# L08 Software Management, Build Systems, and DevOps

Dr. Marco Elver
Systems Research Group
https://dse.in.tum.de/

# Today's learning goals

- **Part I:** Software configuration management
  - Source code management
  - Version control systems
  - Branch management
- **Part II:** Build systems
  - Task-based build systems
  - Artifact-based build systems
  - Distributed builds
  - Dependency management
  - Hermeticity
- **Part III:** Release management
  - Release planning
  - Software versioning
  - Software upgrades

# Today's learning goals

- **Part IV:** Configuration *
  - Continuous Integration
  - Continuous Delivery
  - Continuous Deployment
  - Continuous Fuzzing

# Outline

- **Part I: Software configuration management**
  - Source code management
  - Version control systems
  - Branch management
- **Part II:** Build systems
- **Part III:** Release management
- **Part IV:** Continuous *

# Why software configuration management?

*"Software engineering is programming integrated over time."*

📖 T. Winters et al., "Software Engineering at Google," 2020. URL: https://abseil.io/resources/swe-book

# Why software configuration management?

**Software and organizational evolution:**

- Multiple people work on evolving software systems.
- Software systems constantly changing and evolving:
  - New features, bug fixes, optimizations
- Supporting multiple versions of software:
  - Software under development
  - Software released to customers
  - Variants optimized for specialized use case or environment

# Software Configuration Management

*Policies, processes, and tools for managing software systems*

**Pillars of Configuration Management:**

1. **Version management:** Keeping track of multiple versions of system components.
2. **System building:** Collecting components to create an executable system.
3. **Change management:** Keeping track of requests for changes to delivered software from customers and developers.
4. **Release management:** Preparing software for external release and keeping track of the system versions that have been released.

📖 I. Sommerville, “Configuration management” (Chapter 25), In “Software Engineering,” 2015

# Version Management

- **Codeline (or Branch):** Set of versions of a *component* (and its dependendencies).
- **Baseline:** Collection of *component versions* that make up a system – i.e. "snapshot" of all components.
- **Mainline:** Sequence of baselines.

📖 I. Sommerville, "Version management" (Chapter 25.1), In "Software Engineering," 2015

# Version Management

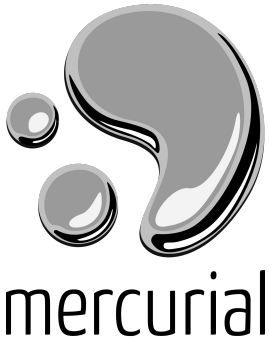
- **Codeline (or Branch):** Set of versions of a *component* (and its dependendencies).
- **Baseline:** Collection of *component versions* that make up a system – i.e. "snapshot" of all components.
- **Mainline:** Sequence of baselines.

**Example:**

| | |
|---|---|
| Codeline A | A1.0 → A1.1 → A1.2 → A1.3 |
| Codeline B | B1.0 → B1.1 → B1.2 → B1.3 |
| Dependencies | {Lib1, Lib2, Dat1, Dat2} |
| Baseline V1 | {A1.0, B1.2, Lib1, Lib2, Dat1} |
| Baseline V2 | {A1.3, B1.2, Lib1, Lib2, Dat2} |
| Mainline | V1 → V2 |

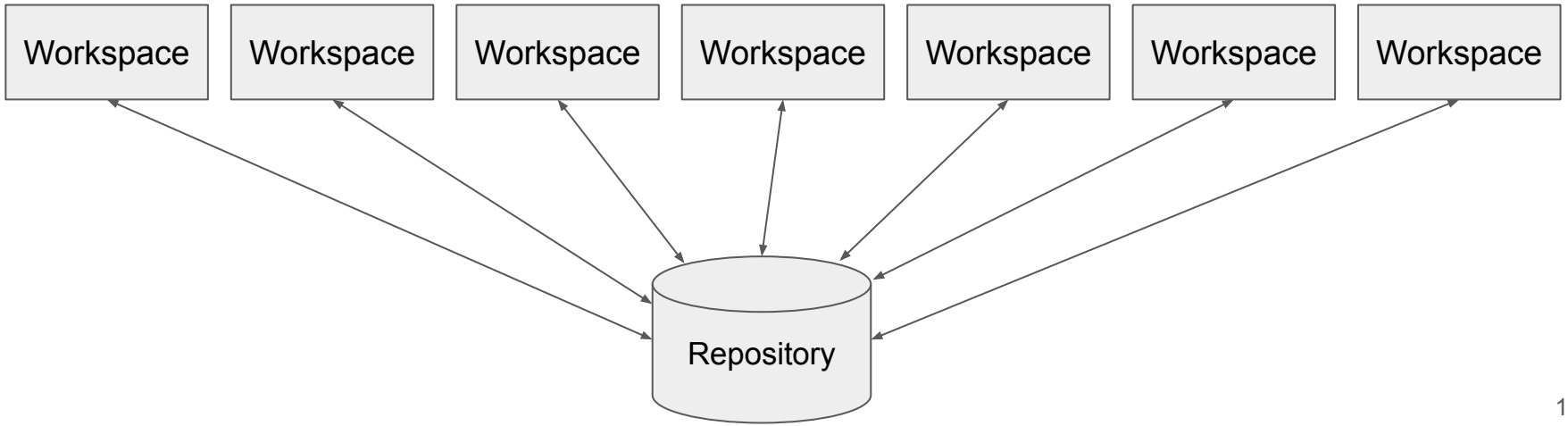
# Version Control Systems

- Version control system (VCS) keeps track of different versions of software.
- Ensures that *concurrent* changes made by different developers do not interfere.
- May track source code, binaries, assets, etc.
- All versions of components and metadata tracked in a **repository.**


# Centralized Version Control Systems

- Centralized **repository:** single repository stores all version of components being developed.
- Developer only has local **workspace(s),** which are snapshots of a given repository state allowing modification of components.

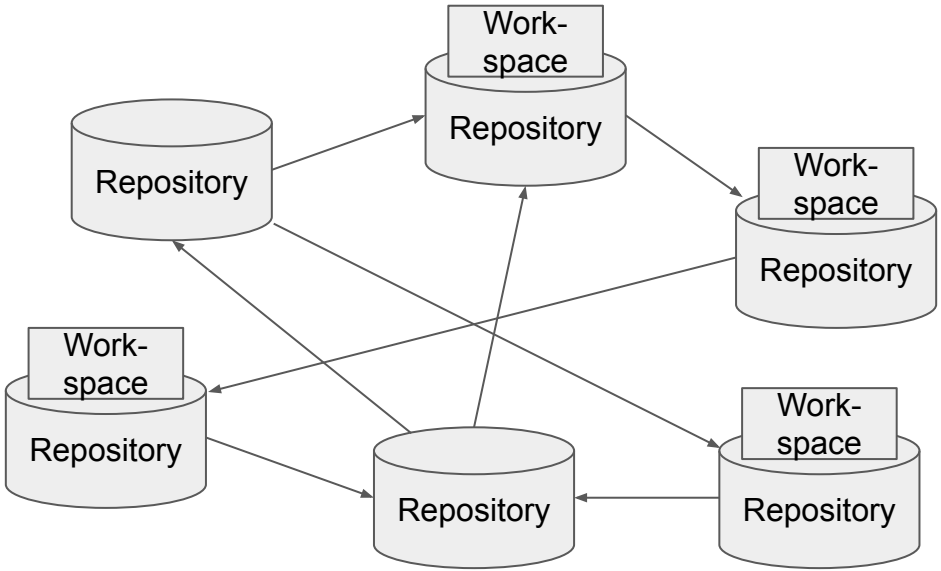


# Centralized Version Control Systems

- Centralized **repository:** single repository stores all version of components being developed.
- Developer only has local **workspace(s),** which are snapshots of a given repository state allowing modification of components.
- **Examples:**
  - Subversion: subversion.apache.org – open source, still widely used
  - Perforce: www.perforce.com – proprietary, mostly enterprise use
  - Concurrent Versions Systems (CVS): www.nongnu.org/cvs/ – open source, no longer recommended for new projects

# Distributed Version Control Systems

- Distributed **repository:** multiple replicas of repository exist concurrently, not necessarily synchronized (i.e. replicas can be in different states).
- Developer has a local copy of a **repository** snapshot, along with local **workspace(s)** to modify components.



# Distributed Version Control Systems

- Distributed **repository:** multiple replicas of repository exist concurrently, not necessarily synchronized (i.e. replicas can be in different states).
- Developer has a local copy of a **repository** snapshot, along with local **workspace(s)** to modify components.
- **Examples:**
  - Git: git-scm.com – open source, one of the most popular DVCS
  - Mercurial: www.mercurial-scm.org – open source
  - Darcs: darcs.net – open source
  - BitKeeper: www.bitkeeper.org – started proprietary, now open source, influenced creation of Git

# Version Control Systems – Git

- Initially developed for version management needs of the Linux kernel
- Distributed
  - Strong support for non-linear development (thousands of branches).
- Performant
  - Able to handle many gigabytes of data and metadata with ease.
- Simple yet powerful design
  - Easy to use, yet difficult to master.


📖 S. Chacon & B. Straub, “Pro Git,” Apress 2023. URL: git-scm.com/book

# Git: Configuring

```
$ git config --global user.name "First Last"
$ git config --global user.email first.last@tum.de
```

As a first step, you should tell Git who you are, so that it correctly attributes commit authorship.

The global configuration is located in “~/.gitconfig”.

# Git: Creating a repository from scratch

```
[1]$ git init my_project
Initialized empty Git repository in
…/my_project/.git/
[2]$ cd my_project
[3]$ git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add"
to track)
```

1. `git init <dir>` – Initialize an empty Git repository in directory “my_project” relative to current working directory
2. Change to directory “my_project”
3. Show status of current working tree

The “.git” directory contains all the relevant Git metadata that makes a directory a Git repository. Do not delete it!

Depending on which version and config of Git you are using, the default branch is subject to change. Usually it is “master” or “main”.

# Git: Cloning a remote repository

```
[1]$ git clone https://github.com/git/htmldocs.git
Cloning into 'htmldocs'...
remote: Enumerating objects: 45392, done.
remote: Counting objects: 100% (1887/1887), done.
remote: Compressing objects: 100% (917/917), done.
remote: Total 45392 (delta 1680), reused 1122 (delta
970), pack-reused 43505
Receiving objects: 100% (45392/45392), 42.20 MiB |
6.99 MiB/s, done.
Resolving deltas: 100% (41362/41362), done.
[ ]$ cd htmldocs
[ ]$ git status
On branch gh-pages
Your branch is up to date with 'origin/gh-pages'.

nothing to commit, working tree clean
[2]$ git pull
Updating ebf86a51..a7b2c108
Fast-forward …
```

1. `git clone <url>` – Clone repository at provided URL for use locally
2. Fetch from and merge latest changes from tracked remote repository

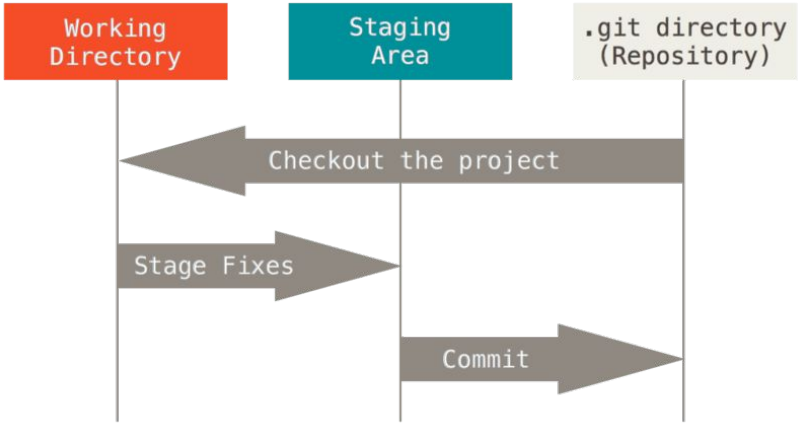
💡 Cloning large repositories can take a long time. If you are not interested in the full history of a repository, you can use "git clone --depth 1 <url>" to only download what is required for the latest version.

💡 "git pull" is a shortcut for a combination of "git fetch" and "git merge".

# Git: Adding files to staging area

```
[1]$ echo “hello git” > file1
[ ]$ git status
…
Untracked files:
  (use "git add <file>..." to include in what will be
committed)
    file1

nothing added to commit but untracked files present
(use "git add" to track)

[2]$ git add .
[ ]$ git status
…
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   file1
```

1. Create “file1” with string “hello git”
2. `git add <pathspec>...` – Add all files (new and changed) to staging area



# Git: Creating a commit

```
[1]$ git commit
[main 56e8b99] Initial commit
 Date: Thu Jun 15 16:55:36 2023 +0200
 1 file changed, 1 insertion(+)
 create mode 100644 file1
[ ]$ git status
On branch main
nothing to commit, working tree clean

[2]$ git log
commit 56e8b9977a605f1a02149f208f837669 (HEAD -> main)
Author: Marco Elver <marco.elver@tum.de>
Date:   Thu Jun 15 16:55:36 2023 +0200

     Initial commit

     Added file "file1".
```

1. `git commit` – Create a new commit (starts editor to compose commit message)
2. `git log` – Show current branch's history

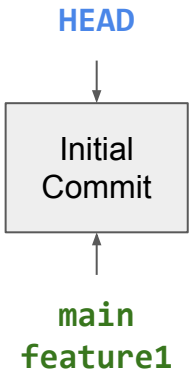
💡 If you do not want to enter an editor on commit, use "git commit -m <message>" to create a commit that contains the given short message. It is recommended to write descriptive commit messages, for which the editor is almost always the better choice.

