Winning with Flaky Test Automation

Wayne Matthias Roseberry

wayner@microsoft.com

Abstract

Flaky test automation drives everybody nuts. Run it once, it fails, run it again - nope! So you stop running those flaky tests, and everyone is happy until a nasty bug gets in there that those tests would have caught.

It is a trap. If you want to run faster and release more often, you cannot afford the wasted time from noisy flaky tests, but in order to go faster, you have to know about the bugs. This paper will discuss how the Office team came to terms with this problem, embraced the reality that real bugs live under the flaky crust and figured out how to use those tests, and their flaky behavior, to their best advantage.

Biography

Wayne Roseberry is a Principal Software Engineer at Microsoft Corporation, where he has been working since June of 1990. His software testing experience ranges from the first release of The Microsoft Network (MSN), Microsoft Commercial Internet Services, Site Server, SharePoint and Microsoft Office. He currently works on the Office Engineering team, with a focus on test automation systems, strategies and architecture.

Previous to working for Microsoft, Wayne did contract work as a software illustrator for Shopware Educational Systems.

In his spare time, Wayne plays music, paints, and writes, illustrates and self-publishes children's literature.

1 Introduction

Flaky test automation presents a frustrating and special challenge to any team of software engineers. The tensions to mitigate risk with as much coverage as possible while maintaining small, safe feedback loops with efficient and fast releases pull in opposite directions. Tests that do not yield a consistent signal with failures that do not reproduce easily make that tension even more difficult to manage.

Unmanaged, the problem creates an enormous technical debt that destroys the value of the test automation. Engineers ignore results or bypass processes meant to protect the product from releasing bugs into the market. Teams and projects pay huge costs in lost time, investigation, or attempting to patch bugs that escape.

The problem of flaky automation becomes even more relevant as development cycles increase in speed and releases increase in frequency. Attention shifts from long test phases post code complete to short and fast release cycles and all the way to the developer desktop where developers execute tests during coding (the developer inner loop).

This document describes approaches by the Microsoft Office product group to manage flaky automation. Some of these approaches are centrally run by the engineering team, others are one-off methods used by specific teams or individual engineers within Office. While the examples here are mostly from Office specifically, the practices and approach that are emerging across Microsoft, and outside Microsoft in other companies, appears to be similar.

This document prescribes a three-part approach:

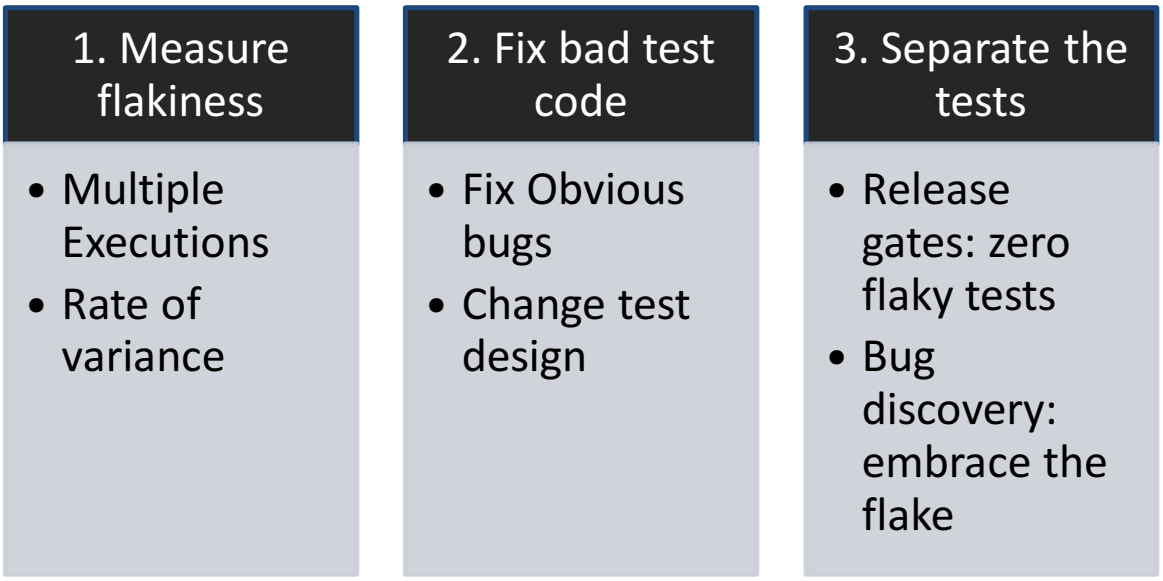

Figure 1 Approach for Addressing Flaky Automation

2 Definition and Measure of Flaky Automation

Automation is flaky when it does not give a consistent signal. This may be because of flaws in the test code, but it is also frequently the system under test (SUT) that yields an inconsistent result. For sake of this document, inconsistency shall be measured in the following way:

1. Run a test (or suite of tests) multiple times under the same conditions (build, environment, etc.)
2. Count the number of distinct failures the test yields. A test may fail more than once, but do so differently. These would be separate distinct failures.
3. Count the number of times the test fails.
4. Calculate a reliability rate that rewards for consistent pass or failure, and penalizes for variation

```
Consistency = IF Failures > 0 THEN
 (1 / Distinct Failures) * (Failures / Executions)
ELSE 1
```

The formula above yields a measure of how often a test gives a variant result, whether or not it passes or fails. For simplicity, one can reduce the formula to a binary measure

```
IsReliable = (Consistency == 1)
```

It is then also useful to assess how often a given test is consistent across builds (`CountIf(IsReliable)/Count()`). This gives an image of behavior of a test or suite of tests over time. It is likely you will find that a subset of tests remain in constant stability/reliability flux, while others remains largely stable over time.

3 Test Characteristics and Flaky Automation

Test automation is problematic if it is being used in a way that it is not suited for. Flaky automation that is the result of buggy test code is always bad. Flaky automation that is the result of buggy product code is serving its primary purpose, which is to expose the bugs, but such automation is bad when the test is being used improperly.

