Build process & build management

Jana Kostičová

Build process

Building SW = processing source code and its dependencies into a form that can be executed or used by a computer system.

- Writing / generating sources
- Static checking
- ….

} **Broader build process**

- Preprocessing (C / C++, some Fortran dialects, ..)
- Compilation (compiled languages)
- Linking (C / C++, assembly language, ..)
- Dependency resolution
- Packaging / bundling

} **Core build process**

- Automated tests
- Deployment
- ….

} **Broader build process**

It depends on various factors what activities are involved (programming language, size and type of application, target environment, ...)

Build tools

- Manual building - may suffice for extremely small projects with minimal dependencies
- Build tools - highly recommended if we have anything more complex

GNU make, Maven, Gradle, Vite, …

Key functionalities

- Build automation
  - Scripting tasks for efficient and repeatable builds
- Build configurations
  - Typically development, testing, production
- Bundling / creating artifacts
- Running automated tests
- Facilitating Continuous Integration (CI)

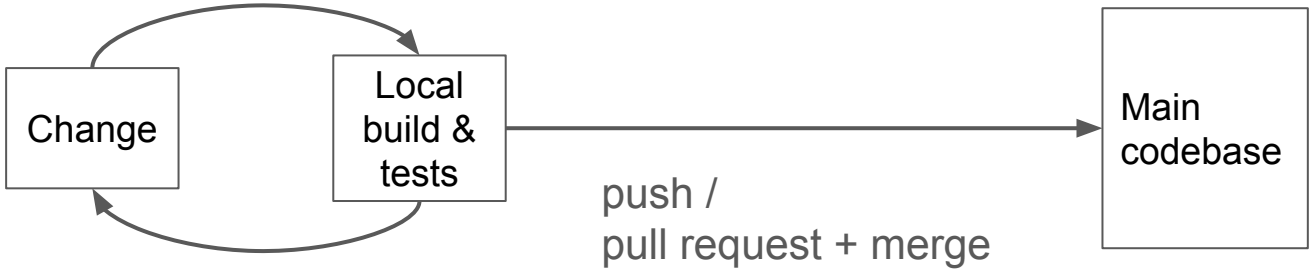
Dependency management

- Automatically managing external libraries needed by the software
- Not a core responsibility of build tools, but might be integrated into them (Maven, Gradle)

→ Package managers (specialized tools)

