2018年3月7日(水)−3月8日(木)　会場：日本大学理工学部 駿河台校舎１号館
JaSST'18 Tokyo
ソフトウェアテストシンポジウム 2018 東京
JaSST'18 Tokyo ： Japan Symposium on Software Testing in Tokyo 2018

Google

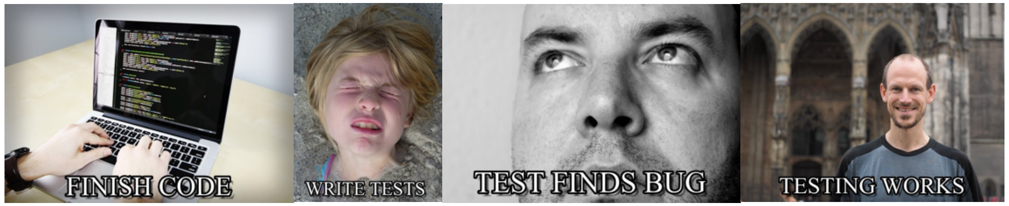
Advances in Continuous Integration Testing @Google

By: John Micco - jmicco@google.com

投稿者：ジョン・ミッコ

Testing Scale at Google

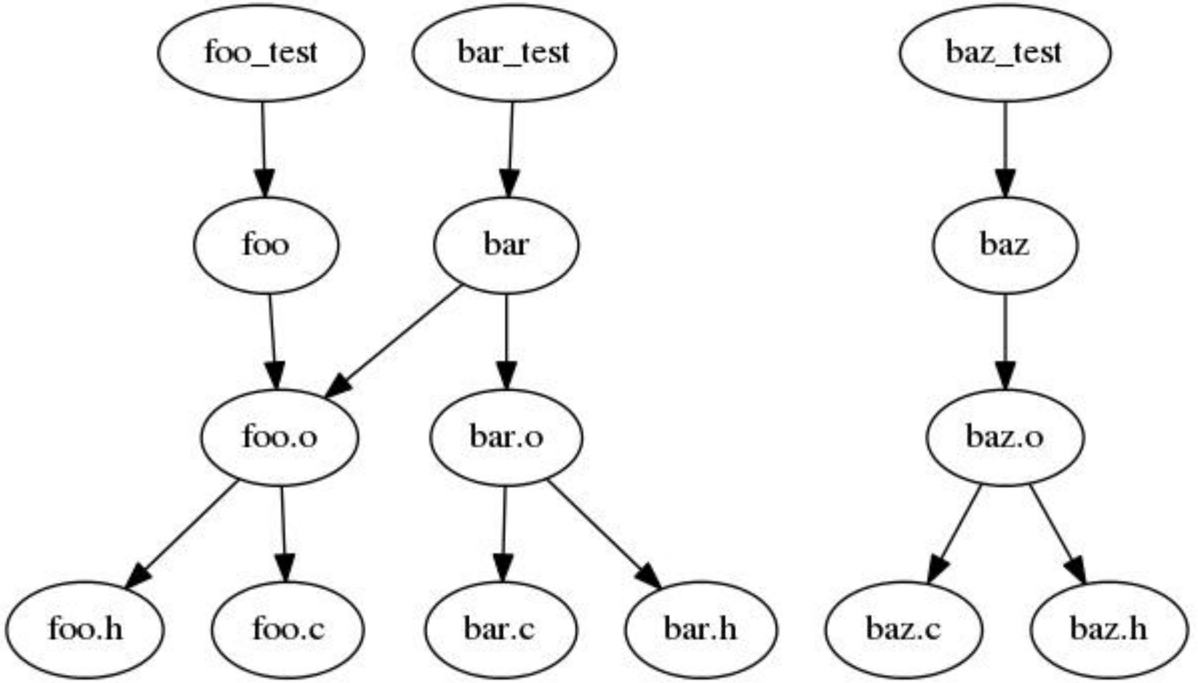
- 4.2 million individual tests running continuously
  - Testing runs before and after code submission
- 150 million test executions / day (averaging 35 runs / test / day)
- Distributed using internal version of bazel.io to a large compute farm
- Almost all testing is automated - no time for Quality Assurance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



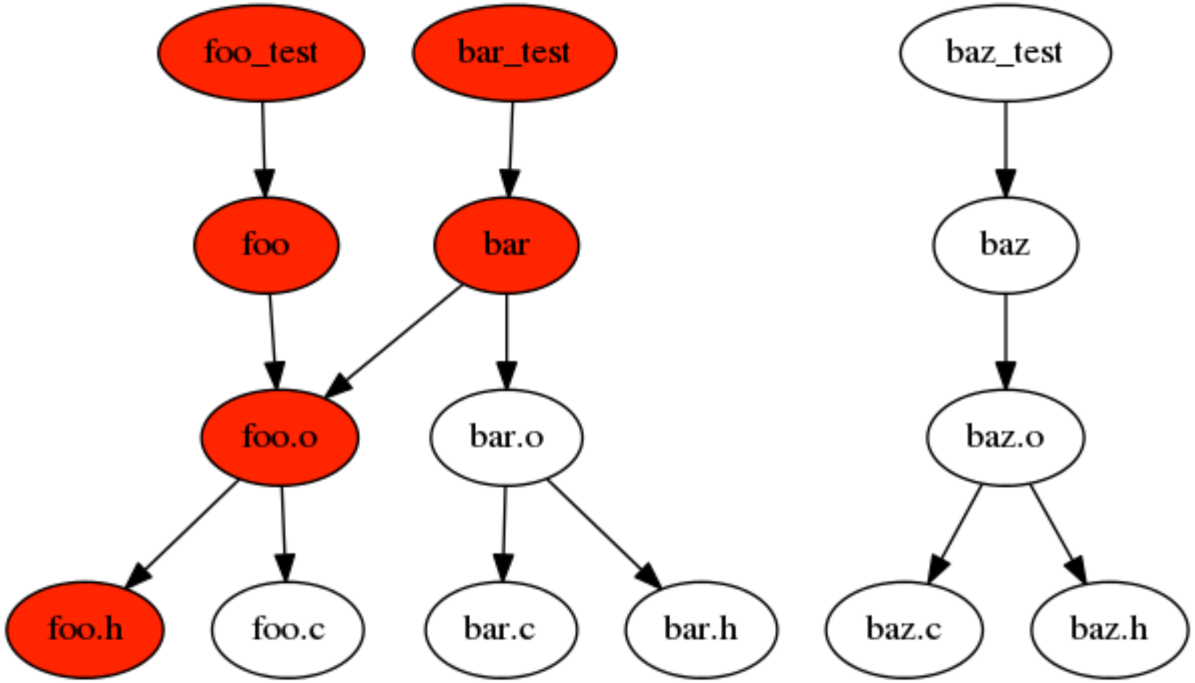
Testing Culture @ Google

- ~11 Years of testing culture promoting hand-curated automated testing
  - Testing on the toilet and Google testing blog started in 2007
  - GTAC 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 waltkthroughs, Mutation testing, etc.
  - Not a large fraction of overall testing

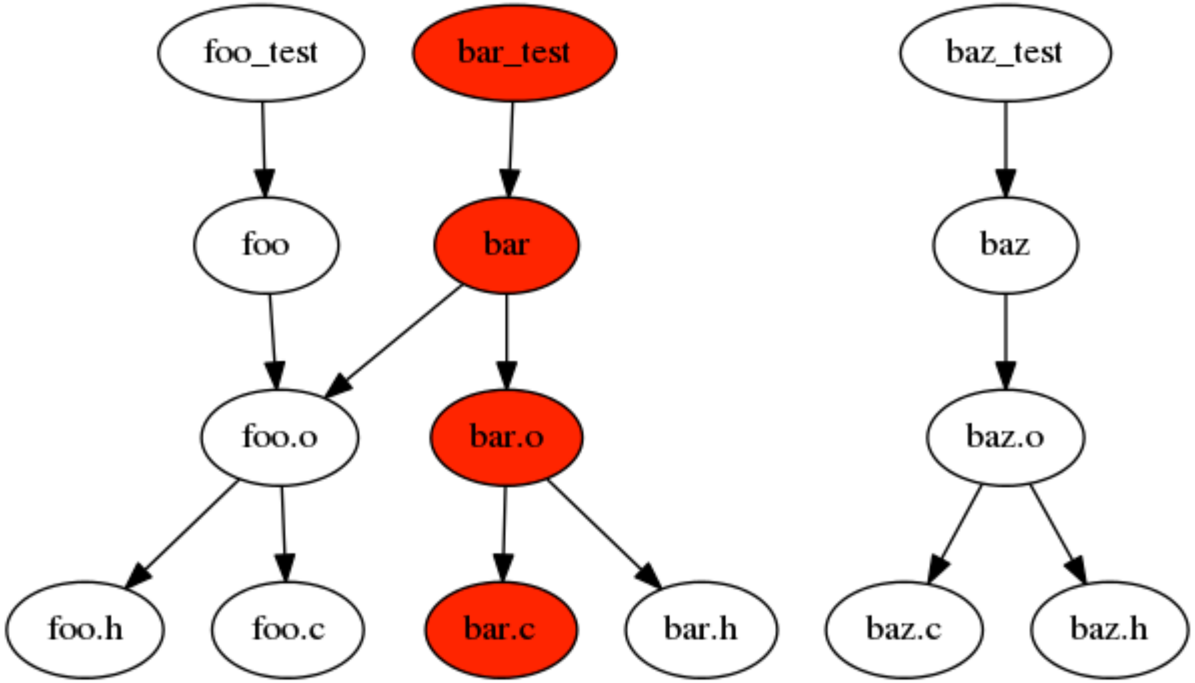
Regression Test Selection (RTS)



Regression Test Selection (RTS)



Current Regression Test Selection (RTS)

