SERVICE VIRTUALIZATION

Best Practices

Insights from Industry Leaders

PARASOFT®

Table of Contents

Preface: Accelerate Releases and Reduce Business Risk 1

Service Virtualization, Agile, DevOps, and Quality @ Speed 2

Simulating Snowstorms in July with Service Virtualization and Extreme Automation 12

How Service Virtualization Enables Continuous Delivery 19

Service Virtualization, Performance Testing, and DevOps 24

Service Virtualization is Essential for Functional Testing of Complex, Distributed Systems 26

We Couldn't Do Agile Development Without Service Virtualization 29

Time to Market...with Quality 31

Service Virtualization with API Testing 34

Bridging the Technical Gap for Executive Buy-In 36

A Love Letter from Cloud Dev/Test Labs to Service Virtualization: "You Complete Me" 38

How Service Virtualization Impacts Business Strategy 40

About Parasoft 43

Preface: Accelerate Releases and Reduce Business Risk

Wayne Ariola, Chief Strategy Officer at Parasoft

The concept of leveraging a simulated test environment to "shift left" quality efforts has recently gained much attention among large enterprise IT organizations. The idea of simulating interactions with an application under test certainly isn't new; organizations have been using a stub strategy for quite some time in order to isolate tests from dependent environments. This stubbing approach allows independent tests to be executed, but it certainly introduces a level of risk since the functionality being stubbed is truly at the discretion of each independent developer. It's like asking an assortment of independent contractors to all contribute and patch together different sections of a bridge rather than having an engineering company plan and coordinate the effort.

Applying a unified service virtualization initiative has proven to be a powerful tool for assisting organizations to accelerate the SDLC:

- An organization has a single version of the truth, removing the risks created by having independent brittle stubs. Service virtualization introduces an environment-based approach, allowing the entire organization to access common artifacts that represent critical functionality.
- Service virtualization allows for much more complete tests to be executed earlier in each iteration, helping the organization discover application or business risks much earlier.
- Service virtualization, in conjunction with hypervisor technologies and cloud, have solved the nagging issue associated with test environment access and control, allowing an organization to truly remove the constraints associated with testing and accelerate an application's release cycle.

Over the past years, Parasoft has been heavily engaged with assisting Global 2000 companies to adopt service virtualization technology. Reflecting back upon on our communications with service virtualization adopters and evangelists, we have selected a handful of stories, viewpoints, and advice that we believe will help organizations make the most of service virtualization technology. The contributions in this "living document" span topics ranging from gaining executive buy-in for service virtualization to implementing service virtualization in ways that enable more agile testing processes. We hope that this collection provides valuable insights for executives, managers, and users who are looking to either adopt or expand their service virtualization footprint.

Service Virtualization, Agile, DevOps, and Quality @ Speed

Diego Lo Giudice (Forrester's VP, Principal Analyst) and Mike Puntumapanitch (CareFirst's Director of Service Transition and Quality Management)

Diego and Mike P. recently connected to explore experiences, best practices, and trends on service virtualization, Agile, and DevOps—particularly in relation to how these topics enable "quality @ speed." Here are some highlights from the discussion...

Accelerating Speed without Compromising on Quality

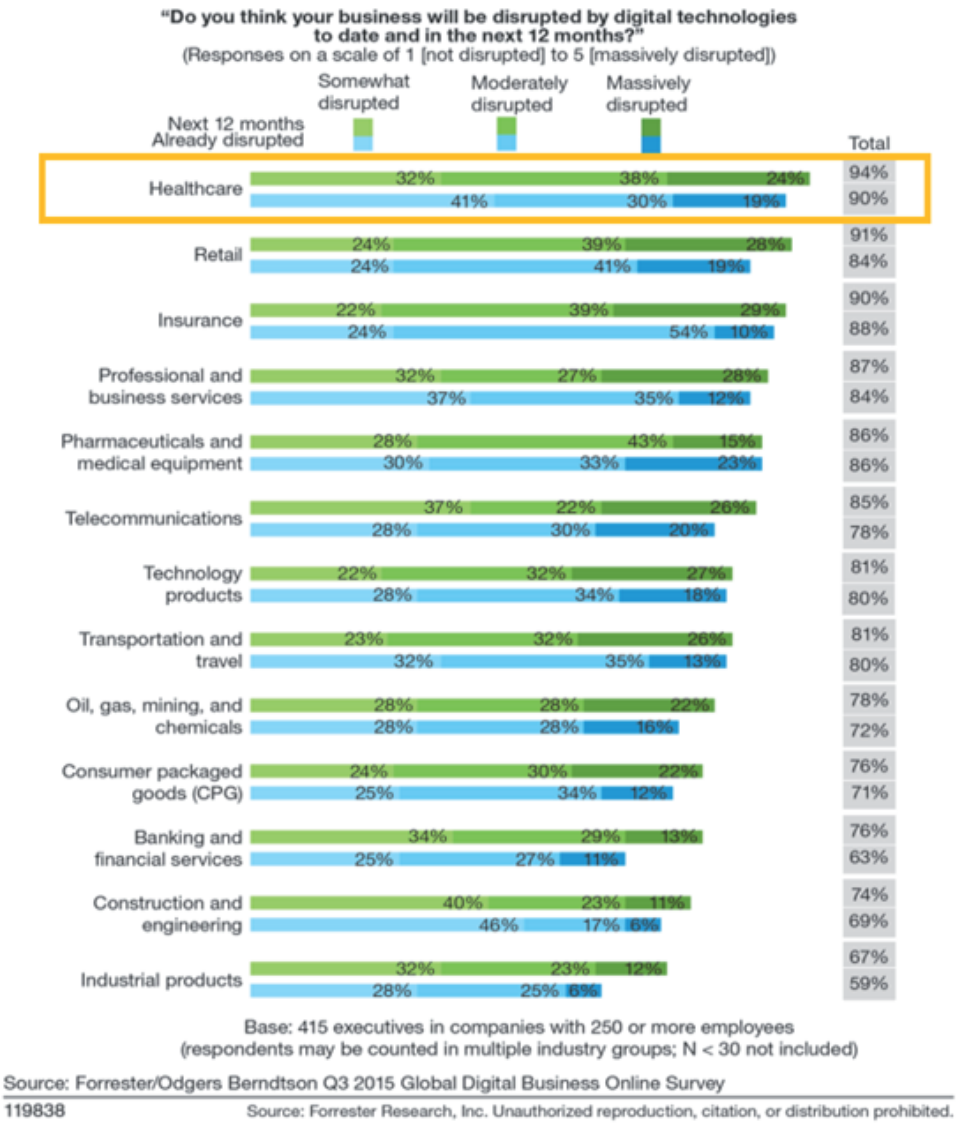
DIEGO: Across all industries today, software really matters. Whether you're a technology startup, a bank, insurance, retail, etc., you must be excellent at developing, testing, and delivering software. That's why we keep talking about Agile, DevOps, Continuous Delivery, and Continuous Integration. If we want to make our business more successful, software is absolutely crucial. It's not just about speed—it's about speed with quality.

The old school of testing is that the more testing you do (if you do it right), the better your quality. But if testing is an afterthought (as it is in many organizations), it occurs between development and deployment—and it feels like something that will stop you from delivering faster. Historically, the business says, "It doesn't matter, just deploy, it's the date that's important." But these days, that's changing because a bug going into production in the age of digital is a real big problem. It will hit the news.

MIKE P: CareFirst's executive team is very insistent that we continuously strive for, and achieve, excellence—both in our internal operations and all our interactions with customers. This places a lot of pressure on us, especially as we're also asked to deliver more outcomes—faster and cheaper. We complete about 160 projects a year at a very aggressive pace and over 90% of those are delivered on time and on budget. Compromising on quality to meet these deadlines is not an option, so we had to determine how to make IT more effective. As I'll explain a little later, service virtualization was a large part of this.

Reducing Complexity with Service Virtualization

DIEGO: As Forrester's research shows, no industry is immune to digital disruption and the complexity it brings—including industries that are not typically perceived as being aggressive in innovation, like construction and industrial products. Even in those industries, the overwhelming majority of companies are reporting that they have already experienced digital disruption and/or anticipate digital disruption in the next 12 months. And, what's



probably not too surprising to Mike, healthcare is at the top of the "digital disruption" list. 90% of healthcare IT executives reported that their business is currently facing digital disruption and 94% expect to experience digital disruption within the next 12 months.

MIKE P: Yes, we've definitely experienced this firsthand—since 2008, in fact. The challenges we've faced can be grouped into three general waves.

With the realization that the health care industry would need to take giant leaps towards modernization, the first wave can be best characterized as business transformation. There was an executive-driven initiative to modernize our IT, perform systems consolidation, align ourselves to new market segments, grow/retain business, measure ourselves differently (with metrics that matter)—all while keeping the lights on. This was a two year journey. In retrospect, I believe this helped us lay the foundation to become more agile.

In the next waves, the Affordable Care Act ("Obamacare") was the primary impetus for change. From a business perspective, it presented us with the opportunity to ramp up innovation. To meet the demands of the Affordable Care Act, we needed to reduce the complexity of internal systems, sunset legacy systems, globalize our IT teams, and essentially leverage technology to do more with less.

The next—but certainly not final—wave revolved around achieving continuous improvement with a member-driven approach. With the proliferation of smartphones, we anticipated that members would want to interact with us using their mobile devices, so we launched the "mobile first" initiative.

