Release Engineering Best Practices at Google

Dinah McNutt, Senior Release Engineer

What is a Release Engineer?

- Release engineering - discipline of building and releasing software
- Skill set includes development, configuration management, test integration and sysadmin
- Experts in SCM, compilers, automated build tools, package managers and installers

Role of a Release Engineer

- Define best practices to ensure consistent and repeatable processes
- Make sure tools do the right thing by default
- Developing tools (build automation, project metrics, etc.)
- Work with SREs and project teams to develop strategies for deployment

Philosophy

- Self-service Model
- High Velocity
- Hermetic Builds
- Enforcement of Policies and Procedures

Self-Service Model

- Teams must be self-sufficient to work at scale
- Teams decide when and how often to release
- Release processes can be automated to the point of minimal effort
- Releases are truly automatic, not just automated

High Velocity

- Frequent builds have fewer changes between releases
    - Easier to troubleshoot problems
- Some teams build hourly or daily and then decide which builds to release based on test results and features
- Other teams have adopted a “Push on Green” philosophy

Hermetic Builds

her·met·ic

/hərˈmedik/

adjective

1. (of a seal or closure) complete and airtight.
   "a hermetic seal that ensures perfect waterproofing"
   synonyms: airtight, tight, sealed, zip-locked, vacuum-packed; More
2. of or relating to an ancient occult tradition encompassing alchemy, astrology, and theosophy.

Hermetic Builds

- Build tools must ensure consistency and repeatability
- Builds are insensitive to the libraries and software installed on the build machines
- Build process is self-contained
- Build tools are versioned
  - A re-build of a project released last month will use the same version of the compiler

Enforcement of Policies and Procedures - Gated Operations

- Approving source code changes
- Defining what actions are performed during a release
- Creating a release
- Deploying a release
- Making changes to the build configuration

Continuous Build and Deployment

- Rapid is our automated release system
- Leverages Google technologies to deliver release processes that are
  - Scalable
  - Hermetic
  - Reliable

Building

- Blaze (open sourced as Bazel)
  - Engineers define build targets and dependencies
  - Both are built automatically
- Rapid configuration files specify the build targets and test targets
- Rapid passes build flag to Blaze to include unique build identifier
  - We can easily associate a binary with how it was built

Branching

- All code is checked into main branch of repository
- We branch from the mainline before beginning a release
  - Changes are never merged back to the main branch
  - Bug fixes are submitted to main branch and “cherry picked” into branch

Fast Branches

- Branching is very fast
- Created instantly using a specific revision number in the main branch
- Reference created to our source-based filesystem
  - /src/depot/1234567/google
- Files are copied to the branch as needed, in the background
- Scales very well

Testing

- Continuous test system runs unit tests against the mainline each time a change is submitted
- Tests are re-run during the release process
  - Build flags are different
  - Test targets *might* be different
  - Once a cherry pick is performed, the branch probably contains a version of the code that does not exist elsewhere

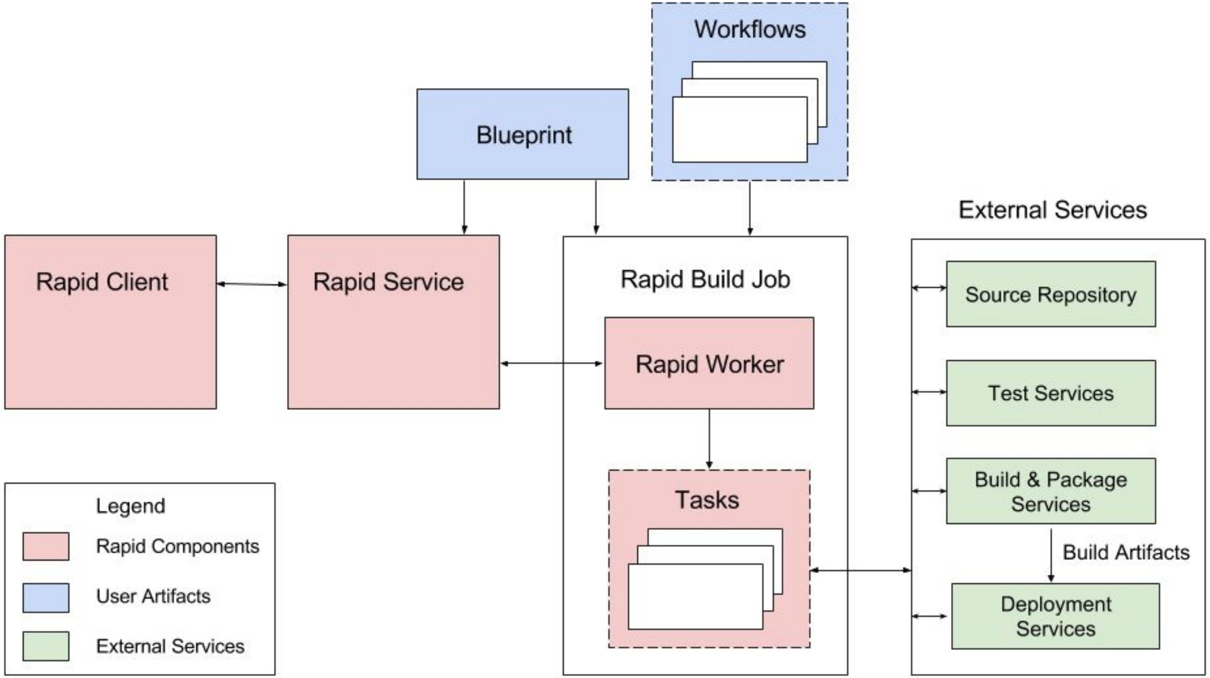
Midas Package Manager (MPM)