We need to consider some aspects of test automation and how they relate to test consistency.

3.1 Precision Versus Recall

Test automation can be optimized to yield a precise pass/fail signal, to detect exactly one failure condition at a time. Test automation may also be optimized for recall, to find as may failures as possible. When test automation is optimized for precision, it compromises on recall. When test automation is optimized for recall, it compromises on precision.

The precision versus recall classification dichotomy cuts across many other types of tests. It is more about the test's optimizations than it is about exactly what kind of test is happening or exactly what process the test reflects.

3.1.1 Precision Tests Are Highly Stable

Precision test validation is binary in nature. The result of the test is an absolute PASS or an absolute FAIL, requiring no further analysis of the results to establish the result.

Precision automation targets specific anticipated failure cases and fails exactly for those cases. The failure message will be something precise, such as "*Expected: 3, Actual: 2. Ensure that the newly added parent node relationship is counted as part of the set.*" Precision tests will fail for no reason other than

what the test code asserted or unanticipated extremely critical conditions such as application crash. Unit tests are an example of high precision automation.

Precision automation is necessary for high speed, critical path processes, such as developer inner loop, code submission, and build release. The faster the release cycle, the more necessary the test precision. The process does not afford spurious failure beyond very specific targets.

Flaky tests are sometimes introduced as precision tests, but they fail to meet the goal by the way they introduce spurious, unanticipated failures into the result set. Time is lost trying to track down root cause, or trying to reproduce the failures.

3.1.2 Recall Tests Are Flaky by Nature

Recall tests force us to be analytical. The final PASS or FAIL state of the test will often require analysis of logged test result data, and sometimes deep team triage.

Recall tests are designed to catch failures not anticipated. They exercise as much code as possible in as many different ways as possible. While failure discovery may come in the form of the same sorts of test assertions as precision tests, typically failure discovery is a result of culling as many possible sources as available; processes that do not disappear, deltas between pre and post execution state, exceptions, errors and assertions in product logs, and variances in execution time. It is often the case that in-code test assertions are turned off or ignored in favor of sustaining as much on-going random activity as possible, with the intent to discover failures from post execution data analysis.

One example of a recall optimized test is an end to end test, or end user test. Imagine a test called “insert image and resize”. The steps may be as follows:

1. Boot application
2. Repeat following for every document in test library, every image in test library
3. Load document from test library
4. Scroll to location image is desired
5. Insert/Image, select test file from test library, OK
6. Select image in document, format/resize, enter new size, OK
7. Confirm:
    a. Image is expected size
    b. New document length is as expected
    c. Text still displays as expected
8. Close document

The test above is designed to happen along as many failures as possible on the way to inserting and resizing the image. Errors may come up relating to document or image handling for specific files. Errors may come up in terms of steps not exiting properly or unanticipated errors. Dialogs may take longer to dismiss than expected, causing errors in the automation navigation control. Virtually any unanticipated condition could trigger a failure, and that is the point, to report when something unexpected occurs. Were the above a precision optimized test the steps would be reduced to a single point of functionality – such as only “insert image” or “resize image”, skipping as many steps as possible.

Another example of a recall optimized test is one where the validation has been broadened. Imagine the same “insert image and resize” test as above, but this time, add the following validation:

a. Compare screen shot at final state against prior, report diffs as failure
b. Scan for processes still running after completion of entire sequence
c. Scan for windows open that were not open prior to the test starting
d. Execute in debug mode and capture any assert as a failure
e. Scan product logs for the string “error” or “unexpected” or “exception”

Such a list of possible failure states is going to capture a lot of interesting bugs, but is also likely to capture a lot of false positives. These are the sorts of errors that when product ships with them people ask “Didn’t anybody test this thing?” but when bugs are reported in bulk the same people ask “Why are we wasting our time on these bugs?”

Recall optimized tests operate in an intrinsically flaky world. There is a lot of noise in the signal. There are a lot of real failures that do not manifest every time.

Why we sometimes use the wrong sort of tests

Coverage pressure typically motivates engineers to use recall test suites in situations where high precision test suites are required. This is often a product of dysfunctional relationships between engineering functions or bad practices in release processes.

Regression control panic

For example, if engineers in development and test roles do not feel a strong obligation to each other’s needs, then test engineers may “fatten up” the build validation with recall optimized tests, leading to developers ignoring the results, leading to testers adding even more tests to the suite. Long release processes also tend to motivate addition of recall optimized test automation into the regression and build validation suites, as long times between builds adds greater risk of regression and greater pressure on test engineers to achieve coverage, the result being that the releases take even longer as the automation suite takes longer and longer to execute and results longer and longer to analyze.

Kitchen sink

Sometimes test engineers are instructed by developers to direct all automation at UI and end to end level to get all bugs from the top down. This is proposed as an efficiency technique, “get them all at once”. This doesn’t work as well as it sounds, as test fragility and flaky test results investigation costs overwhelm whatever gains were made by writing and executing fewer tests.

Protect against changing behavior

Sometimes test automation is written at the UI and end to end level to protect against having to change tests when lower components change behavior. Developers frequently want the freedom to change lower level code behavior, being accountable for the top level behaviors only. This is a fine distinction between unit/component/system tests, but it is a poor excuse for having no mid-level layer automated tests at all. The tradeoffs and risks are similar to the “Kitchen sink” case stated above.

The conclusion in this paper on this point is that whether automation is optimized for recall or precision is an important distinction to be made, and that it is best practice to keep tests well separated according to their purpose.

3.2 Simple Versus Complex Test Conditions

The complexity of the test condition has a huge impact on the consistency of a test.

3.2.1 Simple Test Conditions Yield Consistent Tests, Fewer Product Bugs

A simple test condition is one where the conditions of the test are both small in number and easy to control. Almost any variable that can change the behavior of the SUT is accessible to the test code. These sorts of tests are almost always unit tests, or very narrowly scoped component tests. Nearly every variable, such as environment, configuration, prior SUT state, integrated components or external systems are removed so long as they do not explicitly pertain to the exact test. These variables are introduced later as test conditions get more complex.