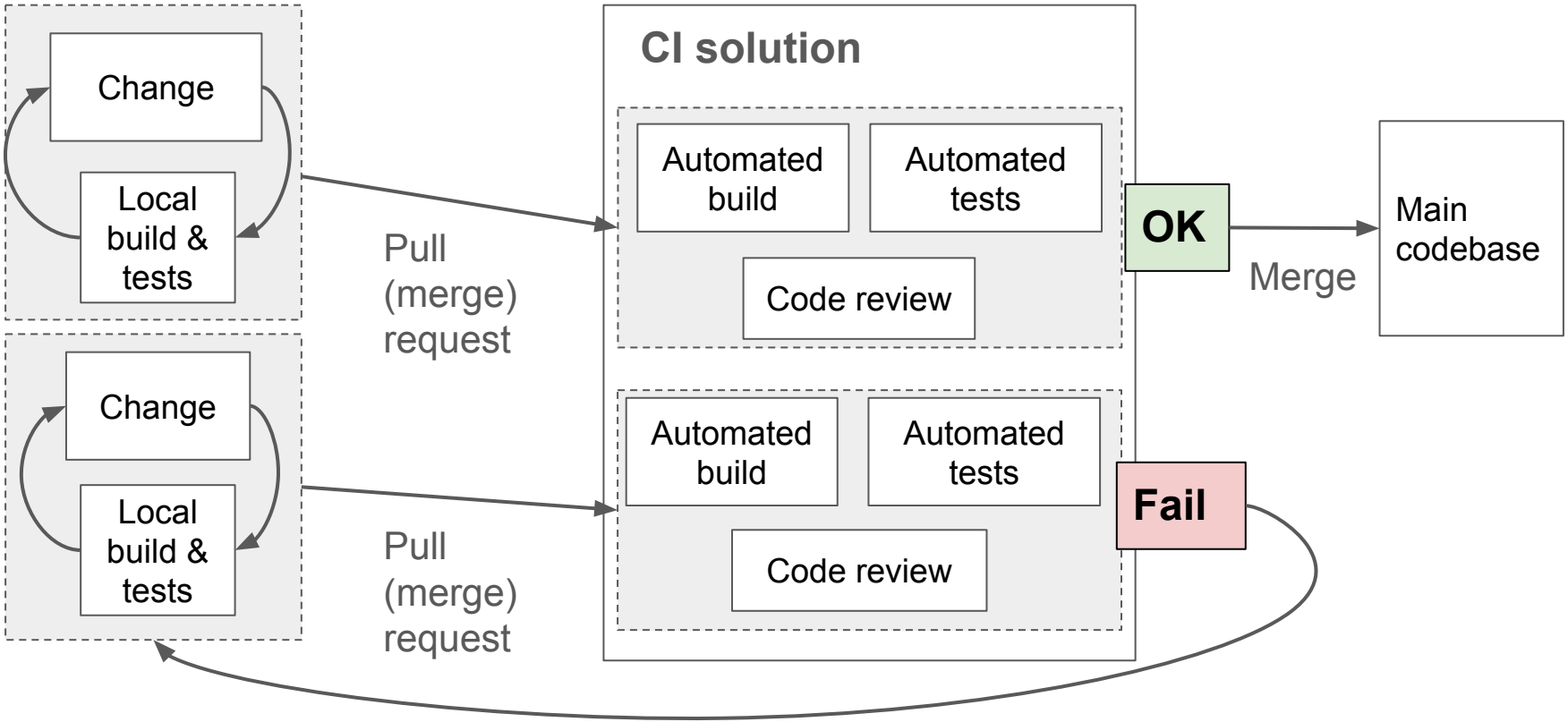
Npm, yarn, pip, …

Simple projects



- Build tools might not be needed
- Code quality is only checked locally

Continuous Integration (CI) workflow



Local builds vs CI builds

- Local builds
  - Fast feedback loop for developers
  - Catch errors early (before committing)
- CI builds
  - A “safety net” catching errors (local / integration) before merging into the main codebase
  - Feedback to the developer (success / failure)
  - Requires **automatization of build process**

- Both of them should use the same build tool and similar configuration
  - Some differences may appear (e.g., omitting some tests in local environment)

Local builds vs CI builds - AI assistance

Local builds - coding & debugging assistants (often integrated in IDE)

