An aerial photograph of terraced rice fields at sunset. The fields are filled with water, reflecting the orange and yellow light of the setting sun. The terraces are arranged in a grid-like pattern, with some fields being larger than others. The surrounding area is lush with green trees and vegetation. The overall scene is peaceful and scenic.

Breaking the  
chain: securing  
dependencies  
one by one.

MAURO PARRA  
@MAUROP



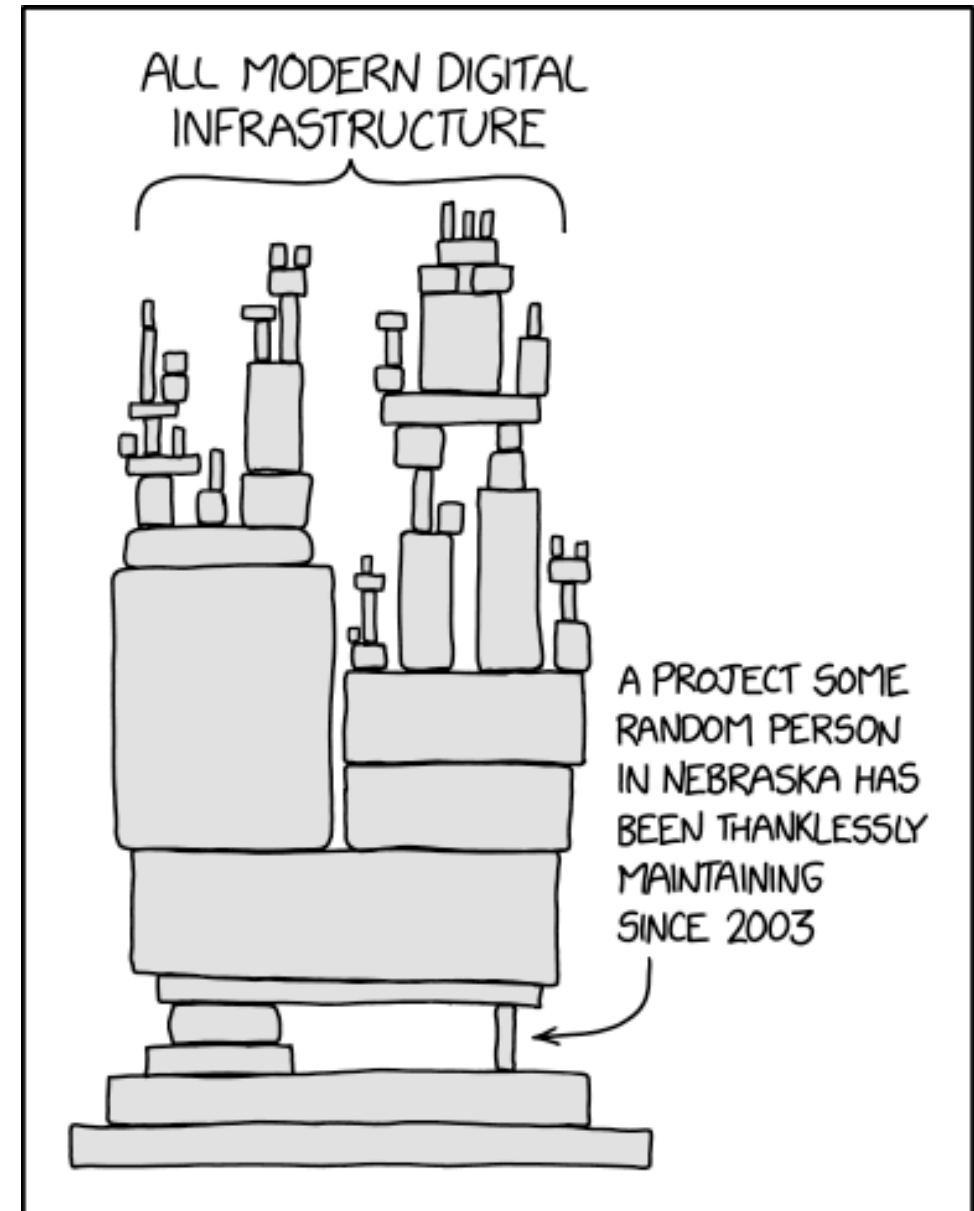
---

# Problem – Software supply chain

- **Security** – You have few dozens explicit dependencies, and hundreds/thousands of implicit ones. You want to make sure you don't have vulnerabilities, specially from implicit dependencies unknown to you. Tampering, Malicious Code, Dependencies, Access Control, Incident Response.
- **Intellectual property value** – Are you the owner of your software product? Or just partial owner + other hundred coders that contribute via public code? Software Bill of Materials (SBOM) is a list of the components and modules that make up a software application, along with the supply chain relationships between them. Licensing, copyrights, patents, risk of lawsuits.
- **Compliance side** – Are you able to tell your customers, your board, your government if the code you are redistributing or using is complaint with all different rules? Think: banks, hospitals, insurance.

