Lecture 19: Dependency management

CS 5150, Spring 2026

Administrative Reminders

- Sign up for final presentation slot (see Canvas)
- Report #3 due soon (Apr 27)
- Some confusion about user study: If you have already conducted the user study, feel free to include in Report #3. Otherwise, defer to Final Report.
- Client meetings: Keep meeting your client even in last sprint (remind them to update meeting sheets)

Lecture goals

- Manage application dependencies and associated risks

Dependencies

Poll: PollEv.com/cs5150sp26

Ubuntu 26.04 LTS (long-term support) was released last week. Should you upgrade your Linux (virtual) machines before the end of the semester?

We are all familiar with this dependency jungle…

```
(tmp.w1c9uFxoRF) cayla@run:~$ pip-compile -r -vv
Using indexes:
  https://pypi.org/simple

                        ROUND 1
Current constraints:
  django==1.11.16
  django-haystack==2.5.0

Finding the best candidates:
  found candidate django==1.11.16 (constraint was ==1.11.16)
  found candidate django-haystack==2.5.0 (constraint was ==2.5.0)

Finding secondary dependencies:
  django-haystack==2.5.0 not in cache, need to check index
  django-haystack==2.5.0    requires Django<1.10
  django==1.11.16 not in cache, need to check index
  django==1.11.16           requires pytz

New dependencies found in this round:
  adding [u'django', '<1.10', '[]']
  adding [u'pytz', '', '[]']
Removed dependencies in this round:
Unsafe dependencies in this round:
------------------------------------------------------------
Result of round 1: not stable

                        ROUND 2
Current constraints:
  django<1.10,==1.11.16
  django-haystack==2.5.0
  pytz

Finding the best candidates:
Could not find a version that matches django<1.10,==1.11.16
Tried: 1.1.3, 1.1.4, 1.2, 1.2.1, 1.2.2, 1.2.3, 1.2.4, 1.2.5, 1.2.6, 1.2.7, 1.3, 1.3.1, 1.3.2, 1.3.3,
4, 1.3.5, 1.3.6, 1.3.7, 1.4, 1.4.1, 1.4.2, 1.4.3, 1.4.4, 1.4.5, 1.4.6, 1.4.7, 1.4.8, 1.4.9, 1.4.10,
1, 1.4.12, 1.4.13, 1.4.14, 1.4.15, 1.4.16, 1.4.17, 1.4.18, 1.4.19, 1.4.20, 1.4.21, 1.4.22, 1.5, 1.5.
5.2, 1.5.2, 1.5.3, 1.5.4, 1.5.5, 1.5.6, 1.5.7, 1.5.8, 1.5.8, 1.5.9, 1.5.10, 1.5.11, 1.5.12, 1.5.12,
1.6, 1.6.1, 1.6.1, 1.6.2, 1.6.2, 1.6.3, 1.6.3, 1.6.4, 1.6.4, 1.6.5, 1.6.5, 1.6.6, 1.6.6, 1.6.7, 1.6.
6.8, 1.6.8, 1.6.9, 1.6.9, 1.6.10, 1.6.10, 1.6.11, 1.6.11, 1.7, 1.7, 1.7.1, 1.7.1, 1.7.2, 1.7.2, 1.7.
7.3, 1.7.4, 1.7.4, 1.7.5, 1.7.5, 1.7.6, 1.7.6, 1.7.7, 1.7.7, 1.7.8, 1.7.8, 1.7.9, 1.7.9, 1.7.10, 1.7
1.7.11, 1.7.11, 1.8, 1.8, 1.8.1, 1.8.1, 1.8.2, 1.8.2, 1.8.3, 1.8.3, 1.8.4, 1.8.4, 1.8.5, 1.8.5, 1.8.
8.6, 1.8.7, 1.8.7, 1.8.8, 1.8.8, 1.8.9, 1.8.9, 1.8.10, 1.8.10, 1.8.11, 1.8.11, 1.8.12, 1.8.12, 1.8.1
8.13, 1.8.14, 1.8.14, 1.8.15, 1.8.15, 1.8
, 1.9, 1.9.1, 1.9.1, 1.9.2, 1.9.2, 1.9.3,
1.9.8, 1.9.8, 1.9.9, 1.9.9, 1.9.10, 1.9.1
.10.1, 1.10.1, 1.10.2, 1.10.2, 1.10.3, 1
10.7, 1.10.8, 1.10.8, 1.11, 1.11, 1.11.1
