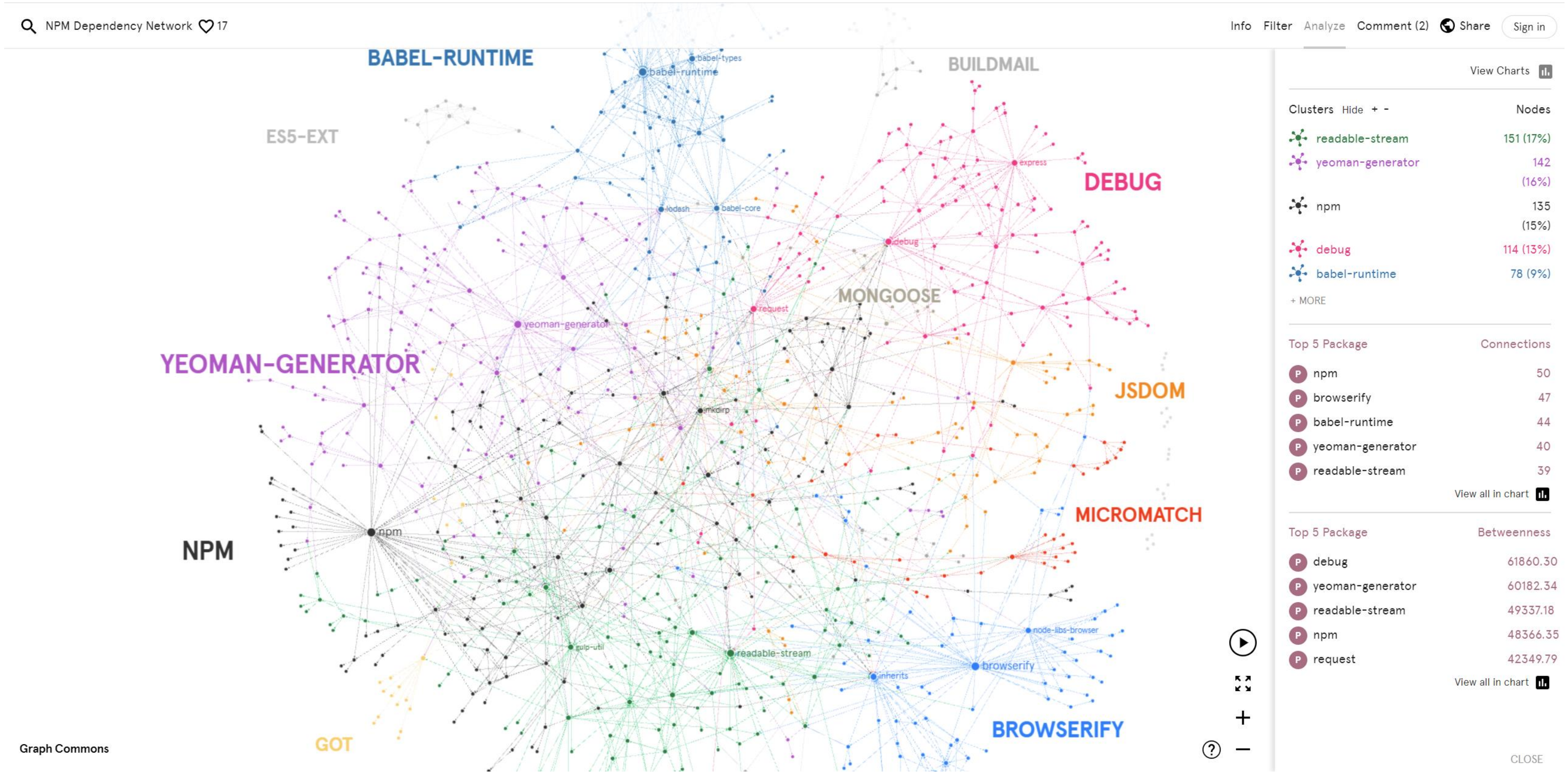
Lab 5: Dependency Management

何昊

2022.11.02

The "Software Ecosystem" in Open Source



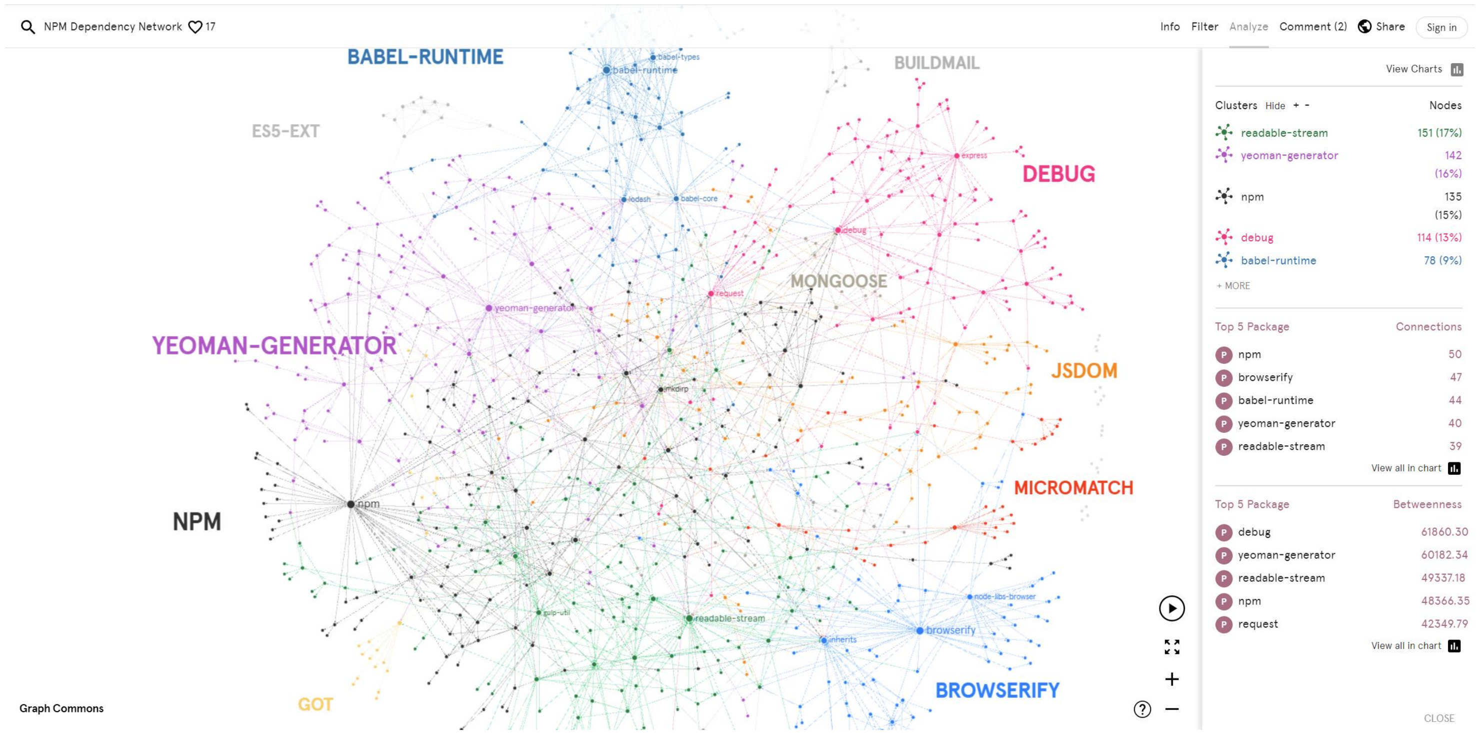
Clusters	Nodes
readable-stream	151 (17%)
yeoman-generator	142 (16%)
npm	135 (15%)
debug	114 (13%)
babel-runtime	78 (9%)

Top 5 Package	Connections
npm	50
browserify	47
babel-runtime	44
yeoman-generator	40
readable-stream	39

Top 5 Package	Betweenness
debug	61860.30
yeoman-generator	60182.34
readable-stream	49337.18
npm	48366.35
request	42349.79