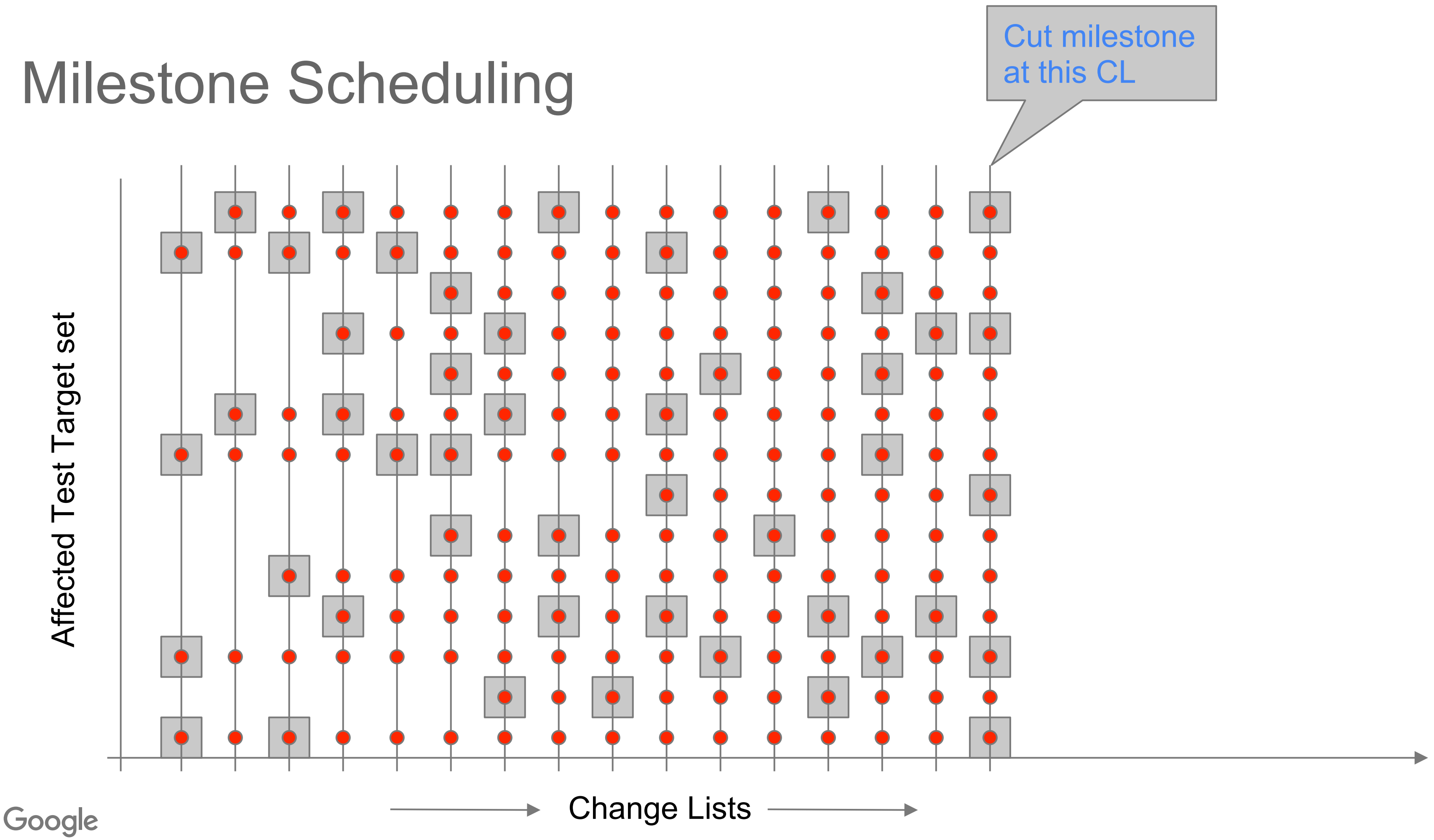


Postsubmit testing

- 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

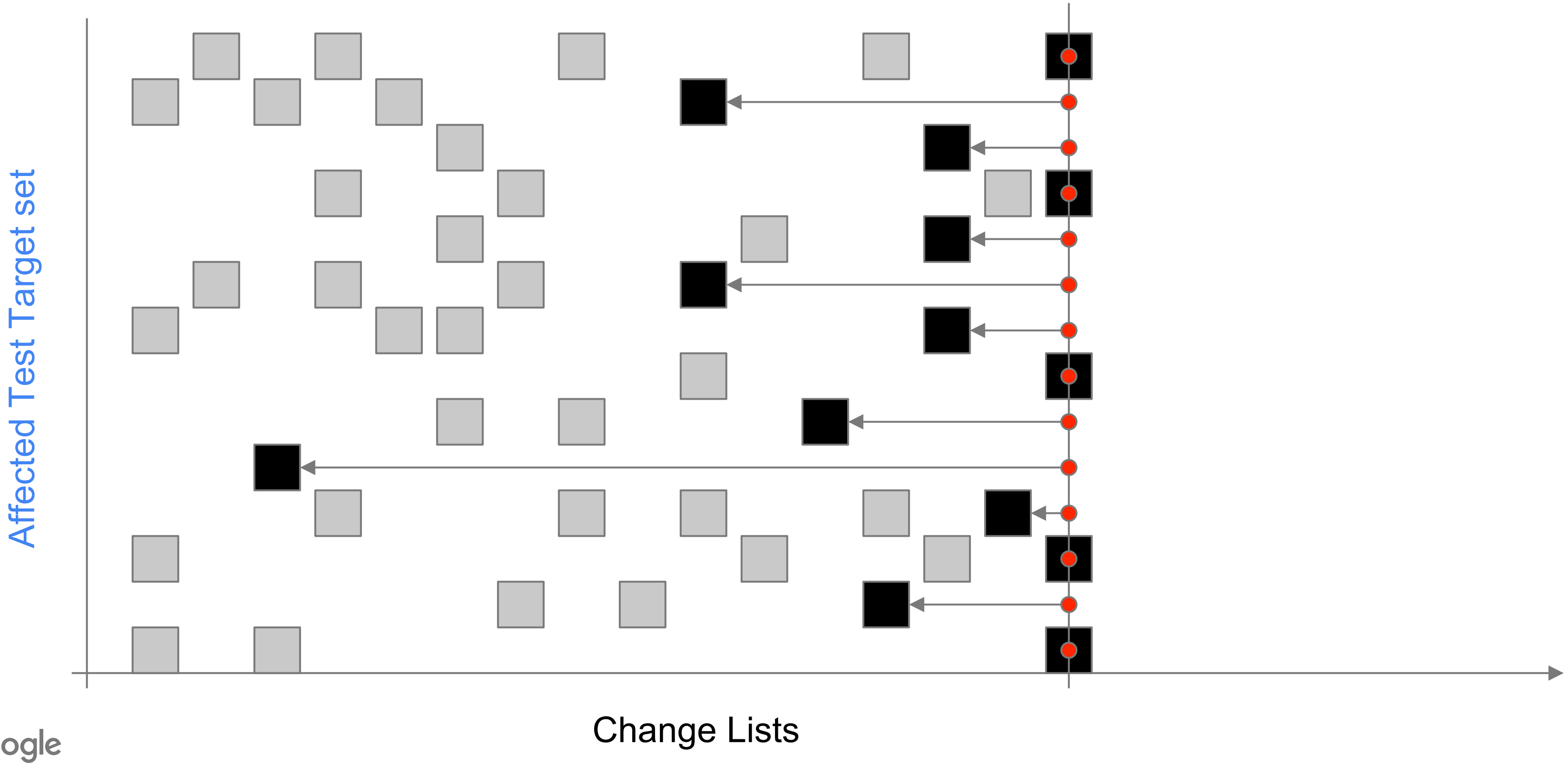
See: prior deck about Google CI System, See this paper about piper and CLs

