[00:00:00] **James Walker** Hi, everyone, and welcome to our talk Stop Overtesting Taming the Combinatorial Explosion. My name is James Walker and I'm the co-founder and CTO of Curiosity Software,

[00:00:14] **Marcus Merrell** I'm Marcus Merrell, I'm the Vice President of Technology Strategy for SauceLabs. And today, we want to talk about Taming the Combinatorial Explosion. This is something that I think everyone's familiar with, even though you may not use the term combinatorial explosion, you'll recognize what happens when you make a single change to a system and sort of like a chaos theory of the butterfly wings that has ripple effects that you never intended and never anticipated all over the software. I'm reminded of one e-commerce company that I worked for several years ago, where there is a list of voucher codes or coupon codes on a page related to a single store like Kohl's or Macy's, some sort of a store that you would order online goods from. And there are deals and you click a coupon code. It drops an attribution cookie, opens up a new tab in a new browser, and then you are allowed to shop. And once you shop, commission gets paid to the e-commerce company because we led to the conversion of the sale for the e-commerce site. And that was great and people thought, Oh, it's simple, I click on a coupon and it works. Well, it turns out that every kind of coupon had its own sort of category. These categories had different keywords. These vouchers could be listed in the browser in a different way in different ranks at different sections of the page on the rails, in the middle, on the top, on the bottom, and the carousel. And then there were links that were from text, links from images, and links from the coupon codes themselves that we would use, leading to something like 2400 different kinds of code pass that you could get through. You could use to get through and convert your coupon. And this led to the kinds of discoveries that you don't ever want to hear about where you find out that if a coupon is in the fourth position in the carousel on a Tuesday when you're using Safari two versions old and people haven't updated their code pass at all or they're their versions, then no one gets attributed, no one gets converted, and no one really understands what happened at all, because that code patch just happened to be broken for one reason or another. And is it reasonable to cover all 2400 code pass? Well, that's what we're going to talk about today. I'll turn it back over to James to talk more about model-based testing.

[00:02:31] **James Walker** Great. So I think at the heart of that story is the software can be pretty complex, with lots of changes coming from many different places. Change can come from edits to code by developers. It may be changes made to requirements by business unless user behaviors also shift constantly, including the devices and environments they use. Third parties might also change the systems that integrate with your own, along with external laws or standards that might also impact. Lastly, business priorities often evolve while testing itself might uncover new information like bugs, which lead to further change requests. Some of these changes are largely under an organization's control, but others can be external and are often unforeseen. Testing plays a vital role in de-risking the possible impact of rapid changes. So generally, the tighter integration between these components and the more that coupled together, such as the requirements, the code, the data, the environment, and the test cases, the faster we can develop and also the higher quality we can achieve. And we also have a greater chance of being able to make sense of what's actually going on. I think, unfortunately, in the real world, the reality is these artifacts are often very far apart, and our understanding of what's going on is often out of date. The challenges many tools manage the estate. This can be fine for small teams where we're all talking to each other all the time. But if you start moving into much larger enterprise software development, this makes it difficult to have a clear, coherent understanding of how everything is connected and have confidence in our testing strategy. So these are kind of a few questions the typical stakeholders may ask as the project is ongoing. So do you understand all of your data and user environments? Do you have all the data and environments to develop and test the requirement? Do you know all the code affected by a requirement? If you change your code, which requirements need to be updated? Does the test have the data and environments needed to run? Do you know which tests cover which environments? Do you test, test the requirement? All the tests are linked to requirements. Do you have all the data to develop with? If the data characteristics change, do you know which code to change based off of that? If you think about it, these are actually really difficult questions to answer without having one crucial component. And that is traceability. As Marcus mentioned, the beginning, there is more than we could ever test exhaustively in a sprint across all of these combinations. A typical system can contain millions of interrelated nodes. Most enterprises also consist now of a hybrid of state. They may have some mainframe monoliths, they may have some modern microservices, and a lot of organizations are now looking at serverless offerings. We also now live in the world of the API. There are lots of third-party systems that are all software is consuming. As soon as we add third-party tooling with exponentially growing the complexity of the integrated nodes and components that we're operating with. There's also an extremely diverse range of users, devices, environments, and also geographies, which our customers are using to access our systems. As we said, we obviously can't test every change exhaustively. When a change occurs, there's simply too much to test. So to deliver value, we must prioritize based on the risk and impact of a change.