https://graphcommons.com/graphs/a7ec343d-2a0c-47bb-9658-bb8315e8a096?auto=true&show=analysis-cluster

The Reuse of Existing Software



View Charts

Clusters Hide + -	Nodes
readable-stream	151 (17%)
yeoman-generator	142 (16%)
npm	135 (15%)
debug	114 (13%)
babel-runtime	78 (9%)

+ MORE

Top 5 Package	Connections
npm	50
browserify	47
babel-runtime	44
yeoman-generator	40
readable-stream	39

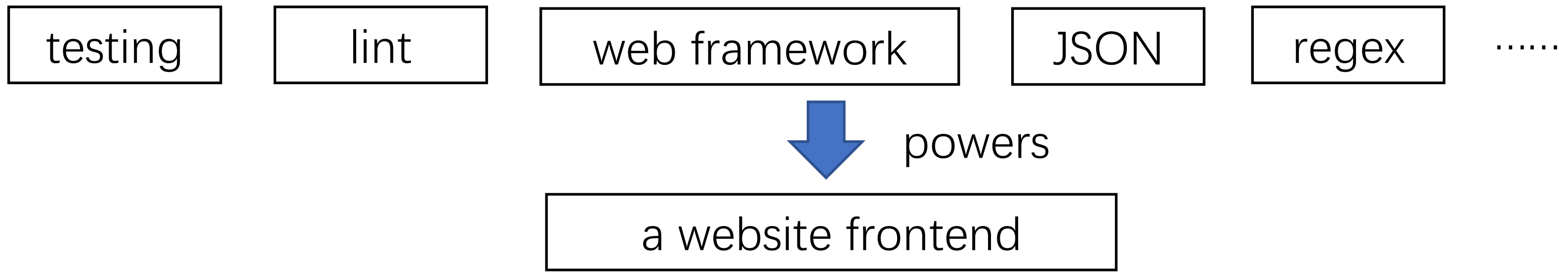
View all in chart

Top 5 Package	Betweenness
debug	61860.30
yeoman-generator	60182.34
readable-stream	49337.18
npm	48366.35
request	42349.79