💡 To commit without explicitly adding changed/new files to staging, you can include all changes in a commit with "git commit -a".

# Git: Creating a new branch

```
[1]$ git checkout -b feature1
Switched to a new branch 'feature1'
[2]$ echo "hello branch" > file2
[3]$ git status -s
?? file2
[4]$ git add file2
```

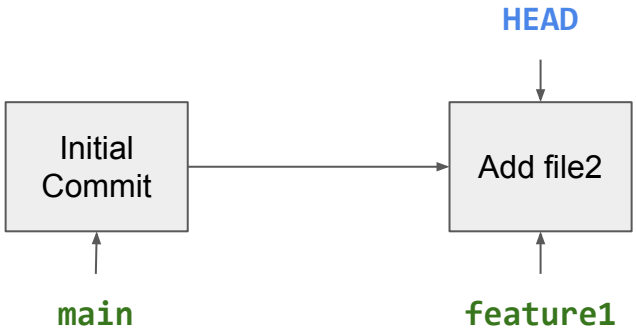
1. Create branch “feature1” and immediately start working on it
2. Create a new file “file2”
3. Show short status (“??” denotes unknown files)
4. Add new file to staging



# Git: Creating a new branch

```
[1]$ git checkout -b feature1
Switched to a new branch 'feature1'
[2]$ echo "hello branch" > file2
[3]$ git status -s
?? file2
[4]$ git add file2
[5]$ git commit -m "Add file2"
[feature1 a26e920] Add file2
 1 file changed, 1 insertion(+)
 create mode 100644 file2
```

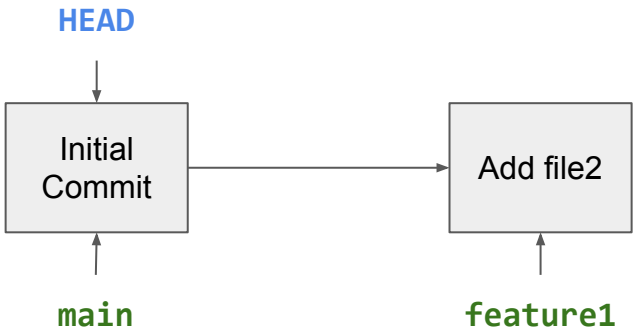
1. Create branch “feature1” and immediately start working on it
2. Create a new file “file2”
3. Show short status (“??” denotes unknown files)
4. Add new file to staging
5. Create commit that adds “file2”



# Git: Merging branches (fast forward)

```
[1]$ git checkout main
Switched to branch 'main'
```

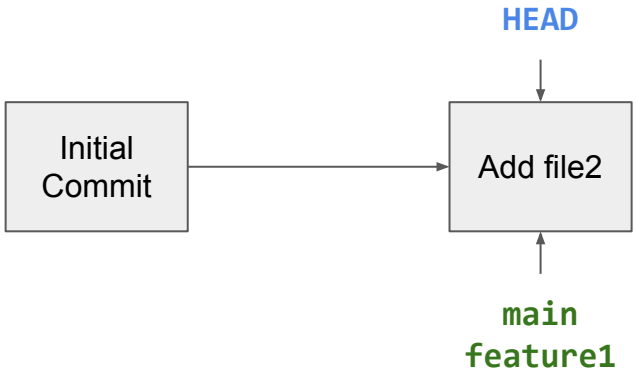
1. Switch back to branch “main”



# Git: Merging branches (fast forward)

```
[1]$ git checkout main
Switched to branch 'main'
[2]$ git merge feature1
Updating 94f251a..3776e54
Fast-forward
 file2 | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 file2
```

1. Switch back to branch “main”
2. Merge branch “feature1” into “main” branch by the “fast-forward” strategy which does not create a merge commit

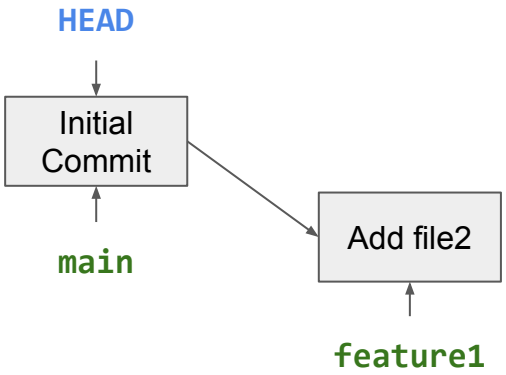



Fast-forwarding can only be done if the to-be-merged branch is a direct descendant of the current branch (at HEAD).

# Git: Merging branches (no conflict)

```
[1]$ echo “hello main” > file3
[2]$ git add file3
```

1. Create “file3”
2. Add “file3” to staging



# Git: Merging branches (no conflict)

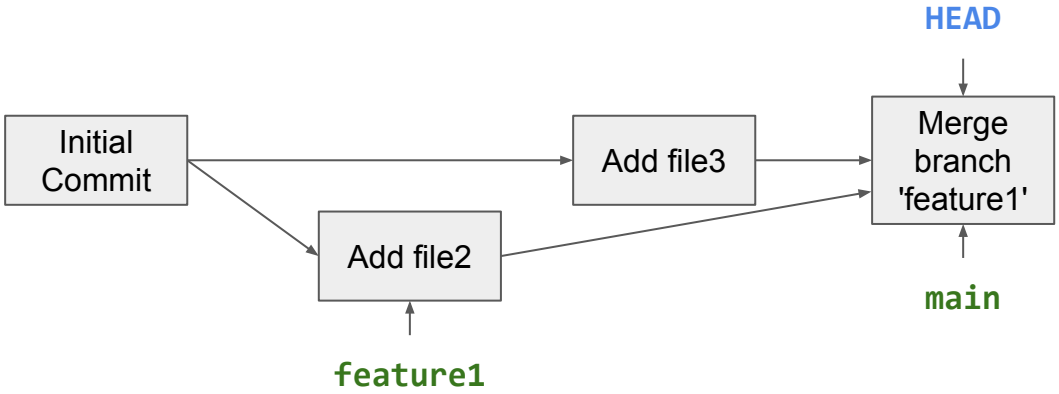
```
[1]$ echo “hello main” > file3
[2]$ git add file3
[3]$ git commit -m “Add file3”
[main d0e0489] Add file3
 1 file changed, 1 insertion(+)
 create mode 100644 file3
```

1. Create “file3”
2. Add “file3” to staging
3. Create new commit

# Git: Merging branches (no conflict)

```
[1]$ echo “hello main” > file3
[2]$ git add file3
[3]$ git commit -m “Add file3”
[main d0e0489] Add file3
 1 file changed, 1 insertion(+)
 create mode 100644 file3
[4]$ git merge feature1
Merge made by the 'ort' strategy.
 file2 | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 file2
```

1. Create “file3”
2. Add “file3” to staging
3. Create new commit
4. Merge branch “feature1” into “main” by creating a merge commit

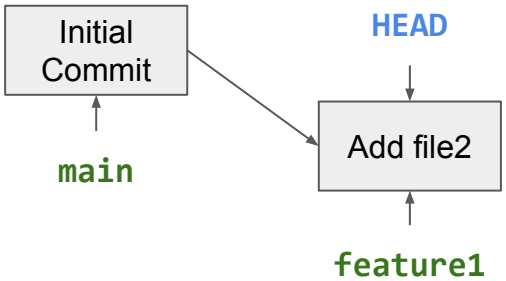



💡 The strategy “ort” is the default merge strategy when merging a divergent branch.

# Git: Merging branches (conflict)

```
[1]$ git checkout feature1
[2]$ echo “hello from feature1” >> file1
[3]$ git status -s
 M file1
```

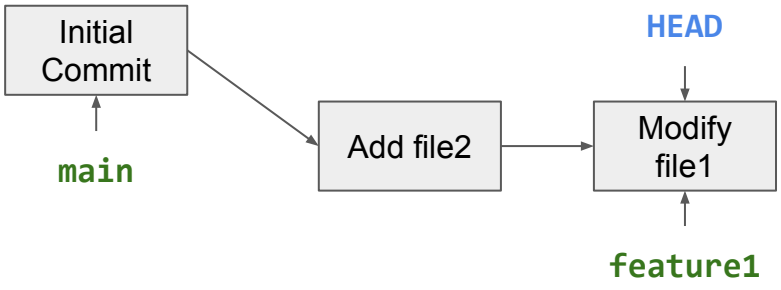
1. Switch to branch “feature1”
2. Modify “file1” on “feature1” branch
3. Show current status (“M” denotes modified files)



# Git: Merging branches (conflict)

```
[1]$ git checkout feature1
[2]$ echo “hello from feature1” >> file1
[3]$ git status -s
 M file1
[4]$ git commit -am "Modify file1"
[feature1 0d49def] Modify file1
 1 file changed, 1 insertion(+)
```

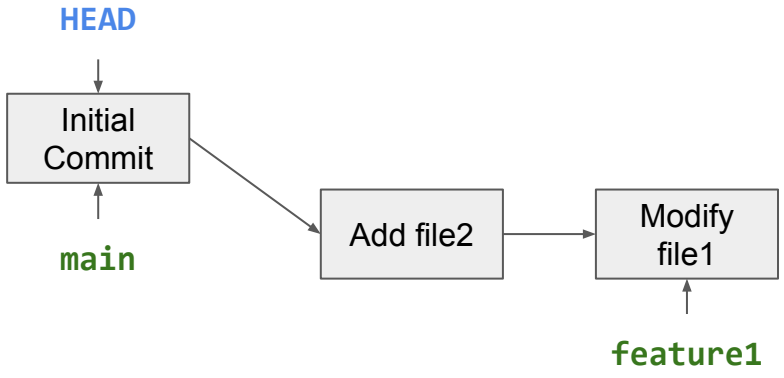
1. Switch to branch “feature1”
2. Modify “file1” on “feature1” branch
3. Show current status (“M” denotes modified files)
4. Create commit with modification on “feature1”



# Git: Merging branches (conflict)

```
[1]$ git checkout feature1
[2]$ echo “hello from feature1” >> file1
[3]$ git status -s
 M file1
[4]$ git commit -am "Modify file1"
[feature1 0d49def] Modify file1
 1 file changed, 1 insertion(+)
[5]$ git checkout main
```

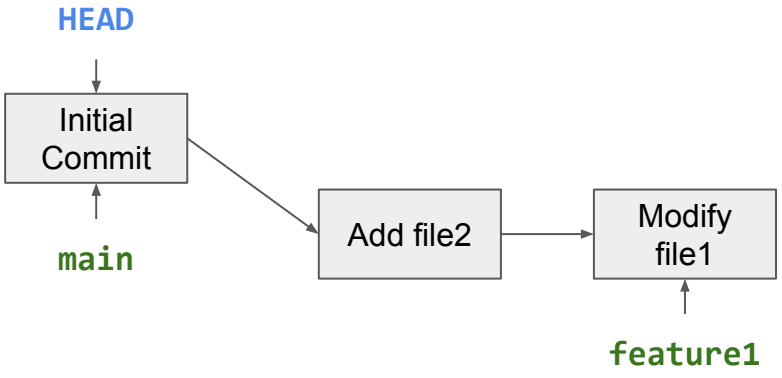
1. Switch to branch “feature1”
2. Modify “file1” on “feature1” branch
3. Show current status (“M” denotes modified files)
4. Create commit with modification on “feature1”
5. Switch back to the “main” branch



# Git: Merging branches (conflict)

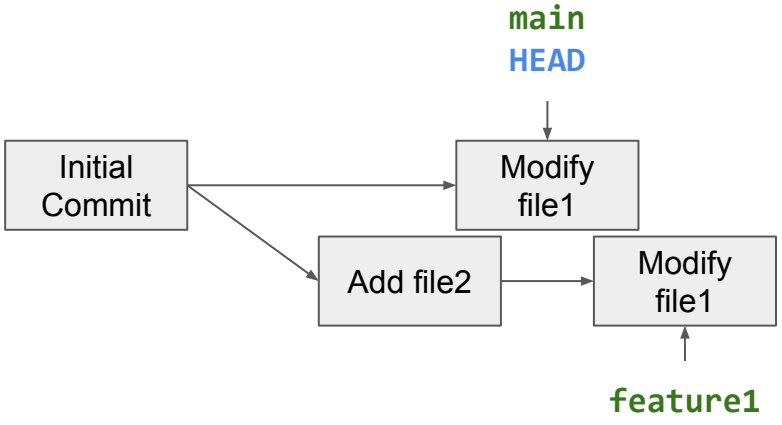
```
[1]$ git checkout feature1
[2]$ echo “hello from feature1” >> file1
[3]$ git status -s
 M file1
[4]$ git commit -am "Modify file1"
[feature1 0d49def] Modify file1
 1 file changed, 1 insertion(+)
[5]$ git checkout main
[6]$ echo “hello from main” >> file1
```

