Principles of Software Construction: Objects, Design, and Concurrency

DevOps continued and Ethics

Michael Hilton    Bogdan Vasilescu
Administrivia

- Final Exam: Monday, May 6, 2019 05:30 p.m. - 08:30 p.m.
  - LOCATION: GHC 4401
  - Review Session Saturday, May 4th, 1-3pm in NSH 3305
Simple Layers App

Our Cool App

- Frontend
- Backend
- Server
- Database
More functionality
Even more functionality
Organize our backend

Our Cool App

Frontend

Backend Server

Mod 1

Mod 3

Mod 5

Mod 2

Mod 4

Mod 6

Database
How to scale?

Our Cool App

Frontend

Backend Server
- Mod 1
- Mod 2
- Mod 3
- Mod 4
- Mod 5
- Mod 6

Backend Server
- Mod 1
- Mod 2
- Mod 3
- Mod 4
- Mod 5
- Mod 6

Backend Server
- Mod 1
- Mod 2
- Mod 3
- Mod 4
- Mod 5
- Mod 6

Database
Monolith

- What happens when we need 100 servers?
- What if we don’t use all modules equally?
- How can we update individual models?
- Do all modules need to use the same DB, language, runtime, etc?
Microservices

Our Cool App
- Frontend
- "Dumb" Backend

NodeJS, Firebase
- Todos
  - REST service
  - Database

Google Service
- Accounts
  - REST service
  - Database

Java, MySQL
- Mailer
  - REST service
  - Database

Search Engine
- REST service
  - Database
  - Java, Neo4J

Analytics
- REST service
  - Database
  - C#, SQLServer

Facebook Crawler
- REST service
  - Database
  - Python, Firebase

AJAX
Microservice costs

- Distribution
- Eventual Consistency
- Operational complexity
- Leads to more API design decisions
Why DevOps?

• Developers and Operations don’t have the same goals
  – Devs want to push new features
  – Ops wants to keep the system available (stable, tested, etc.)
• Poor communication between Dev and Ops
• Limited capacity of operations staff
• Want to reduce time to market for new features
• Reduce “Throw it over the fence” syndrome
DevOps Definition

- “DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality.”
What are implications of DevOps?

• Quality of the code must be high
  – Testing

• Quality of the build & delivery mechanism must be high
  – Automation & more testing

• Time is split:
  – From commit to deployment to production
  – From deployment to acceptance into normal production

• Goal-oriented definition
  – May use agile methods, continuous deployment (CD), etc.
  – Likely to use tools

• Achieving it starts before committing
Microservices rely on DevOps
DevOps Toolchain

• Code — code development and review, source code management tools, code merging
• Build — continuous integration tools, build status
• Test — continuous testing tools that provide feedback on business risks
• Package — artifact repository, application pre-deployment staging
DevOps Toolchain continued

- Release — change management, release approvals, release automation
- Configure — infrastructure configuration and management, Infrastructure as Code tools
- Monitor — applications performance monitoring, end–user experience
DevOps Toolchain - Code

- **Code development** and review
- Source code management tools
- Code merging
DevOps Toolchain - Code

- Code development and **review**
- Source code management tools
- Code merging

More on Code Review in 17-313
DevOps Toolchain - Code

- Code development and review
- **Source code management tools**
- Code merging

![Git + Github](git-github.png)
DevOps Toolchain - Code

- Code development and review
- Source code management tools
- Code merging
DevOps Toolchain - Build

- Continuous integration tools
- Build status
Example

- Create Pull Request
- GitHub tells Travis CI build is mergeable
- It builds and passes tests
- Travis updates PR
- PR is merged
Findings

Information Security

Speed

Flexibility

Simplicity

Certainty

Access
Experiences

Do developers on projects with CI give (more/similar/less) value to automated tests?
27 Experiences

Do developers on projects with CI give (more/similar/less) value to automated tests?

Do projects with CI have (higher/similar/lower) test quality?
Experiences

Do developers on projects with CI give (more/similar/less) value to automated tests?

Do projects with CI have (higher/similar/lower) test quality?

Do projects with CI have (higher/similar/lower) code quality?

(B)road (F)ocused

(B)road (F)ocused
Experiences

Do developers on projects with CI give (more/similar/less) value to automated tests?

Do projects with CI have (higher/similar/lower) test quality?

Do projects with CI have (higher/similar/lower) code quality?

Are developers on projects with CI (more/similar/less) productive?