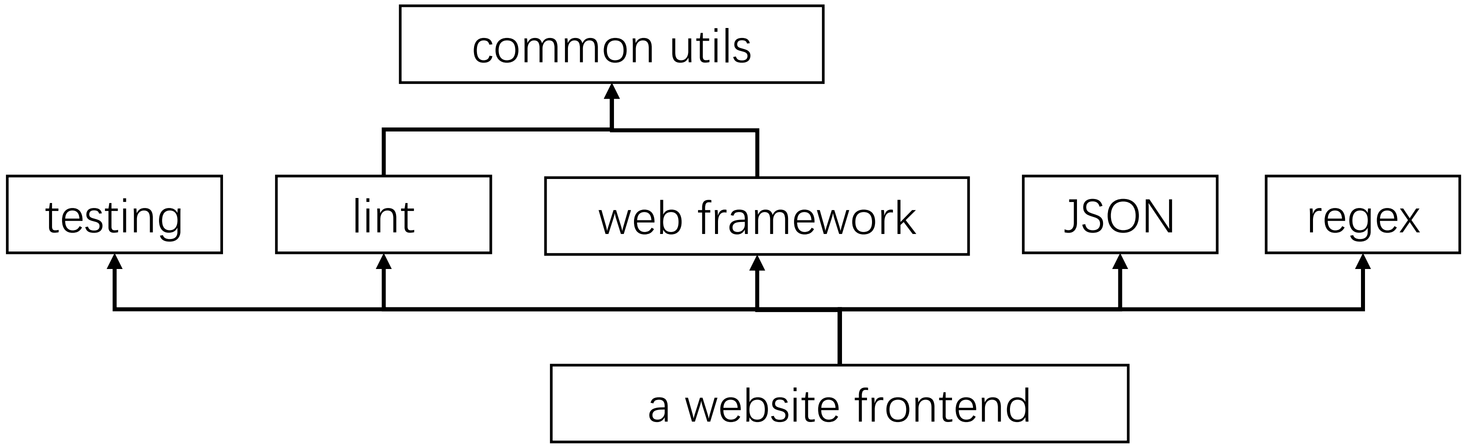
View all in chart

CLOSE

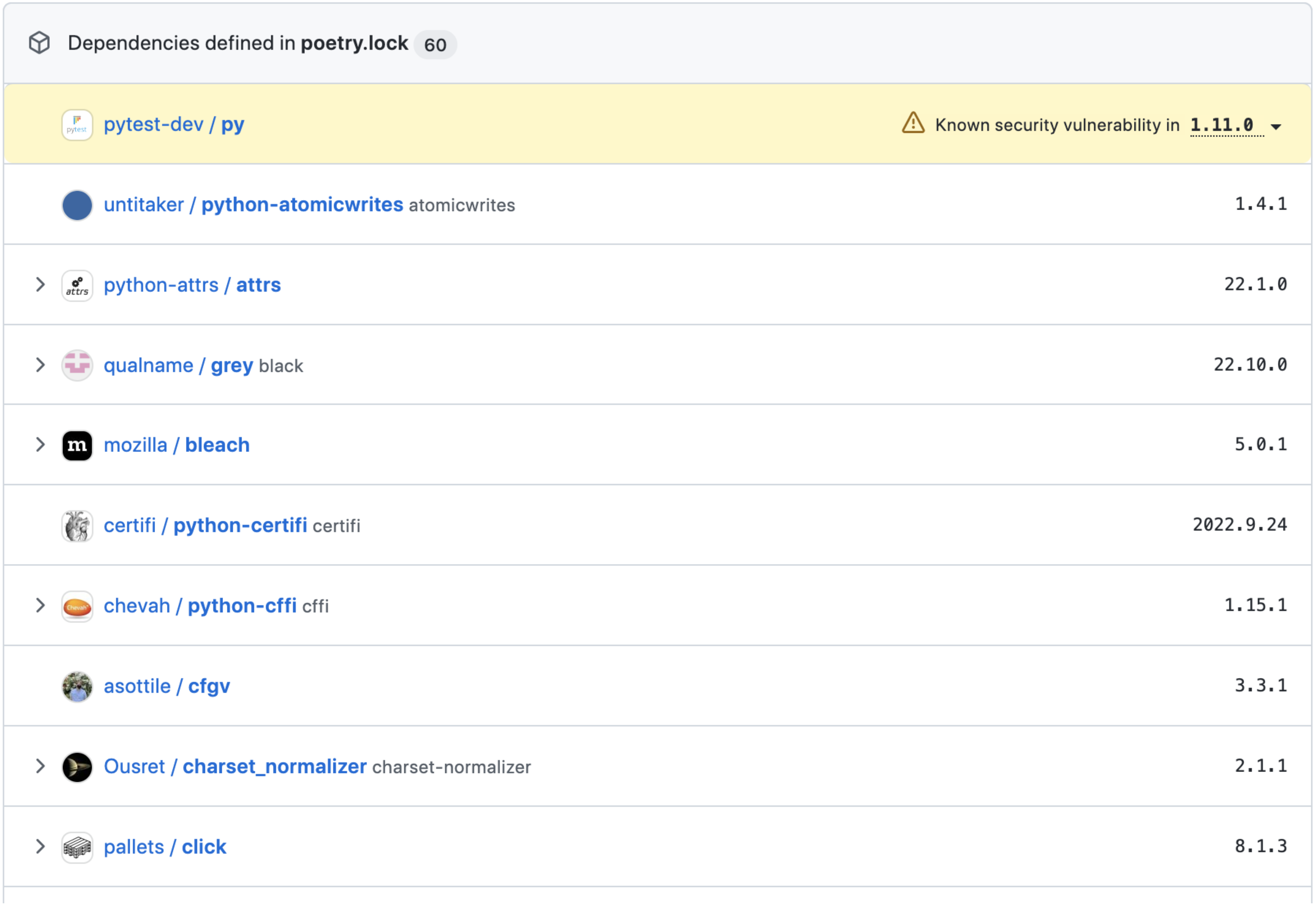
- Stand on the shoulder of giants
  - …your project is not going to anywhere if all your newbies need to reimplement a JSON parser or regular expression engine…



