**Basics**

**QA>** How do We deal with flaky tests?

**Ans**. A test suite-a collection of tests that check for bugs that were introduced into our code – is a crucial part of our CI/CD and when our test fails or succeed at seemingly random intervals without any code change then we consider our test as flaky tests.

Here are seven highly effective strategies to fix flaky tests.

1. Visualizing Test Runs

While test-run visualizations alone can give Wean idea of how well our tests work, (in combination with multiple randomized test runs) we will also get a clear depiction of whether flaky tests are increasing or decreasing over time. A simple table with rows showing time and columns showing tests can be enough for this.

2. Quarantining Flaky Tests

Once you’ve found our flaky tests, we should create a separate test suite for them to serve as a quarantine. Our non-flaky tests don’t have to be run multiple times, so creating an extra test suite will save we from duplicating part of the work. Google has even created a tool to help with this by automatically putting flaky tests in a separate test suite.

This practice will also help when fixing the flaky tests because it allows, we to focus on the flaky tests independently. If isolating the tests in their own suite fixes the flakiness, that change alone will give We an idea if inter-test dependence is the reason for the flakiness.

3. Cleaning up State

Remove all state and data generated before a test run, so our test can’t be derailed by existing data we forgot about. This state can live in caches, databases or even variables. We will also want to check that our tests clean up correctly after they’re done — clean-up errors are often silently ignored in test suites. In a worst-case scenario, we will need to rebuild the whole system for every test run.

For databases, it can be helpful to use transactions. These can be rolled back after a test run, bringing the database back to the state it was in before the test was started.

4. Looking for Timeouts

Asynchronous tests that access network resources are especially prone to flake due to timeouts. The network can be quick or slow depending on the number of services using it. A too-short timeout can cause a test to flake. Setting our timeout variables in bulk will allow us to change them quickly in the future.

If We have a complex test relying on asynchronous services, try to check the service for availability before starting the test. This will save time when our timeouts become too long.

5. Using Test Doubles

We can create a simplified version if We test a service that isn’t deterministic. A common critique of this practice is that test doubles don’t always accurately mimic the actual service. By ensuring that the test double doesn’t deviate from the original, we can account for updates. Writing contract tests can help mitigate this problem.

6. Checking the System Clock

If our code depends on data that can’t be known in advance, such as the system clock, wrap these data sources in our code and don’t rely on them directly. This will allow us to replace their outputs with hard-coded data before running a test.

7. Checking for Memory Leaks

Profile our test code to get a feeling for its memory usage over time. If our code has memory leaks, we will see our test suite’s memory usage grow with every run test. Depending on the available resources and other systems running on that hardware, a memory leak could very well be the source of our flakiness problems.

**QB>** Let's suppose there is a test pipeline taking about 1 hour to finish, what would We do to decrease the time of it?

**Ans**. We can decrease the time of a test pipeline which is taking long time to complete by using below methods–

* We should verify that there is no flaky test in our suite before running the pipeline
* Look for long running tasks and examine the constraints
* Look for faster network
* Certain scripts taking a long time, optimise those test scripts.

**QC>** Imagine We have the possibility to ask software engineers to develop tools for We that will increase our productivity as full-stack QA, please describe them our requirements.

**Ans**. I could like him to design a testing tool which will have 3 different workspaces for –

* API Automation
* UI Automation
* Performance Testing

Which will provide different features give below –

* Different test execution report for UI, API & Performance.
* Different language support like (Java, C# & Python)
* Different plugin and framework support.