Bachelor Thesis

Evaluation of Dependency Blossoms on the Maven Central Repository

Marvin Jütte

December 1, 2025

Reviewer:
JProf. Dr.-Ing. Ben Hermann
Prof. Dr. Falk Maria Howar

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl V - Programmiersysteme
Fachgruppe Softwaretechnik sicherer Systeme
https://sse.cs.tu-dortmund.de

Abstract

Reusing code is a common practice by leveraging dependencies, libraries, or frameworks. Yet, there is little to no research targeting framework detection on the Maven Central Repository. One proposed way to detect those frameworks are dependency blossoms consisting of artifacts sharing a common group and the same version. Our goal is to evaluate this definition by providing our own, less strict definition of dependency blossoms. Our approach is based on the assumption that artifacts belonging to the same framework will frequently be published within a small and shared time window. We further propose a pipeline on how to compile the blossoms and implemented parts of the pipeline. In order to test our implementation, we ran it on 10,000 randomly selected releases provided by the Goblin framework dataset. We find that 75% of dependency blossoms, using our approach, consist of a minimum of 1 and up to 6 artifact groups for certainty thresholds of 0.8 or higher. This certainty threshold indicates how certain we are that the given artifact is part of the blossom. Thus, our results indicate that the restriction of dependency blossoms to only consist of artifacts that share a common group and the same version is too strict.

Zusammenfassung

Die Wiederverwendung von Code ist eine etablierte Praxis, die meist durch Einbindung von Artefakten, Bibliotheken oder Frameworks erfolgt. Es gibt jedoch kaum bis gar keine Forschung, die sich mit der Erkennung von Frameworks im Maven Central Repository befasst. Ein in der Forschung vorgeschlagener Weg zur Erkennung von Frameworks sind Dependency Blossoms, wobei davon ausgegangen wird, dass diese aus Artefakten bestehen, die Teil einer gemeinsamen Gruppe sind und die gleiche Version haben. Unser Ziel ist es, diese Definition zu evaluieren, indem wir eine eigene, weniger strikte Definition von Dependency Blossoms aufstellen. Diese basiert auf der Annahme, dass Artefakte, die zum selben Framework gehören, häufig innerhalb eines gemeinsamen und engen Zeitfensters veröffentlicht werden. Wir schlagen außerdem eine Pipeline vor, mittels derer wir die Dependency Blossoms ermitteln können. Teile dieser Pipeline implementieren und evaluieren wir in dieser Arbeit. Zum Testen haben wir unsere Implementierung 10.000 zufällig ausgewählte Releases des Goblin Framework Datensatzes auswerten lassen. Als Ergebnis konnten wir sehen, dass 75% der Dependency Blossoms, die mittels unserer angepassten Definition ermittelt wurden, aus mindestens 1 und maximal 6 Artefaktgruppen bestehen, bei einem Gewissheitsgrad von 0,8 oder höher. Das Gewissheitslevel gibt an, wie sicher wir uns in der Annahme sind, dass einzelne Artefakte Teil des Dependency Blossoms sind. Unsere Ergebnisse zeigen, dass die Beschränkung von Dependency Blossoms auf Artefakte, einer gemeinsamen Gruppe und mit der gleichen Version, zu strikt ist.

Contents

1. Introduction

1.1. Motivation

In modern software development, the usage of (external) third-party components or libraries has become a regularly used practice. It is used in commercial as well as in open-source development. Advantages of using third-party libraries are usually reducing development costs and improving the overall quality [40]. However, sometimes those third-party dependencies can introduce security threats (e.g. Log4Shell or XZ Utils) [12, 14, 21]. To avoid being targeted by those threats, developers need to update their dependencies regularly. Package managers like Maven, npm, or Cargo exist to help manage and update dependencies for software projects. Further tools like DependaBot [10], RenovateBot [34], or Snyk [36] exist to automate the process of updating the dependencies even more by automatically creating pull requests containing the newest software dependency version. Despite those tools, many developers are still hesitant to update to avoid regressions or other unwanted side effects. Those side effects include problems like difficult framework upgrades or breaking changes like upgrading the framework core without updating other dependencies, working only with a specific framework version [29]. Existing work by Dann et al. [9] provides an approach to create updates while trying to ensure minimal incompatibilities. They identified issues with frameworks or other highly interconnected dependencies. In order to achieve that, they provided a simple definition of a dependency blossom, being all dependencies that share the same group and version.

In this work we want to provide a broader approach with a different definition of those dependency blossoms. We do that by lifting the strict same group or version restriction to allow for external dependencies that are not part of the framework but also depend on it to be part of the blossom. To achieve that, we planned a pipeline to compile the new blossoms for a dependency repository. We then compare our approach with the simpler definition provided by Dann et al. [9] by showing the average amounts per group with our blossom definition based on 10,000 dependency releases from Maven Central Repository.

1.2. Thesis Structure

This first chapter provides an introduction to our work in the context of dependencies and states our research questions. In the second chapter, we then provide the required background knowledge to understand this work and most of the implementation. This includes a brief overview of how graph databases like Neo4j work, what Maven is and how it handles dependencies, and our definitions of artifacts, releases, and dependency blossoms.

The main part contains chapters 3 to 5. The third chapter focuses on the overall study design. We propose a pipeline to calculate dependency blossoms. We also point out our requirements, the infrastructure, and overall software architecture, the algorithmic approach, and the most important parts of our implementation. Inside the fourth chapter, we then explore the raw data and figures based on our implementation, whereas in chapter 5 the implications of those findings are discussed, and we try to answer our research questions. Chapter 6 puts our work in context to other topics that are researched in this field as well as contextualizing our work with other related papers.

Finally, we will wrap up our work in chapter 7, in which we draw our conclusions and discuss further work to improve our approach.

1.3. Research Questions

To evaluate our approach we want to gain some insights and quantized understanding on frameworks or dependency blossoms. In our first research question we therefore investigate how many artifacts are part of a dependency blossom: **RQ-1: How many artifacts are part of a dependency blossom?**

We also set out to evaluate the current definition of dependency blossoms provided by Dann et al. [9]: **RQ-2: Is the reduction to packages of the same group and version sufficient for a dependency blossom?**

We then further want to examine how different blossoms of base artifacts that share a common artifact group are composed. Using those insights we also can evaluate whether the dependency blossom provided by Dann et al. [9] can be used a simplification: **RQ-3: How large is the relative overlap between dependency blossoms of base artifacts sharing a common artifact group?**

1.4. Scientific Contribution and Results

In this work we provide a proof-of-concept implementation to compile the dependency blossoms based on our broader definition. This tool allows the compilation of dependency blossoms for a provided dependency repository (e.g. Maven Central Repository) and therefore the creation of datasets with highly interconnected dependencies. Those dependency blossoms can be used to further study update incompatibilities in frameworks or to better evaluate the framework updates provided by tools like DependaBot, RenovateBot, or Snyk.

Further, we provide a different definition to the newly formed term of dependency blossoms by Dann, et al. [9] in this work.

We answer the research questions stated in Section 1.3:

- For **RQ-1**, we find that dependency blossoms vary vastly in size when applying our approach, with sizes resulting between 1 and 1930 artifacts, indicating that our approach might return too large blossoms.
- Regarding **RQ-2**, our results show that 75% of the dependency blossoms consist of a minimum of 1 and up to 6 artifact groups given a certainty threshold of 0.8 or higher, thus indicating that the approach by Dann et al. [9] is too restrictive.
- For **RQ-3**, we analyzed the overlap between blossoms for base artifacts sharing a common group, finding that the overlap is typically below 10%.

Those numbers should be seen as an overestimation because we keep candidates even if they are not a part of the dependency tree of a given base artifact. Due to the high computational cost of constructing the dependency trees and the subsequent traversing time, we do not filter for candidates that are not a part of the dependency tree in this work.

2. Background

With this chapter we provide the required background knowledge to understand this thesis. We introduce the graph database Neo4j and the Maven package manager ecosystem.

2.1. Neo4j

First, we outline the functionality of graph databases using Neo4j as an example implementation. We chose Neo4j because we use the Goblin framework dataset by Damien et al. [17]. This dataset provides a Neo4j database dump to recreate their mined data and the dependency graph of the Maven Central Repository. The following sections will explain the functionality of graph databases and how Cypher the query language of Neo4j works. This is needed to understand how we query the Goblin Framework in our implementation.

2.1.1. Property Graph Database Model

Graph databases store data using nodes and edges in graph structures or representations [30].

Neo4j uses a (labeled) property graph database model [25]. Property graph database models consist of nodes that model discrete objects and can be connected by relationships, which are represented by the graphs edges and are usually unidirectional. Nodes and relationships can be tagged [25, 30]. Neo4j calls those tags *labels* for nodes and *types* for relationships. In Neo4j, labels are implemented to model the domain by classifying nodes into sets of the same type, where all nodes with the same label are part of the same set. Types are used to define the relationship based on the given domain [25]. For example, Damien et al. [16, 17] used the labels *Release*, *Artifact*, and the types *dependency*, *relationship_AR* in their Neo4j database. A minimal example of this can be seen in Fig. 2.1.

In property graph database models, nodes and relationships can hold an arbitrary number of attributes or properties. In Neo4j these properties are key-value pairs [25, 30]. The minimal example of Fig. 2.1 uses properties like *releaseId*, *timestamp*, or *artifactId* for the nodes.

