The Visual Testing Revolution

[Slide]

Hello everyone! Thanks for attending my talk today. My name is Andrew Knight, but you can call me “Pandy” for short. I’m the Automation Panda. Historically, I’ve been a Software Engineer in Test, building high-scale solutions for testing problems. For almost a year now, I’ve been *the* Developer Advocate at Applitools.

[Slide]

Another cool thing I do at Applitools is serve as Director of Test Automation University. TAU is one of the best platforms online for learning about testing, automation, and software quality, and all the courses are FREE! I’ve taught a few courses on Python. If you’ve taken any TAU courses, drop their names in the chat so I can see!

[Slide]

Remember when we were kids how our activity books had those spot-the-difference challenges? Let’s do one together. Take a look at these two pictures. They depict a cute underwater scene. How many differences can you spot? Call them out in the chat, and then drop the number you see. [Pause]

Okay, let’s see how many differences there actually are!

[Slide]

There are a total of 10 differences between these two pictures! Did anyone catch them all?

Collectively as a group, it took us a while to spot all the changes, and we didn’t even catch them all!

The comparison you see here was generated by the Applitools Images CLI. It zeroed in on all differences in a matter of seconds. This is the power of Visual AI. Humans can’t always catch everything, even when given lots of time. Visual AI is much faster and more accurate.

[Slide]

Solving kids' problems is pretty cool, but Visual AI is very useful for software testing. Imagine that you’re developing a web app or a mobile app. How do you know that your latest changes didn’t skew a layout region, or nullify a title, or cause overlapping text? It’s hard to find visual bugs when you must continuously grind out new features, especially when traditional automated testing typically misses these sorts of bugs.

[Slide]

Visual testing is indispensable for UI and UX these days. We develop software for humans. We can't disregard visual aspects. Visual bugs make apps just as broken as backend bugs. You risk your business and your reputation if you ignore them. Thankfully, we have the technology to automate visual testing at scale.

[Slide]

What you'll find is that visual testing isn't really a new form of testing. It's just plain-old functional testing – something our industry has done for decades. In fact, visual testing was arguably the first kind of testing ever done: someone always had to physically look at computer output to determine if it was right or wrong! Technologies like Applitools Visual AI enable us to automate visual testing at scale. It works with your existing tech stack, too. You can integrate it with the tools and frameworks you're already using today.

[Slide]

By the end of today's session, you'll also see that visual testing is *easier* than traditional test automation. Instead of thinking about the hundred different things to check on a page, you can make one-line snapshot calls to cover everything at once. Easy peasy! Visual testing isn't an "advanced" technique. You don't need to have an expert team or a mature project to get value from it. Visual testing makes functional testing easier and stronger. In fact, it's something teams should do first before attempting to automate longer, more complicated tests with traditional techniques.

Big claims, right? Let's see what I mean. We are going to automate a web test together in Java with Selenium WebDriver using traditional interactions and verifications, and then we will supercharge it with visual testing techniques using Applitools.

[Slide]

First, we need a web app to test. We could test an app of any size, but I'm going to choose a small one for the sake of our demo. This is Applitools' demo site. It mimics a banking application. You can try it yourself at https://demo.applitools.com/.

The login page has a main icon, username and password fields, and a sign-in button. Since this is a demo site, you can enter any username or password to log in.

[Slide]

After clicking the sign-in button, the main page loads. There’s a lot of stuff on the main page. The top bar has the name of the app, a search field, and icons for your account. The main part of the page shows financial data. The left sidebar shows different account types.

[Slide]

We could write a basic login test for this app in four steps:

1. Load the login page.
2. Verify that the login page loads correctly.
3. Log into the app.
4. Verify that the main page loads correctly.

This could be a smoke test. There’s nothing fancy here. The trickiest part for automation would be deciding which elements to check on the loaded pages.

[Slide]

We could automate this test in Java using Selenium WebDriver. Technically, we could automate it using any popular language and tool. Personally, right now, I really like Python and Playwright, but based on different reports I’ve seen, Java is still one of the most popular languages for test automation, and Selenium WebDriver remains the most popular browser automation tool. JavaScript, C#, and Ruby are other popular languages for test automation, and Cypress is a very popular alternative to Selenium.

In Java, we can write a JUnit test class named “LoginTest” and create a test case method named “login”. This test case method calls four helper methods – one for each step.

[Slide]

The first method, “load login page”, loads the demo app’s login page in the browser.

[Slide]

The second method, “verify login page”, verifies the appearance of five critical elements on the page: the logo, the username field, the password field, the sign-in button, and the remember-me checkbox. It waits for each of these elements to appear using a helper method named “wait for appearance”.

[Slide]

The third method, “perform login”, enters a username and password, and then clicks the sign-in button. So far, so good. Nothing too bad.

[Slide]

The fourth method, “verify main page”, is a doozy. Remember all the things on that page? Well, they’ll need several assertions to verify. Some assertions merely check the appearance of elements.

