Advances in Continuous Integration Testing @Google

By: John Micco  - jmicco@google.com
Testing Scale at Google

- 4.2 million individual tests running continuously
  - Testing runs before and after code submission
- 150 million test executions / day (averaging 35 runs / test / day)
- Distributed using internal version of bazel.io to a large compute farm
- Almost all testing is automated - no time for Quality Assurance
- 13,000+ individual project teams - all submitting to one branch
- Drives continuous delivery for Google
- 99% of all test executions pass
Testing Culture @ Google

- ~11 Years of testing culture promoting hand-curated automated testing
  - [Testing on the toilet](https://testingonthetoilet.com) and Google testing [blog](https://blog.google/) started in 2007
  - [GTAC](https://gtac.google.com) conference since 2006 to share best practices across the industry
  - First internal awards for unit testing were in 2003!
  - Part of our new hire orientation program

- **SETI** role
  - Usually 1-2 SETI engineers / 8-10 person team
  - Develop test infrastructure to enable testing

- Engineers are expected to write automated tests for their submissions

- Limited experimentation with model-based / automated testing
  - Fuzzing, UI walkthroughs, Mutation testing, etc.
  - Not a large fraction of overall testing
Regression Test Selection (RTS)
Regression Test Selection (RTS)
Current Regression Test Selection (RTS)
Postsubmit testing

- Continuously runs 4.5M tests as changes are submitted
  - A test is affected iff a file being changed is present in the transitive closure of the test dependencies. (Regression Test Selection)
  - Each test runs in 1.5 distinct flag combinations (on average)
  - Build and run tests concurrently on distributed backend.
  - Runs as often as capacity allows

- Records the pass / fail result for each test in a database
  - Each run is uniquely identified by the test + flags + change
  - We have 2 years of results for all tests
  - And accurate information about what was changed

See: prior deck about Google CI System, See this paper about piper and CLs
Milestone Scheduling

Cut milestone at this CL
Milestone Scheduling
Milestone Scheduling
Milestone Scheduling
Milestone Scheduling

Affected Test Target set

Change Lists
Reducing Costs

- RTS based on declared dependencies is problematic!
  - A small number of core changes impact everything
  - Milestone Scheduling ends up running all tests
  - Distant dependencies don't often find transitions
  - 99.8% of all test executions do not transition
    - A perfect algorithm would only schedule the 0.2% of tests that do transition
  - There must be something in between 99.8% and 0.2% that will find most faults
RTS Affected Target Counts Frequency

- Stats:
  - Median 38 tests!
  - 90th percentile 2,604
  - 95th percentile 4,702
  - 99th percentile 55,730

- A tiny number of CLs is causing over-scheduling
- It only takes 1 CL on the long tail to force a milestone to run all tests
NOTE: Presubmit testing makes post-submit failures relatively rare - but we still spend 50% of testing resources on post-submit testing.
Project Status and Groupings

- Tests are grouped into "projects" that include all relevant tests needed to release a service
- This allows teams to release when unrelated tests are failing
- Current system is conservative
  - Gives a green signal iff all affected tests pass
  - 100% confidence that a failing test was not missed
- We require a definitive result for all affected tests (selected by RTS)
  - Projects only receive a status on milestones
  - We say that projects are "inconclusive" between milestones - when they get affected
  - Since milestones are far apart projects are frequently inconclusive
Project Status and Groupings

Google
Greenish Service

- Reducing over-scheduling means < 100% confidence
  - Not all tests will be run!
  - Milestones will be far apart
- Need a signal for release
- Introduce "Greenish" service
  - Predicts likelihood that skipped tests will pass
  - Provides a probability rather than certainty of green
New Scheduling Algorithms

- Skip milestones and schedule tests with highest likelihood to find transitions
- Occasional milestones will find transitions missed by opportunistic scheduling
- Goal: Find all transitions using vastly reduced resources
- Decrease time to find transitions

Google
Transitions?