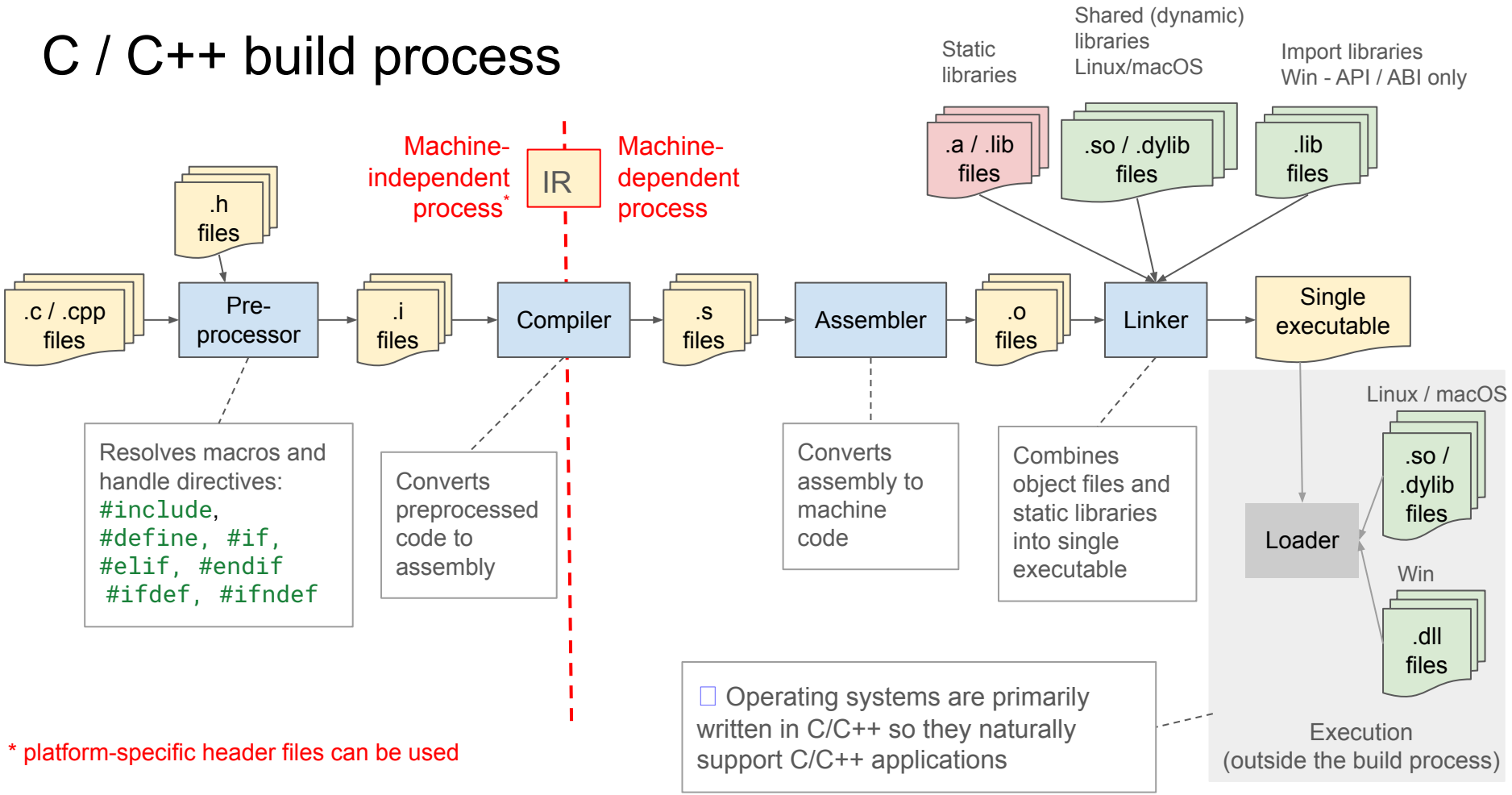
- Suggest code as you type, generate functions / tests, explain errors, generate Dockerfiles / build configs / scripts
- GitHub Copilot, JetBrains AI assistant, VS IntelliCode, …

CI builds

- AI-assisted code review (CodeRabbit, DeepCode)
  - Detect bugs or risky patterns
  - Manual intervention still needed (following business objectives, architecture decisions)
- AI analyzing test failures
  - Summarize logs / explain errors / suggest fix
- AI security scanning
  - Detect vulnerabilities / suggest safer code patterns

C / C++

C / C++ build process



* platform-specific header files can be used

Static vs shared libraries

	Static linking (static libraries)	**Dynamic linking (shared libraries)**
Library contents	Copied into the executable	Referenced
Executable size	Larger	**Smaller**
Runtime dependency	**None**	Yes
Library updates	Requires rebuilding the application	No rebuilding (supposing API / ABI does not change)
Portability	**Portable**	Fragile
Startup performance	**Usually faster**	Usually slower due to runtime linking
Memory usage	Generally larger	Generally **smaller**, especially if many application use the library