 1.11.5, 1.11.6, 1.11.6, 1.11.7, 1.11.7, 1.11.8, 1.11.8, 1.11.9, 1.11.9, 1.11.10, 1.11.10, 1.11.11,
11, 1.11.12, 1.11.12, 1.11.13, 1.11.13, 1.11.14, 1.11.14, 1.11.15, 1.11.15, 1.11.16, 1.11.16
Skipped pre-versions: 1.8a1, 1.8b1, 1.8b2, 1.8rc1, 1.9a1, 1.9b1, 1.9rc1, 1.9rc2, 1.10a1, 1.10a1, 1.1
1.10b1, 1.10rc1, 1.10rc1, 1.11a1, 1.11b1, 1.11rc1, 1.11rc1
There are incompatible versions in the resolved dependencies.
(tmp.w1c9uFxoRF) cayla@run:~$
```

```
Collecting jupyter-client>=6.1.12 (from ipykernel)
  Using cached jupyter_client-7.1.1-py3-none-any.whl (130 kB)
  Using cached jupyter_client-7.1.0-py3-none-any.whl (129 kB)
  Using cached jupyter_client-7.0.6-py3-none-any.whl (125 kB)
  Using cached jupyter_client-7.0.5-py3-none-any.whl (124 kB)
  Using cached jupyter_client-7.0.4-py3-none-any.whl (124 kB)
  Using cached jupyter_client-7.0.3-py3-none-any.whl (122 kB)
  Using cached jupyter_client-7.0.2-py3-none-any.whl (122 kB)
  Using cached jupyter_client-7.0.1-py3-none-any.whl (122 kB)
  Using cached jupyter_client-7.0.0-py3-none-any.whl (122 kB)
  Using cached jupyter_client-6.1.12-py3-none-any.whl (112 kB)
ERROR: Cannot install ipykernel and jupyter-client because these package versions have conflicting dependencies.

The conflict is caused by:
    jupyter-core 5.3.0 depends on pywin32>=300; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 5.2.0 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 5.1.5 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 5.1.4 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 5.1.3 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 5.1.2 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 5.1.1 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 5.1.0 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 5.0.0 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 4.12.0 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 4.11.2 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 4.11.1 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 4.10.0 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 4.9.2 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 4.9.1 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 4.9.0 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 4.8.2 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 4.8.1 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 4.8.0 depends on pywin32>=1.0; sys_platform == "win32" and platform_python_implementation != "PyPy"
    jupyter-core 4.7.1 depends on pywin32>=1.0; sys_platform == "win32"
    jupyter-core 4.7.0 depends on pywin32>=1.0; sys_platform == "win32"
    jupyter-core 4.6.3 depends on pywin32>=1.0; sys_platform == "win32"
    jupyter-core 4.6.2 depends on pywin32>=1.0; sys_platform == "win32"
    jupyter-core 4.6.1 depends on pywin32>=1.0; sys_platform == "win32"
    jupyter-core 4.6.0 depends on pywin32>=1.0; sys_platform == "win32"
```