Milestone Scheduling

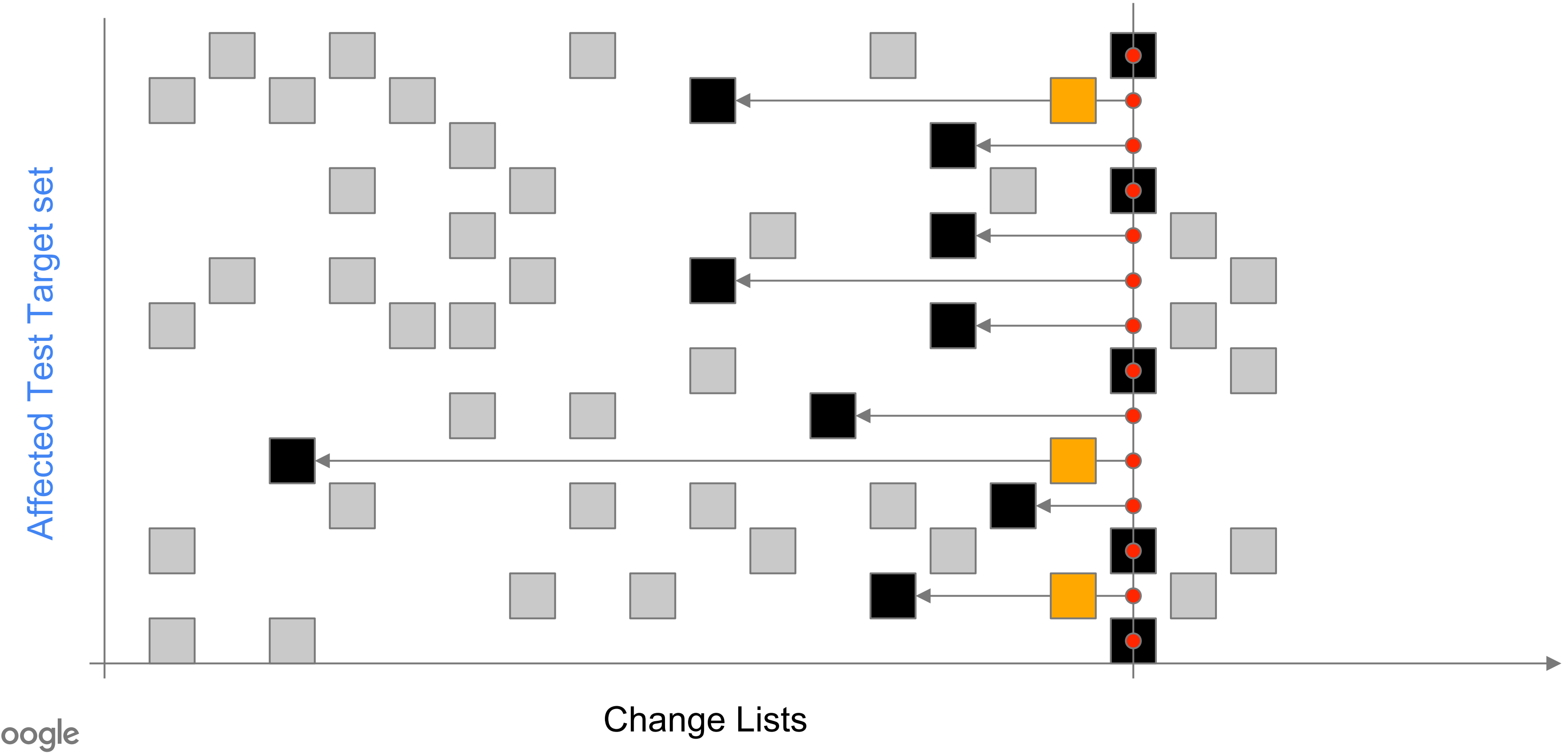


Google

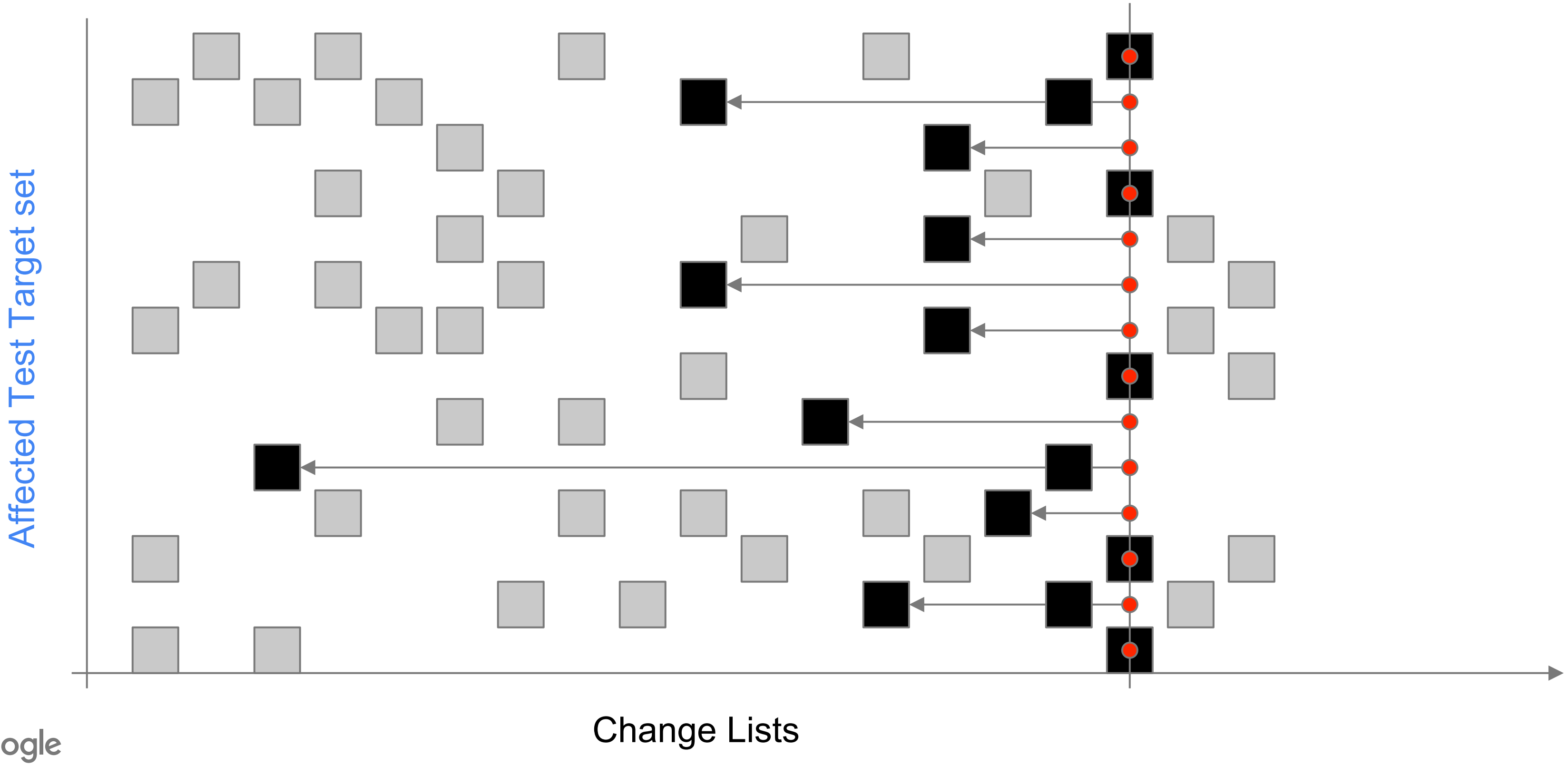
Milestone Scheduling



Milestone Scheduling

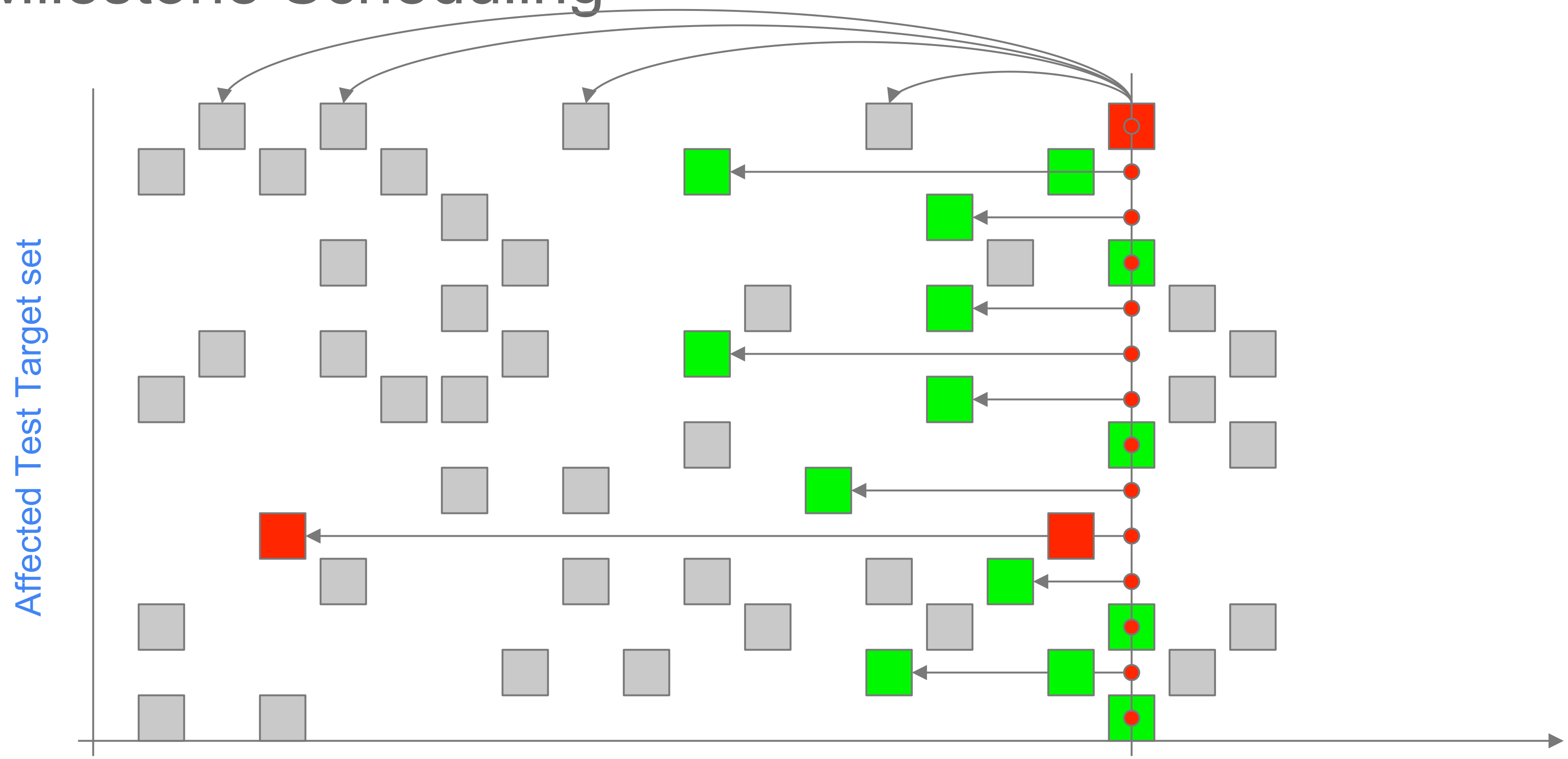


Milestone Scheduling



Google

Milestone Scheduling

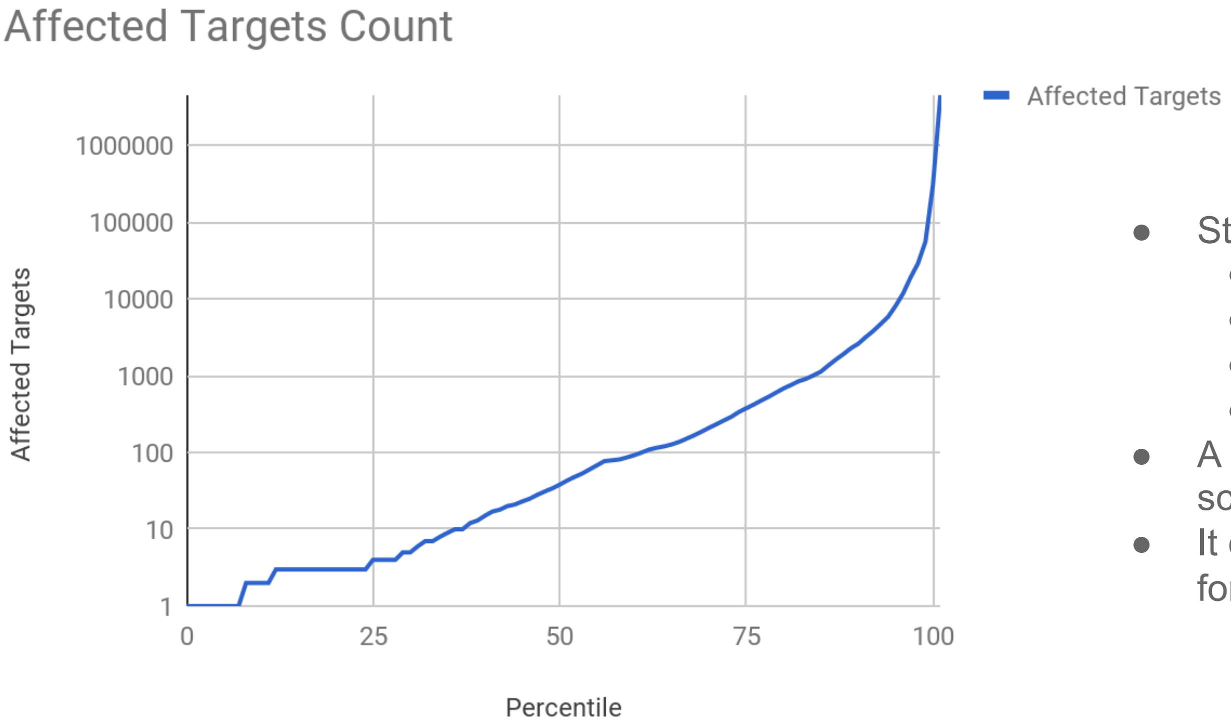


Google

Reducing Costs

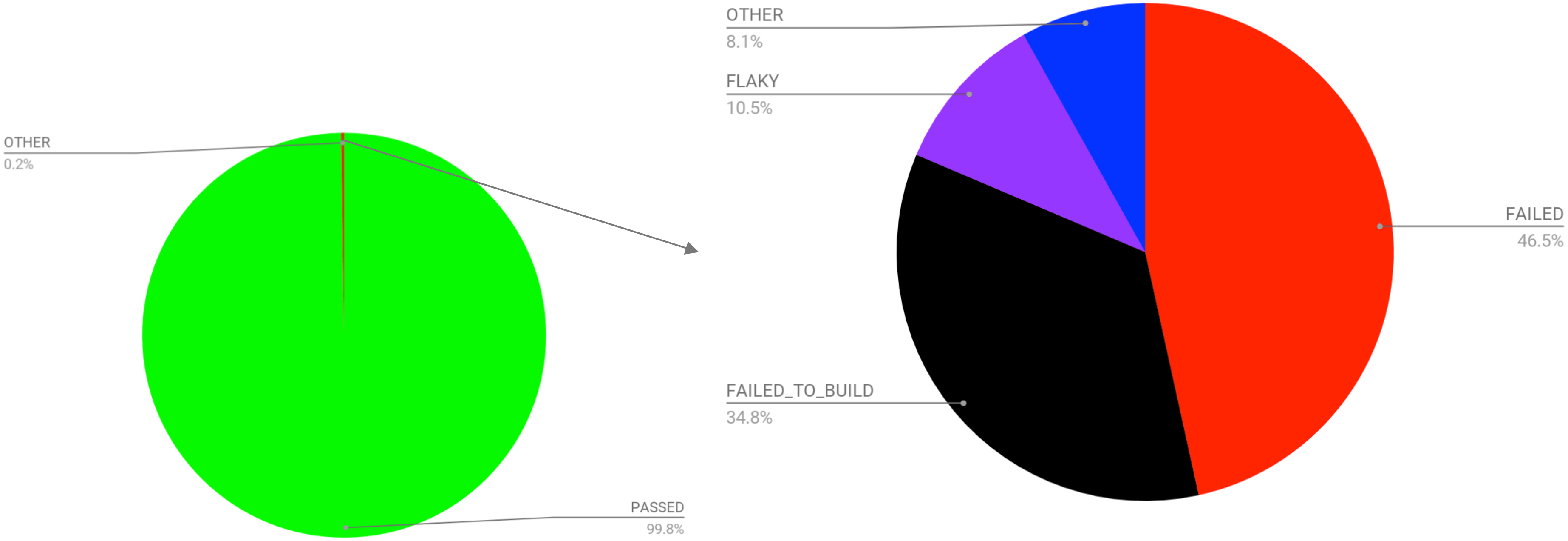
- RTS based on declared dependencies is problematic!
  - A small number of core changes impact everything
  - Milestone Scheduling ends up running all tests
  - Distant dependencies don't often find transitions
  - 99.8% of all test executions do not transition
    - A perfect algorithm would only schedule 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 perentile 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

Test Results



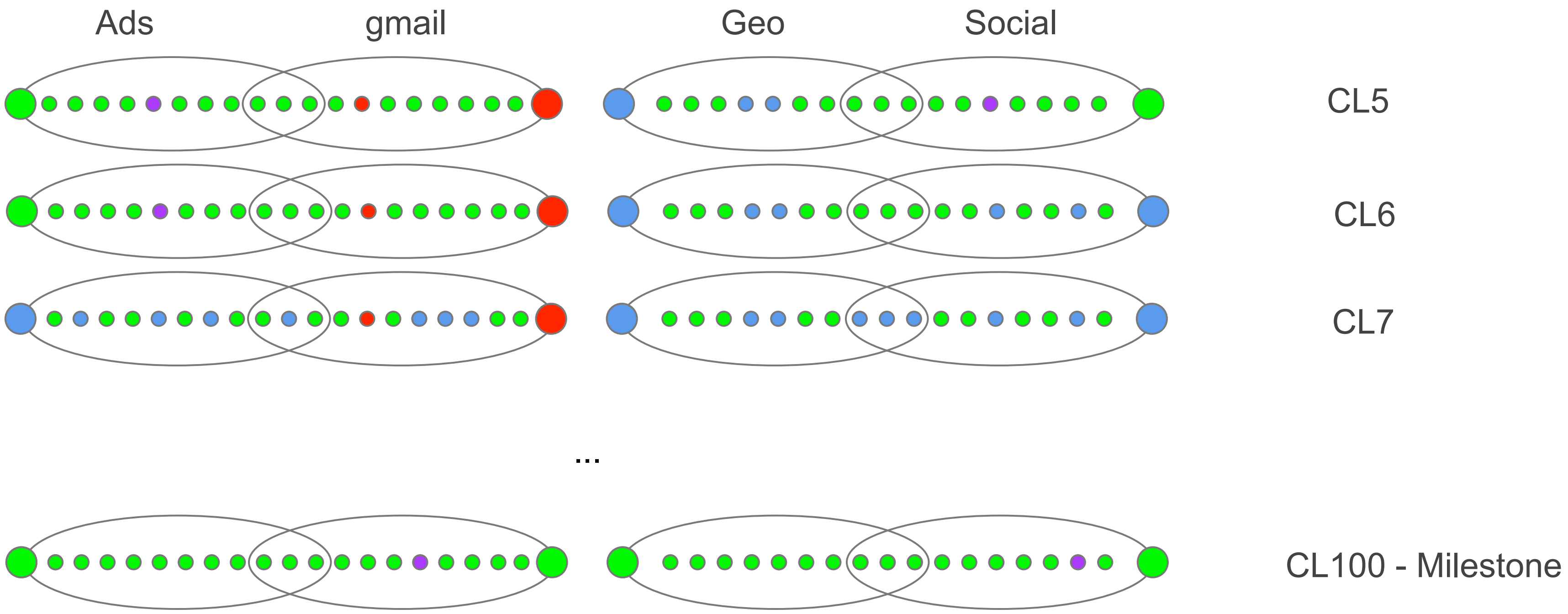
NOTE: Presubmit testing makes post-submit failures relatively rare - but we still spend 50% of testing resources on post-submit testing.

Google

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

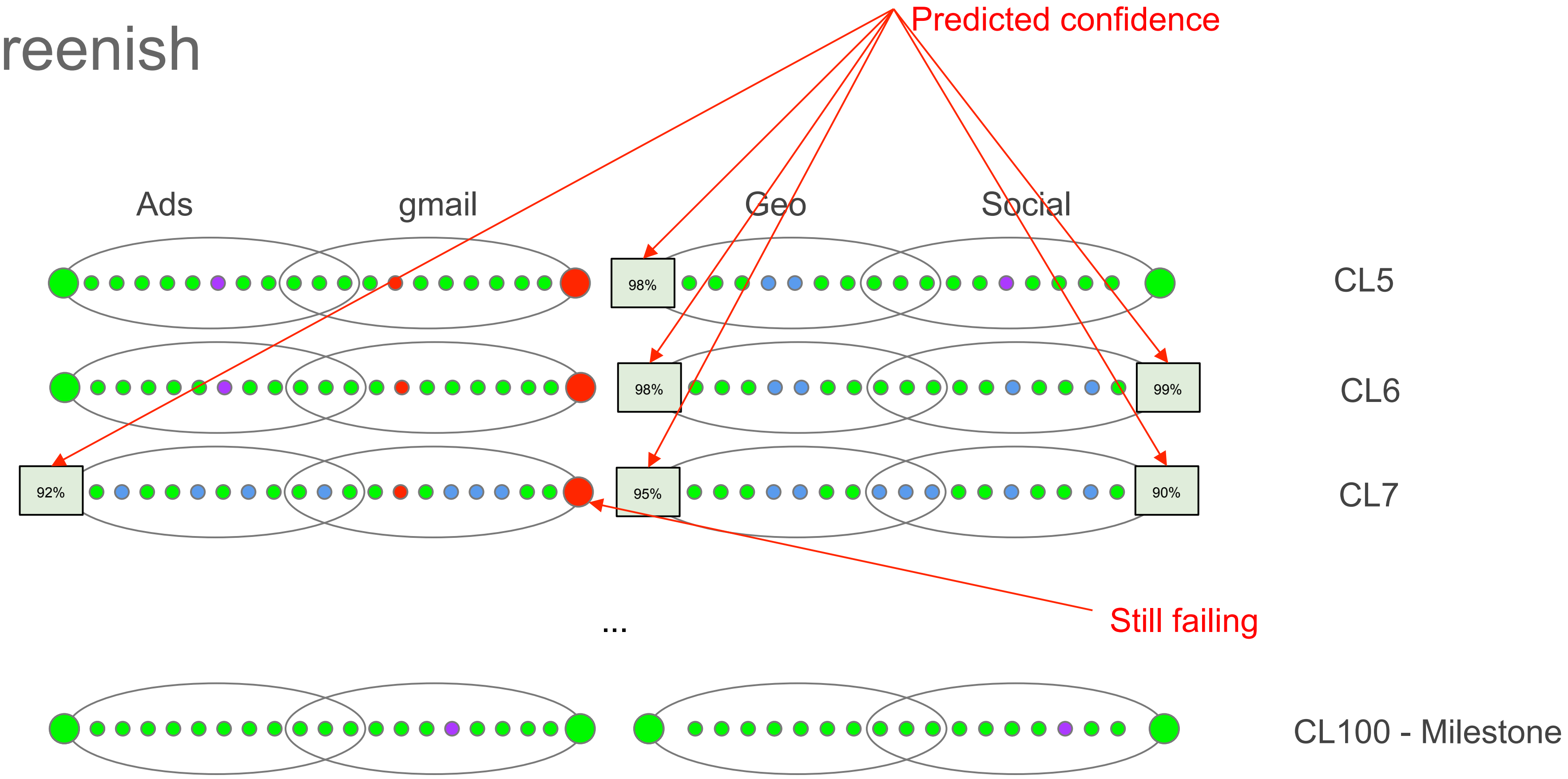
Project Status and Groupings



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

Greenish



Google

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

Safe Results *skipping this target would not miss a transition*

Time ⟶

Changelist	CL1	CL2
Target Result	P	P
Safety	**-**	**Safe**
Transition	-	P->P

** = affected*

Safe Results *skipping this target would not miss a transition*

Time →

Changelist	CL1	CL2
Target Result	F	F
Safety	**-**	**Safe**
Transition	-	F->F

** = affected*

Safe Results *skipping this target would not miss a transition*

Time →

Changelist	CL1	CL2	CL3
Target Result	P	*	P
Safety	**-**	**Safe**	**Safe**
Transition	-	P->P	P->P