Header files

- “Interface” of a library or another source file
  - Declarations - function prototypes, external variables, structures, typedefs, classes (C++)…
  - Definitions (rarely)
  - Macros
- Header file need to be included for each
  - set of functions used from a static library
  - set of functions used from a shared library
  - set of functions used from another source file

→ need by compiler: type checking, separate compilation

System libraries

- Provide system functions such as memory management, process creation, file handling,..
- C standard library
  - Shared version (.dll / .so) always globally available
  - Static version is mostly available or can be installed
  - Linker mostly finds it automatically, no need to provide the path/name manually
  - Header files: Part of C/C++ specifications, OSs should follow it (OS-specific extensions exist)
  - Implementation: Different, OS-specific
- Other
  - Win - Windows API: kernel32.dll, user32.dll, and gdi32.dll, …
  - Linux - libpthread, libm, libX11 …
  - macOS - libobjc, libpthread.libm, …
  - Available often only as shared libraries

DLL hell

- Windows 95/98/NT
- Applications crashing because of incompatible DLL versions, missing / conflicting DLLs ,...

C / C++ 3rd party and user-defined libraries

- 3rd party
  - Examples: *OpenSSL, SQLite, libcurl (network transfer), ..*
  - Mostly provided in both static and shared version
  - Recommended to use package managers (Conan, vcpfg)
- When to use your own library
  - Common reasons (modularity, reusability, encapsulation, ...)
- When to use your own static library
  - You want to avoid surprises at runtime
  - You want single executable deployment
  - You don’t mind larger executable
- When to use your own shared library
  - More applications uses the same functionality and you want to take advantage of memory sharing
  - You want to update the library independently of the applications that use it
  - You want smaller executable size

Shared libraries - ABI must match at run time !

Where OSs look for shared libraries at runtime

Linux (ld.so, ld-linux.so)

1. rpath
2. LD_LIBRARY_PATH
3. /lib, /usr/lib, /lib64, usr/lib64
4. /etc/ld.so.conf

Windows (kernel32.dll / ntdll.dl)

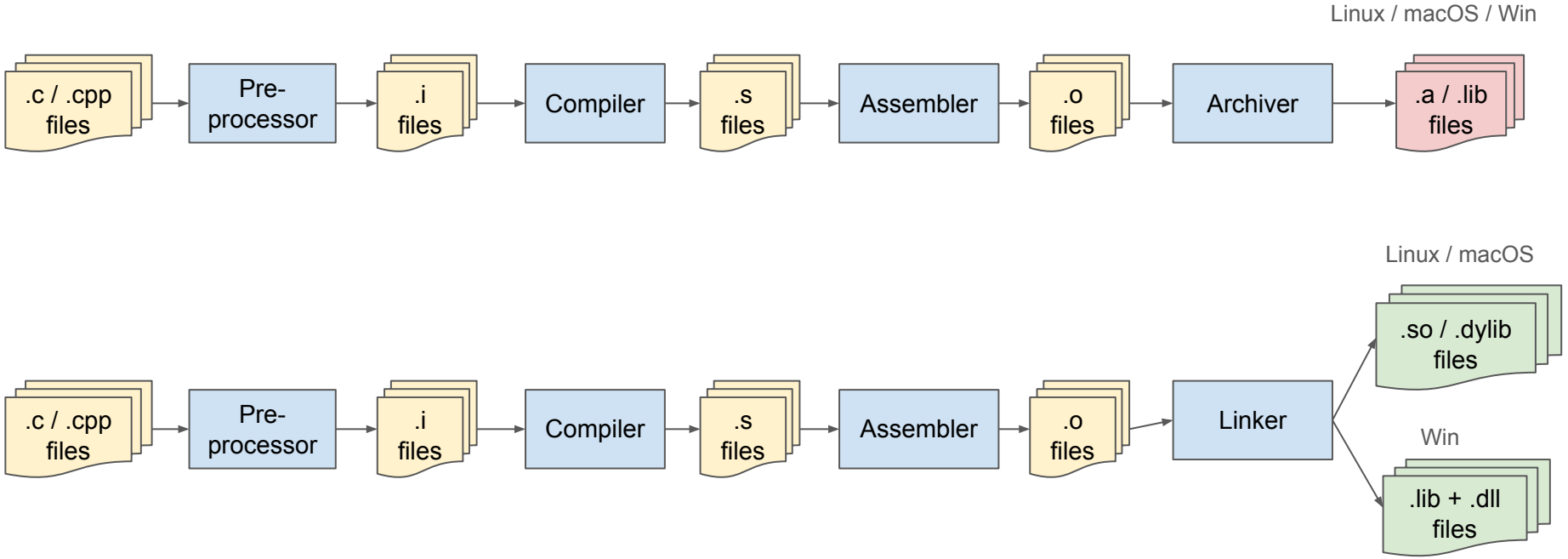
1. Directory of the executable
2. Current working directory
3. C:\Windows\System32, C:\Windows\SysWow64, C:\Windows
4. PATH
5. Registry