The following table sums up these three recent waves of disruption:

Phase	IT	Business	Financial	Industry
Business Transformation	IT modernization System consolidation	Business transformation Grow/retain the business Align to market segmentation Metrics that matter	Keep the lights on	
Ramping up Innovation	Virtualization adoption IT globalization Legacy sunset	Industry disruptive events Innovation-driven transformation	Do more with less	Reduce complexity
Continuous improvement with a member-driven approach	24/7 delivery	Attain/maintain excellence	Provide cost transparency Bend the cost curve Deliver MORE outcomes, better, FASTER, cheaper	Strategic thinking

Responding to Complexity with Service Virtualization

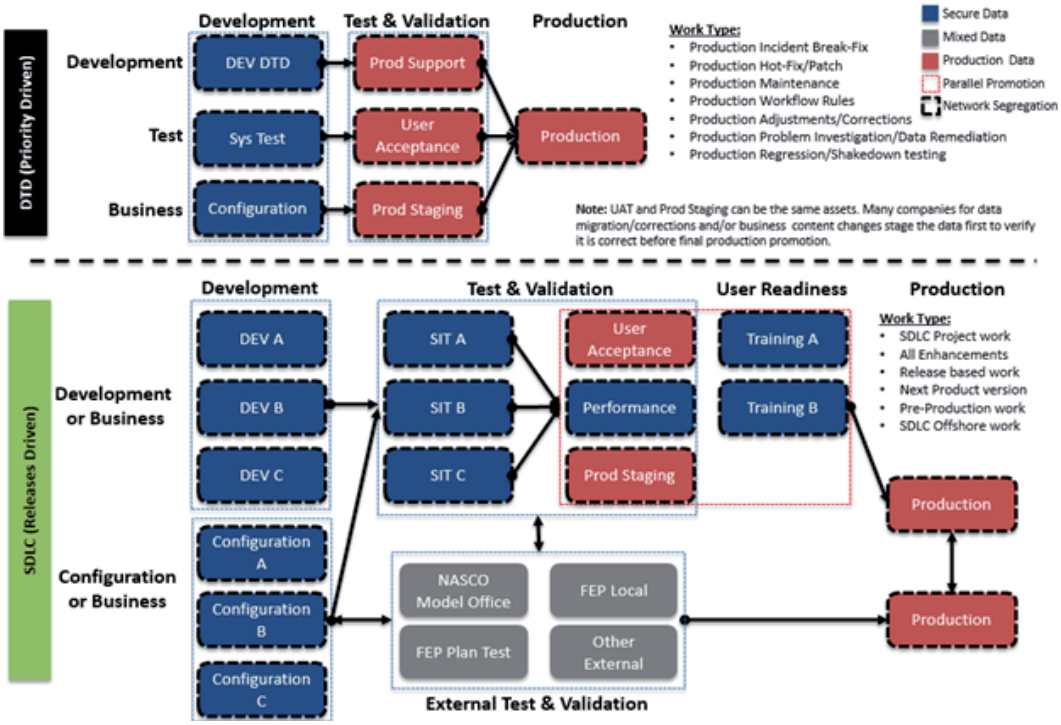
MIKE P: In response to all this disruption, we optimized many parts of our process to move faster. However the complexity of our technical architecture was slowing us down and holding us back; it made it nearly impossible to rapidly provide DevTest teams the test environments they needed for a continuously-evolving SDLC. Each environment is extremely complex to reproduce in a test environment. There are a lot of integrations, external dependencies, configuration steps, and so on...and this makes environments very expensive to create and maintain.

DIEGO: This same struggle reaches across all industries. The difficulty of testing effectively given today's complex environments is one of the largest barriers to making IT more effective. There's a formidable number of dependencies to deal with. For example, ING has 150 scrum teams and PayPal has 680 scrum teams around the world. Just imagine all the dependencies to contend with there.

To get around this complexity, you need to start simulating things—for example, if you don't have access to a mainframe that you need for testing, or if you're testing against a third-party service. Service virtualization can allow you to test and integrate on a continuous basis. That's how you start doing integration testing from the beginning. It's introducing simulation into the development process. What an idea! This approach is used in many other industries, but we haven't really used it in the software industry. I like to think about this as a wind tunnel: you put the airplane in it and test the airplane by simulating all the conditions around it. That's what we need to do—and service virtualization helps you get there.

I've found that the organizations with the most complex systems, the ones with the most legacy systems and the most applications to integrate, tend to be the ones most likely to invest in service virtualization.

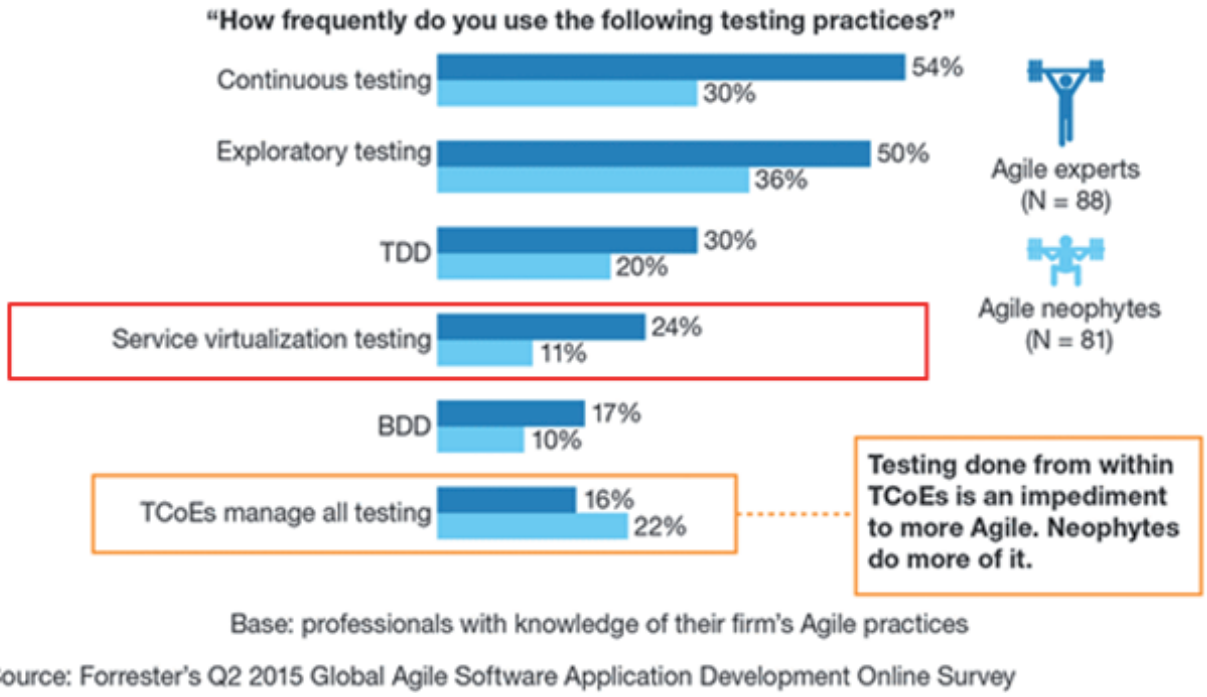
MIKE P: That's exactly what happened at CareFirst. To understand the scope of the complexity we're facing, consider the following diagram, which describes the promotion of changes by work type:



As you can imagine, it's just not feasible to physically provision all the environments that DevTest teams need, whenever they need them. That's why we turned to service virtualization.

How Service Virtualization Enables Quality@Speed

DIEGO: Many leading organizations across all industries use service virtualization to help achieve quality @ speed. Our research at Forrester found that "Agile experts" use service virtualization twice as much as "Agile neophytes." The "Agile experts" also are doing more continuous testing, exploratory testing, and TDD—and they rely less and less on traditional Testing Centers of Excellence (TCoEs), where all testing is executed far away from the team and not integrated into the dev team and dev process.



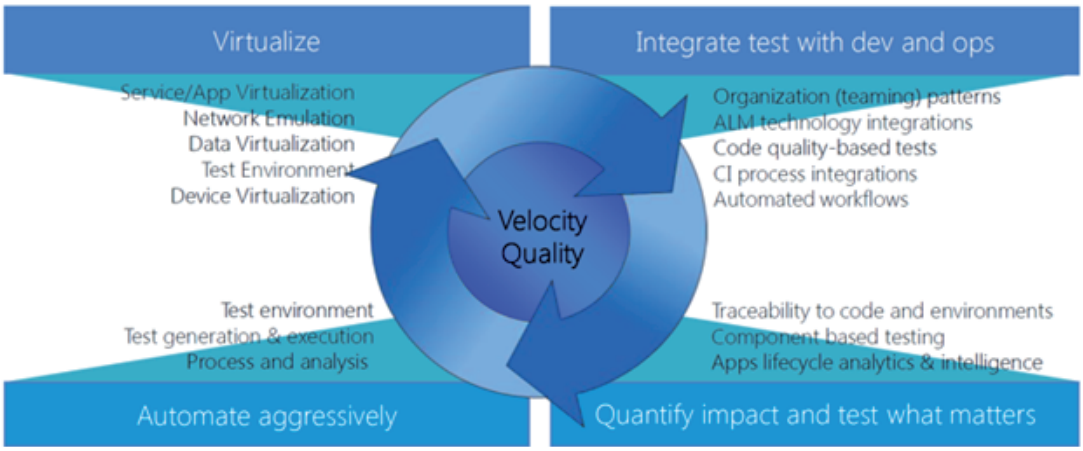
Base: professionals with knowledge of their firm's Agile practices

Source: Forrester's Q2 2015 Global Agile Software Application Development Online Survey

Formula One pit stops offer a good analogy for how testing needs to change to enable quality at speed. In the past 10 years, pit stops have gone down from 8-9 seconds to 2.3 seconds. In 2.3 seconds, these guys change 4 tires and add just a bit of gasoline because if they don't have to load the car up with gasoline, it can go faster. They've turned pit stops from a necessary evil for keeping the cars running to a differentiating element of the race, part of the strategy for winning the race. I think that's how we need to think about testing. These Formula One pit stop guys also test continuously. Testing is happening from the very first moment when they start thinking about the design of the car, during the week as they make adjustments to the car, as they're driving the day before, and even during the race.