1. Switch to branch “feature1”
2. Modify “file1” on “feature1” branch
3. Show current status (“M” denotes modified files)
4. Create commit with modification on “feature1”
5. Switch back to the “main” branch
6. Modify “file1” on “main” branch



# Git: Merging branches (conflict)

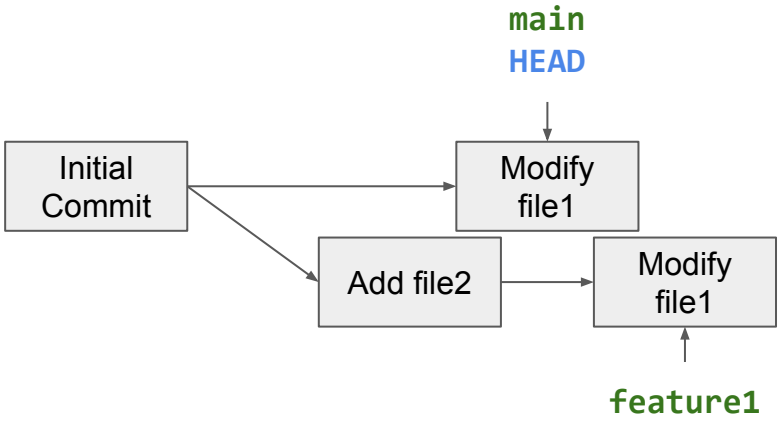
```
[2]$ echo “hello from feature1” >> file1
[3]$ git status -s
 M file1
[4]$ git commit -am "Modify file1"
[feature1 0d49def] Modify file1
 1 file changed, 1 insertion(+)
[5]$ git checkout main
[6]$ echo “hello from main” >> file1
[7]$ git commit -am “Modify file1”
[main c47cfd9] Modify file1
 1 file changed, 1 insertion(+)
```




1. Switch to branch “feature1”
2. Modify “file1” on “feature1” branch
3. Show current status (“M” denotes modified files)
4. Create commit with modification on “feature1”
5. Switch back to the “main” branch
6. Modify “file1” on “main” branch
7. Create commit with modification on “main”

# Git: Merging branches (conflict)

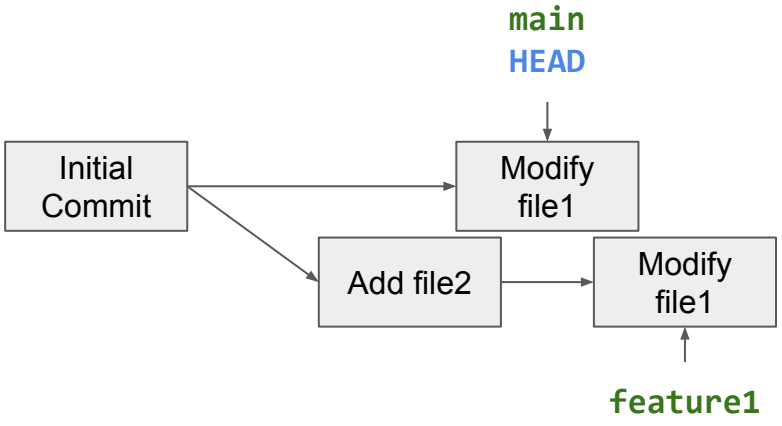
```
[8]$ git merge feature1
Auto-merging file1
CONFLICT (content): Merge conflict in file1
Automatic merge failed; fix conflicts and then commit
the result.
```




1. Switch to branch “feature1”
2. Modify “file1” on “feature1” branch
3. Show current status (“M” denotes modified files)
4. Create commit with modification on “feature1”
5. Switch back to the “main” branch
6. Modify “file1” on “main” branch
7. Create commit with modification on “main”
8. Merge branch “feature1” into “main”

# Git: Merging branches (conflict)

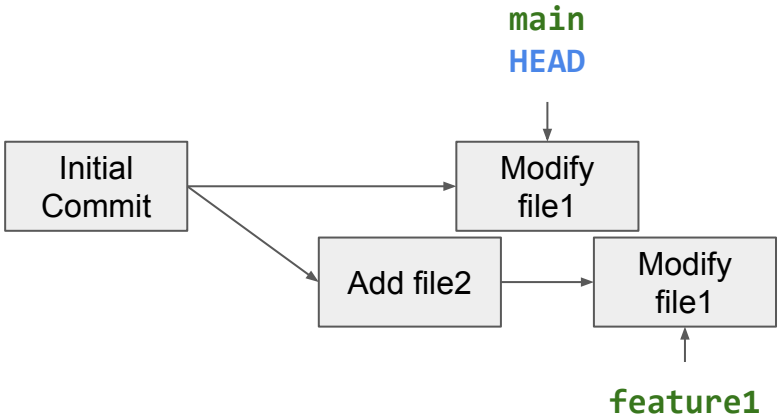
```
[8]$ git merge feature1
Auto-merging file1
CONFLICT (content): Merge conflict in file1
Automatic merge failed; fix conflicts and then commit
the result.
[ ]$ git status -s
UU file1
A  file2
[9]$ nano file1
```




1. Switch to branch “feature1”
2. Modify “file1” on “feature1” branch
3. Show current status (“M” denotes modified files)
4. Create commit with modification on “feature1”
5. Switch back to the “main” branch
6. Modify “file1” on “main” branch
7. Create commit with modification on “main”
8. Merge branch “feature1” into “main”
9. Resolve merge conflict manually

# Git: Merging branches (conflict)

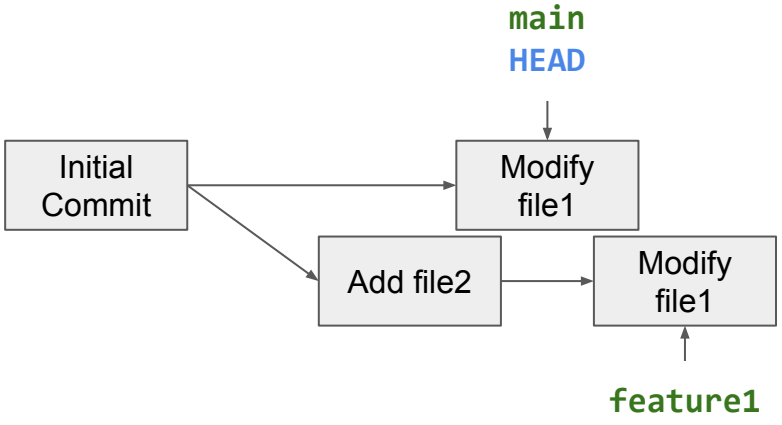
```
hello git
<<<<<<< HEAD
hello from main
=======
hello from feature1
>>>>>>> feature1
```




1. Switch to branch “feature1”
2. Modify “file1” on “feature1” branch
3. Show current status (“M” denotes modified files)
4. Create commit with modification on “feature1”
5. Switch back to the “main” branch
6. Modify “file1” on “main” branch
7. Create commit with modification on “main”
8. Merge branch “feature1” into “main”
9. Resolve merge conflict manually

# Git: Merging branches (conflict)

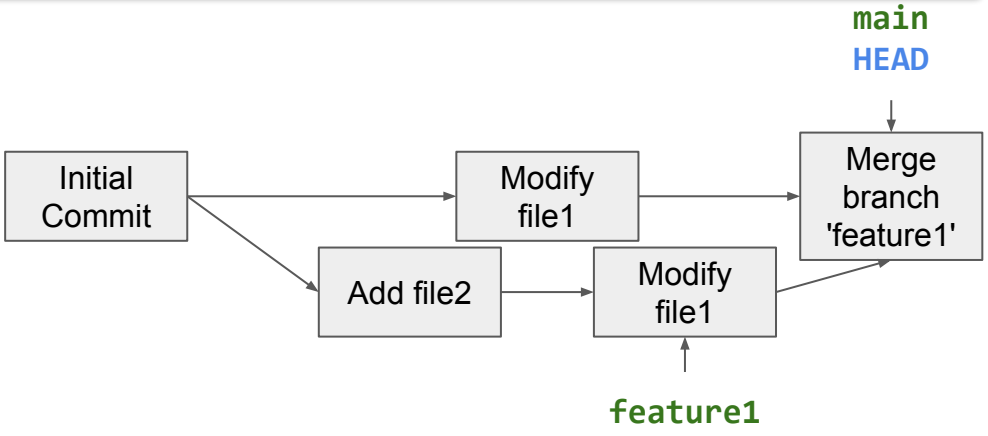
```
hello git
hello from main
hello from feature1
```




1. Switch to branch “feature1”
2. Modify “file1” on “feature1” branch
3. Show current status (“M” denotes modified files)
4. Create commit with modification on “feature1”
5. Switch back to the “main” branch
6. Modify “file1” on “main” branch
7. Create commit with modification on “main”
8. Merge branch “feature1” into “main”
9. Resolve merge conflict manually

# Git: Merging branches (conflict)

```
[10]$ git add .
[11]$ git commit
[main 57b2945] Merge branch 'feature1'
```




1. Switch to branch “feature1”
2. Modify “file1” on “feature1” branch
3. Show current status (“M” denotes modified files)
4. Create commit with modification on “feature1”
5. Switch back to the “main” branch
6. Modify “file1” on “main” branch
7. Create commit with modification on “main”
8. Merge branch “feature1” into “main”
9. Resolve merge conflict manually
10. Mark all conflicts resolved
11. Complete merge

# Git: Other useful commands

| | |
|---|---|
| `git push` | Pushes local branch to a remote branch |
| `git commit --amend` | Modifies the last commit on the current branch |
| `git reset --hard HEAD~1` | Undoes the last commit on the current branch |
| `git rebase -i <onto>` | Swiss-army knife of history editing! Can be used to shuffle, remove, replace, and squash commits and completely rewrite history. |

If in doubt, check out Git's man pages for a given command:

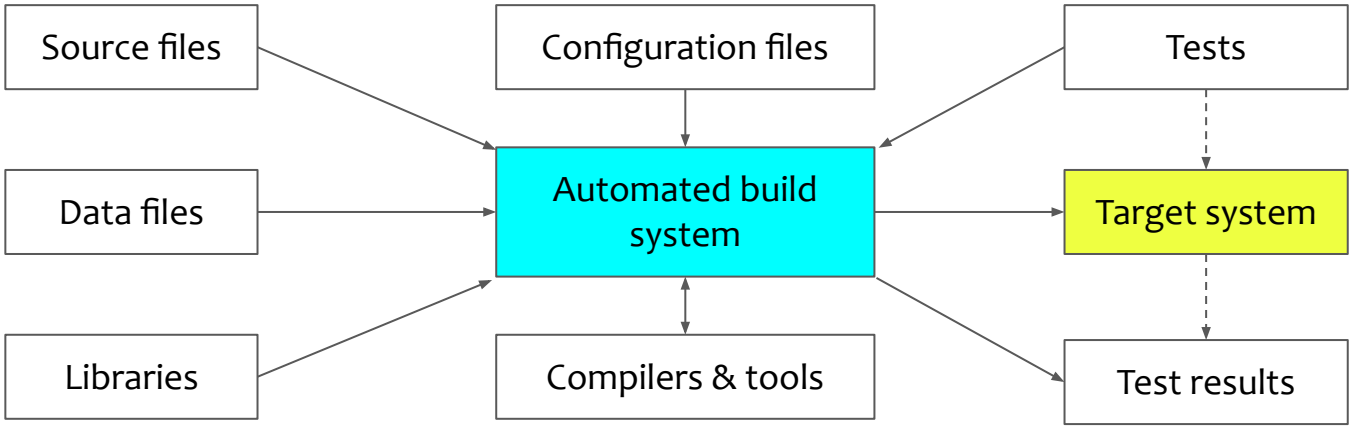
```
git help <command>
```

# Outline

- ~~**Part I:** Software configuration management~~
- **Part II:** **Build systems**
- **Part III:** Release management
- **Part IV:** Continuous *