Why is dependency management so hard?

```
ERROR: ResolutionImpossible: for help visit https://pip.pypa.io/en/latest/topics/dependency-resolution/#dealing-with-dependency-conflicts
--force, or --legacy-peer-deps
rect (and potentially broken) dependency resolution.
```

Internal vs. External dependencies

Internal

- Maintainers' goals are (hopefully) aligned
- Can audit for all uses of a library
- Can coordinate large-scale changes of all code using library (facilitated by monorepo)
- Can manage with source control tools, policies

External

- Cannot assume coordination between library and users
- Cannot enforce compatibility, maintenance policies
- Cannot control release schedule
- Danger of **diamond dependency** problem
- Domain of dependency management

Why depend on external code?

Pros

- **Increase productivity**
- Benefit from higher quality
  - External expertise
  - Incorporate experience from diverse users
- Outsource maintenance burden
  - Bug fixes, modernization, refactoring

Cons

- Dependence on code outside of your control
  - Do you have the resources to audit it?
- Potential for **dependency bloat**
- Potential for **incompatibilities**
- Supply chain vulnerabilities

Where to get dependencies from?

- Defer to users / distributors
  - E.g., List of **Debian packages** to install
  - Common for libraries, system software (C/C++); often used for "standard" dependencies
  - **Build system** should confirm that dependencies are satisfied
  - *May* assume elevated privileges, may mask portability
- "**Vendoring**"
  - Copy third-party source code (or artifacts) into your repository
  - Increases project size, hard to maintain

- Artifact repositories
  - Download binary artifacts and their transitive dependencies
  - E.g., **Maven Central**, **PyPi (Python)**, **Debian packages**
- Source code repositories
  - Download the source code and compile locally
  - E.g., Cargo.io, npm
- Private Cloud Registry

Dependency Management Practices

- **Version pinning**: select the exact dependency version
  - But only applies to direct dependencies
  - Problems?
- Signature and hash verification
- **Lockfiles**: pinning+sig/hash verification for full dependency tree
  - Compiles all dependencies and sub-dependencies (entire dependency tree)
  - Better reproducibility and consistency
- **Dependency confusion attack**: publishing projects with the same name as an internal project to open-source
- **Vulnerability scanning**: scan lock files to check the artifact versions

https://cloud.google.com/blog/topics/developers-practitioners/best-practices-dependency-management

Supply Chain Attack (Example)

- https://pytorch.org/blog/compromised-nightly-dependency
- PyTorch-nightly Linux packages installed via pip during that time installed a dependency, **torchtriton**, which was compromised on the Python Package Index (PyPI) code repository and ran a malicious binary. This is what is known as a **supply chain attack** and directly affects dependencies for packages that are hosted on public package indices.
- A malicious dependency package (**torchtriton**) that was uploaded to the Python Package Index (PyPI) code repository with the same package name as the one shipped on the PyTorch nightly package index.
- This malicious package was being installed instead of the version from the official repository.
- This malicious package contains code that uploads sensitive data from the machine.

Compromised PyTorch-nightly dependency chain between December 25th and December 30th, 2022.

If you installed PyTorch-nightly on Linux via pip between December 25, 2022 and December 30, 2022, please uninstall it and torchtriton immediately, and use the latest nightly binaries (newer than Dec 30th 2022).

```
$ pip3 uninstall -y torch torchvision torchaudio torchtriton
$ pip3 cache purge
```

PyTorch-nightly Linux packages installed via pip during that time installed a dependency, torchtriton, which was compromised on the Python Package Index (PyPI) code repository and ran a malicious binary. This is what is known as a supply chain attack and directly affects dependencies for packages that are hosted on public package indices.

NOTE: Users of the PyTorch **stable** packages **are not** affected by this issue.**

Supply Chain Attack (Litellm)

- **LiteLLM**: widely used python library for LLM API calls
- On March 24, 2026, production systems running LiteLLM **started dying**, and engineers saw runaway processes, **CPU pegged at 100%,** containers killed by out-of-memory (OOM) errors!
- **Versions 1.82.7 and 1.82.8 contained malicious code that stole cloud credentials, SSH keys, and Kubernetes secrets!**
- Downloaded file “litellm_init.pth” (34kb) contained:

```
import os, subprocess, sys; subprocess.Popen([sys.executable, "-c",
"import base64; exec(base64.b64decode('...'))"])
```

- This parsed and uploaded your secrets next time you ran Python

https://medium.com/@tahirbalarabe2/the-litellm-pypi-supply-chain-attack-what-you-need-to-know-6ab536d4aeb3

https://www.trendmicro.com/en_us/research/26/c/inside-litellm-supply-chain-compromise.html

Types of dependencies

- **Direct**: Declared in the project dependencies
- **Transitive**: Dependencies of dependencies
- Node.js/NPM provides lock files to list all dependencies
  - Generates a package.lock.json file with versions of all dependencies

- See https://deps.dev/npm/glob/11.0.1/dependencies

Challenges in dependency management

- Management of networks of libraries, packages, and dependencies that we don’t control
- Concerns:
  - How to update between versions of external dependencies?
  - How do we describe versions?
  - What changes are allowed/expected?
  - How to decide whether its wise to depend on other org’s code?
- Cascade of upgrades