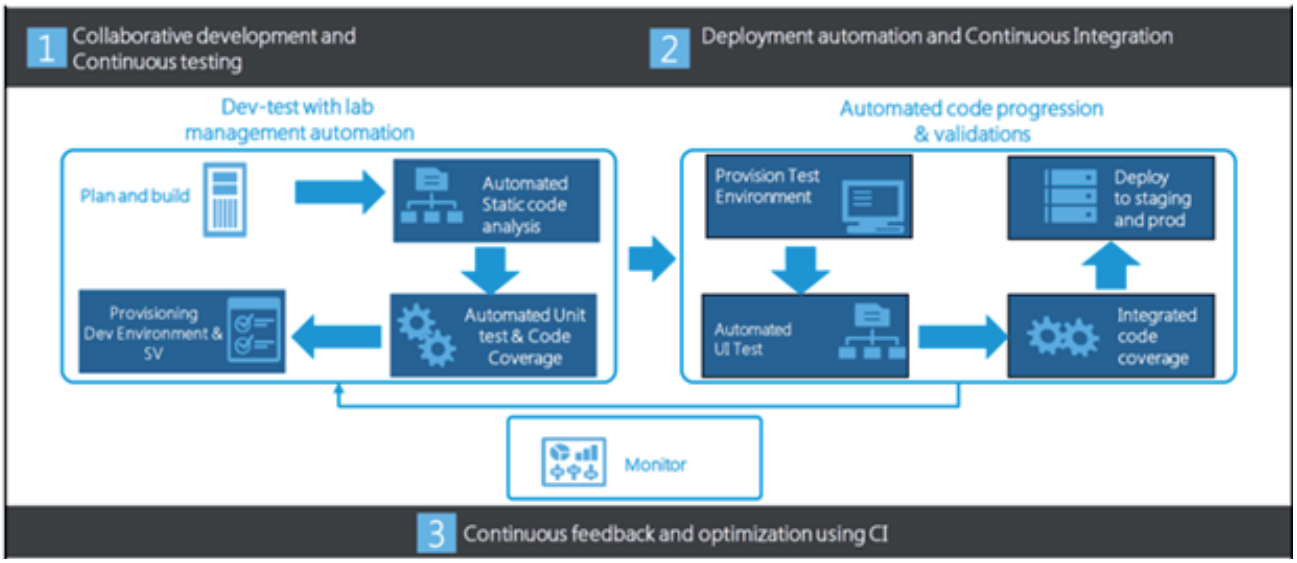
MIKE P: Exactly. In the spaces I control, we cannot allow quality to slip. I love the message of "quality at speed." We've been advocating that for years within our organization. I'm glad it resonates and I'm glad that Forrester is out there bringing that message to the world.

I've prepared a couple graphics with the help of Cognizant and Infosys to show our strategy for enabling our groups to move at the speed that the business requires. In the first graphic, you see the various strategies and technologies we're applying to implement quality at speed. Our goal is to "engineer quality in." This includes things like integrating test with development and operations, doing more reviews on requirements, and being able to test earlier and continuously thanks to service virtualization.



One of our main strategies for achieving quality at speed is to try to expose defects much earlier in the lifecycle—when it's easiest and cheapest to fix them. Service virtualization really helps us to do that.

Service virtualization allows Development to have an immediate response from a virtual service, which means they can discover unexpected behaviors (or defects) in the application sooner—before it's built and before it's shipped to an integration environment. This really helps Development tune their code very early in the process. They can get the code to the state where they're satisfied with it, then ship it to the testers, who can continue testing it in different ways.



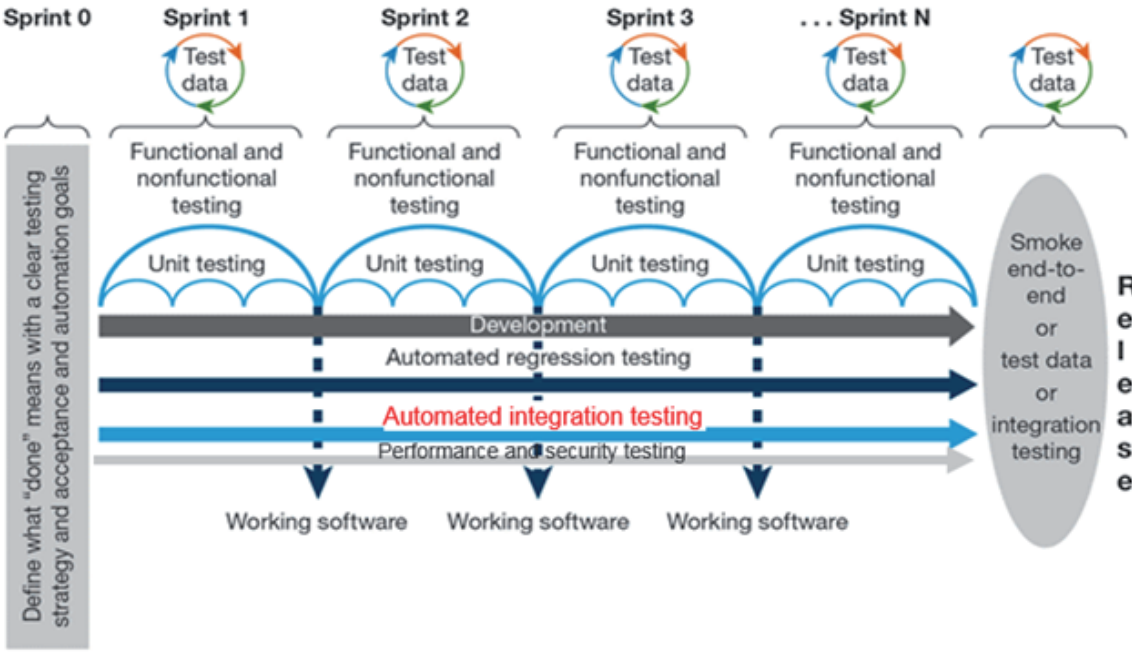
Service virtualization exposes issues before code reaches an integration environment, where it's very expensive to transact business. It's critical to have our test environment available 24/7. One developer adding bad code could bring down the test environment for everyone. When you have hundreds of geographically-distributed DevTest team members, this risk escalates. Service virtualization mitigates this risk by providing a simulated test environment; this allows us to automate more testing and enable automated promotions.

DIEGO: Yes, embedding quality in from the very beginning is definitely key for quality at speed. The most successful organizations are those that start testing early in the sprints, perform as many different types of testing as possible, and automate this testing as much as possible. This really lies at the heart of Agile and DevOps and at the heart of delivering better quality software.

Scaling Agile with Service Virtualization

DIEGO: We've found that dependencies among the various teams developing systems in parallel is one of the greatest obstacles to scaling Agile. For example, imagine that Team A is deploying software which needs to be tested against components delivered by Team B—but Team A is on a more aggressive schedule than Team B. You don't want Team A to have to wait on Team B in order to complete all the development and testing tasks that involve Team B's components.

Service virtualization decouples these dependencies so that teams don't reach deadlocks while waiting on one another. If Team B has descriptions of these dependencies… for example, Swagger…then those components' responses can be simulated for Team A. Once the actual component is available, they can be swapped in to replace the virtual assets.



MIKE P: Within our "Wagile" process (a hybrid of waterfall and Agile), we also use service virtualization to enable different roles on the team to work in parallel.

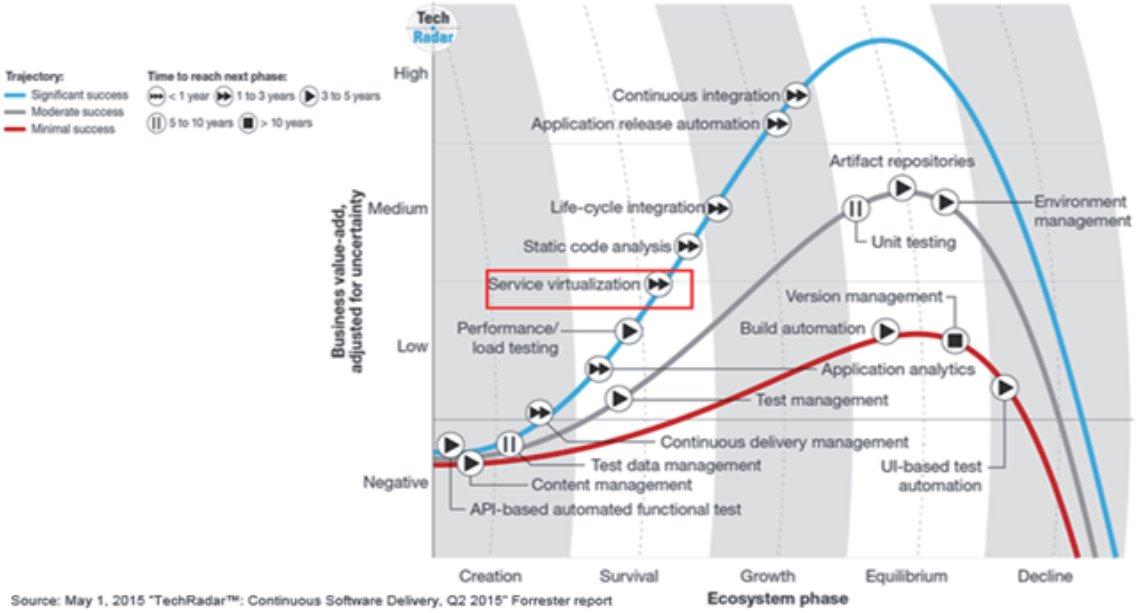
For example, before developers start coding in each iteration, they create service virtualization assets that simulate the anticipated behavior of the new components. And as they write and fine-tune their code, they run it against service virtualization assets representing the AUT's dependencies to check how their code works in the context of the complete system.

The developers share these service virtualization assets with testers. This allows the testers to start defining tests and checking their scripts even before the code is written. Tests are then redirected to the new functionality once it's implemented. Moreover, end-to-end tests leverage the service virtualization assets representing the AUT's dependencies. In this way, the same service virtualization assets that the developers use are also consumed by testers to define, check, and execute their end-to-end tests.

With service virtualization being used in concert with our automated build and deployments, potential agile/parallel development traffic jams are virtually eliminated and we are achieving a tremendous level of automation, which is vital for our Continuous Testing. The team's service virtualization and test assets become a permanent part of the inventory for build verification and regression testing—which can then be performed continuously so we're constantly checking that modifications don't introduce new risks or negatively impact the user experience.

What's Next with Service Virtualization?

