# Release Engineering

# Release engineering

- Release engineering is the discipline of software engineering that can be described as **building and delivering software**.
    - Includes the use of source code management, compilers, build configuration languages, automated build tools, package managers, and installers.
- Release engineers work with devs and SREs.
- It's important that releases are repeatable and aren’t "**unique snowflakes**".
- Releasing software is a critical operation.
    - Production environments need to be stable and have a known state.
    - Typically, there is a defined procedure for releasing software.

# Self-service model

- Release automation can be a **platform**.
- Every team can use **standard tools and best practices**.
- Each team individually decides how often and when to release.
- Release processes can be heavily **automated**.
  - Only need human attention when the release deployment fails.

# High Velocity

- Rebuilt binaries frequently to **roll out features quickly**.
  - Daily releases are a common practice.
- Frequent releases result in **fewer changes between versions.**
  - Makes testing and troubleshooting easier.
- Some teams frequently create builds and select which to deploy.
- Other teams adopted a "**push on green**" approach.
  - Deploy every build that passes all tests.

# Hermetic Builds

- Builds must be **consistent and repeatable**.
  - The same product with the same revision should always build to the exact same binary.
  - Must be insensitive of locally installed libraries.
  - The build process must be self-contained: The result depends on the build system version, but not on external services.
- **Cherry picking**: How to fix a bug in software running in production.
  - Use the same build environment and revision as the original build.
  - Only apply the specific change for the bug fix.

# Continuous Build and Deployment

- Building
- Branching
- Testing
- Packaging
- Deployment

# Enforcement of Policies and Procedures

Several layers of security and access control determine who can perform specific operations (typically gated by code reviews):

- Approving source code changes
- Specifying the actions to be performed during the release process
- Creating a new release
- Approving the initial integration proposal and subsequent cherry picks
- Deploying a new release
- Making changes to a project’s build configuration

# Building and Branching

**Building**

- Build tool (Blaze / Bazel) for binaries and unit tests
- Automatically builds all dependencies

**Branching**

- Releases are build from branches in the source code repository
- Changes in this branch are never merged back into the mainline
- Bug fixes in the mainline are cherry-picked into the branch
- Avoids picking up unrelated changes in bug fix releases

# Testing

- Unit tests are run against mainline each time a change is submitted
- Build failures are detected quickly
- Release candidates are built from the last revision passing tests
- Unit tests are rerun during the release process
    - Cherry pick builds use a different codebase than mainline
    - Also can detect flaky tests
- System-level tests complement the unit tests as part of the release process