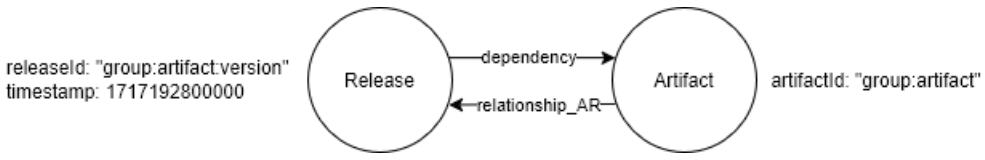


Figure 2.1.: Minimal Example of a Graph with two Nodes and Relationships

2.1.2. Querying Graphs

Since we use an already prepared Neo4j database based on the Goblin framework provided by Damien et al. [16, 17], we are not concerned about how to create or how to ingest data into a graph database. Instead, this subsection will focus on explaining how one can query graph databases using Cypher, the query language from Neo4j [3] based on the Fig. 2.1 model. In this work we are primarily concerned about querying single nodes that fulfill a filter, or we want to find all end nodes of a certain relationship with a specific start node.

Querying Nodes

To query nodes that are compliant with one or multiple specific patterns one can use the `MATCH` clause. The following query has no patterns specified and simply returns all nodes:

```
MATCH (n) RETURN n
```

Node labels like *Release* and properties like *id: group:artifact:version* can be used in order to reduce the result to nodes that have the *Release* label and a key-value pair *id: group:artifact:version*. The following query will return all nodes that have the *Release* label and where the key *id* equals the value *group:artifact:version* [5]:

```
MATCH (r:Release {id: "group:artifact:version"}) RETURN r
```

If we would like to set certain constraints on those patterns, we can use the `WHERE` subclause. For example, if we would like to query for values that are smaller or bigger than a given value, we can achieve that using the `WHERE` subclause. The following query uses the `WHERE` subclause to return all nodes that have the *Release* label and have a timestamp that is before 2025-01-01 [8].

```
MATCH (r:Release) WHERE r.timestamp < 1735686000000 RETURN r
```

Querying Paths

Similar to querying nodes, one can use the `MATCH` clause to query relationships. Neo4j offers some flexibility on how to match relationships. By using `--`, one will match any relationship, disregarding its type or its properties. If one wants to query for a directional relationship, one can do so by adding angle brackets corresponding to the desired direction. Also, similar to nodes, one can filter for a relationship type by providing it like `-[t:Type-]`. E.g., if one would like to query dependencies of a release, one could use the following query:

```
MATCH (r:Release)-[d:dependency]->(a:Artifact) RETURN r,d,a
```

In the above query, we have a relationship of type dependency that goes from a release *r* to an artifact *a*. The query will return all releases, dependency relationships, and artifacts that fulfill the given pattern. Because of the `->` in front of the artifact node, we provide the direction of this relationship [5].

Other Useful Clauses

Because we are working with a rather large dataset, we implemented the Neo4j queries with paging. For that we need certain clauses like `SKIP` and `LIMIT`.

`SKIP` is used to skip the first n provided rows and will start including the provided row in the output [7]. We use this later on to ensure that we will skip all the already processed nodes and only query unprocessed ones.

`LIMIT` can be added to the query to set a maximum amount of elements that will be returned by the query [4].

However, official documentation [4, 7] states that `SKIP` and `LIMIT` do not guarantee the results. To achieve guaranteed results for both of them, we have to use the `ORDER BY` clause, which sorts the nodes by the given criteria [6].

```
MATCH (r:Release) RETURN r ORDER BY ID(r) SKIP 50 LIMIT 50
```

The above query returns 50 releases ordered by their internal id while the query skips the first 50 nodes and therefore providing nodes 51–100.

2.2. Maven

Maven is a software project with the goal to simplify different steps of a software release cycle, including builds, dependency management, or distribution/publication, by controlling all these steps using a single project object model (short: POM) pom.xml file. This file can then be used by the CLI tool to properly resolve the dependencies and set properties like the name or version of the release [2].

2.2.1. pom.xml

The entire build and publish configuration for each Maven project resides inside the `pom.xml`. This file manages the name of the group, name, and version of the artifact. It also contains the dependencies of the project. A sample pom.xml can be found in listing 2.1.

This pom.xml shows that the project is part of the *org.example.group* group, has the name *example-artifact*, and the current build version is *1.0.0*. We can also see that this project depends on the *other-example* artifact with a version range starting from *0.1* up to, but not including, *0.2*.

```xml
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <packaging>jar</packaging>

    <groupId>org.example.group</groupId>
    <artifactId>example-artifact</artifactId>
    <version>1.0.0</version>

    <name>example</name>
    <description>This is simply an example</description>

    <dependencies>
        <dependency>
            <groupId>org.example.group</groupId>
            <artifactId>other-example</artifactId>
            <version>[0.1.2,0.2)</version>
        </dependency>
    </dependencies>
</project>
```

Listing 2.1: Example pom.xml

2.2.2. Maven Central Repository

Maven also aims to simplify the distribution of releases.

Therefore, it offers a publicly available central repository. It provides entries for multiple different build management tools like Maven itself or gradle. Both of them were the most used JVM-based build management tools based on the StackOverflow 2025 user survey [39] making it *the* default registry for JVM-based projects and build management tools. As of now (2025-10-20), the Maven Central Repository contains 18,096,408 packages [23]. Its popularity as well as the availability of the Goblin framework dataset are the reasons why we implemented our work based on the Maven Central Repository.

Alternative public and private repositories do exist but are much smaller, with the second biggest publicly available repository being *Atlassian External*. It is only 1/5 of the size of Maven Central and are therefore not as frequently used [23].

Package registries like Maven Central are a common concept across multiple programming language ecosystems like the npm registry for JavaScript, the Python Package Index for Python, or community crates for Rust [26, 32, 35].

For Maven projects the dependencies are usually addressed by a triplet containing the package **G**roup, the **A**rtifactId and **V**ersion (GAV). Maven Central allows its users to simply download the given dependencies or publish new releases via HTTPS, enabling developers to easily share and reuse code. This is possible because when publishing a new release, the corresponding pom.xml is released together with the package and can therefore be used to build a dependency tree or evaluate statistics of those packages.

2.3. Definition of Artifact, Release, and Blossom

Maven differentiates between artifacts and releases. Within this section we provide short definitions on how we use the terms artifact and release. We then present our own definition and approach towards dependency blossom.

2.3.1. Artifact and Release

In this work we differentiate between artifacts and releases and follow the definitions used by Damien et al. [16]. When we are talking about artifacts we refer to a dependency based on group and artifact id but without a specific version. Releases are a specific version of an artifact. For example *junit:junit* refers to the JUnit artifact whereas *junit:junit:4.13.2* refers to a release of JUnit. Artifacts can therefore be understood as a group of releases.

2.3.2. Blossom

We want to provide a tool that can identify dependency blossoms for package registries. In order to do that we give a definition of the term blossom. In this work we explicitly do not refer to the blossom algorithm used in graph theory to find a maximum matching in a graph as defined by Edmonds [13].

Rather, we want to concentrate on the idea of Dann et al. [9], that a dependency blossom is a set of artifacts that are part of the same group and usually require the same version to work properly. They based their definition on the findings of a study by Paschenko et al. [28] that found that frameworks commonly include dependencies that share the same group.

However, in this work we want to put this definition to the test by developing a pipeline that finds blossoms based on release dates. We define dependency blossoms as a set of artifacts that are highly interconnected belonging to the same library or framework. With our definition, we alleviate the same group and same version restrictions introduced by the definition from Dann et al. [9].

We base our definition on the assumption that developers publish their tightly coupled or highly interconnected libraries or frameworks with as little delay as possible. Because of scarcely available research regarding release behavior of libraries or frameworks, we based our assumption on our personal experience as developers and users of libraries and frameworks.

3. Study Design

In this chapter, we explain the multiple stages for establishing our blossom calculation pipeline:

1. Gathering metadata like dependencies and release dates of published libraries (out of scope)
2. Finding possible candidates that could be part of a blossom based on release dates
3. Compiling potential blossoms
4. Filter out dependencies that do not reach the required blossom threshold (out of scope)
5. Filter out dependencies that are not part of the dependency tree for a given release (out of scope)

The first step of the pipeline is already done for the Maven Central Repository by Jaime et al. [16]. Their resulting dataset is available as a Neo4j dump [17]. The last two steps of the pipeline are not implemented in this work and could be part of future work. In the rest of this chapter we present or design requirements, algorithmic approach, infrastructure setup and implementation of this pipeline.

3.1. Requirements

We start out with deriving requirements. We also want to emphasize that we will implement a proof-of-concept rather than a fully implemented system.

3.1.1. Non-Functional Requirements

We identified the following non-functional requirements for our work by considering our (computational) resources, reproducibility and the size of the dataset. Because of the heterogeneous systems as a mix of the authors' devices and a hosted portainer instance provided by the university we want to ensure, that our work can run on all available devices. Further, we want to allow for distributed systems because some databases are reaching sizes of 40 GiB and more. In order to handle smaller disk sizes on mobile devices we want to be able to host the databases on a different system then the implemented pipeline steps. Because we want to process all releases of the Maven Central Repository included in the Goblin framework dataset [17], we need to handle about 16.2 million releases. Therefore, we want to ensure scalability in form of computational resources as well as being able to distribute the work across multiple machines. A summary of the non-functional requirements can be found in table 3.1

Table 3.1.: Non-Functional Requirements

NFR-1	The work should run on the authors' devices and on the hosted portainer instance provided by the university
NFR-2	The setup should be rather simple to allow others to recreate the results with minimal effort
NFR-3	The work should be scalable to ensure reasonable performance on larger repositories
NFR-4	The system should be able to be run distributed across multiple systems

For **NFR-3**, we want to point out that for this proof-of-concept, we only want scalability in the form of available threads or active database connections rather than having a distributed algorithm.

