# **Continuous Testing**

Chapter: Functional Test Automation via AI

***Jason Arbon, CEO test.ai***



***Jason*** *is a test nerd, and currently the CEO of test.ai, where his mission is to automate the testing of world’s apps with AI. He has also been the director of product and engineering at Applause.com/uTest.com. Jason previously held engineering leadership roles at Google (Chrome/Search) and Microsoft (WindowsCE, SQL Server, BizTalk, Bing). He also co-authored the books: How Google Tests Software and App Quality: Secrets for Agile App Teams. In his spare time, Jason likes to read up on AGI and consciousness and is working on a new personalized search engine. Note that this chapter outlines the basics of how to Build AI-based test Automation Systems, and test.ai has filed patents for specific implementations of these concepts.*

# Introduction

Software testing is about to be transformed by Artificial Intelligence (AI) and Machine Learning (ML). While other aspects of software engineering have improved dramatically in the past decade, software testing still looks much the same. Testing hasn’t changed much because testing requires human judgement, manual activities, domain knowledge, and empathy for the end user--all of these require human-level intelligence. AI is a way to build software that can replicate human judgement. The resurgence of AI as a field combined with affordable computing, means that AI can be applied to some of the most challenging aspects of automated testing, and deliver AI-assisted testing for humans.

AI is a broad category of software including machine learning where the software is ‘trained’ to do basic tasks with or without human instruction. AI even includes work on Artificial General Intelligence (AGI)--the work to construct conscious, even superintelligent machines. This chapter is humbly focused on those parts of ‘AI’, particularly machine learning techniques, that can be practically applied to common software test automation tasks.

Of all professions, Software Testing is the most ripe to be automated via AI. Most applications of AI rely on pattern matching. Today, AI is being applied to many fields, from radiology, to driving cars, to making hair appointments over the phone. These applications use the output of the AI training process and apply that to specific problems such as: analyzing MRI scans, recognizing a stop sign and stopping a car, or composing conversations with hair stylists. These are problems which consist of inputs and comparing the outputs to expected results. Testing is another matter altogether though as the basic processes of software testing are similar to the processes used to train AI. Testing is fundamentally an AI training problem. The good news is that since the processes are so similar, all the money spent today on infrastructure and researching AI training techniques is really an investment in AI test automation.

Testing is fundamentally the process of applying inputs to an application / system-under-test, observing the outputs and checking those outputs against the expected values. (See figure A)

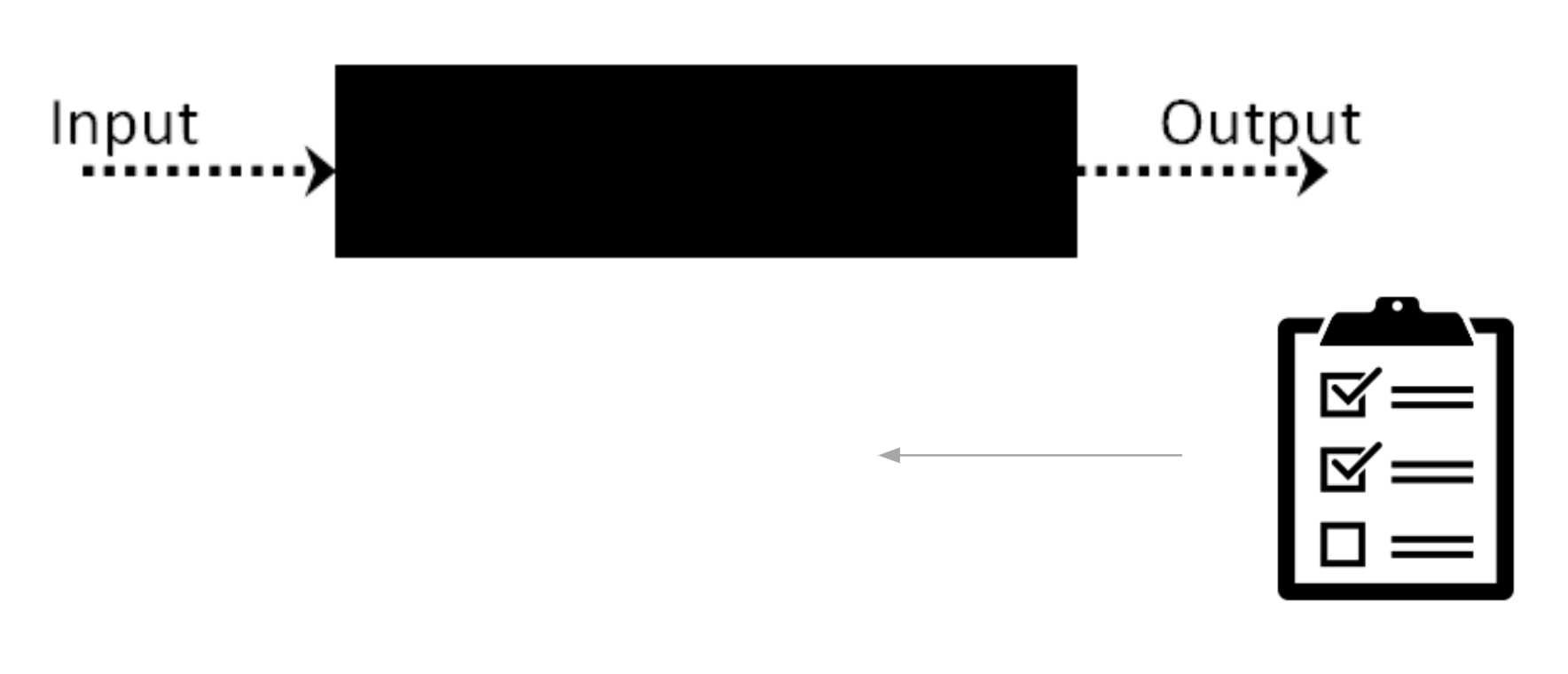


Fig. A: System Testing

Training AI Systems is very similar to testing. Instead of applying inputs and measuring outputs to an application under test, AI systems apply inputs to the neural network (or other model) and measure the outputs from that model. AI training systems also compare the output of the neural network with the expected value (training data). This looks very similar to Testing. It \*is\* testing. (See figure B)

