BROKEN BUILDS

AUTOMATIC DETECTION OF CHANGES THAT INDUCE TEST FAILURES

GROUP 16

MOTIVATION

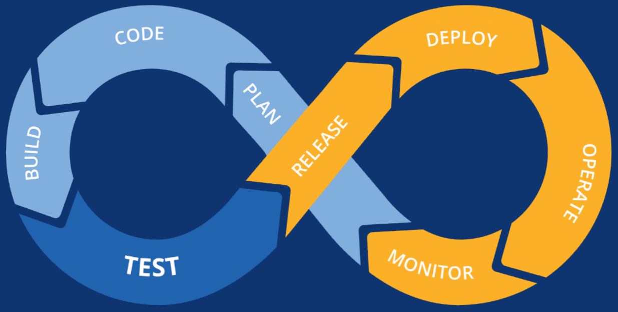
Modern software development happens at a fast pace. Changes to huge codebases are done by hundreds of individuals multiple times in a single day. It is no wonder that many of these changes introduce *bugs* that induce testing failures which result in *Broken Builds*. Identifying these faulty changes and addressing them is often a dreary and time consuming task. This has sparked interest in developing techniques that automate the identification of faulty changes and provide developers with tools to address them.

WHAT IS A BROKEN BUILD?

A broken build is a version of a code base that contains an unidentified bug that results in test failures or compilation errors

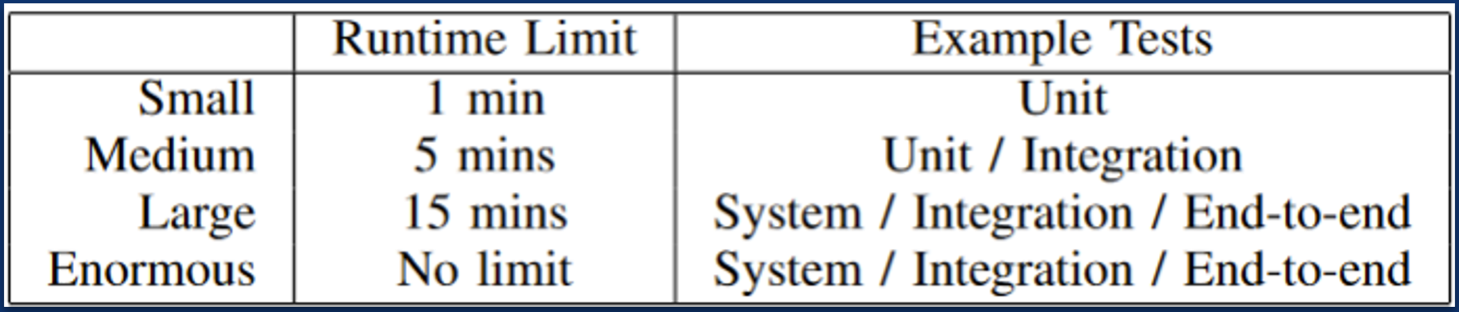
CONTINUOUS INTEGRATION

- Each change to the code base will trigger a compilation and suite of tests to run.
- Failures during continuous integration are called *regressions.*
- A regression can be triggered by a failed compilation and more often than not a test failure.
- Makes it easier for developers to identify the specific set of changes that triggered the regression.



TYPES OF TESTS

Tests can be classified by their length and run time. Often times these are separated into four categories. These categories become relevant when tests are used as part of continuous integration. The categories used by Google are shown below.



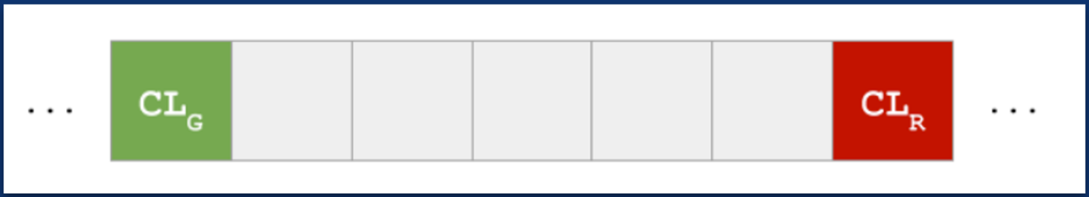
	Runtime Limit	Example Tests
Small	1 min	Unit
Medium	5 mins	Unit / Integration
Large	15 mins	System / Integration / End-to-end
Enormous	No limit	System / Integration / End-to-end

