Tracking Dependencies and Release Frequency in the Maven Architecture using Neo4j and Goblin Weaver

04/01/25

Daniel Pang, Ahmed El Shatshat

UNIVERSITY OF WATERLOO | FACULTY OF MATHEMATICS

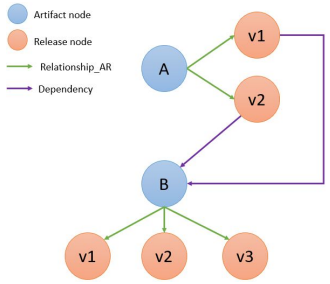
Introduction

- Dependency management is a key influencer of software architecture
  - Dependency counts may grow or shrink for a variety of reasons (new functionality, remove bloat, security risks, etc.)
- We believe minimizing dependency count is preferable, but does reality reflect ideals?
- Further, we have seen the rise of Agile and modern code review speed up releases tremendously.
- Might there be a correlation between dependency counts and release speeds?
- We conduct a case study on Maven Central to find out

UNIVERSITY OF WATERLOO | FACULTY OF MATHEMATICS

Maven and Neo4j

Neo4J



Maven

- Artifact: Libraries
- Release: Releases of Libraries
- Relationship_AR: Describes a version of a library
- Dependency: Describes what library a release depends on
- AddedValues: an extra node type with CVE , Freshness, and Popularity data. Connected to Release nodes
  - Artifacts also have one with Speed data

UNIVERSITY OF WATERLOO | FACULTY OF MATHEMATICS

Research Questions

RQ1: Has Maven trended towards software architecture that has fewer dependencies over the years?

RQ2: If Maven has trended towards fewer dependencies, has this resulted in faster software lifecycles?

RQ3: Has faster software life-cycles reduced security vulnerabilities in Maven?

UNIVERSITY OF WATERLOO | FACULTY OF MATHEMATICS

Methodology

Research Question 1 - The Baseline

- Retrieve 2000 of the earliest and latest releases
- Counts dependencies of each release
- SKIP 1000 to get the other 1000 for the 2000 total releases for each query
- Basic statistical analysis on each sample
  - mean
  - median

```
MATCH (r:Release)-[e:dependency]->(dep)
WHERE r.timestamp<1431653069000
RETURN r.timestamp, COUNT(dep),
r.id ORDER BY r.timestamp ASC
```

Listing 1: Retrieve 1000 Earliest Releases

```
MATCH (r:Release)-[e:dependency]->(dep)
WHERE r.timestamp>1700000000000
RETURN r.timestamp, COUNT(dep),
r.id ORDER BY r.timestamp DESC
```

Listing 2: Retrieve 1000 Latest Releases

Research Question 1 - The Popular Libraries

Might there be noticeable difference in trends between the more popular libraries and the general data?

- This query utilizes the AddedValue node with popularity
  - This measures number of dependents of a particular release
- Snowball sampling to choose relevant libraries
  - Ensure sufficient data

```
MATCH (pop:AddedValue)
WHERE pop.type="POPULARITY_1_YEAR"
RETURN pop.id, pop.value
ORDER BY toInteger(pop.value) DESC LIMIT 100
```

Listing 3: Retrieve 100 Most Popular Releases

Research Question 1 - Dependencies in Popular Libraries

- Having chosen popular libraries this query retrieves all important metrics for each release, given a library id
  - Dependency count
  - Release id
  - Timestamp
  - CVE count

```
MATCH (a:Artifact)-[e:relationship_AR]->(r:Release)
WHERE a.id=~"org.jetbrains.kotlin:kotlin-stdlib"
WITH a,r WHERE (r)-[f:dependency]->()
WITH count(f) AS depCount, a, r
MATCH (r)-[f:addedValues]->(g:AddedValue) WHERE g.type="CVE"
RETURN r.id, r.timestamp, depCount,
size(split(g.value,"cwe"))-1 AS CVECount
ORDER BY r.timestamp ASC
```