(B)road (F)ocused
BRANCH WORKFLOWS

https://www.atlassian.com/git/tutorials/comparing-workflows
1. Centralized workflow

- Central repository to serve as the single point-of-entry for all changes to the project
- Default development branch is called `master`
  - all changes are committed into master
  - doesn’t require any other branches
Example

John works on his feature
Example

Mary works on her feature
Example

John publishes his feature
Example

John publishes his feature

git push origin master
Example

Mary tries to publish her feature

git push origin master
error: failed to push some refs to '/path/to/repo.git' hint: Updates were rejected because the tip of your current branch is behind hint: its remote counterpart. Merge the remote changes (e.g. 'git pull') hint: before pushing again. hint: See the 'Note about fast-forwards' in 'git push --help' for details.

Mary tries to publish her feature

git push origin master
Example

Mary rebases on top of John’s commit(s)

git pull --rebase
origin master
Mary’s Repository

Origin/Master

Master

Master
Example

Mary resolves a merge conflict
Example

```
git rebase --continue
```
Example

Mary successfully publishes her feature
2. Git Feature Branch Workflow

• *All* feature development should take place in a dedicated branch instead of the master branch

• Multiple developers can work on a particular feature without disturbing the main codebase
  – master branch will never contain broken code (enables CI)
  – Enables pull requests (code review)
Example

```
mary begins a new feature

```

```
git checkout -b marys-feature master

```

```
git status

git add <some-file>
git commit
```
Example

Mary goes to lunch

git push -u origin marys-feature
Example

Mary finishes her feature

git push
Example

Bill receives the pull request
Example

Mary makes the changes
Example - Merge pull request

Mary publishes her feature

```bash
git checkout master
git pull
git pull origin marys-feature
git push
```
3. Gitflow Workflow

- Strict branching model designed around the project release
  - Suitable for projects that have a scheduled release cycle
- Branches have specific roles and interactions
- Uses two branches
  - master stores the official release history; tag all commits in the master branch with a version number
  - develop serves as an integration branch for features
GitFlow feature branches (from develop)
GitFlow release branches (eventually into master)

no new features after this point—only bug fixes, docs, and other release tasks
GitFlow hotfix branches

used to quickly patch production releases
Summary

• Version control has many advantages
  – History, traceability, versioning
  – Collaborative and parallel development

• Collaboration with branches
  – Different workflows

• From local to central to distributed version control
DEVELOPMENT AT SCALE
Releasing at scale in industry

- Facebook: https://atscaleconference.com/videos/rapid-release-at-massive-scale/
- Google: https://www.slideshare.net/JohnMicco1/2016-0425-continuous-integration-at-google-scale
- F8 2015 - Big Code: Developer Infrastructure at Facebook's Scale: https://www.youtube.com/watch?v=X0VH78ye4yY
Pre-2017 release management model at Facebook
Diff lifecycle: local testing

- Write Code
- Test Locally
- Diff

Test and lint locally
Diff lifecycle: CI testing (data center)
Diff lifecycle: diff ends up on master
Release every two weeks
Quasi-continuous push from master (1,000+ devs, 1,000 diffs/day); 10 pushes/day
Aside: Key idea – fast to deploy, slow to release

Dark launches at Instagram

- **Early**: Integrate as soon as possible. Find bugs early. Code can run in production about 6 months before being publicly announced ("dark launch").
- **Often**: Reduce friction. Try things out. See what works. Push small changes just to gather metrics, feasibility testing. Large changes just slow down the team. Do dark launches, to see what performance is in production, can scale up and down. "*Shadow infrastructure* is too expensive, just do in production.
- **Incremental**: Deploy in increments. Contain risk. Pinpoint issues.
Aside: Feature Flags

Typical way to implement a dark launch.

http://martinfowler.com/bliki/FeatureToggle.html
Issues with feature flags

Feature flags are “technical debt”

Diff lifecycle: in production
What’s in a weekly branch cut? (The limits of branches)
Post-2017 release management model at Facebook

Quasi-continuous web release

- Release
- Test
- Build
- Deploy
Google: similar story. HUGE code base

<table>
<thead>
<tr>
<th><strong>Google repository statistics</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>As of Jan 2015</strong></td>
</tr>
<tr>
<td><strong>Total number of files</strong></td>
</tr>
<tr>
<td><strong>Number of source files</strong></td>
</tr>
<tr>
<td><strong>Lines of code</strong></td>
</tr>
<tr>
<td><strong>Depth of history</strong></td>
</tr>
<tr>
<td><strong>Size of content</strong></td>
</tr>
<tr>
<td><strong>Commits per workday</strong></td>
</tr>
</tbody>
</table>