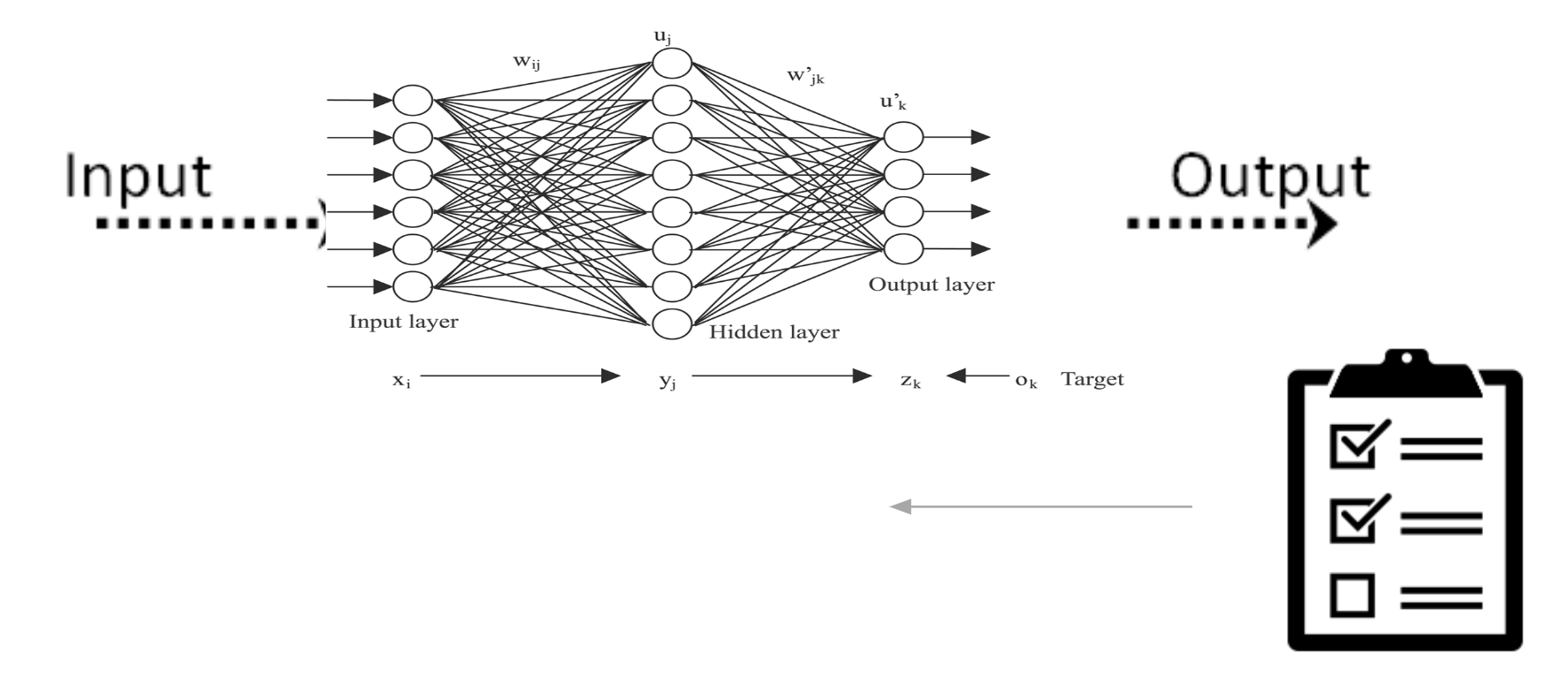


Fig. B: AI Training is very similar to Testing

Not only is AI training similar to testing, but most testing activities consist of quick visual inspections to determine what to do next, or to determine whether the application’s functionality is correct. Modern AI is great at solving such quick-twitch judgment calls. Even better, the AI can be trained on the judgement of not just one tester, but on the collective wisdom of thousands of smart testers, and have all that brainpower encoded in the machine.

*"Pretty much anything that a normal person can do in <1 sec, we can now automate with AI.." --Andrew NG, Twitter[[1]](#footnote-0) Stanford CS faculty. Former head of Baidu AI Group/Google Brain.*

Test automation is a combination of testing best practices and software development so it is often as time consuming and expensive as application development. There are many types of testing. The most painful today is UI (User Interface)-based functional regression testing as the application is constantly changing, the test code needs to drive another application, not just an API (Application Programming Interface), or call a function (Unit Testing). UI test automation is fundamentally composed of four major tasks:

1. Test Definitions
2. Screen and Element Identification
3. Test Step Sequencing
4. State Verification

We explore below how each of these aspects of test automation can be written with an AI-first approach. For each test automation task, we’ll also explore how AI improves the following aspects of testing:

1. Efficiency of Test Development
2. Reliability of Test Execution
3. Reduced cost of maintenance
4. Re-use of Test Artifacts across platforms and applications.

Lastly, the sum of all these improvements in test automation will usher in a new world of software testing. Ultimately, most test automation will be centralized thanks to re-use and AI. Most test cases will already be written for an app--before it is even implemented. And most importantly, AI-powered test automation means competitive benchmarking in quality is finally possible. AI won’t just make test automation better faster and cheaper, it will fundamentally revolutionize the testing profession and help standardize measures of quality as the same ‘test’ can now be performed on different platforms and even different applications.

# AI Test Definitions

Manual test cases are often written in human language, as their purpose is to describe the test for people to read and execute. Most test case definitions however are written in procedural code, or long winding paths of poorly written Python or Java test scripts. This approach is less than ideal:

1. Only programmers can create or modify the tests
2. Test code is difficult to write, test, and debug
3. Most of the code written is for test case setup and tear down--not the test.
4. Test code must be manually updated when the application changes

The astute reader may be thinking of the Cucumber test framework. On the surface, Cucumber tries to tackle this test definition problem so that tests can be constructed in a human like language, and then executed by a second system or model of the application. The reality is that Cucumber projects often fail because they still have the same problems of procedurally coded tests. The Cucumber tests need a few programmers to connect the human like language of the test definition to executable code that drives the application. We’ll see how AI can do all that magic for the humans below.

*"Cucumber is not a tool for testing software. It is a tool for testing people's understanding of how software (yet to be written) should behave." --Aslak Hellesoy, Creator of Cucumber, Hacker News[[2]](#footnote-1).*

AI test cases need to be written in some sort of language. Ideally, these test definitions should be abstracted from the application’s implementation as far as possible to make the test case execution as flexible and reusable. The test definition should also be human-readable so both machines and humans can work from the same test case definition and be free from the need to know or bother with programming languages for test creation or execution. At this point, the best candidate for such a test case definition format is the Abstract Intent Test (AIT) language, which conveniently has “AI” in its acronym. This format is an open standard and actively worked on by the AI for Software Testing Association (https://www.aitesting.org).

AIT borrows from the learning of Cucumber and its Gherkin[[3]](#footnote-2) language as it is designed for human readability, and often used by designers or product managers to specify the functionality of an application. AIT just adds some additional syntactic sugar to the steps of a Gherkin scenario so that it is readily readable by machines. AIT enables test definitions to be as precise or as general as the test author likes. Steps that are left out, or obvious, are performed by the AI automatically. For those steps that are specific and necessary for the test, the AI will execute those steps exactly as declared by the test author.

The full AIT language specification is beyond the scope of this chapter, but a simple example should suffice. Here is simple AIT defined for executing a test case against a search engine, searching for the ‘Gradient’, and verifying that a search result with the word ‘Gradient’ is returned.

