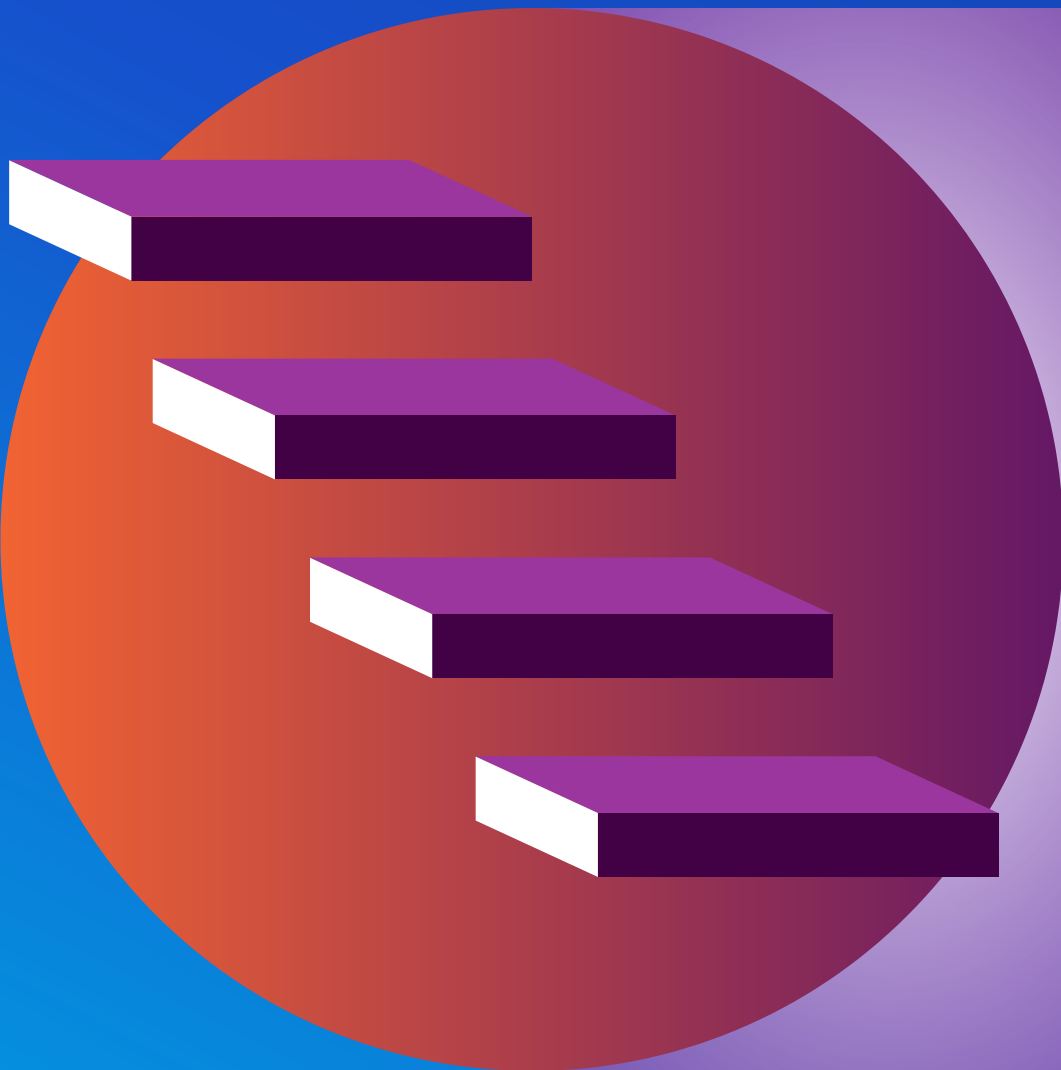


A Complete Guide to Optimizing Slow Tests



A Complete Guide to Optimizing Slow Tests

semaphoreci.com

Professional software development is a feedback-based process — each new iteration is informed by past results. Feedback is powered to a considerable degree by tests.

When tests slow development down, engineering teams lose momentum and become frustrated, because they can't meet their goals. A slow test suite puts the brakes on CI/CD, making release and deployment more difficult. This often means that organizations can't ship out products on time, and risk losing their competitive edge.