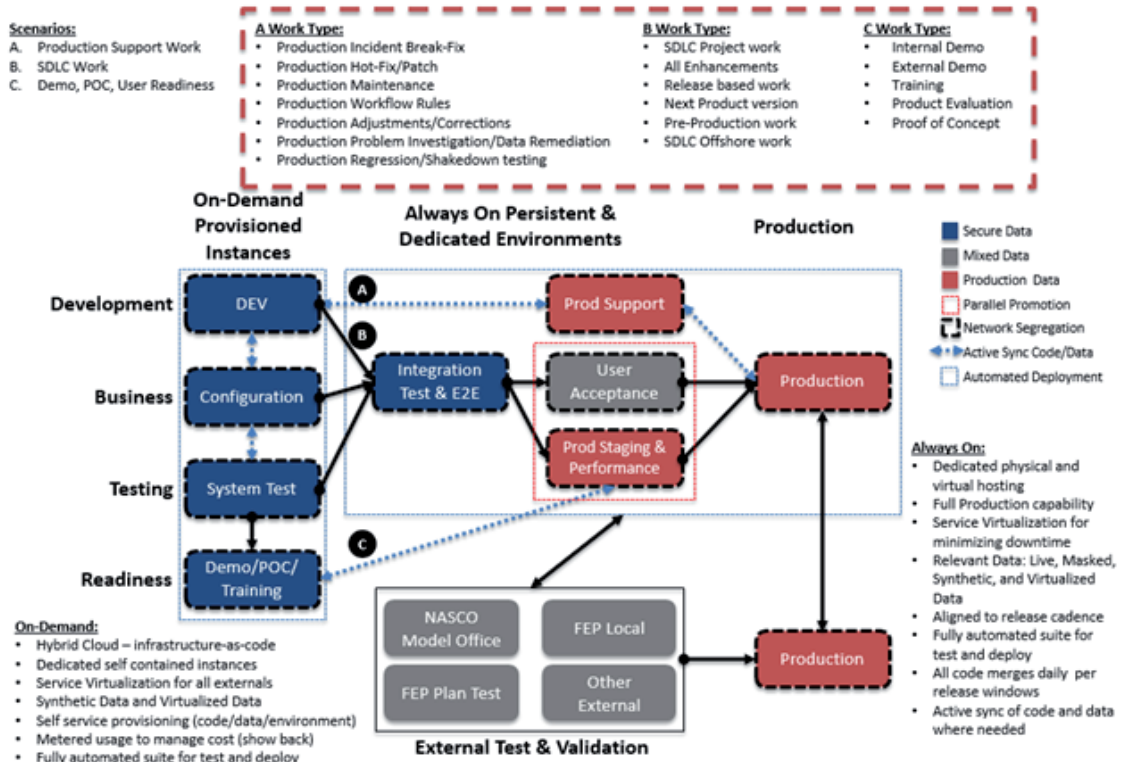
DIEGO: My research shows that service virtualization seems to be on a successful trajectory in terms of adoption. Technologies go through several different phases: creation, survival, growth, equilibrium, and then decline.



Source: May 1, 2015 "TechRadar™: Continuous Software Delivery, Q2 2015" Forrester report

Service virtualization has already passed the creation stage and is now in the survival stage, approaching the rapid growth stage. It should get there within the next 1-3 years. This isn't surprising given that it's really a mandatory technology for enabling organizations to deliver software faster and at a larger scale.

MIKE P: At CareFirst, we're currently working towards an infrastructure where any of our development, system integration, and performance testing environments can be provisioned on demand in a self-service model. This enables us to test much earlier, faster, and more completely; that's why we call this initiative our "testing superhighway." The technical nuances of this system are outlined in the following graphic:



With such an infrastructure, you can system test and do everything you need to do—including business prototyping—before integration.

Benefits of Service Virtualization

DIEGO: Based on Forrester's research, organizations tend to report that the greatest benefits gained from service virtualization adoption are:

- Improved time to deployment
- Business value related to quicker releases
- Revenue increases related to quality improvements in software and applications
- Reduced third-party testing and service fees

MIKE P: Several years ago, environment availability was the #3 problem being cited at our IT executive meetings—people talking about the time required to access environments and how this impacted the speed of development. Once we realized that we typically didn't need the physical thing, we just needed the response, we adopted service virtualization and we were able to respond to the business much faster. That's been huge for our ability to keep pace with digital disruption and save on costs so money can be better invested elsewhere.

I'd say that the most valuable benefits we've achieved so far with service virtualization are:

- Faster delivery speed
- Higher availability
- Accelerated automation
- Increased defect detection
- Increased control of environments

For example, with service virtualization providing simulated responses for our system functions, we can start running our automated tests as soon as we complete them—and with Parasoft's API Testing solution, the exact same tests can automatically switch from the virtual endpoints to the real ones (once they're ready/available). Now, the DevTest teams aren't delayed waiting for all the different dependencies to be completed and available in a test environment. Plus, we gain the 24/7 test environment availability that's critical for automated testing. With service virtualization working together with our automated build and deployment capacity—as well as our test data management solution which injects more comprehensive data sets into application to increase test coverage—we are able to bypass all sorts of roadblocks and delays. It's like having access to the FasTrak or Carpool lane versus having to endure rush hour traffic.

In addition to helping catch the defects that our Development teams introduce before they impact the integration environment, service virtualization also helps us mitigate the impact of disruptions stemming from the third-party services we rely on. For instance, disruptions in our middleware layer were causing us to have approximately 15,000 hours of unplanned down time each year—and about 60% of that is related to third-party web services. Previously, such disruptions would force us to curtail any DevTest activities that depended on these services. When these services fail, service virtualization automatically simulates the required behavior associated with those third-party services until the real service is back online again. DevTest work can continue as normal and continuous testing is not disrupted since automated regression test suites can execute as scheduled.

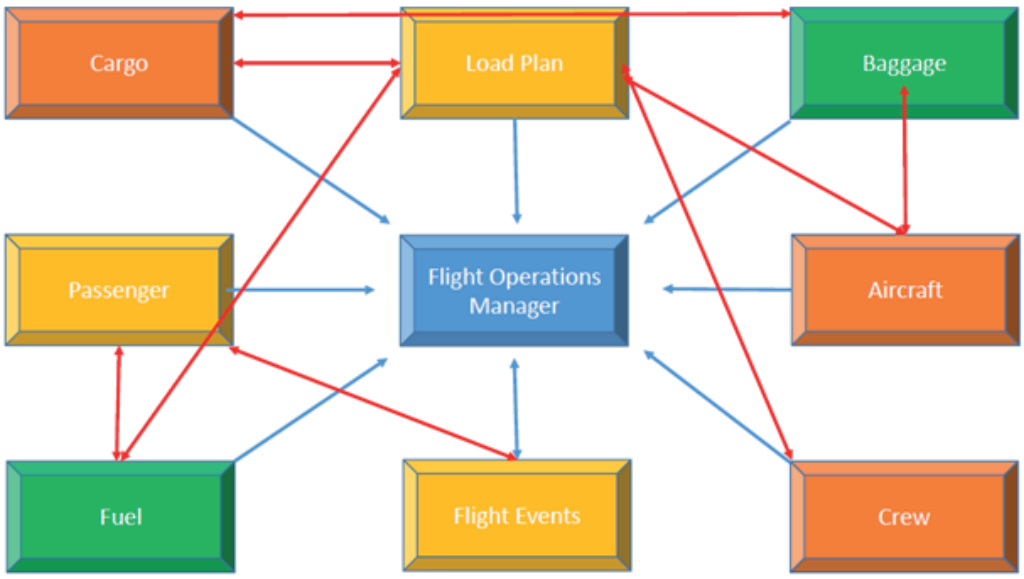
Basically, we've found that service virtualization really helps you do more with existing resources so you can reinvest that savings in yourself and you can improve the quality faster, earlier in the lifecycle.

Simulating Snowstorms in July with Service Virtualization and Extreme Automation

Ryan Papineau, Alaska Airlines Automated Test Engineer

How did Alaska Airlines receive J.D. Powers' "Highest in Customer Satisfaction Among Traditional Carriers" recognition for 9 years in a row, the "#1 On-Time Major North American Carrier" award for the last 5 years, and "Most Fuel-Efficient US Airline 2011-2015"? A large part of the credit belongs to their software testing team. Their industry-leading, proactive approach to disrupting the traditional software testing process ensures that testers can test—faster, earlier, and more completely. This selection details how Ryan and his team used advanced automation in concert with service virtualization to rigorously test their complex flight operations manager. The result: operations that run smoothly—even if they encounter a snowstorm in July...

At Alaska Airlines, the flight operations manager application is ultimately responsible for transporting 29 million customers to and from over 100 global destinations via 320,000 flights per year—safely and efficiently. As you might imagine, the application that drives all these operations is rather complex, and has an intricate web of dependencies. Consider the following diagram:



The application under test (AUT)—the flight operations manager—is at the middle, and all the boxes connected to it represent its core dependencies. These dependencies include:

- **Baggage:** A web service that provides details about baggage quantity, weight, size, shape, etc.
- **Cargo:** A similar web service that provides cargo details.
- **Crew:** A web service that provides crew details.
- **Fuel:** A web service that provides fuel details.
- **Load plan:** A back-end system that provides details on how the plane should be loaded.
- **Flight events:** MQ events associated with flight status (departure, takeoff, arrival, etc.).
- **Passengers:** MQ events associated with passenger status (booked, checked in, boarded, etc.).
- **Aircraft communications:** System that manages all pilot interactions with the flight operations manager.

