Better Flaky Failure Analysis

Michael Robinson, Jenava Sexton

mirobin@microsoft.com, jesexton@microsoft.com

Abstract

Test automation is expected to provide a consistent pass/fail signal when run. However, when run repeatedly with no changes, tests sometimes report different answers. The transient nature of these failures makes it difficult to assess changes in behavior – was the change in signal due to a recently introduced issue, or was it behavior that was already present at some low frequency? This unclear signal results in regressions being missed.

We have tried many different approaches to improve our reporting of these types of failures. While some approaches initially had a positive impact, ignoring flaky failures allowed more flaky regressions to be introduced. Over time, this negated the initially observed benefit and ultimately degraded the value of our entire suite of automation.

Our latest approach uses a baseline dataset and the results of a single rerun to analyze failures. The baseline dataset enumerates typical failures but also helps produce a statistical model that predicts low frequency failure behavior. Combining data from the baseline, a single rerun, and our statistical model, we can more accurately assess if a failure is new or existing.

Using this approach, we have seen a sustained improvement in validation job pass rates, moving from 55% to 80%. We have also seen a substantial reduction in duplicate jobs, higher pass rates for validation jobs associated with submissions, and a dramatic reduction in negative developer feedback regarding automation results.

Biography

Michael Robinson has been working on test automation at Microsoft for 19 years. During that time, he has worked on automation systems and frameworks used to validate Microsoft Office at scale and has seen those systems grow from executing a few hundred tests a week to millions per day. Michael has a Bachelor of Science in Computer Science from the University of Missouri-Rolla. He is married with two small children and enjoys playing video games, taking care of his cars, everything related to technology, and gardening.

Jenava Sexton is a Senior Product Manager at Microsoft, working with test automation systems for the MS Office Engineering System. She has worked in her role for three and a half years. Over the course of her 20-year career, Jenava has created and implemented many business systems across a wide variety of industries and company sizes. She is passionate about building software that makes complex business processes and decisions clear and straightforward for users. Jenava earned an MBA from the University of Washington and a BA from Bennington College. She has two small children, a spouse, and a large standard poodle. In her free time, she enjoys sewing, crocheting, and yoga.

1 Introduction

Test automation is used to help assess product quality and verify it performs as expected. Our system runs a series of these tests against a developer change to verify the change has not introduced a regression. When the tests have finished, our system evaluates the results and sends the developer a signal indicating if their change introduced a regression.

Ideally, the test's results would provide a consistent signal assessing the product's performance. In practice, most tests do not produce a consistent signal; not all failures represent a regression introduced by the change being tested. As a result, determining if a regression was introduced requires some level of analysis that considers information beyond the test's result.

Errors in this analysis can result in regressions being introduced into the product. An analysis that is not sensitive enough will hide regressions that were detected by automation. Analysis that is too sensitive will cause real regressions to be lost amongst the noise of unrelated failures. The quality of this analysis also has direct effects on how much developers trust automation results, and how much effort is wasted investigating unrelated failures.

2 Background

Our system supports 2,000 developers, who trigger 14,000 jobs/day, performing 5.5 million test executions/day.

Developers in Office use our system by running a command which looks at the changes they've made. It identifies tests to run against those changes, packages the build outputs and test dependencies, then creates a validation job to test those changes. The validation job includes steps necessary to obtain test machines and deploy the product with their packaged changes. The job may consist of a handful of tests up to tens of thousands of tests; typical jobs consist of hundreds to low thousands of tests. Once the job is created and the command exits, developers wait for an email that reports on what was found during their job.

The tests run in our system are predominately end to end "scenario" based tests. They are responsible for copying dependencies, starting the product and other setup steps, interacting with and verifying product behavior, before finally shutting down and cleaning up. A single scenario is functionally equivalent to tens of thousands of unit tests and spends substantial periods of time exercising code unrelated to the specific test. Issues with automation system infrastructure, unrelated product code executed while setting up the test, bugs in the test itself, the underlying operating system, or even hardware malfunctions can result in a failure and contribute to flaky behavior.

