package, which needs to be fulfilled by the depending repository. We explain the increased false-positive rate for Sourcegraph dependencies by our observation that packages found on npm are deliberately published and typically stand out by their higher cohesion and commitment to maintenance, whereas GitHub-only projects contain a higher share of hobby or students' projects. We stress that a reduced precision does not impair the quality of results displayed to package developers as all dependencies not containing at least

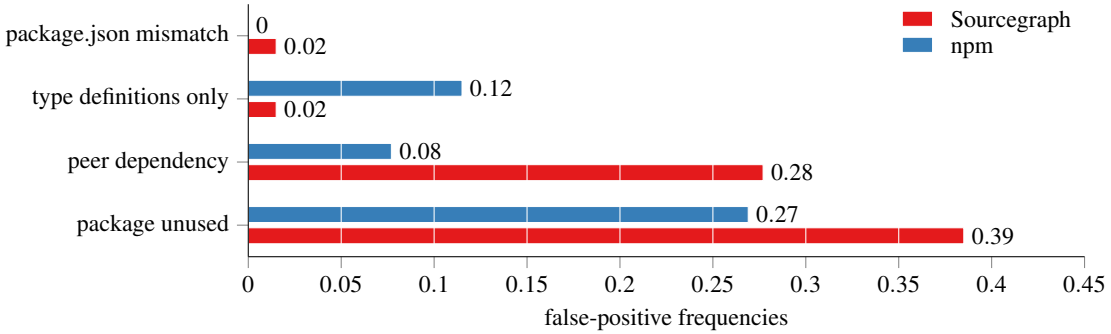


Figure 3: Causes for false-positive dependency matches and their frequencies.

one usage sample can be easily filtered out, but downloading and analyzing any irrelevant packages lowers the performance of the approach.

Recall. Besides the precision of the collected dependencies, their recall is of interest, too. Since dependencies are collected in excerpts from two large datasets, a quantitative analysis of false negatives would require costly manual annotation and is target for future work. Nevertheless, some causes will prevent a dependency from being found by our methods:

- Packages without a proper dependency manifest cannot be detected as our method is based on parsing these metadata.
- Only published packages are found on npm, leading to a bias for generic and professional software.
- On both platforms, results will be sorted based on intransparent criteria (which, as we speculate, include the number of direct dependents on npm and the recent update frequency on Sourcegraph, respectively). As we only fetch the first dependencies from both data sources, this is a likely source of further biases. These biases could be fought by always fetching all dependencies, but this would drastically reduce performance and would require a workaround for the npm query limitation.

Performance. Table 2 gives some basic metrics comparing the performance of both data sources. While searching downstream dependencies is slower by average on npm as we use a web scraper instead of an API for this source, npm packages are usually smaller than many mono-repositories on OSS platforms that contain multiple node.js packages. In addition, they are faster to download since the `download-git-repo` package that we use for repositories found on Sourcegraph only uses zipballs instead of tarballs (`.tar.gz` files), while the latter would offer a higher compression rate.

5.2 RQ2: Usage mining

To assess the quality and quantity of usage mining, we examine this processing step's precision, recall, and performance. Due to the complexity of a detailed annotation process, we only consider the existence of found usage samples on the dependency level. The absolute quantity of found usage samples largely varies for different packages and dependencies based on the semantic extent of the package and the coupling.

Precision. Under laboratory conditions, false-positive usage samples cannot be emitted by our method if we assume the correctness of the TypeScript compiler. There are only two theoretical exceptions to this invariant: (i) In case of a name collision between two packages, invalid or false positive usage samples may be emitted. However, the npm infrastructure attempts to rule out these collisions by using the package name as a unique ID for every published package. (ii) If dependency developers deliberately interfere with the TypeScript compiler (for instance, by suppressing type errors using `@ts-ignore` comments, both false positive and false negative matches are possible. Additionally, some npm packages contain minified sources only that are still searchable, valid code but can be significantly harder to read.