*The total number of files includes source files copied into release branches, files that are deleted at the latest revision, configuration files, documentation, and supporting data files.*
Exponential growth

Millions of changes committed (cumulative)
Google Speed and Scale

- >30,000 developers in 40+ offices
- 13,000+ projects under active development
- 30k submissions per day (1 every 3 seconds)

- All builds from source
- 30+ sustained code changes per minute with 90+ peaks
- 50% of code changes monthly
- 150+ million test cases / day, > 150 years of test / day
- Supports continuous deployment for all Google teams!
Google code base vs Linux kernel code base

Some perspective

Linux kernel
• 15 million lines of code in 40 thousand files (total)

Google repository
• 15 million lines of code in 250 thousand files changed per week, by humans
• 2 billion lines of code, in 9 million source files (total)
How do they do it?
1. Lots of (automated) testing

Google workflow

- Sync user workspace to repo
- Write code
- Code review
- Commit

- All code is reviewed before commit (by humans and automated tooling)
- Each directory has a set of owners who must approve the change to their area of the repository
- Tests and automated checks are performed before and after commit
- Auto-rollback of a commit may occur in the case of widespread breakage
2. Lots of automation

### Additional tooling support

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critique</td>
<td>Code review</td>
</tr>
<tr>
<td>CodeSearch*</td>
<td>Code browsing, exploration, understanding, and archeology</td>
</tr>
<tr>
<td>Tricorder**</td>
<td>Static analysis of code surfaced in Critique, CodeSearch</td>
</tr>
<tr>
<td>Presubmits</td>
<td>Customizable checks, testing, can block commit</td>
</tr>
<tr>
<td>TAP</td>
<td>Comprehensive testing before and after commit, auto-rollback</td>
</tr>
<tr>
<td>Rosie</td>
<td>Large-scale change distribution and management</td>
</tr>
</tbody>
</table>


** See "Tricorder: Building a program analysis ecosystem". In International Conference on Software Engineering (ICSE), 2015
3. Smarter tooling

- Build system
- Version control
- ...


3a. Build system
Standard Continuous Build System

- Triggers builds in continuous cycle
- Cycle time = longest build + test cycle
- Tests many changes together
- Which change broke the build?
Google Continuous Build System

- Triggers tests on every change
- Uses fine-grained dependencies
- Change 2 broke test 1

![Diagram showing test dependencies and failures for changes 1, 2, and 3.](image-url)
Google Benefits

- Identifies failures sooner
- Identifies culprit change precisely
  - Avoids divide-and-conquer and tribal knowledge
- Lower compute costs using fine grained dependencies
- Keeps the build green by reducing time to fix breaks
- Accepted enthusiastically by product teams
- Enables teams to ship with fast iteration times
  - Supports submit-to-production times of less than 36 hours for some projects
Costs

- Requires enormous investment in compute resources (it helps to be at Google) grows in proportion to:
  - Submission rate
  - Average build + test time
  - Variants (debug, opt, valgrind, etc.)
  - Increasing dependencies on core libraries
  - Branches

- Requires updating dependencies on each change
  - Takes time to update - delays start of testing
Which tests to run?

**GMAIL**
Test Target:
- name: //depot/gmail_client_tests
- name: //depot/gmail_server_tests

**BUZZ**
Test targets:
- name: //depot/buzz_server_tests
- name: //depot/buzz_client_tests
Scenario 1: a change modifies common_collections_util
Scenario 1: a change modifies common_collections_util

When a change modifying common_collections_util is submitted.
Scenario 1: a change modifies common_collections_util

When a change modifying common_collections_util is submitted.
Scenario 1: a change modifies common_collections_util

All tests are affected! Both GMail and Buzz projects need to be updated.

When a change modifying common_collections_util is submitted.
Scenario 2: a change modifies the youtube_client

When a change modifying youtube_client is submitted.
Scenario 2: a change modifies the youtube_client

Only buzz_client_tests are run and only Buzz project needs to be updated.