# Problem – Software supply chain

- **Business Efficiency** – Is npm down? Did someone at github get mad and delete their code? Are you resolving your dependencies at run time? Compile time? Do you depend on the internet? Do you have a connection at all?
- **QA** – Are you doing QA at all? Do you have unit testing? Do you have functional testing? If any dependency changes of version, are you ready to test the change? Is it possible at all?



# “God is in the details.”

---

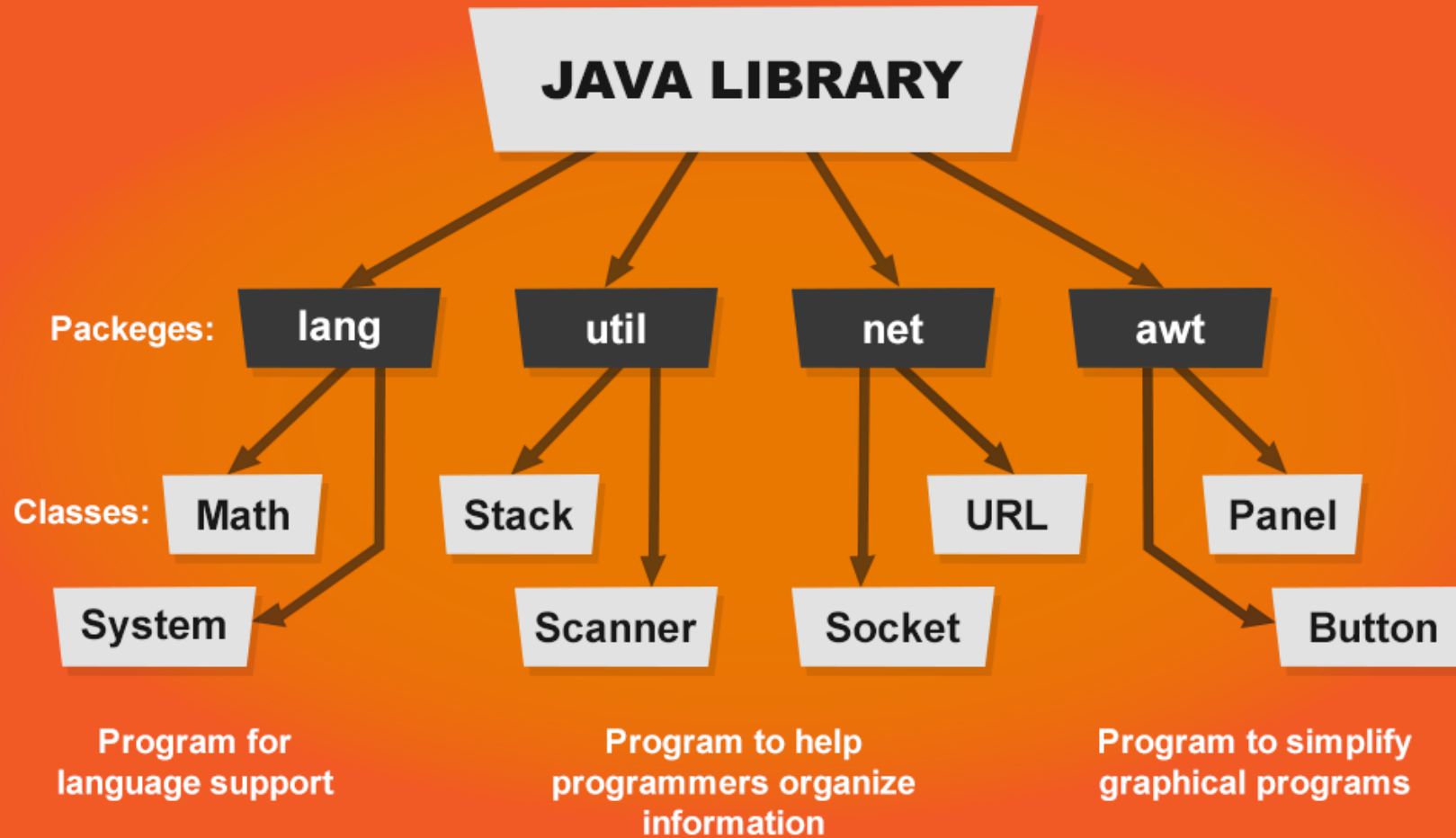
—Ludwig Mies van der Rohe



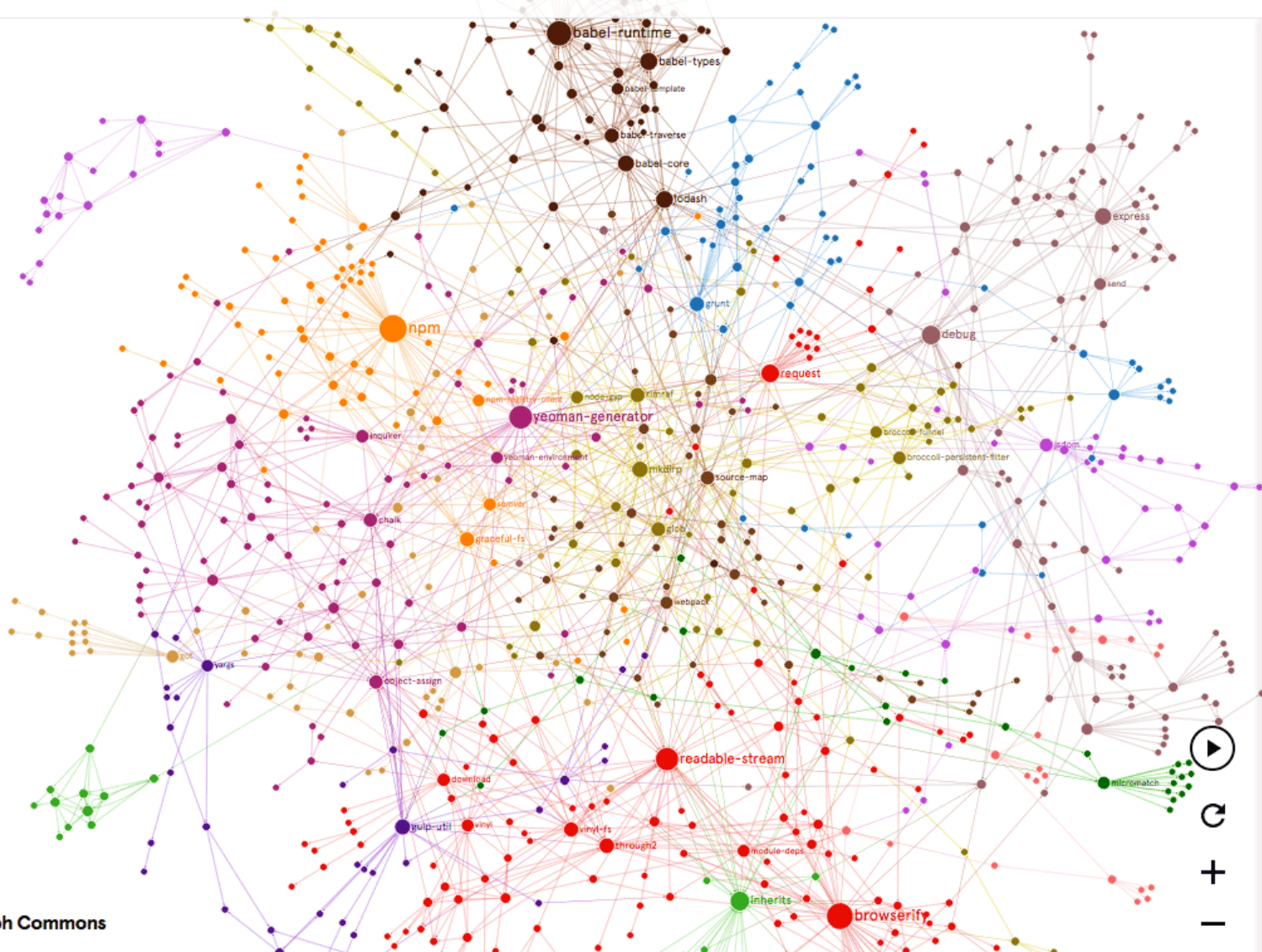
LESS IS MORE.

```
File Edit Search Run Compile Debug Tools Options Window Help
[ ] SPARKLE.PAS 1=[ ]
Program Sparkle;
Uses Crt;
Var
  Counter, X, Y, Char, Color : Integer;
Begin
  Clrscr;
  Textbackground(Black);
  Randomize;
  For Counter := 1 to 1000 do
  Begin
    X := Random(80);
    Y := Random(25);
    Color := Random(7);
    Char := Random(4);
    Gotoxy(X,Y);
    If Color = 0 Then Textcolor(Red);
    If Color = 1 Then Textcolor(Green);
    If Color = 2 Then Textcolor(Blue);
    If Color = 3 Then Textcolor(Yellow);
    If Color = 4 Then Textcolor(Cyan);
    If Color = 5 Then Textcolor(Magenta);
  End
  End
1:1
```