Recall. To evaluate the recall of our method, we have refined the experimental dataset collected in section 5.1 and removed all false-positive dependencies. Within this cleansed dataset, the overall rate of dependencies for that our method cannot find at least one usage sample accounts for 47.9 %. To explain these false negatives, we have identified some systematic causes:

- Many projects on OSS platforms have complex build configurations that include additional transpilation or code generation steps before a final version of the source code is reached that is valid

Table 2: Performance metrics and remarks for both dependency collection methods using npm and Sourcegraph.

Metric		npm	Sourcegraph
Search speed [a]	s/pkg	1.58	0.04
Download speed [a,b]	s/pkg	0.26	8.80
Storage	MB/pkg	5.80	27.20
API limitations		max. 400 results	none known

[a] Test machine: 7 vCPUs Intel Xeon Cascade Lake at 2.80 GHz, internet down speed 1.8 Gbit/s.

[b] Effective speed downloading multiple packages in parallel to manage latencies.

to the node.js interpreter or the TypeScript compiler. Unless we add explicit support for such build configurations, type analysis and usage mining for these projects will fail.

- In some situations, additional type definitions are required to perform a complete type inference of a dependency. This applies to every parametrized callback from a package for that type definitions are not available. This limitation could be resolved by downloading or generating type definitions for all upstream dependencies of every downstream dependency; however, this would reduce the approach's performance.
- The static type analyzer of TypeScript has some limitations. For instance, the type analysis will have a limited recall for certain control flow patterns or metaprogramming constructs such as meta-circular evaluation using the `eval()` function or dynamic function binding using `Function.bind()`. These limitations could be resolved with AST preprocessing and partial evaluation of the source code.

Performance. The total time spent performing type analysis for a repository from the experimental dataset and mining usage samples averages about 3 seconds. The complete ASF of an average parsed repository consumes between 10 MB and 500 MB memory, depending on the size of the repository.

5.3 RQ3: Requirements

To assess the applicability of our tool, we investigate different options for users who would like to answer the questions raised in section 3. We further analyze how far our tool meets the requirements posed above.

Answering User Questions. To answer question Q1 without our tool, users could access the data sources used by our tool manually to search for and view downstream dependencies. The source code of repositories can be browsed on Sourcegraph, but if an npm package is not available on an OSS platform, users will need to download and extract it. With our tool, the data collection process is condensed into a single button that incrementally displays all references from both data sources. Users can hover or click any dependency to view its documentation or implementation. To answer questions Q2 and Q3 without our tool, users need to scan each found dependency separately to count or read all usage samples. If they intend to analyze the usage of a member with an exact name, they can perform an expeditious string search. This is possible using Sourcegraph or another search engine if the dependency is indexed there; otherwise, users will need to download the repository and search it locally. If the member of interest has an ambiguous name, users will need to download the project and view it in an IDE such as VS Code that supports reference search. Alternatively, they can try the precise code intelligence mode on Sourcegraph (see section 2); however, this feature is available for a few repositories only. All usage samples can be collected automatically and merged into a single list within our tool. Once the collected dependencies have been processed, users can select a member of interest and view all its usage samples with a single click. For the usage analysis of Java APIs, the tool *Exapus* also considers different kinds of member usages such as instantiation vs. inheritance of a class (De Roover et al., 2013); a similar classification could improve the usability of our tool.

Meeting Requirements. To satisfy requirement R1, the application needs to be easy to set up. As our extension can be installed with two clicks from the Visual Studio Code Extension Marketplace and the installation takes less than 10 seconds on our test machines, it fulfills this requirement.

We break down requirement R2 into two acceptance criteria: application liveness and small resource footprint. As discussed before, our tool can process 5 up to 12 dependencies per minute while requiring about 500 MB memory in total and less than 30 MB storage per package. After the tool is activated, the first results usually appear after less than 10 seconds

on the UI. This is an acceptable delay regarding Shneiderman's definition of acceptable application response times, so the liveness criterion is fulfilled. Even if the tool downloads a few hundred dependencies, it will occupy less than 10 GB storage that can be released at any time, which we consider a small resource footprint at a time where modern operating systems require 64 GB storage; thus, the second criterion is fulfilled as well.