A “Dependency Network” That Needs to be “Managed”



- Manage a complex network (DAG) for that to work well on your project
  - Adoptions
  - Version management
  - Deprecations
- A software engineering problem, **not** a programming problem
  - Trivial if scale is small and timespan is short
  - Becomes increasingly complex with scale and time


Dependencies defined in **poetry.lock** 60

pytest-dev / **py**	⚠ Known security vulnerability in **1.11.0**
untitaker / **python-atomicwrites** atomicwrites	1.4.1
python-attrs / **attrs**	22.1.0
qualname / **grey** black	22.10.0
mozilla / **bleach**	5.0.1
certifi / **python-certifi** certifi	2022.9.24
chevah / **python-cffi** cffi	1.15.1
asottile / **cfgv**	3.3.1
Ousret / **charset_normalizer** charset-normalizer	2.1.1
pallets / **click**	8.1.3

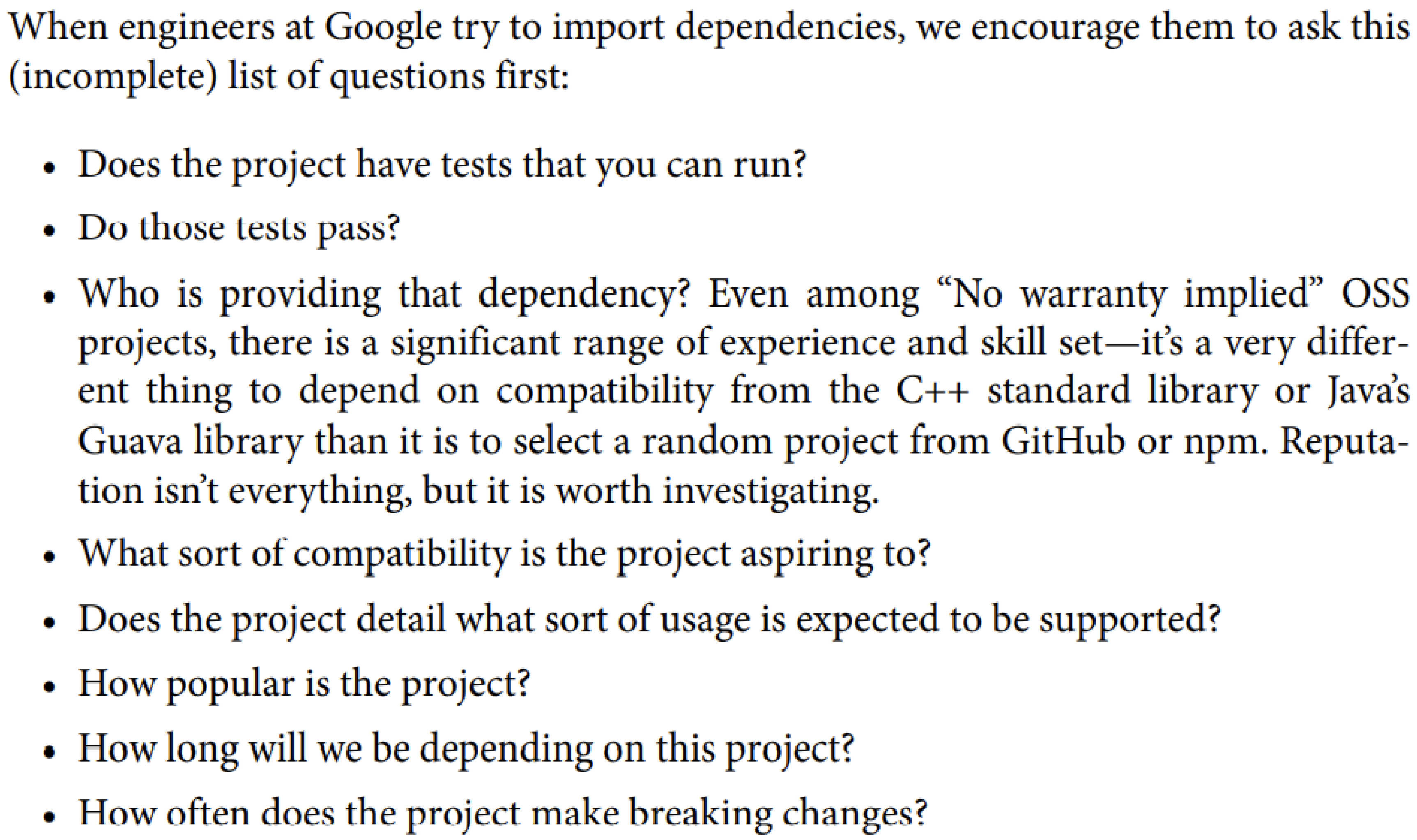
Important Notes About Dependency Management

- Dependency management—the management of networks of libraries, packages, and dependencies that we don’t control—is one of the least understood and most challenging problems in software engineering

- We don’t have first-hand evidence of solutions that work well across organizations at scale…if we could, we wouldn’t be calling this one of the most important problems in software engineering