** = affected*

Safe Results *skipping this target would not miss a transition*

Time →

Changelist	CL1	CL2	CL3
Target Result	F	*	F
Safety	**-**	**Safe**	**Safe**
Transition	-	F->F	F->F

** = affected*

Unsafe Results *skipping this target would definitely miss a transition*

Time →

Changelist	CL1	CL2
Target Result	P	F
Safety	**-**	**Unsafe**
Transition	-	P->F

** = affected*

Unsafe Results *skipping this target would definitely miss a transition*

Time →

Changelist	CL1	CL2
Target Result	F	P
Safety	**-**	**Unsafe**
Transition	-	F->P

** = affected*

Maybe Unsafe Results *skipping this target might miss a transition*

Time →

Changelist	CL1	CL2	CL3
Target Result	P	*	F
Safety	**-**	**Maybe unsafe**	**Maybe unsafe**
Transition	-	P->F	P->F

** = affected*

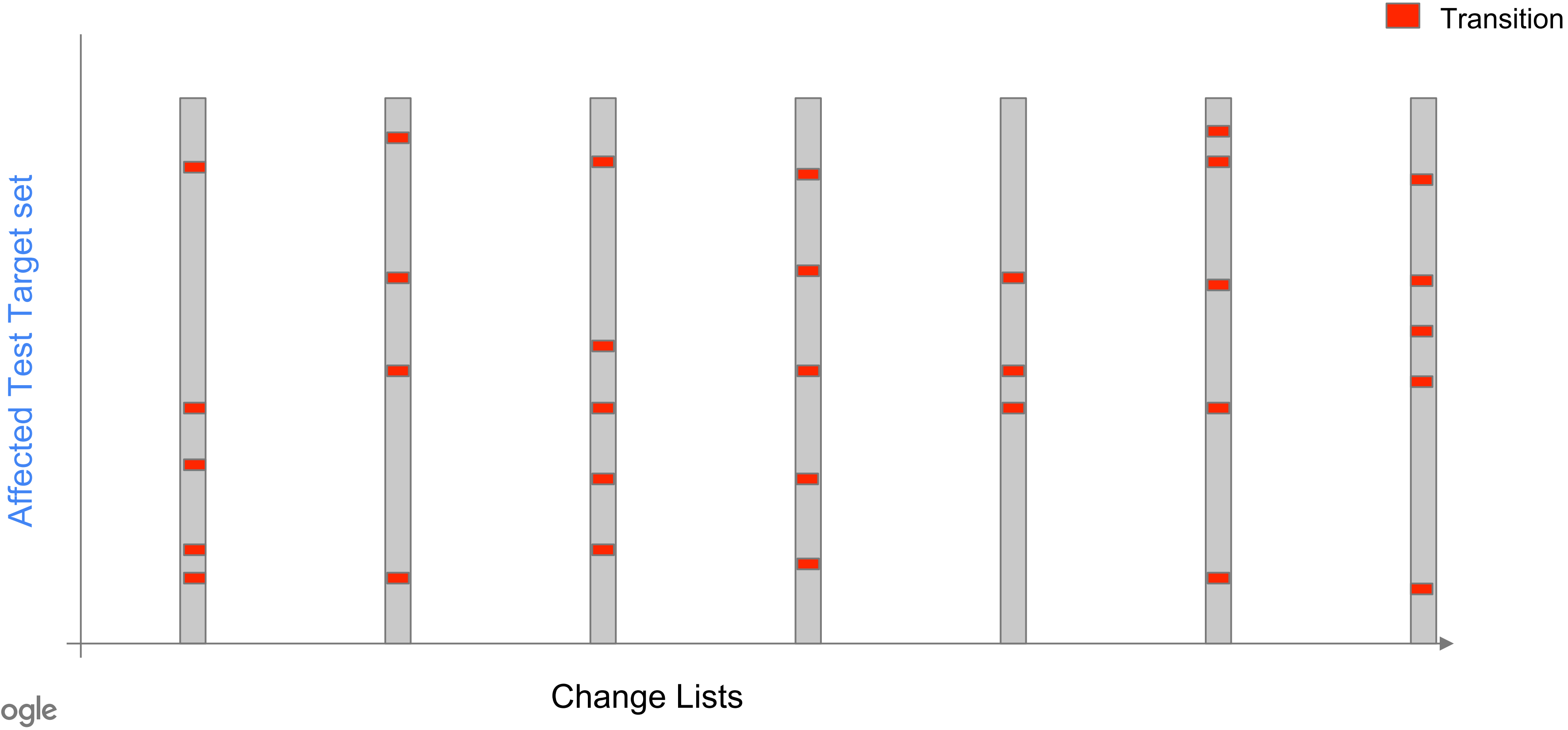
Maybe Unsafe Results *skipping this target might miss a transition*

Time →

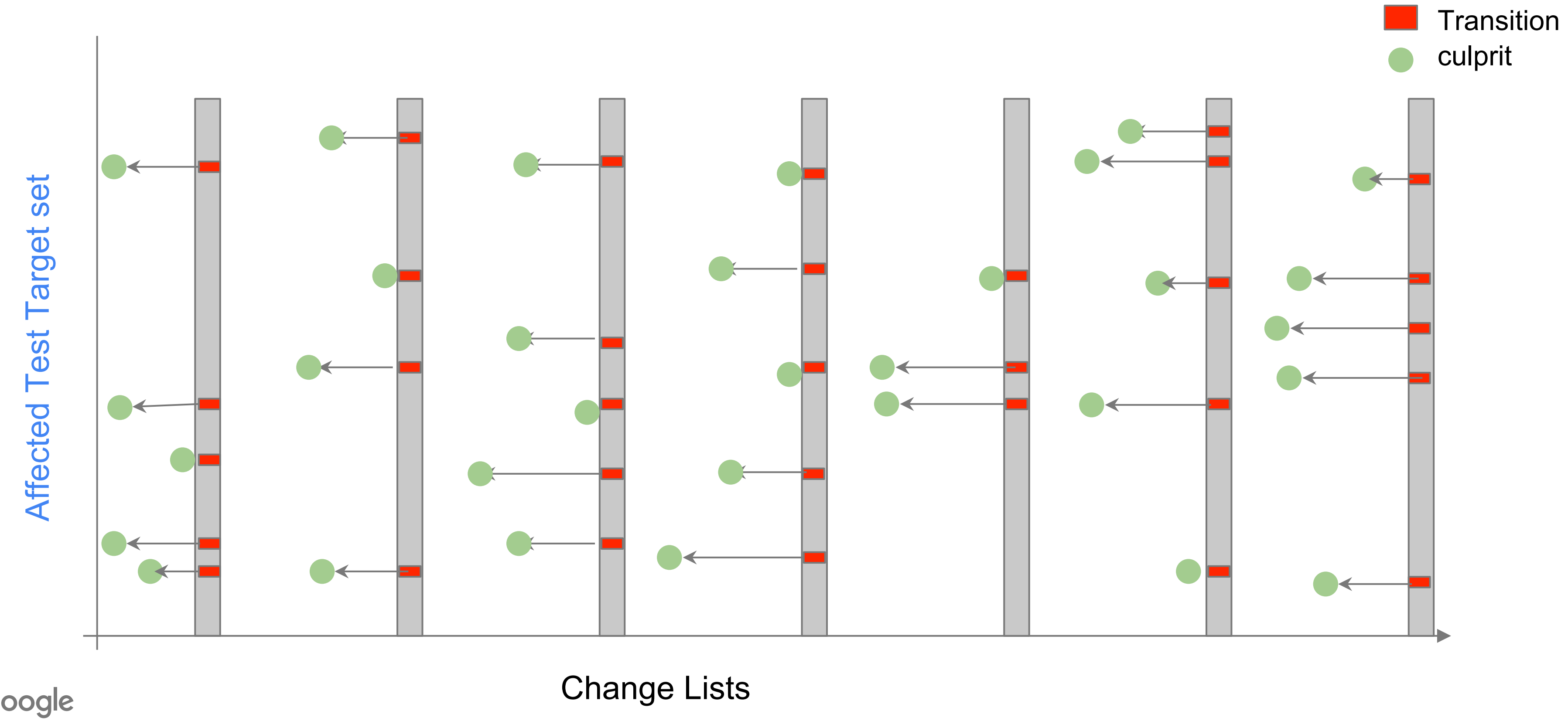
Changelist	CL1	CL2	CL3
Target Result	F	*	P
Safety	**-**	**Maybe unsafe**	**Maybe unsafe**
Transition	-	F->P	F->P

** = affected*

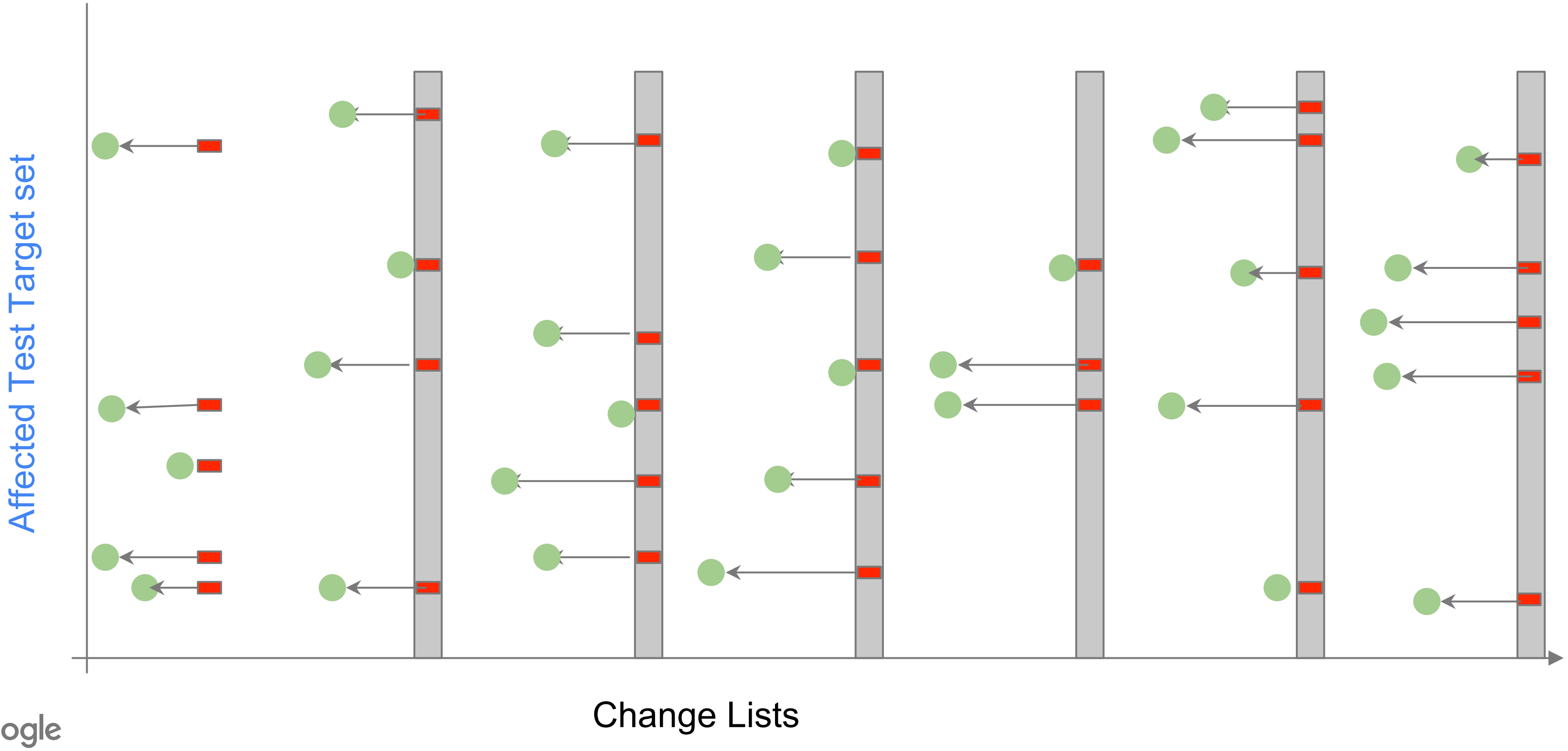
Skipping milestones: <1% test targets detect breakages