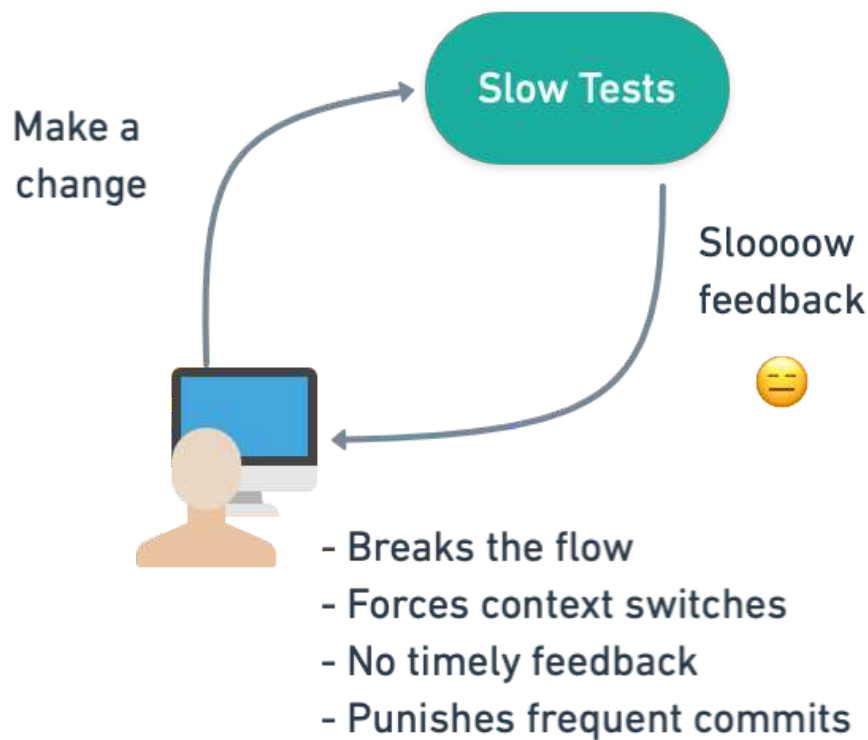
Choosing a scalable cloud platform like Semaphore is a great start. Semaphore offers some features that are helpful in dealing with slow tests, which we will discuss later in the article.

What's wrong with waiting for tests?

Tolerating a slow test suite is like making the minimum credit card payment when you could pay off your balance: by not dealing with it now you have a bit more cash in the short term, but will have to pay much more down the road. It doesn't make any sense, but people do it because the costs are not immediately obvious. When faced with slow tests, developers typically respond in one of three ways:

- Do something else and pay the cognitive cost for the context switch.
- Wait for results and lose focus on the problem at hand.
- Trudge on blindly without feedback.

Whatever happens, development speed falters due to the lack of timely feedback.



Fortunately, we have a battle-tested plan that makes identifying and fixing slow tests much easier.

The complete guide for making your slow tests fast

This guide consists of two parts:

- **Part 1** lays down a framework to identify, prioritize and optimize the slow tests in your suite.
- **Part 2** deals with the most common sources of slow test performance and their solutions.