Software Engineering at Google, Chapter 21

Be Careful on Which Software to Depend on

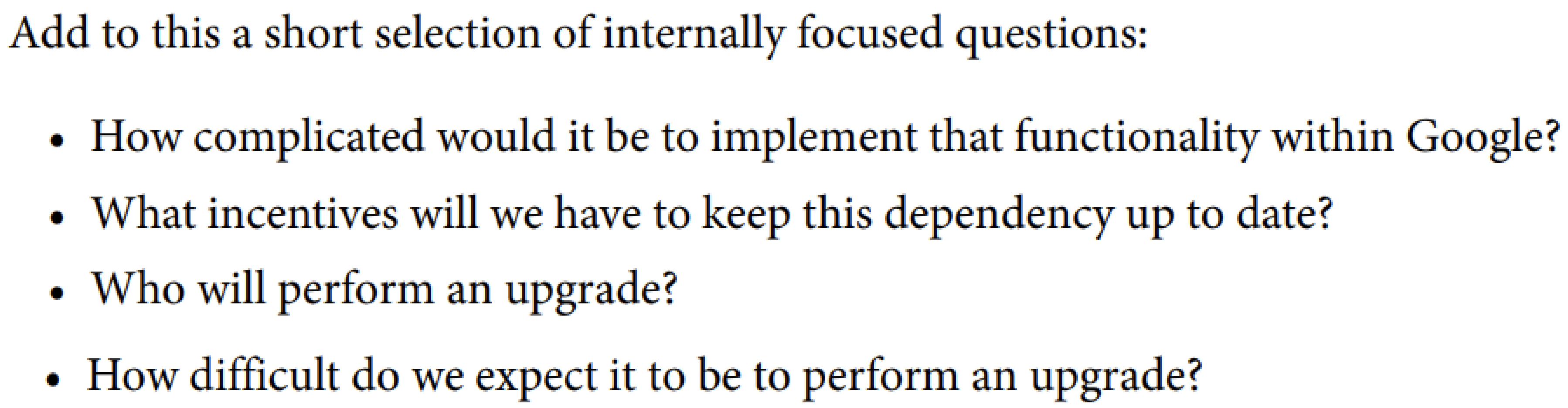


When engineers at Google try to import dependencies, we encourage them to ask this (incomplete) list of questions first:

- Does the project have tests that you can run?
- Do those tests pass?
- Who is providing that dependency? Even among "No warranty implied" OSS projects, there is a significant range of experience and skill set—it's a very different thing to depend on compatibility from the C++ standard library or Java's Guava library than it is to select a random project from GitHub or npm. Reputation isn't everything, but it is worth investigating.
- What sort of compatibility is the project aspiring to?
- Does the project detail what sort of usage is expected to be supported?
- How popular is the project?
- How long will we be depending on this project?
- How often does the project make breaking changes?

- Key Attributes
  - Quality
  - Popularity
  - Reputation
  - Evolution Strategy
  - Responsibility
  - Legal Concerns
  - …

Be Careful on Which Software to Depend on



Add to this a short selection of internally focused questions:

- How complicated would it be to implement that functionality within Google?
- What incentives will we have to keep this dependency up to date?
- Who will perform an upgrade?
- How difficult do we expect it to be to perform an upgrade?

- Key Attributes
  - Quality
  - Popularity
  - Reputation
  - Evolution Strategy
  - Responsibility
  - Legal Concerns
  - …

Four Strategies of Dependency Management

- Nothing ever changes!
  - You never change the version of your dependencies
  - The starting point of almost all software projects: you don’t know how long they are going to live beforehand – can be two weeks or two decades
  - Over a long time period, this strategy is not sustainable
    - Security vulnerabilities
    - Version conflicts
  - Technical debt accumulates very fast
    - The cost of upgrade can grow exponentially with time
  - Examples:
    - Your course projects