Test Name: Search for Gradient

Description: Perform a simple, single word, web search for the term “Gradient”

Tags: “Single\_Word”, “Search”

Step: Get To Search

StepType: Navigation

Labels: “Search Button”

Step: Enter Search Text

StepType: TextInput

Text: “Gradient”

Labels: “Search Box”

Step: Execute Search

StepType: Action

Action: Tap

Labels: “Search Button”

Step: Verify Result Appears

StepType: Verify

VerificationString: “Gradient”

Match: 3

Labels: “Search Result”

The reader, without programming knowledge, can easily determine that this is a test case that searches for the word “Gradient” and verifies that there is a result with the word “Gradient” in it. Lets walk through each step quickly and note a few things.

The Test Name and Description are straightforward in meaning. The “tags” field is a way for organizing and searching for tests when there are large numbers of tests.

The “Get to Search” test step does just that. It is a step of type ‘navigation’, which tells the AI to ‘find the search page’ and we’ll start the test from there. The AI looks through the app for any element that can be classified as ‘search page’. The magic of this step is that we can just tell an AI, just like we would a human, to ‘start the test on the search page’. Because the mechanics and specifics of how to do this aren’t specified in code, or in exact steps, we are leaving it to the AI (or human) to figure out how to get there. If we had to specify the exact buttons or links to click to get to the search page, the test would ‘break’ everytime that part of application changed. We’ll discuss later how this magic works.

The “Enter Search Text” step is a TextInput step. It instructs the person or AI to put the text “Gradient” into the “Search Box”. How does the AI know what a search box is without knowing a magic ID, XPATH, CSS Selector, etc. of the search box element? That magic is described below. The beautiful thing here is that the test author doesn’t have to know anything about the implementation of the application. The AI, just like a human, should be able to figure out how to find the search box and enter the text.

The “Execute Search Step” executes the search by clicking on a button called “Search Button”. Again, the magic of how the AI knows it is a search button is described later.

The “Verify Result Appears” step verifies that there is a search result with the word “Gradient” in it. The AI looks on the page for things that are called ‘Search Result’, and scans each of them for the presence of the word “Gradient”. The match clause specifies that for this step to count as a passing test, this matching of the word “Gradient” has to appear in at least 3 test result objects on the page.

Here we have defined a test case that is human-readable, doesn’t require a programmer, and promises that there is a whole lot of AI magic to convert such high level steps into an automated test case.

It should also be noted here that this test case doesn’t have a lot of things required for most test automation: It is missing things like magic IDs, CSS selectors, accessibility labels, or XPATHs to find element. It is also missing any platform-specific code. The same test could run on a mobile device, or a web page. Most interesting is that this test case also doesn’t mention the actual application. This test could run on any search engine, or application with search functionality. AIT test cases are by design written to be platform and application agnostic. Given the right human or AI test execution, this test can be reused everywhere. This test need only be written once in the world.

AI-powered testing means that test cases are also quick to write, with no programming knowledge needed, and can magically execute test cases across platforms and applications--just like a human could. The same test case can deal with very different user interfaces, numbers of steps, platforms, and apps.

Now that we can define the ideal test case for execution by AI (and humans), let’s get into the nuts and bolts of how to get AI to magically execute all these AIT test cases for us.

# AI Screen and Element Identification

A key aspect of test automation is the ability to identify elements in an application. Test automation needs to find elements to interact with them, via taps, swipes, and text input. Test automation also needs to find elements to verify the correct output or results of a test case: i.e. finding the search result item and verifying that it is the expected result by examining the text description of the link.

When humans test an app, they can readily identify what type of screen it is, e.g. search, login, profile, etc. Humans can also do a very good job of determining if a button is a search button, or if a textbox is a search box, or if an image is a picture of a product. People can even readily identify screens and elements in applications they have never seen before, even if the buttons are a bit larger, a different color, or different position on a page because the person has likely seen similar objects in other apps. Much like people are trained to recognize basic application screens and elements, we can teach machines to do the same.

To teach an AI how to classify a screen or element we need lots of training data. Training data is simply a large set of examples of say search buttons. Some may be small, some large, some red, some blue, but we need lots of examples. Many will have the word ‘search’, or ‘go’, in the text of the button. Most will be centered, or on the right hand side of the application, and often in the middle or top near the search text box. We humans know this intuitively, we just need to give lots of examples to an ML model to ‘learn’ to recognize search buttons too.

So how to get a large corpus (set) of training data? There are three steps: First, write a crawler bot that will download thousands of applications(or browse websites) and take screenshots of everything. Second, break down the screenshots into individual images of buttons, text boxes, images, etc. Third, you need to get the labels to train the AI. Labeling is the process of naming each individual image with a label such as ‘search\_button’, ‘product\_image’, or ‘shopping\_cart\_button’. We now have a set of training data to teach the machines to think like human testers.

How do you get labels for hundreds of thousands of images? Amazon’s Mechanical Turk is a common solution. You can pay people pennies per label. Simply send the service a list of images and define the job as ‘please pick one of the following that describe the picture’ (see Figure D). Posting this type of job is pretty easy.

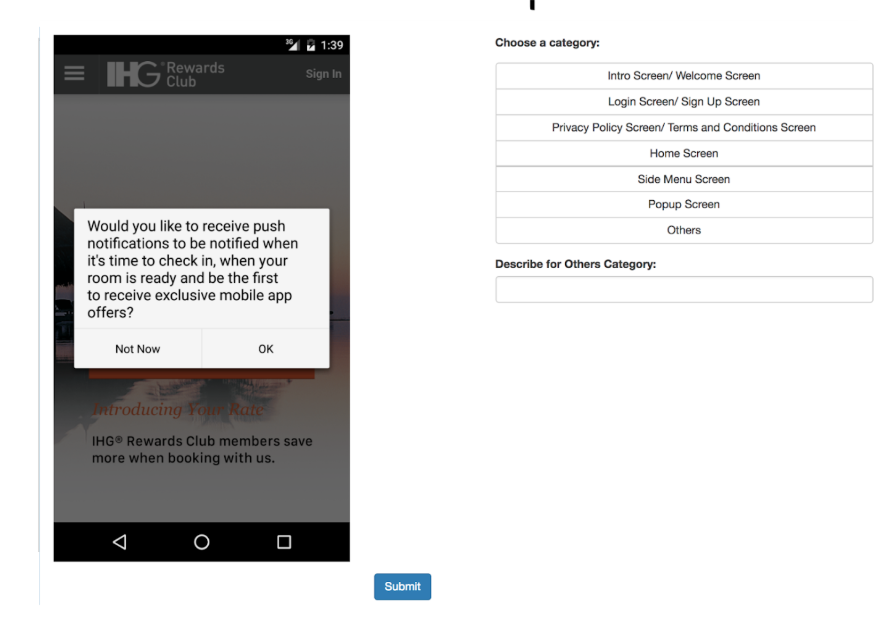


Fig. D: User Interface for Mechanical Turk Labeling

Now, powered with a bunch of training data, which is simply a large set of images that are labeled, we just pass this data to some machine learning infrastructure such as TensorFlow or SciKitLearn python libraries, etc. The detailed mechanics of doing that is beyond the scope of this chapter, but note that you basically just put all the images and labels into a giant array, and call a function to say ‘train’. The AI starts by randomly guessing the correct label for each element. When it is right, it tries to remember how it was configured. When it is wrong, it reconfigures (changes its internal connections) and tries again. In practice this happens hundreds of thousands of times until it can get most labels for elements correct.