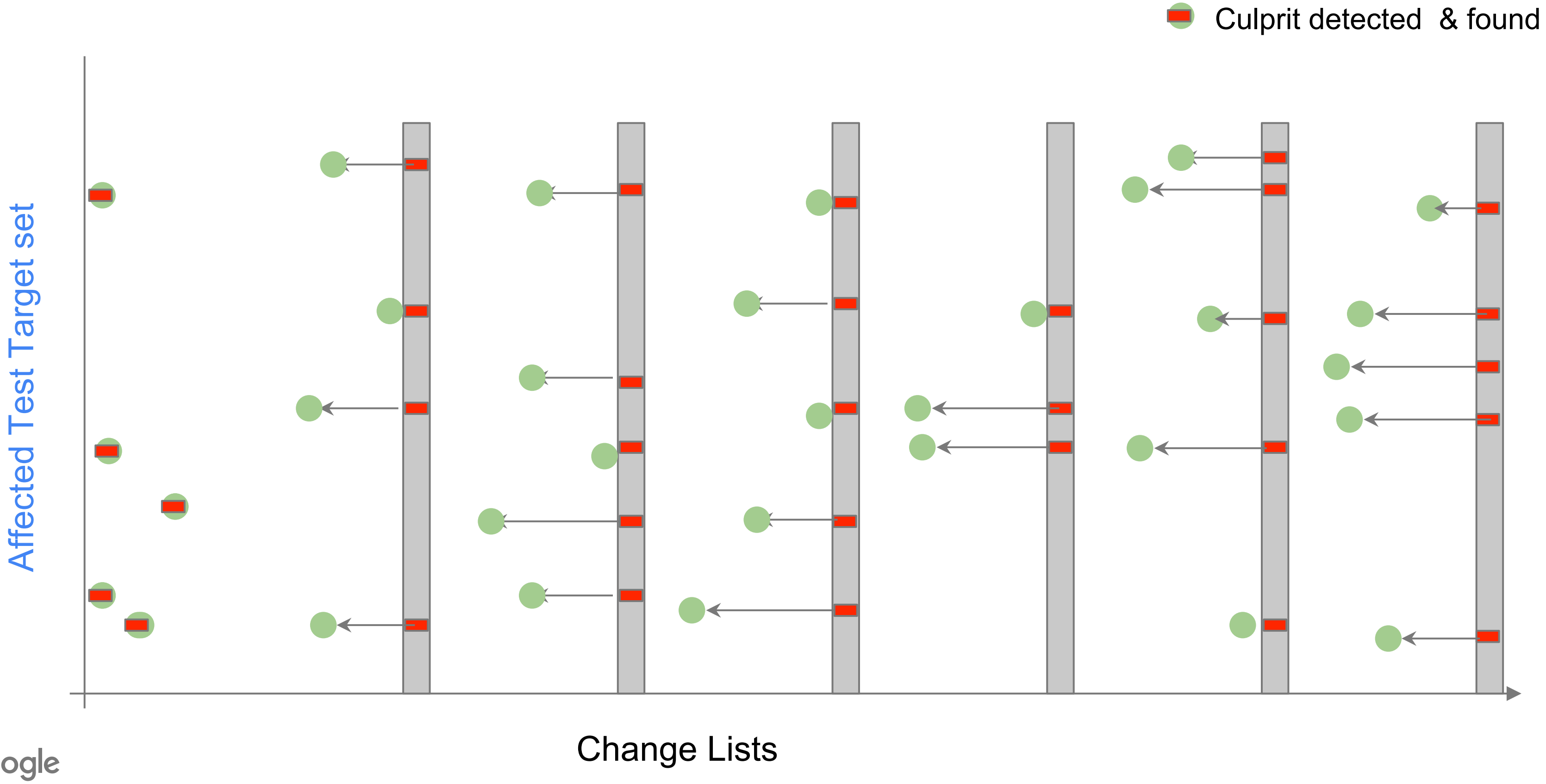
Skipping milestones: breakages imply cuprit finding