USING TESTS TO PREVENT REGRESSIONS

- Running all tests would hinder development speed
- Workflow is applied to increase efficiency
  - Small and medium tests are run before changes are integrated
  - Large and Enormous tests are run after
- Larger and Enormous tests run at a set time interval or after a certain threshold of changes.
  - Larger tests can fail at a version of the codebase which does not include the changes that induced the failure

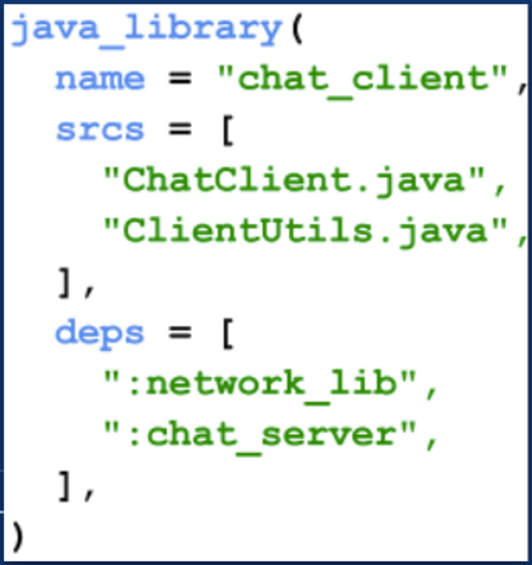
IDENTIFYING CHANGES THAT INDUCE A REGRESSION

- The list of changes that caused a regression must be in the range: $(\mathrm{CL}_G, \mathrm{CL}_R]$ where CLg was the version where all tests were passing and CLr was the version were the tests failed.
- Techniques to identify changes causing a regression
  - Binary Search / N-ary Search
- Due to the runtime of larger tests, these techniques are not a feasible solution.

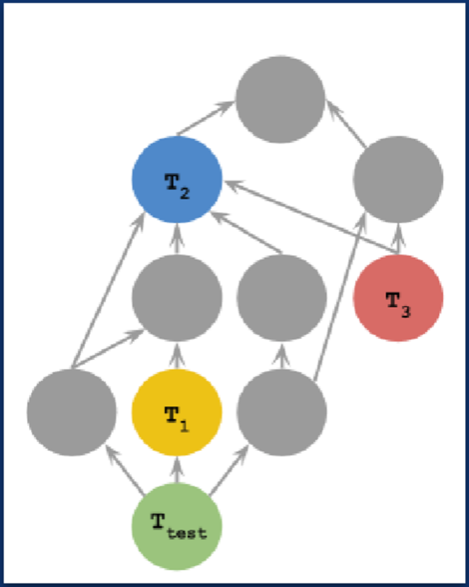


TARGETS

- To improve searching efficiency, we divide the repository into *Targets*
- Targets are logical structures that group source files with their associated dependencies
- Graphs, called Build Trees, can be used to represent Targets and how each Target depends on one another

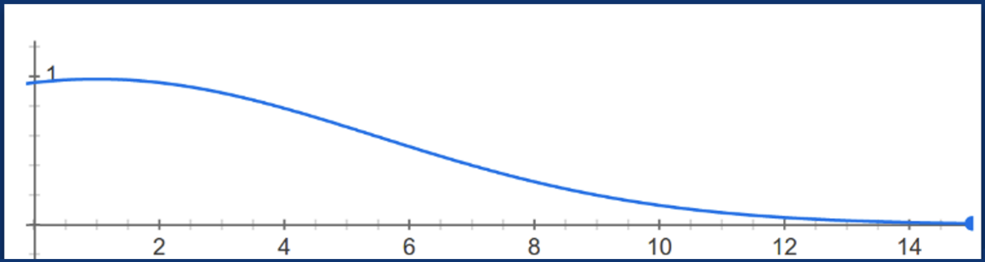


```
java_library(
  name = "chat_client",
  srcs = [
    "ChatClient.java",
    "ClientUtils.java",
  ],
  deps = [
    ":network_lib",
    ":chat_server",
  ],
)
```