Next, feature values need to be computed. Machines cannot really ‘see’ images, so we need to translate things that humans can see into numbers between 0 and 1, so the computer can ‘see’ them too. You can think of features as a function that takes the image and document object model as an input, and then returns a value between 0 and 1. A few feature examples that have proven to be useful in test automation:

Features for Elements:

* **%Height**: determine the height of the image in pixels and divide that value by the max height of the device’s screen. This results in a value between 0 and 1, representing the relative height of the image.
* **%Width, X, Y, Area,** etc (same as % Height)
* **Average Color**: Look at every pixel in the image. For each pixel determine the different Red, Green and Blue (RGB) components which are values between 0 and 255. Then, for each color, divide that value by 255 to normalize the numbers to a value between 0 and 1.
* **Text match**: Run Optical Character Recognition (OCR) on the image to extract any possible text. For interesting values such as ‘Login’, or ‘Allow’, create a separate feature that sends a 0 or a 1 depending on whether that text is present in the image.
* **Text length**: Similar to text match, but length of string divided by the maximum expected string length.

Features for Screens:

* **Number of Elements**: Number of total elements on the screen. Simply count the number of elements on the screen and divide by a maximum number of screen elements expected (e.g. 1000).
* **Average Color**: (see above)
* **Number of Elements By Type**: Number of buttons, text boxes, images, etc.

If you think about how you determine what a screen or element is in your own mind, you may realize that your brain is doing very similar things. E.g. login screens usually have only a few elements, often ‘blue’, and often only a single button. Whereas a product listing page might have hundreds of images, one text box, and only a couple buttons, and mostly white. Computing these features and passing this data to the AI training systems enable the AI to ‘see’ similar to how humans see the screen. AI might even learn some patterns or hints that we humans don’t even notice. A production system might include hundreds of these ‘features’.

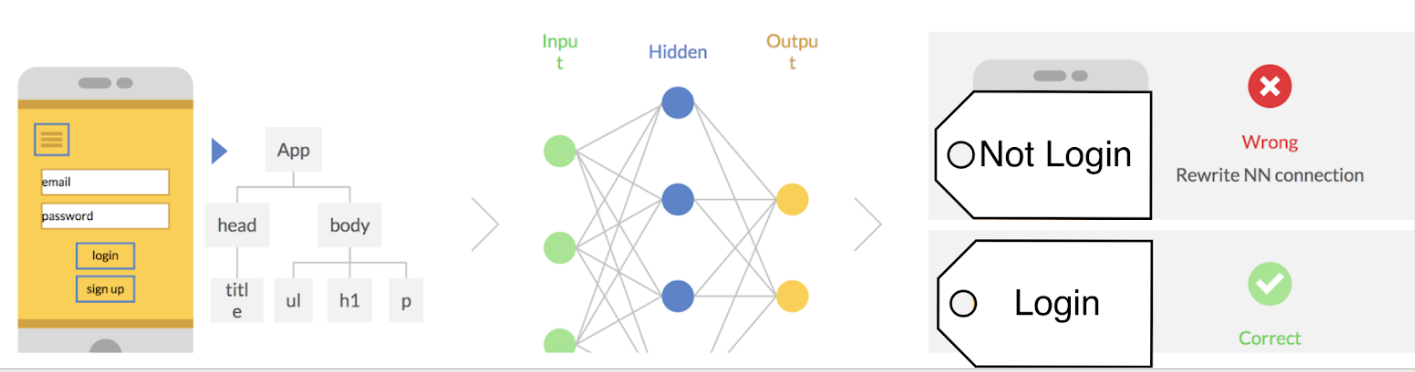


Figure E. Example of AI Training for a Login Screen

Now, these vectors of feature values for each image, along with its correct label (training data) are passed to the AI training system. A high-level view of the training process follows:

1. A randomly configured ‘Brain’ is generated
2. The system presents each image and its ‘feature values’ to the neural network and measures whether the network guesses the correct label.
   1. If the label was correct, the network is ‘reinforced’ that the current configuration is a good one.
   2. If the label was incorrect, the network is ‘changed’ in hopes that next time it will guess correctly.
3. The system repeats this process hundreds of thousands of times until the network has ‘learned’ to produce the correct results.

After 24 hours of computation on high-end machines, the AI training system generates a neural network that can now take the input of an image, and suggest the correct label for that screen or element. Just like a human can. Perhaps better than a human. Testing the correctness of such a system is also out of scope for a quick chapter on the topic, but essentially the brain is tested using a separate set of images that were not used in training, and the brain is tested to see how accurately the brain, like a human brain, performs on those new images. This AI approach is known as supervised learning as we have supervised the learning of the system by giving it many examples and letting the machine learn to reproduce the same output as the humans.

With all this work, we now have an alternative way of identifying parts of an application. Remember, traditional non-AI software automation finds elements by hardcoded searches in the application for a magic accessibility or other ID value, XPATH, or CSS Selectors. AI based element identification however finds elements just like the human brain.

The AI-based approach to identifying elements is far more complex to setup, but it has two distinct advantages versus traditional methods:

* **Speed**: The engineering time it takes a human to write the code to identify an individual element can be several minutes. With an AI trained on thousands of different elements, the time to identify an element is less than a second and can be done at runtime. This represents a 10X development speed/cost improvement. Moreover, these AI classifiers can be shared between app teams, so only one or a few people on the planet need to build these classifiers, and everyone else can benefit.
* **Robustness/Maintenance:** A key problem in test automation today is that the applications are constantly changing. The color, size, location, or text of a search button may change and break test code, but to the eye of a human, or well-trained AI, the search button still looks like a search button. Where traditional element identification will fail with even minor changes to the application. AI-based identification still keeps working and doesn’t require maintenance.

It is worth noting that the trained AI is the collective knowledge of all people and testers that contributed to the training set of images and labels. When the AI is trained on hundreds of thousands of images with labels from hundreds of different people, it has seen more search buttons than most humans ever will in their lifetime. This means the AI might just be smarter than any single tester at identifying elements in applications.