Skipping milestones: culprits detected and found

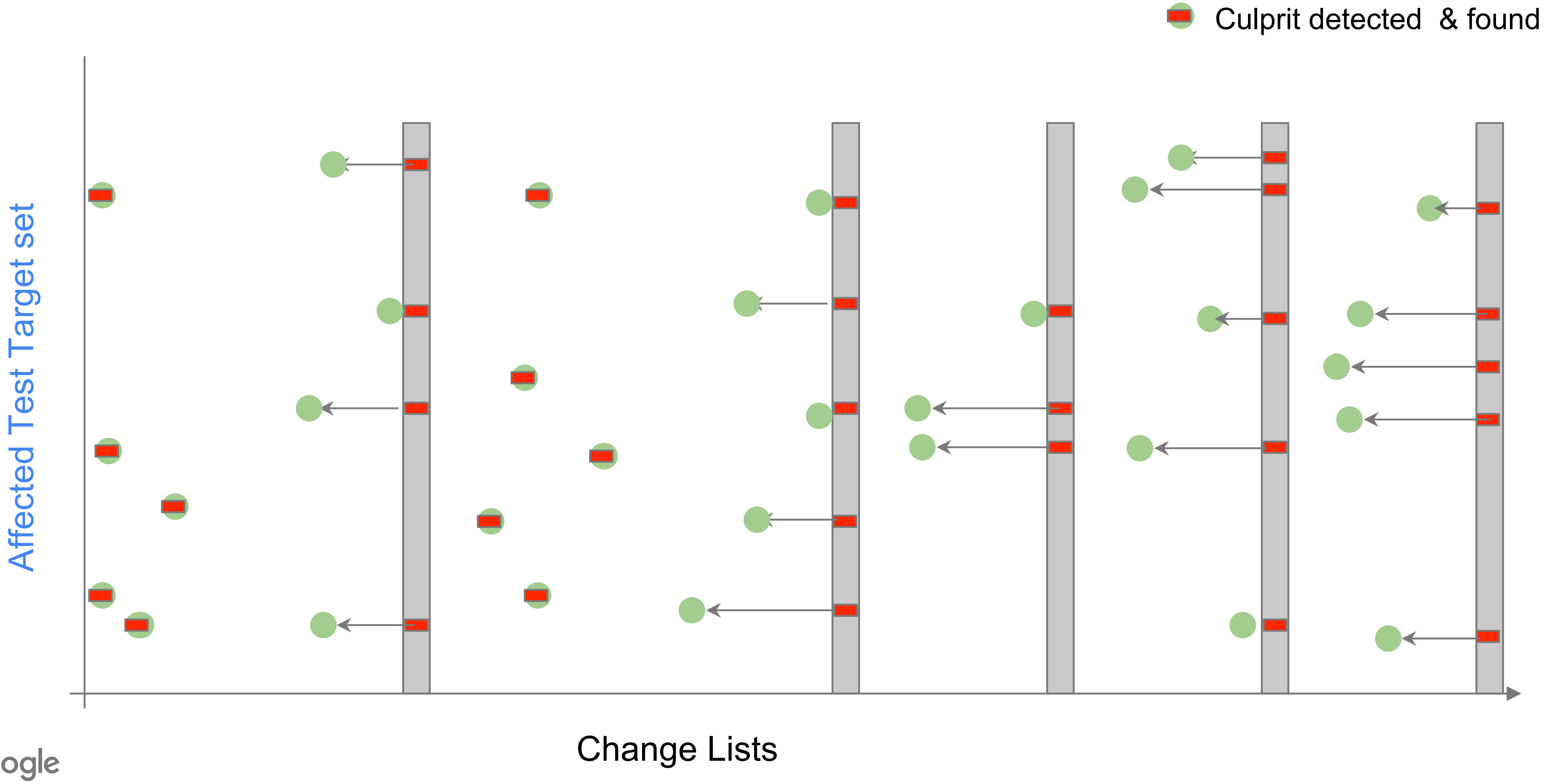


Skipping milestones: culprits detected and found

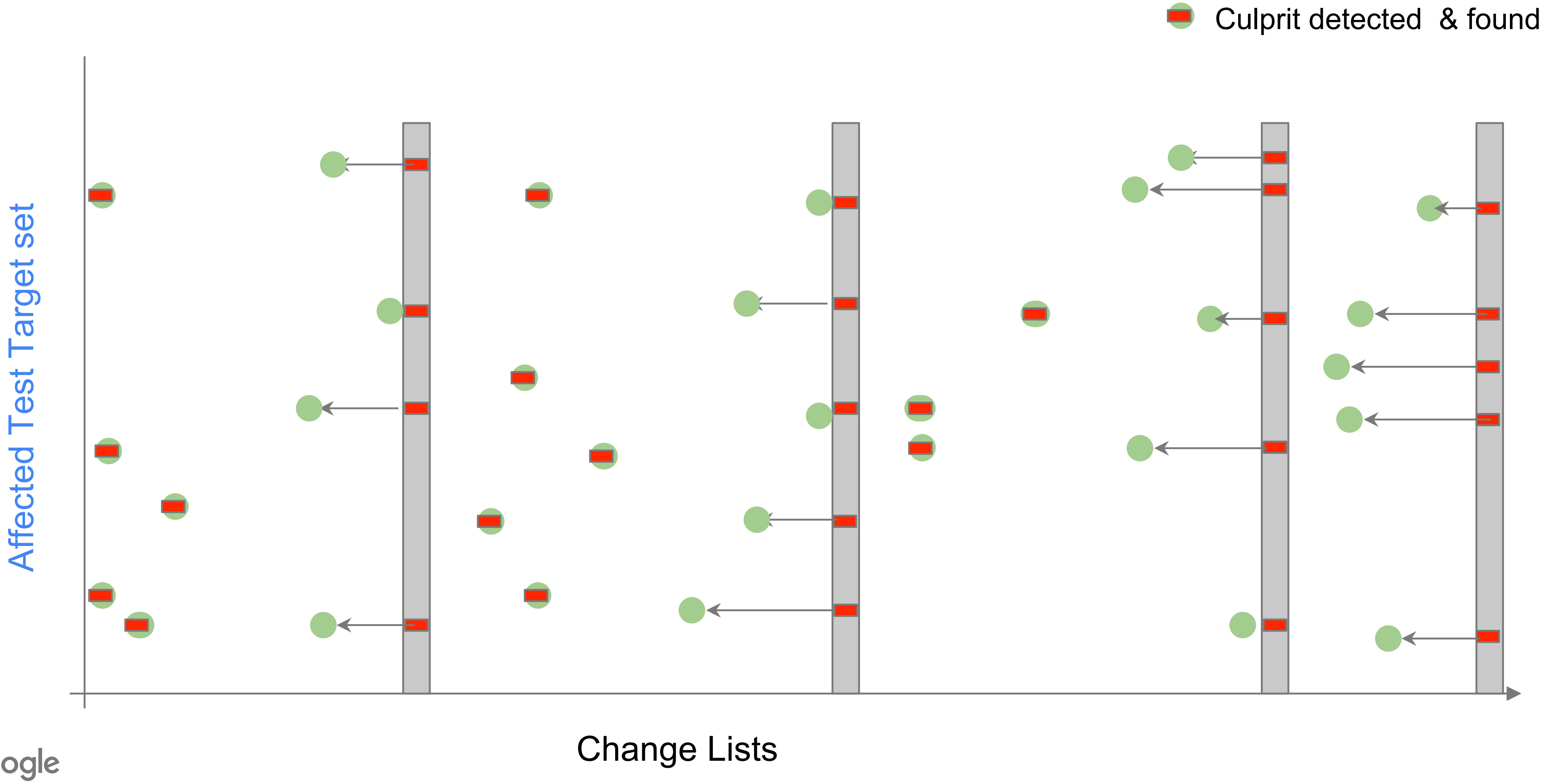


Google

Skipping milestones: culprits detected and found

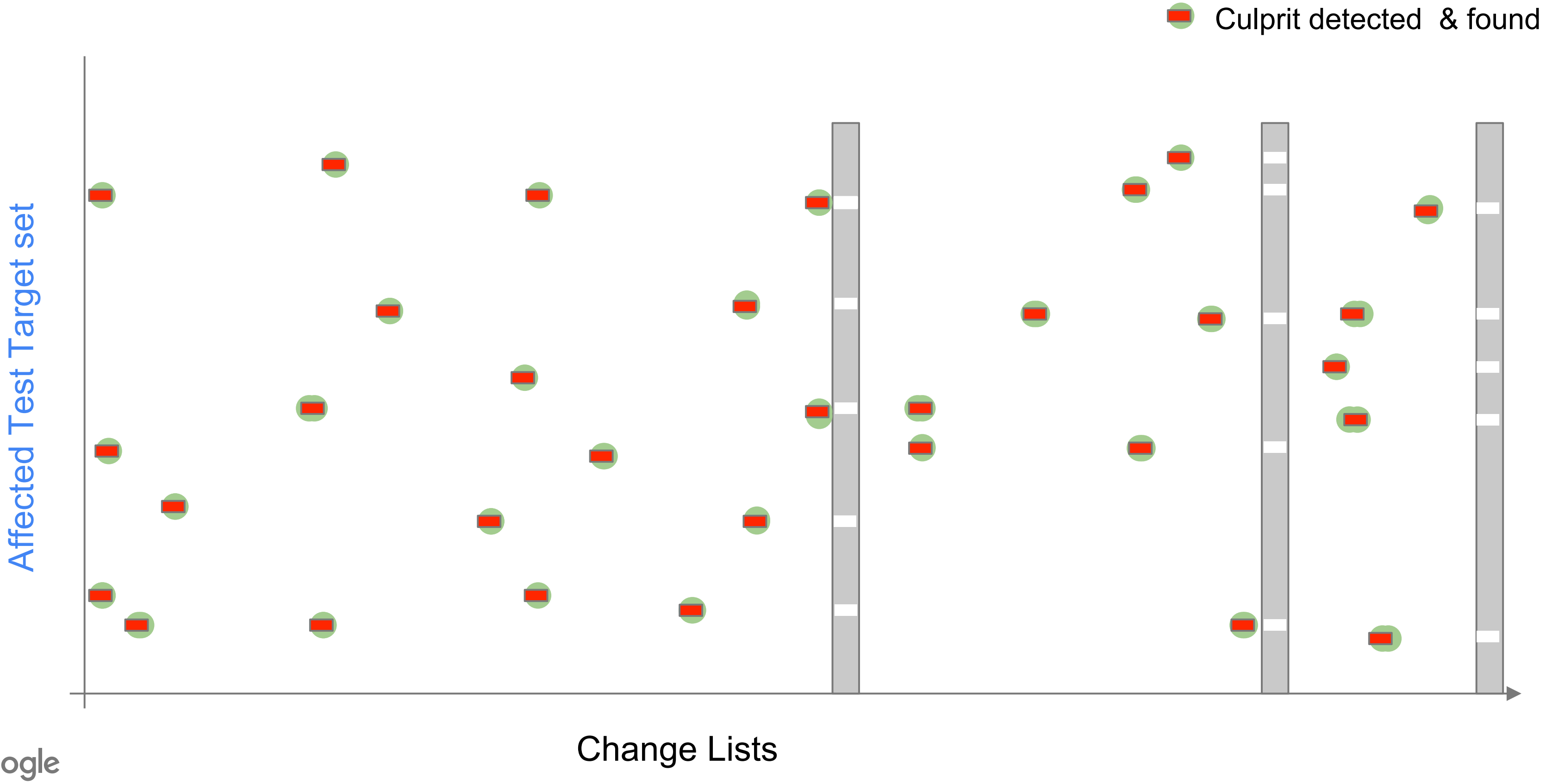


Skipping milestones: culprits detected and found



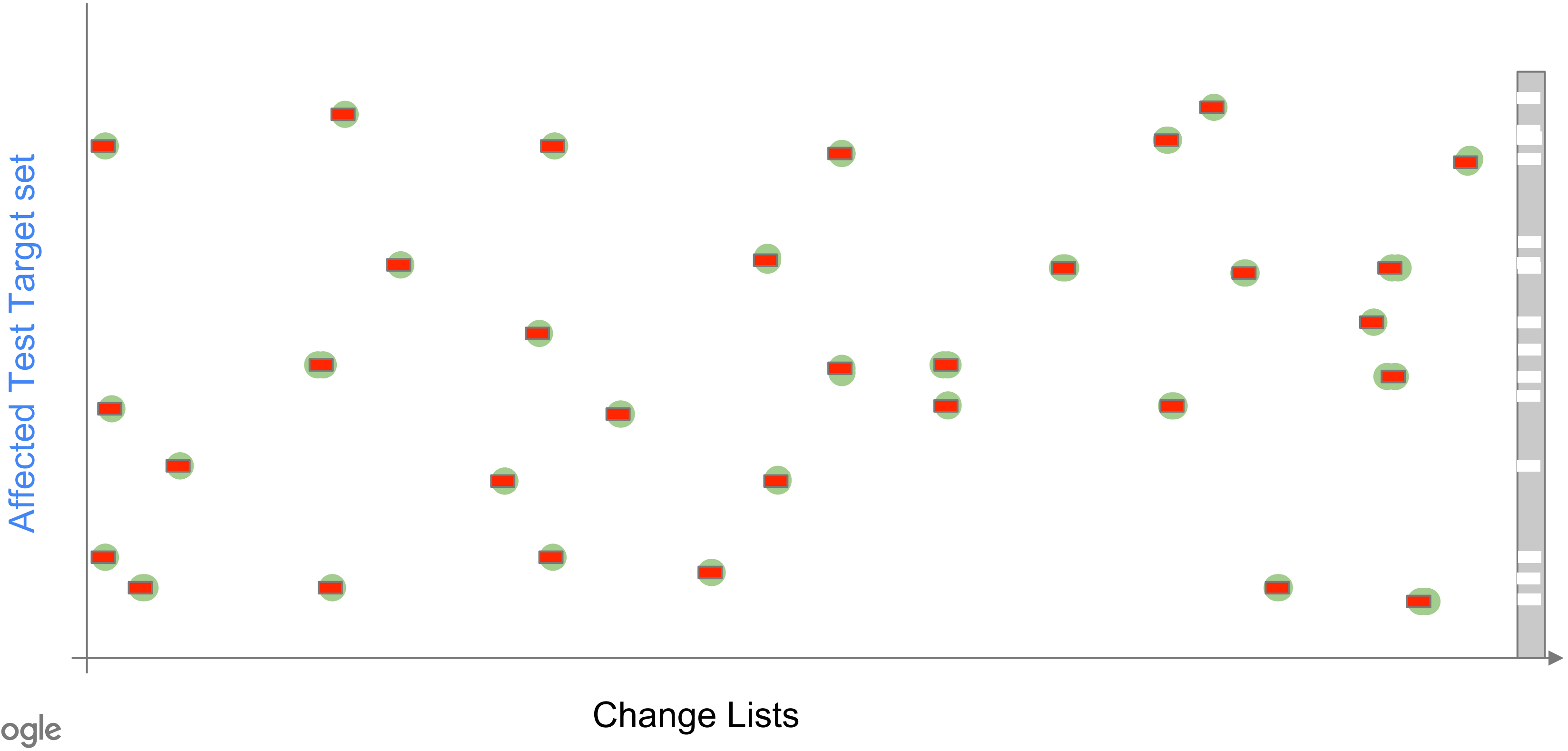
Google

Skipping milestones

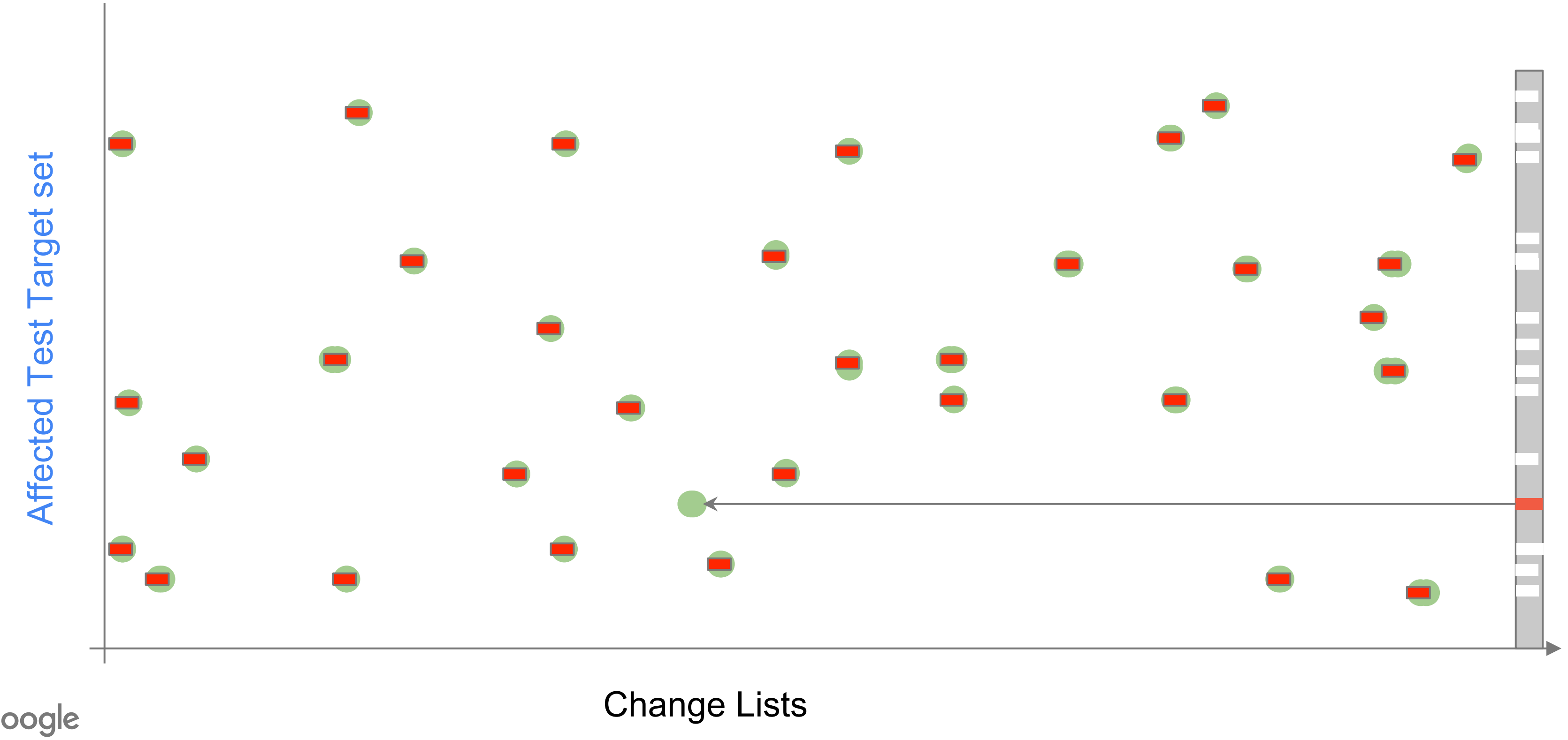


Google

Skipping milestones

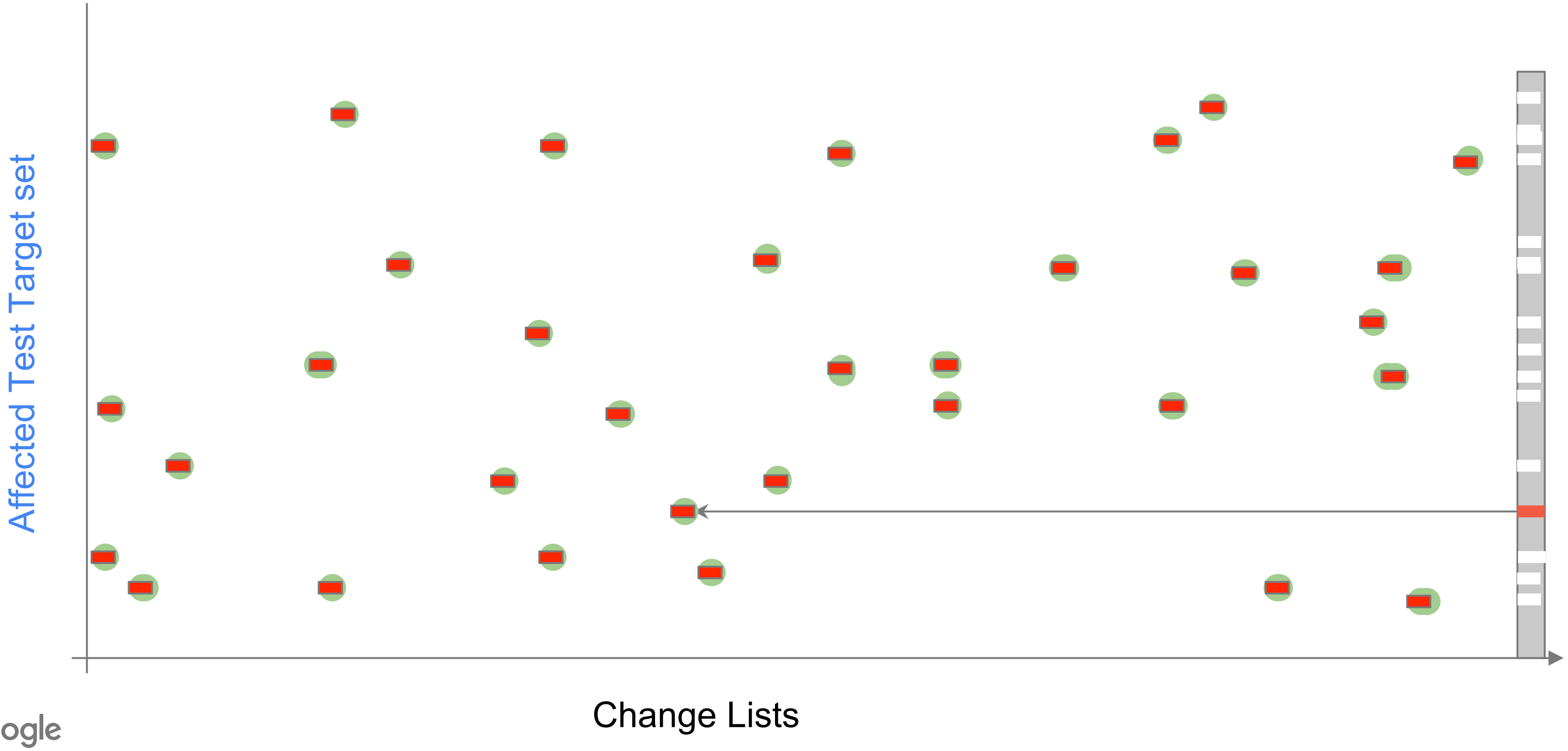


Skipping milestones: cuprit finding, acceptance tuning



Google

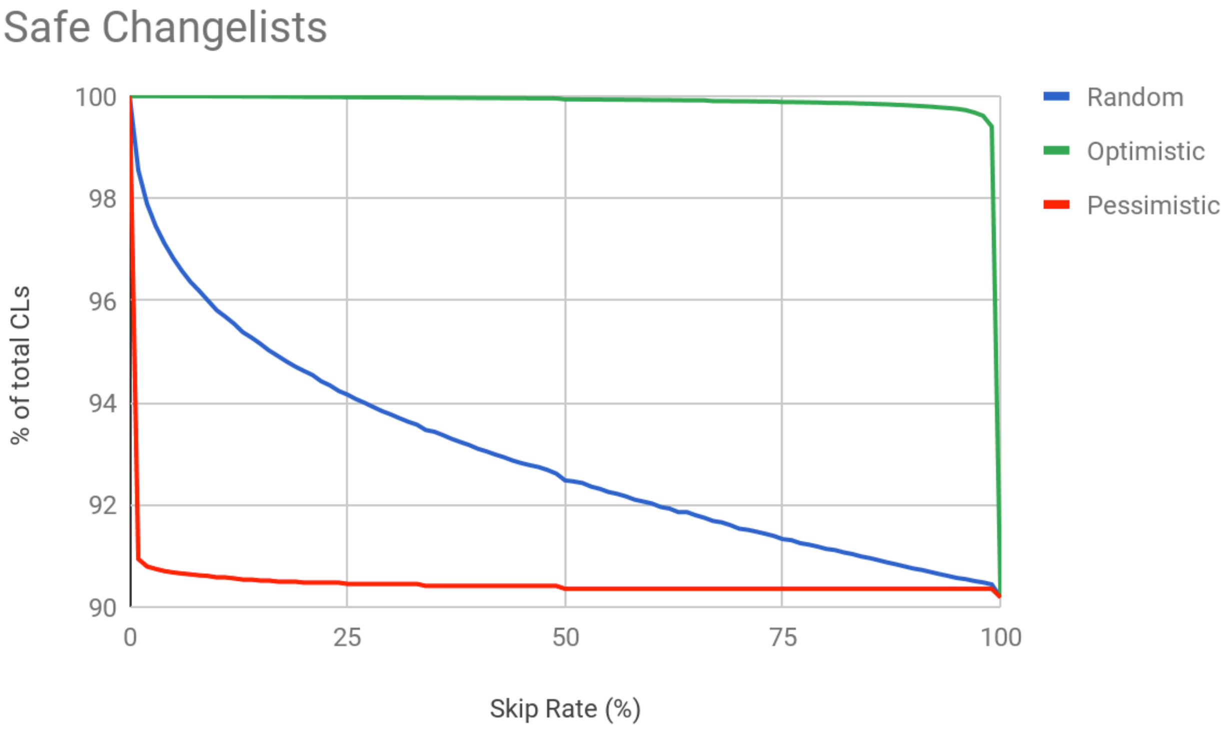
Skipping milestones: cuprit finding, acceptance tuning



Evaluating Strategies

- Goals
  - Low testing cost
  - Low time to find a transition
  - Low risk of missing transitions
- Exclude Flakes using 3 different exclusion mechanisms
- 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

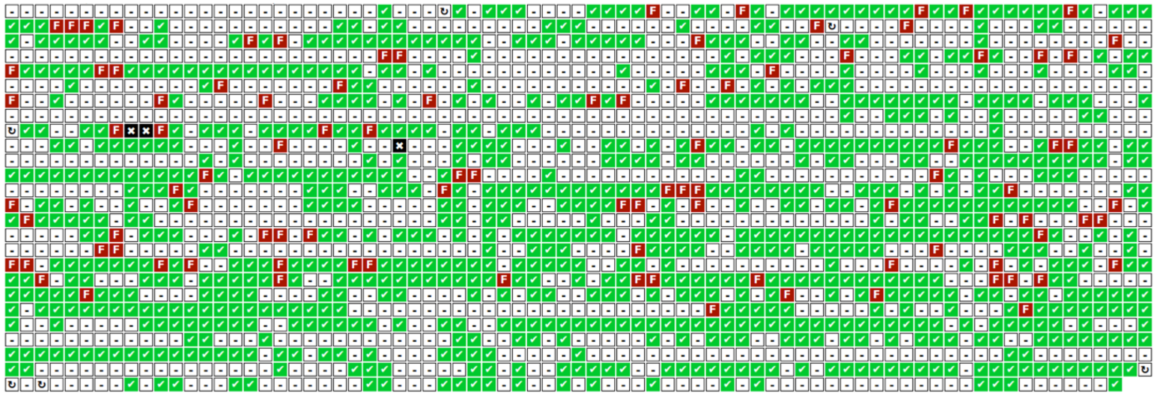
Offline Safety Evaluation



- 91% of changes do not cause a transition - we could safefly 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
- Random is a curve due to probability distributions and large impact changes

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



Analysis of Test Results at Google

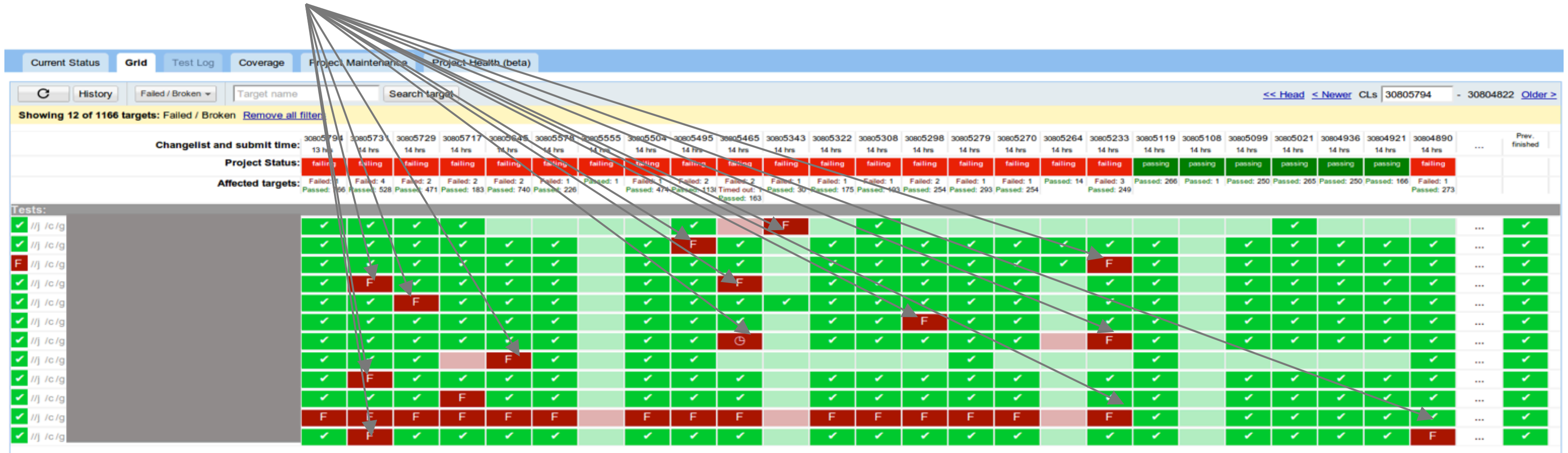
- Analysis of a large sample of tests (1 month) showed:
  - 84% of transitions from Pass -> Fail are from "flaky" tests
  - Only 1.23% of tests ever found a breakage
  - Frequently changed files more likely to cause a breakage
  - 3 or more developers changing a file is more likely to cause a breakage
  - 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

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

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

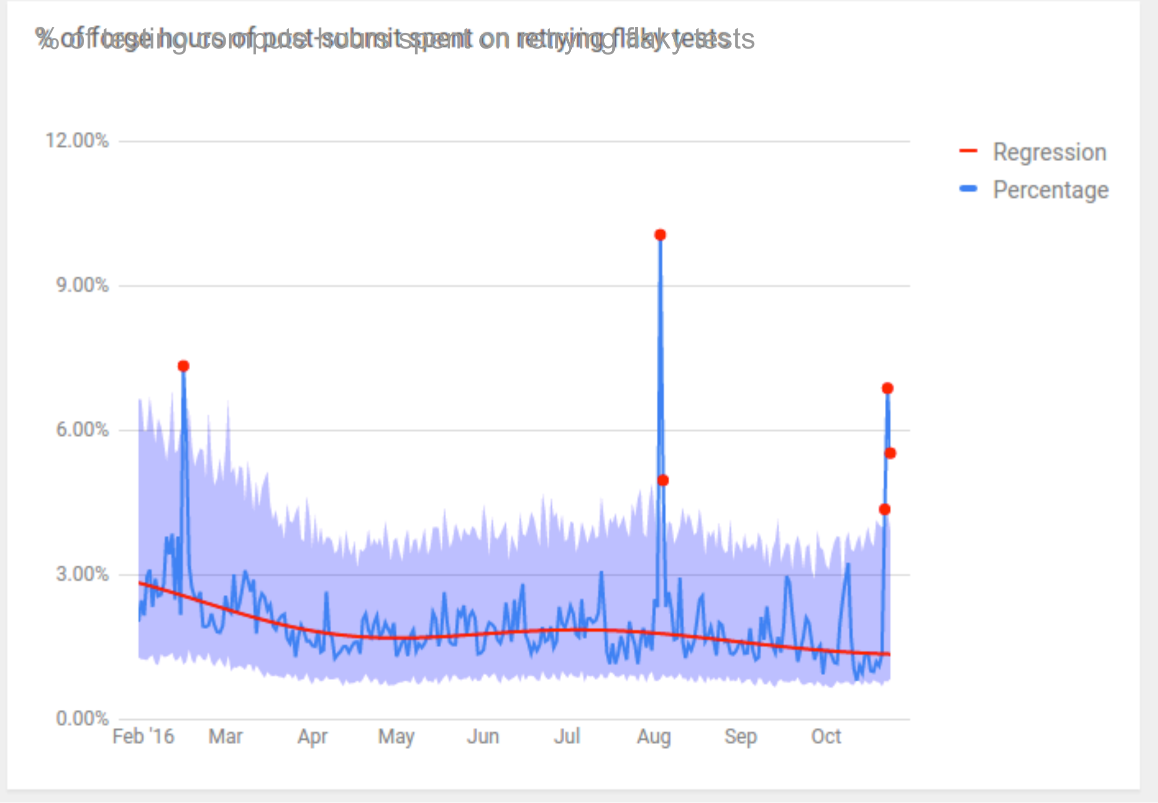
Flakes



Showing 12 of 1166 targets: Failed / Broken Remove all filters

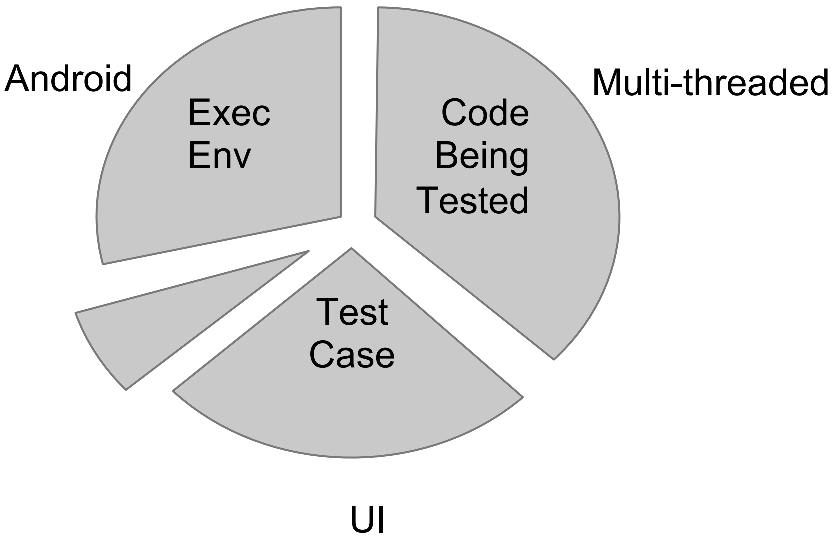