Further complicating matters, many of these dependencies also have their own web of dependencies. Several years ago, all these dependencies were impacting our testers' ability to test as follows:

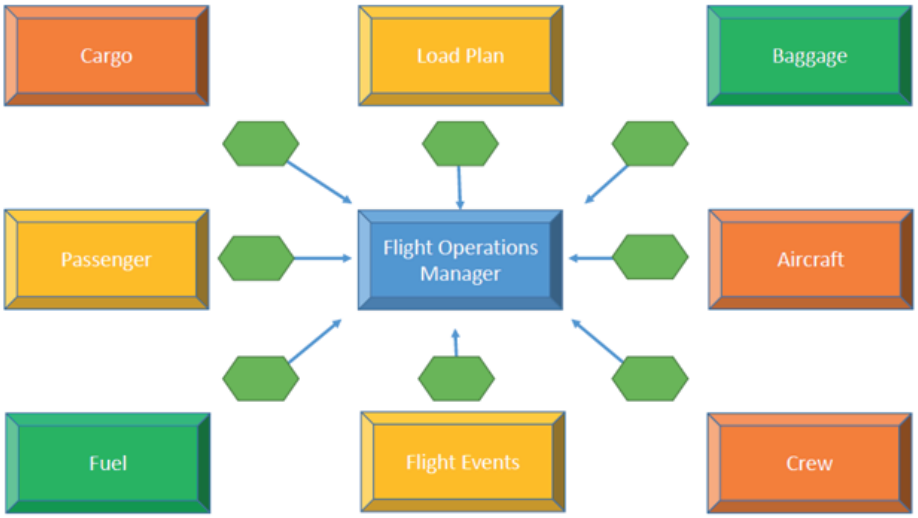
- **Environment:** Our test environment was shared and continuously evolving. As we updated our application, we impacted others who depended on us. And when those other teams updated their own applications, all the other teams—including ours—were also impacted. As a result, teams spent a lot of unnecessary time troubleshooting "issues" that were actually just side effects of environment instability.
- **Test Data:** Often, the data required to set up realistic and comprehensive tests did not exist or was too inconsistent to meet our needs. As I'll explain in more detail later, cloning from production was not ideal for our testing needs.
- **Services:** Likewise, the services we need to interact with were often unavailable or too inconsistent for our testing purposes.
- **Events:** Since the AUT is an event-driven system, we need it to be populated with realistic events before we can test it. However, due to all these dependencies and their complexity, it was extremely challenging to get realistic sets of events (such as passenger check-in/boarding/seating scenarios and flight departure/takeoff/landing/arrival sequences) into the system. Adding the events needed to test extreme and negative conditions was even more complicated.

How did we address these challenges? By fully isolating the AUT, then simulating advanced events and test data.

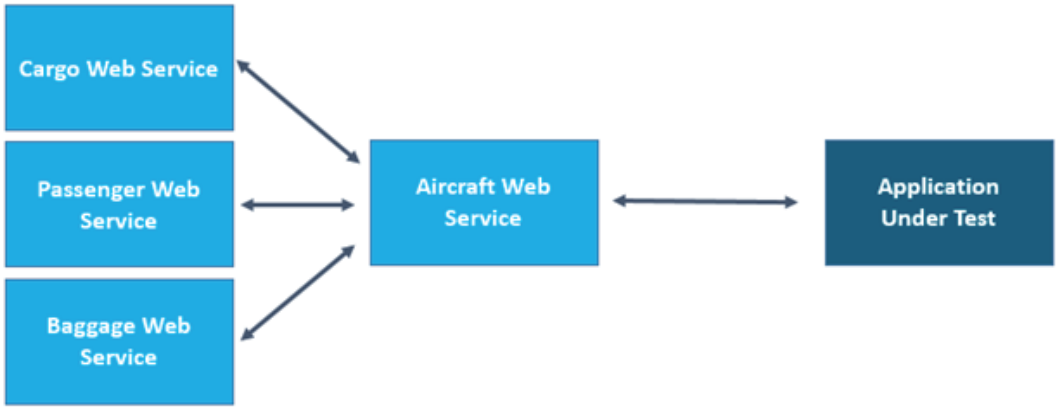
Isolating the AUT

First, we used VMware to establish an entirely new "CERT" test environment that's completely isolated from our production environment as well as our other dev/test environments. This way, we can control when we incorporate other teams' updates, and we're not constantly impacted by instabilities and unintentional changes.

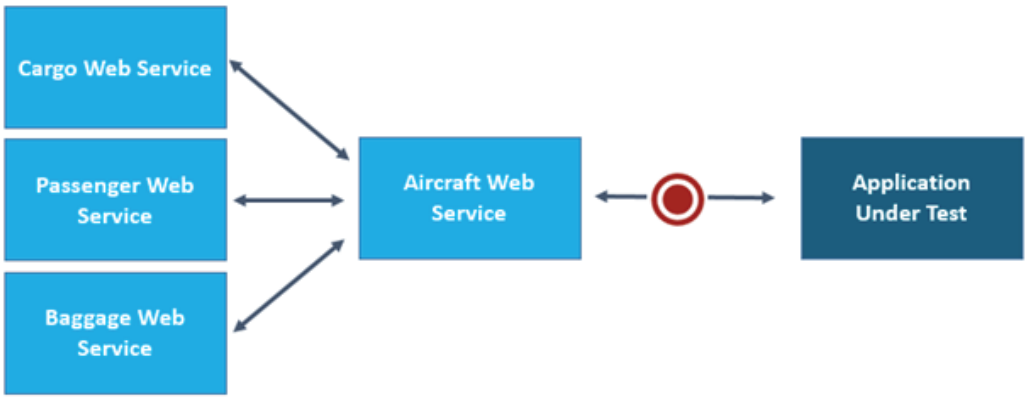
Within this isolated CERT environment, we then isolated the AUT from all of its dependencies (e.g., cargo, baggage, passengers, fuel, etc.). This lets us give testers access to whatever dependencies are required for testing—all configured to the exact conditions that their test scenarios require.



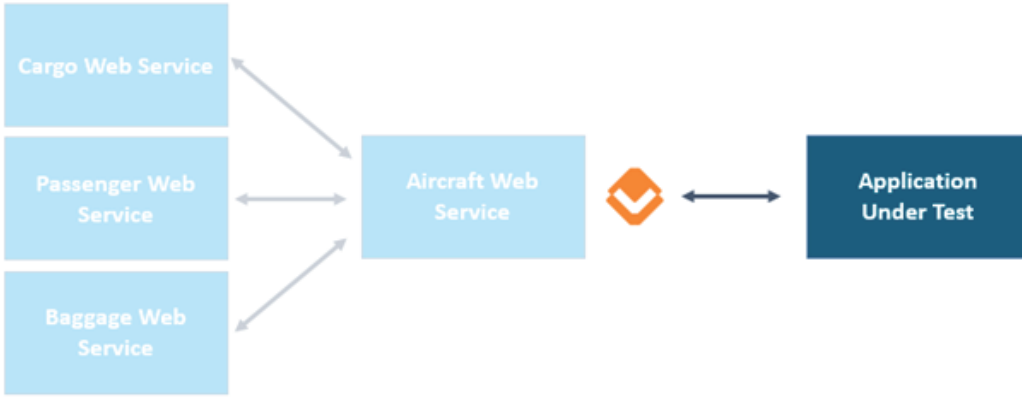
We started by removing the dependencies on the services that were too unreliable and inconsistent for our testing purposes. Using service virtualization, we recorded the AUT's interactions with its various dependencies. For an example of how this works, let's take the case of the AUT's interactions with the aircraft web service, which in turn interacts with several other web services.



To simulate the AUT's interactions with these services, we placed a service virtualization proxy between the flight operations manager and the aircraft web service.



We ran through an array of core scenarios and the proxy recorded that traffic. This enabled us to disconnect all the back-end infrastructure and have our AUT interact directly with the virtual asset simulating that service.



Now, when we run through our scenarios, we get consistent responses from those dependencies. If the tests don't achieve the expected results, we're fairly certain that something is wrong with our AUT.

We applied this same strategy to simulate realistic request/response traffic for our various synchronous web services.

Simulating Advanced Events and Test Data

Next, we moved on to a more daunting challenge: simulating the responses of events and data that were beyond the scope of request/response recording. The test environment and AUT were sufficiently isolated for testing, but we needed a way to ensure that complex (and highly-correlated) sets of events and test data were configured realistically and flexibly.

Two of the most challenging aspects of configuring the test environment involved how to represent asynchronous flight events and passenger events. Each time a passenger books a ticket, checks in, boards a flight, etc., an event is added to the airline's system.

An appropriate set of such events must be present in order for the flight operations manager to process a flight. Processing a flight also requires the presence of flight events: for example, the events added when departing from the gate, taking off, landing, and arriving at the gate.

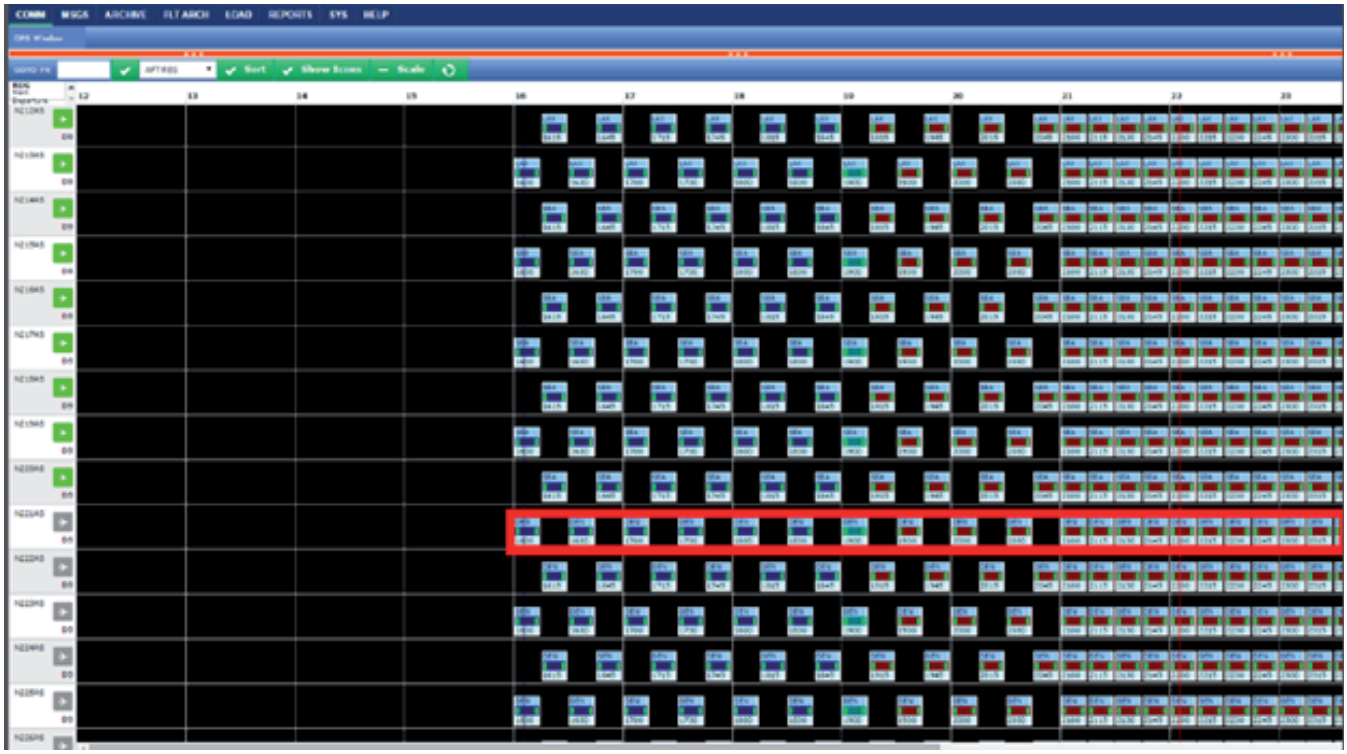
Without all of these events in a feasible state—and without all the different systems' events, data, and behavior all properly aligned—we simply can't test the flight operations manager. As you can imagine, manually configuring and correlating all these details would be incredibly time-consuming and tedious. Cloning data and events from production would be an improvement, but this approach has some key shortcomings:

