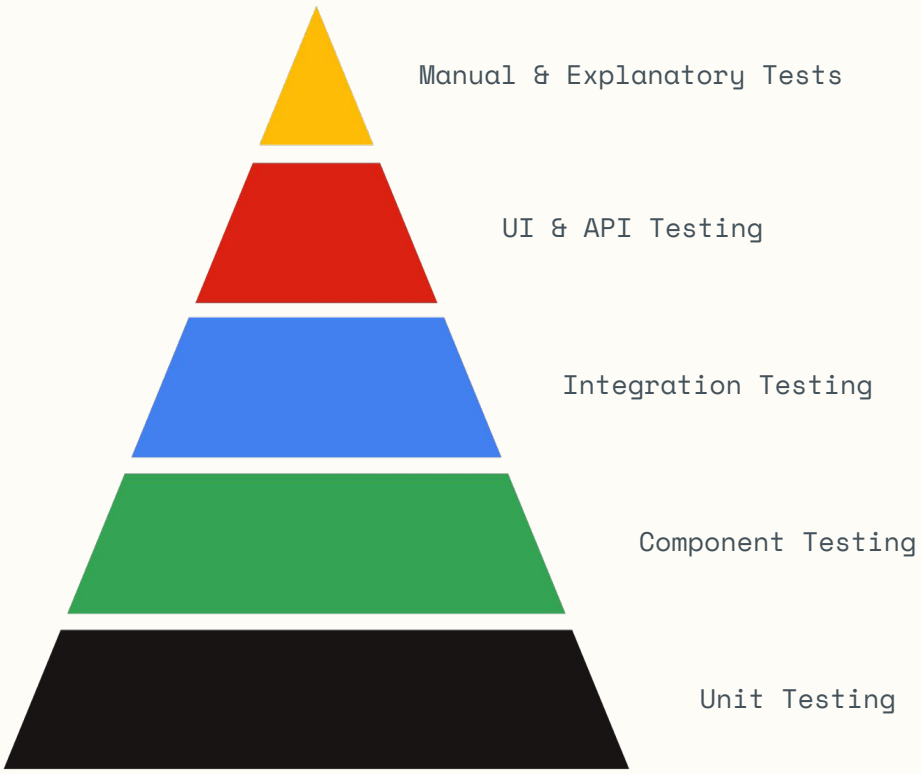
OnPath

Your QA tester's hierarchy of needs.

What is the agile testing pyramid?

The **agile testing pyramid** helps testers understand how to implement automated testing. This valuable guide was originated by agile expert, **Mike Cohn.**

The pyramid provides a starting place for new QA testers to jump off from, and it gives a 'single source of truth' for established QA testers to refer back to. In doing so, it meets a range of **needs.** This is what we're talking about:



A quick disclaimer about the agile testing pyramid

First off, yes, there are many other ways of visualizing this process. A diagram that illustrates something this complex or abstract is never going to tell the whole story. Over the years the number of tiers, arrows, asterisks and thought bubbles added to this diagram could make your head spin. We're keeping the visual simple, and will unpack the nuance here.

It's all about ROI

> "It costs almost five times more to fix a coding defect once the system is released that it does to fix it in unit testing."

— Paul Grossman, IT consultant, in **'Automated testing ROI: fact or fiction?'**
The two categories at the bottom of the pyramid — unit and component testing — represent those types of testing that benefit most from **automation**.

Why automate testing?

As your testing process matures, you may start to recognize opportunities for:

- More test coverage at the same cost
- Increasingly efficient test cycles
- Testing that scales and can be used across projects
- A lower price tag on fixing bugs in development
- Fewer defects that make it through to production

These benefits are much more evident at the base of the pyramid, so if you're considering **automating your testing**, that's where to start. Why? Because **Unit testing** focuses on individual pieces of code — the smaller the better. This enables a detailed view of code quality and performance. And because these units require limited functionality to run, you get feedback fast. As such, it makes sense to automate these tests. Furthermore, automated unit testing makes it easier to pinpoint where the problems are for fast remediation.

Component testing comes after the unit testing and tests objects independently, such as modules, classes, and programs. These tests are more thorough than unit

testing and often need access to a database or other components, so are slower than unit tests. That said, because you can define inputs and expected outputs, you can easily create automated tests without involving the user interface (UI).

Automating unit and component tests makes the testing process faster, more predictable, and repeatable. But then comes integration testing. And that's where the question of automation becomes a little more complicated.

What is integration testing?

Integration testing combines the individual software modules and tests them together as a group. It verifies whether or not the modules and services that make up your software work well together while exposing any defects in how they interact. This is especially important when different developers have worked on different components.

How much ground an integration test covers will impact how long the test takes. And the longer the test, the more disruptive it will be to shorter running tests, such as unit and component tests. There are a number of ways around this:

- Make integration tests modular and testing only what you want for a given code check-in/ release.
- Have multiple test environments available, or even better, have DevOps script this process so you can spin a new environment up as needed.
- Arrange the SDLC process so the developers are performing unit testing locally before check-in, and QA engineers are integration testing in QA or elsewhere.

You must consider your process, environment, and testing needs carefully, and create a strategy that gives you the best in development efficiency, test coverage, and resource utilization. In the end, properly planned and executed integration automation will supercharge your testing efforts and deliver repeatable results with less instance of human error.

There's no definitive answer to exactly when you should stop automating your tests. But as you move up the hierarchy, the return on investment is less (and naturally reaches a zero point at the moment you hit manual and exploratory tests, which are non-automated). The earlier and more often in the process you test, the better and more cost-effective the outcome.

It costs to automate high-level tests

As you move higher up the agile testing pyramid, you encounter issues like:

- **Test fragility** (tests that break easily and unexpectedly, even when changes
- 'shouldn't' have influenced the test)
- **Longer feedback time**
- **Increased effort levels**
- **Higher costs to implementation**
- **More specialized knowledge required**

For example, UI tests break easily, often because the UI structure changes over time. Low-level tests are faster and more stable, so this is where we recommend you put the vast majority of your effort - into unit, component, and integration tests. Then write fewer higher-level tests, like UI and end-to-end tests. Use tools like **Selenium** to avoid the **testing ice-cream cone**.

Quite the climb

At the start of your journey, the agile testing pyramid can look pretty unscalable. But, if you break it down, that summit is closer than you might think. For more information about testing types, see **our article on software testing basics.**

About Onpath Testing

For more than 20 years, OnPath Testing, a Colorado-based, award winning QA service, has partnered with organizations in a wide range of industries to ensure their software works as expected and as designed.

OnPath has led the way in establishing a QAOps model within dev teams, and has taken a forward stance on IoT and AI-based tools and methods. OnPath champions WCAG digital accessibility standards in theory and in practice as we move into a new era of connectivity.