Requirement R3 effectively calls for a minimum number of context switches that are required to use the tool, i.e., for minimizing the temporal, spatial, and semantic distance (Ungar et al., 1997) between the downstream dependency UI and the IDE used by package developers. Once the data are available in the UI, interaction is possible with the same speed as with local source files in the package. This meets Shneiderman's requirement to frequent tasks and indicates a low temporal distance. The spatial distance varies for the different views of the tool: A separate toolbar has to be opened for the dependency and usage browsers, which indicates a higher spatial distance; still, the artifacts are available in the same IDE. Also, the dependency artifacts have no strong relation to any existing IDE artifacts, so any closer integration into an existing view would increase coupling between independent domains. The code annotations are displayed close to the package source code with a small spatial distance. The semantic distance can be described as the perceived similarity of artifacts displayed both in the tool's UI and in the existing IDE, i.e., the package and dependency members. Most items in the usage browser bear the same label as the corresponding identifier in the source code, which reduces the semantic distance. However, dependency member path nodes that refer to anonymous expressions are displayed differently compared to the outline view of VS Code (see fig. 2 (a)), slightly increasing the semantic distance. With a low temporal distance, a low-to-medium spatial distance, and a mainly low semantic distance, our tool also fulfills requirement R3.

6 Conclusions and Future Work

Downstream dependencies offer a promising perspective for package developers interested in understanding how their interfaces are used in practice by other software developers. In this paper, we have proposed an automated approach to making this sort of information accessible to package developers by automating the dependency collection and mining of usage samples from every dependency. We have identified two kinds of rich data sources for efficiently collecting downstream dependencies: public package repositories, such as npm, and online code search engines for finding dependent repositories by their package manifest files, namely Sourcegraph. We accomplish the usage mining by scanning the ASTs of every dependency for certain usage patterns that refer to a type from the target package; if the dependency uses a dynamically typed language, we perform a type analysis before. We have demonstrated the usability of our approach with a tool that embeds the downstream dependency data into the VS Code IDE. Our analysis suggests that the collected dependencies and the extracted usage samples have a viable quality in terms of precision and extent and further that the resource requirements of our prototype are sufficiently low for collecting all data on the local machine of a package developer.

Nevertheless, we have identified several causes for false negatives over the whole mining process that can bias the output usage samples, including the popularity and recency of dependencies but also the proper declaration of packages and their renunciation of complex toolchains and metaprogramming patterns. Our analysis, however, is not yet fully supported by data and would require further quantitative evaluation to assess the exact precision and recall of our approach for mining usage samples as well as the efficacy of our tool. As for the first question, manual annotation of source code repositories with usage samples will be required to compare them to the outputs of our implementation. Close insights for the second question could be gained from a user study that measures the impact of our tool on the efficiency of package developers solving downstream dependency-related tasks.

We believe that our tool has further potential to support package developers in surveying downstream dependencies. As future work, we envision other purposes for analyzing the collected downstream dependency data, including usage pattern mining, automated convenience protocol suggestions, and the generation of metrics for classifying downstream dependencies or measuring the cohesion and impact of libraries within an ecosystem. By extending the static usage analysis with a dynamic approach, the detected usage samples could be enriched with valuable runtime data describing involved parameters, the fine-grained code coverage of package members, or the invocation context of members. Returning to the abstract problem that we have formulated in the introduction – how can we improve developer knowledge about their packages' usage? –, further data sources next to source code files emerge that could be mined for package references, too. For instance, we are looking forward to solutions that mine the change history of projects for package-related changes (potentially indicating break-

ing changes), search conversation platforms such as public issue trackers or Q&A forums for mentions of package members and analyze their sentiments, e.g., potentially indicating confusing behavior or bad documentation), or even scan continuous integration logs for failure stack traces (e.g., potentially revealing bugs in the package of interest.