MacOS (dyld)

1. rpath
2. DYLD_LIBRARY_PATH
3. /usr/lib, /System/Library/Frameworks, Library/Frameworks
4. dyld cache

rpath is specified at link time

Building C/C++ libraries



GNU make

- Emerged in the late 1970s, still used today
- Popular for C/C++, can be used also for other languages
- Configuration file: Makefile

Main features

- Tracks dependencies between target files and their prerequisites
- Rebuilds only what's necessary based on changes (incremental build)
- Extensible through scripting and external tools

Limitations

- Platform-dependant (relies on shell scripting, specific tools, ..)
- Works well with small / medium sized projects, large projects can become hard to maintain
- Incremental build does not work well for file collections
- No support for managing external dependencies

} limits *make* usage for Java projects

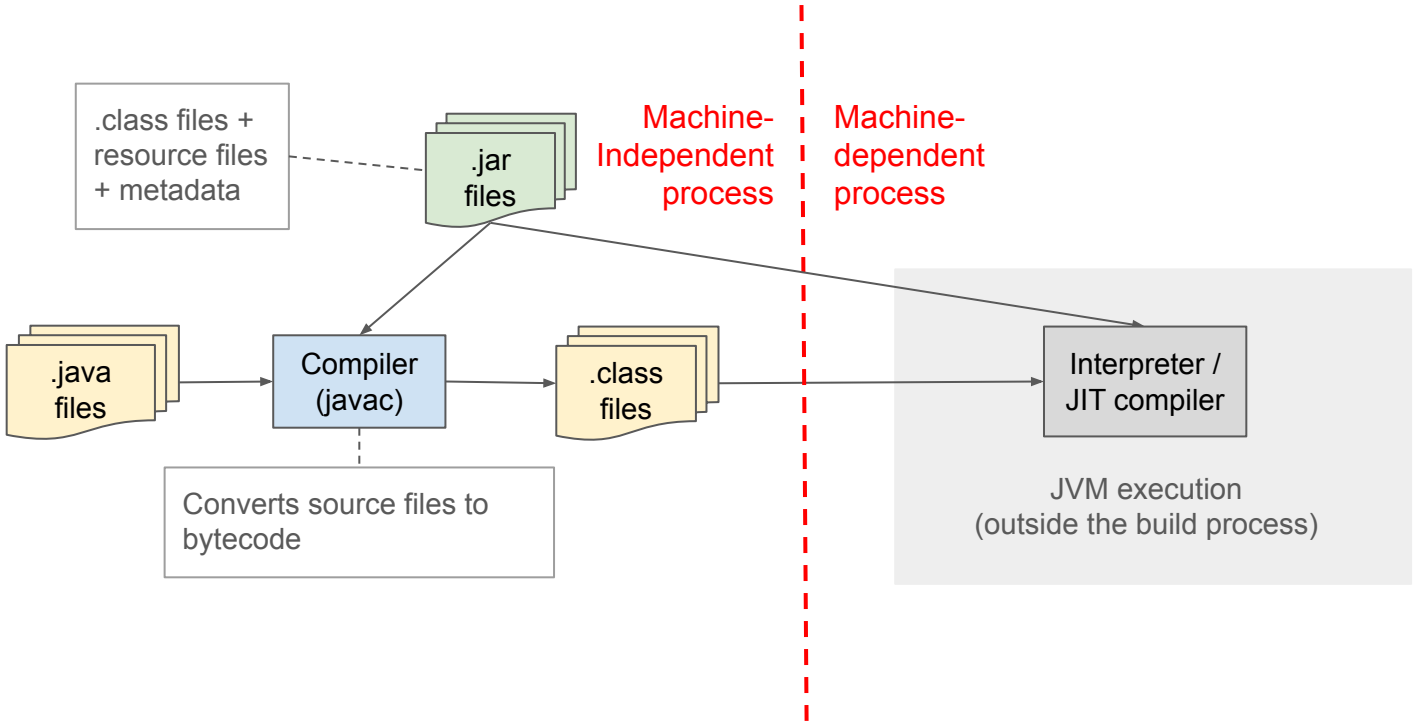
Manual: https://www.gnu.org/software/make/manual/