Reliability and availability are not specified in the non-functional requirements because of the poc nature of this implementation.

3.1.2. Functional Requirements

To achieve the intended functionality we broke down the two steps into key functionalities we need to implement. In order to process the provided metadata we need to be able to connect the dataset and be able to access and query the provided data to compile a candidates list. Further we need the option to store our results so that the next phases can use the compiled candidates in order to calculate the certainty with which the candidate is part of a blossom. For evaluating the approach or answering our research question we want to query different statistics like amount of groups per blossom based on different blossom member thresholds, we need an option to query those statistics. We also need an interface that enables us to be able to trigger the steps of the pipeline manually and allows us to query the result data or statistics. The interface should be easy to use and fairly standardized to allow others to reuse or extend this work.

In total, we gain the following key functional requirements:

Table 3.2.: Functional Requirements

FR-1	The pipeline should be able to access the gathered metadata to compile a candidates list
FR-2	The pipeline should be able to store results to allow for easy testing of the following phase
FR-3	The pipeline should be controllable via a web interface
FR-4	The pipeline should offer a option to query some statistics on the final results as well as on the metadata

3.2. Architecture

We first present the runtime infrastructure design and then concentrate on the overall software architecture.

3.2.1. Infrastructure

The first decision we made for the implementation is that we want to run this implementation inside a docker container. This way we meet the criteria of **NFR-1** by allowing it to be run on the hosted portainer instance as well as on any other devices that has some sort of container engine (e.g. Docker Desktop) running on it.

Further we achieve **NFR-2** by providing a docker-compose.yaml inside the code-repository [18]. With utilizing *docker compose* we can control our entire application stack from a single configuration YAML file [11].

We also want to run the Goblin Neo4j and our result database as containers to ensure that they can be deployed anywhere and the provided data is not coupled directly to the implementation done in this work. Thus, we have ensured that **NFR-4** can be fulfilled by the implementation.

In order to store our (interim) results we need a database on our own. For that we choose PostgreSQL because it allows for *JSONB* as column type [31]. The *JSONB* column type allows for more complex column content like an array of key-value pairs. We use this array of key-value pairs to store, for each base artifact, the certainty with which a candidate is part of the blossom. Thus, we ensure our implementation having a way to store the results and therefore enable the achievability of **FR-2**.

3.2.2. Software

To tackle **NFR-3**, **NFR-4**, and **FR-1** to **FR-4** we choose to implement the work with the *spring framework*. The spring framework allows for a simple integration of databases into the Kotlin code base by leveraging dependencies like spring-boot-starter-data-neo4j or spring-boot-starter-data-r2dbc. Those integrations allow to communicate with databases that are available via network and can be easily configured via the application.yaml or environment variables [37, 38]. Therefore, we meet the criteria of **NFR-4**.

The spring-boot-starter-data-r2dbc library allows for reactive database repositories which ensures having as little active waiting time on a database response as possible. This is achieved by utilizing reactive streams and *Publisher* and *Subscriber* patterns to offer a non-blocking data access [15]. With that we can simply use as many threads for those pipelines as are available on the host system and thus fulfilling **NFR-3**. Using spring-boot-starter-data-r2dbc in combination with PostgreSQL **FR-2** is fulfilled.

FR-1 is achieved by the usage of the spring-boot-starter-data-neo4j dependency. This dependency allows us to access the gathered metadata stored in the Neo4j graph database based on the dump of the goblin database published by Jaime et al. [17].

FR-3 and **FR-4** are achieved by using spring-boot-starter-web to have a REST API exposed that can be used to trigger each step implemented by this work separately. The REST API is also used to query the statistics used in this work. A documentation of the REST API endpoints can be found in the Appendix A. API Documentation.

To further distinguish between algorithmic implementations, controller used to handle the API requests, database abstraction logic and some constants we separated those into different packages. The algorithms are implemented inside the services and therefore reside inside the *service* package. The controller handling API request like querying certain statistics or handling the trigger for specific phases of the pipeline can be found inside the *controller* package. In general the controller do not handle any domain specific logic. They are used to specify the API endpoints and internally call the proper functions of the implemented services.

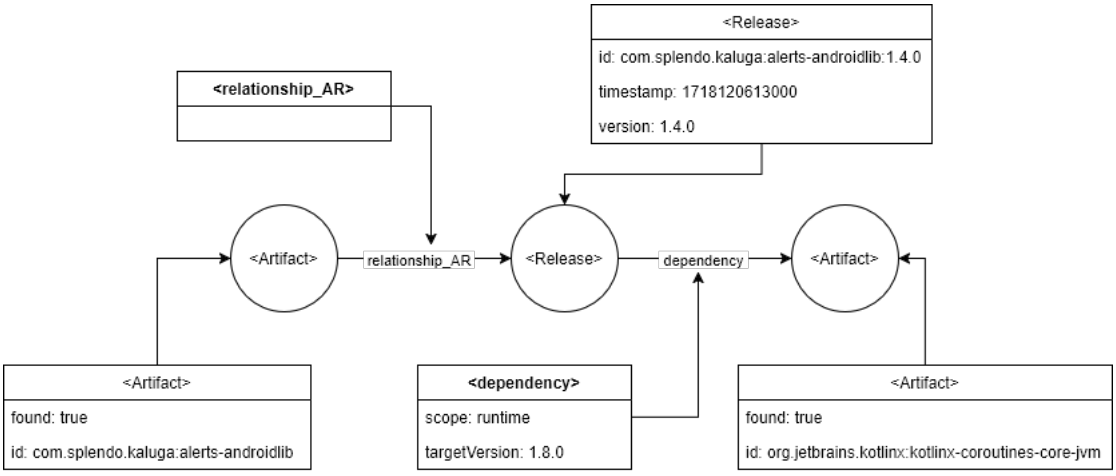
The database implementation uses the spring-boot-starter-data-neo4j and spring-boot-starter-data-r2dbc as abstractions. The necessary implementations for those abstraction layers can be found inside the *database* package.

3.2.3. Database Design

Neo4J from Damien et al.

In this section we want to briefly outline the Neo4j model provided by the Goblin framework by Damien et al. [16, 17]. Their database uses *Release* and *Artifact* labels, in which releases are concrete versions of artifacts. To model relationships, they use the types *dependency* and *relationship_AR*. The *dependency* relationship ties a release to an artifact on which it is dependent. The context of the *dependency* relationship holds additional information such as the dependency scope or target version. The second type, *relationship_AR*, on the other hand, binds a concrete release to the artifact for which it represents a version. An example can be seen in Figure 3.1.

Figure 3.1.: Minimal Example of Goblin Neo4j Structure



Result Database

In total we have two databases. One for storing candidates and one for the certainties of a candidate being part of the blossom. Both databases are used as a key value tables. For the candidates the key is the artifactId of the base artifact and the value is a text array containing all artifactIds of candidates. The following shows an example entry:

id	candidates
org.springframework.data:spring-data-commons	cn.eova:eova, io.joynr:cpp, io.joynr:java, io.joynr:joynr, ...

The table for the blossoms is similar. As a unique key we still have the artifactId of the base artifact. The value column is filled with a json object which contains all artifactIds of the candidates as a key and the certainty as a double. The following table shows an example entry from the blossoms table:

id	blossom
org.springframework.data:spring-data-commons	{"cn.eova:eova": 0.111111111111111, "io.joynr:cpp": 0.0510204081632653 1, ...}

3.3. Algorithmic Approach

In our proposed pipeline we identified two steps or phases that are to be implemented within this thesis:

- Finding possible candidates that could be part of a blossom based on release dates
- Compiling potential blossoms

In the following sections we will take a closer look on how we want to achieve each step.

3.3.1. Phase 1: Determining Dependency Blossom Candidates

Calculating those dependency trees for over 16.2 million releases is resource intensive and time-consuming.

Therefore, we designed our approach and pre-filter potential artifact candidates based on whether they have at least one release published within a six-hour time window, defined as three hours before or after a single release of the base artifact. If they have, they are a candidate. If there is not a single release inside a six-hour time window the artifact can never be part of a blossom according to our own definition. We arbitrarily choose 6 hours for the time window to allow for enough time to build and publish multiple releases but not being too liberal and thus keeping the number of candidates low.

To achieve that goal we iterate over all releases and for each release we query the artifact ids of releases that are within the six-hour window and finally store the results in our result database.

A pseudocode implementation can be found in algorithm 1.

Algorithm 1 Phase 1: Finding potential blossom candidates

Require: Metadata gathered from mining the targeted registry/repository
Ensure: Map of artifacts and list of potential candidates for the blossom

releases ⟵ get releases from metadata database
for each release ∈ releases **do**
    artifactId ⟵ artifact id of release
    candidateIds ⟵ artifact ids from releases that were released up to 3 hours prior or after release
    currentCandidates ⟵ query current candidateIds for artifactId
    candidateIds ⟵ candidateIds ∪ currentCandidates
    save map (artifactId, candidateIds) in result database
end for

Algorithm 2 Phase 2: Calculate certainty for each candidate

Require: Map of artifacts and list of potential candidates for the blossom
Ensure: Map of artifacts and list of key-value pairs of candidates with their certainty

candidatesMaps ⟵ query all entries from candidates result table
for each candidatesMap ∈ candidatesMaps **do**
    baseArtifact ⟵ artifactId of candidatesMap
    baseReleases ⟵ query all releases of baseArtifact
    baseReleaseTimestamps ⟵ map every release of baseReleases to their timestamp
    candidateIds ⟵ candidateIds of candidatesMap
    for each candidate ∈ candidateIds **do**
        candidateReleases ⟵ query all releases of candidate artifact
        timeWindowCount ⟵ amount of candidateReleases released in time window of a baseRelease
        percentage ⟵ 0.0
        if #candidateReleases ≠ 0 **then**
            percentage ⟵ timeWindowCount / #candidateReleases
        end if
        currentBlossom ⟵ query current blossom certainties from database
        blossom ⟵ currentBlossom ∪ (candidateId, percentage)
        save map (baseArtifact, blossom)
    end for
end for

3.3.2. Phase 2: Calculate Candidate Certainty

With the results of Phase 1 saved in our own database we need to calculate the certainty for each candidate with which they are part of the blossom of our base artifact.