- **Testing times are restricted:** The cloned sequences of data and events are correlated to specific times, and manually altering the timing could easily result in misalignment. As a result, testing would have to occur whenever the production data sets the system to the desired state. For example, if you wanted to test versus 7 AM EST flights, you'd have to run this test very early in the morning. Moreover, if you wanted to repeat the test under the exact same conditions, you'd have to wait 24 hours for the planes and flights to be aligned in that same way again.
- **Testing scope is restricted:** Cloned data is not flexible enough for exercising negative tests, corner cases, etc. Given our on-time record, it's really hard to find real data we can use to create all the delays, cancellations, etc. we want to test against. Providing an environment where testers could test against exceptional conditions (such as minor delays at a particular airport or regional delays resulting from a storm) would require considerable manual adjustment and re-alignment.

Rather than clone production data and events, we designed flexible event models that can dynamically generate the events and data needed for any given test scenario. Using the data captured during service virtualization recording as a baseline, we can instantly populate the test environment with whatever type, number, and combination of departure/takeoff/landing/arrival events that testers need in order to exercise different scenarios. Simulating a flight is as easy as providing the departure and arrival details, then all the other events—such as take-off, landing, and the time needed to unload, clean, and reload the plane—are dynamically generated on-demand.

For example, the following screenshot shows our flight operations manager representing the actual aircrafts and events scheduled in a given time period. Each row represents one airplane, and the blocks represent the various flights scheduled for that airplane.

By simulating events with our event modelling strategy, we can instantly populate a test environment with whatever number and combination of flights (and delays or cancellations) that a tester needs. For example, we can instantly spin up something like this:



This way, testers can access the exact test environments they need at whatever times they want. Moreover, they can instantly re-create a test environment at any point (e.g., if they need to reproduce a defect or verify that a defect was resolved).

We take a similar approach to test data modelling for passenger events. Given a specific passenger scenario and departure time, we can apply a model that allows us to generate test data sets representing everyone on that flight at different intervals before departure. For example, 120 minutes before departure, the test data set might have 33% of the passengers checked in but none boarded. 30 minutes before departure, 95% might be checked in and 45% boarded. As with our flight events, all the variables are entirely adjustable; as long as we know what the tester needs for a given test scenario, we can dynamically spin up the appropriate test data.

When dynamically generating test data in this manner, it is simple to switch between testing versus a mostly empty flight with on-time boarding to testing versus an overbooked flight where boarding is delayed. We can also adjust the model to accommodate different business rules. For example, checkins for Mexico flights need to occur 45 minutes early, but we can easily adjust all the timing for each passenger accordingly.

Results

Thanks to all this simulation and automation, testers can test whatever scenario they want, whenever they want. Our tests are now 100% reliable and repeatable, and we've eliminated all the variables that previously caused false positives and wasted time. Enabling testers to instantly access the exact test environment configurations that their test plans call for helps Alaska Airlines ensure smooth flight operations when faced with anything from a brief air traffic control outage to a snowstorm in July.

How Service Virtualization Enables Continuous Delivery

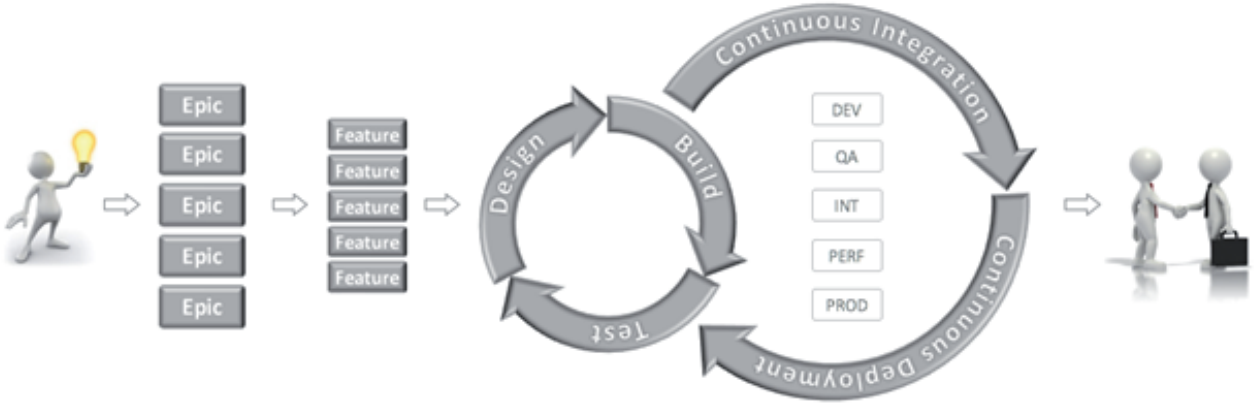
Senior Director of Technology at Capital One

Making the leap to Continuous Delivery is precarious for any organization, but the concerns are greatly exacerbated when you're a financial organization servicing over 65 million customer accounts worldwide—predominantly over digital customer interactions. This submission recounts how the Sr. Director of Technology at a US-based digital banking leader transformed his organization's DevTest culture, processes, and technical infrastructure in their evolution to Continuous Delivery.

Business Drivers for Continuous Delivery

Several years ago, like most financial institutions, we were working on 3-6 month waterfall release cycles. However, it was becoming apparent that our competitors were not only the other banks in the industry; we were also competing with the leading technology companies. To stay ahead of this competition, we needed to transform into a technology company that continuously innovates and delivers top-quality products to our customers. As our CEO has stated, "Digital is who we are and how we do business."

After considerable research, we determined that the best way to achieve this goal was to adopt Agile, transition to Continuous Delivery, and live in a DevOps-driven environment. For us, "Continuous Delivery" means having an effective software assembly line. Upon each commit, an automated process triggers a build, executes the appropriate tests (unit tests, security scans, etc.), deploys the new build into test environments, runs an additional level of testing, and then completes all the additional processes and validations needed to bring the application to production. This process runs multiple times a day, every day.



The Impact to DevTest

Adopting Continuous Delivery requires change across the delivery teams. For example, developers are now expected to be:

- Accountable for writing "automatable" code
- Responsible for passing tests (all tests)
- Fungible in all aspects of testing

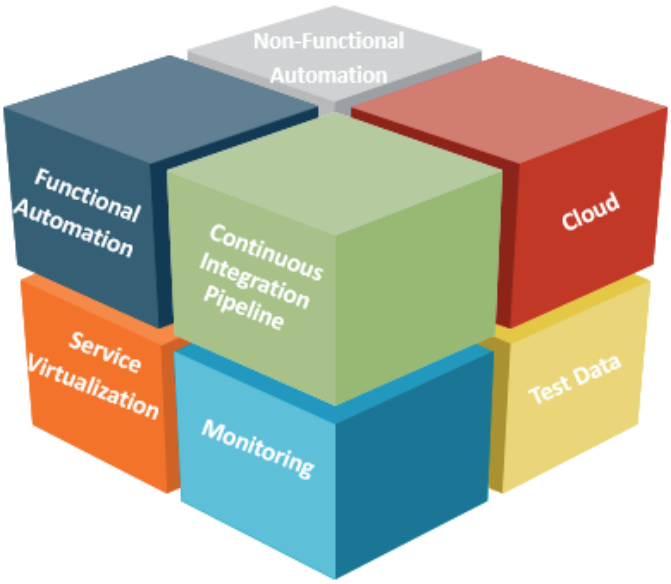
These are indeed significant changes. Nevertheless, I think that the greatest impact has been to the role of the tester. In waterfall processes, testing is often deferred until very late in the SDLC, which diminishes the voice and influence of software testers. With Agile, testing is brought into the "inner circle" and testers become an integral part of the team.

When teams first transition to Agile, it's not uncommon for a user story to be implemented in one sprint, then tested in the next one. After a little while, you might start talking about automation and maybe performing some test automation—ideally, in the same sprint in which the related functionality is implemented. However, many teams hit roadblocks at this point: the functionality may not be completed by the time that testing needs to begin, or downstream systems and environments required for testing might not be available. When this occurs, teams usually end up having to schedule a hardening or regression sprint later on—which means that your "Agile" process is really just waterfall in a somewhat different configuration.

To really embrace Agile and enable Continuous Delivery, testing needs to change significantly. Testers need to be able to rapidly build robust, valuable tests that are geared towards automation. This involves transforming testers' skills, processes, tools, and even the metrics being measured.