# AI Test Step Sequencing

A test case is a sequence of steps. A series of inputs and outputs. Find the search screen, enter text in the search box, click the search button, then verify the search results seem relevant to the query. We’ve seen how AI can be trained to identify the individual parts of an application such as search boxes and buttons, but now we need to teach it to accomplish a task that is a series of steps.

Traditional test automation expresses test step sequences in procedural code such as Python or Java. Each step is hard-coded, step-by-step[[4]](#footnote-3), to interact with elements in the application. Procedural code for test steps is problematic in three ways:

1. **Test Automator needs to know how to program**. There are a limited number of competent programmers in the world and they are expensive. Generally speaking, test engineers are not the most experienced engineers, nor do they produce beautiful test code.
2. **Time to Develop Tests**. Programming is labor intensive. Programming is ironically manual in that each line of code must be hand crafted, and it also needs to be tested.
3. **Brittleness/Maintenance**. The biggest issue with procedural code is that if the flow/structure of the application changes, the test automation breaks. Often it breaks at the exact moment the team looks to the automation to verify that the application still works. A/B testing, redesign, interstitial dialogs, etc. can appear during execution and break the expectations of the procedural code.

So, let’s explore ways to motivate an AI brain to build and execute these test case step sequences for us.

The following approach is borrowed from the work of Google Deepmind and OpenAI teams, and how they teach machines to play video games. Again, all this AI work is really just test infrastructure. Playing video games like Mario Brothers are sequences of steps like running, jumping and spinning. People learn to play Mario Brothers by playing around a bit. If they run into a mushroom--they power up. If they run into a coin, they get more points. The humans are rewarded and punished into learning how to play the game correctly. By analogy, paths through a video game look a lot like the paths taken in an app by an automated test case. Just think of training an AI to automatically navigate the application along the waypoints that constitute a test case, instead of walking through the levels of a game. This general AI method is called reinforcement learning. The details are beyond the scope of this chapter, but the astute, curious, clever reader can find a lot more details on how Google’s DeepMind team uses reinforcement learning to play Atari games.[[5]](#footnote-4)

Playing games is fun, but executing test cases is geekily-amazing. To teach an AI how to execute the test steps you want, we need a ‘map’ of the application, equivalent to a game’s level design, and we need to place ‘coins’ or rewards on this map to encourage the AI to follow the path we want. Then we let the AI train on thousands of attempts to walk around and explore the map of the application, learning how to execute the path we want it to achieve, to maximize its reward.

The map of the application, aka ‘AppGraph’ is critical to the training of an AI to execute test cases. The map is a simple graph view of the various states of an application. One node may be the home screen, another node may be the search screen. Lines between the nodes represent ways to get from one page to the other, e.g. clicking the search button on the home screen takes the user to the search result screen state. Once we have a map of the application, which can be generated by classic link-following or checking-methods[[6]](#footnote-5), we can begin to train the AI bot to accomplish the testing task.

## How to Train AI to execute a Shopping Cart test case

Let’s explore how to teach an AI to accomplish a basic test case. For a sample test case, lets verify that the application has a shopping cart. First we assign a start state (the home screen), and also set a reward for reaching the first step (the shopping cart in this example). We also set the ‘game score’ to zero to begin. The rules of the game are pretty simple:

1. The AI bot can step from one state to any other state in the app graph.
2. If the AI bot takes a step that doesn’t contain a reward, the bot is penalized one point.
3. If the AI bot takes a step and lands on a state that contains a reward, it gets 100 points.

To train the AI, we let it take thousands of attempts at walking through the graph to accomplish the testing task, which in this case is to get to the shopping cart. Once the AI has been trained to get to the shopping cart, the process begins again to train it to get to the next reward state, which is the next test step state we want the bot to take in sequence.