Turbo Pascal  
1983+



Java  
1995+



Graph Commons

# Analysis

Edit Analysis Blocks

CLUSTERS	Tips	Nodes
	Cluster of browserify	120
	Cluster of debug	98
	Cluster of yeoman-generator	94
	Cluster of mkdirp	72
	Cluster of babel-runtime	72

+ MORE  
Hide Clusters Run clustering

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
-	

CLOSE

NodeJS  
2009+



“The quality of  
your life is built  
on the quality  
of your  
decisions”

---

- WESAM FAWZI





---

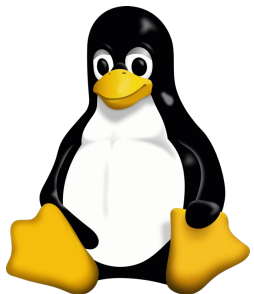
# OS issues – Package managers (yum, apt, dnf)

- **Dependency Hell:** When installing a package, it might require specific versions of other packages that conflict with the versions already installed.

*Example:* Installing PackageA might need libX v1.5, but PackageB already uses libX v1.3. Upgrading or downgrading libX to satisfy one package could break the other.

- **Circular Dependencies:** Two or more packages depend on each other, creating a loop that the package manager cannot resolve.

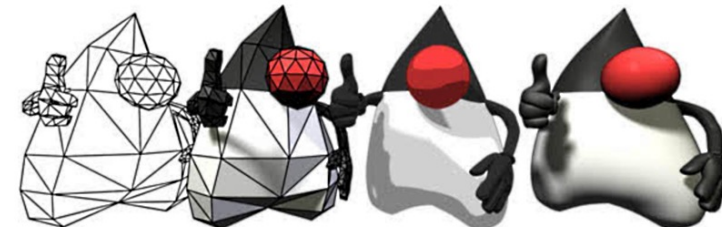
*Example:* PackageA depends on PackageB, and PackageB depends on PackageA.



---

# Java – Package managers (maven, gradle)

- **Transitive Dependency Conflicts:** A library might depend on a version of another library that conflicts with a different dependency in the project.  
*Example:* LibraryA depends on LibraryC v1.0, while LibraryB depends on LibraryC v2.0. The project cannot resolve which version of LibraryC to use without explicit configuration.
- **Dependency Bloat:** Adding one library can bring in unnecessary transitive dependencies, increasing the project's size and complexity.



---

# NodeJS – Special horrible corner case – npm

- **Nested Dependencies:** *NPM allows multiple versions of the same package to coexist.*  
This can lead to bloated node\_modules directories and hard-to-debug issues.  
*Example:* PackageX depends on PackageY v1.0, but another library in the project depends on PackageY v2.0. Both versions are installed, potentially causing runtime errors.
- **Breaking Changes in Semver:** Developers sometimes release major changes under minor version updates, violating Semantic Versioning rules and causing projects to break unexpectedly after updates.  
*Example:* Updating PackageZ to a patch version introduces an API change that breaks your application





---

# Python – pkg manager (pip, conda)

- **Version Locking Issues:** Installing packages without specifying versions can lead to breaking changes when dependencies are updated.  
*Example:* A requirements.txt file specifies pandas but doesn't pin the version, and an incompatible update to pandas later breaks the application.
- **Incompatible Environment:** Different Python environments or versions of the interpreter (e.g., Python 2 vs. Python 3) can cause dependencies to fail to install or run correctly.



---

# Ruby – pkg manager (bundle)

- **Gem Dependency Conflicts:** Similar to Maven or NPM, a Gem might require a specific version of a dependency that conflicts with another Gem in the project. *Example:* GemA requires Rails v6.0, but GemB requires Rails v5.2, creating an incompatibility.