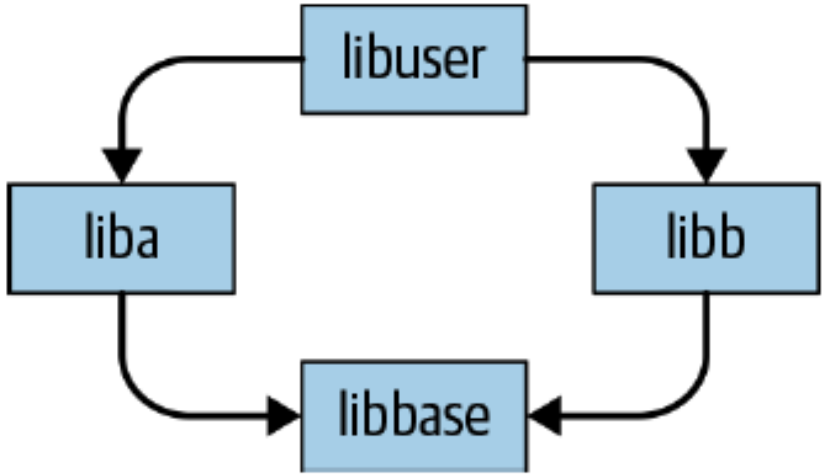
Repository mirrors

- Depending on public repositories is risky
  - What if their servers are not available?
  - What if packages are removed?
  - Do you trust that an artifact will never change?
  - Does your employer's firewall block binaries? Do they need to scan for viruses?

- Can point build tools to an **internal repository mirror**, rather than the public internet
  - Tradeoff between maintenance and control

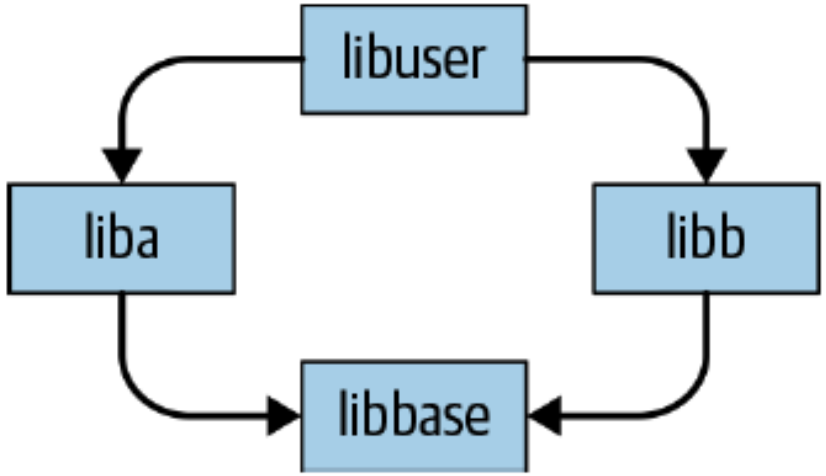
Diamond dependency problem

- Consider an application that uses a computer vision library and a GUI toolkit
- Suppose the CV library depends on libpng-1.4, but the GUI toolkit is linked against libpng-1.2. These versions are incompatible
- What version of libpng can your application link against?



Diamond dependency problem

- Solutions:
  - Downgrade a library?
  - Upgrade both?
  - Manually patch?



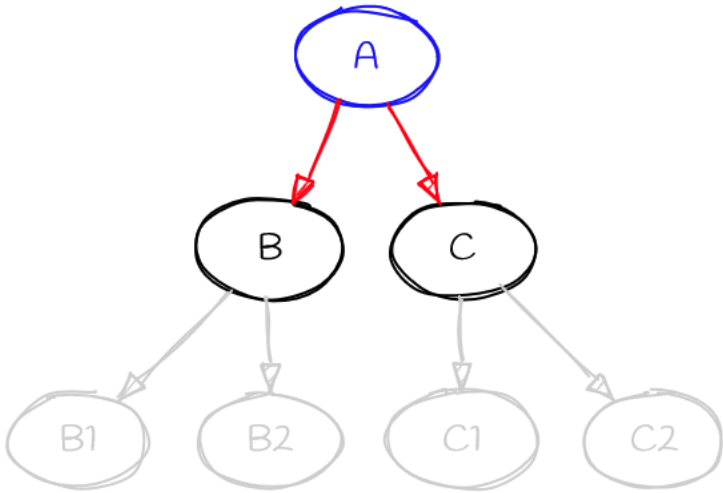
Dependency Resolution

- Given a set of dependencies, find a set of package to install
  - **NP-Hard problem**!
  - The more dependencies you have, the harder and longer it gets to resolve dependencies