CMake

- First release in 2000
- Cross-platform
- Uses compiler-independent instructions for building applications
- Provide generators that translates these instructions to native build tools
  - Makefile, Visual Studio project files, …
  - Generated files are not intended to be edited manually

Java

Java - build process

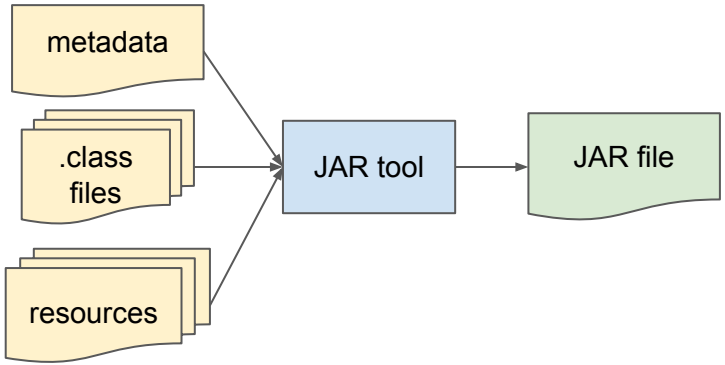


Java libraries

JAR files

= standard way to represent Java libraries

- They typically consist of classes, resource files and metadata (manifest file)
- JARs are runtime dependencies, dynamically loaded by JVM's class loader
- Each Java application typically loads classes from JARs into its own memory space (some mechanisms for class sharing exist - e.g., Class Data Sharing, Application Servers)



Java build tools

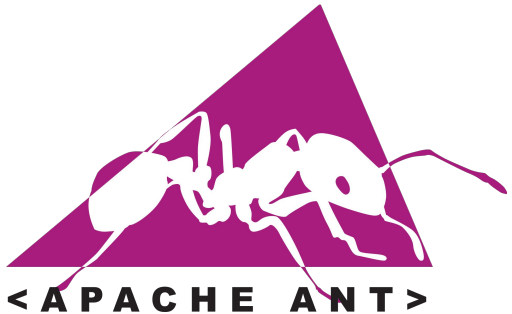
Make ❓ - issues arise:

- Java language-level dependencies during compilation
  - javac handles these dependencies within a given set of source files automatically
  - Makefile is primarily designed to have a single target file per rule - it doesn't work well at collection level (everything is recompiled even in case of a single change)
- Java project structure
  - Java projects typically organize source code by package structure that often result in deeply nested directories such as src/main/java/com/example/myapp/model/
  - Makefile can become cumbersome and difficult to maintain.
- External dependencies, manual JAR creation, classpath handling..