Under a simple test condition state, there is almost no reason why a test should ever yield different results from prior times it executes. Flaky tests for apparently simple problems at the unit test level are almost always the first sign of either bad test code, or that the product itself is written wrong (making simple problems non-simple). This is almost always the appropriate time to begin refactoring product code in order to simplify the test condition for sake of unit tests. This document does not go into detail on refactoring, but outstanding examples and methodologies can be found in the books “Refactoring”, “Working Effectively with Legacy Code” and “xUnit Design Patterns” referenced in the bibliography.

But such test code, once stabilized, tends to stay stable beyond the developer inner loop (within the inner loop, such tests yield high reward, rapidly catching regressions that the developer can often fix within seconds of discovery). They do not add much value catching large, complex bugs of the sort that have huge customer impact. They tend more to catch the kinds of bugs that would have immediately shown up during engineering – typically the first time a tester tried to use the code. The value of these tests is a safer, faster inner loop that forces developers to write better code (an almost inevitable side effect of refactoring code for testability).

3.2.2 Point to Point Integration: First Hint of Complexity, First Point of Flake

The first big unavoidable problem spots are the code boundaries between integration points, such as working with external objects, third party APIs or IO. Even when code is refactored, these integration points remain, just abstracted and componentized into isolated pieces of code, and there is almost no way to completely bring their behaviors under 100% control of the test. It is also the case that integration points are one of the most common sources of bugs. Here are some examples of integration factors that simultaneously cause bugs and drive flaky tests:

- Mistaken assumptions about exact behavior under inputs
- Unknown run-time states that may impact the behavior of integration points
- Mistaken assumptions about exception and failure condition contracts
- Inaccessible control points that affect behavior of integration point
- Asynchronous or multi-threaded operations that may alter behavior or drive race conditions
- Global/static state shared which may affect behavior

It is very common that all unit tests will pass splendidly, only to release code that fails because of something unexpected at an integration point. Lack of control of the test condition means the failure may not always reproduce when executed.

3.2.3 Complex Test Conditions Yield Inconsistent Tests, More Product Bugs

As test conditions get complex, inconsistency becomes inevitable. Particularly with end to end testing, aspects of the SUT and the fixture are beyond test control, and as such the test is unable to guarantee that the product, fixture or test will behave in the same way every time.

As we add other types of tests that fit complex test conditions, our list becomes expansive and broad. This is usually where one would introduce load tests, stress tests, configuration and environment tests, state based testing, fault mode and error handling tests. Each of these new test categories increases the number of variables under test, and decreases test fixture and test code control of the system state.

In this case, the complex test condition is not a disease to be avoided. The hardest, most difficult bugs in the system manifest in these complex test conditions only. The bugs are not the sort that are visible by code inspection, because they are almost always caused by a misunderstanding of the behavior of an external dependency. Perhaps error codes and exceptions are different than expected. Perhaps memory management and garbage collection behaviors of consumed classes are poorly understood. Such misunderstandings manifest in the code, but almost always only during execution in the end to end state, and usually under complex workloads.

4 What do We Know About Flaky Automation?

4.1 Stable end to end automation is feasible for most cases

There are lots of opportunity in fixing bad test code patterns.

There are a number of sloppy, bad test code patterns, which occur all too often and contribute to flaky and unreliable results. This document will not explore these patterns exhaustively (see Meszaros for an outstanding treatment of the topic), but a few common ones are:

- **Sleeps to wait out UI events**: change to either explicit check for UI objects, or inject an event-based construct in the code that tests can attach to
- **Timers to wait for asynchronous events**: similar to UI, launch something asynchronous and wait with a timer. It is better to poll status on an event, or create an event handler
- **Dirtying static and global state**: test fixtures commonly share state between tests such as test settings, application settings, database content, etc. Ideally the state can be crafted such that every test is completely independent from all others (for product state, this usually means refactoring), or the test code needs to be highly reliable with controlling state collision, initialization and cleanup
- **"Flying blind" assuming controls and state are ready without validation**: Often seen in combination with "Sleep and wait", test code all too frequently begins sending commands to controls or resuming progress without confirming the test fixture and SUT are ready to proceed. Sometimes this happens because an application may not provide appropriate hooks (e.g. one time I inherited a piece of automation where in comments was the phrase "Hope to God that the icon is in the upper left of the window" right before the code to initiate a click).

The first attempt at stabilizing automation will likely close large gaps through cleaning up bad test code alone. I have seen teams make as much as 20 percentage point gains in test reliability with this sort of effort.

Do less in the test

As discussed in the end to end test example prior, doing more offers more opportunity for failure. If the point of a test is to test one behavior, then the test code should do as much as it can to only exercise that one behavior. This is easy at unit and component test level, but harder with integration and system tests, especially UI. There are several ways to approach this:

> **Direct navigation**: Whether it is inside of an application or a web site, it is often possible to go directly to the point where the test executes rather than navigating through a long path of menus, pages, buttons, controls and dialogs. Web pages often allow direct access to some sequence via a URI. Applications may or may not have a means for automation to get directly to some state. In either case, it is sometimes necessary to change product code to allow such direct connections to avoid spurious failures on the path to a specific test.
> **API calls for initialization, UI for test**: One semi-direct path for a test is to call product APIs to initialize test state, and then use the UI and user level controls to perform the actual test.
> **Reduce surrounding content/state**

Shifting from out of proc to in-proc

Most end to end and system tests operate out of process. The test code runs in a completely different process than the SUT. The test code may be sending UI events to the application via a message queue, or it may be sending requests across a network connection, or perhaps dropping packages or files off into

work queues for processing. Whatever the means, out of process testing offers very little control of the test fixture for the test code is almost always very flaky.

Many times, teams will evaluate their tests and move many of them to in-process testing. This means that behaviors being tested must be exposed via an API that is directly callable by the test code and loads into the same process as the test code. This gives the test code more control over the fixture state. It also has the added benefit that call stacks at point of failure will contain whatever product code was active at that point in time alongside the test code.