- MPM assemble packages based on Blaze rules:
  - build artifacts
  - owners
  - permissions
- Package metadata
  - Name (e.g. search/shakespeare/frontend)
  - Unique Hash identifier
  - Package signer (for authenticity)

MPM Labels

- Label can be applied to packages
- Useful for indicating where a package is in the release process: dev, canary, released
- Rapid applies a label containing a unique build id that makes it easy to associate the package with how it was built (e.g. shakespeare_2015_11_12_RC0)
- Packages can be installed by specifying the name and label

Continuous Build and Release System - Rapid



Typical Release Process

- Rapid uses requested revision number to create release branch
- Rapid uses Blaze to compile binaries and execute unit tests (often in parallel)
- Build artifacts are made available for system testing and deployment (usually an MPM)
- Results are logged for each step
- Report of changes since last release generated

Deployment

- Rapid can drive simple deployments directly (by updating the Borg jobs to use the newly-built MPMS)
- For more complicated deployments, we use Sisyphus

Sisyphus

- General purpose roll-out automation framework
- Developed by SRE
- Provides Python classes to support any type of rollout
- Dashboard for controlling rollout and monitoring progress
- Rapid creates a rollout in a long-running Sisyphus job
  - Uses build label to specify which MPM to rollout

Rollout Process

- Can update all jobs at once or rollout over longer period of time
- Deployment process should match risk profile
    - We might build and push hourly in pre-production environments
    - Large, user-facing services, we might start in one cluster and expand exponentially
    - Critical infrastructure services may take several days

Configuration Management

- For the purposes of this talk, defined as releasing binaries and associated configuration files
- Our approach has changed over time
- Well, actually we have adopted more approaches over time

Deployment Schemes

- Use mainline for configuration files
- Package binaries and configuration files together
- Package configuration file into config-only packages
- Read configuration files from external store

Use Mainline for Configuration Files

- Read configuration files directly from mainline
- Changes are reviewed and available immediately upon submission
- Jobs must be updated to pick up changes
- Binaries and configs are decoupled
  - Can lead to skew between running version and checked-in version

Package Binaries and Configs Together

- Ideal for projects with few configuration files or where configs change with each release
- Tightly bind configs with binaries
    - simplifies deployment - only one package to install
    - limits flexibility as new packages must be built when only config changes

Package Configs into Config-only MPMs

- Binaries can be released separately from configs
  - Cherrypick in one does not require building both
- MPM labeling indicates which MPMs should be installed together

Read Configs from External Store

- Configs that change frequently or dynamically
- Can be stored in Chubby, BigTable or our source-based file system
- We have more than one option for almost everything!

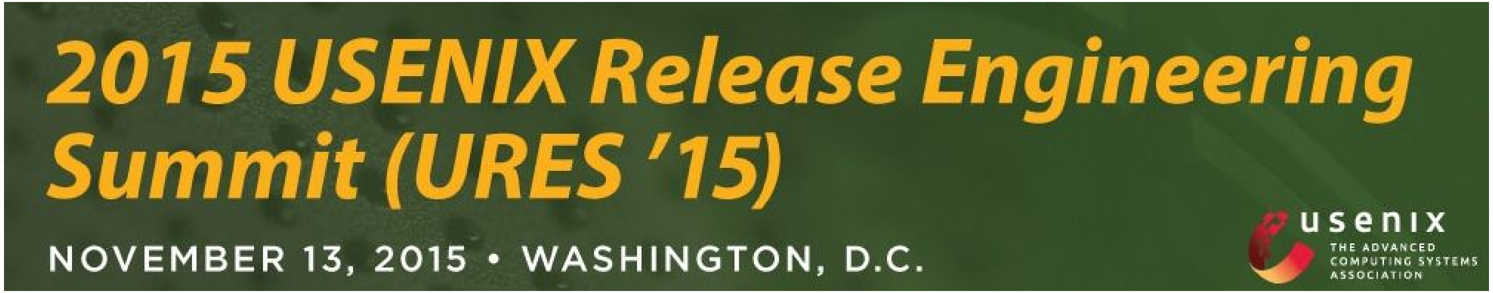
(My Personal) Conclusions

- Releasing software can be automatic, not just automated
- Solving release engineering problems “at scale” make solutions for smaller environments easier
- Many companies face the same issues (regardless of size):
  - How do you version your packages?
  - How often should you release?
  - Do you use a Push on Green model?

Release Engineering from the Beginning

- Release engineering is usually an afterthought
- Budget for it up front - cheaper in the long run
- Define best practices
- Development teams, SREs and release engineering should work together
- Discipline is still evolving

Shameless Plug #1



2015 USENIX Release Engineering Summit (URES '15)

NOVEMBER 13, 2015 • WASHINGTON, D.C.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

- Drop into URES on Friday
- Your LISA badge gets you in!
- Summit begins immediately after the Friday morning keynote in Lincoln 5
- https://www.usenix.org/conference/ures15 (or page 27 of your conference directory)

Shameless Plug #2

Look for "Site Reliability Engineering" from O'Reilly in 2016

Written by Googlers!