# Build Systems

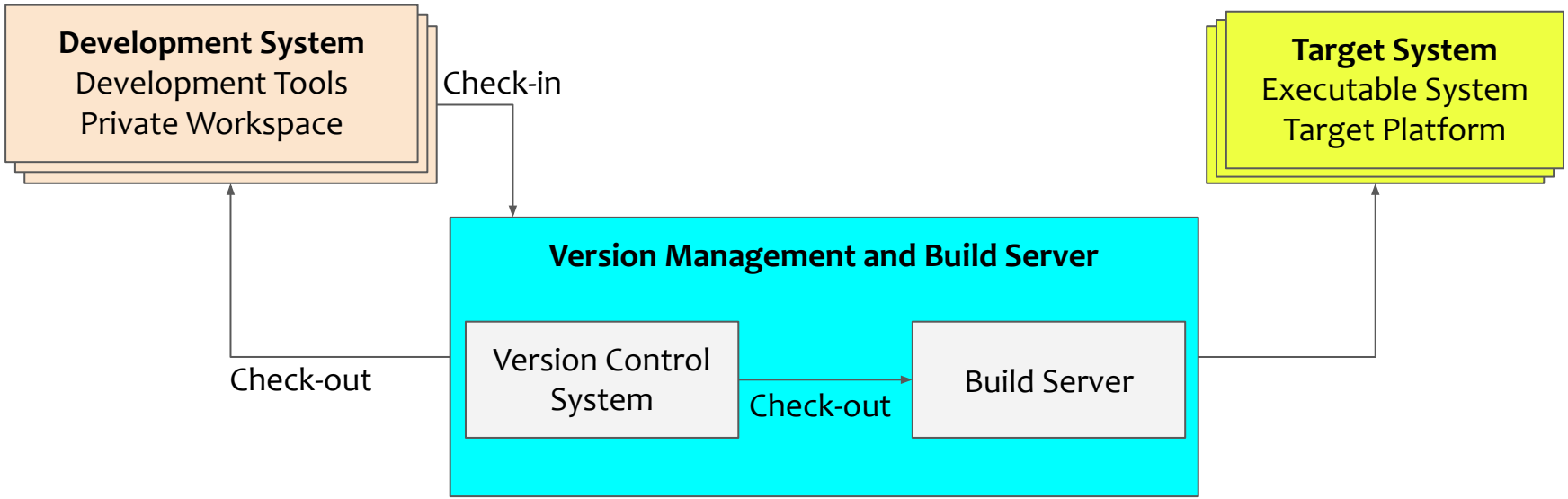
*Assemble and create complete & executable system*




📖 I. Sommerville, “System building” (Chapter 25.5), In “Software Engineering,” 2015

# Build Systems

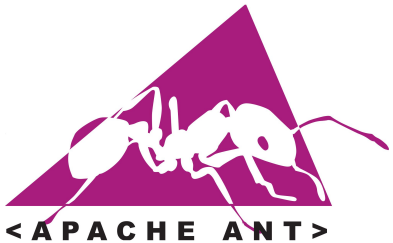
*Build system must seamlessly account for different platforms*

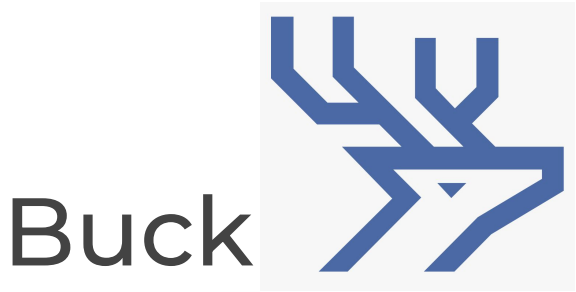



📖 I. Sommerville, “System building” (Chapter 25.5), In “Software Engineering,” 2015

# Build Systems

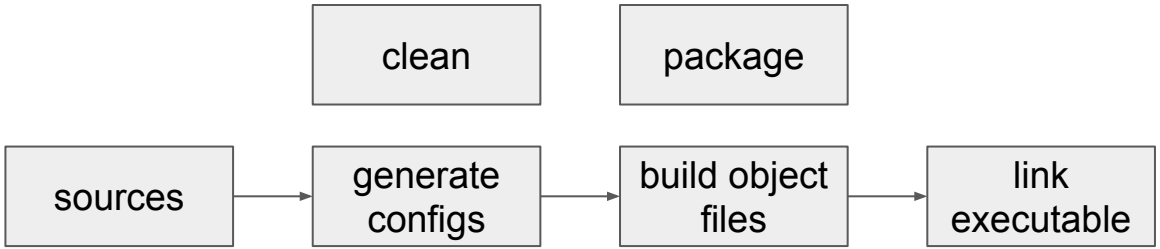



# Task-based Build Systems

- Fundamental unit of work a “task”.
- Each “task” runs a set of commands.
- Tasks in turn depend on other tasks.
- **Examples:** Ant, Maven, Gradle, Grunt, Rake, Makefiles




📖 Task-Based Build Systems (Bazel Documentation). URL: https://bazel.build/basics/task-based-builds

# Properties of Task-based Build Systems

**Advantages:**

- *Control:* Precise control over executed commands.
- *Flexibility:* Can work with any language and toolchain.

**Disadvantages:**

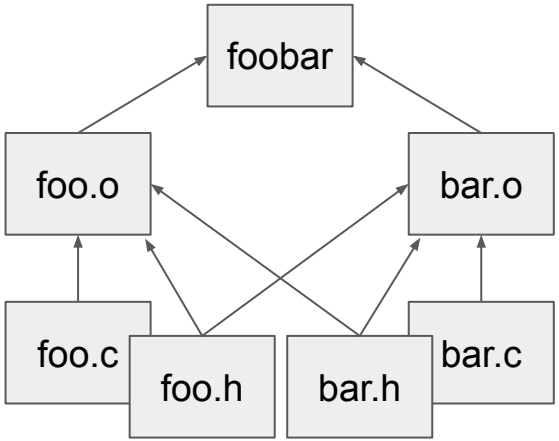
- *Parallel Builds:* Unclear if it is safe to run independent tasks concurrently ⇒ up to developer to make tasks not interfere with each other.
- *Incremental Builds:* Easy to include **hidden** dependencies in tasks ⇒ up to developer to split tasks in fine-enough granularity with all dependencies listed.
- *Maintenance:* Hard to maintain and evolve ⇒ too much flexibility and control often leads to developers taking shortcuts (which are hard to debug if builds fail).

# Makefiles

```
# Makefile
foobar: foo.o bar.o
        cc -o foobar foo.o bar.o

foo.o: foo.c foo.h bar.h
        cc -c -o foo.o foo.c
bar.o: bar.c foo.h bar.h
        cc -c -o bar.o bar.c

.PHONY: clean
clean:
        rm -f foobar *.o
```

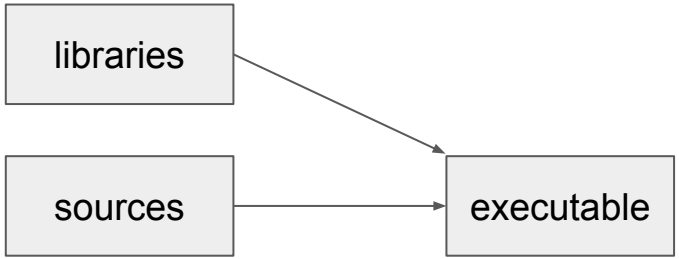
```
$ make
… builds “foobar” …

$ make clean
… removes listed outputs …

$ make foo.o
… only builds target foo.o …
```

- Each rule specifies a set of dependencies (files or other rules)
- Each rule lists a set of commands to execute to produce the output
- Rules that do not produce real files are called “phony” (e.g. “clean”)



# Artifact-based Build Systems

- Declarative build scripts describe “what” (not “how”) to build.
- Build system responsible for how to build artifacts.
- Build system in full control over which commands get executed.
- **Examples:** Bazel, Buck, Pants




📖 Artifact-Based Build Systems (Bazel Documentation). URL: https://bazel.build/basics/artifact-based-builds

# Properties of Artifact-based Build Systems

**Advantages:**

- *Parallel Builds:* Build system in full control over which commands run when, and able to safely parallelize builds.
- *Incremental Builds:* Dependencies must be specified accurately (hidden dependencies “break” the build).
- *Maintenance:* Build files follow clear patterns and are well-specified.

**Disadvantages:**

- *Control:* More difficult to execute arbitrary commands as part of build.
- *Flexibility:* Only works with supported languages and toolchains.

# Hermetic Builds

- Given the same inputs and configuration, a **hermetic build system** always returns the same output.
- Requires isolating build process from any inputs that may change arbitrarily. Examples include:
  - Embedding current date and time in binary
  - Embedding unique identifiers in binary
  - Relying on tools on host system that may be change with system upgrade

# Distributed Builds

- Build systems for large codebases must deal with tens of thousands of targets
  - Takes too long to build on a single developer machine ⇒ Productivity suffers!
- *Distributed builds* can scale builds across thousands of build servers.
  - Artifact-based builds make this possible through fine granularity of dependencies.

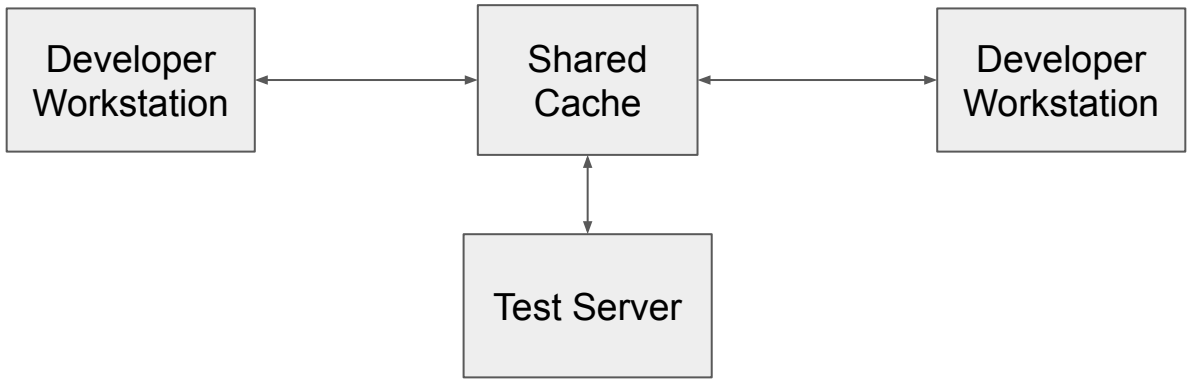
**Distributed build setups:**

- Remote caching
- Remote execution

📖 Distributed Builds (Bazel Documentation). URL: https://bazel.build/basics/distributed-builds

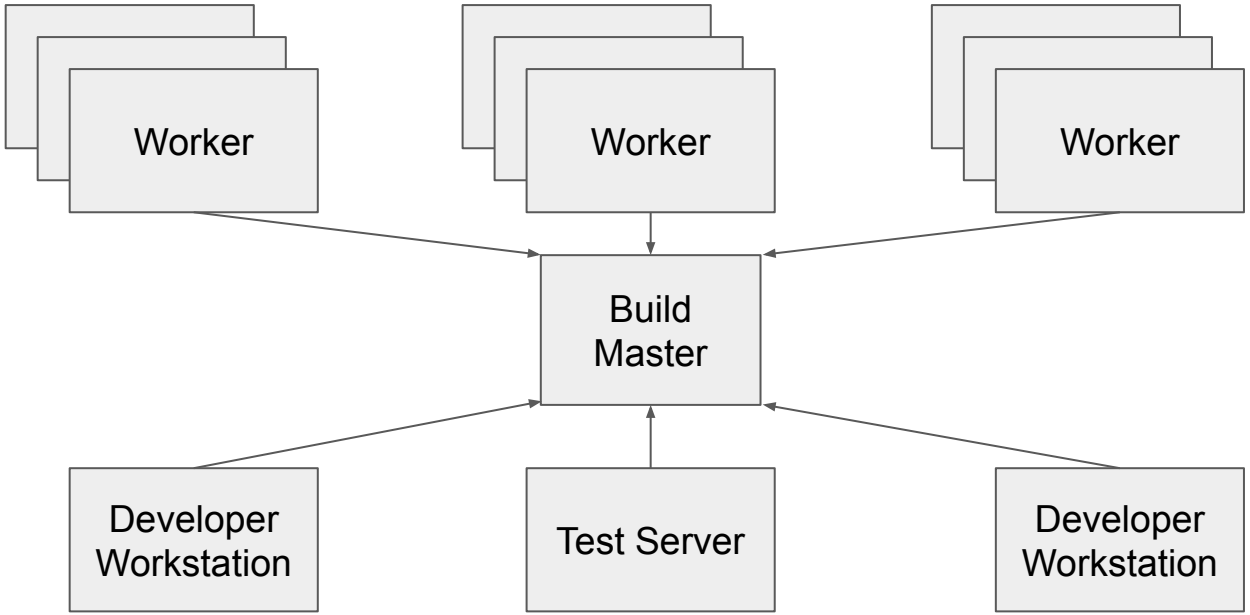
# Distributed Builds: Remote Caching

- Builds send to or get artifacts from a remote caching server.
- Requires completely reproducible (“hermetic”) builds.
- All inputs and configuration used as “key” to lookup build artifact.
- Beneficial only if downloading artifact faster than building from scratch.