Changelist and submit time:	30805794 13 hrs	30805731 14 hrs	30805729 14 hrs	30805717 14 hrs	30805645 14 hrs	30805578 14 hrs	30805555 14 hrs	30805504 14 hrs	30805495 14 hrs	30805465 14 hrs	30805343 14 hrs	30805322 14 hrs	30805308 14 hrs	30805298 14 hrs	30805279 14 hrs	30805270 14 hrs	30805264 14 hrs	30805233 14 hrs	30805119 14 hrs	30805108 14 hrs	30805099 14 hrs	30805021 14 hrs	30804936 14 hrs	30804921 14 hrs	30804890 14 hrs	...	Prev. finished
Project Status:	failing	failing	failing	failing	failing	failing	failing	failing	failing	failing	failing	failing	failing	failing	failing	failing	failing	failing	passing	passing	passing	passing	passing	passing	failing		
Affected targets:	Failed: 1 Passed: 56	Failed: 4 Passed: 528	Failed: 2 Passed: 471	Failed: 2 Passed: 183	Failed: 2 Passed: 740	Failed: 1 Passed: 226	Passed: 1	Failed: 1 Passed: 474	Failed: 2 Passed: 113	Failed: 2 Timed out: 1 Passed: 163	Failed: 1 Passed: 30	Failed: 1 Passed: 175	Failed: 1 Passed: 193	Failed: 2 Passed: 254	Failed: 1 Passed: 293	Failed: 1 Passed: 254	Passed: 14	Failed: 3 Passed: 249	Passed: 266	Passed: 1	Passed: 250	Passed: 265	Passed: 250	Passed: 166	Failed: 1 Passed: 273		
Tests:																											
✓ //j /c/g	✓	✓	✓	✓					✓		F		✓									✓				...	✓
✓ //j /c/g	✓	✓	✓	✓	✓	✓		✓	F	✓		✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	...	✓
F //j /c/g	✓	✓	✓	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓	✓	F	✓		✓	✓	✓	✓	✓	...	✓
✓ //j /c/g	✓	F	✓	✓	✓	✓		✓	✓	F		✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	...	✓
✓ //j /c/g	✓	✓	F	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	...	✓
✓ //j /c/g	✓	✓	✓	✓	✓	✓		✓	✓	✓		✓	✓	F	✓	✓		✓	✓		✓	✓	✓	✓	✓	...	✓
✓ //j /c/g	✓	✓	✓	✓	✓	✓		✓	✓	◷		✓	✓	✓	✓	✓		F	✓		✓	✓	✓	✓	✓	...	✓
✓ //j /c/g	✓	✓	✓		F	✓		✓	✓						✓				✓						✓	...	✓
✓ //j /c/g	✓	F	✓	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	...	✓
✓ //j /c/g	✓	✓	✓	F	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	...	✓
✓ //j /c/g	F	F	F	F	F	F		F	F	F		F	F	F	F	F		F	✓		✓	✓	✓	✓	✓	...	✓
✓ //j /c/g	✓	F	✓	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	F	...	✓

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
  - ...