Listing 4: Retrieve All Releases and Relevant Metrics

Research Question 2 - The Baseline

- The goal was to compare release speeds over Maven's lifetime
- The query finds releases that have recent releases and very old releases (10+ year gap)
  - Finds the time between the first 2 releases
  - Finds the time between the last 2 releases
- Basic statistical analysis on each sample
  - mean
  - median
  - variance

```
MATCH (a:Artifact)-[er:relationship_AR]->(r:Release)
WHERE r.timestamp>1700000000000 OR
r.timestamp<1331653069000
WITH a,r ORDER BY r.timestamp DESC
WITH a, collect(r) AS releases
WHERE releases[1].timestamp>1700000000000 AND
releases[size(releases)-2].timestamp<1331653069000
RETURN a.id, releases[0].timestamp,releases[1].timestamp,
releases[size(releases)-1].timestamp,
releases[size(releases)-2].timestamp
ORDER BY releases[0].timestamp DESC
```

Listing 5: Retrieve 1000 Latest Releases With Early Releases, Return Two Oldest and Two Newest

Research Question 2 - Release Scheduling in Popular Libraries

- The former query retrieved all data already
- Find difference in timestamp between releases
  - Also noted differences in releases
  - These differences will be expanded upon in the results section

```
MATCH (a:Artifact)-[e:relationship_AR]->(r:Release)
WHERE a.id=~"org.jetbrains.kotlin:kotlin-stdlib"
WITH a,r WHERE (r)-[f:dependency]->()
WITH count(f) AS depCount, a, r
MATCH (r)-[f:addedValues]->(g:AddedValue) WHERE g.type="CVE"
RETURN r.id, r.timestamp, depCount,
size(split(g.value,"cwe"))-1 AS CVECount
ORDER BY r.timestamp ASC
```

Listing 4: Retrieve All Releases and Relevant Metrics

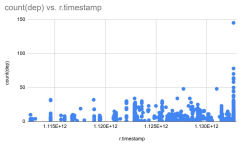
Research Question 3 - The Cutting Floor

- Unfortunately, establishing a baseline proved infeasible under our circumstances
- Further, although we had the CVE data it proved very difficult to work with
  - This will be expanded upon in the Discussion section

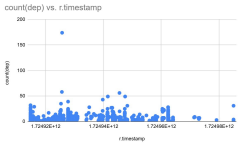
Results

Research Question 1 - The Baseline

- Generally speaking: some dependency bloat across Maven



Dependency Counts of the 2000 Earliest Releases



Dependency Counts of the 2000 Latest Releases

	Mean	Median
Earliest	7.11	5
Latests	9.1265	7

Research Question 1 - Dependencies in Popular Libraries

- No real trend among the popular libraries
  - Some increased in dependencies
  - Some decreased in dependencies
  - Each tell a different story
    - More on that in the Discussion and Threats to Validity sections

Library Name	Start	End	Start-End %	Low	High
org.apache.logging.log4j:log4j-core	13	24	185%	12	57
org.junit.jupiter:junit-jupiter-engine	10	3	-67%	3	10
org.junit.vintage:junit-vintage-engine	12	3	-75%	2	12
ch.qos.logback:logback-classic	3	20	667%	3	26
com.fasterxml.jackson.core:jackson-databind	8	12	50%	6	12
commons-io:commons-io	1	10	1000%	1	10

Table 1: Dependency Changes for Selected Libraries

Research Question 2 - The Baseline

- A clear decrease in time between releases across Maven
  - Variance is incredibly high (as to be expected) but the median also has a significant decrease

	Average	Median	Variance
Earliest	5433446102 (62.89 days)	3110888000 (36.01 days)	8.03379E+19 (Very high)
Latest	3687004220 (42.67 days)	1550125500 (17.94 days)	2.70521E+19 (Very high)
Difference	20.22 days (a factor of ~0.68)	18.07 days (a factor of ~0.49)	(Very high)

Research Question 2 - Release Speeds in Popular Libraries

- When viewing popular libraries
  - Those with decreasing dependencies
    - Most have an upwards trendline
    - One has a very marginal downward trend
  - Those with increasing dependencies had no notable trends
- Viewing the differences between major and minor releases also illustrated no general trends
  - Some had much longer times for major releases vs minor and others had the inverse

Figure 3: org.apache.logging.log4j:log4j-core

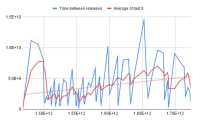


Figure 4: org.junit.jupiter:junit-jupiter-engine

Figure 5: org.junit.vintage:junit-vintage-engine

Figure 6: ch.qos.logback:logback-classic

Discussion

Research Question 1

- One would hope that dependency counts do go down to maintain code quality and security
  - In retrospect, seeing dependency bloat should have been expected
- Software systems have only become more and more complex
- Dependency management is likely technical debt that is difficult to handle
  - Even some of the more popular libraries struggle to keep dependency counts from increasing over time
- Average of minimal change in dependency counts among popular libraries illustrates an effort to combat dependency bloat