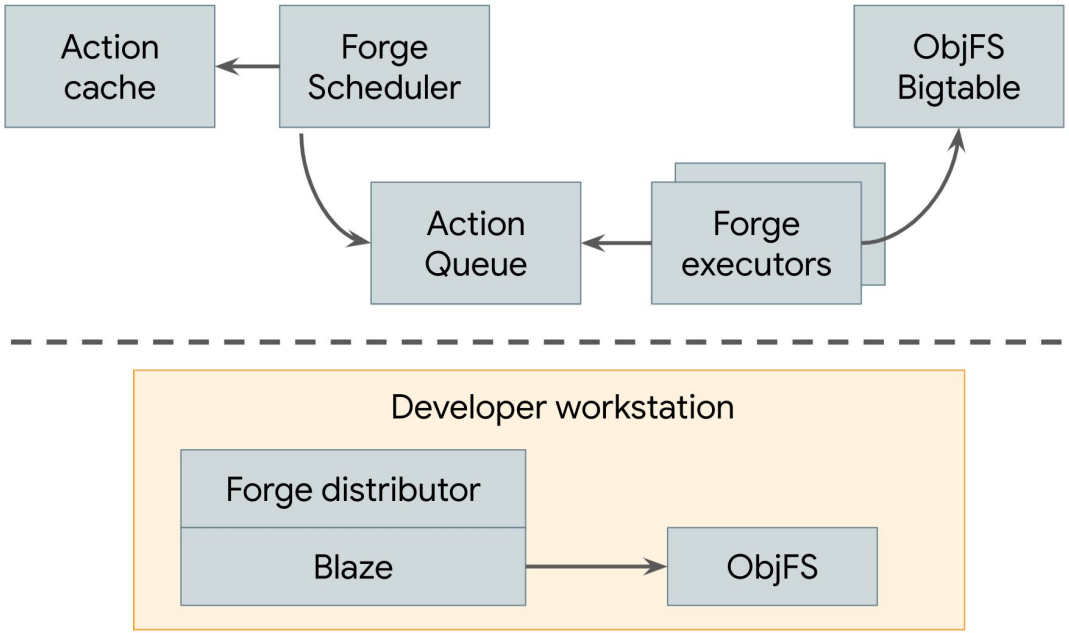
# Distributed Builds: Remote Execution

- Build requests are sent to a remote server.
- Remote server farms builds out to dedicated workers.
- Build server typically caches results itself.



# Distributed Builds: Google Case Study

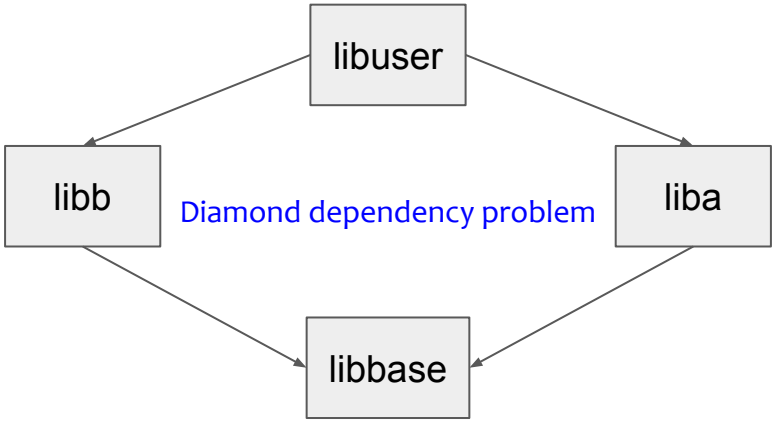
*Hybrid remote caching & execution: millions of builds producing petabytes of build outputs from billions of lines of source code every day*




**Reference:** https://bazel.build/basics/distributed-builds

# Dependency Management

- Management of complex graph of dependencies.
  - How to deal with version incompatibilities? Example: libbase makes backwards incompatible change, and liba has to be updated as well.
  - How to deal with dependencies on different versions of same library? Example: liba depends on v1 of libbase, and libb depends on v2 of libbase.




📖 T. Winters, “Dependency management” (Chapter 27), In “Software Engineering at Google,” 2020

# Dependency Management

- Granularity at which to split up project affects:
  - *Build performance*: finer granularity allows for more precise incremental builds.
  - *Testability:* each individual component can be tested independently (in parallel).
  - *Reusability*: each individual component can be reused.

- External, open source, libraries often challenging:
  - Where to acquire?
  - Which version?
  - Security and quality?

# Introduction to Bazel

- Artifact-based build system
- Supports multiple languages and toolchains
- Supports hermetic builds, distributed builds
- Cross platform


**Installation:**

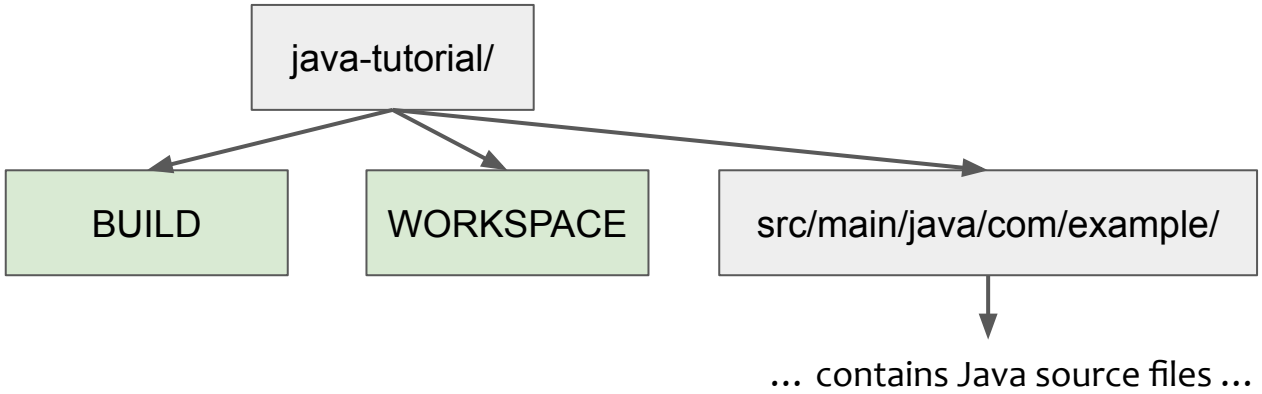
- Obtain Bazelisk binary from: https://github.com/bazelbuild/bazelisk/releases
  - A simple wrapper around the latest Bazel release
- Or follow instructions from: https://bazel.build/install

# Introduction to Bazel

```
# BUILD

load("@rules_java//java:defs.bzl", "java_binary")

java_binary(
      name = "ProjectRunner",
      srcs = glob(["src/main/java/com/example/*.java"]),
)
```

Import Java language rules from "@rules_java" repository

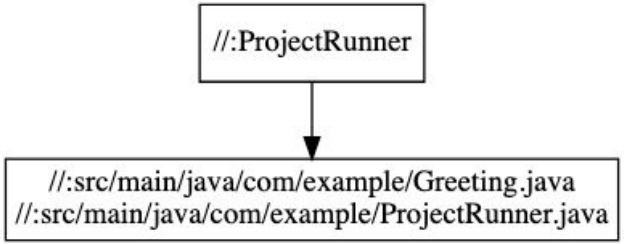
Build an executable (wrapper around .jar)

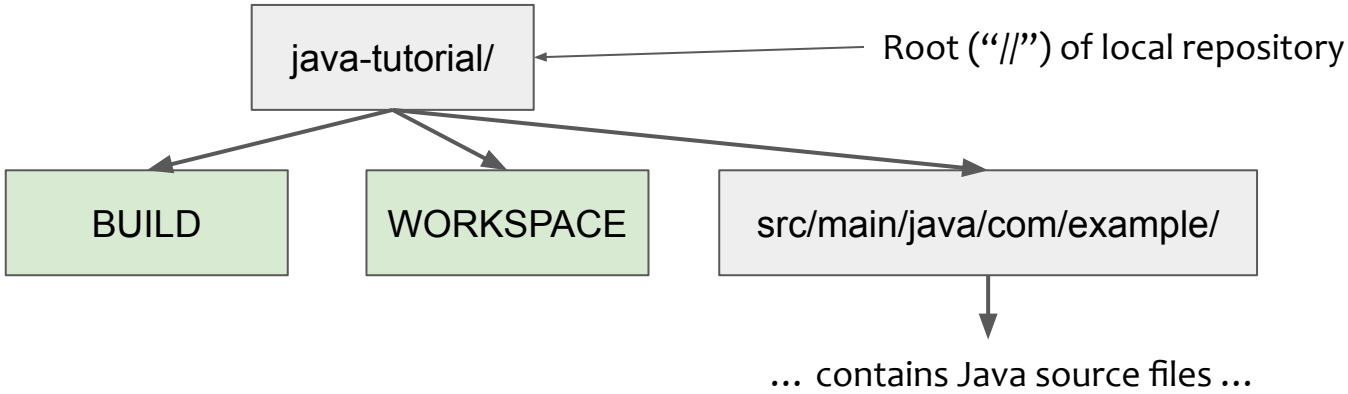



**Full example available at:** https://github.com/bazelbuild/examples/java-tutorial

# Introduction to Bazel

```
# BUILD

load("@rules_java//java:defs.bzl", "java_binary")

java_binary(
      name = "ProjectRunner",
      srcs = glob(["src/main/java/com/example/*.java"]),
)
```



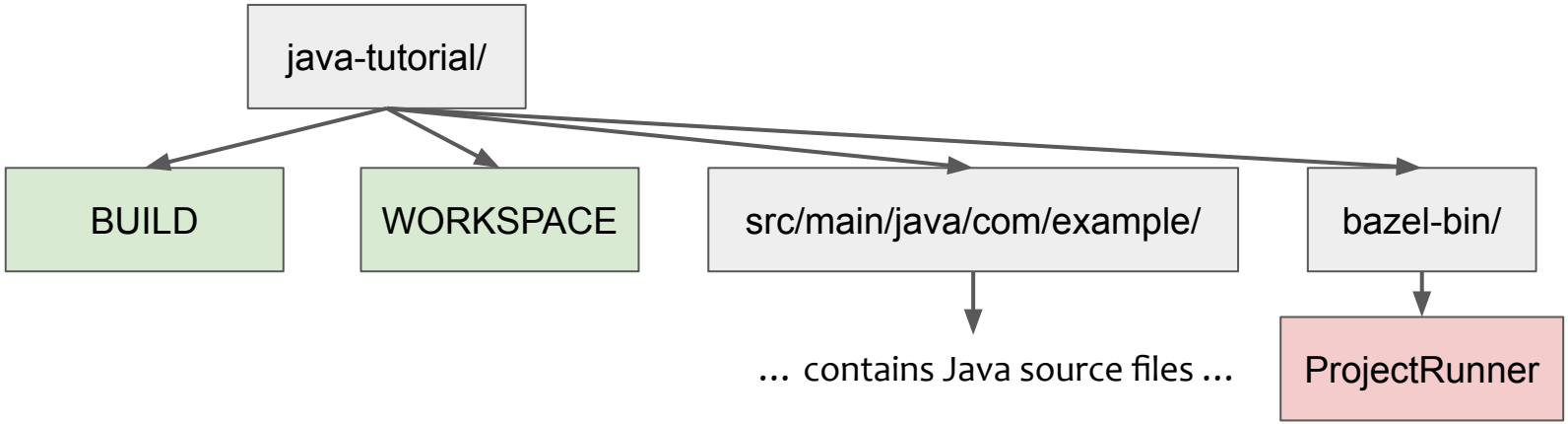




**Full example available at:** https://github.com/bazelbuild/examples/java-tutorial

# Introduction to Bazel

```
$ cd java-tutorial/
$ bazel build //:ProjectRunner
INFO: Analyzed target //:ProjectRunner (45 packages loaded,
593 targets configured).
INFO: Found 1 target...
Target //:ProjectRunner up-to-date:
  bazel-bin/ProjectRunner.jar
  bazel-bin/ProjectRunner
```

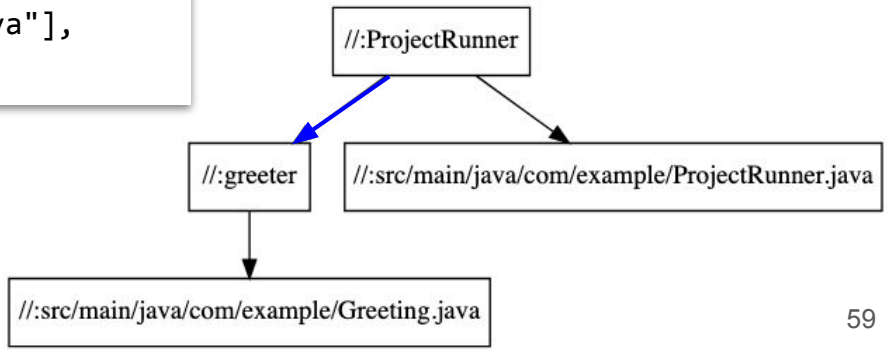
Binary target outputs to “bazel-bin”



# Introduction to Bazel

```
# BUILD

load("@rules_java//java:defs.bzl", "java_binary")

java_binary(
      name = "ProjectRunner",
      srcs = ["src/main/java/com/example/ProjectRunner.java"],
      main_class = "com.example.ProjectRunner",
      deps = [":greeter"],
)

java_library(
      name = "greeter",
      srcs = ["src/main/java/com/example/Greeting.java"],
)
```