Ant, Maven, Gradle ✅

- Better choice

Apache Ant



- Released in 2000
- Configuration file: build.xml

Main features

- Uses XML
- Specifies build steps and tracks compilation dependencies similarly to “make” tool (imperative approach)
- **<javac>, <jar> and <java> tasks** simplify build and run
- Incremental build for collection of source files
- Limited management of external dependencies
- Cross-platform, extensible, adapts to existing project layout

Limitations

- Limited external dependency management - Ant has no means of downloading external dependencies from a central repository or resolve version conflicts
- No direct support for running automated tests

Apache Maven

- Released in 2004
- Motivation: to address specific needs of Java development
- Configuration file: pom.xml

Main features

- It uses XML, but less verbose than Ant
- Predefined build phases + plugins are bound to build phases (declarative approach)
- Robust dependency management
  - Lbraries are downloaded from repository, versions are tracked explicitly (default: Maven Central Repository - NOT maintained by Apache SW foundation)

Limitations

- Steeper learning curve
- It works best with standard Maven project structure
- Complex customization
  - A need to be familiar with Maven plugin development

Apache Maven - default build phases

- validate
- compile
- test
- package (e.g. to JAR / WAR / EAR file)
- integrate-test
- verify (additional checks on the packaged artifact)
- install (installing packaged artifact into local Maven repository)
- deploy (deploying packaged artifact into remote repository)

Default plugin bindings exist for phases

Gradle

- Released in 2009
- Configuration file: build.gradle

Main features

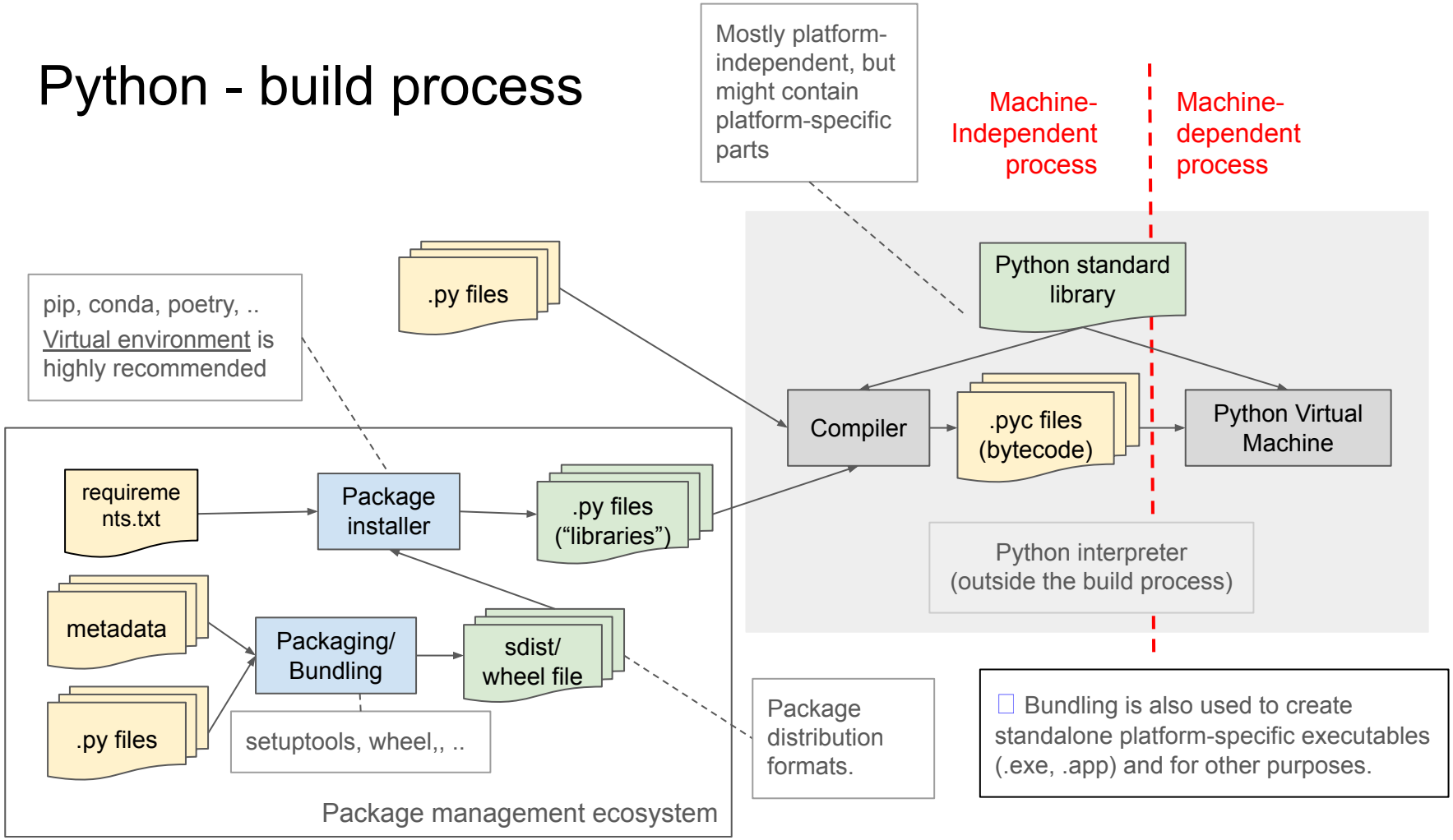
- Concise syntax
- Flexible approach to determine the execution order of the tasks (DAG)
- Robust dependency management