- How pip works:
  - **Finder**: Get package data from an “index” (e.g., all pandas versions available)
  - **Resolver**: Considers all “candidates” from finder, and outputs the final list of resolved dependencies

Dependency Resolution

- Resolver algorithm steps (See ResolveLib):
  - **Identify**: identify candidates and requirements
  - **Get Preference**: Selects which requirement to look at "next"
    - Includes a bunch of heuristics
  - **Find matches**: Given a set of constraints, find what candidates exist that satisfy them (uses finder). May use a SAT Solver.
  - **is_satisfied_by**: checks if a candidate satisfies a requirement
  - **Get_dependencies**: get dependency metadata for a candidate.



https://pip.pypa.io/en/stable/topics/more-dependency-resolution

https://pypi.org/project/resolvelib

PollEv.com/cs5150sp26

- You are using pip to install a Python project with following dependencies.
- Package
  - A requires: B >= 1.0, < 2.0, C >= 2.0, < 3.0
  - B requires: D >= 1.0, < 2.0
  - C requires: D >= 2.0, < 3.0
- Can pip resolve this dependency?

PollEv.com/cs5150sp26

- You are using pip to install a Python project with following dependencies.
- Package
  - A requires: B >= 1.0, < 2.0, C >= 2.0, < 3.0
  - B requires: D >= 1.0, < 2.0
  - C requires: D >= 2.0, < 3.0
- Can pip resolve this dependency?

- No. B and C depend on conflicting/non-overlapping versions of D (**diamond dependency issue**)
- **Solution**: update the constraints

PollEv.com/cs5150sp26

- You are using pip to install a Python project with following dependencies.
- Package
  - A requires: B >= 1.0
  - B requires: C >= 1.0
  - C requires: A >= 1.0
- Can pip resolve this dependency?

PollEv.com/cs5150sp26

- You are using pip to install a Python project with following dependencies.
- Package
  - A requires: B >= 1.0
  - B requires: C >= 1.0
  - C requires: A >= 1.0
- Can pip resolve this dependency?

- Maybe yes. But there is a **circular dependency** problem
- **Solution**: Break the cycle, invert dependencies

Dependency management

- What versions of dependencies should you import?
- When should you upgrade dependency versions?

- SwE@Google book outlines four options:
  - Never upgrade
  - Semantic versioning
  - Bundled distributions
  - "Live at HEAD"

Never upgrade (Static Dependency Model)