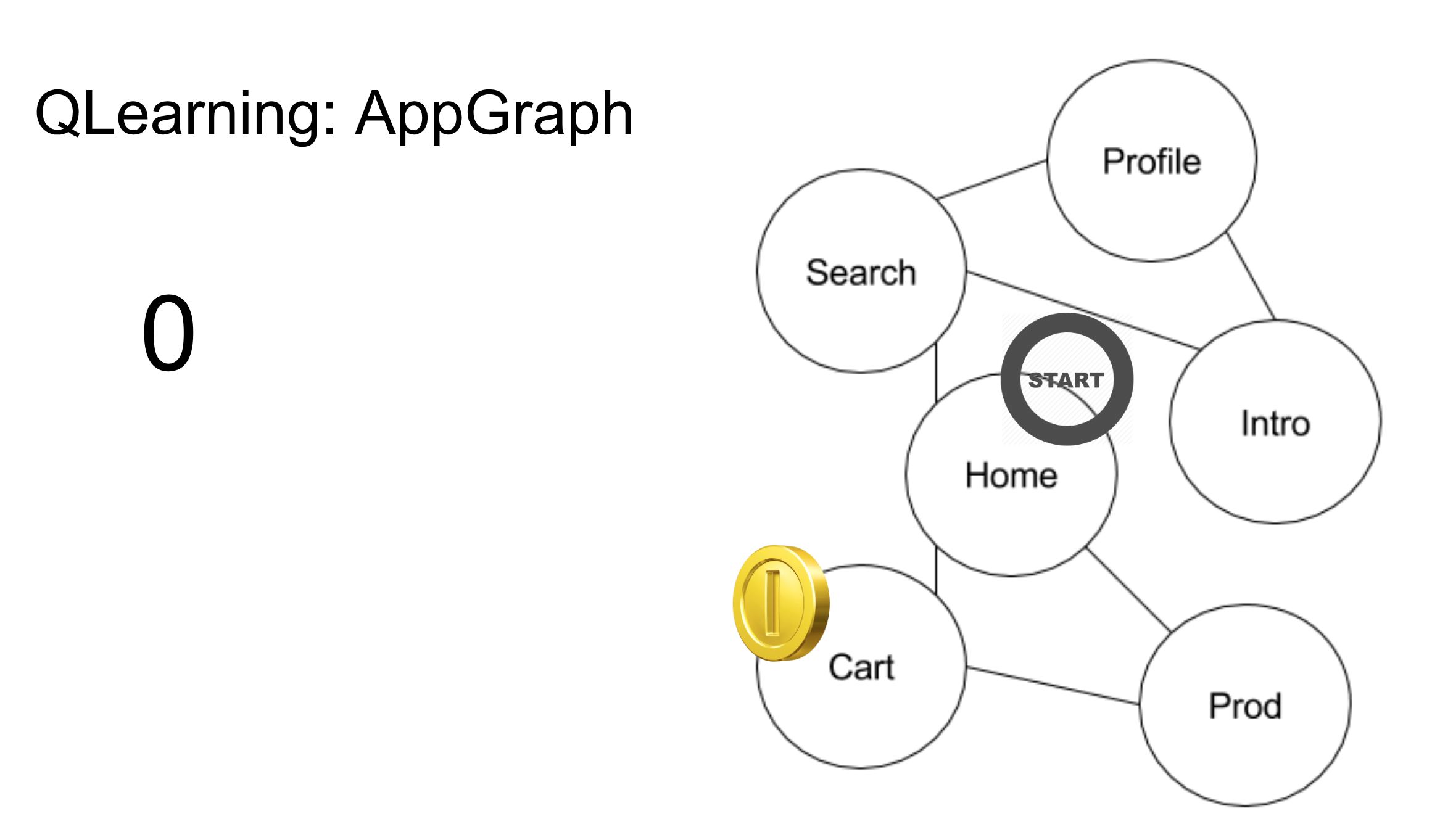


Figure E: Graph of App with Reward at Cart Page

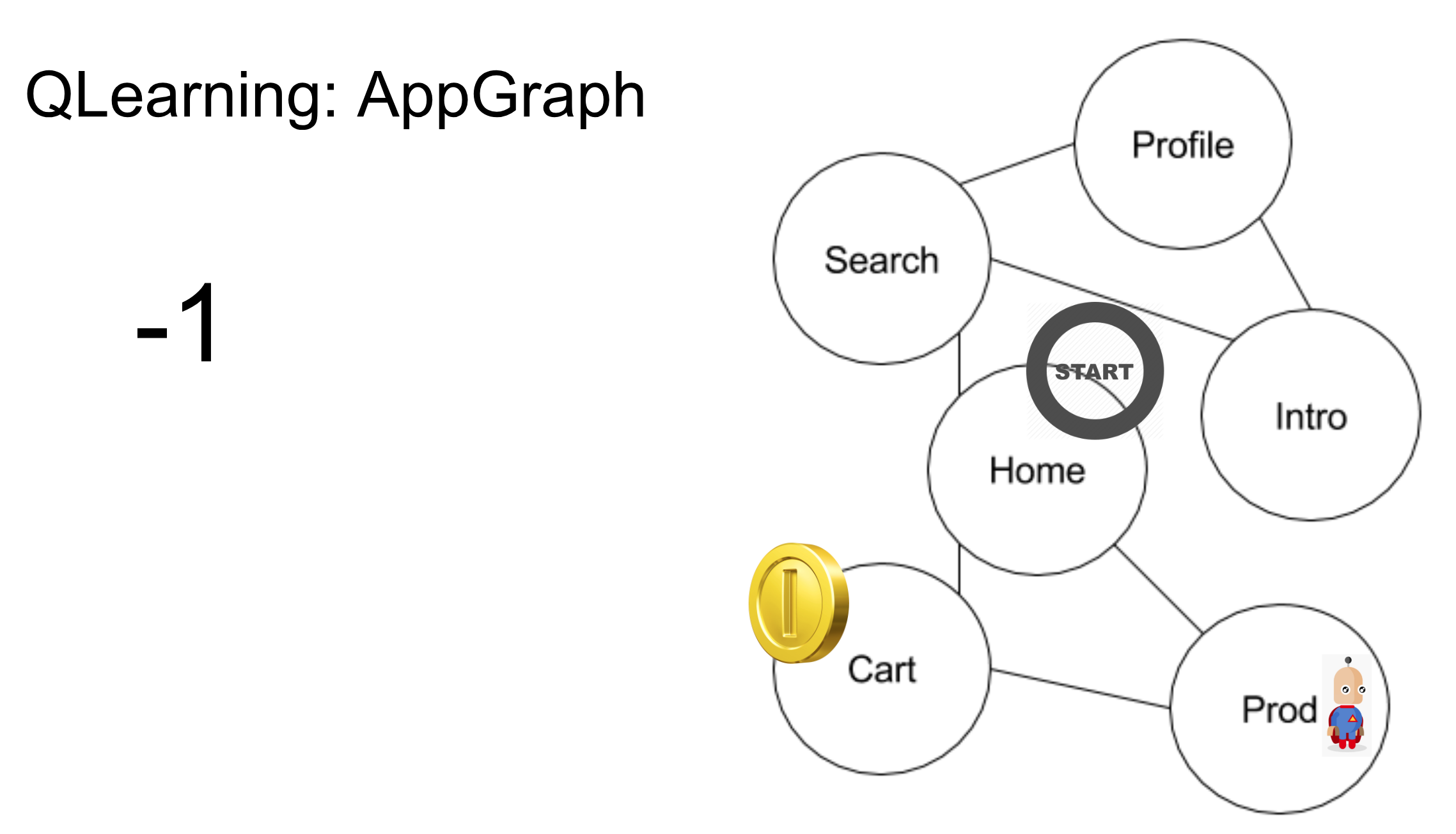


Figure F: Bot takes step to a product page, loses one point.