Limitations

- High flexibility may lead to complex configuration files with hard-to-understand semantics

Python

Python - build process



	Pre-process-ing	Compi-lation	Static linking	Build tools	(External) dependency management	Packaging / bundling
C / C++	Yes	Yes	Yes	make, NMake, CMake	System libraries are available Recommended for 3rd party libraries (vcpkg, Conan) System level package managers - apt, Homebrew,..	Static linking -> single executable External bundling (installers,OS-level packages,...)
Java	No	Yes (to bytecode)	No	Ant, Maven, Gradle	Recommended (Maven, Gradle)	Recommended (jar / war / ear)
C#	No	Yes (to CIL / bytecode)	No	MSBuild	Optional (pre-installed standard libraries vs. NuGet)	Recommended (MSBuild)
Python	No	No	No	setuptools Poetry, hatch, ..	Recommended (PyPI repository & pip & venv, conda, Poetry. hatch)	Recommended (setuptools, wheel, build)
JavaScript	No	Optional / transpiling to older JS	No	Vite, Webpack, esbuild, …	Recommended (npm, yarn)	Recommended (Webpack, Parcel, esbuild, Vite - includes esbuild)

Modern languages - examples

	Pre-processing	**Compi-lation**	**Static linking**	**Dynamic linking / loading**	**Build tools**	**(External) dependency management**	**Packaging / bundling**
Rust	No (only pre-compilation processing)	Yes (rustc compiler)	Yes	Often for system libraries	Cargo (the one and only…)	Cargo	Cargo (A statically linked single executable is created)
Go	No	Yes (Go compiler)	Yes - “norm”	Possible, but rare.	Go	Go modules	Go (A statically linked single executable is created)

- Simplified compiled languages (ahead-of-time)
- Strongly optimized for static linking
- Unified toolchain (build tool + dependency management)

References

- Wikipedia: Software build
- GitHub Actions: https://github.com/features/actions
- ISO/IEC 9899:2024 (en) — N3220 working draft (PDF), open-std.org. 2024-02-22 (almost identical to official C23 standard)
- GNU make: https://www.gnu.org/software/make/
- CMake: https://cmake.org/
- Apache Ant: https://ant.apache.org/
- Apache Maven: https://maven.apache.org/
- Gradle: https://gradle.org/