The typical split is to take business logic related tests, the flow of the system from one state to another, and move them to in-process tests. They are separate from UI tests, which focus exclusively on whether or not the UI behaves correctly based on test conditions. This allows deeper, more in-depth coverage of complex business logic state without having to absorb the difficulty and flakiness of UI testing.

Shifting from end-to-end to component and unit tests

Similar to moving tests from out of process to in process, much value is gained by moving tests from end to end and system testing level to component and unit test level. Unit tests are stable, fast and reliable and leveraged appropriately can return much more power than the same tests performed end to end.

The trick is to pick the correct tests. Traditionally, there has been an artificial and dysfunctional separation of “developers do verification tests” and “testers do all the other tests.” But a large portion of the testing domain includes tests which are far more efficient and practical at the unit testing level, but were written as end to end or system tests purely because the tester did not have permission to add unit tests to the project.

My rule of thumb is “fat test domains should run as unit tests.” A fat test domain is one where the number of tests expands rapidly from combinations of test variables against single aspects of the behavior under test. Some examples:

- Special character handling
- Complex data parsing
- Complex state permutations
- Incorrect data
- Error and exception state handling
- International and localized data
- International environment settings
- Etc.

All of the above in a typical test strategy are paired up against different system inputs or variables and result in an extremely large number of tests. Almost all of these tests could be crafted as well written data-driven test engines that execute against low level product APIs. The difference in time to execute is huge (sub-second versus sometimes multi-minute) and reliability enormous.

Testing in Production

This document does not go into many details about testing in production, but moving tests from end to end automation into some sort of production system monitor and telemetry signal test is an often-used technique. Done correctly, and in a very intentionally well-planned manner, testing in production can yield faster bug discovery at lower cost. The service under test must meet certain criteria, though, before such practices are safe. See “Testing Services in Production” by Stobie for an excellent treatment of the subject.

It is also possible to do testing in product/end to end testing hybrids. Rather than deploy a simulated environment into a test laboratory, deploy the code under test into a slice of the production environment,

but isolated from customer workloads. From there, the test automation targets this slice, generating a synthetic workload against a real deployment. This practice is becoming more and more common as service deployment and update gets more and more streamlined.

4.2 Small quantities of inconsistent tests have a huge impact

It is common that 15-20% of tests remain flaky, even after efforts to stabilize them (Google reports 16% of tests are flaky, Micco). This creates a problem as pressure to release more often pushes teams to execute more tests more often. Therefore, the faster and more often you release, the more of an impact inconsistent tests have on the team. The more tests are executed by average engineer, the higher the average test consistency is needed to keep the team efficient. The more frequently any given test is executed, the more reliable it needs to be to avoid exposing somebody to a flaky result.

Consider a test with a 99.9% constancy rate executed 1000 times a day. On average, it is going to yield a false signal every day. Consider a test suite of 1000 tests, where the average test consistency rate is 99.9%. On average, that test suite is going to yield a false signal on every execution.

Example:

The average Office engineer executes ~300 automated tests prior to every code submission. This means that average test consistency rate needs to be better than 99.7% in order to avoid every single job yielding a failure.

4.3 More bugs fixed, but lower fix rate come from inconsistent tests

Despite our frustrations with flaky tests, they yield real product bugs, and removal of tests that that exhibit flaky results represent real risk against product quality. The following is gained from an analysis of test automation failure triaging inside Office. Results are likely to vary a great deal in other teams, companies or environments.

One analysis we did was to compare total product bugs found by test automation and fixed in product against only those that came from automated tests that were stable (60% of tests at time of analysis). The end result yielded a 27.5% reduction in product bugs fixed. In other words, those bugs would have escaped past the test automation phase and into the hands of end users:

Total Bugs (test and product bugs)	331
Total Product Fixed Bugs	120
Total Bugs Remaining After Removing Flaky Scenarios	220 (66.4%)
Total Product Fix Bugs Remaining After Removing Flaky Scenarios	99 (82.5%)
Total Scenarios	1278
Remaining Scenarios (after removing)	779 (60%)

The above analysis was performed before Office had started execution of reliability runs, so there was little known about the behavior of tests over large numbers of iterations against the same build. Afterward, many practices had changed in terms of bug triaging practices and automation failure turnaround. Later analysis showed even more interesting trends:

Reliability Rate	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	Total Bugs
Bugs Fixed	**12**	**5**	**11**	**14**	**15**	**26**	**14**	**53**	**40**	**48**	**11**	**249**
Percent of Fixed Bugs	5%	2%	4%	6%	6%	10%	6%	21%	16%	19%	4%	

Tests that were 100% reliable accounted for only 4% of the bugs fixed overall (not distinguishing between product and all bugs). There was an obvious trend toward mostly fixing bugs where reliability was between and 60-99%, but still a non-trivial collection of fixed bugs remained.

This higher fix quantity comes at the price of much, much higher noise.

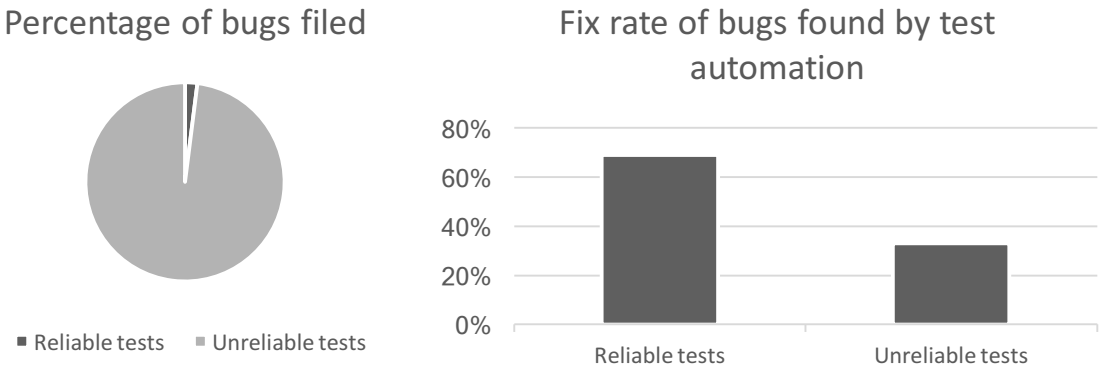

Figure 2 Reliability of Tests and Bug Fixing

As shown in the chart above, for the same set of tests, bugs found by non-reliable tests accounted for 98% of the bugs reported, but only 33% of those bugs were fixed. Meanwhile, reliable tests accounted for just 2% of the bugs reported, but 69% of those bugs were fixed. It is clear that unreliable tests were yielding far more bugs than reliable tests, but did so at a much lower rate of fixing.