ACKNOWLEDGEMENTS

We want to thank the anonymous reviewers for their valuable comments and suggestions to improve this article. This work is part of the "Software-DNA" project, which is funded by the European Regional Development Fund (ERDF or EFRE in German) and the State of Brandenburg (ILB). This work is also part of the KMU project "KnowhowAnalyzer" (Förderkennzeichen 01IS20088B), which is funded by the German Ministry for Education and Research (Bundesministerium für Bildung und Forschung).

REFERENCES

Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., and Shihab, E. (2017). Why do developers use trivial packages? an empirical case study on npm. In *Proc. 11th Joint Meeting on Foundations of Software Engineering*, FSE '17, pages 385–395. ACM.

Antal, G., Hegedus, P., Tóth, Z., Ferenc, R., and Gyimóthy, T. (2018). Static javascript call graphs: a comparative study. In *Proc. 18th International Working Conference on Source Code Analysis and Manipulation*, SCAM '18, pages 177–186. IEEE.

Bogar., A. M., Lyons., D. M., and Baird., D. (2018). Lightweight call-graph construction for multilingual software analysis. In *Proc. 13th International Conference on Software Technologies*, ICSOFT '18, pages 328–337. INSTICC, SciTePress.

Bogart, C., Kästner, C., and Herbsleb, J. (2015). When it breaks, it breaks: how ecosystem developers reason about the stability of dependencies. In *Proc. International Conference on Automated Software Engineering Workshop*, ASEW '15, pages 86–89. IEEE.

Boldi, P. and Gousios, G. (2020). Fine-grained network analysis for modern software ecosystems. *ACM Transactions on Internet Technology*, 21(1):1–14.

Chaturvedi, K. K., Sing, V., and Singh, P. (2013). Tools in mining software repositories. In *Proc. 13th International Conference on Computational Science and Its Applications*, ICCSA '13, pages 89–98. IEEE.

Collard, M. L., Decker, M. J., and Maletic, J. I. (2013). srcML: an infrastructure for the exploration, analysis, and manipulation of source code: a tool demonstration. In *Proc. International Conference on Software Maintenance*, ICSM '13, pages 516–519. IEEE.

De Roover, C., Lämmel, R., and Pek, E. (2013). Multi-dimensional exploration of api usage. In *Proc. 21st International Conference on Program Comprehension*, ICPC '13, pages 152–161. IEEE.

Decan, A., Mens, T., and Constantinou, E. (2018). On the impact of security vulnerabilities in the npm package dependency network. In *Proc. 15th International Conference on Mining Software Repositories*, MSR '18, pages 181–191. ACM.

Goldberg, A. (1984). *Smalltalk-80: the interactive programming environment*, chapter 10, pages 196–201. Addison-Wesley Longman Publishing Co., Inc.

Hanam, Q., Mesbah, A., and Holmes, R. (2019). Aiding code change understanding with semantic change impact analysis. In *Proc. International Conference on Software Maintenance and Evolution*, ICSME '19, pages 202–212. IEEE.

Hejderup, J., Beller, M., Triantafyllou, K., and Gousios, G. (2021). Präzi: from package-based to call-based dependency networks. *arXiv preprint arXiv:2101.09563*.

Hejderup, J., van Deursen, A., and Gousios, G. (2018). Software ecosystem call graph for dependency management. In *Proc. 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results*, ICSE '18 – NIER Track, pages 101–104. IEEE.

Hora, A. and Valente, M. T. (2015). apiwave: keeping track of api popularity and migration. In *Proc, International Conference on Software Maintenance and Evolution*, ICSME '15, pages 321–323. IEEE.

Jensen, S. H., Møller, A., and Thiemann, P. (2009). Type analysis for javascript. In *Proc. International Static Analysis Symposium*, SAS '09, pages 238–255. Springer.