When a change modifying youtube_client is submitted.
3b. Version control

- Problem: even git can get slow at Facebook scale
  - 1M+ source control commands run per day
  - 100K+ commits per week
3b. Version control

- Solution: redesign version control
3b. Version control

• Solution: redesign version control
  – Query build system's file monitor, Watchman, to see which files have changed
3b. Version control

• Solution: redesign version control
  – Query build system's file monitor, Watchman, to see which files have changed → 5x faster “status” command
3b. Version control

- Solution: redesign version control
  - Sparse checkouts?? (remember, git is a distributed VCS)
3b. Version control

• Solution: redesign version control
  – Sparse checkouts:
  – Change the clone and pull commands to download only the commit metadata, while omitting all file changes (the bulk of the download)
  – When a user performs an operation that needs the contents of files (such as checkout), download the file contents on demand using existing memcache infrastructure
3b. Version control

- Solution: redesign version control
  - Sparse checkouts → **10x faster clones and pulls**
  - Change the clone and pull commands to download only the commit metadata, while omitting all file changes (the bulk of the download)
  - When a user performs an operation that needs the contents of files (such as checkout), download the file contents on demand using existing memcache infrastructure
4. Monolithic repository
Monolithic repository – no major use of branches for development

Trunk-based development

Combined with a centralized repository, this defines the monolithic model

- Piper users work at “head”, a consistent view of the codebase
- All changes are made to the repository in a single, serial ordering
- There is no significant use of branching for development
- Release branches are cut from a specific revision of the repository
Did it work? Yes. Sustained productivity at Facebook

Lines Committed Per Developer Per Day

Growth of the size of the Android and iOS dev teams
MONOREPO VS MANY REPOS
A recent history of code organization

• A single team with a monolithic application in a single repository
  ...
• Multiple teams with many separate applications in many separate repositories
• Multiple teams with many separate applications microservices in many separate repositories
• A single team with many microservices in many repositories
  ...
• Many teams with many applications in one big Monorepo
What is a monolithic repository (monorepo)?

• A **single** version control repository containing multiple
  – Projects
  – Applications
  – Libraries
• often using a common build system.
History of Version Control

Before Git/Mercurial we all used Subversion and monorepos where widespread.
What is a Monolithic Repository (monorepo)?

A **single** version control repository containing multiple
- projects
- applications
- libraries,

often using a common build system.
Monorepos in industry

Google (computer science version)

COMMUNICATIONS OF THE ACM

CONTRIBUTED ARTICLES

Why Google Stores Billions of Lines of Code in a Single Repository

By Rachel Potvin, Josh Levenberg
Communications of the ACM, Vol. 59 No. 7, Pages 78-87
10.1145/2854146
Comments (3)

Early Google employees decided to work with a shared codebase managed through a centralized source control system. This approach has served Google well for more than 16 years, and today the vast majority of Google's software assets continues to be stored in a single, shared repository. Meanwhile, the number of Google software developers has steadily increased, and the size of the Google codebase has grown exponentially (see Figure 1). As a result, the technology used to host the codebase has also evolved significantly.

Back to Top

Key Insights
Scalable Mercurial at Facebook

© 7 January 2014 · INFRA · OPEN SOURCE · PERFORMANCE · OPTIMIZATION

Scaling Mercurial at Facebook

Durham Goode · Siddharth P. Agarwal

With thousands of commits a week across hundreds of thousands of files, Facebook's main source repository is enormous—many times larger than even the Linux kernel, which checked in at 17 million lines of code and 44,000 files in 2013. Given our size and complexity—and Facebook's practice of shipping code twice a day—improving our source control is one way we help our engineers move fast.

Choosing a source control system

Two years ago, as we saw our repository continue to grow at a staggering rate, we sat down and extrapolated our growth forward a few years. Based on those projections, it appeared likely that our then-current technology, a Subversion server with a Git mirror, would become a productivity bottleneck very soon. We looked at the available options and found none that were both fast and easy to use at scale.

Our code base has grown organically and its internal dependencies are very complex. We could have spent a lot of time making it more modular in a way that would be friendly to a source control tool, but there are a number of benefits to using a single repository. Even at our current scale, we often make large changes throughout our code base, and having a single repository is useful for continuous

Recommended

- Scaling memcached at Facebook
- Flashcache at Facebook: From 2010 to 2013 and beyond
Monorepos in industry

Microsoft claim the largest git repo on the planet
Monorepos in open-source

foresquare public monorepo

<table>
<thead>
<tr>
<th>Branch: master</th>
<th>New pull request</th>
</tr>
</thead>
<tbody>
<tr>
<td>538 commits</td>
<td>1 branch</td>
</tr>
</tbody>
</table>