5 What to Do with the Flaky Automation

Once test automation reliability is understood and measured, you need to do the right thing with it. The recommendation this document makes is to separate it into two suites. A gating control suite and a bug discovery suite.

5.1 Divide Execution into Gating Control and Bug Discovery

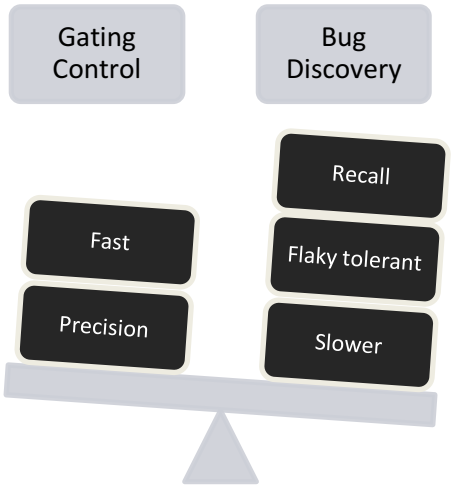

Figure 3 Gating Control Versus Bug Discovery

Gating Control is any test suite that is used to control release of code to a next step in the engineering process. Such steps may be: engineer submits code to repository, branch integrates to parent branch, main build is released from a branch to engineering team, product code is released to end users. Failures on gating control slow down the release, creating expensive delays. Some gating control suites, such as submission to repository are executed at very high frequently, and others, such as product release to end-users are executed at a lower frequency. The primary purpose of gating control is criteria evaluation and regression detection. Gating control requires precision tests.

Bug Discovery is any test suite that is used to discover bugs in the product. Such a test suite does not have to gate release into any particular phase of the release process. Failures in bug discovery should generate work in the product backlog to analyze results and fix product bugs. The primary purpose of bug discovery is to discover as many bugs as possible. Bug discovery requires recall-optimized tests.

Example:

Office runs several types of automation test jobs:

Developer Jobs (gating control)
Automation run by developer prior to code submission to repository. Tests are a subset of the product BVT tests (determined by what code was changed on submission). Developers are expected to address failures prior to code submission. Hundreds of jobs per day.

Team Branch Looping Jobs (gating control)
Regular jobs against current code in team branch of repository. Tests are typically same as product BVT (Build Verification Tests). Failures may result in a back-out of any submissions since previous job. Dozens of jobs per day.

Integration Jobs (gating control)
Jobs run prior to push of product team branch into parent (main) branch. Tests are typically same as product BVT tests. Failures are addressed prior to integration. One job every 1-5 days.

Main Build Validation Test (BVT) Jobs (gating control)
Jobs run daily against integrated main branch. Output is official drop of Office.

Failures in suite set individual product team “zones” into read-only mode until the failures are addressed – i.e. integrations and submissions to that code are prevented. One job per day.

Main Build Code Validation Test (CVT) (bug discovery)

Jobs run daily against integrated main branch. Failures are recorded as bugs, but do not gate any processes at all.

5.2 Create a Reliability Measurement Run

The tests used for both gating control and bug discovery need a reliability calculation. This means the tests must execute several times, the more times the better. How many times is determined by the amount of reliability precision you need to measure. In order to assert a 99% test reliability at least 100 executions of the test under the same conditions are required. Likewise, to assert 99.9% reliability, at least 1000 executions are required.

This many executions may not be practical or cost effective. It depends on the capabilities of the automation system and on the execution times of the tests. You may be able to accept a compromise. For example, if you can only manage 20 executions of a suite per build (which can assert, at best, an 1/20=5% failure precision) you could achieve up to 1% failure precision by aggregating the reliability of tests across 5 builds. The measurement is somewhat inaccurate, as tests where reliability got either better or worse inside the span of 5 builds will be obscured, but this may be a manageable inaccuracy. The measurement is not so much about extremely high precision every build as it is about generating a number capable of managing a suite for efficiency over larger periods of time.

Example from Office: Reliability Run:

A background process picks up both the BVT and CVT test suites and executes them repeatedly against whatever most recent build completed the BVT run. During the week, where builds come out daily and there is less pressure on the automation system, 100-200 executions of the entire suite execute per build. Failures are automatically identified and checked against known failures, and new bugs automatically filed for any new failures. Existing bugs are updated with failure hit statistics. Reports show teams reliability statistics for every test in the suite.