Others need to perform text matching. For example, [Slide] to check the banner at the top that says, “Your nearest branch closes in X minutes,” we need to find the element, get its text, and then perform a regular expression match. [Slide] Checking the account types and status fields require getting lists of elements, mapping their text values, and transforming the resulting data for comparisons.

[Slide] Despite this heavy lifting, this page still doesn’t check everything on the main page. Dates, amounts, and descriptions are all ignored as a risk-based tradeoff. We could add more assertions, but they would lengthen this method even more. They could also be difficult to write and become brittle over time. Tests can’t cover everything.

[Slide]

If we run this login test against our web app, it should pass without a problem. But what if the page changes? [Slide] Here’s a different version of the same page with some slight visual differences. Can you see what they are? Let me go back and forth a few times for you to see. [Slide back; Slide forward]. Will our login test still work? Will it pass or fail? *Should* it pass or fail?

[Slide]

What about this page? Here, I stripped off all styling. The HTML structure remains, but the layout, the colors, and the background are all gone. Would the Selenium test we wrote still pass? Yes, actually, it would. The locators we wrote used IDs and CSS selectors. As long as the HTML is the same, the script will still find the elements it needs. That’s a *huge* test gap!

[Slide]

Looking at different versions of the page side-by-side makes comparison easier. Here are the first two versions we saw. The logos are different, and the sign-in buttons are different. While I’d probably ask the developers about the sign-in button change, I’d categorically consider that logo change a bug. Unfortunately, as long as the page structure doesn’t change, our login test will still pass. It wouldn’t detect these changes. We probably wouldn’t find out about these changes if we relied exclusively on traditional test automation.

[Slide]

Applitools makes this type of side-by-side comparison easier. The first time you run your tests, Applitools saves each page as a “baseline” image. Then, reruns are called “checkpoint” snapshots. If no visual differences are detected, then the test automatically passes. If there is a difference, then Applitools highlights it in magenta. If the change looks good, you can mark it with a thumbs-up, which will pass the test and save the checkpoint as a new baseline. However, if the change looks bad, you can fail it with a thumbs-down.

[Slide]

This is what we call *visual testing*. If a picture is worth a thousand words, then a snapshot is worth a thousand assertions.

[Slide]

One visual snapshot captures *everything* on the page. As a tester, you don't need to explicitly state what to check: a snapshot implicitly covers layout, color, size, shape, and styling. That's a huge advantage over traditional functional test automation. It can do what human eyes do at scale.

[Slide]

That's how one visual check could capture all ten differences in our spot-the-difference picture from before. All I need to do is run it through Applitools and…

[Slide]

… See the differences?

Wait, what just happened? Why is the whole picture highlighted in magenta? That's not right.

[Slide]

Not all visual comparisons are made equal. What I'm showing here is an exact pixel-to-pixel comparison. When I created these two images, I shifted them to have slightly different padding so that their pixels wouldn't exactly align. This is common for UIs of all kinds. Thus, every single pixel is marked as being different. This kind of result is useless – it's just noise that doesn't help me find the "real" differences.

[Slide]

Visual AI overcomes this limitation. It focuses on the changes that matter and ignores the ones that don't. It uses machine learning to robustly analyze images as if it had human eyes.

[Slide]

That's the second advantage of good automated visual testing: Visual AI, like what Applitools does, focuses on meaningful changes to avoid noise. Visual test results shouldn't waste testers' time over small pixel shifts or things a human wouldn’t even notice. They should highlight what matters, like missing elements, different colors, or skewed layouts. Not all visual testing tools rise above pixel-to-pixel comparisons.

[Slide]

So, let’s update our login test to do visual testing with Applitools. First, we need to create an Applitools account. Anyone can create one for free at the link I’m showing here. You can use your GitHub account or an email address, and you don’t need to enter a credit card.

The account will come with an API key that must be set as an environment variable for testing.

[Slide]

Next, we need to add the Applitools Eyes SDK to our project. Since we are using Selenium WebDriver with Java, we need to add the “com.applitools.eyes-selenium-java3” Maven dependency to our pom.xml file.

[Slide]

Then, we need to set up visual testing objects in a test setup method. We will still use the WebDriver object, but we also need a VisualGridRunner to upload snapshots to Applitools Ultrafast Grid, and we need an Eyes object to take visual snapshots during test execution. The Configuration object sets the API key, the test batch name, and the browsers we want to test.

[Slide]

The steps for our login test are the same, but before and after the test steps, we must “open” and “close” Applitools Eyes so that we can take snapshots during the test. Opening requires the WebDriver reference, the app name, and the test name.

[Slide]

The interaction methods can remain unchanged, but we need to update the verification methods. Traditional assertions can be completely replaced by one-line snapshot calls. Take a look at “load login page”: five lines reduced to one, and the visual snapshot technically has far greater coverage.

[Slide]

The impact on “verify main page” is far greater. One visual snapshot eliminates the need for several lines of assertions.