We do that by comparing all release timestamps of our base artifact with the release timestamp of every release of a candidate. We then divide the amount of candidate releases that are within a six-hour time window of a release of our base artifact by the amount of candidate releases. This way we get percentage value that we use as certainty with which a candidate is part of the blossom. We formalized that approach in algorithm 2.

3.4. Implementation

In this section we want to focus on the concrete implementation of the algorithms and on the provided REST API.

3.4.1. Release Subset Selection

Because of limited access to computational resources, we were not able to calculate the certainties over all releases. Therefore, we decided to reduce the number of releases, we use as a starting point, down to 10,000. In every other step along our pipeline we still look at every single release.

In order to achieve reducing our start point releases, we decided to utilize the internal Neo4j id of each release node. We generated numbers between the lowest elementId of a release node (in our case 3) and the highest elementId of a release node (in our case 16,939,660). Because of depublications, some releases were removed from the original dataset, leading to missing elementIds. To account for that, we checked for each generated number that a release node with the same elementId does exist. If not, we discard this number and generate a new one. To avoid duplicates, we utilize the set functionality and generate random numbers until the set size is equal to our targeted size. This way we can ensure that we pick exactly 10,000 releases.

We then saved the resulting release elementIds as a .json file, which is also included in the code repository [18]. The implementation can be found inside the `RandomService` class.

3.4.2. Algorithm Implementation

In section 3.3 we already provided the pseudocode implementation of the algorithms used to complete both of the steps of our pipeline we want to implement in this work. In the next two sections, we explain why we deviate from the pseudocode in order to take advantage of database optimizations and use multithreading to further improve computing time.

Implementation of Phase 1

In phase 1 we implemented the database query to utilize a paging approach in order to reduce memory cost. Paging describes the process of querying smaller batches of releases and then processing them before querying the next batch. This way we only keep a few hundred releases in memory. We also needed to tweak the Neo4j query because we want to limit the releases we process down to those 10,000 we randomly selected. For that we use the following query:

```
MATCH (r:Release) WHERE id(r) IN $limitedIds RETURN r ORDER BY id(r) SKIP $skip LIMIT $limit
```

`$limitedIds` is the list of our randomly selected ids with which we limit the releases to the randomly selected ones. `$skip` and `$limit` are used to make the paged query. `$skip` tells how many releases to skip, whereas `$limit` limits how many releases are part of the result. Using both of them together allows us to control exactly how many releases we have and how many we want to skip. Thus, ensuring we always get a page with a maximum of `$limit` releases, skipping all already processed releases.

In our implementation we also use reactive programming in order to avoid idle time while waiting on a response from the databases. For that, we used *Flux* and *Mono* from the *Reactor 3* library because they implement reactive streams (*Flux* for $n \in \mathbb{N}$ elements, and *Mono* for 0 or 1 element).[1] Therefore, we implemented the algorithm using pipeline chaining in order to process the releases in streams, avoiding breaking the asynchronous reactive stream and therefore losing the performance boost they might offer.

Using them, we simply *map* the results of each previous map until we reach our goal.

```
1  fun phase1() {
2      ...
3      return Flux
4          .generate({ 0 }) { state, sink ->
5              sink.next(state)
6              state + pageSize
7          }.takeWhile { skip -> skip < releaseCount }
8          .concatMap { skip -> fetchPage(skip) }
9          .concatMap { release ->
10             getReleasesInReleaseTimespan(release.id)
11             .flatMap { upsertCandidates(release.idToArtifactId(), it.candidateIds) }
12         }
13         .then()
14 }
```

Listing 3.1: Implementation of Phase 1

In line 2 we start with a state where our skip is set to 0. In the next two lines we set our state, and afterwards we increase our skip by the amount of our page size so that our next state will query the next page. In line 5 we tell the stream to take elements as long as we have not reached all releases. Inside the *concatMap* we map the skip to the result of the `fetchPage(skip)` function. The `fetchPage(skip)` function is a helper function that internally calls the Neo4j query. We introduced it to allow for easy switching between our approach limited to 10,000 releases and the approach that handles all releases. In the following *concatMap* we tell our stream to transform the incoming releases by getting all releases that are released within a six-hour time window. The function handling this task is called `getReleasesInReleaseTimespan`.

[1] Further information on Flux or Mono can be found here: https://projectreactor.io/docs/core/release/reference/gettingStarted.html

We then save the base artifact id with the returned `candidateIds` inside our database. The `.then()` in the end simply returns a `Mono<Void>` once the stream is fully processed.

But with the current implementation, we are missing the step where we merge the newly found candidates for that artifact with the already found ones. We moved this logic into the `upsertCandidates` to reduce traffic between the database and code to reduce potential networking issues.

```
1  private fun upsertCandidates(releaseId: String, candidateIds: List<String>): Mono<Long> {
2      val sql = """
3          INSERT INTO candidates AS c (id, candidates)
4          VALUES ($1, $2)
5          ON CONFLICT (id)
6          DO UPDATE SET candidates = (
7              SELECT ARRAY(
8                  SELECT DISTINCT elem
9                  FROM unnest(c.candidates || EXCLUDED.candidates) AS elem
10             )
11         )
12     """.trimIndent()
13
14     return databaseClient.inConnection {
15         val stmt = it.createStatement(sql)
16         .bind(0, releaseId)
17         .bind(1, candidateIds.toTypedArray())
18     val exe = stmt.execute()
19     Mono.from(exe)
20         .flatMap { result -> Mono.from(result.rowsUpdated) }
21     }
22 }
```

Listing 3.2: upsertCandidates

We achieve the merging step in the database write commit by utilizing the implemented *ON CONFLICT* functionality of PostgreSQL.[2] In general, the idea is trying to insert the current `candidateIds` for a given base artifact id as a key. If our result database already has an entry for that key, a conflict occurs. It then will update the existing entry by distinctly merging all elements together and thus ensuring uniqueness for each candidate id in our final result.

Implementation of Phase 2

We also altered the implementation of phase 2 a bit. We are again using the reactive streams in the form of *Flux*. Further we diverged on our merging approach. This time we wait until every candidate is processed by the code and then collect the *Maps* of candidateId and their corresponding percentage id as a list. We then save this list of candidate ids and their corresponding certainties to our blossom database.

[2] More on the *ON CONFLICT* can be found here: https://www.postgresql.org/docs/current/sql-insert.html#SQL-ON-CONFLICT

```
1  fun phase2(): Mono<Void> {
2      return candidatesRepository.findAll()
3          .flatMap { candidatesMap ->
4              val baseArtifact = candidatesMap.id
5              val candidateIds = candidatesMap.candidates
6
7              mavenCentralRepository.findReleasesByArtifactById(baseArtifact)
8                  .map { it.timestamp }
9                  .collectList()
10                 .flatMap { releasesTimestamps ->
11                     Flux.fromIterable(candidateIds)
12                         .flatMap { candidateId ->
13                             mavenCentralRepository.findReleasesByArtifactById(candidateId)
14                             .collectList()
15                             .map { candidateReleases ->
16                                 val timeWindowCount = candidateReleases.count { candidateRelease ->
17                                     releasesTimestamps.any { releaseTimestamp ->
18                                         candidateRelease.timestamp in
19                                             (releaseTimestamp - constants.TIMESPAN_OFFSET)..
20                                             (releaseTimestamp + constants.TIMESPAN_OFFSET)
21                                     }
22                                 }
23                                 val percentage = if (candidateReleases.isNotEmpty()) {
24                                     timeWindowCount.toDouble() / candidateReleases.size
25                                 } else 0.0
26                                 candidateId to percentage
27                             }
28                         }
29                         .collectList()
30                         .flatMap { percentagesList ->
31                             upsertBlossoms(
32                                 baseArtifact,
33                                 jacksonObjectMapper().writeValueAsString(percentagesList.toMap())
34                             )
35                         }
36                 }
37         }
38         .then()
39 }
```

Listing 3.3: Implementation of phase 2

In the pseudocode of Phase 2 (Algorithm 2) we directly map the found releases of the base artifact to their release timestamp. We then collect all of those release timestamps as a list and use them further on (lines 7 and 8).

After that we create a *Flux* in order to have reactive streams to avoid idle time. Inside those *Flux*, we iterate over all candidate ids and conceptually follow the pseudocode implementation.

In lines 29–35 we diverge a bit from the pseudocode again. Instead of updating the database entry for each new processed candidate, we process all candidates and then push all of them at once as a list using the `upsertBlossom` function.