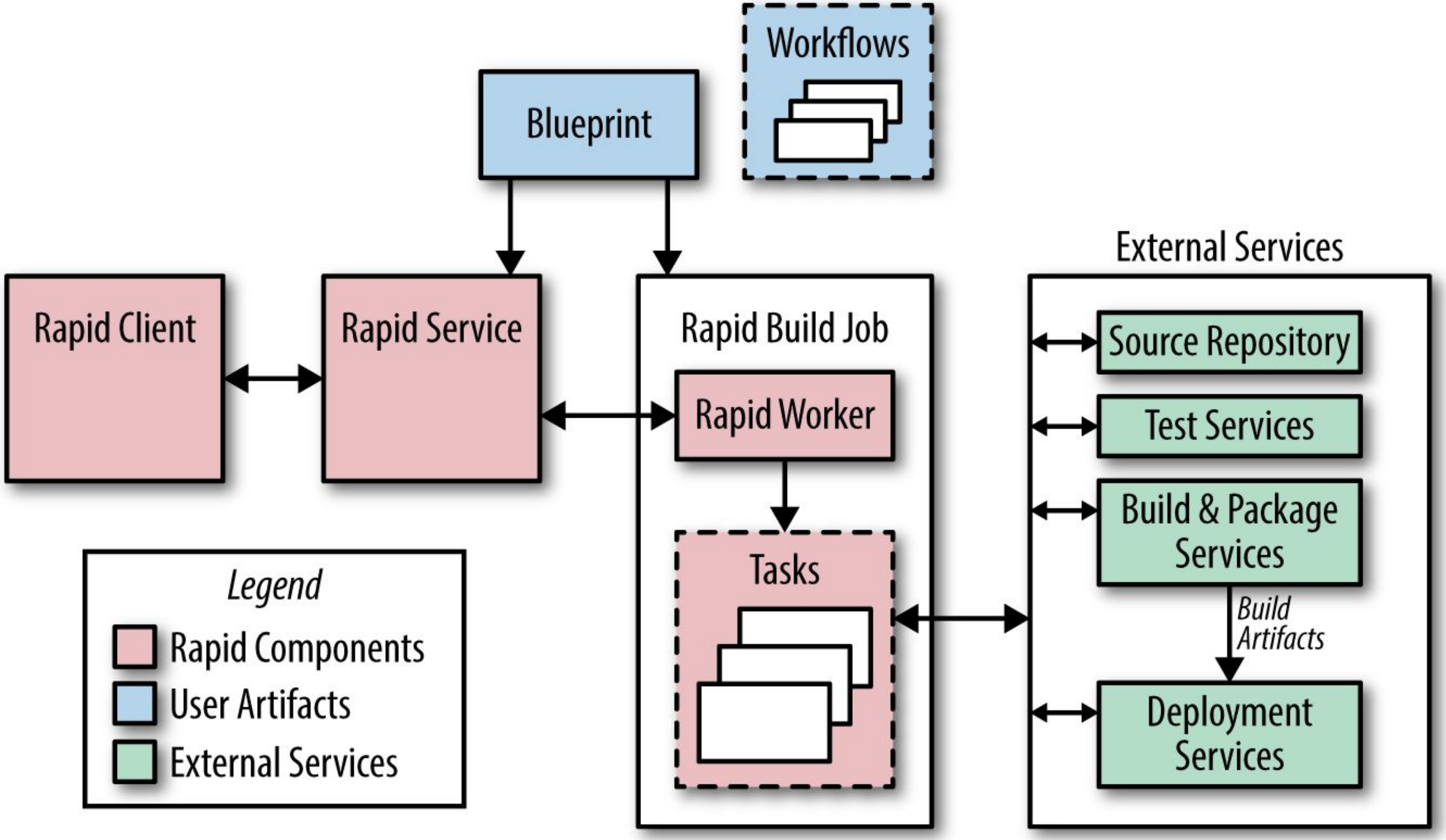
# Packaging

- Software distribution uses the Midas Package Manager (MPM)
- MPM assembles packages based on Blaze rules
- Each package has a name is signed and versioned with a unique hash
- Labels are used to indicate the package's location in the release process
    - Typical labels are dev, canary, production
- Borg jobs use the package name and label to find the right binary

# Rapid

- Rapid is the release automation service
- It uses blueprint files for configuration
  - Contain build and test targets,  rules for deployment, and administrative information
- Workflows define the actions to perform during the release process
- Typical workflow:
  - Create branch at requested revision number
  - Use Blaze to build binaries and execute unit tests
  - Deploy release to a small set of tasks for testing ("canary")
  - Roll-out to the rest of the tasks

# Rapid


Workflows
Blueprint
External Services
Rapid Client
Rapid Service
Rapid Build Job
Rapid Worker
Tasks
Source Repository
Test Services
Build & Package Services
Build Artifacts
Deployment Services
Legend
Rapid Components
User Artifacts
External Services

# Deployment

- Simple deployments can be driven by Rapid directly
- More complex deployment processes are handed-off to Sisyphus
  - Flexible deployment procedures defined as Python classes
- A deployment can contain multiple steps of incremental rollouts, e.g.,
  - A sandbox environment for devs and script testing
  - A single cluster or a small percent of tasks as a canary
  - Additional incremental canary steps (e.g. one canary on each continent)
  - Staggered rollout to the production jobs to avoid too many parallel updates in-flight

# Configuration management

- Config changes are a major cause of instability
- Config is stored in the main code repository
  - Subject to code review policies
  - Versioned
- Different strategies:
  - Use the mainline for configuration
  - Include configuration files and binaries in the same MPM package
  - Package configuration files into MPM "configuration packages".
  - Read configuration files from an external store.

# Use the mainline for configuration

- Use HEAD from repository mainline
  - Can be directly modified by devs and SREs, normal code review applies
- Binary releases and configuration changes are decoupled
- Jobs must be kept in sync with repository HEAD
- Binaries may accidentally become incompatible with configuration
  - E.g., flags set in config are not defined in binary

# Combined configuration/binaries MPM package

- Include configuration files and binaries in the same MPM package
- Limits flexibility, but is straightforward to deploy
- Useful for projects with few config files and config changes tightly coupled to releases
- Each config change requires a new release or cherry-pick

# Configuration packages

- Package configuration files into MPM configuration packages
- Applies the hermetic principle to configuration management
- Both binary and config package can be generated from the same repository version
  - Retains ability to change each package independently
  - Config changes become cherry-picks
  - Config changes does not need new binary build
- Can use labels to  indicate which versions of config and binary should be installed together

# External configuration store

- Read configuration files from an external store
    - Can be stored in Bigtable, Chubby, etc.
- Useful for services with frequent config changes
    - Especially when changes happen while the config is running (no restart required)
- May lose the advantages of the central source code repository
    - Versioned with the same numbering scheme as the binary
    - Integrated code reviews and established workflows
    - Well-known place to search for config settings

# Not only for planet-scale deployments

- Most companies struggle with the same questions:
  - How should you handle versioning of your packages?
  - Should you use a continuous build and deploy model, or perform periodic builds?
  - How often should you release?
  - What configuration management policies should you use?
  - What release metrics are of interest?
- Google has developed its own custom toolchain, but the principles apply to smaller-scale operations too.

# Start Release Engineering at the Beginning

- It’s cheaper to put good practices and process in place early
- Developers, SREs, and release engineers have to work together
  - Developers shouldn’t build and “throw the results over the fence”
- Teams should budget for release engineering resources