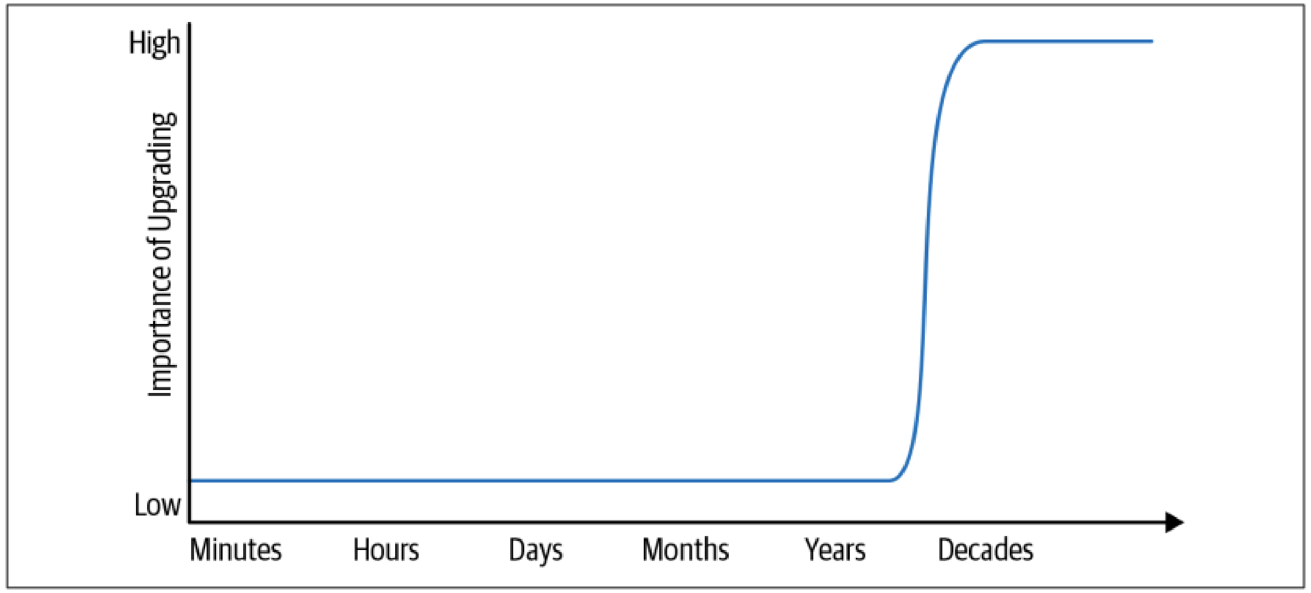


Figure 1-1. Life span and the importance of upgrades

Four Strategies of Dependency Management

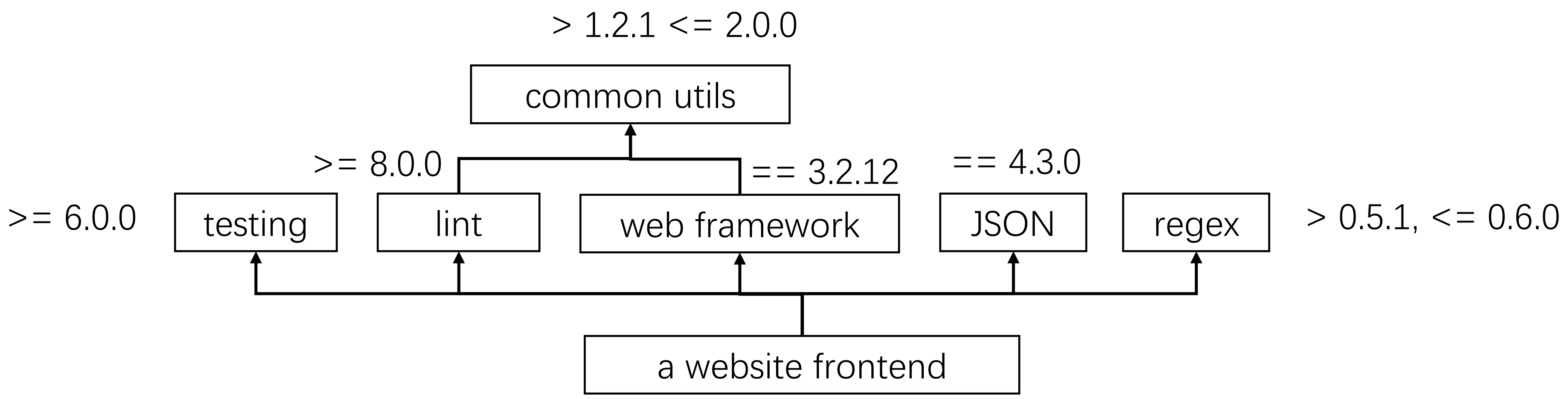
- Semantic Versioning
  - Upstream dependencies declare versions in the following format

$$Major.Minor.Patch$$

  - Increment Major if there are breaking changes in a new release
  - Increment Minor if there are new functionalities in a new release
  - Increment Patch if only bug fixes in a new release
  - Minor and Patch upgrades should be compatible
  - Software projects declare a loose constraint for automated upgrades
    - $>=$ 1.2.1, $<$ 2.0.0
  - Use a tool to find a compatible version set given a set of compatibility constraints
  - https://research.swtch.com/version-sat proof of NP-Completeness
  - Example:
    - npm, PyPI, …
    - A “loosely adopted” best practice for almost all open source ecosytems

Four Strategies of Dependency Management

- Example of Semantic Versioning



- Resolves into
  - web framework == 3.2.12, JSON == 4.3.0, regex == 0.5.19, common-utils == 1.8.9, lint == 8.0.19, testing == 8.2.19

Four Strategies of Dependency Management

- Limitations of Semantic Versioning
  - **Over-constraint**
    - You may break an API not actually used by anyone
  - **Under-constraint**
    - You can break someone's code with even the most trivial changes
  - **Decisions are local**
    - It is hard to know how your APIs are used
  - **Hyrum's Law:**

With a sufficient number of users of an API,
it does not matter what you promise in the contract:
all observable behaviors of your system
will be depended on by somebody.

Four Strategies of Dependency Management

- Bundled Distribution Model
  - Since integration of many pieces of software are so difficult…
  - Some people curate a tested set of software and bundle the distribution (distributor)
  - Sounds great, but only applicable to a specific application scenario
    - Operating system,
    - Cloud computing,
    - …
  - The distributor needs a commercial model to sustain itself
- Examples
  - Linux Distributions: Ubuntu, CentOS, …