[00:06:31] **Marcus Merrell** So the thing that's been emerging throughout the talks so far has been how to balance risk with coverage. Now, you always want to add coverage, right, because there are all these paths you have to cover, but first of all, it's very, very difficult, if not impossible, to find out what your coverage actually is, especially when you consider interrelated nodes as well as external systems. It's impossible to measure your coverage. It's also next to impossible to know what your coverage target even could be. The better bet is to start talking about risk to business and risk to the goals of and the revenue of the business itself. What you want to know is what's the most important thing I have to test to make sure that money still flows into my company? Or that the most important metric that I care about is even if it's not revenue, still operates properly. And then you spiderweb out from there to things of less and less and less and less important. I once worked on a project where we wrote hundreds and hundreds of tests for an area of the product that fewer than .5 percent of any users actually ever visited. It wasn't important. We shouldn't have spent time on it. How do we begin to approach the problem of what coverage is important to the risk to our business? It can be broken into sort of three dimensions of coverage. We look at system logic and data combinations. What percentage of the system's logic is being tested? Where is this being focused? Like, how do we know we're focused on the right area of the software? The most critical path for that the user could take based on user journeys, based on analytics, based on actual behaviors in production? And then what's the device mix, whether it be web browsers or mobile devices, in what environments and in what configurations have the tests already been run in? And what am I missing from that data? And that's not just something you need to gather internally, but you need to look at the market as a whole and find out where your customer is coming from and where all could they be coming from? If you expand into a new market? Then you have to look at system tiers and architecture and find out how are the different user interfaces API is back in tests? How are these systems making up what you're doing today and how are they factoring into your decisions about what you're testing and when? We've gathered some market data, and we show that in the US. It's we see spikes now and then from Apple, from Samsung, as the market makers do a tug of war between themselves. And in general, we see that if you want to guarantee a certain amount of coverage roughly if you follow this chart, if you do Apple and Samsung, roughly 70 percent of your users are going to be covered. And you can be confident in that doesn't mean you shouldn't test it all with Lenovo or OnePlus or others. But it does mean that you have pretty high confidence that you're covering your largest user base. Now, suppose you're U.S. only and then say the e-commerce company in this actually happened at the one I worked at. We bought a company in Germany and then we bought a company in the UK. Well, suddenly you need to start paying attention to the entire world of market share and market and devices in the market and what your customers are doing. And suddenly, the picture changes. Apple and Samsung now only give you around 40 percent of the entire market share in order to be able to get confidence in your users. You now need to be able to test in Xiaomi, Vivo, and Oppo devices. And that doesn't even get you to the 70 percent. You've still got Huawei. You've got all sorts of other manufacturers that are in the market that you need to be able to cover in order to get market share. Now, you can get a lot in general because most of these devices run Android. Android operating system. You can run Android tests. But there again, there's a combinatorial explosion because Android has, Android itself has devices that support five or six different versions of the operating system at different times, and each one of them is kind of on their own branch with certain proprietary and certain, you know, branched off areas of functionality and operating systems that you can't always covered for and easily. So even covering Android itself can sometimes add four or five different combinations to this. So what we're trying to go for is explaining to people what is out there in the market and what could be possible and then how to do the smallest amount of work that covers the most amount of platforms that will get you to some area of confidence that you can say, I've at least covered this much. Because the thing to remember about smartphones in this may seem obvious, but you didn't write any of the code that's being executed in that smartphone. It's a variable that's a black box that you can't control. You can't even impact in any way. So that variable needs to be factored into your risk profile so that you can understand exactly what the risk is to you to your business.

[00:11:49] **James Walker** Okay, we're now going to start moving into a hands-on demo. We're going to be introducing a few approaches we believe can help with some of these challenges we've been talking about so far. What approach we're going to show at the beginning is introducing the idea of modeling. This is where we can model out the different pieces of functionality of an application, and we can start to auto-generate tests based on different risk profiles, which are going to test different aspects of the system that we care about. We can then also generate test scripts into the appropriate automation frameworks to execute these tests as automated test cases. We're not going to execute tests in these frameworks using SauceLabs to test across a range of devices that we express within the model. Now, what we're going to see is the often tests will fail, and there could be many reasons for that test actually failing. Was it an environment problem? Was it a test data problem? Was it actually a false negative? Was something wrong with the system and it was actually a defect? So what we're going to show is the ability to go in and pinpoint a particular failure to go off and explore that failure further to generate more test cases dynamically and gather more data to help us understand why that failure has occurred and what the impact of it was. Now, everything we're going to generate, execute, or import is going to revolve around this concept of a traceability lab, which is what we're introducing. Now, a traceability lab uses a graph database to pull changing information from the software development lifecycle. Okay, it's going to create a central network applying analysis techniques to track connectivity between different artifacts like the requirements, the test cases, and the code. Okay. And what we're going to be doing is we're going to be querying that graph to take a look to understand what the impact of the changes and also understand how all these different components are related in our software development lifecycle? Now, this is very similar technology to what companies like Netflix use to make recommendations based on your personalization and what kind of shows you're watching. We're just applying the same technology and looking at how we can use that within the software development landscape. So if you think about this in the context of everything we've discussed, a requirement code, a test. These are all notes with connections between them. And we're just going to be querying this graph to give us all of the data that we care about for the particular challenges that we're investigating. We're now going to jump in and take a look at the actual demo.