3.4.3. REST API Implementation

As mentioned in **FR-3** in section 3.1.1 we want a web interface to control our implementation or to query data. For that we settled with the *spring-boot-starter-web* library, which is used to provide REST APIs in JVM projects.[3]

Because this is not the main focus of the work but part of our functional requirements, we want to give a short overview of how we used spring-boot-starter-web to make a REST API available. In order to do that, we will use our `TriggerController` as an example.

```
@RestController
@RequestMapping(value = ["/trigger"])
class TriggerController(
    private val blossomService: BlossomService,
) {

    @PostMapping("/phase1")
    fun triggerPhase1(): Mono<String> {
        blossomService.phase1()
            .subscribe()
        return Mono.just("Phase 1 started.")
    }

    @PostMapping("/phase1limited")
    fun triggerPhase1Limited(): Mono<String> {
        blossomService.phase1(true)
            .subscribe()
        return Mono.just("Phase 1 started.")
    }

    @PostMapping("/phase2")
        fun triggerPhase2(): Mono<String> {
        blossomService.phase2()
            .subscribe()
        return Mono.just("Phase 2 started.")
    }
}
```

Listing 3.4: TriggerController

What can be inferred by this implementation is that all the post mappings are available at the following path: `/trigger/<endpoint>`. In this case the options for `<endpoint>` are `phase1`, `phase1limited`, and `phase2`.

Those endpoints can be called by tools like Postman[4], Insomnia[5], or cURL[6]. The following listing shows how one can trigger Phase 1 using curl.

```
curl -X POST http://localhost:8080/trigger/phase1
```

[3]More information available at https://docs.spring.io/spring-boot/reference/web/servlet.html
[4]https://www.postman.com/
[5]https://insomnia.rest/
[6]https://curl.se/

Each of these endpoints calls the respective implementation inside the `BlossomService`. The `phase1limited` endpoint is special because it also calls `blossomService.phase1()` but it provides a parameter. As already stated in section 3.4.2 we needed to tweak our Neo4j query to only return our randomly selected releases. For that, we introduced a helper function called `fetchPage`. Depending on the parameter passed on to `blossomService.phase1()` the helper function switches between returning all releases (false, which is the default) or only the randomly selected releases (true). We use the `.subscribe()` to keep the function running even if the overall API handling method finishes and returns the message back.

A full API specification can be found in the Appendix A. API Documentation.

Warning: The API is not protected by any security measures. Please avoid exposing it publicly without additional security measures in place.

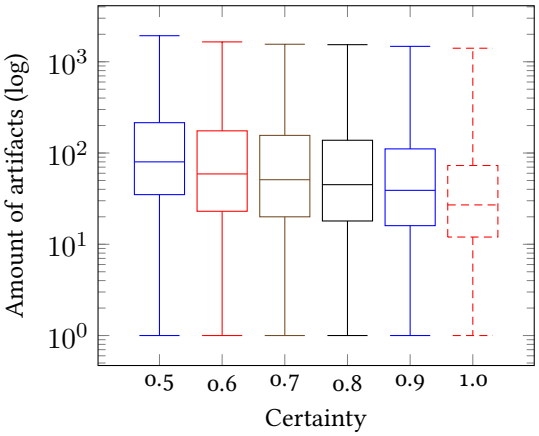
4. Research Results

In this chapter we present the raw results based on our statistical analysis of the results of our found dependency blossoms. All the 10,000 randomly selected releases resulted in 9,151 potential dependency blossoms.

4.1. Dependency Blossom Sizes

To analyze how the certainty threshold influences the blossom sizes, we computed statistics for all blossoms. We looked at each blossom and filtered for all entries that match the certainty threshold or exceed it. Across all thresholds, the blossom sizes vary between 1 and 1,930 artifacts. As illustrated in Figure 4.1, increasing the certainty threshold reduces the median blossom size as well as the interquartile range. We explicitly chose to represent that data without outliers because preliminary analysis indicated that 12.6% of data points would be classified as outliers above the maximum (1,151 outliers for a certainty threshold of 0.5). At a certainty threshold of 0.5, the median size of a blossom is 80 artifacts, with an interquartile

Figure 4.1.: Sizes of Blossoms based on Certainty



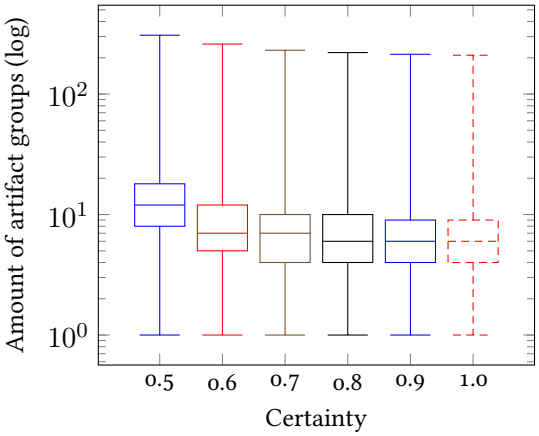
Certainty	Min	Q1	Median	Q3	Max
0.5	1	35	80	215	1930
0.6	1	23	59	175	1651
0.7	1	20	51	156	1556
0.8	1	18	45	138	1539
0.9	1	16	39	111	1475
1.0	1	12	27	73	1407

range from 35 to 215. The biggest blossom (with a certainty threshold of 0.5) has a maximum of 1930 artifacts. With an increasing certainty threshold, the data indicates a decreasing median, quartiles, and upper whiskers. With a certainty threshold of 1.0, we have a median of 27 artifacts in size, with an interquartile range from 12 to 73.

4.2. Groups inside a Dependency Blossom

The number of artifact groups per blossom was analyzed for different certainty thresholds. Figure 4.2 shows the amount of groups for all dependency blossoms based on the certainty threshold. Across the thresholds, the number of groups ranges from 1 to 300. The exact median and quartile values for each certainty threshold can be seen in table of figure 4.2. At a certainty threshold of 0.5, the median number

Figure 4.2.: Amount of Artifact Groups in Blossoms based on Certainty



Certainty	Min	Q1	Median	Q3	Max
0.5	1	8	12	18	308
0.6	1	5	7	12	260
0.7	1	4	7	10	231
0.8	1	4	6	10	221
0.9	1	4	6	9	214
1.0	1	4	6	9	210

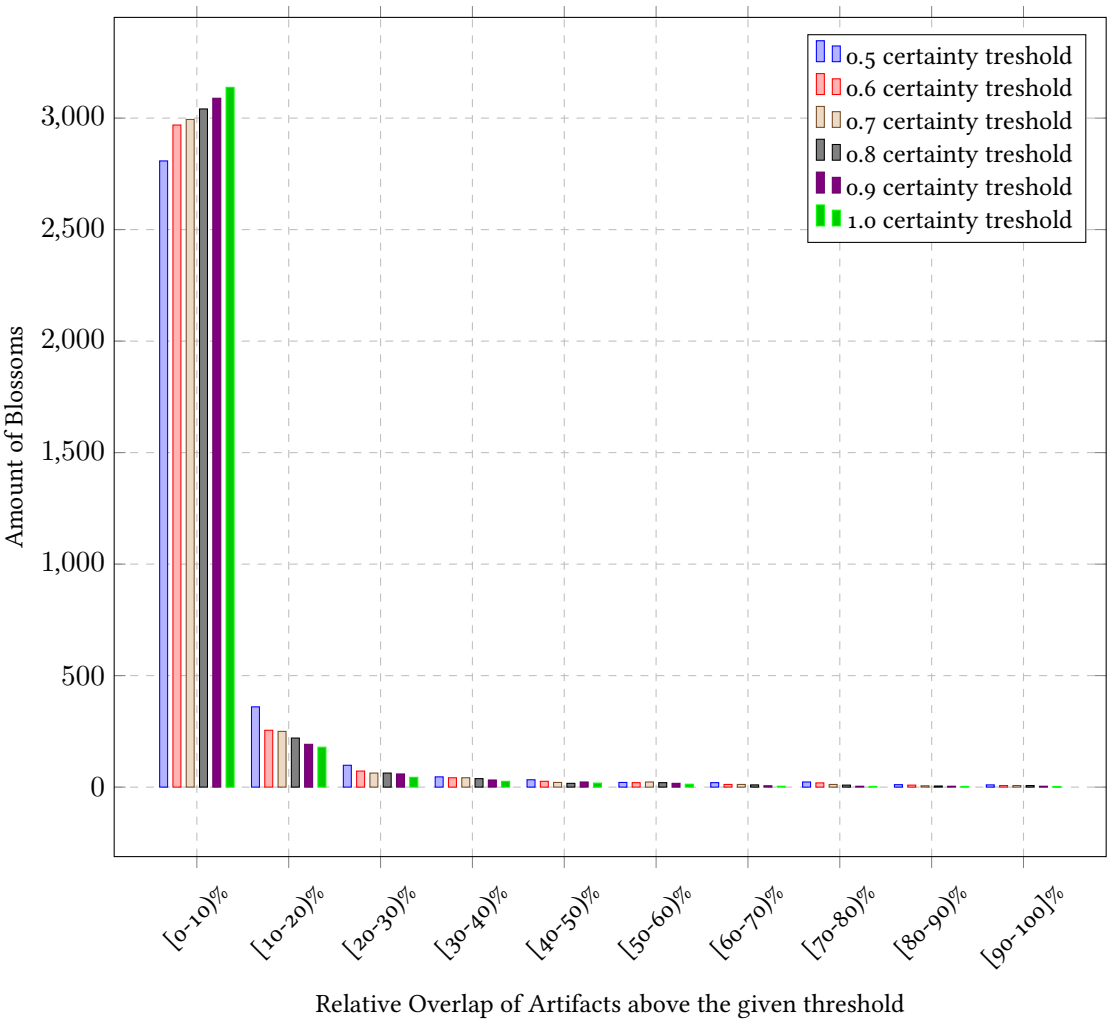
of groups is 12, with an interquartile range from 8 to 18. With an increasing certainty threshold, the median decreases gradually, reaching median values of 6 for thresholds 0.8 and above. A similar pattern exists for the quartiles and upper whisker. Only the lower whisker remains constant at a value of 1.