EFFICIENTLY IDENTIFYING REGRESSION CULPRITS

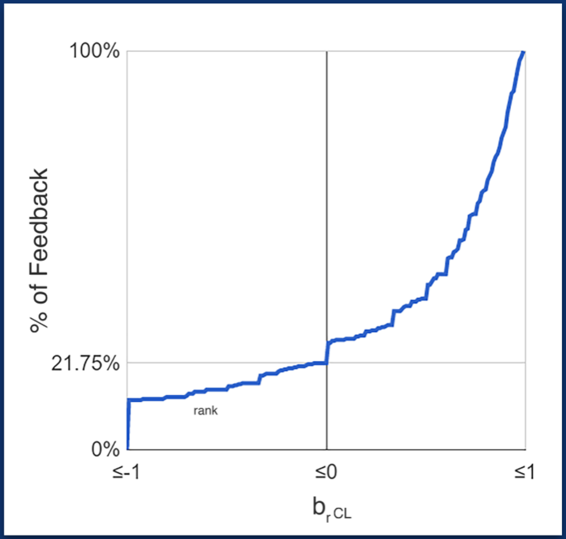
- Suspicious changes are those in the range $(\mathrm{CL}_G, \mathrm{CL}_R]$
- Suspicious changes can be ranked based on:
    - The number of dependencies between the change list and the failing test
        - If there are none, the changes can immediately be discarded as a suspect
    - The amount of changes made
    - Distance from Failing Target to nearest change in the CL's Build Tree
- Using these rankings each change can be given a suspiciousness score
- The suspiciousness score can help by
    - Narrowing the number of change lists we need to look at to identify the first broken build
    - Reducing the amount of resources (real and CPU time) needed to address a broken build

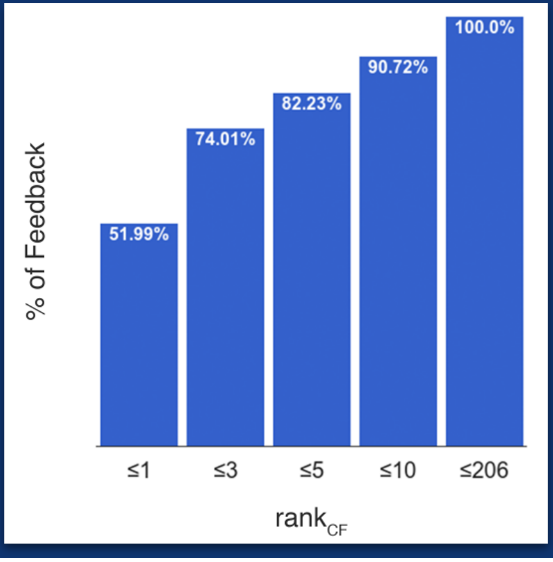


As the distance between targets increases the impact of the distance on the suspiciousness score decreases

EFFICACY OF USING SUSPICIOUSNESS SCORES

- In a study conducted at Google
  - 78.25% of the time utilizing the suspiciousness score led to a zero or positive benefit in reducing workload for identifying causes of broken builds
  - 51.99% of the time the culprit of the broken build was ranked as #1 in the suspect list, and 82.23% of the time, it was in top 5.





CULPRIT FINDER PERFORMANCE

- Performance will suffer under the following conditions:
    - Highly interdependent code
        - High amount of coupling
    - Large number of Targets
        - Time to traverse Build Tree increases
    - Large amount of changes in a Change List
- These factors also hurt performance of typical broke build finding methods

CONCLUSION

- Code repositories continue to increase in complexity, and the pace of development continues to grow faster and faster
- Continuous Integration offers a way to reduce bugs in software by running small/medium sized tests before allowing any changes to be made
- Finding the source of bugs that are detected by large/enormous tests can be a slow process that hinders development productivity
- The utilization of tools such as the *Culprit Finder* and *Suspiciousness Scores* has been shown to reduce the time and resources needed to identify these issues
- More research and development of novel techniques to identify where bugs are introduced is needed to keep up with the increased demand of software development

QUESTIONS

- Do you think these strategies are worth it for smaller scale projects, or is it only useful for huge repositories like Google’s?
- How do you see these strategies evolving in the future?

RESOURCES

Ziftci, Celal, and Reardon, Jim. “Who Broke the Build? Automatically Identifying Changes That Induce Test Failures In Continuous Integration at Google Scale –.” *Google Research*, Google, 2017, research.google/pubs/pub45794.

Slides Template and Icons – slidesgo.com