Research Question 2

- Baseline clearly reflects what is a generally empirically observed trend in industry
- However, the popular libraries have more varied behaviours amongst them
  - Some do follow this trend, while others have very different trends
- These other libraries have many minor releases that are steadily released in between major releases
  - This includes steady maintenance patches, pre-release versions, milestones, and release candidates

Research Question 3

- The structure of the data within the AddedValue nodes stores CVE data as JSON strings
  - Cypher does not work well with JSON
- This makes extracting severity and ID values difficult without additional extraction technology
- Further, it's difficult to know what nodes are relevant with CVE data for establishing a baseline

Related Work

Related Work - MSR Mining Challenge

- "Chasing the Clock: How Fast Are Vulnerabilities Fixed in the Maven Ecosystem?" - Rabbi et al.
  - Investigates software vulnerability resolution time depending on severity, library popularity as measured by number of dependents, and version release frequency
- "Faster Releases, Fewer Risks: A Study on Maven Artifact Vulnerabilities and Lifecycle Management" - Shafin et al.
  - Evaluates how release speed affects software security and lifecycle
- "On the Evolution of Unused Dependencies in Java Project Releases: An Empirical Study" - Suwanachote et al.
  - Examines how unused dependencies are introduced and removed
  - Unused packages are common (52% of projects) but releases tend not to introduce new unused dependencies (9%)
  - 59% of resolved unused dependencies are removed and 41% are later used

Related Work - Other Work

- “Streamlining Software Bloated Dependencies” - Wang et al.
  - Attempts to tackle the issue of libraries that are introduced but are unnecessary
  - Introduces a technique, Slimming, that removes bloated dependencies from projects reliably
- “Understanding the impact of rapid releases on software quality” - Khomh et al.
  - Performed numerical analysis of Mozilla Firefox releases
  - Analyzed how shorter release cycles impact crash rates, uptime, and bug rates

Release cycle timeframe and security vulnerability analysis are all hot-button topics in the field

Threats to Validity

Threats to Validity - Release Issues

- We retrieved the data ordered on a timestamp variable
  - In many cases, multiple versions of a library receive updates simultaneously
  - For example, log4j-core versions 2.3.2 and 2.12.4 were released after 2.17.1, indicating that different resources were allocated to separate releases
  - This complicates the meaning of time between releases, as updates may not be directly related
- Another challenge arises from pre-release versions, which are all labeled differently
  - This includes those marked as alpha, beta, Mx (milestone x), or RCx (release candidate x)
  - It may be preferable to exclude all pre-release versions, and possibly even minor version updates

Threats to Validity - Dependency Graph

- The structure of the dependency graph itself presents a challenge
  - No way to confirm whether a particular release or library is actively used within the Maven system
  - When a minor version update for an older major release enters the system, its continued relevance remains unclear
- Even with the augmented dataset, the AddedValue nodes only contain popularity metrics for the previous year
  - Generating a new graph with popularity metrics stretching out to the early days of Maven was beyond the scope of this study
  - It is unknown if such data is even available that far out.
- Finally, the sheer scale of the dependency graph introduces a potential for human error in data interpretation

Future Work and Conclusion

Future Work

- Primarily, addressing the notes from Threats to Validity
  - Iterating on techniques used, addressing issue in data extraction
  - Sorting on version number instead of on timestamp
- Spending more time on extraction and interpretation of CVE data
  - Would be interesting to see how the metrics we collected correlate with CVE count, CVE response time, and how these correlate with CVE severity
  - Precedent for this form of analysis from Related Work
- We did observe that popular libraries displayed unique trend behaviour
  - Would be interesting to see if there are further differences between popular and unpopular libraries
  - Comparing similar metrics done in this study, and as listed above

Conclusion

- We set out to assess and analyze dependency changes, release frequency, and security vulnerabilities in the Maven ecosystem
- Our findings challenge many of the assumptions one has about the evolution of a software system
  - Dependency counts increasing, general trend of bloated libraries performing later-stage optimizations
- While minimization of dependencies is an ideal goal, real world constraints and consequences keep us from such an ideal
- Further work should explore more finely grained metrics, improve security

UNIVERSITY OF WATERLOO

FACULTY OF MATHEMATICS

YOU+WATERLOO

Our greatest impact happens together.