Definition: A non-flaky change in state of a test from Pass -> Fail or Fail -> Pass. The goal of CI is to find transitions quickly - it is important to know when tests are broken or fixed by code submissions

- No transition
- No transition
- No transition
- No transition
- Transition Pass -> Fail (showing culprit finding)
- Transition Fail -> Pass (No culprit finding)
- Transition Pass -> Fail (showing culprit finding)
- Transition Fail -> Pass (No culprit finding)

Note: It is also important to eliminate / ignore flaky tests and to have good information about flaky tests.
Skipping milestones: <1% test targets detect breakages
Skipping milestones: breakages imply culprit finding
Skipping milestones: culprits detected and found
Skipping milestones: culprits detected and found
Skipping milestones: culprits detected and found

Culprit detected & found
Skipping milestones: culprits detected and found

- Culprit detected & found

- Affected Test Target set

- Change Lists
Skipping milestones
Skipping milestones

Affected Test Target set

Google

Change Lists
Skipping milestones: cuprit finding, acceptance tuning
Skipping milestones: cuprit finding, acceptance tuning
Evaluating Strategies

- **Goals**
  - Low testing cost
  - Low time to find a transition
  - Low risk of missing transitions

- **Measure "Safety"**
  - Skipping a test is "safe" if it did not transition
  - 100% safety means all transitions are found

- **Evaluate new strategies against historical record**
  - Allows Fast algorithm iteration time
  - Must excludes flaky test failures
Offline Safety Evaluation

- 96% of changes do not cause a transition - we could skip all testing for them!
- Of the remainder, a perfect algorithm could skip more than 98% of the currently selected tests and find all transitions.
Analysis of Test Results at Google

- Analysis of a large sample of tests (1 month) showed:
  - 84% of transitions from Pass -> Fail are from "flaky" tests
  - Only 1.23% of tests ever found a breakage
  - Frequently changed files more likely to cause a breakage
  - 3 or more developers changing a file is more likely to cause a breakage
  - Changes "closer" in the dependency graph more likely to cause a breakage
  - Certain people / automation more likely to cause breakages (oops!)
  - Certain languages more likely to cause breakages (sorry)

- See our accepted [Paper](#) at ICSE 2017
Flaky Tests

- Test Flakiness is a huge problem
- Flakiness is a test that is observed to both Pass and Fail with the same code
- Almost 16% of our 4.2M tests have some level of flakiness
- Flaky failures frequently block and delay releases
- Developers ignore flaky tests when submitting - sometimes incorrectly
- We spend between 2 and 16% of our compute resources re-running flaky tests
Flaky test impact on project health

- Many tests need to be aggregated to qualify a project
- Probability of flake aggregates as well
- Flakes
  - Consume developer time investigating
  - Delay project releases
  - Waste compute resources re-running to confirm
Percentage of resources spent re-running flakes
Sources of Flakiness

- Factors that cause flakes
  - Test case factors
    - Waits for resource
    - sleep()
    - Webdriver test
    - UI test
  - Code being tested
    - Multi-threaded
  - Execution environment/flags
    - Chrome
    - Android
  - ...
Flakes are Inevitable

- Continual rate of 1.5% of test executions reporting a "flaky" result
- Despite large effort to identify and remove flakiness
  - Targeted "fixits"
  - Continual pressure on flakes
- Observed insertion rate is about the same as fix rate

Conclusion: Testing systems must be able to deal with a certain level of flakiness. Preferably minimizing the cost to developers.

Google
Flaky Test Infrastructure

- We re-run test failure transitions (10x) to verify flakiness
  - If we observe a pass the test was flaky
  - Keep a database and web UI for "known" flaky tests
Finding Flakes using the historical record

- 84% of test transitions are due to flakiness
- Concentrated in 16% of the total test pool
- Conclusion: Tests with more transitions are flaky

![5 HOUR PERIOD](image)

TEST 1

TEST 2

5 HOUR PERIOD
Number of Edges Per Target by % Flakes/NotFlakes
Number of Transitions Per Target by % Flakes/NotFlakes

**Take away message**: Test targets with more transitions in their history are more likely to be flakes. (Number of edges = signal for flake detection)
Flakes Tutorial

- Using Google BigQuery against the public data set from our 2016 paper
- Reproduce some of our results
  - Techniques to identify flaky tests using queries
  - Hands on!
- Hope to see you there!
Q&A

For more information:

- Google Testing Blog on CI system
- Youtube Video of Previous Talk on CI at Google
- Flaky Tests and How We Mitigate Them
- Why Google Stores Billions of Lines of Code in a Single Repo
- GTAC 2016 Flaky Tests Presentation
- (ICSE 2017) "Who Broke the Build? Automatically Identifying Changes That Induce Test Failures In Continuous Integration at Google Scale" by Celal Ziftci and Jim Reardon

Google