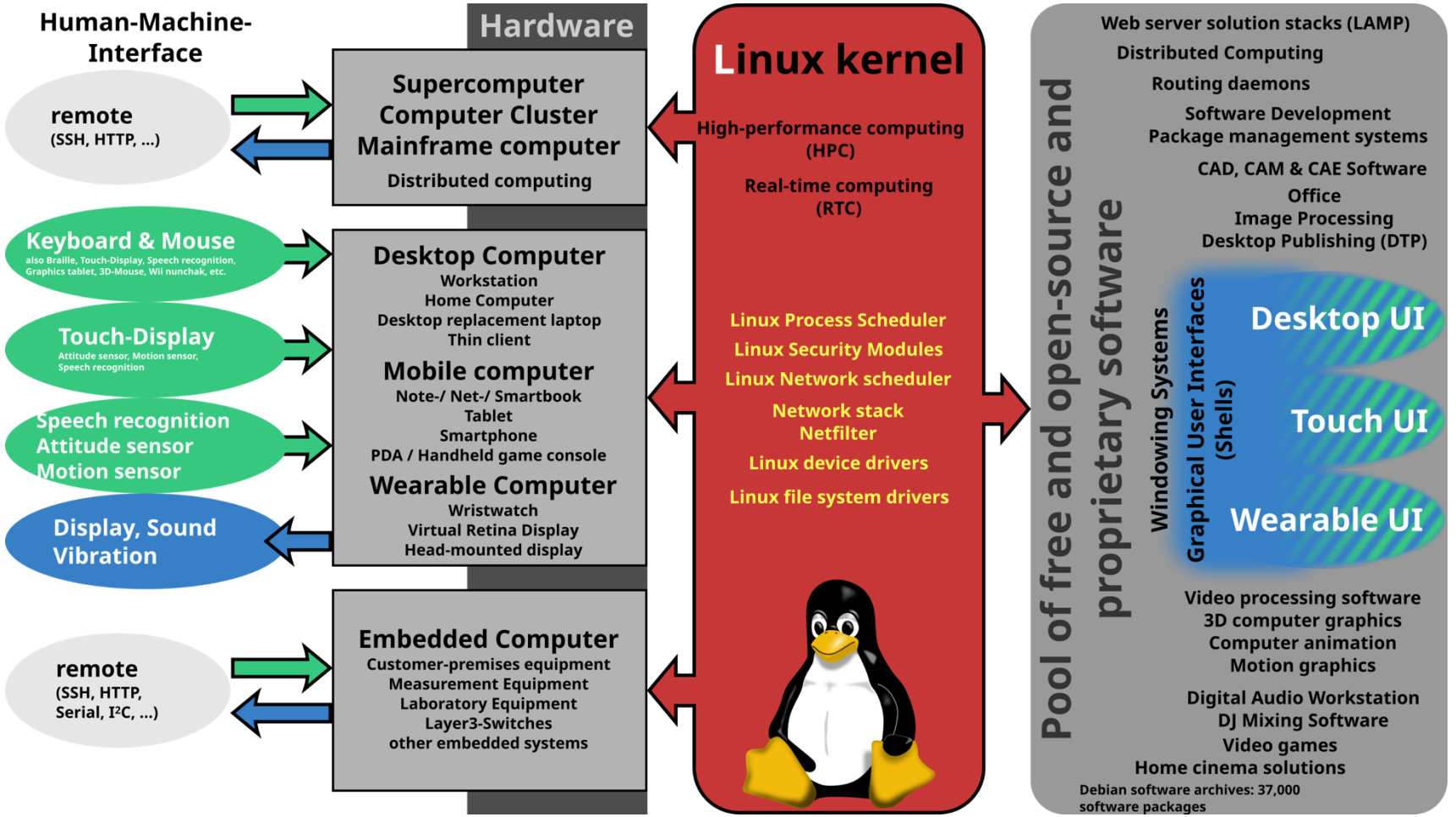
- Predictable
  - Avoids failures due to changes outside of your control
- Natural when starting out, or for short-lived projects
  - Compatible with "vendoring"
- What happens when a dependency has a security vulnerability?
- What happens when a new dependency depends on newer versions of old dependencies?

Bundled distributions

- Defer dependency management to the distribution maintainer
  - Responsible for maintaining compatibility while incorporating security updates
- Depend on the bundle and whatever dependency versions it provides
  - Common for commercial applications
- **Example**:
  - Linux distributions: Debian (non-commercial), Fedora Linux (Red Hat), OpenSUSE
  - Android, ChromeOS: Built on Linux Kernel
  - More niche: Raspberry PI OS
- **Distributors**: responsible for finding, patching, and testing a mutually compatible set of versions to include.
- Limitations:
  - Limits (verified) portability
  - Can't leverage latest features

Linux Distributions

- Typically includes
  - Linux Kernel
  - Package manager
  - Init system
  - GNU tools
  - Networking
  - GUI
  - …



Semantic versioning (SemVer)

- Dependency version numbers obey MAJOR.MINOR.PATCH format
  - Changes to PATCH should be fully compatible (bug fixes, security fixes)
  - Changes to MINOR may add functionality in a backwards-compatible manner
  - Changes to MAJOR indicate API changes (potentially breaking)
- Assumed by many build tools
  - Depend on a specific MAJOR version and a minimum MINOR version
- Challenges
  - Not all dependencies follow this scheme
  - Human maintainers make mistakes
  - **Hyrum's Law**: one person's "bug" is another's "feature"
  - Can be over-constraining (no solution to SAT problem)
    - Heuristics for relaxing some requirements

Hyrum's Law

"With a sufficient number of users, every observable behavior of your system will be depended upon by someone"

- SemVer's patch versions may not be "safe" (beyond input-output spec of an API)
  - Adding a delay in time-sensitive API
  - Logging format changes
  - Change order of results in a stream
  - Changing the order of importing dependencies…
- Many of these patches can be "breaking"

SemVer works when…

- Your dependency providers are accurate and responsible (to avoid human error in SemVer bumps)
- Your dependencies are fine-grained (to avoid falsely over-constraining when unused/unrelated APIs in your dependencies are updated, and the associated risk of unsatisfiable SemVer requirements)
- All usage of all APIs is within the expected usage (to avoid being broken in surprising fashion by an assumed-compatible change, either directly or in code you depend upon transitively)

Hard to satisfy these when operating at large (Google) scale…

Which version to choose?