A monorepo that holds all of Foursquare’s opensource projects

- pants
- foursquare
- monorepo
- mongdb
- rogue
- scala

- **mator** committed with **mator**: Upgrade Fsq.io Travis config to use mongdb3.0+ (#780) 3 months ago
- **build-support**: Monolithic Ivy resolve commit (#530) 3 months ago
- **scripts/fsqio**: Add a check for the current file before deleting (#709) 3 months ago
- **src**: Add installation instructions to pom 3 months ago
- **test**: Spindle: Make ThriftParserTest actually depend on its input (#735) 3 months ago
- **.dockerignore**: Update fsqio/fsqio Dockerfile and add one for fsqio/twofishes 2 years ago
- **.gitignore**: Update upkeep to no longer clobber global variables 10 months ago
- **.travis.yml**: Upgrade Fsq.io Travis config to use mongdb3.0+ (#780) 3 months ago
- **BUILD.opensource**: Monolithic Ivy resolve commit (#530) 3 months ago
- **BUILD.tools**: Drop a BUILD.tools in Fsq.io 8 months ago
- **CLA.md**: Move deployed files to consolidated directory 2 years ago
- **CONTRIBUTING.md**: Post a CONTRIBUTING.md 2 years ago
The Symfony monorepo

43 projects, 25,000 commits, and 400,000 LOC

https://github.com/symfony/symfony

Bridge/
5 sub-projects

Bundle/
5 sub-projects

Component/
33 independent sub-projects like Asset, Cache, CssSelector, Finder, Form, HttpKernel, Ldap, Routing, Security, Serializer, Templating, Translation, Yaml, ...
Common build system

Bazel from Google

Buck from Facebook

Pants from Twitter

Pants: A fast, scalable build system

Pants is a build system designed for codebases that:
* Are large and/or growing rapidly.
* Consist of many subprojects that share a significant amount of code.
* Have complex dependencies on third-party libraries.
* Use a variety of languages, code generators and frameworks.

Pants supports Java, Scala, Python, C/C++, Go, JavaScript/Node, Thrift, Protobuf and Android code. Adding support for other languages, frameworks and code generators is straightforward.

Pants is a collaborative open-source project, built and used by Twitter, Foursquare, Square, Medium and other companies.

Getting Started
* Installing Pants
* Setting Up Pants
* Tutorial

Cookbook
The Common Tasks documentation is a practical, solutions-oriented guide to some of the Pants tasks that you’re most likely to carry out on a daily basis.
Some advantages of monorepos
High Discoverability For Developers

- Developers can read and explore the whole codebase
- `grep`, IDEs and other tools can search the whole codebase
- IDEs can offer auto-completion for the whole codebase
- Code Browsers can links between all artifacts in the codebase
Code-Reuse is cheap

Almost zero cost in introducing a new library

- Extract library code into a new directory/component
- Use library in other components
- Profit!
Refactorings in one commit

Allow large scale refactorings with one single, atomic, history-preserving commit

- Extract Library/Component
- Rename Functions/Methods/Components
- Housekeeping (phpcs-fixer, Namespacing, ...)

Qafoo
passion for software quality

Copyright Qafoo GmbH, All Rights Reserved

talks.qafoo.com
Another refactoring example

- Make large backward incompatible changes easily... especially if they span different parts of the project

- For example, old APIs can be removed with confidence
  - Change an API endpoint code \textbf{and} all its usages in \textbf{all} projects in \textbf{one} pull request
Some more advantages

- Easy continuous integration and code review for changes spanning several projects
- (Internal) dependency management is a non-issue
- Less context switching for developers
- Code more reusable in other contexts
- Access control is easy
Some downsides

• Require collective responsibility for team and developers
• Require trunk-based development
  – Feature toggles are technical debt (recall financial services example)
• Force you to have only one version of everything
• Scalability requirements for the repository
• Can be hard to deal with updates around things like security issues
• Build and test bloat without very smart build system
• Slow VCS without very smart system
• Permissions?
Summary

• Configuration management
  – Treat infrastructure as code
  – Git is powerful

• Release management: versioning, branching, ...

• Software development at scale requires a lot of infrastructure
  – Version control, build managers, testing, continuous integration, deployment, ...

• It’s hard to scale development
  – Move towards heavy automation (DevOps)

• Continuous deployment increasingly common

• Opportunities from quick release, testing in production, quick rollback