Scenario ID	Scenario Name	Config	Scenario Path	Consistency Rate	Pass Rate	Executions
#Scenario: 1117						
211367	ATL_WordUnmanagedSimpleCustomTaskPane	client64-win10d-next	OfficeVSO\OC\Word\Core Engineering\Programmability	0.00%	19.64%	532
155518	Backwards Compatibility - Save Docx as Doc	client64-win10d-next	OfficeVSO\OC\Word\File IO\Format\OpenXML	0.00%	30.18%	532
137638	Roundtrip VB Project	client64-win10d-next	OfficeVSO\OC\Word\Core Engineering\Programmability	0.00%	31.96%	532
195503	InsertOCXviaRibbon	client64-c2r-win81	OfficeVSO\OC\Word\Client Services\Document Content\OCX	0.00%	42.77%	556
195503	InsertOCXviaRibbon	client64-c2r-win10d	OfficeVSO\OC\Word\Client Services\Document Content\OCX	0.00%	46.59%	557
183633	StylesPaneUI	client64-c2r-win10d	OfficeVSO\OC\Word\Client Services\Formatting\Styles	0.00%	67.12%	554
204326	Acceptance: Insert Envelopes	client64-c2r-win81	OfficeVSO\OC\Word\File IO\Mail Merge	12.50%	71.49%	554
379635	InProcAutomation : Client : Rtc : CVT	client64-c2r-win10d	OfficeVSO\OC\Word\Collaboration\Coauthoring\RTT	0.00%	75.00%	4
174440	OpenComplexFile_SaveAsDocm	client64-c2r-win10d	OfficeVSO\OC\Word\File IO\Open-Save	12.50%	77.31%	554
177535	OpenComplexFile_SaveAsHtm	client64-c2r-win10d	OfficeVSO\OC\Word\File IO\Open-Save	12.50%	77.35%	554
397308	Word OCS - Bluechicken E2E (IL+OL)	wac-dc	OfficeVSO\OC\Word\Collaboration\OCS Integration\Server-side	0.00%	79.47%	1683
380428	InProcAutomation : Client : EDP : CVT	client32-c2r-win10d	OfficeVSO\OC\Word\EDP	12.50%	88.98%	554
141434	OpenComplexFile_SaveAsDoc	client64-c2r-win10d	OfficeVSO\OC\Word\File IO\Open-Save	0.00%	89.20%	560
311152	BootOpenTypeText	client64-win10d-uni-next	OfficeVSO\OC\Word\File IO	14.29%	89.66%	536

Figure 4 Office test automation reliability report.

5.3 Move Automation into Gating Control Based on Reliability

Once the suites are separated into gating runs and bug discovery runs, it is time to manage the suites based on reliability numbers. Here are the principles to keep in mind

- Gates demand reliable “A fail is always a fail” signal – hence high reliability
- More frequently the gate is tested, the higher the reliability required
- More frequently the gate is tested, the fewer tests should be in the suite
- If there is no gate, then frequency of execution has no penalty

Pick a reliability threshold for each gating suite. Pick something pretty high, but not so high as to be impossible. If 50% of the suite is at 80% reliability and below, then you might need to compromise with 80% for a short period of time to give teams the chance to address reliability issues. Or maybe that 50% of tests does not need to gate releases. This is a balance you will need to gauge on your own.

Once you set your threshold, begin moving and removing tests. If you trust your threshold, consider doing so via an automated process. Anticipate a lot of back and forth conversation about whether a test should be just moved or removed completely. Anticipate a lot of anxiety over tests that cover critical functionality, but which do not meet the bar. Engineers will want exceptions to the threshold. You get to decide how rigid you want to be. Experience in Office has shown that a period of getting used to the idea works well before dropping a firm “no exceptions” hammer.

Ultimately, the threshold that works best is one that balances coverage goals and how much time you can afford delaying releases for spurious failure investigation. If at least one flaky failure manifests on every single job, dozens to hundreds of times a day, then perhaps moving to the bar to only happening one out of four jobs, or one out of ten jobs is a huge gain.

Coverage is similar. A brutal, numbers only movement of tests to later and later gates and into bug discovery risks a bug discovery later than you want. Maybe a customer sees a bug before your automation finds it. Or maybe an official build is released to engineers that breaks a feature from some other team. This is inevitable and the best way to approach it is to individually pull back tests where the cost of that escape is unacceptable. This means that whatever test it is needs to be stabilized before moving to the higher gate. Either that or accept the escape risk as a balance against faster releases.

Example:

BVT Test Demotion: Office runs an automatic daily process that examines the reliability results of all BVT tests for the last 7 builds. Any test that does not sustain an aggregate 95% reliability over 7 builds is automatically moved from the BVT suite into the CVT suite. Beyond automatic demotion, individual teams use the reliability report data to push their BVT test reliability as far as 99% and higher. Current (as of July 27, 2016) overall Office BVT test reliability rate is > 98%, often 99% or higher.

This same push has also substantially improved the developer experience. Previously, every automation job launched by a developer to evaluate code for check in had at least 1 failure, 100% of all jobs failed. Engineers would either execute twice (and hit a different failure) or scan prior results to see if their failure was known previously. The current rate of failed jobs is 60% - a full 40 percentage point improvement in passing job rate. It is well worth noting that the average number of tests for passing jobs is 85, whereas the average number of tests for failing jobs is 801, confirming positions elsewhere in this paper regarding the impact of test suite size on reliability demands.

This notion of curating automation suites based on test reliability is not isolated to Office or Microsoft. A similar practice as stated in the example happens at Google (Micco).

5.4 Move Flaky Automation into Bug Discovery

Any test that does not meet the reliability criteria for gating processes should be used for bug discovery. This permits the execution of tests where stabilization is either unrealistic, or would render the test too sanitized to effectively discover new bugs.

Working with flaky automation requires a different approach, both in terms of how the test is executed and in how the results are processed.

Change in Execution Style

Almost all tests contain some sort of pass or fail assertion inside the test method. But at this point it is best to modify that approach. Look for more signals of potential failure. Some approaches might be looking for processes that are left open after test completion, windows or dialogs that are not dismissed, files left open, asserts thrown by the application. Application log files and event logs should be mined for exceptions and error content. System resource usage should be monitored for leaked memory, resources, over-active file system access. Latency and throughput checked for diminished performance.

Some forms of testing remove formal validation completely and just focus on applying load. This has been a long time common practice for performance, scale, load and volume tests. It is not too uncommon with “monkey” tests, applying stochastic load patterns to drive the application to error prone states. Rather than trying to increase the ability of the automation code to successfully maintain the complex state of the test fixture, reduce the amount the automation code cares about. Just keep the SUT moving.