- For **deterministic builds**, choice shouldn't depend on when dependency resolution is performed
  - **Lock files**: capture results of dependency resolution
  - **Newer dependencies** will only be considered if locked versions do not satisfy constraints
  - Commit lock file to repository
    - It will be changed (and should be recommitted) when dependency resolution is run
- **Go** recommends choosing *minimum* (MINOR) version required by dependency network
- If MINOR versions are maintained as release branches, hopefully security fixes will be backported to them as PATCH releases

Minimum Version Selection (MVS)

- SemVer: Chooses the newest possible versions of dependencies that satisfy requirements
- **MVS**: Select the lowest satisfiable version
- **Intuition**: Produce **high-fidelity builds** in which dependencies are as close as possible to what the developer used
- Proposed by Russ Cox for Go: https://research.swtch.com/vgo-mvs

PollEv.com/cs5150sp26: Which version would you upgrade to?

You are maintaining a Java web service that uses the library **json-utils** for parsing and generating JSON. Your current version is: 2.4.1

Upgrade reason: Your team needs better performance when serializing large JSON payloads, especially due to recent load testing that revealed bottlenecks. You heard that newer versions have optimized this.

Version	Change Type	Release Notes Summary
2.4.2	Patch	Fixed a memory leak in edge-case deserialization. No API changes.
2.5.0	Minor	Improved JSON serialization performance by 30%. Backward-compatible.
3.0.0	Major	Rewritten core APIs; significantly faster, but deprecated several classes and changed behavior for null values.
3.1.0	Minor	Adds new streaming APIs and improves documentation. Still same breaking changes as 3.0.0.

"Live at HEAD"

- Dependency management analogue of *trunk-based development*
- **Principles**:
  - Always depend on current stable version of everything
  - Never change anything in a way that is difficult for dependents to adapt
- Dependency maintainer responsible for not breaking all users
  - Effectively requires continuous integration for all software in the world (except closed-source dependents)
  - If compatibility cannot be maintained, maintainer will provide an upgrade tool
- **API providers**: Ensure smooth migration; **API consumers**: Provide tests

"Live at HEAD"

- Some of this infrastructure already exists
  - "Rolling" Linux distributions (e.g., Gentoo) integrate tens of thousands of packages continuously
  - Programming languages (e.g. Scala, Rust) proactively test all changes against major libraries/applications
- **Version selection**: What is the latest stable version of everything?

Dependency vulnerabilities

- NPM has a history of dependency-related disasters
  - **left-pad** unpublished
  - Bitcoin theft transitive dependency in **event-stream**
  - Ukraine war **"protestware"** in **node-ipc**
- Why was impact so large?
  - Tools depended on external repository services rather than internal mirror
  - Projects depended on **floating** instead of fixed versions (e.g., >1.5, 5.*)
  - Projects were built "too continuously"
  - Fine-grained dependencies depended upon by many other libraries (cascading)

https://en.wikipedia.org/wiki/Npm_left-pad_incident

https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/hacker-infects-node-js-package-to-steal-from-bitcoin-wallets

https://orca.security/resources/blog/cve-2022-23812-protestware-malicious-code-node-ipc-npm-package

Vulnerabilities

- CVE: Common Vulnerabilities and Exposures
  - Common identifier for specific vulnerabilities (not vulnerable systems)
  - CWE: Common Weakness Enumeration (type of vulnerability)
  - May be crosslinked with other databases (e.g., severity, product, weakness category)
    - NIST's National Vulnerability Database (NVD) includes common links and history
    - Common Vulnerability Scoring System (CVSS) standardizes measures of severity
- Example CVE: https://nvd.nist.gov/vuln/detail/CVE-2025-32955
- Others: https://mvnrepository.com/artifact/org.opencms/opencms-core/16.0

Reading

- *Software Engineering at Google*, Chapter 21: Dependency Management