Tests in our system report their results by writing log entries; a failure is simply log entry containing a human readable string flagged as a failure. When our system receives a failure, it clusters it with other related failures based on the failure message and where it was logged from. This allows us to correlate failures across different automation jobs and builds. The method we use to group failures is not perfect – however its accuracy is roughly as good as what we would see with a human performing this task.

The nature of the tests we run means that there are thousands of different ways each test could fail for a reason unrelated to the change being tested. Statistically, given the volume of automation we run, we are likely to observe some of these failures in every validation job we run. Individually, the rate at which these occur is extremely low, but when combined with the total number of possible failures, you end up with tests that fail for seemingly random reasons. The repro rate for a specific one of these failures is very low, making it unlikely that the failure can be observed by simply rerunning the test a few times. These are perceived as "flaky" failures by developers using our system.

3 Problem

If our tests have a 99% pass rate, and we run a validation job with 100 tests, there is only a 37% chance that all of the tests will pass. If we run 1000 tests, there is a 0.004% chance that all tests will pass.

Clearly, we can’t assess the result of a validation job by seeing if any of the test failed – something *will* fail even if there is no defect in the change being tested. The challenge is separating failures caused by the change being tested from failures that would still occur absent the changes being tested.

This problem is exacerbated by the imperfect nature of how we group failures together. This can cause existing failures to look like new failures, or it can hide new failures amongst other existing failures.

Due to the broad amount of code covered by a scenario-based test, there is substantial resistance to disabling tests. Even a failing test still tells you everything is functioning correctly up to the point of failure. Disabling a scenario-based test for one failure is roughly equivalent to disabling all unit tests for a component because one unit test failed. This prevents us from solving the problem by requiring tests to meet sufficiently high pass rate.

4 Past Solutions

Over the years we have tried several different ways to filter out or ignore flaky failures, with varying degrees of success. Ultimately, none of these produced results we were happy with.

4.1 Class A / B

One of our earliest attempts at dealing with this problem was to place tests into one of two groups: Reliable (Class A) and Unreliable (Class B). Failures in Class A scenarios were considered regressions that needed to be fixed, while failures in Class B scenarios could be ignored. In practice, Class A scenarios were not immune to flaky failures, and this ultimately did not help improve job pass rates. Results from Class B scenarios were ignored, causing legitimate regressions to be ignored. Worse, once a Class A scenario was demoted to Class B it would tend to remain there, degrading the overall effectiveness of the automation suite.

Using this approach, automation job pass rates remained < 1%.

4.2 Fix the tests

Over the years there have been various efforts to remove tests with high failure rates, or to fix test bugs to improve their overall pass rate. It sounds simple – just fix the broken tests! Unfortunately, statistics works against us here. To have a 99% chance of a 1000 test automation job avoiding a flaky failure, each test needs to be 99.999% reliable. The engineering effort required to get tests to perform at this level is extremely high, and tests will not stay at that level without continuous maintenance. Teams, understandably, are not willing to invest that amount of effort maintaining their test automation.

This approach reduced the total number of flaky failures seen in automation jobs. It had little impact on the overall job pass rate, benefiting primarily teams that had few tests.

4.3 Enumerate all failures

There have been a few strategies based around the idea of fully enumerating all failures that tests could possibly generate. This strategy depends on being able to identify most failures within a reasonable number of executions of the automation suite. In practice this was able to identify some flaky failures and substantially improved result quality over the preceding solution, but in absolute terms it still wasn’t great. Knowing what we know now, our models suggest we would need to run the entire automation suite nearly 68,500 times to get to a point where only 10% of jobs encountered an unknown failure.

This strategy was able to increase automation pass rates to about 25%.