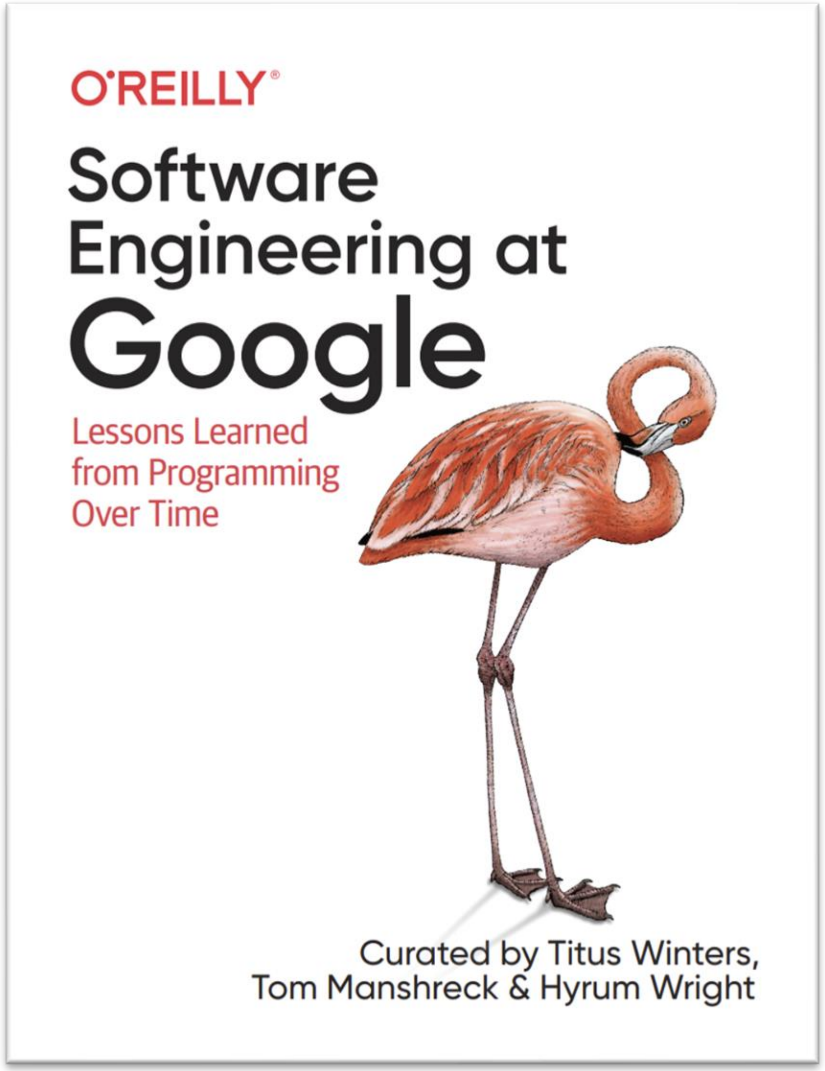
Four Strategies of Dependency Management

- Live at Head
  - Keep all versions to the latest one, or remove versions at all
  - Costly, but scales with an exceptional development infrastructure
- Example:
  - Google does not adopt any dependency management for internal dependencies
  - All projects must keep one and only one version, and ensures that it does not break others
  - Monorepo
  - The dependency management problem is reduced to the easier source control problem

The (Ideal) Dependency Management

- Theoretically,
  - Build the infrastructure necessary for analyzing upstream/downstream compatibility
    - ecosystem level call graph
    - ecosystem level dependency graph
    - source code query language
    - large scale CI/CD
    - security vulnerability monitoring
  - For each specific dependency network
    - Use all the infrastructure to automatically identify dependency issues
- However,
  - The infrastructure for enabling this is lacking in open source
  - Ongoing effort
    - GitHub Dependabot, CodeQL, Dependency Graph, …