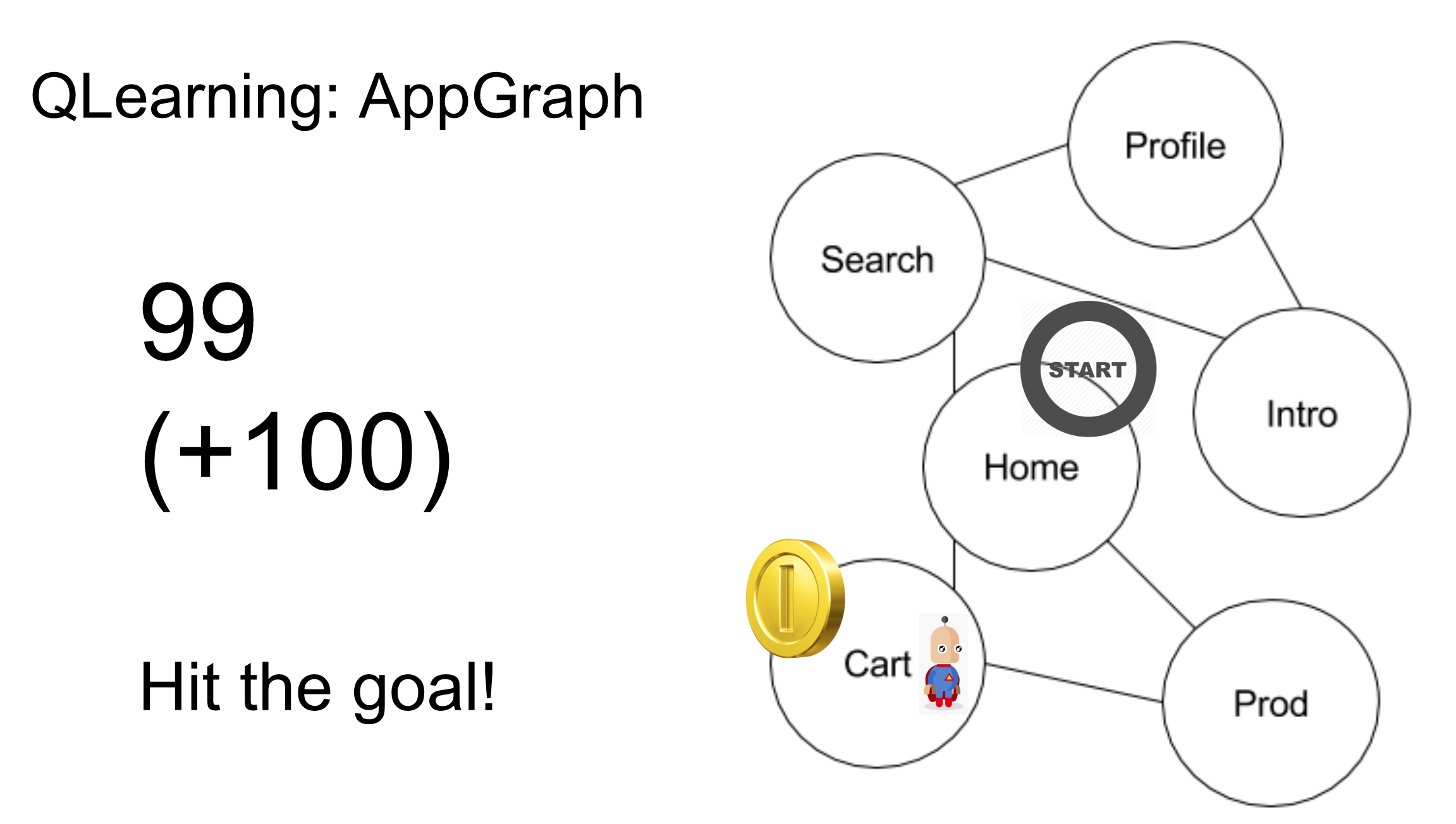


Figure G: Bot takes step to cart page, where the reward lies. Bots net score for this path is 99.

In the figures above the bot found a pretty good way to get to the cart page, but it isn’t the optimal path. On a later training run, the bot may find how to click the shopping cart icon on the home page to go directly to the cart page, which would give it a score of 100 for that path, which is higher than the 99 point score it got for going through the product page first on the way to the cart. The AI training process does this thousands of times, for each step in every test case. The training process produces a ‘heat map’ of how to navigate the app graph in search of the next test step. This is the ‘brain’ of the reinforcement training.

There are three powerful implications of this approach to ‘teach’ an AI how to accomplish a test sequence, versus traditional hard coding of steps:

1. **No Code**. No human programmers were harmed in the construction of this test case. There is no code to be written. The AI training process reads in the test step sequences desired in the AIT and trains an AI to execute those steps in sequence
2. **App change resilience**. The AI learns many ways to get to the cart page. If the default path is removed in a future build of the application, the AI brain, just like a human brain, can use other ways to still find the cart and execute the test case despite app change.
3. **Cross-application**. Each app can have a different graph structure, but the process of training an AI brain to execute the same high-level test is the same, meaning this approach to step sequencing will work on all apps.

Important to note, and easy to miss, is the ability of AI-based test sequencing to take the test execution to a particular place (state) in an application without specifying all the steps to get there. So, only the steps needed to execute the test need to be specified. Ironically, the less specific a test case definition is, the more robust, efficient, and reusable it is.

Figure Z is a demonstration of the same test case ‘Search for the string “Gradient”’, trained and executing across different applications (Bing, Google, Yahoo, and YouTube), and across different platforms (Android and Desktop Web).

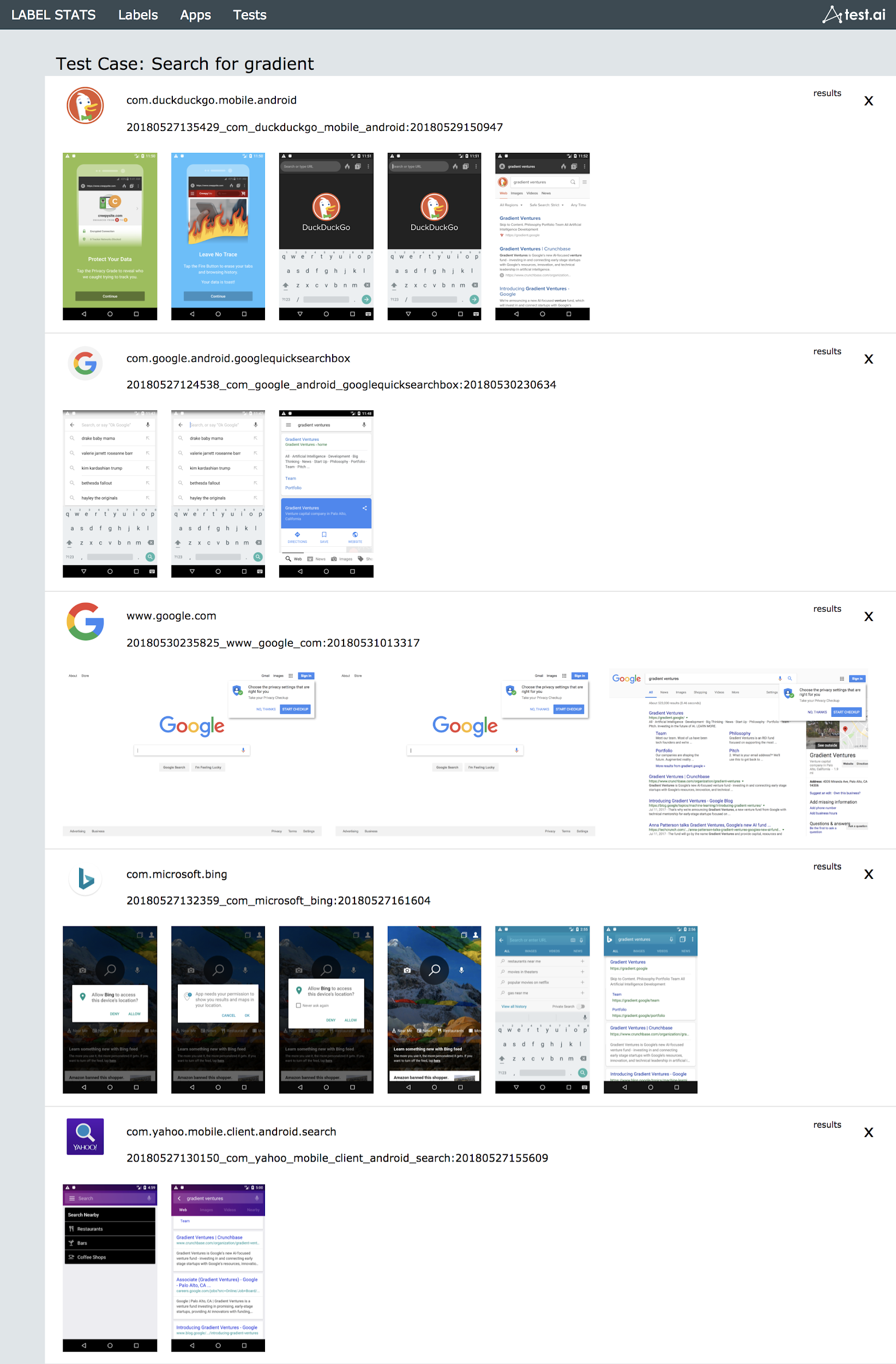


Fig. Z: Test Execution Report for Search Test. Cross-Platform and Cross-application.