Transitioning to Continuous Delivery: What's Needed

Once your team has committed to running tests every day, multiple times a day, it becomes rather clear that traditional testing technologies and methods no longer suffice. Like most testers, our testers previously relied on GUI-focused tools that allowed the tester to test without getting into the "guts" of the application. However, to achieve the speed and automation that our initiatives required, we had to shift away from this approach and adopt more technical strategies such as ATDD and BDD, which we implemented with tools such as Cucumber, Ruby, and Selenium. These tools require some level of programming knowledge, but our testers weren't programmers. To close that gap, we made a very concerted effort to provide training and coaching, set up pair programming, and so on.



With these tools in place and the team ready to use them, a new constraint typically manifests itself: you can't access all the dependent components required to execute a meaningful test. This is where service virtualization comes into play. For example, assume that you're on a team building a UI and the downstream APIs or back-end systems aren't ready yet. How can you effectively test your UI? With service virtualization, you can simulate the dynamic responses that those dependencies would provide. You can then run your tests against these virtual assets to start validating and fine-tuning the front end's interaction with the back end. Later, when those actual back-end components are completed, you can just flip your tests to hit the actual endpoints instead of the virtual ones. At this point, you're just confirming that the AUT's behavior does not change when interacting with the real system.

For another example, assume that your end-to-end tests involve mainframes. This is extremely common at financial institutions, and it presents quite a challenge. Mainframes have transaction limits, and once you perform an operation on an account (opening a new account or posting a specific payment), you can't repeat that exact same transaction. How can you run the same test over and over again, multiple times a day, when that account expires after the first test run? We can avoid this predicament by simulating mainframe behavior with service virtualization. It also helps to have a test data management solution that provides instant access to data that's reusable across our test and service virtualization efforts.

After overcoming those roadblocks, you need to learn how to plug your tests into the Continuous Integration pipeline, which is essentially your automated assembly line. This involves understanding how to use a tool like Jenkins to call the test suite and how to tag tests to indicate in which phase(s) each test should run—and what to do when it fails.

Service Virtualization as a Process Tool

Our approach to implementing service virtualization was to establish a centralized service virtualization practice driven by our enterprise team (a center of excellence).

When we first started with service virtualization, our initial focus was on removing performance testing constraints. Establishing properly-scaled performance test environments in a non-cloud environment is extremely costly, so we wanted to use service virtualization to replace systems that were not available in our performance testing region.

We created an enterprise team and decided to have that team serve as service virtualization experts. They would create the first round of virtual assets, then turn those assets over to the teams they supported. Once the enterprise team created the first set of virtual assets, they trained the performance testers on how to access them, manage them, extend or modify them, and how to create new ones. The result: an army of performance testers skilled at performing service virtualization.

To begin the next phase of the rollout, we identified other business application development teams that were wrestling with system constraints. After exploring the scope of their constraints, we developed strategies for how they could apply service virtualization to eliminate these constraints. In some cases, we trained them on how to build virtual assets from the start; in others, we handled the initial asset creation, then helped them take it over from that point forward.

Today, the enterprise team does not create any virtual assets; the application development teams all handle it themselves. People are able to take the Parasoft documentation and supplemental training material, then start creating the virtual assets that their team needs. We still provide guidance as needed, but the responsibility for creating assets is borne solely by the application development teams. The enterprise team's focus is on exploring more advanced applications of service virtualization. For example, we're currently looking into how to:

- Take advantage of Docker with service virtualization
- Use cloud solutions like AWS to create test environments and load generators on the fly
- Plug service virtualization into different pipelines
- Ensure that no human interaction is needed to execute performance or functional tests, scale them up or down, and ensure that the test results are automatically sent to the appropriate people

Tips for Transforming Test

When I discuss our DevTest transformation at conferences, a lot of people are rather shocked. The role of the tester at our organization is significantly different than it is at many other organizations—especially in the financial industry. Testers don't necessarily need to be developing applications, but they do need to understand things like Groovy and Java in order to perform the necessary automation and take advantage of the tools and best practices we've standardized on. They also need a good understanding of the Continuous Integration pipeline, how to apply service virtualization to remove constraints, how to create tools to get data dynamically using the cloud, and so forth.

It's not easy, of course. But as testers learn these new skills through training, pair programming with developers, and so on, they really extend their role to become an integral part of the team and to help us achieve "quality at speed."

Service Virtualization, Performance Testing, and DevOps

Frank Jennings, Director TQM Performance Testing at Comcast

Before service virtualization, Comcast's Performance Testing team often ran into scheduling conflicts around sharing the test infrastructure. Sometimes downstream systems were not available. Other times, test engineers would try to run tests at the same time, which could affect the test results. This led to variability between tests, which made it challenging to isolate particular problems. Learn what results they've been able to achieve after approximately 3 years of service virtualization—and why service virtualization is a key component of their DevOps initiative...

We turned to service virtualization for two main reasons. First, we wanted to increase the accuracy of performance test results. Second, we were constantly working around frequent and lengthy downtimes in the staged test environments.

My team executes performance testing across a number of verticals in the company—from business services, to our enterprise services platform, to customer-facing UIs, to the backend systems that perform the provisioning and activation of the devices for the subscribers on the Comcast network. While our testing targets (AUTs) typically have staged environments that accurately represent the performance of the production systems, the staging systems for the AUT's dependencies do not.

Complicating the matter further was the fact that these environments were difficult to access. When we did gain access, we would sometimes bring down the lower environments (the QA or integration test environments) because they weren't adequately scaled and just could not handle the load. Even when the systems could withstand the load, we received very poor response times from these systems. This meant that our performance test results were not truly predictive of real world performance.

Another issue is that we had to work around frequent and lengthy downtimes in the staging environments. The staging environment was not available during the frequent upgrades or software updates. As a result, we couldn't run our full performance tests. Performance testing teams had to switch off key projects at critical time periods in order to keep busy–they knew they wouldn't be able to work on their primary responsibility because the systems they needed to access just weren't available.

These challenges were driving up costs, reducing the team's efficiency, and impacting the reliability and predictability of our performance testing. We knew we had to take action—and that's why we started looking at service virtualization. Ultimately, we found that the time and cost of implementing service virtualization was far less than the time and cost associated

with implementing all the various systems across all those staging environments—or building up the connectivity between the different staging environments.

Results After 3 Years of Service Virtualization

We've been doing service virtualization for about 3 years now. Our initial focus was on the biggest pain points in terms of scheduling conflicts within the performance testing teams, unavailable systems, and systems where our testing would impact other development or test groups. Since we started, we've been able to virtualize about 98% of the interfaces involved in our tests, and we've seen a 65% annual reduction in the amount of time it takes us to create and maintain test data (factoring in the time we spend creating and updating virtual assets). We've also reduced staging environment downtime by 60%.

Our tests are now more predictable, more consistent, and more representative of what would be seen in production. Moreover, we're also able to increase the scope of testing in many cases. For example, we can't put production loads on certain actual services, but when we're working with virtual services we can ramp it up with production-level loads and get realistic responses, both in terms of data and performance. We can really isolate the AUT, not just from a performance testing perspective, but also from a performance profiling perspective. Instead of just telling development "this is the performance of your system," we can also say "this is where we're seeing the bottlenecks and this is where we think changes might improve the throughput of the application."

The key benefit for the performance testing team is the increased uptime and availability of test environments. Service virtualization has allowed us to get great utilization from our testing staff, complete more projects on time, and also save money by lowering the overall total cost of performing the testing required for a given release.

Service Virtualization and DevOps

Beyond performance testing in our staging environments, we've also been able to use service virtualization for everything from unit testing and regression testing in the development environment, to baseline performance testing in early downstream environments, to functional and regression testing in the QA/integrated environment, to manual/exploratory testing in an environment that's quite close to production (but uses virtual assets in some cases).

All the configuration and deployment of virtual assets for the various environments is automated as part of our DevOps infrastructure. Environments automatically switch between virtual assets and actual assets according to the business rules we've defined—for example, based on what endpoint the traffic is coming from, the data contained in the test packets, etc.

Compressing Testing From 2 Weeks to 2 or 3 Days

Since we can start working on our scripts versus virtual assets in the development environment, we've typically got everything ready to go quite early in each sprint. Before, we used to need 2 weeks to performance test the code (for example, with average load tests, peak load tests, endurance tests, etc.) once we got it in our staging environments. Now, we've shrunk that to just 2 or 3 days.

Service Virtualization is Essential for Functional Testing of Complex, Distributed Systems

Sanjay Chablani, Head of SQA Biogen Idec, reflecting on the service virtualization implementation he oversaw as Director of eCommerce Quality Management at Staples

---

Staples is committed to making everything easy for its customers, but ensuring positive customer experiences on their eCommerce site is far from simple. Functional testers must contend with the high number of dependent systems, subsystems, and services that are required to complete almost any eCommerce transaction—but rarely available for dev/test purposes. Learn how the eCommerce functional testing team leveraged service virtualization to more rapidly and more exhaustively test complex transactions across highly-distributed systems...

As the director of QA for the Staples eCommerce site, my team and I were responsible for ensuring that every transaction involving the eCommerce application operated seamlessly and reinforced the company's commitment to providing an easy customer experience.

Like so many modern applications, the eCommerce application interacted with hundreds of other systems and services, all of which were reused widely across the company and had multiple integration points. This complex distributed architecture was great for providing consistent reusable functionality across the organization and enabling us to minimize application footprints. However, it definitely complicated our team's ability to perform the functional testing needed to ensure that end-to-end transactions across the eCommerce site met expectations.

Some of the main challenges we faced with our functional testing included:

- **Environment availability:** With today's enterprise systems, being able to fully replicate environments for testing is no longer a possibility. Replicating a fully-connected environment multiple times would require too much money and effort—and sometimes it's simply impossible (such as when we need access to not-yet-implemented components being developed in parallel). Nevertheless, having complete test environment access was critical to our ability to execute almost any test. Inability to access just one dependent subcomponent could impact the testing of our entire eCommerce site.