---

# Containerization – docker

- **Image Layer Conflicts:** Base images and application layers may include different versions of the same software, leading to runtime errors.  
*Example:* A Docker image uses a specific version of glibc that is incompatible with a binary included in the image.
- **Version Drift:** Without version pinning, a Dockerfile can produce different images over time as newer versions of dependencies are installed.



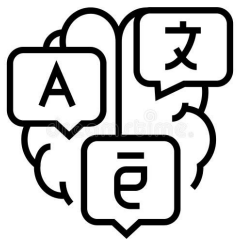


---

# Cross-Lang Issues

- **Polyglot Projects:** When a project uses multiple languages (e.g., Python for backend and Node.js for frontend), each language's dependency manager might bring in conflicting dependencies or libraries that don't work well together.

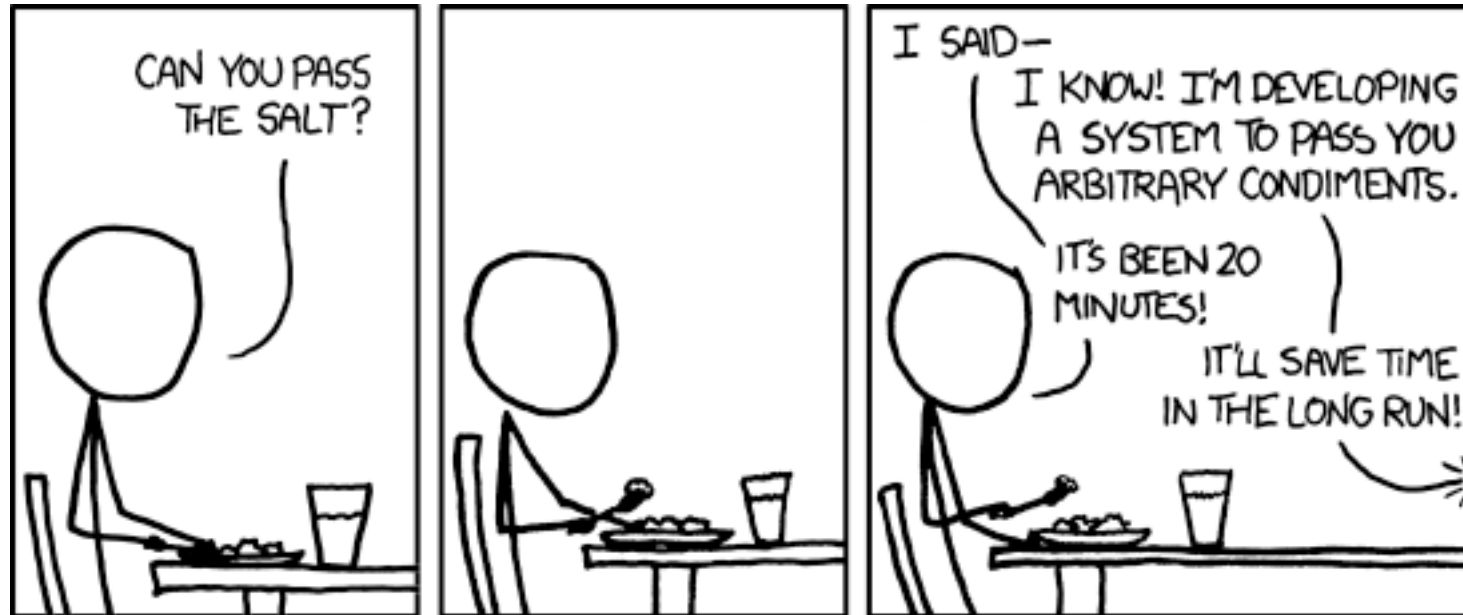
*Example:* A Python script calls a Node.js library, and an underlying C++ binary dependency conflicts with the OS package version.