See: https://pdfs.semanticscholar.org/02da/46889ee3c6bc44bfa0fc45071195781b99ce.pdf

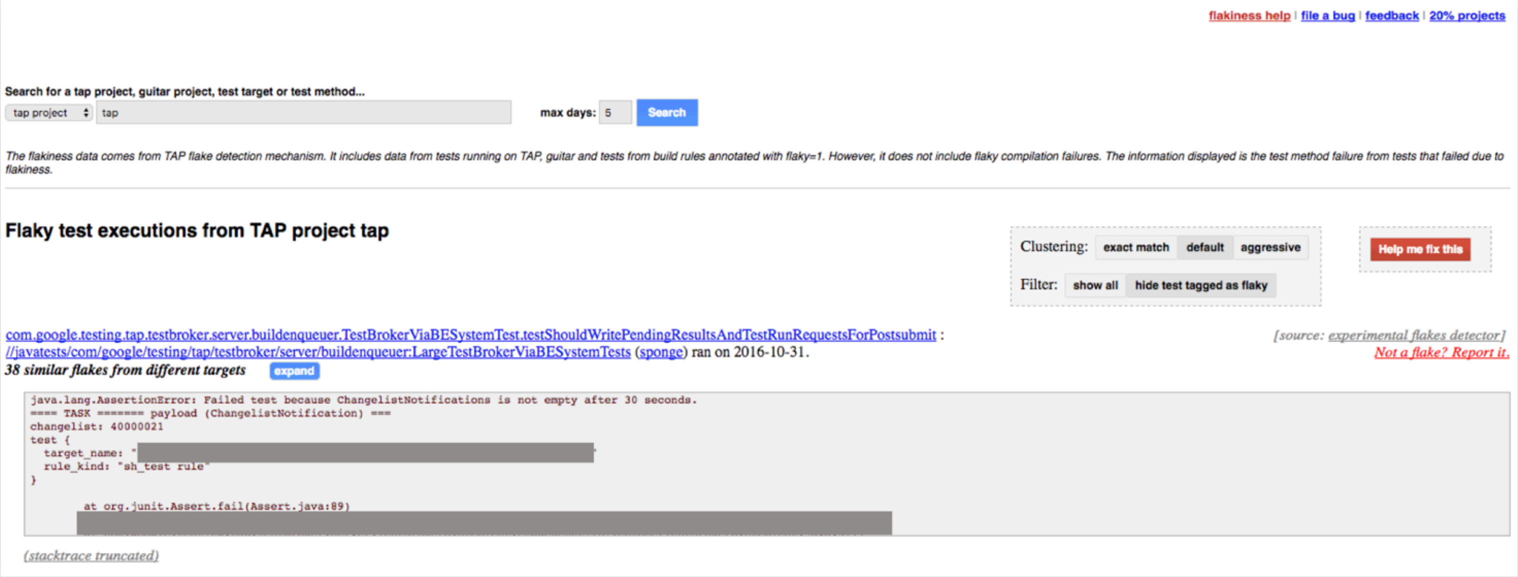
Flakes are Inevitable

- 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

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


flakiness help | file a bug | feedback | 20% projects
tap project
Search
Clustering: exact match default aggressive
Help me fix this
Filter: show all hide test tagged as flaky
expand

Search for a tap project, guitar project, test target or test method...

tap project: tap

max days: 5

The flakiness data comes from TAP flake detection mechanism. It includes data from tests running on TAP, guitar and tests from build rules annotated with flaky=1. However, it does not include flaky compilation failures. The information displayed is the test method failure from tests that failed due to flakiness.

Flaky test executions from TAP project tap

com.google.testing.tap.testbroker.server.buildenqueuer.TestBrokerViaBESystemTest.testShouldWritePendingResultsAndTestRunRequestsForPostsubmit : //javatests/com/google/testing/tap/testbroker/server/buildenqueuer:LargeTestBrokerViaBESystemTests (sponge) ran on 2016-10-31.

[source: experimental flakes detector]

Not a flake? Report it.

38 similar flakes from different targets

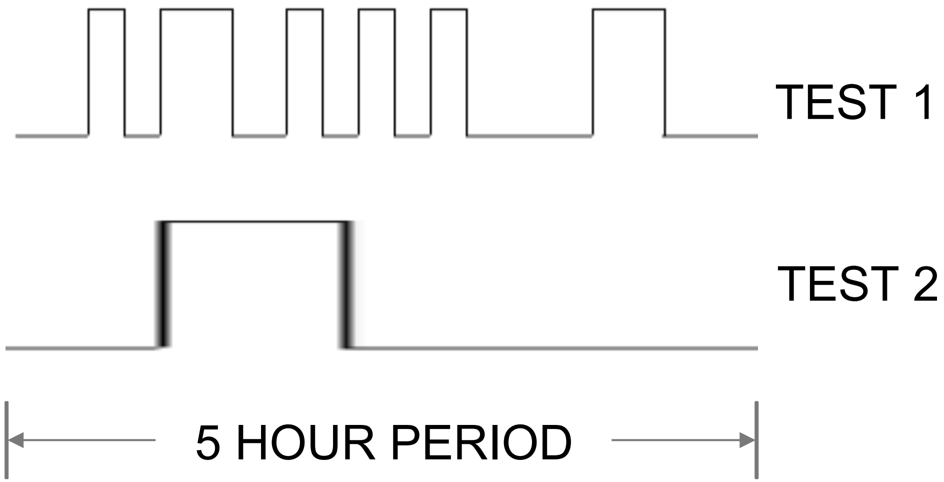
```
java.lang.AssertionError: Failed test because ChangelistNotifications is not empty after 30 seconds.
==== TASK ======= payload (ChangelistNotification) ===
changelist: 40000021
test {
  target_name: "[redacted]"
  rule_kind: "sh_test rule"
}

        at org.junit.Assert.fail(Assert.java:89)
        [redacted]
```

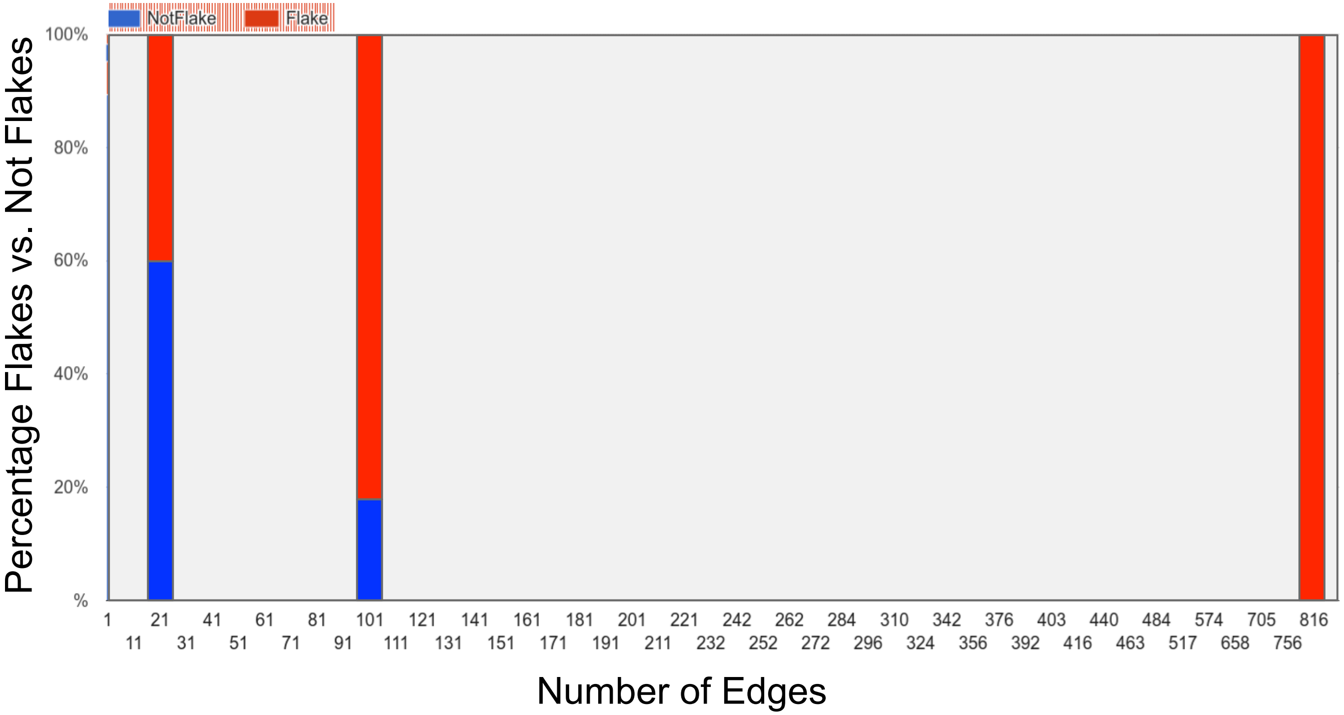
(stacktrace truncated)

Finding Flakes using the historical record

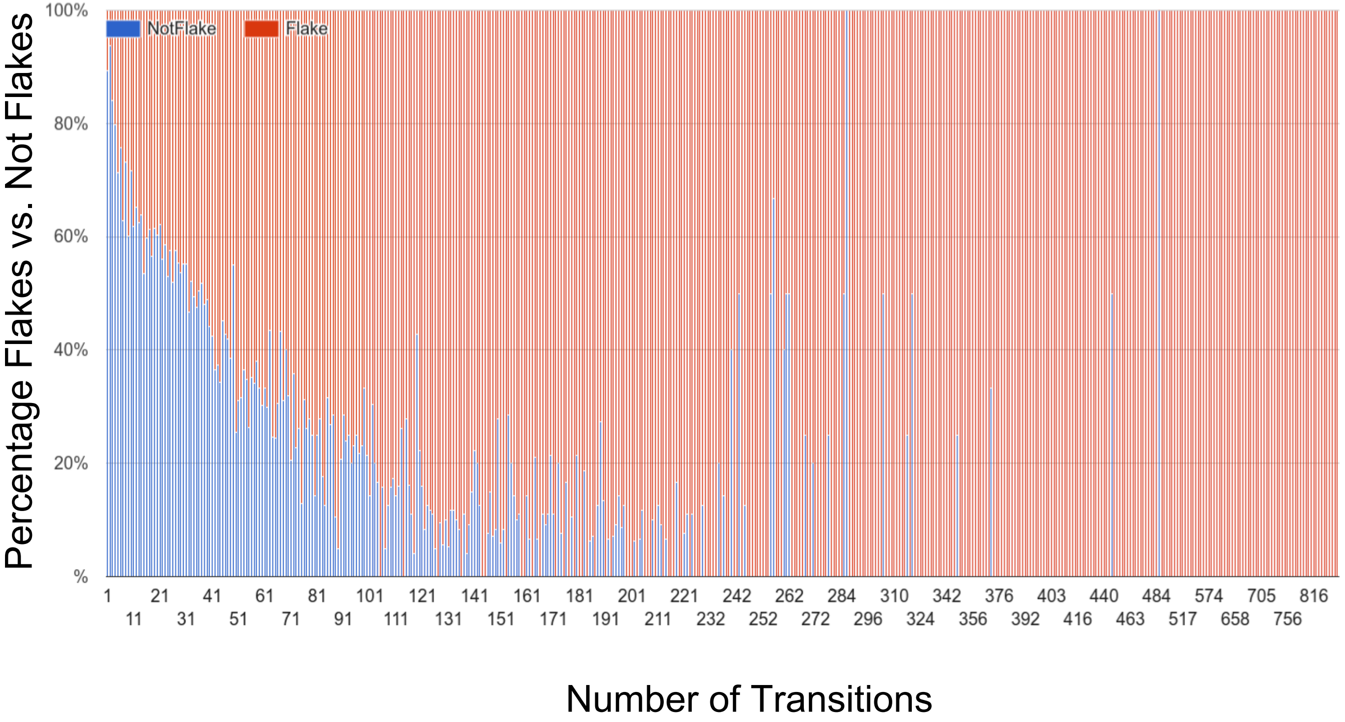
- 84% of test transitions are due to flakiness
- Concentrated in 16% of the total test pool
- Conclusion: Tests with more transitions are flaky



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!

- NOTE: A Google account is required for the hands-on portion
  - Send your Google account to john.micco@gmail.com before the lab if possible!

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
- (ICSE 2017) "Taming Google-Scale Continuous Testing," by Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski and John Micco