4.3. Overlap in Blossom of Artifacts Sharing a Group

To analyze the relative overlap of dependency blossoms within artifact groups, we first grouped the blossoms to artifact groups based on the group of their base artifact. We then calculated the amount of artifacts that appear in and also exceed the certainty threshold in all the dependency blossoms being part of this group. This was done for thresholds ranging from 0.5 to 1.0. Figure 4.3 visualizes the relative overlap over all artifact groups. Each bar corresponds to a range of overlap percentage and a certainty threshold.

In total, the dataset contains 3,430 artifact groups. The majority of blossoms within an artifact group have a low overlap. For a threshold of 0.5, 2,808 blossoms share less than 10% of the same artifacts with other blossoms of the same group. With increasing certainty thresholds, the amount of blossoms sharing less than 10% per group increases to 3,138 for a certainty threshold of 1.0.

Figure 4.3.: Relative Overlap Between Blossoms of Artifact Sharing a Group



5. Discussion

In this chapter we provide our interpretation of the research results, answer our research questions, and point out threats to validity.

5.1. Interpretation of Results

Since we provide a different definition of dependency blossoms with a different approach to compiling them, we want to evaluate our approach in this section. In chapter four we already explored the sizes of blossoms, where the span of the found blossom sizes is between 1 and 1,930 for a certainty threshold of 0.5. However, in their work, Dann et al. [9] defined the term dependency blossom to be artifacts of the same artifact group and version. They further conclude that this indicates that they belong to the same framework.

The only research on framework or cluster detection on Maven Central we were able to find is the work of Lake and Zibran [20]. But their work mostly focuses on how to leverage those dependency clusters to identify high-risk clusters, and they identified those clusters using the Leiden algorithm. This results in an even broader definition for their clusters than we would like to apply to our blossoms. Therefore, we have no statistical data to compare our found blossom sizes to or to see if they fit the average size of a framework on Maven Central Repository.

Because of the lower and upper quartiles discussed in section 4.1 we find the sizes to be plausible enough that our blossoms actually represent frameworks and our definition is not too broad. The median values between 27 artifacts (for a certainty threshold of 1.0) and 80 artifacts (for a certainty threshold of 0.5) do support our conclusion as well. However, the size alone is not sufficient to evaluate whether our definition actually represents frameworks in a meaningful manner. So further work is still needed to show that our approach could be used to identify frameworks on the Maven Central Repository.

In section 1.1 we stated that we would like a broader approach to the dependency blossom definition. We achieved that by lifting the same group and same version restrictions. Data shows that, given our definition, the median values for the amount of artifact groups inside our blossoms are between 6 and 12. Thus, indicating restricting blossoms to the same group and version might not be a sufficient way to detect frameworks on the Maven Central Repository. But, since our approach does not consider the dependency tree, it keeps false positives in the candidate set, causing the resulting numbers to be higher. Therefore, our results should be interpreted as an upper limit rather than final values.

Nonetheless, while inspecting the blossom of the base artifact `aws.kotlin.sdk.kotlin:eks` we found entries like

- `com.amazonaws:aws-java-sdk-eksauth` with a rounded certainty of 83%,
- `com.amazonaws:aws-java-sdk-bedrock` with a rounded certainty of 77%,
- `software.amazon.awssdk:supportapp` with a rounded certainty of 54%,
- or `software.amazon.awssdk:costoptimizationhub` with a rounded certainty of 90%.

Those entries indicate that the artifacts, or at least their releases, are controlled by Amazon. Therefore, we found at least one example where there are multiple groups potentially belonging to the same Amazon AWS SDK library, thus confirming that the same group and same version restraint might be too strict for proper framework detection.

Further, our data indicates that all dependency blossoms of artifacts of the same artifact group only share up to 10% of artifacts above a given certainty threshold. The amount of blossoms sharing more than 10% and less than 20% of artifacts above a given threshold plummets down by a factor of 10 in comparison to those only sharing 10%. For the other intervals [20-30) up to [90-100] the trend clearly indicates that the amount of artifact shared by those blossoms is shrinking continuously but by a smaller factor. This indicates that either the evaluation of the overlap is too strict (we enforced the certainty threshold amongst all blossoms of the same group) or our approach might be too broad because we did capture many blossoms where only less than 10% of the artifacts are present in multiple blossoms from the same base artifact group. A possible cause for this low overlap could be individual dependencies that coincidentally matched the release frequency of artifacts, belonging to a framework, and are therefore considered part of the blossom. Thus, it would support the theory that our current definition might be too broad. Another cause might be that our random selection of 10,000 releases favoured sparsely interconnected frameworks which would imply a too strict evaluation of the overlap. We interpret those results to be mostly caused by individual dependencies of single artifacts belonging to a framework rather than sparsely interconnected frameworks. We do this because we observed some more densely connected frameworks like *springframework*. Further, we found multiple artifact groups of blossom members which indicate that they are controlled by a different entity than the base artifact.

But the data does suggest that the same group and version restriction may be too strict. The small overlap of blossoms of artifacts sharing a common artifact indicates that artifacts belonging to a common group do not share a lot of common blossom members and thus indicate that artifacts from the same group do not share a lot of dependencies above a given certainty threshold.

5.2. Threats to Validity

In the following section, we point out possible threats to the internal or external validity of our findings.

5.2.1. Threats to Internal Validity

There is a possible selection bias because of the 10,000 randomly selected artifacts from the Maven Central Repository based on the dataset provided by Damien et al. [17], which in itself holds the threat of not being a full representation of the Maven Central Repository. Additionally, the random selection causes some artifacts to be overrepresented in this study, while others might be vastly underrepresented. This is caused by our random selection of 10,000 releases, which favors artifacts with more releases over artifacts with fewer ones. Furthermore, our definition is based on personal experience that entities tend to release new framework versions by (re-)publishing all artifacts belonging to that framework. The six-hour time window in which artifacts have to be released in order to be a candidate for the base artifact was selected arbitrarily, which may threaten the internal validity as well. The lack of research on framework life cycles or release cycles makes it difficult to assess whether our assumptions about the release-cycle and time windows based on personal experience are valid. Those threats may also prohibit generalization across the Maven Central Repository. Another threat to internal validity would be the subjective selection of the certainty thresholds used to present the results and inside the discussion part of this thesis.

The first threat can be mitigated by evaluating the whole dataset provided by Damien et al. [17].

5.2.2. Threats to External Validity

As for external validity, we also identified the random selection of the 10,000 releases for the same reasons mentioned in section 5.2.1. There is also a bias on the Maven ecosystem because all implemented algorithms and the used dataset only used data from the Maven Central Repository. Therefore, a generalization to other build- and package-management ecosystems for other non-JVM-based languages might not be accurate. The definitions of terms like *framework* might not be transferable to other non-JVM ecosystems and thus threaten generalization across ecosystems and programming languages. The work also depends on the current standard of how Maven represents dependency trees and resolves dependency versions. Future changes might affect results.

We tried to mitigate the generalization across ecosystem threats by building a modular application allowing for integration of different datasets from different package repositories. By doing this, future work can be done on other ecosystems as well, resulting in more consistent and ecosystem-independent data. As mentioned in section 5.2.1, the threat caused by our random selection can be addressed by using the whole dataset provided by Damien et al. [17].

6. Related Work

In this chapter we provide an overview of research based around the Maven Central Repository and on the topic of dependency updates.

6.1. Dependency Updates

Researchers investigated practices around updating dependencies [19, 24]. Kula et al. [19] explored to what extent developers are updating their library dependencies, how they respond to release announcements, and why they are non-responsive to security advisories. This was done by analyzing real-world projects with a focus on popular Java projects using Maven libraries on GitHub and tracking their system and dependency updates. In order to answer how developers respond to release announcements, they did case studies on a subset of the initially used projects. To determine how developers respond to the release announcements or security advisories, they tracked how much time had passed until projects of the subset performed the library update. For the last aspect, they sent a survey out to developers of contactable projects. They specifically did not target projects having no public communication besides issue management or mailing lists.

Their research shows that about 81.5% of actively developed projects on GitHub remain with outdated dependencies. They also found an inverse correlation between estimated migration efforts and the likelihood of updates. This correlation exists because certain updates, such as those requiring an upgrade to a higher Java platform, may cause architectural changes and additional migration efforts. Further, their results show developers being unaware of vulnerabilities in their dependencies and that the decision to update is mainly based on project-specific priorities. They conclude the community should improve developers' perception of third-party updates, especially when they require more effort.

Mirhosseini and Parnin [24] researched how existing tools such as GreenKeeper, which was rebranded to Snyk, affect software dependency updates by analyzing 7,470 GitHub projects from the Node.js/npm ecosystem. They compared projects using such tools against a control group that does not use any dependency management tools by checking pull requests created by dependency management tools based on the GitHub Archive dataset. Further, they identified concerns of developers causing them to be reluctant with dependency updates by surveying developers. For their survey, they targeted developers of projects who closed pull requests created by GreenKeeper specifically, but they also posted links to their survey in various programming forums.