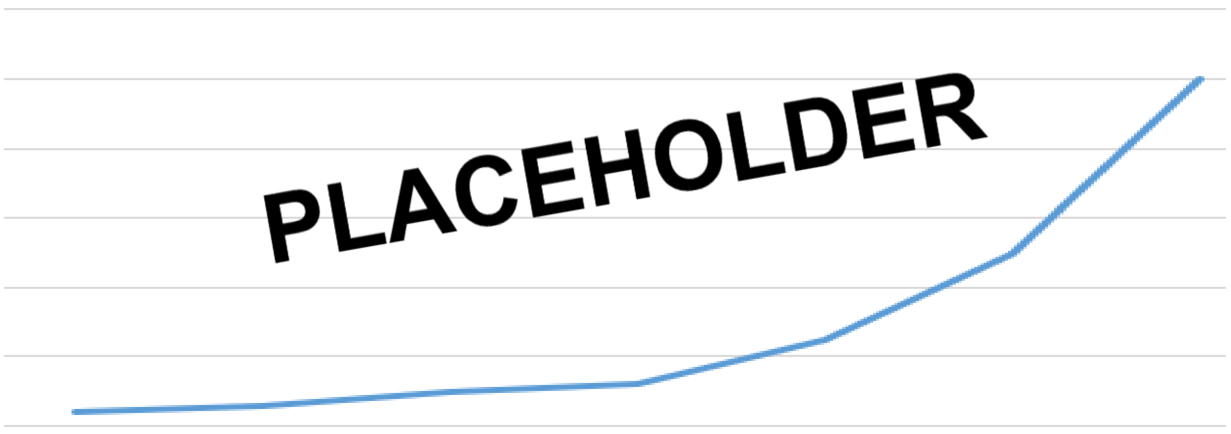
4.4 Rerun failures

This strategy seeks to filter out any failure that doesn't reproduce consistently. Any test which passes on rerun is considered a pass, effectively increasing the pass rate of a test. This approach did initially improve automation pass job rates, but that improvement did not persist over time. This strategy permits any non-deterministic failure to be checked in, ultimately degrading the overall pass rate of the automation suite. Over time more and more reruns are required to get the same result. Left unchecked long enough, tests will start to fail for different reasons every time they are run.

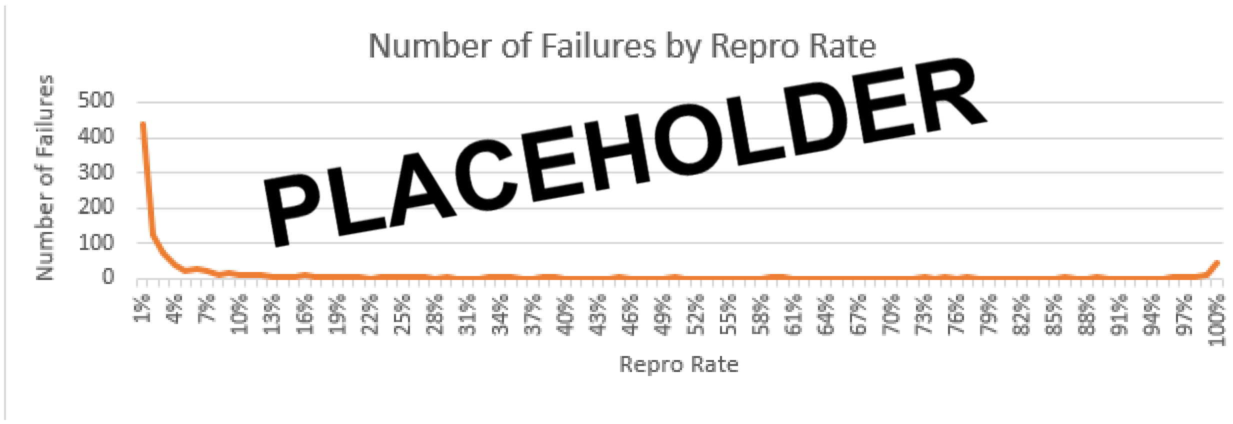
This strategy usually improves job pass rates initially, but without some sort of backstop to guard overall automation health that improvement is lost over time.