Comma-separated list of dependencies



# References

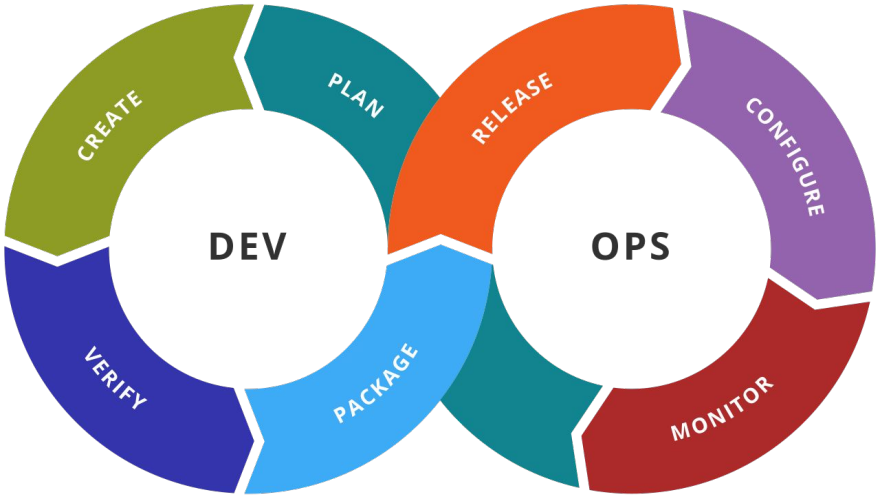

1. Ian Sommerville, “Software Engineering,” Global Edition, Pearson Education, Limited, 2015. URL: https://ebookcentral-proquest-com.eaccess.tum.edu/lib/munchentech/detail.action?docID=5831848
2. Titus Winters et al., “Software Engineering at Google”, 2020. URL: https://abseil.io/resources/swe-book
3. Scott Chacon, Ben Straub, “Pro Git,” Apress 2023. URL: git-scm.com/book
4. Bazel Documentation. URL: https://bazel.build/basics/

# Outline

- ~~**Part I:** Software configuration management~~
- ~~**Part II:** Build systems~~
- **Part III:** **Release management**
- **Part IV:** Continuous *****

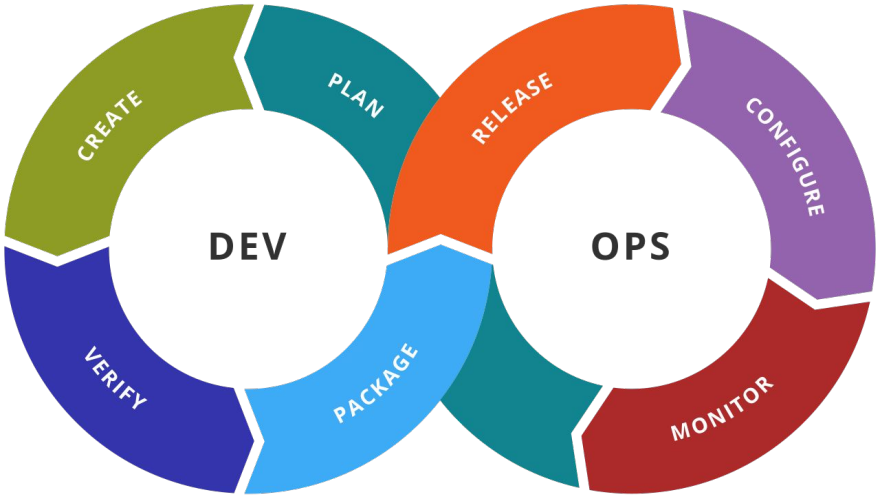
# Release Management

***Release management*** *covers the entire software development lifecycle: development, testing, releases, updates*



# Release Management

***DevOps:*** *Development philosophy aiming to combine and integrate the work of* ***dev****elopment and* ***op****eration****s*** *teams to accelerate quality software releases.*



# Release Management

**Release Management Goals:**

- Increase release frequency to deliver new features and enhancements to users.
- Reduce bottlenecks in the workflow by streamlining processes and collaboration.
- Shorten feedback loops to gather feedback and incorporate into future releases.
- Limit unplanned work by ensuring a well-defined release plan and scope.
- Reduce defects and production incidents by testing & quality assurance.
- Allow teams to add value with features and updates that align with user needs.

# Release Management Process

1. **Change request and approval:** Documentation and review process for new changes.
2. **Release planning and design:** Release scope, requirements, timeline, success criteria (performance indicators and metrics); release plan should answer what is being built, objective of the release, and value to the business and end users.
3. **Release building:** Compiling and packaging the software for deployment.
4. **Acceptance testing:** Verify functionality and adherence to requirements; includes the gathering of feedback from end-users by allowing them to use early version.
5. **Release preparation:** Final preparations, such as documentation, reaching out to users, and readiness checks; includes a final quality report by the Quality Assurance (QA) team.
6. **Release and deployment:** Controlled deployment of the software to production environment.

***Process varies based on the type of application, industry, and company requirements.***

# Software Versioning

*Software versioning is about assigning unique identifiers to software releases.*

- Helps track and manage different versions of the software.
- Common formats include Semantic Versioning which conveys information about backward compatibility and changes made.
- Many schemes possible:
  - Major-minor style: 1.0, 2.6.30
  - Year-month style: 2023.06, 2022.02, 202101

# Semantic Versioning

Given a version number `MAJOR.MINOR.PATCH`, increment the:

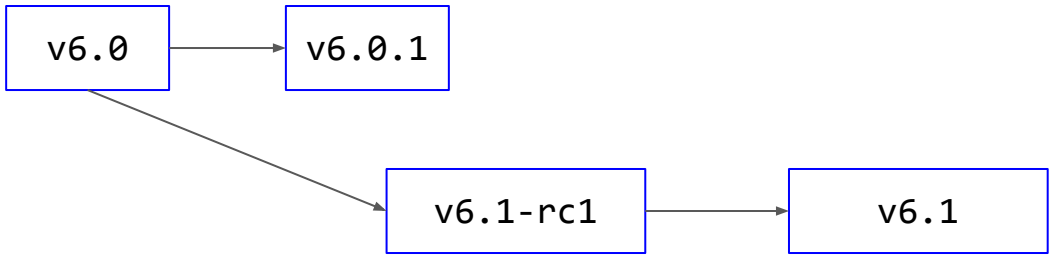
- `MAJOR` version when making incompatible API changes.
- `MINOR` version when adding functionality in a backwards compatible manner.
- `PATCH` version when making backwards compatible bug fixes.

For more details, see semver.org.

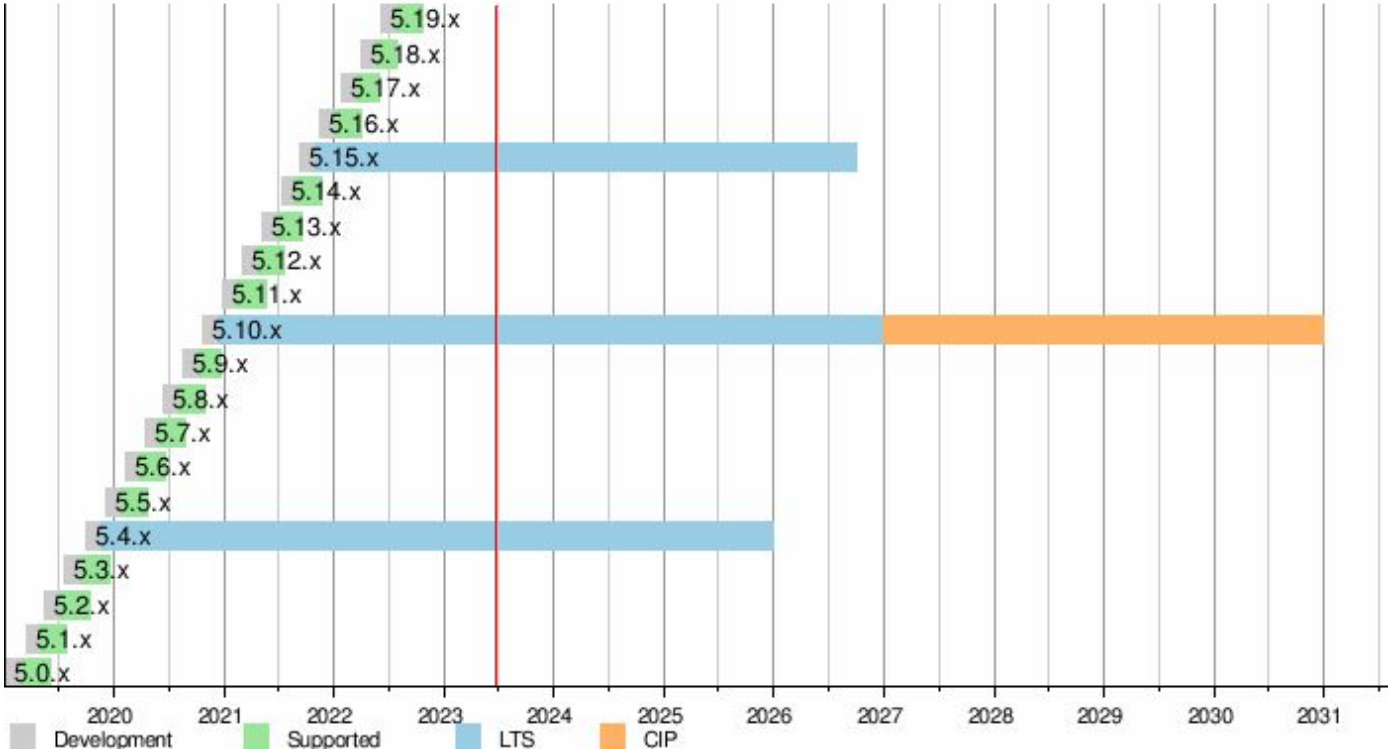
# Software Upgrades

*Transitioning from an older version to a newer version.*

- Upgrades can introduce new features, improve performance, fix bugs, and address security vulnerabilities.
- Key considerations for software upgrades include careful planning, testing, user communication, and rollback strategies if needed.



# Software Upgrades: Linux Kernel Case Study




**References:**

- https://www.kernel.org/doc/html/latest/process/2.Process.html
- https://en.wikipedia.org/wiki/Linux_kernel_version_history

LTS = Long Term Support
CIP = Civil Infrastructure Platform

# Release Management & Agile Development

*Agile principles emphasize the delivery of deployable code at any time.*
*Continuous Delivery (CD) is a critical aspect of agile development.*

Effective Agile Release Management requires at least:

- **Deployment Automation:** Use of automated tools and processes to remove bottlenecks, reduce manual effort (**CI/CD pipeline**).
- **DevOps mindset:** Collaboration, communication, and coordination between development, operations, and other stakeholders involved in the release.

# Release Management Best Practices

1. **Standardized Release Management Process:** Develop a *consistent and documented process* that defines roles, responsibilities, and workflows.
2. **Change Management:** Establishing a *structured change control process* that includes documentation, *impact analysis*, and proper *approval mechanisms* to manage and track changes.
3. **Testing and Quality Assurance:** Rigorous testing and quality checks to validate *functionality, performance, and compliance* of the software.
4. **Software Deployment:** Controlled deployment process to minimize risks, ensure proper configuration, and *facilitate rollback* if necessary; adopting *canary releases* (gradual rollout to a subset of users), *feature flagging* (enabling or disabling specific functionality without deploying new code), and A/B testing.
5. **Implement Automation:** Leverage automation tools and practices to streamline and accelerate release cycles, such as *Continuous Integration (CI), and Continuous Delivery (CD)*.
6. **Communication and Collaboration:** Maintain effective communication channels between development teams, stakeholders, and users to *ensure transparency, gather feedback, and address concerns*.
7. **Process Evolution:** Regularly *evaluate, iterate, and refine the release management process* based on lessons learned, user feedback, and emerging industry practices.

# References


- G. Kim, K. Behr, G. Spafford, “The Phoenix Project”, O’Reilly 2013. URL: https://www.oreilly.com/library/view/the-phoenix-project/9781457191350/
- B. Beyer, C. Jones, J. Petoff, N. R. Murphy, “Site Reliability Engineering”, O’Reilly 2016. URL: https://sre.google/sre-book/table-of-contents/
- B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, S. Thorne, “The Site Reliability Workbook”, O’Reilly 2018. URL: https://sre.google/workbook/table-of-contents/

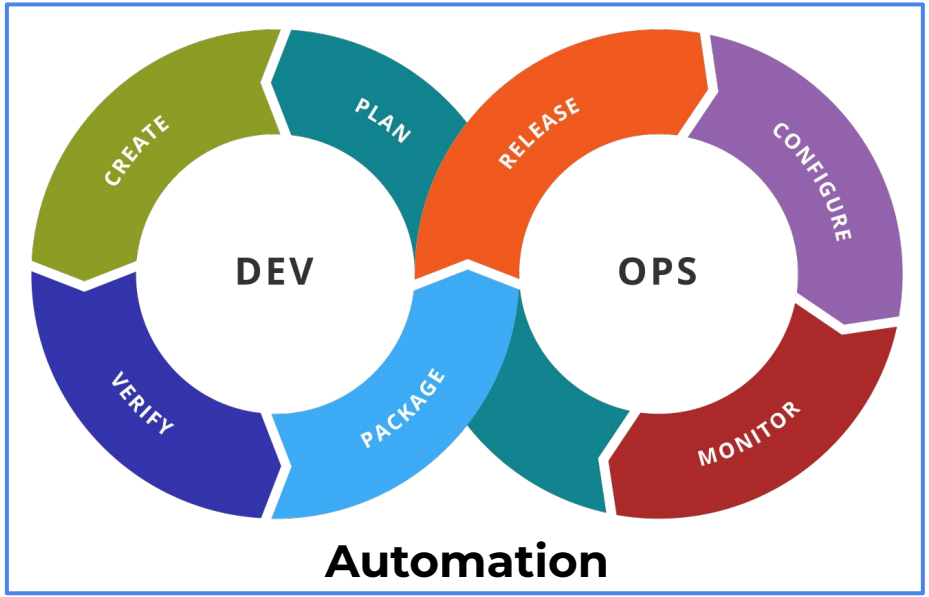
# Outline

- ~~**Part I:** Software configuration management~~
- ~~**Part II:** Build systems~~
- ~~**Part III:** Release management~~
- **Part IV: Continuous ***
  - Continuous Integration
  - Continuous Delivery
  - Continuous Deployment
  - Continuous Fuzzing

# Deployment Automation

*Process of automating steps involved in deploying software*

- Manual deployments are *time-consuming, error-prone, and inconsistent*.
- Deployment automation helps streamline the process (or parts of it).