A framework for making slow tests fast

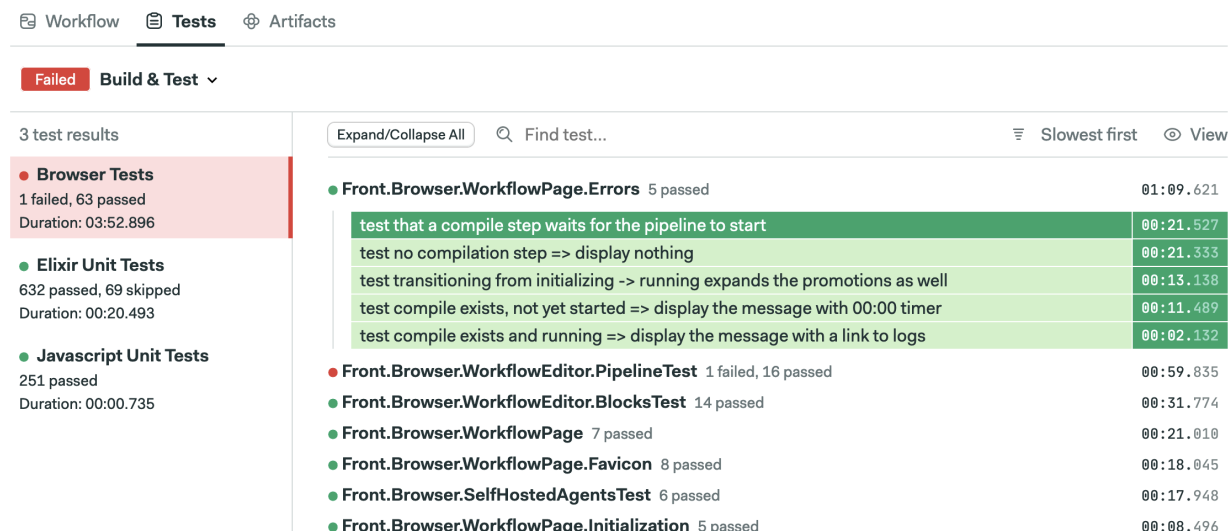
Dealing with slow tests requires both a concerted effort and a sound plan: 1. **Identify**: which tests are bad performers. 2. **Prioritize**: pick a batch of the slowest tests. See if there are some outliers that could be easy to fix. 3. **Profile**: zoom in and capture metrics to find out what your tests are doing behind the scenes. 4. **Optimize**: make the tests snappy. 5. **Repeat**: go back to Step 1 and repeat the process until you test suite is in top shape and your team is ☑

Let's be clear. This is not a one-off endeavor. It is part of the lifecycle of the project. Over time, tests slow down as the codebase grows and more tests are added. Therefore, you'll need to repeat the whole process **at least once per quarter** to be in good shape.

Step 1 — Identify high-value candidates

It can be hard to find the slowest tests when you have an extended CI/CD pipeline. Luckily, Semaphore supports [Test Reports](#), which provide an effective and consistent view of your test suite in a CI/CD workflow.

There's a little bit of setup required: you need to configure the test's output to the [JUnit format](#), as well as add a few commands. The result is, however, well worth the effort. In the detailed dashboard, you can spot problems, filter skipped tests, or order them by duration.



The screenshot shows the Semaphore Test Reports dashboard. At the top, there are tabs for 'Workflow', 'Tests', and 'Artifacts'. Below the tabs, there's a 'Failed' button and a 'Build & Test' dropdown. The main content area shows '3 test results'. On the left, there's a summary for 'Browser Tests' (1 failed, 63 passed, Duration: 03:52.896). Below this, there are sections for 'Elixir Unit Tests' (632 passed, 69 skipped, Duration: 00:20.493) and 'Javascript Unit Tests' (251 passed, Duration: 00:00.735). The main list of tests is sorted by 'Slowest first'. The tests listed are:

- Front.Browser.WorkflowPage.Errors (5 passed, 01:09.621)
 - test that a compile step waits for the pipeline to start (00:21.527)
 - test no compilation step => display nothing (00:21.333)
 - test transitioning from initializing -> running expands the promotions as well (00:13.138)
 - test compile exists, not yet started => display the message with 00:00 timer (00:11.489)
 - test compile exists and running => display the message with a link to logs (00:02.132)
- Front.Browser.WorkflowEditor.PipelineTest (1 failed, 16 passed, 00:59.835)
- Front.Browser.WorkflowEditor.BlocksTest (14 passed, 00:31.774)
- Front.Browser.WorkflowPage (7 passed, 00:21.010)
- Front.Browser.WorkflowPage.Favicon (8 passed, 00:18.045)
- Front.Browser.SelfHostedAgentsTest (6 passed, 00:17.948)
- Front.Browser.WorkflowPage.Initialization (5 passed, 00:08.496)

Once you have a list of slow candidates to work on, you're ready for the next step.