5 Insights

Our past attempts to solve this problem were based around assumptions regarding the distribution of failures by their repro rate. We assumed that most failures that a test can generate reproduce consistently, and that each test has a small number of failures that occur a small percentage of the time:



The actual distribution of failure types does not match assumed behavior at all!



It is quite clear that the flaky failure problem is dramatically worse than assumed and explains why rerun and enumeration based attempts to limit their impact did not have the expected result.

Our first observation was that the number of failures on the low side substantially outnumbers failures on the high side. Worse, every time we run the automation suite, the distribution of failures on the above graph can *only* move to the left – our set of consistent behavior is finite and can only shrink as we continue to run our tests!

Our second observation was that failures only present with consistent behavior or flaky behavior – there are very few failures which reproduce at rates between 10% and 95%.

We also noticed that each time we ran our automation suite, we found new failures that hadn't been seen in the previous iteration.

6 Analysis

Instead of using reruns to determine if the test can pass or not, what if we used reruns to determine if the failure is on the low or high side of the repro chart? If we get the same failure when rerunning, it is a high frequency failure. If it passes or returns a different failure on the next run, we're looking at a low frequency failure.

6.1 High frequency failures

Reporting failures on the high side is straightforward – we can fully enumerate those by just running the tests a few times. If a failure exhibits high frequency behavior in the validation job and in a baseline dataset, we know we're looking at an existing issue. Otherwise it is a regression!

6.2 Low frequency failures

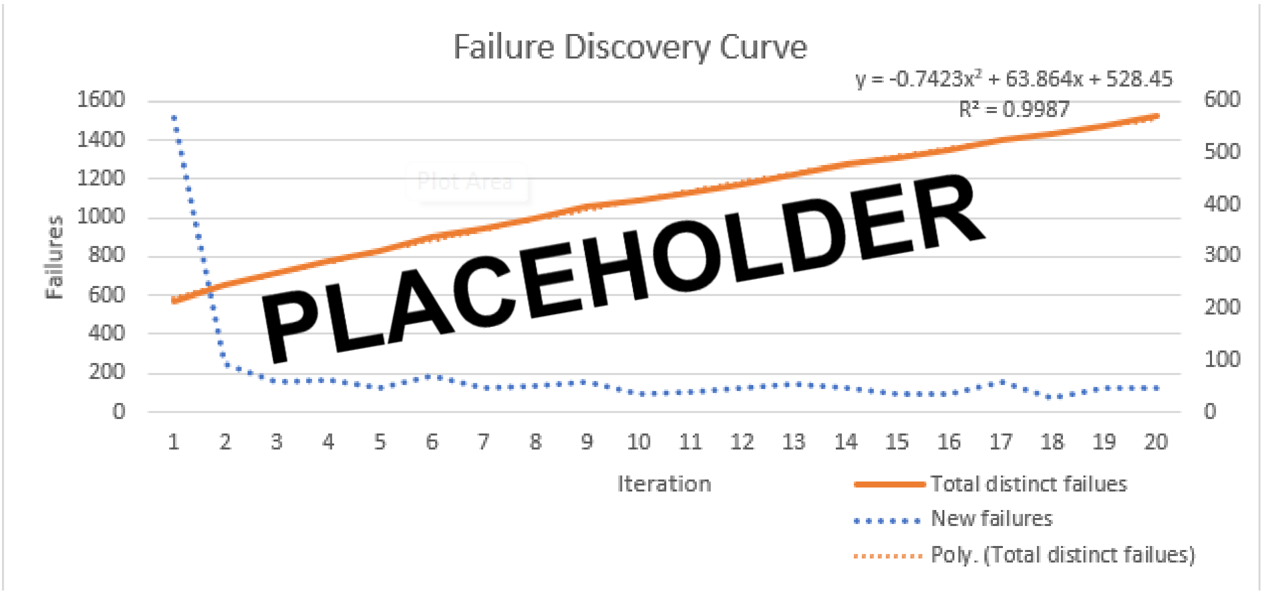
Reporting failures on the low side isn't as simple, as we are unable to fully enumerate these failures. Our instinct is to ignore these types of failures – dismissing them as random or "not real", but there are many legitimate issues that can surface via flaky behavior.