Their results also show developers neglecting updates, confirming the findings of Kula et al. [19]. They found breaking changes and migration efforts to be one of the main drivers for not updating dependencies. Further, their results reveal that projects using automated pull requests are 20-50% more likely to use the latest versions of their dependencies compared to the baseline. However, their results also show using pull request based, automated upgrades caused a 35% higher downgrade rate compared to the baseline. The downgrade usually happens in 2–3 days after the upgrade. They identified code failing to function properly as the main reason, forcing them to downgrade again.

Both studies explore the topic of how developers handle library or dependency updates in general. They found that one of the main reasons for developers to be reluctant with updates is the fear of breaking changes. Those might be minimized if tools proposing version upgrades via pull requests ensure update compatibility for the projects. Dann et al. [9] implemented UpCy trying to minimize update incompatibilities by checking whether API calls suffer from source, binary, or semantic incompatibilities. Their approach also includes a second step where UpCy explores complex updates potentially involving updates of multiple dependencies to minimize those incompatibilities. They evaluated their approach by testing how UpCy performed for different tasks like updating frameworks or libraries, for which they coined the term dependency blossom. Our work focuses on an alternative approach to dependency blossoms, trying to find a more fitting approach to detect frameworks and libraries. A better detection of frameworks can lead to a better dataset, which then can be used to better evaluate the performance of dependency management update tools.

6.2. Structures on Maven Central Repository

Others investigate the overall structure, clustering, and interconnectivity of the Maven Central Repository [20, 27]. Ogenrwot et al. [27] explored the overall connectivity of the Maven Central Repository by using the top 5,000 highly connected artifacts and then applying a breadth-first-search to traverse through the graph and capture all releases reachable by this approach. This way they created a graph containing 1.3 million nodes on which they analyzed multiple metrics like PageRank or betweenness centrality. Their results show that the Maven Central Repository functions as a universal supply chain, with 99.81% of artifacts belonging to the same dependency network. The found metrics further demonstrate the existence of predominant hubs consisting of core ecosystem infrastructure, testing frameworks, or general-purpose utility libraries. Based on their findings, they conclude that there is a high rate of proper code reuse.

Lake and Zibran [20] used a different approach by leveraging the Leiden community algorithm to identify clusters and evaluate their overall risk factors by weighing them using principal component analysis. This study found 67,669 distinct interconnected clusters, most of them containing fewer than 100 nodes. They also found some larger clusters with sizes up to 1.8 million nodes that act as dominant clusters acting as hubs between smaller clusters, thus confirming the findings of Ogenrwoth et al. [27]. Their risk evaluation showed a low to moderate security risk for most of the highly connected clusters, while smaller and less connected ones have a high security risk.

In contrast to their research, our work focuses less on the overall structure of the whole Maven Central Repository and rather on identifying frameworks or libraries we estimate to be smaller than the analyzed clusters. However, finding smaller communities within those clusters by applying the Leiden algorithm could suffice as another viable approach to reliably identifying frameworks or libraries.

6.3. Mining Software Repositories

Further research provides queryable dependency graph datasets, allowing for weaving in data that changes in a timely manner (e.g., CVE scores) [16]. Damien et al. [16] developed the Goblin framework to extract artifacts and releases from the Maven Central Index Archive and provide a Neo4j database to enable others to query the Maven dependency graph. They compared their dataset to others provided by [1, 9, 22, 33] and pointed out that the Goblin framework has the advantage of being extendable without having to recalculate from scratch. This is achieved by only applying their Goblin-Miner on those entries inside the Maven Central Index Archive that are newer than the latest release data to be found in the currently existing dependency graph. The Goblin-Weaver allows researchers to include real-time (meta)data such as CVEs to analyze them as well. By leveraging the approach of computing and adding metrics on demand, their framework enables researchers to include the latest CVE entries without having to recalculate most of the dependency graph. Thus, the Goblin framework is offering fairly up-to-date datasets and metadata for the Maven Central dependency graph.

Their Maven Central dependency graph allows research such as ours to reliably test our theories against the Maven Central Repository without having to mine it, thus helping us to focus on our theories.

7. Conclusion

In this thesis, we provide a different approach to detect *dependency blossoms*, first introduced by Dann et al. [9]. We further suggest a pipeline to compile those dependency blossoms based on our new approach. As recent research shows, developers are hesitant to update their software dependencies because they fear breaking changes in frameworks or libraries. Thus, further research on automated framework detection is required to help tools like DependaBot, RenovateBot, and Snyk evaluate update quality when updating versions of highly interconnected parts of the dependency tree. Dann et al. [9] provided an approach to identify those highly interconnected parts inside the dependency tree by assuming that those blossoms are identifiable by applying a same group and version restriction. We wanted to evaluate this approach. Therefore, we introduced a broader approach leveraging a time window around the releases to determine candidates. We then calculate the relative amount of the candidate releases that match a given time window around releases of the base artifact. The approach was evaluated on 10,000 randomly selected releases from the Goblin dataset [17]. Our results show that the approach towards dependency blossoms provided by Dann et al.[9] appears to be too strict. For all evaluated certainty thresholds, our data shows that the lower quartile suggests that 75% of blossoms contain 4 artifact groups or more and 50% of blossoms contain 6 artifact groups or more. As discussed in chapter 5 those values are an upper limit because we currently do not exclude candidates that are not part of the dependency tree. We implemented this approach to languages that are compatible with the Maven Central Repository. However, our implementation can be adapted to support other package managers or repositories as well.

7.1. Results of Research Questions

RQ-1 *How many artifacts are part of a dependency blossom?*
Our data shows a big span in our blossom sizes, spanning a range from 1 up to 1,930 artifacts. This indicates that our current approach is too broad and captures not only artifacts that belong to a given framework. Implementing the missing steps of our proposed pipeline could help remove the wrongfully added artifacts.

RQ-2 *Is the reduction to packages of the same group and version sufficient for a dependency blossom?*
The results show that 75% of blossoms consist of a minimum of 1 and up to 9–18 artifact groups, depending on the chosen certainty threshold. Thus, the data indicates that the reduction of blossoms to artifacts belonging to the same group and having the same version is too strict. However, the given data is most likely an overapproximation because we keep candidates that are not part of the dependency tree of the base artifact.

RQ-3 *How large is the relative overlap between dependency blossoms of base artifacts sharing a common artifact group?*
The general overlap of blossoms belonging to artifacts sharing the same artifact group is less than 10% for 81.6% – 91.5% of the blossoms, depending on the chosen certainty threshold. These findings weakly validate that the reduction to the same group and same version is too strict, because, inside a shared artifact group, blossoms seem to consist of different artifacts. But, since our blossoms might be too big after all, the overlap could increase once wrongfully added candidates are removed.

7.2. Future Work

As mentioned in chapter 5 further research on automated framework or library detection is needed in order to allow for tools like DependaBot, RenovateBot, or Snyk to improve their framework update behavior. The automated framework detection could further help study the properties like life cycles, release cycles, or quality of provided updates of frameworks, giving the community a better understanding of how frameworks are developed and used.

Another way to extend this thesis, is to implement the missing steps of the pipeline mentioned in chapter 3. Implementing those pipeline steps allows for a lower false positive rate in candidates and could therefore lead to a better representation of how frameworks are structured in the Maven Central Repository.

To allow for even more generalization across other package managers and programming languages, dependency graph datasets of repositories for other programming languages could be utilized. This could provide additional insights into how different ecosystems handle framework updates and could deepen our understandings of frameworks across ecosystems.

7.3. Data Availability

Our source code as well as our result database dump is publicly available on GitHub `https://github.com/marvinjuette/blomace` and Zenodo `https://zenodo.org/records/17772757`.

7.4. Used Resources

The `.sql` files provided with the source code are generated using LLMs. In this case we generated them using GitHub Copilot with ChatGPT-4.1 as a base model. Those files were only used for debugging and were never used to query presented data in this thesis.

Bibliography

[1] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. 2019. The maven dependency graph: a temporal graph-based representation of maven central. In *2019 IEEE/ACM 16th international conference on mining software repositories (MSR)*. IEEE, 344–348.

[2] Hervé Boutemy. [n. d.]. Introduction - Apache Maven — maven.apache.org. https://maven.apache.org/ref/3.9.11/. [Accessed 20-10-2025].

[3] Cypher introduction [n. d.]. Introduction - Cypher Manual — neo4j.com. https://neo4j.com/docs/cypher-manual/current/introduction/. [Accessed 20-10-2025].

[4] Cypher LIMIT clause [n. d.]. LIMIT - Cypher Manual — neo4j.com. https://neo4j.com/docs/cypher-manual/current/clauses/limit/. [Accessed 20-10-2025].

[5] Cypher MATCH clause [n. d.]. MATCH - Cypher Manual — neo4j.com. https://neo4j.com/docs/cypher-manual/current/clauses/match/. [Accessed 20-10-2025].

[6] Cypher ORDER BY clause [n. d.]. ORDER BY - Cypher Manual — neo4j.com. https://neo4j.com/docs/cypher-manual/current/clauses/order-by/. [Accessed 20-10-2025].

[7] Cypher SKIP clause [n. d.]. SKIP - Cypher Manual — neo4j.com. https://neo4j.com/docs/cypher-manual/current/clauses/skip/. [Accessed 20-10-2025].

[8] Cypher WHERE subclause [n. d.]. WHERE - Cypher Manual — neo4j.com. https://neo4j.com/docs/cypher-manual/current/clauses/where/. [Accessed 20-10-2025].