# Deployment Automation

Benefits of Deployment Automation:

- **Increased efficiency:** Reduces manual effort, enabling faster, more frequent deployments.
- **Consistency and reliability:** Ensures consistent deployment processes, reducing the risk of human error.
- **Faster time to market:** Facilitates quicker delivery of new features and bug fixes.

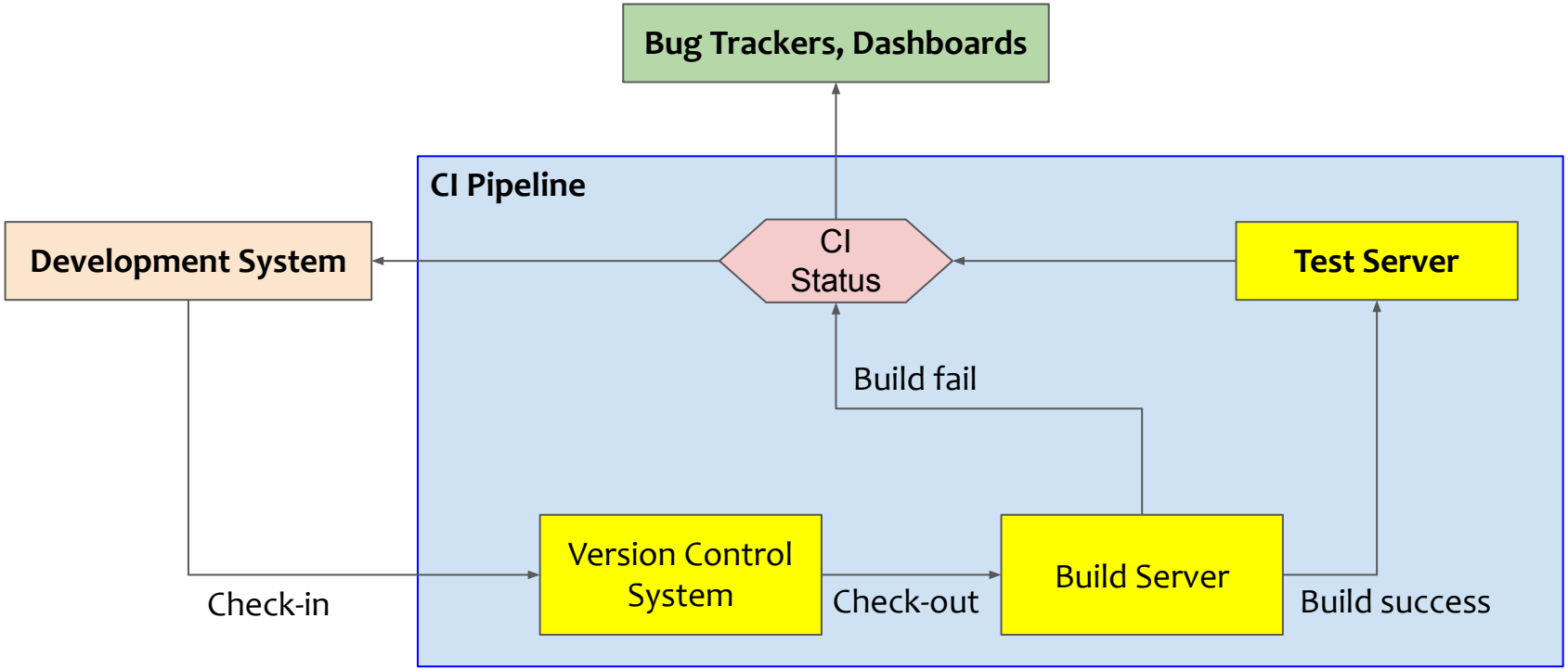
# Continuous Integration

*Continuous Integration (CI) is the practice of regularly merging code changes into a shared repository and performing automated builds and tests.*

- **Motivation:** Detect integration issues early, ensuring that the codebase is always in a working state.
- **Requirements:**
  - *Version Management:* Version control systems are used to track changes to codebase.
  - *Build Automation:* System builds can be automated, along with consistent and reproducible (hermetic) builds.
  - *Test Automation:* Tests can be executed automatically.

📖 T. Winters, “Continuous Integration” (Chapter 29), In “Software Engineering at Google,” 2020

# Continuous Integration



# Continuous Integration

Benefits of Continuous Integration:

- **Early bug detection:** Frequent integration and automated testing help identify and address issues at an early stage.
- **Code quality:** Regular code integration encourages developers to write clean, modular, and well-tested code.
- **Collaboration:** CI promotes better collaboration among teams and provides visibility into the overall project status.

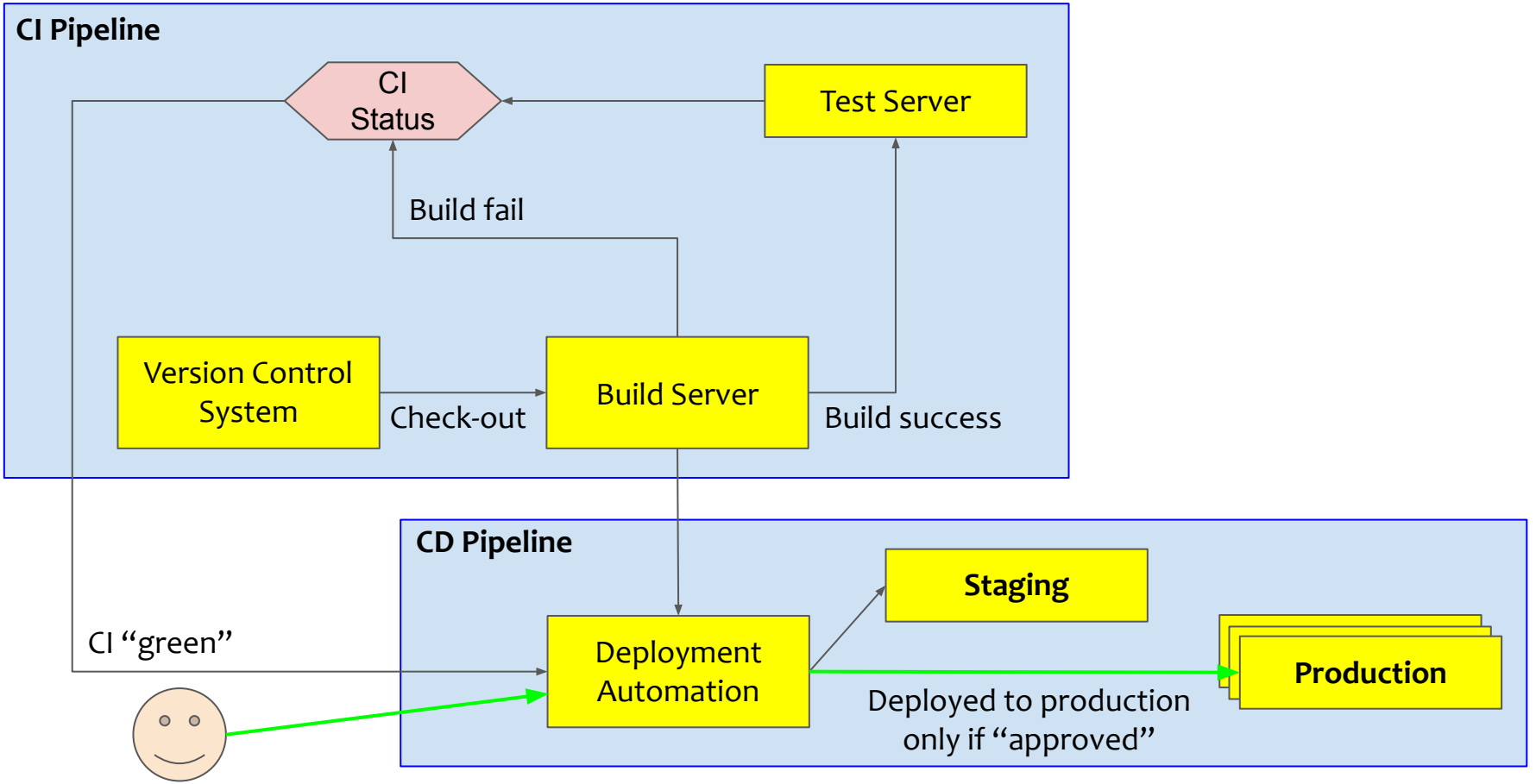
# Continuous Delivery

*Continuous Delivery (CD) is an approach that ensures software is always in a releasable state, ready for deployment.*

- **Motivation:** Automate the process of delivering software changes to production reliably and frequently.
- **Requirements:**
  - A working CI pipeline (continuous build & test).
  - System to automatically deploy packaged software to various environments, such as staging and production.

📖 T. Winters, “Continuous Delivery” (Chapter 30), In “Software Engineering at Google,” 2020

# Continuous Delivery



# Continuous Delivery

Benefits of Continuous Delivery:

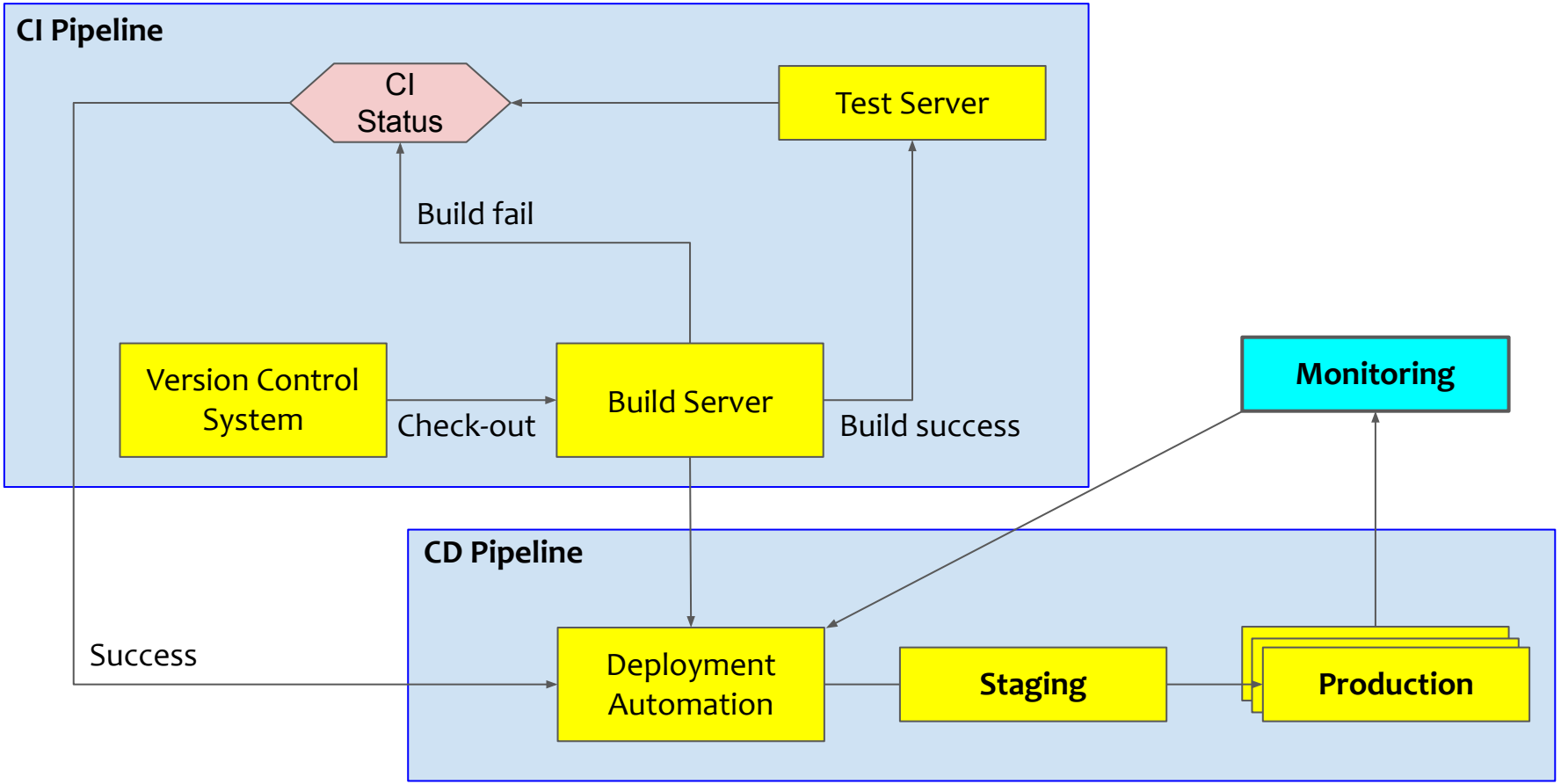
- **Faster feedback loop:** Frequent releases allow for quick user feedback, enabling faster iterations and improvements.
- **Increased customer satisfaction:** Continuous delivery enables faster delivery of new features and bug fixes, satisfying user expectations.
- **Reduced deployment risk:** Automated deployments reduce the chances of errors and inconsistencies during the deployment process.