Change in Analysis

Recall optimized tests demand more analysis and time than relying on a report of which tests passed and which tests failed. The widened definition of potential failure and the lower validation of load generates a lot of false signals. These signals largely demand a significant investment in human time and effort, which is why the activity is used to feed the product backlog with work rather than gate the release processes.

The challenge, then, is to sort through the volume. The most common techniques are:

Failures that have never been seen before

The easiest way to address failures is to tackle everything that is new, but if the rate of intermittent failure is high enough, or the suite large enough, this becomes an overwhelming problem. Teams that start here, before cleaning up their product and test suite, usually abandon it for one of other methods mentioned here.

Teams also tend to ignore failures that appear in a run when that failure has been seen previously. The justification for this is that whatever change was submitted is not likely the cause of the failure, hence the engineer should proceed with code submission. This makes sense when focusing purely on whether or not a specific change has caused a regression, but extensive use of the practice creates a large pile of engineering debt and a general distrust of the test automation.

This was the first technique the Office team used to get a handle on its intermittency, and the impact on test system distrust was dramatic. Over time it came to a point where every automation job used to gate a check-in had some sort of failure in it, and engineers would either ignore them, or run the job twice, submitting code if the failure did not happen both times. The overall passing rate of the suite diminished over time until it was a largely ineffective way to evaluate a build. Eventually, the Office team followed the suite splitting practices advised in this document and adopted a strict “100% pass” policy on code submission and build validation jobs. The result was a substantial improvement in the engineer code submission job passing.

To use this technique, the automation system must be able to identify failures as pre-existing or new. Automated tests may fail for many different reasons. The Office team uses a mechanism that utilizes a combination of string matching templates, call stack frame matching and comparing edit distance between failure messages to determine if a failure is new or already known (Robinson). Once a given failure is known and identified, the automation system can track when it was first seen, what builds it has been seen in and how often it occurs.

Failures that occur more often than others

This is typically the first, and easiest priority decision with any failure. More frequent failures are easier to diagnose and easier to validate when fixed. But it is also clear that a failure which happens more often is going to cost more overall than a failure that is rare, all other aspects such as severity and scope being equal. Frequency therefore usually, although not always, works as a good rule of thumb for priority.

Failures that are likely to also occur inside gated processes

It is advised to include gating tests inside the bug discovery suite. In addition to helping compare results between tests, it is also useful for ferreting out lower occurring intermittent failures in order to fix them in the gating tests. It is also useful to compare results of tests in one context or another. Test automation environments are not always completely clean, and factors like system load, resource availability and machine re-use may cause tests to change their intermittent failure. Regardless, tests that gate processes have high priority for fixing intermittent issues.

Failures that occur inside code paths that are known to be problematic for customers

This requires more insight about how the product is used and where customers are having difficulty. If a feature is moderate to high priority, then fixing intermittent failures in that feature will have a larger return on investment than on features that are almost never used.
Don't fall into the trap at only looking the highest priority features, those that fit into marketing team demos. Usage patterns on any product or service tend to have a long, fat tail, which means that there is a very large quantity of behaviors that range from low to moderate usage, and any failure hitting that range is going to have a big cost impact on the product.

Failures that suddenly change their rate of occurrence between builds

This technique acknowledges the system experiences intermittent failures, and uses sudden large increases in the rate of occurrence indicate a change in the system for the worse. We have seen teams inside of Office use this as a mechanism to draw team attention to failures they might have ignored otherwise. This also a failure investigation pattern that has been reported by companies such as Google (Micco).

Failures that suddenly appear "new" against the same build

I discovered an instance of this while collecting charts for this paper, and realized it was an important detail to examine. When a test suite is executed multiple times on the same build of the product the new failure discovery rate is expected to start high, and then drop off progressively with each iteration. There will be some degree of variance around the slope away from a projected trend line, but that variance should either be single spikes or not much larger than the average variance.
When there is either a series of spikes in "new issues" around the same sequence in time, or when the variance is much larger than the average new failure count discovered per iteration, then that is a clue that something systemic is affecting the test runs.

The chart below is taken from the Office test automation reliability run, all for the same build on July 24, 2016. The suite contains ~21k tests, mostly end to end. The horizontal axis shows progressive iterations of the suite; the vertical axis shows the number of new failures reported per each iteration. The slope follows a diminishing number of discoveries over time, as expected, but there are several "new issue" spikes that vary a great deal from the trend line that suggest something was affecting the suite behavior overall. Such information offers clues as to the source of intermittency.

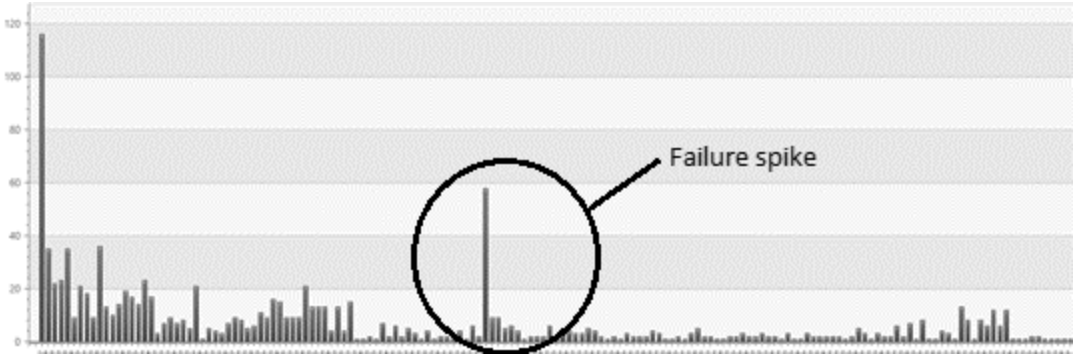

Figure 5 Office test automation reliability suite, July 24 2016

Projecting "When will these tests stop discovering bugs":