[00:14:47] **James Walker** So we are in an e-commerce system which is called Shopizer. And within here we can order different products. We can go to different pages. We can register new users. We can sign in as new users. And we can also go through the shopping cart and checkout process. Now, if you think about this application is actually many different steps, you can go through many different scenarios you can walk through to achieve specific tasks. This can end up being a huge amount of complexity. Now, what we have done is we have moved into the world of modeling and we have modeled out some of the subcomponents that exist. And we have chained them together to create some end-to-end scenarios. Now, modeling can be done in Visio, It can be done in BPM, and it can be done in any diagraming-based tool. And there is also specific model-based testing tools, which is what we're going to be looking at here, which can be used to generate models and generate tests from them. There were a few Open-Source ones out there, like Graph Walker, and also a few commercial tools that were available. Now what we've done here is if we come in and take a look, we have modeled out different screens. This represents the process of registering a new user and logging in. We can expand these and we can see all the different process that can take place for registering a new user, entering first names, last names, passwords, email addresses, and at the end we end up successfully registering a new account or some kind of error message is displayed to the user, and then we either end or go through to our next process. Now, the interesting piece about this application is it can be used across several different environments which are supported. These are different web-based browsers and different mobile devices. What we've done here is we have created a model which represents the different platforms, the different browsers, and the different environments that our application is supported. And this model represents all the different configurations and all the different permutations that exist throughout. If we come in here and we do it, generate, what we'll see are all the different devices and different web browsers. We are going to test across and we can come in here and we can do an export and we can look at the different test scenarios that it's producing. You open this up, you'll see the different device mixes so within these columns. Now, we've connected this up to a much larger end-to-end flow here. And what this allows us to do is to come in and create different coverage profiles. The first profile we've generated is for all devices. We're going to come in and we're going to play a medium level of coverage across the whole model. It's going to cover every edge and we're going to exhaustively test all the different devices that exist. This ends up with us having 286 different combinations across our application. We now have 286 automated tests we can go through to test all these different combinations of devices. We'll see as we're flicking through here, the different permutations and combinations that are being tested by each of these scripts that are coming through. The next one that we have done is we have created a mix. We are going to cover every single device once throughout the rest of our scenarios, we're going to have a medium level of coverage which is going to cover every edge. This leads to us going from 286 parts down to 20. We can see here as we're going through the different combinations that are being tested throughout each path. This is basically gone down and created the minimum number of permutations across environments to maximize our coverage throughout the rest of the functional scenarios we're testing. The final one we've created here is to have a focal point on testing Android platforms. We're going to test every permutation of Android environment possible and we'll see now we have 76 test cases, which are available. Well, this allows us to do is to create different risk profiles focusing on different environments, but being able to optimize our coverage or the scenarios that we want to test. And once we're ready, we can come in and we can run these tests so we can go off and execute them. And what this will do is it will go off into our cloud environment in this case, SauceLabs. It will head off and it will start executing those tests across the different environments that we have specified. And this can also be done in parallel to speed things up. And we've got some results back here and we'll see. We've got some parties that have come free. We've also got a couple of failures that have come in here for specific platforms that have been executed across. If we have any failures, what we can do is we can come in and we can perform something called pinpoint analysis. And when this enables us to do is to focus and create a whole new test suite revolving around any failed test cases that have occurred. And this could be focusing on specific environments which may have impacted that failure, or it may also be specific functionality that has caused a failure to occur. This is all about gathering more information to understand where a failure has occurred and why it may have occurred. This produces 17 paths that we can now go off and test further to identify the specific cause of a failure. Now, a big challenge in the software development lifecycle is traceability. Now, whenever anything is generated from the model or whenever something used in an external system like a requirements management tool or a test case management system, those changes are tracked inside a graph database. Now, this is some inside Neo4j, which is a very popular graph database engine. And what we have done in here is we have connected Neo4j into tools like Jira into tools like GitHub to xray for our test case management. We can see here on the left, we have different nodes and we can query them so we can see the different environments we're testing against. We can see the different people involved in our project. We can get a subset of some of the requirements that are taking place and we can go into look at some of the commits and some of the files as well. Now the great thing about a graph database is it's all about connectivity. The first thing we're going to query for is to find all commits that have a link to requirements. Where are you going to return 25 here because we have quite a large volume of data. We've been tracking this is going to show us here are some of the requirements we have in Jira, some of the user stories, and some of the commits that have been associated with them. Now, this is similar technology to what companies like Netflix use to go and make relationships between different programs and make recommendations based off of your personalization. Here, we're using the same technology to track links between different artifacts in the software development lifecycle and to allow us to see in this case which Jira tickets are linked to, which commits. We're going to apply another query here, which is a little bit more sophisticated. And what this is going to do is it's going to find all the commits which have requirements linked to them and all the test cases, which therefore link that specific new commit of code that's been made. So we'll do a query here on an experiment, and we'll take a look now in the center here. We have our requirement, which is for withdrawing money based off of this, we have a few commits that have been made in our source code repository, which linked this requirement. And then based off of this, we have free test cases which are going to go and test the specific requirement. So this is telling us for this, these new commits that have been made, we have free test cases which can go up and specifically test the new code, which has been added. Now we can get very sophisticated with some of these queries we're casting to answer some questions we may have. This is a much larger query that we're going to cast here. And will this is going to look for is new commits that have been made and we're going to look specifically for people who have made those commit to do not have experience with those files that have been edited. This is going to come back with a table. It's going to show us the file names of the codes that have been edited, the authors, and how many times they have edited that file before. And you'll see here code files and the different amounts that those specific developers have edited them. We can go much, much more sophisticated here, and we can start casting all sorts of queries for different questions we may want to answer in our software development lifecycle. And what we have done is we have built out some specific dashboards for the Traceability Lab, which sit on top of our graph database, and these can be used for looking at particular requirements, commits test cases, people or just exploring the data and expanding it out. And what we can see here is we are looking at a specific requirement that has been created for withdrawing money from an ATM machine. Now, some quality checks have been applied, some specific queries in the graph database to go and look for things that might be good or bad about this requirement that's being created. The requirement has an assignee. It has a linked model. It has some link test cases it has a sentiment score, which is positive. The length is above a certain threshold. And also we've looked at something called Scope Master to perform an analysis of that requirement. This has gone off and performed thousands of checks against the requirement to look for things like ambiguities, to look at the objects, and to look at the natural language within that requirement to look for things that may be potentially missing. Scope Master has come out with a score of 93 percent, with one bad and two advisory results. We can see down the bottom here some of the connectivity, some of the traceable elements that have been created here we have for subtasks, we have free commits that have been made for this requirement, we have for specifically linked test cases and for indirectly associated tests. And we've also got two models that have been linked here, which have been used to define this requirement. We're not going to jump over and look at the commit here that has been made. Now the commit is above a certain threshold, 193 code changes. We'll see here that we have that linked requirement from earlier, which has been connected using the graph, we have free source control files that have been edited and we know for this specific commit the four test cases that are going to be used to test it. We can if we want to come off and generate a new test suite here. And these are the tests which are going to pinpoint and specifically focus on testing any code that has been directly or indirectly modified, which is going to directly test any new code which has been added or changed. We can also see down here to directly impacted functionality. Here is our user story, which relates to this specific commit. We've also got four impacted commits. These are commits that are joined together using the source code that we have available. These are overlapping commits that may not directly impact the functionality. There are shared classes and shared functions that have been edited, which overlap this functionality that has been edited.