As an exercise, consider the fallout from a bug that causes a server to restart every 10 minutes.

We know from experience that simply ignoring these failures is not a viable long-term solution. What we need is a way to determine if the behavior we are seeing is normal; could we possibly model expected behavior and compare that against what actually happened?

6.2.1 Aggregate Failure Threshold

Let's go back and look at our baseline dataset… Even though each baseline job hits a similar number of total failures, not all the failures are the same – each job "discovers" new failures that were not hit by previous jobs. As additional jobs are run, we also observe that the rate we discover new failures decreases. In fact, if we graph the total number of discovered failures by iteration, we end up with a curve that can be fit with a non-linear regression!



This means we have a formula that can be used to predict the total number of failures we should know about by a given iteration! If we generate that formula using N baseline jobs, a prediction for the number of new failures we should expect to see in a change validation job represents N+1.

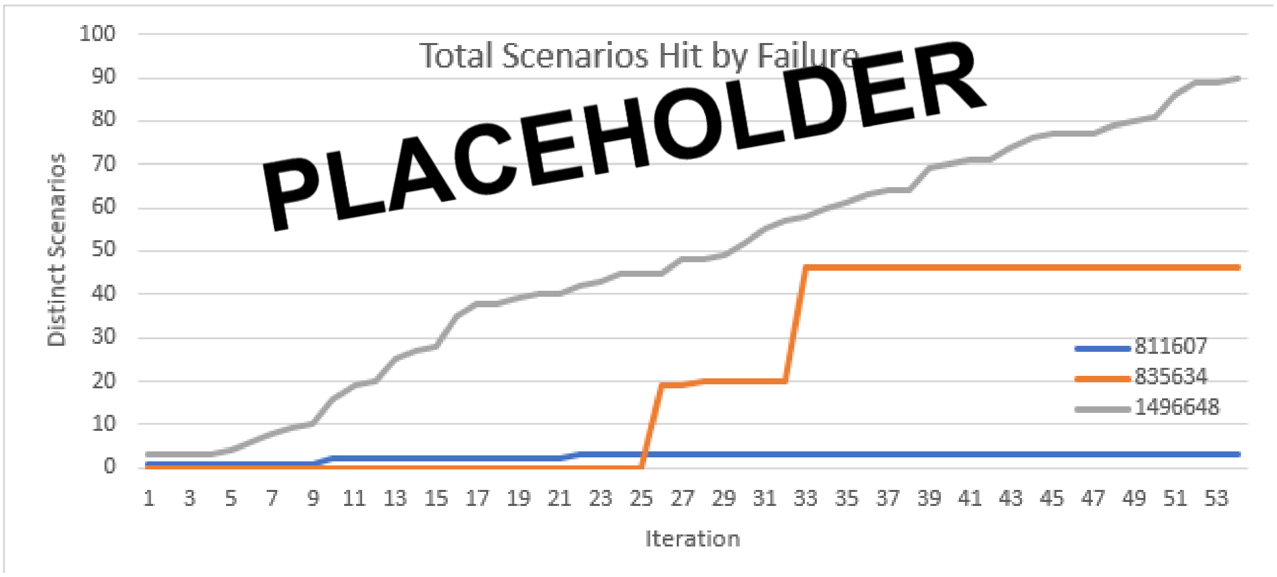
This model can be used to establish a threshold for flaky behavior that applies to the entire validation job. The number of permitted flakes changes based on how reliable the tests are in the first place, serving as a backstop that helps keep tests performing at their existing level.

If a developer change introduces a flaky regression, the total number of new flakes discovered during their job will be larger than the prediction, and we can communicate that break to the developer with their job results.