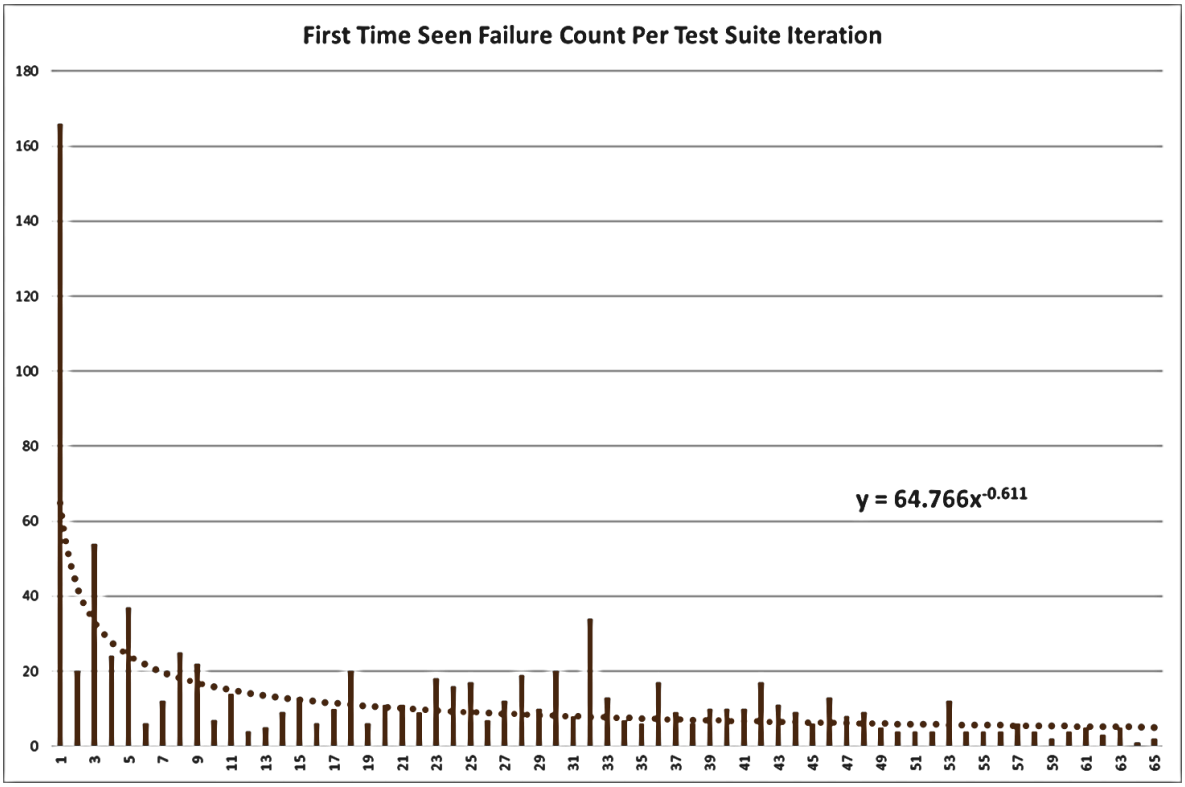

Figure 6 Office test automation reliability suite, new failures from 7/19/2016

The fall off rate of new failures discovered per iteration tends to be a power curve, where the area under that curve is the total number of new failures it is possible to discover with that test suite on that build. While the value of new failures may be slowly approaching zero for a long time, it is possible to arbitrarily set a threshold (e.g. “less than 1 new failures discovered per run”) and project how many iterations are needed to stop discovering failures. In the above example, that formula would be:

Fall off rate = $64.766\ x^{-0.611}$

Iterations at < 1 failure per run = $\frac{1}{64.766} = x^{-0.611} = \left(\frac{1}{64.766}\right)^{\frac{1}{-0.611}} = \sim 921\ iterations$

Likewise, total number of bugs discovered at that point is an integral of the equation bound by X= 1 and X= *921*. That formula looks like:

$\int_1^{921} 64.766\ x^{-0.611} dx = \sim 2202\ discovered\ failures$

From this, you can extrapolate how many iterations are needed to see any given percentage of the total bugs the suite will find. The progression is non-linear, so a substantial percentage of the total bugs is discovered with few iterations, with diminishing returns for each iteration. The values from the example are as follows:

1102 (50%) = 185 iterations

1652 (75%) = 467 iterations

1981 (90%) = 716 iterations

Doing the above gives a very powerful assessment of the discovery value of a suite. You can balance the value of extra iterations against the value of the bugs found with each iteration. You can also use the number of iterations required to hit your threshold as a quality statement about the intermittency. In this example, 921 iterations would require very large numbers of machines. We can find about 50% of the bugs the entire suite will find in 185 iterations, but even that is a substantial investment. A goal, then, would be to reduce that number so that all the failures the suite is found are discovered more quickly – in fewer iterations.

6 Conclusion

While flaky test automation presents many challenges and difficulties, there are practical ways to address those challenges. Sometimes the flake is a sign of bad test code, sometimes it is signal that there is a lot of bugs to be found with the test. Measure the automation for reliability and consistent results, and then use that information to either target fixes against the automation and product, or separate the automation into precise/reliable tests that protect phase gates in the engineering process or recall optimized less reliable tests that find bugs in large quantity. Different analytical approaches and practices can help any team get their test automation under control.

References

Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, Erich Gamma, 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional

Michael Feathers, 2004. *Working Effectively With Legacy Code.* Prentice Hall

Gerard Meszaros, 2007. *xUnit Design Patterns; Refactoring Test Code.* Addison-Wesley

David Scherfgen, Online Integral Calculator, http://www.integral-calculator.com/ (accessed July 26, 2016)

Robinson, M. P. Test failure bucketing, U.S. Patent 8,782,609 July 15, 2014

John Micco, Flaky Tests at Google and How We Mitigate Them, http://googletesting.blogspot.com/2016/05/flaky-tests-at-google-and-how-we.html, blog entry May 27, 2016

Keith Stobie, Testing Services in Production, PNSQC paper 2011, http://www.uploads.pnsqc.org/2011/papers/T-11_Stobie_paper.pdf (accessed July 7, 2016)