The AI-based approach to test sequencing eliminates the need for crafting hard coded steps in procedural code. The AI approach is not only better, cheaper, and faster. The bots have effectively done most of the work that humans do today. The Singularity is near...

# Benchmarking

Software quality, performance, etc. has traditionally been ‘of the tests we have bothered to write so far, x% of them passed’ or ‘we are better than yesterday’. Neither of these answers to quality seems very sophisticated, engineering-like, or confidence inspiring. But, with the arrival of test cases that execute both cross-platform and cross-application, we can now answer the quality question with a number relative to other apps, relative to a benchmark.

AI means it is finally financially feasible to not just automate your own applications, but that of your competitors and see how the two stack up. Which app has a faster login page? Which app is missing particular functionality? Which app fails login tests in production less often?

Figure X shows the performance of search page loading times across several mobile apps. This data was gathered during the test execution above. Google itself would probably be surprised by the results, as even they don’t have the time to write traditional UI regression tests for competitive apps. Benchmarking quality can be motivating to organizations to increase their focus on quality as a competitive advantage.

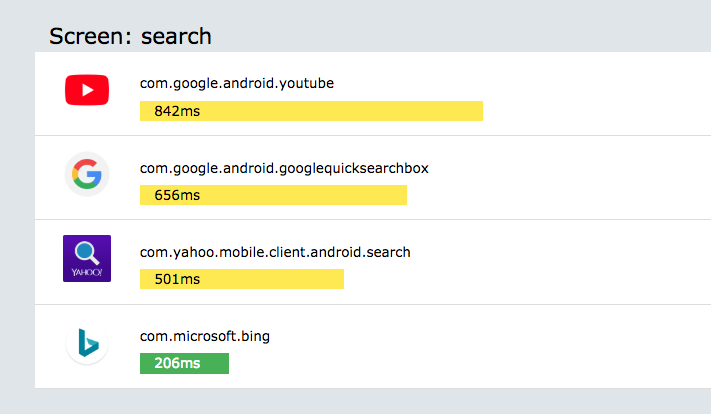


Fig X: Relative performance of mobile app search pages.

Figure X is a rendering of real test results. The very AIT test case defined above, running cross-platform and cross-application. Not only does AI-powered test automation work today, but it also delivers insights into the behavior of applications. Notice how Bing takes far more steps to accomplish a search versus the Google app? Notice that yahoo’s app design means that searching for something takes only two steps and bests Google’s design in user efficiency?

# Centralization of Testing

AI-powered test automation is here. Even in its early form, AI-first approaches to testing are already beating traditional procedural code based approaches to test definition, element selection, and test step execution. This is the first time that test artifacts are truly reusable -- not just across platforms, but across different applications. The test search test case we’ve been exploring above need only be written once. If the first engineer to write the ‘Search for “Gradient”’ test case worked at Google, theoretically with zero porting work, the same test could be leveraged by the team at Bing. AI means teams can now collaborate and share test artifacts. Previously, most every test was rewritten from scratch by every app team. Maybe AI will encourage testers to be more social now that they have an icebreaker--reusable test artifacts.

Better, what if there was a global repository of test cases for shopping carts, login tests, maps, that every app team in the world could leverage with zero code. This means that if you started your own search engine app from scratch today, an automated test suite for that application already exists before you started building it. Instant-on test coverage. The ultimate in Test Driven Development (TDD) where tests are written before the application is coded.

AI-Powered test cases are a step-function in software testing as the test artifacts are now truly reusable. The implication of this is that if testers shared their AIT test cases, test.ai among others can execute AIT defined test cases, there really only needs to be one global repository for most automated testing.

Sure there will always be unique and corner-cases that need test development, or even demand human intervention, but the next wave of app teams won’t have to start from scratch on the basic tests every time, and can focus on the app-specific, and differentiating features of their app. AI-powered testing should finally free engineers from the mundane, and let them focus on the hard problems. That was the original promise of software test automation with procedural code, but the implementation proved to be less than adequate.

# Summary

We’ve explored just one approach to applying AI to software test automation. AI and machine learning are core technologies that can be applied in a near infinite array of ways to many different testing problems. Unsupervised learning is applied at Concur to automatically identify whether servers are acting oddly. Ultimate Software is using AI to generate additional test cases by learning from existing test cases. King (of Candy Crush fame) uses AI to automatically test new level designs. Many testing vendors are right now figuring out how to integrate AI because they don’t want to be left behind. Others are so motivated that they claim to be using AI before they know what it means. All this is evidence that AI will transform software testing whether we like it or not.

AI for software test automation is real, it's here and it is running on real world apps today. Many testers will be in denial. Many testers are intimidated or confused by “AI”. Many testers ask if their automation or manual jobs are in peril. Many will say, it can’t perform this edge case, or that they are too invested in the procedural testing world to change. By analogy, motor cars were initially noisier, more complex, more expensive, couldn’t travel well on muddy roads, needed gas, even ran over humans, but you don’t see many horses around town anymore. AI is just as transformational a technology for the testing profession as motor cars were to transportation. We are in the early days of figuring out how to apply AI to testing, but the revolution has begun. Regardless of what people want to believe, AI for software testing is here today and it will rapidly transform what we know as test automation in the coming years.

1. Twitter, Andrew Ng, https://twitter.com/AndrewYNg/status/788548053745569792 [↑](#footnote-ref-0)
2. Hacker News, Aslak Hellesoy, https://news.ycombinator.com/item?id=10194242 [↑](#footnote-ref-1)
3. Gherkin, Wikipedia, https://en.wikipedia.org/wiki/Cucumber\_(software)#Gherkin\_language [↑](#footnote-ref-2)
4. New Kids On the Block, “Step By Step”, https://www.youtube.com/watch?v=ay6GjmiJTPM [↑](#footnote-ref-3)
5. Google DeepMind Atari Reinforcement Learning, https://deepmind.com/research/publications/playing-atari-deep-reinforcement-learning/ [↑](#footnote-ref-4)
6. Google Link Checker, https://developers.google.com/adwords/scripts/docs/solutions/link-checker [↑](#footnote-ref-5)