[Slide]

This is the third major advantage visual testing has over traditional functional testing: visual snapshots greatly simplify assertions. Instead of spending hours deciding what to check, figuring out locators, and writing transformation logic, you can make one concise snapshot call and be done. As an engineer myself, I cannot understate the cognitive load this removes from the automation coding process. I said it before, and I'll say it again: If a picture is worth a thousand words, then a snapshot is worth a thousand assertions.

[Slide]

So, what about cross-browser and cross-device testing? It's great if my app works on my machine, but it also needs to work on everyone else's machine. The major browsers these days are Chrome, Edge, Firefox, and Safari. The two main mobile platforms are iOS and Android. That might not sound like too much hassle at first, but then consider:

- All the versions of each browser – typically, you want to verify that your app works on the last two or three releases
- All the screen sizes – modern web apps have responsive designs that change based on viewport
- All the device types – desktops and laptops have various operating systems, and phones and tablets come in a plethora of models

We have a combinatorial explosion! Traditional functional tests must be run start-to-finish in their entirety on each of these platforms. Most teams will pick a few of the most popular combinations to test and skip the rest, but that could still require lots of test execution.

[Slide]

Visual testing simplifies things here, too. We already know that visual testing captures snapshots of pages in our applications to look for differences over time. Note how I used the word "snapshot" and not "screenshot." That was deliberate. A *screenshot* is merely a rasterized capture of pixels reflecting an instantaneous view. It's frozen in time and in size. A *snapshot*, however, captures everything that makes up the page: the HTML structure, the CSS styling, and the JavaScript code that brings it to life.

Snapshots are more powerful than screenshots because snapshots can be rerendered. For example, I could run my test one time on my local machine using Google Chrome, and then I could re-render any snapshots I capture from that test on Firefox, Safari, or Edge. I wouldn't need to run the test from start to finish three more times – I just need to rerender the snapshots in the new browsers and run the Visual AI checker. I could re-render them using different versions and screen sizes, too, because I have the full page, not just a flat screenshot. This works for web apps as well as mobile apps.

[Slide]

Visually-based cross-platform testing is lightning fast. A typical UI test case takes about a minute to run. It could be more or less, but from my experience, 1 minute is a rough industry average. A visual checkpoint backed by Visual AI takes only a few seconds to complete. Do the math: if you have a large test suite with hundreds to thousands of tests that you need to test across multiple configurations, then visual testing could save you hours, if not days, of test execution time per cycle. Plus, if you use a service like Applitools Ultrafast Test Cloud, then you won't need to set up all those different configurations yourself. You'll spend less time and money on your full test efforts. The Ultrafast Grid provides web browser configurations, and the Native Mobile Grid provides Android and iOS configurations.

[Slide]

I've made a lot of bold claims about visual testing, so let's see it in action!

[Screen sharing]

[Off-script:

1. Show the Java test project
2. Run the visual test once to establish baselines
3. Run it again to show passing tests
4. Change the URL to point to the broken site
5. Run the test again to show visual differences
6. Show the differences and thumbs-down
7. Show Root Cause Analysis
8. Show Accessibility contrast advisor
9. Show visual locators
10. Show A/B testing baselines
11. Mention GitHub integration
12. Rerun the tests

]

[Slide]

Before I conclude this talk, there is one more thing I want y'all to consider: when a team should adopt visual testing. I really think teams should do visual testing from the start. Consider this strategy: start by automating a smoke test that navigates to different pages of an app and captures snapshots of each. The interaction code would be straightforward, and the snapshots are one-liners. That would provide an immense amount of value for relatively little automation work. It's the 80/20 rule: 80% of the value for 20% of the work. Then, later, when the team has more time or more maturity, they can expand the automation project with larger tests that use both traditional and visual assertions.

[Slide]

Test automation is hard, no matter what tool or what language you use. Teams struggle to automate tests in time and to keep them running. Visual testing simplifies implementation and execution while catching more problems. It offers the advantage of making functional testing easier. It's not a technique only for those on the bleeding edge. It's here today, and it's accessible to anyone doing test automation.

[Slide]

With visual testing techniques, hopefully your testing can be as satisfying as this visually perfect puppy enjoying her matcha latte.

[Slide]

Thank you all for attending my talk today! Again, my name is Pandy Knight, and I'm the Automation Panda. If you want to keep in touch with me, here are all my links:

- My blog is AutomationPanda.com
- My Twitter handle is @AutomationPanda
- My LinkedIn profile is here
- My puppy, Suki the Frenchie, is on Instagram

I'll wait a minute for y'all to take a screenshot. [Pause]

[Slide]

And here's the money shot: all 5 key advantages of visual testing with links to the example code. I also turned this talk into a blog article, which is linked at the bottom as well. Again, now's your chance to take a screenshot.

Visual testing is a winning strategy. It has several advantages over traditional automated techniques. With Applitools Eyes, you can do visual testing in any major language or test framework you like, and with Applitools Ultrafast Grid, you can do visual testing using any major browser configuration. If you want to give it a try, you can sign up for a free account. You can clone the example project I showed today to try running tests yourself, or you can take one of Applitools' many tutorials.