Suggested Reading

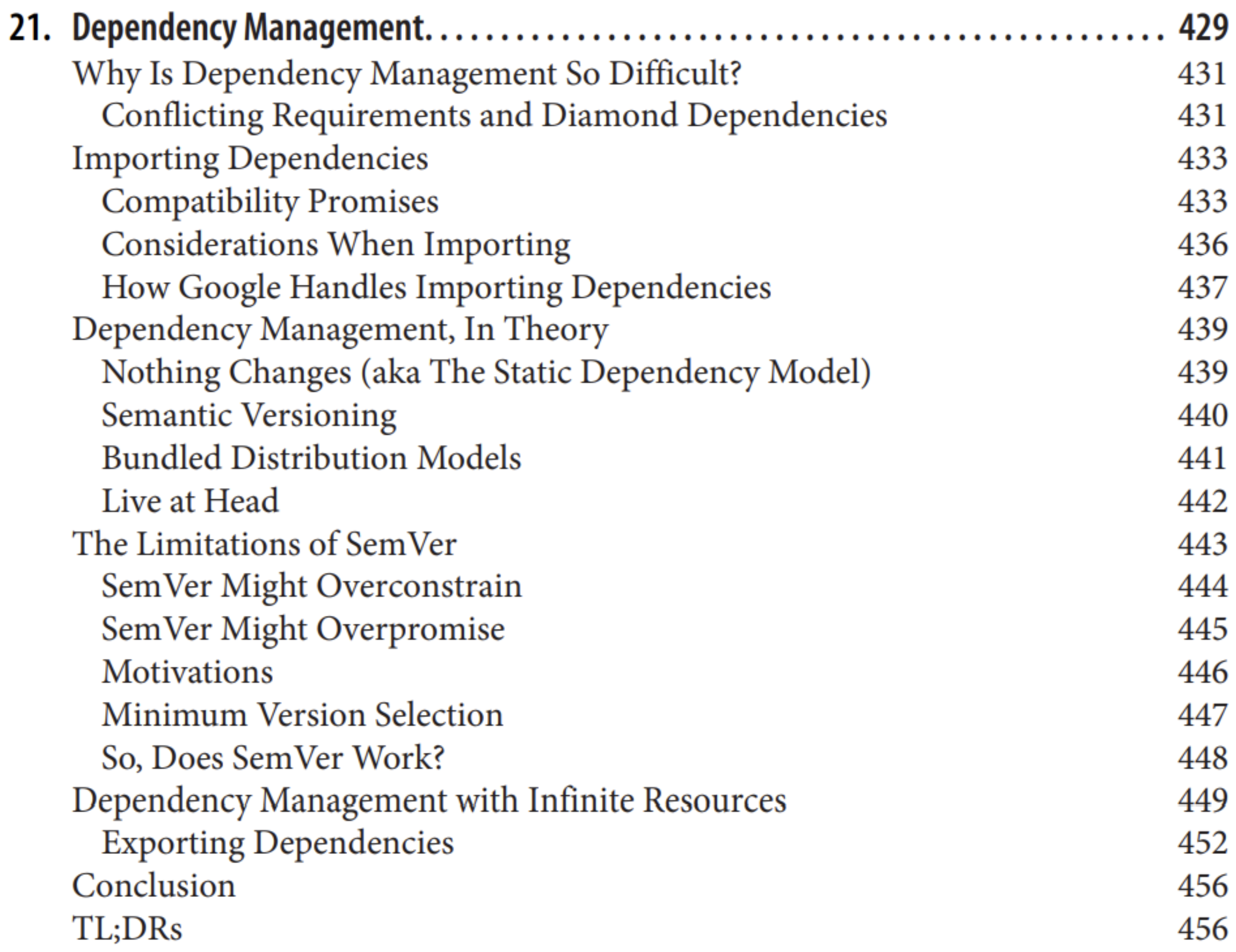


O'REILLY

Software Engineering at Google

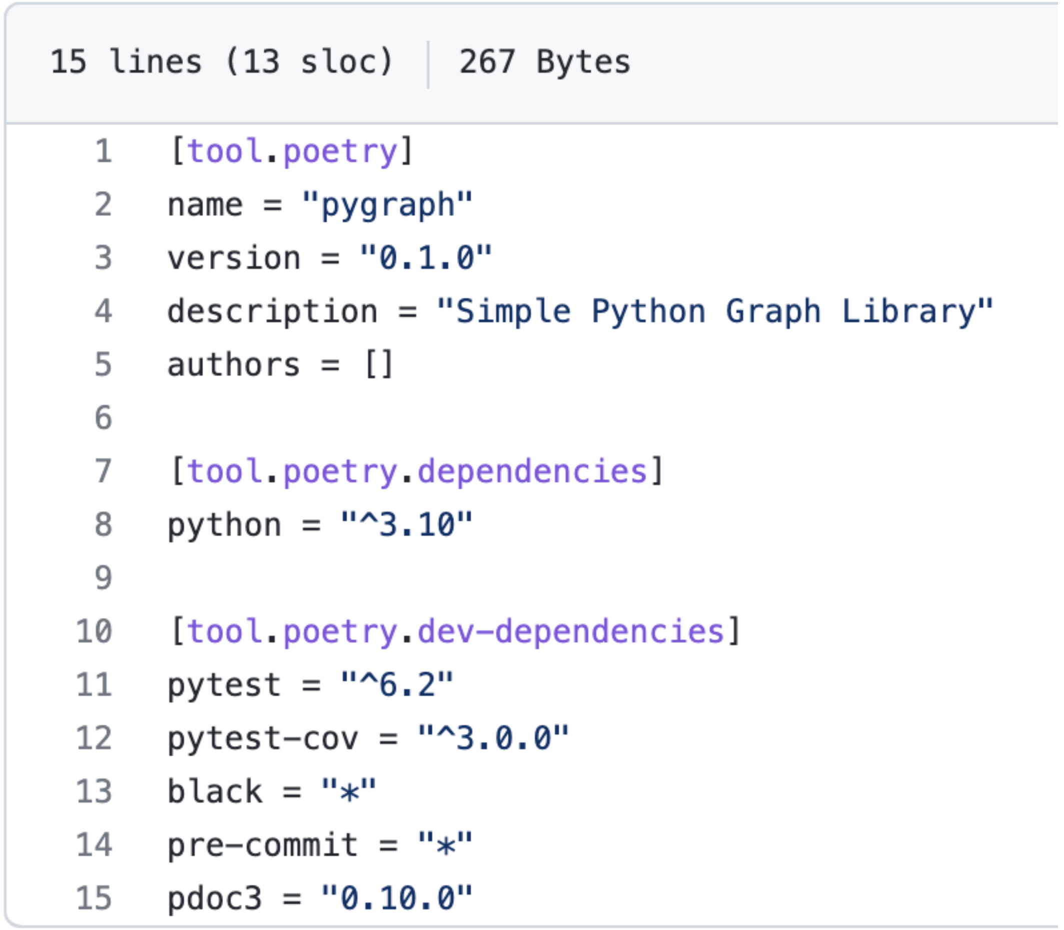
Lessons Learned from Programming Over Time

Curated by Titus Winters, Tom Manshreck & Hyrum Wright



Lab 5: Python开源软件包安装/管理/打包/上载

- Lab 4中已经提供了依赖配置文件pyproject.toml
- 本次Lab的任务：
  - 将pygraph打包和安装在本地系统
  - 将pygraph发布在TestPyPI
  - 配置CI实现向TestPyPI的自动发布
  - 添加依赖实现新功能



15 lines (13 sloc) | 267 Bytes

```toml
[tool.poetry]
name = "pygraph"
version = "0.1.0"
description = "Simple Python Graph Library"
authors = []

[tool.poetry.dependencies]
python = "^3.10"

[tool.poetry.dev-dependencies]
pytest = "^6.2"
pytest-cov = "^3.0.0"
black = "*"
pre-commit = "*"
pdoc3 = "0.10.0"
```