[00:25:38] **James Walker** I think in summary, what we can see today is that testing is really hard. Automation is obviously even harder and it's impossible to test everything that's happening, let alone automate it. There are more combinations of things going on in our software development landscape than there are stars in the universe. But hopefully what you can see in our session are some techniques and concepts and data points which can help reduce and understand the complexity, increase understanding and hopefully help you test more intelligently to tame the combinatorial explosion. So I'm just going to finish up by asking Marcus a quick little question. Marcus, from everything we've seen today, what do you think is the key point people can take away to get started?

[00:26:30] **Marcus Merrell** I think really the key point is that there are a couple of primary workflows that go through the software that you're responsible for. And those primary workflows are the main central nervous system of how revenue or whatever metric that's important flows through your system. And the truth is that I've had a lot of CTOs ask me to guarantee that I've covered 100 percent of the code in the software and now there's 100 percent chance. That I have personally in my career, my 20-year career in QA, I have let really large important defects go in those most core workflows because I spent a lot of time making sure that I went from 96 to 97 percent coverage in an area the product that nobody cares about. The truth is that if you spend your time focus on those main workflows, then the little things that you're potentially letting go of in the smaller areas of code are probably there less important. You need to make sure that those main flows are covered. And I guess the way I put it is I'd rather have 40 good solid tests that run on every single commit. 10 to 20 times a day, then a suite of 400 unwieldy, hard-to-maintain scripts that I only run once a day or once a week. That's the main takeaway for me.

[00:27:59] **James Walker** Amazing. Couldn't agree more. Thank you so much, everyone, for watching this session. And I think now we're happy to open up the questions.