6.3 Failure Signatures

The failure threshold is assessed at the job level, meaning it doesn't actually determine which failure(s) were caused by the regression – they're mixed in amongst the other new failures we were expecting to see. It would be nice if we could point to specific failures that could be responsible for the regression.

Traditionally when looking at failures, we look at it from the perspective of failures that a test has hit. Instead, what if we switched that around and looked at tests a failure has been seen in? When looking at failures from this perspective, we see that failures "discover" tests in predictable patterns.



6.3.1 Decreasing

Failures with this signature typically discover most of the scenarios they impact the first few jobs they were hit. As more jobs are run, the failure is hit by fewer and fewer new scenarios. Failures matching this pattern often impact one area of code and can only surface in tests which exercise that functionality.

If we observe a failure with this signature hitting many new scenarios in a validation job, it indicates that a different area of code is now surfacing this behavior, representing a behavioral change/regression.

```
outlook is still running. Killing the process.
```

6.3.2 Constant

Failures with this signature discover new tests every time they run, and the rate of discovery is roughly the same over time. Failures matching this pattern often exist in code that is broadly used and are not triggered by actions performed by tests.

If a validation job shows that the scenario discovery rate has increased, it means that the code change being validated made the existing issue worse in some way.

```
_BroadcastInvoke: hResult-E_POINTER
```

6.3.3 System

Failures with this signature discover new tests sporadically. The failure may not be seen for many iterations, then suddenly discovers many tests. Failures matching this pattern represent issues with the underlying automation system or its infrastructure and are not the result of a code change.

```
System.Net.WebException: Unable to connect to the remote server --->
System.Net.Sockets.SocketException: No connection could be made because the target
machine actively refused it
```

6.4 Combining it all together

Prior to running a validation job, we run a series of baseline jobs to generate the data necessary to model low frequency failures and to enumerate the expected set of high frequency failures.

While running a change validation job, any test that fails is rerun once.

When the validation job has finished, separate failures into two groups: high frequency and low frequency.

Individual high frequency failures are considered regressions if the failure was not observed in the baseline dataset.

Individual low frequency failures are considered regressions if the test discovery rate is higher than observed in the baseline.

The total number of low frequency failures is also compared against predicted value modeled from the baseline dataset; the job has introduced a regression if the number of new low frequency failures is higher than the predicted limit.

7 Results

7.1 Increase in pass rates

Before the introduction of this approach, the average automation job pass rate was 58%. Pass rates across teams varied from 20.7% for a team with 51,000 tests, to (higher %) for a team with (smaller number). Generally, teams that had worse pass rates before the new algorithm was implemented saw a larger jump in pass rate. In one case, a team that has a (n)% pass rate jumped to an average of nearly 100% and has remained at this level.

Of course, sometimes automation fails because it has detected a regression, and in that case, we expect the algorithm to correctly highlight the failures that are caused by the code change. It is much more difficult to measure if the algorithm is more accurate, and not simply more permissive. However, we did see some teams that had reduced pass rates in the new algorithm, and we also saw that many individual jobs flipped from pass to fail and fail to pass at similar rates so that the difference in signal was not accurately represented by mearing average pass rate.

7.2 Reduction in duplicate validation jobs

Our previous approaches to flaky failures would case users to re-run tests and entire jobs, trying to get a passing signal. With a clearer signal and higher efficiency with a single failure re-run, the load on our automation system has declined from n machine hours per month to n.

7.3 Lower instance of flaky failures causing job failures

One of the most clarifying outcomes for users, and one that contributed to the increase in pass rates, was that most flaky failures could be confidently presented to users as unimportant much more frequently. The noise that users saw in their job results was reduced.

Lastly, our users regularly and verbosely complained about the automation signal in our quarterly engineering systems satisfaction survey. After implementation, the negative comments directed towards automation slowed from N to almost none. The topic of automation flakiness fell from one of the top N complaints about the engineering system to Nth place in frequency of mentions