# Continuous Deployment

*Continuous Deployment is an extension of continuous delivery, where every successful build and test automatically gets deployed to production.*

- **Automated deployment pipeline:** Continuous deployment involves an end-to-end automation of the deployment process.
- **Automated testing and validation:** Requires automated tests, including unit tests, integration tests, and acceptance tests, to ensure the quality of deployments.
- **Continuous monitoring and rollback strategies**: Requires monitoring tools and rollback mechanisms to detect and handle issues in production.

# Continuous Deployment



# Continuous Deployment

**Advantages:**

- **Faster time to market:** Enables rapid delivery of new features and bug fixes.
- **Immediate feedback loop:** Real-time feedback helps iterate and improve the software faster.
- **Enhanced reliability:** Automated testing and monitoring reduce the risk of deployment failures.

**Challenges:**

- **Risk management:** Proper testing and validation processes are essential to minimize the impact of faulty deployments.
- **Cultural and organizational:** Requires close collaboration, trust, alignment across development, operations, and business teams.

# Continuous Fuzzing

**Fuzzing:** Test programs with random, unexpected, inputs.

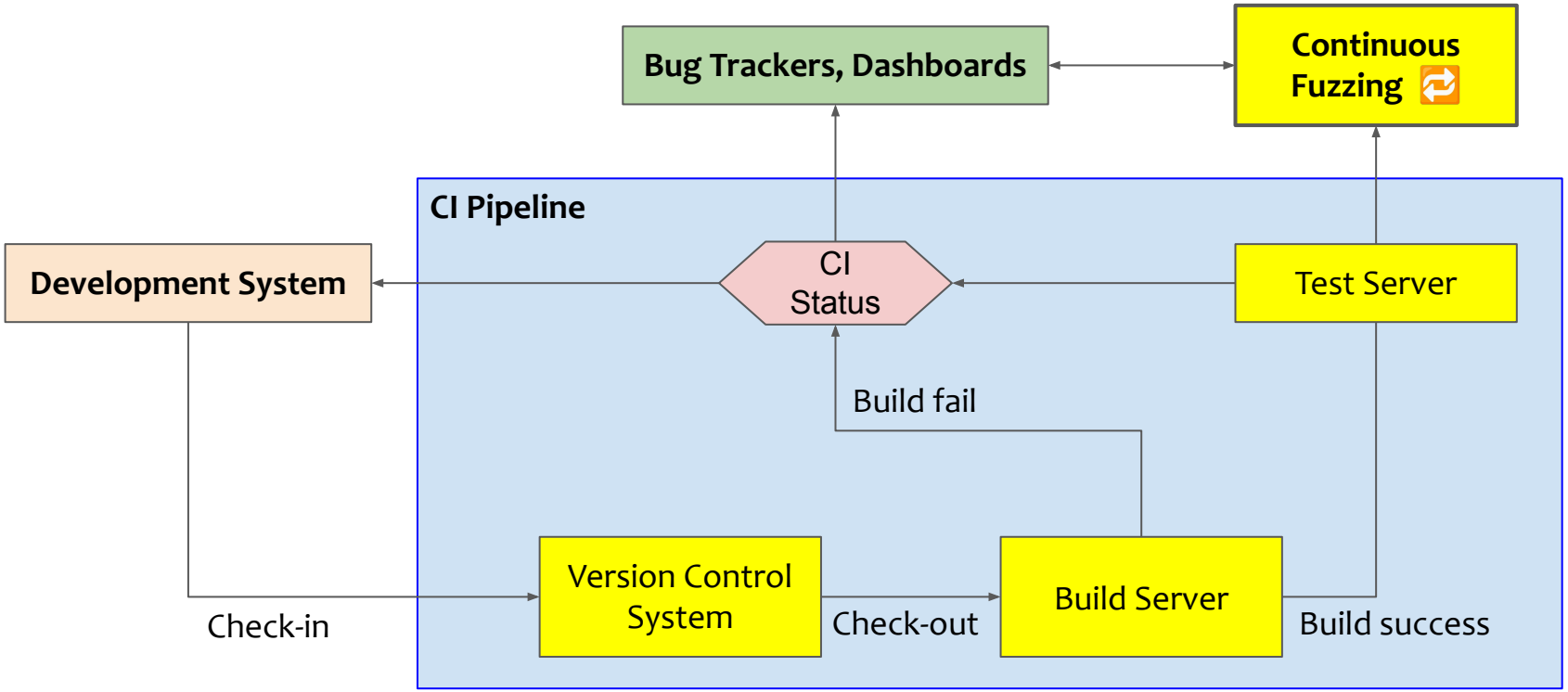
***Continuous Fuzzing:*** *Fuzzing at scale, integrated with a CI pipeline.*

- Continually run all available fuzzers.
- Monitor for crashes and report issues to developers automatically.

**Challenges:**

- Running production binaries vs. binaries with error detection enabled (**sanitizers**).
- When to restart fuzzing jobs (new version push, time based, manual).
- Issue bisection (automatically finding a bug-introducing commit).
- Reporting the same bug only once (deduplication).
- Integration with bug tracking systems.

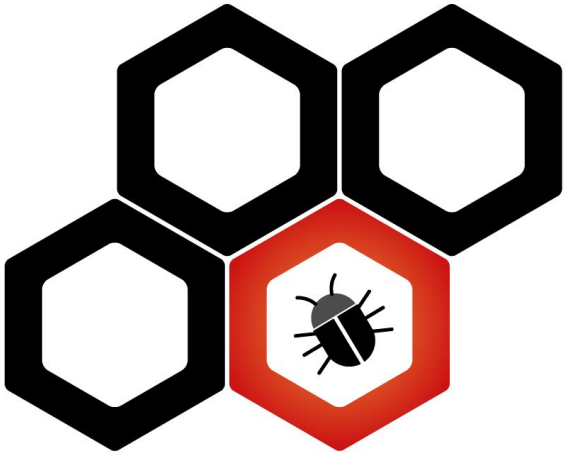
# Continuous Fuzzing



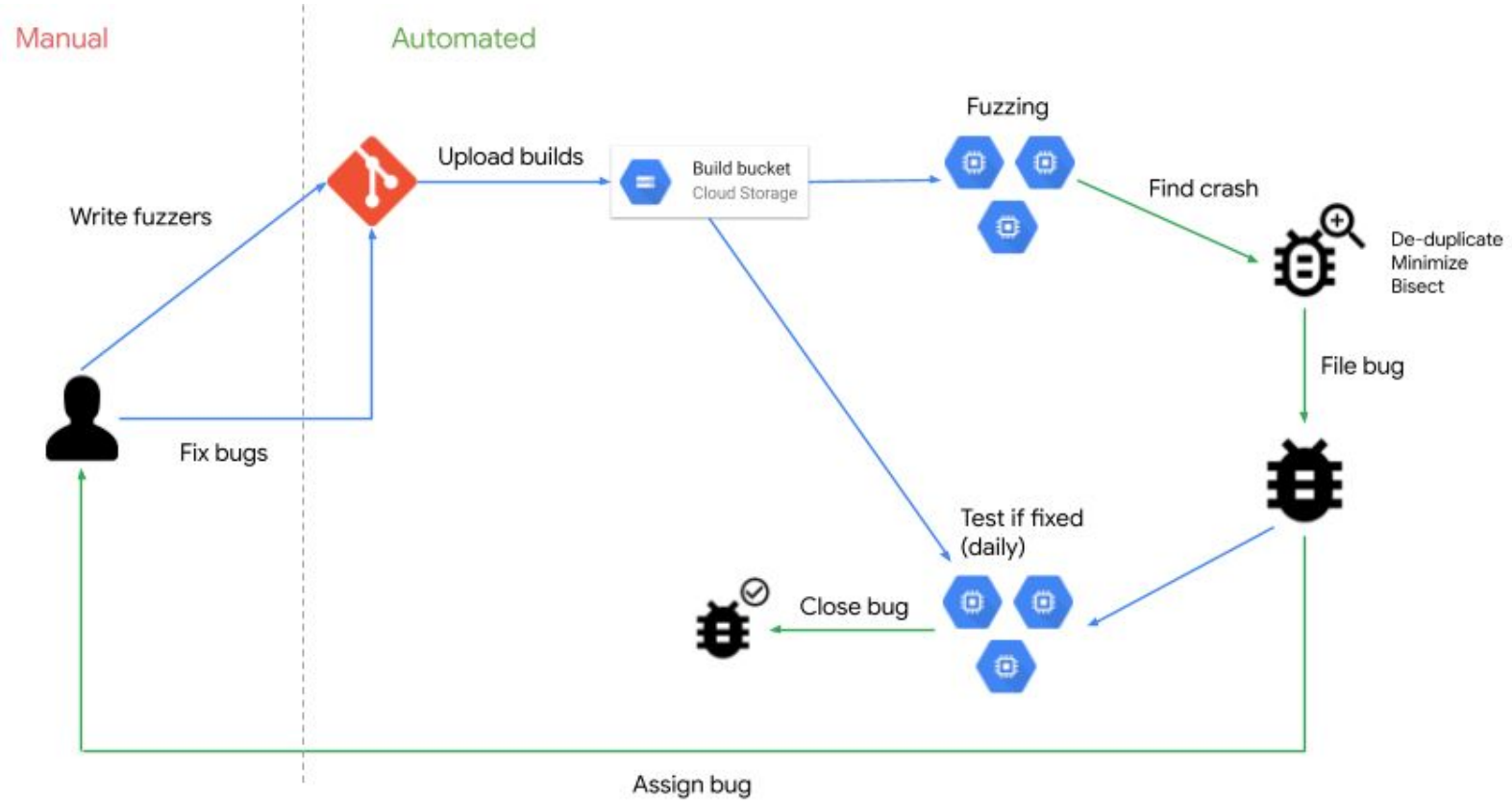
# Continuous Fuzzing

**OSS-Fuzz: Continuous Fuzzing for open source software (run by Google).**

- Fuzzing for common open source software!
- "As of February 2023, OSS-Fuzz has helped identify and fix over 8,900 vulnerabilities and 28,000 bugs across 850 projects."
- Based on Google's ClusterFuzz.



# OSS-Fuzz / ClusterFuzz Architecture




**Reference:** https://google.github.io/clusterfuzz/

# References


1. Titus Winters et al., “Software Engineering at Google”, 2020. URL: https://abseil.io/resources/swe-book
2. OSS-Fuzz. URL: https://github.com/google/oss-fuzz
3. ClusterFuzz. URL: https://github.com/google/clusterfuzz

# Summary

- **Part I:** Software configuration management
  - Source code management
  - Version control systems
  - Branch management
- **Part II:** Build systems
  - Task-based build systems
  - Artifact-based build systems
  - Distributed builds
  - Dependency management
  - Hermeticity
- **Part III:** Release management
  - Release planning
  - Software versioning
  - Software upgrades

# Summary

- **Part IV:** Continuous *
  - Continuous Integration
  - Continuous Delivery
  - Continuous Deployment
  - Continuous Fuzzing

# References


1. Ian Sommerville, “Software Engineering,” Global Edition, Pearson Education, Limited, 2015. URL: https://ebookcentral-proquest-com.eaccess.tum.edu/lib/munchentech/detail.action?docID=5831848
2. Titus Winters et al., “Software Engineering at Google”, 2020. URL: https://abseil.io/resources/swe-book
3. B. Beyer, C. Jones, J. Petoff, N. R. Murphy, “Site Reliability Engineering”, O’Reilly 2016. URL: https://sre.google/sre-book/table-of-contents/
4. B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, S. Thorne, “The Site Reliability Workbook”, O’Reilly 2018. URL: https://sre.google/workbook/table-of-contents/
5. G. Kim, K. Behr, G. Spafford, “The Phoenix Project”, O’Reilly 2013. URL: https://www.oreilly.com/library/view/the-phoenix-project/9781457191350/