Karrer, T., Krämer, J.-P., Diehl, J., Hartmann, B., and Borchers, J. (2011). Stacksplorer: call graph navigation helps increasing code maintenance efficiency. In *Proc. 24th Symposium on User Interface Software and Technology*, UIST '11, pages 217–224.

Katz, J. (2020). Libraries.io open source repository and dependency metadata.

Keshani, M. (2021). Scalable call graph constructor for maven. In *Proc. 43rd International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '21, pages 99–101. IEEE.

Kikas, R., Gousios, G., Dumas, M., and Pfahl, D. (2017). Structure and evolution of package dependency networks. In *Proc. 14th International Conference on Mining Software Repositories*, MSR '17, pages 102–112. IEEE.

Krämer, J.-P., Kurz, J., Karrer, T., and Borchers, J. (2012). Blaze: supporting two-phased call graph navigation in source code. In *Proc. Human Factors in Computing Systems – Extended Abstracts*, CHI '12, pages 2195–2200. ACM.

Lämmel, R., Pek, E., and Starek, J. (2011). Large-scale, ast-based api-usage analysis of open-source java projects. In *Proc. Symposium on Applied Computing*, SAC '11, pages 1317–1324. ACM.

Liu, C., Xia, X., Lo, D., Gao, C., Yang, X., and Grundy, J. (2020). Opportunities and challenges in code search tools. *arXiv preprint arXiv:2011.02297*.

Mileva, Y. M., Dallmeier, V., and Zeller, A. (2010). Mining api popularity. In *Proc. 5th International Academic and Industrial Conference on Testing – Practice and Research Techniques*, TAIC PART '10, pages 173–180. Springer, Springer-Verlag.

Nielsen, B. B., Torp, M. T., and Møller, A. (2021). Modular call graph construction for security scanning of node.js applications. In *Proc. 30th SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '21, pages 29–41. ACM.

Piccioni, M., Furia, C. A., and Meyer, B. (2013). An empirical study of api usability. In *Proc. International Symposium on Empirical Software Engineering and Measurement*, ESEM '13, pages 5–14. IEEE.

Qiu, D., Li, B., and Leung, H. (2016). Understanding the api usage in java. *Information and Software Technology*, 73:81–100.

Saied, M. A., Ouni, A., Sahraoui, H., Kula, R. G., Inoue, K., and Lo, D. (2018). Improving reusability of software libraries through usage pattern mining. *Journal of Systems and Software*, 145:164–179.

Sawant, A. A. and Bacchelli, A. (2017). fine-GRAPE: fine-grained api usage extractor – an approach and dataset to investigate api usage. *Empirical Software Engineering*, 22(3):1348–1371.

Shneiderman, B. and Plaisant, C. (2010). *Designing the user interface: strategies for effective human-computer interaction*. Pearson Education India.

Thiede, C., Scheibel, W., Limberger, D., and Döllner, J. (2022). dowdep: Software Mining of Downstream Dependency Repositories. DOI: 10.5281/zenodo.6338060.

Ungar, D., Lieberman, H., and Fry, C. (1997). Debugging and the experience of immediacy. *Comm. ACM*, 40(4):38–43.

Wang, Y., Chen, B., Huang, K., Shi, B., Xu, C., Peng, X., Wu, Y., and Liu, Y. (2020). An empirical study of usages, updates and risks of third-party libraries in java projects. In *Proc. International Conference on Software Maintenance and Evolution*, ICSME '20, pages 35–45. IEEE.

Wong, W. E., Li, X., and Laplante, P. A. (2017). Be more familiar with our enemies and pave the way forward: a review of the roles bugs played in software failures. *Journal of Systems and Software*, 133:68–94.

Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. (2009). Mapo: mining and recommending api usage patterns. In *Proc. European Conference on Object-Oriented Programming*, ECOOP '09, pages 318–343. Springer.