[9] Andreas Dann, Ben Hermann, and Eric Bodden. 2023. UPCY: Safely Updating Outdated Dependencies. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 233–244. doi:10.1109/ICSE48619.2023.00031

[10] DependaBot [n. d.]. Dependabot quickstart guide - GitHub Docs — docs.github.com. https://docs.github.com/en/code-security/getting-started/dependabot-quickstart-guide. [Accessed 12-11-2025].

[11] Docker Compose [n. d.]. Docker Compose — docs.docker.com. https://docs.docker.com/compose/. [Accessed 22-10-2025].

[12] Johannes Düsing and Ben Hermann. 2022. Analyzing the Direct and Transitive Impact of Vulnerabilities onto Different Artifact Repositories. *Digital Threats* 3, 4, Article 38 (Feb. 2022), 25 pages. doi:10.1145/3472811

[13] Jack Edmonds. 1965. Paths, Trees, and Flowers. *Canadian Journal of Mathematics* 17 (1965), 449–467. doi:10.4153/CJM-1965-045-4

[14] Douglas Everson, Long Cheng, and Zhenkai Zhang. 2022. Log4shell: Redefining the web attack surface. In *Proc. Workshop Meas., Attacks, Defenses Web (MADWeb)*. 1–8.

[15] Ben Hale, Mark Paluch, Greg Turnquist, Jay Bryant, and Elena Felder. [n. d.]. R2DBC - Reactive Relational Database Connectivity — r2dbc.io. https://r2dbc.io/spec/1.0.0.RELEASE/spec/html/#introduction.r2dbc-spi. [Accessed 05-11-2025].

[16] Damien Jaime, Joyce El Haddad, and Pascal Poizat. 2024. Goblin: A Framework for Enriching and Querying the Maven Central Dependency Graph. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. 37–41.

[17] Damien Jaime, Pascal Poizat, and Joyce EL HADDAD. 2025. *Goblin: Neo4J Maven Central dependency graph.* doi:10.1145/3643991.3644879

[18] Marvin Jütte. 2025. *Proof of concept for dependency blossom detection based on release time.*

[19] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417. doi:10.1007/s10664-017-9521-5

[20] George Lake and Minhaz F. Zibran. 2025. Analyzing Dependency Clusters and Security Risks in the Maven Central Repository. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. 260–264. doi:10.1109/MSR66628.2025.00046

[21] Mario Lins, René Mayrhofer, Michael Roland, Daniel Hofer, and Martin Schwaighofer. 2024. On the critical path to implant backdoors and the effectiveness of potential mitigation techniques: Early learnings from XZ. *arXiv preprint arXiv:2404.08987* (2024).

[22] Tobias Litzenberger, Johannes Düsing, and Ben Hermann. 2023. Dgmf: Fast generation of comparable, updatable dependency graphs for software repositories. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 115–119.

[23] Maven Central Repository [n. d.]. Maven Repository: Repositories — mvnrepository.com. https://mvnrepository.com/repos. [Accessed 20-10-2025].

[24] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE '17)*. IEEE Press, 84–94.

[25] Neo4J graph database concepts [n. d.]. Graph database concepts - Getting Started — neo4j.com. https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/. [Accessed 19-10-2025].

[26] npm registry [n. d.]. npm | Home — npmjs.com. https://www.npmjs.com/. [Accessed 20-10-2025].

[27] Daniel Ogenrwot, John Businge, and Shaikh Arifuzzaman. 2025. Structural and Connectivity Patterns in the Maven Central Software Dependency Network. In *International Conference on Software Engineering and Data Engineering*. Springer, 129–151.

[28] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable open source dependencies: counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Oulu, Finland) *(ESEM '18)*. Association for Computing Machinery, New York, NY, USA, Article 42, 10 pages. doi:10.1145/3239235.3268920

[29] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. A Qualitative Study of Dependency Management and Its Security Implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1513–1531. doi:10.1145/3372297.3417232

[30] Jaroslav Pokorný. 2015. Graph Databases: Their Power and Limitations. In *Computer Information Systems and Industrial Management*, Khalid Saeed and Wladyslaw Homenda (Eds.). Springer International Publishing, Cham, 58–69.

[31] PostgreSQL JSONB [n. d.]. JSON Types — postgresql.org. https://www.postgresql.org/docs/current/datatype-json.html. [Accessed 22-10-2025].

[32] Python Package Index [n. d.]. PyPI - The Python Package Index — pypi.org. https://pypi.org/. [Accessed 20-10-2025].

[33] Steven Raemaekers, Arie Van Deursen, and Joost Visser. 2013. The maven repository dataset of metrics, changes, and dependencies. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 221–224.

[34] RenovateBot [n. d.]. Renovate Docs — docs.renovatebot.com. https://docs.renovatebot.com/. [Accessed 12-11-2025].

[35] Rust Crates Registry [n. d.]. Rust Crates. https://crates.io/. [Accessed 20-10-2025].

[36] Snyk [n. d.]. Snyk AI-powered Developer Security Platform | AI-powered AppSec Tool & Security Platform | Snyk — snyk.io. https://snyk.io/. [Accessed 12-11-2025].

[37] Spring Boot Data Neo4j [n. d.]. Getting started :: Spring Data Neo4j — docs.spring.io. https://docs.spring.io/spring-data/neo4j/reference/getting-started.html. [Accessed 22-10-2025].

[38] Spring Boot Data R2DBC [n. d.]. Getting Started :: Spring Data Relational — docs.spring.io. https://docs.spring.io/spring-data/relational/reference/r2dbc/getting-started.html. [Accessed 22-10-2025].

[39] StackoverFlow User Survey 2025 [n. d.]. Technology | 2025 Stack Overflow Developer Survey — survey.stackoverflow.co. https://survey.stackoverflow.co/2025/technology/. [Accessed 20-10-2025].

[40] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhiliang Zhu. 2021. Will dependency conflicts affect my program's semantics? *IEEE Transactions on Software Engineering* 48, 7 (2021), 2295–2316.

Appendices

A. API Documentation

API Spec

2025-12-01

OpenAPI definition

Overview

Version

v0

POST /trigger/phase2

Starts phase 2 - Calculating certainty for each candidate.

Response 200:

OK.

Content: */* | string

POST /trigger/phase1limited

Starts phase 1 - Finding all possible candidates for a limited number of releases as start point.

Response 200:

OK.

Content: */* | string

POST /trigger/phase1

Starts phase 1 - Finding all possible candidates.

Response 200:

OK.

Content: */* | string

GET /stats/blossomSize

Returns box plot values for the sizes of all blossoms for a given threshold.

Request Parameters:

threshold: number;

Response 200:

OK.

Content: */* | BoxPlotStats

```
{
  lowerOutliers: Array<number>;
  min: number;
  q1: number;
```

```
    median: number;
    q3: number;
    max: number;
    upperOutliers: Array<number>;
}
```

GET /stats/blossomCommonArtifactsPerGroup

Return how many blossoms are in a shared relativ overlap bucket (eg. [0%-10%)).

Request Parameters:

threshold: number;

Response 200:

OK.

Content: */* | Array<integer>

GET /stats/blossomAmountOfGroups

Returns the box plot values for the amount of groups found per blossom for a given threshold.

Request Parameters:

threshold: number;

Response 200:

OK.

Content: */* | BoxPlotStats

```
{
    lowerOutliers: Array<number>;
    min: number;
    q1: number;
    median: number;
    q3: number;
    max: number;
    upperOutliers: Array<number>;
}
```

GET /random/nRandomReleases

Generate the specified amount of random numbers within the lower and upper bounds for that a release is existent in the metadata database.

Request Parameters:

amount?: integer;
lowerBound?: integer;
upperBound?: integer;

Response 200:

OK.

Content: */* | Array<integer>

GET /db/releases

Returns the releases with a pagination approach.

Request Parameters:

```
pullDependencies?: boolean;
skip?: integer;
limit?: integer;
```

Response 200:

OK.

Content: */* | Array<Release>

```
{
    identity: integer;
    id: string;
    timestamp: integer;
    version: string;
    dependencies: Array<Dependency>;
}
```

GET /db/releases/{releaseId}/tree

Returns the dependency tree of a given root release.

Request Parameters:

```
releaseId: string;
```

Response 200:

OK.

Content: */* | ReleaseDependencyTree

```
{
    releaseId: string;
    dependencies: Array<ReleaseDependencyTree>;
}
```

Schemas

BoxPlotStats

```
{
    lowerOutliers: Array<number>;
    min: number;
    q1: number;
    median: number;
    q3: number;
    max: number;
    upperOutliers: Array<number>;
}
```

Artifact

```
{
    identity?: integer;
    id: string;
    found: boolean;
}
```

Dependency

```
{
    target: Artifact;
    id: integer;
    scope: string;
    targetVersion: string;
}
```

Release

```
{
    identity: integer;
    id: string;
    timestamp: integer;
    version: string;
    dependencies: Array<Dependency>;
}
```

ReleaseDependencyTree

```
{
    releaseId: string;
    dependencies: Array<ReleaseDependencyTree>;
}
```

B. Eidesstattliche Versicherung

Eidesstattliche Versicherung

(Affidavit)

Jütte, Marvin
Name, Vorname
(surname, first name)

225896
Matrikelnummer
(student ID number)

☒ Bachelorarbeit
(Bachelor's thesis)

☐ Masterarbeit
(Master's thesis)

Titel
(Title)

Evaluation of Dependency Blossoms on the Maven Central Repository

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

01.12.2025, Dortmund
Ort, Datum
(place, date)

M. Jütte
Unterschrift
(signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

01.12.2025, Dortmund
Ort, Datum
(place, date)

M. Jütte
Unterschrift
(signature)

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**