Step 2 — Maximize optimization effort vs benefit

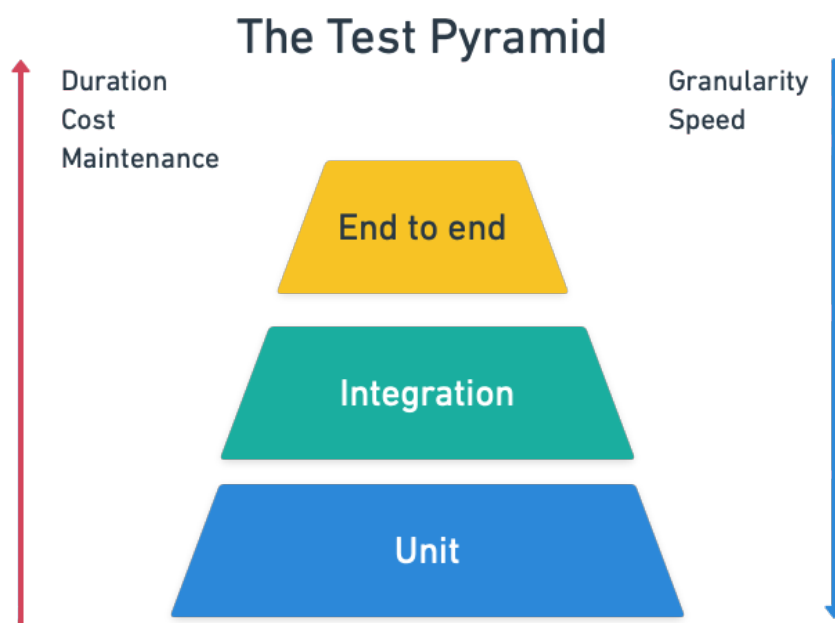
Two factors come into play for deciding where to start: how much faster you can make a test and how long you need to optimize it. We're going to grab the low-hanging fruit first.

In other words, we want to start working on tests that maximize:

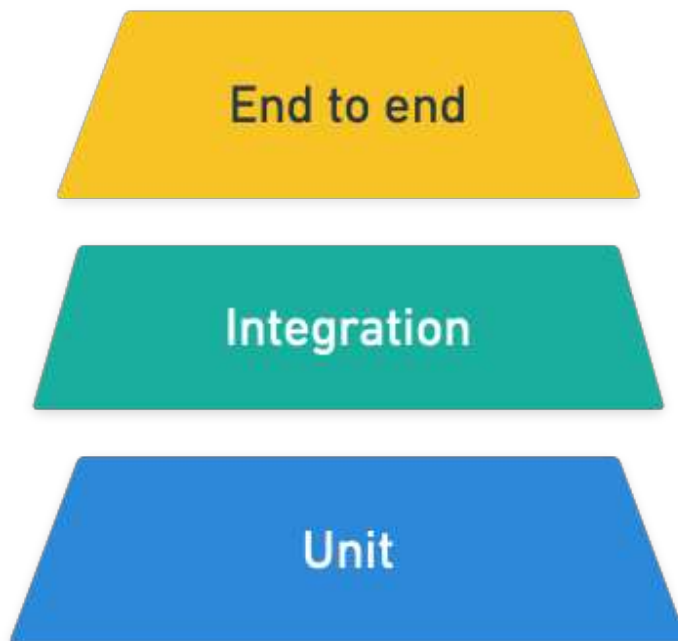
$$\frac{\text{test runtime before} - \text{test runtime after}}{\text{effort}}$$

The trouble is that the only certainty we have at this point is how long the test takes. Everything else that we have is an estimation. Consider starting with a few easy-to-fix tests or deleting ones that do not add value, even if there are slower candidates in your suite. Once you have a good grasp of the process, you can go after slower tests that require more substantial effort to optimize.

The testing pyramid can guide us here. The width of each level reflects the suggested ratio of tests for each type relative to the whole suite.



The pyramid tells us that a good test suite should have many unit tests, some integration tests, and a few end-to-end or acceptance tests. In contrast, slow suites tend to be more top-level heavy, i.e. the opposite of what they should look like.



The way forward lies in cutting the fat at the top, either by deleting some tests or moving them downwards.

Maybe an example can help at this point. Imagine that we want to write an [acceptance test](#) for an online music service:

Feature: Control playback

Scenario: play a song
Given there is no song playing
When user presses the play button
Then the song should start playing

Scenario: pause a song
Given a song is playing
When user presses the play button
Then the song should be paused

It's a valuable test that checks a business-critical feature. You may be able to squeeze some extra seconds of runtime but you can't ever delete it.

At the other extreme, we have this:

Feature: Search for music

Scenario: search song cannot have an emoji symbol
Given the search box is selected
When user types an emoji

Download the full guide

We hope you have enjoyed this small sample of the guide.

Download the the full guide for free here:

<https://semaphoreci.com/resources/complete-guide-to-optimizing-slow-tests>