**SOLUTIONS**

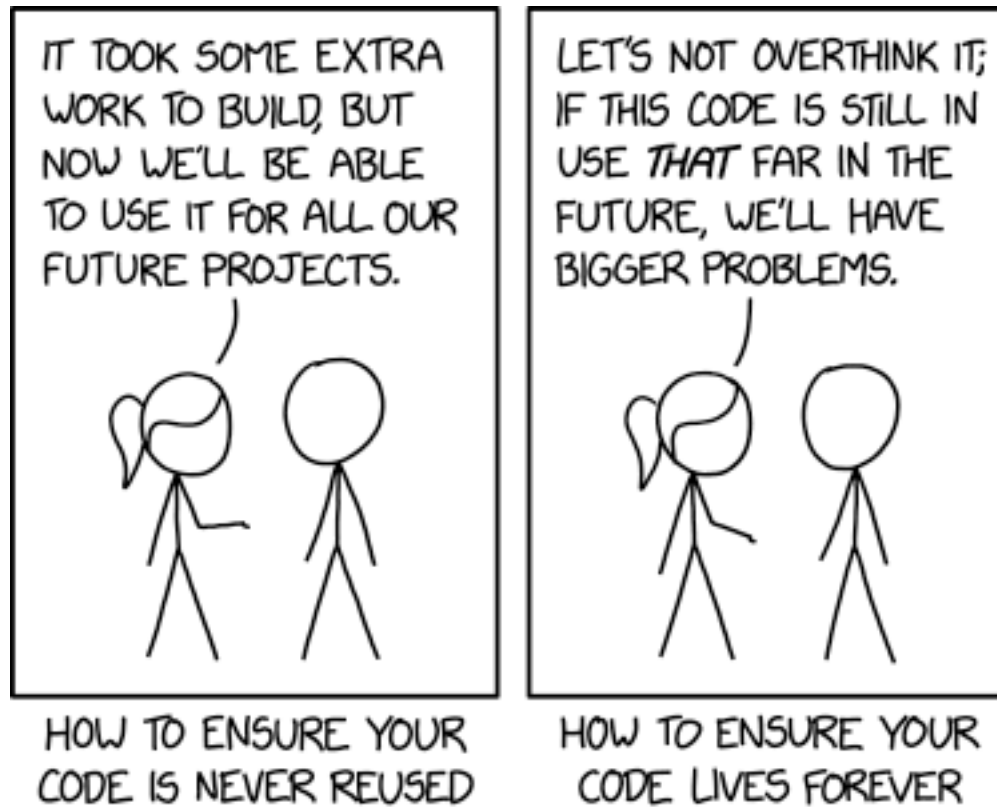
**SOLUTIONS EVERYWHERE**


# Over engineering.





# Reuse public code





## (imperfect) solutions

- **Version Locking** – (e.g., `package-lock.json`, `Gemfile.lock`, or `requirements.txt`).
- **Dependency resolution tools** – Use Maven Enforcer or `pip-tools`.
- **Audit the PR!** – Check for issues in the PR. Take this shit seriously. Test in a clean environment. Follow the CI/CD path and add unit testing as pre-requisite to accept the code.
- **Use `venvs`** – Use virtual environments (e.g., Python's `venv`) or containerization (e.g., Docker) to isolate dependencies. Run unit tests! Perform at least some smoke testing.
- **Automate all the things!** – Automate testing for compatibility with updated dependencies using CI/CD pipelines with built-in basic testing.

**COMPILED WITHOUT ERRORS..**



**DOES IT WORK?**



---

# Prepare for (imperfect) solutions

- Use a healthy Software Development Lifecycle – Create dev, test, staging/pre-prod, prod environments.
- Automate testing – Unit testing, functional testing, UI testing.
- Have a plan for failure – Rollback is not a nice to have, it's a hard requirement!
- Create proper logging and use tools to (easily) analyze the logs.
- Be prepared to generate a list of dependencies so you can quickly answer if you are affected or not to something (remember log4j).

**Your next task is to  
figure out which applications  
in your org use log4j**



# Building Secure & Reliable Systems


Best Practices for Designing, Implementing  
and Maintaining Systems



Heather Adkins, Betsy Beyer  
Paul Blankinship, Piotr Lewandowski  
Ana Oprea & Adam Stubblefield

## Tools

- Syft – A CLI tool and Go library for generating a Software Bill of Materials (SBOM) from container images and filesystems. Exceptional for vulnerability detection when used with a scanner like Grype. – <https://github.com/anchore/syft>
- Gitlab Scans – Gitlab offers security scans built-in your pipeline – see demo: <https://gitlab.navattic.com/gitlab-scans>
- Verdaccio – Your own npm registry, so you don't depend on nodejs world politics! – <https://verdaccio.org/>
- Building secure & reliable systems – <https://google.github.io/building-secure-and-reliable-systems/raw/toc.html>



# FUTURE WORK

WHAT ARE YOU WORKING ON?

TRYING TO FIX THE PROBLEMS I  
CREATED WHEN I TRIED TO FIX  
THE PROBLEMS I CREATED WHEN  
I TRIED TO FIX THE PROBLEMS  
I CREATED WHEN...