- **Environment scheduling:** Since the subsystems, subcomponents, and services that we (and many of the other divisions) relied up on were all owned by different groups, we were always competing for access to these constrained resources.
- **Access to realistic test data:** To complete a single test, we needed to have test data refreshed and synchronized across the various services, subcomponents, and subsystems. Considerable effort and coordination among different teams was needed to get the right test data populated across all the systems.

Our initial attempt to overcome these challenges involved stubbing. At first, we built some basic stubs, then we moved to "semi-intelligent" stubs, which we deployed in our test environments so they could stand in for the actual services. However, stubs had their limitations. They're highly static and they typically require the involvement of developers who are familiar with how the applications interface.

When we heard about service virtualization, our group quickly recognized its potential to address our functional testing challenges, and we were the ones that took the lead on the company's service virtualization initiative. What initially drew us to service virtualization was how much easier it would be—in terms of recording traffic, deploying virtual assets that simulated this traffic, then making those virtual assets available for on-demand provisioning.

The goal of our service virtualization initiative was to create a library of virtual assets that would represent service providers and service consumers, including the message bus (service proxy). Essentially, if a system was not something we were responsible for testing, we would create virtual assets for it. This would allow us to simulate any dependency involved in our various end-to-end tests. To promote the development of such a rich library, we made virtual assets a required deliverable within the SDLC.

Ultimately, we found that service virtualization provided us the following benefits:

- **Faster release cycles:** With service virtualization, we could start testing earlier in each cycle, and reduced the time required to execute our test plans. This was especially critical on parallel development projects, such as when the Retail, Warehouse, and eCommerce teams were all working on functionality related to online ordering with in-store pickup. This was a complex project with a very aggressive timeline. Using service virtualization to simulate resources that were still being developed, each team's development and testing could move forward without waiting on the others. With the virtual assets, we could start integration testing much earlier than if we had to wait for all the dependent components to be completed. This helped us get everything running smoothly even before we integrated all the completed components. Ultimately, we not only completed the project on budget, but actually ended up deploying it two weeks early. This success really helped us promote service virtualization adoption across the company.

- **Greater control over environment stability and application behavior:** We could complete test scenarios that were previously impossible (due to data privacy guidelines) since service virtualization enabled us to mimic behavior that was very data dependent. Moreover, we could not only trust that the dependencies we needed to access would be available, but also exert additional control over them (e.g., to test corner cases, error conditions, etc.) to achieve greater test coverage.
- **Ability to test on our own schedule:** We no longer had to wait on others to complete applications being developed in parallel, and we no longer had to compete with other teams in order to schedule limited windows of access to highly-constrained shared resources.
- **Reduced issue and defect resolution times:** Before, we had to look at all the different subsystems to try to figure out what was causing a problem. By turning on different virtual assets, we could really isolate different subsystems and quickly zero in on the root cause of the problem.

We Couldn't Do Agile Development Without Service Virtualization

Bas Dijkstra, Test Automation and Service Virtualization Consultant

After KPN decided to adopt an Agile process with 2-week iterations, their 6-week long QA test cycles became an immediate concern. Acceleration would be impossible unless they addressed the underlying bottleneck: a lengthy, manual process for setting up test environments and test data. Learn from their experiences using service virtualization to eliminate this constraint—as well as to extend and automate testing...

At KPN, the shift to Agile would not have been possible without service virtualization. Our team focuses on a central order management system which has numerous dependencies; even a simple order provisioning transaction touches at least 10 other systems. One of those systems created a major bottleneck for testing.

To complete almost any end-to-end test case through this order management system, the test team had to execute an order provisioning transaction. However, such transactions could not be completed unless the corresponding order information was already present in the back-end system's database—and the delay between requesting an order and having it manually entered into the database could be 2 weeks or more, due to resource constraints.

Before the shift to Agile, test cycles lasted 6 weeks or more. The test team could execute only a limited number of test cases because they had a limited number of orders at their disposal. Approximately every other test cycle, we'd have a release. Over half of the entire cycle was dedicated to test, which cut into development quite significantly. The process was also delayed by all the rework and retesting that stemmed from defects being discovered so late in the cycle.

If we tried to adopt 2-week Agile sprints without eliminating the 2-week long (or longer) test environment access delay, the sprint would have been over before testing could even begin!

We saved weeks per test cycle by adopting service virtualization. Before, we had to wait for orders to be entered before any testing could begin...then everything had to be tested manually. With service virtualization, orders could be added much faster—in minutes instead of weeks—so we could test earlier and more extensively.

Increased Test Coverage

We could achieve far higher test coverage with service virtualization because the more orders that we had at our disposal, the more test cases could be executed. This enabled much higher coverage versus requirements.

Testing could also cover a much broader range of test data. For example, the customer's zip code determines what types of products are available. If a test retrieves product information from a live product availability service, the test can cover only products that are currently available for a given zip code. With service virtualization, you can easily define and apply data for other scenarios—like product packages that will soon be available in a certain zip code.

Service Virtualization vs Stubbing/Mocking

KPN's development efforts also benefited from service virtualization. Developers are using some of the virtual assets to perform integration testing within their development environment. Prior to service virtualization, they had to mock or stub everything themselves. They are finding that service virtualization is not only faster, but also more flexible and more realistic.

Automated Testing

When you're waiting weeks to be able to execute a single test case, automated testing simply isn't possible. Since service virtualization enables orders to be provisioned in minutes rather than weeks, test automation has become a reality for KPN.

Service virtualization has also helped us automate the testing of some validations that we could never perform before—with either automated or manual testing. For example, the central order management system (the main application the testing focuses on) sends out "fire and forget" messages to ISPs and other people that might be interested in the status of an order or a process. Since this communication is one-way only (no response is received), there was no way of checking whether the updates were actually sent and what the message contained. They're sent out into thin air, then who knows what happens.

To gain visibility into this process, we created virtual assets that act as the ISP backend systems and actually capture the fire and forget messages. The virtual assets then store the messages in a database so they can be picked up by Parasoft's functional testing solution. This enabled us to confirm whether the updates were sent and even enabled us to validate whether the message contents met expectations.

Time to Market...with Quality

The IT Director at one of the largest telecommunications companies in Benelux

When it comes to the classic cost, quality, schedule triangle, one of the largest telecommunications companies in Benelux is not willing to make any of the classic compromises. To advance the company's "Time to Market...with Quality" initiative, they turned to Service virtualization. Learn from their experiences shifting left all phases of their extensive quality practices (e.g., unit testing, integration testing, system testing, load/performance testing, security testing), as well as using Service virtualization to enable a fully-automated build-deploy-test pipeline...

Our vision around service virtualization is focused on "shift left": starting development and testing much earlier in the process. This helps us advance our organization's "Time to market... with quality" initiative.

There are three KPIs critical to project management: time, quality, and budget. When you focus on one, the others tend to suffer. Our organization focuses on both time to market and quality: we want to get the product to market fast, but we will not compromise quality. Quality is critical to our brand, so we never cut corners in that respect.

Service virtualization assists us with both "time to market" and "quality." Of course we want to be able to develop fast, but we've got a very complex environment. We're not a startup—we have a lot of applications and a complex infrastructure with many connections and dependencies. To give you some idea, we have about 2000 total applications. There are about 300 main applications used in our business, and about 70 comprise the core. All these applications talk to each other and they all assume that the others are available. Each time we add or change something, it's a huge effort to validate that the new functionality meets expectations and ensure that it did not have unintended side effects.

Service virtualization gives us the opportunity to begin development and testing earlier. It impacts the full development lifecycle, starting with development and unit testing, to system testing, up to non-functional requirement testing, load and performance testing, and even security testing.

To "shift left" development, we use service virtualization to validate the design that, up to that point, exists only paper. As soon as a design is validated by different teams, we represent it as a virtual asset so we can start understanding how it really works. At that point, anyone who needs to develop against that interface can start immediately—before a single line of code is written.

By deploying virtual assets in our test environment, we can start testing even when parts of the system aren't yet completed or aren't readily available for testing. In addition to simulating expected behavior, we also use service virtualization to simulate outages of various dependencies. This enables us to validate how the application under test responds when other system components fail.

Service virtualization has also provided us value in terms of our training environments. Before, to deliver a complete end-to-end training session, we would need a full end-to-end set of applications. That's a huge cost—and in many cases, training involves just a fraction of a system's capabilities. Now, we use service virtualization to simulate different training cases. This lets us provide complete and realistic training environments without incurring huge infrastructure costs.

Enabling the Automated Build and Deployment Process

We see service virtualization as a layer that supports all of our projects and improvement activities across the full development lifecycle—including the way we develop, the way we test, and the way we release and deploy software.

As part of our "automated software build and deployment" initiative, we have set a goal to have a nightly build that triggers regression testing, unit testing, etc. The results are reported to development daily. Before the application is promoted to integration testing, certain policies for coverage and success rates need to be achieved. We view service virtualization as a way to increase the scope and success rate of the automated regression tests. The more endpoints are available for testing, the more regression tests we can execute, and the fewer false positives we'll have to review.

To support this initiative, we're designing our own "best-of-breed" system that builds the applications and tracks quality metrics. For instance, for Java projects, we have a mixture of commercial and open source tools for source code management system, automated build and deployment, quality management, security testing, and unit testing. Currently, we have around 115 applications subscribed to that framework, and we're in the process of building it out to cover our .NET development projects as well.

Since we also have a lot of software developed by partners, we would like to contractually set the targets that they need to reach in terms of quality and regression testing, unit testing, etc. When dependent systems are not available for testing or are maybe not yet even implemented, service virtualization assets can be used instead.

Server and Service Virtualization

Currently, we have a lot of servers virtualized, but they're all static. We don't have a build and destroy process for our test environments. They're built once and used for quite a while. We have 3 or 4 major releases per year, and we build and destroy the environment each time—and of course, the environment grows each time.

Each time a new application needs to be deployed, we have 5 corporate test environments ranging from integration testing to post-production environments. If you need to deploy an application in each of these environments, a considerable amount of configuration effort is required. That's why we're considering whether we really need the entire application in each environment, or whether a virtual asset will enable us to achieve the same objective much faster.

We've already reached our goal of 80% server virtualization and we're setting a target of 80% service virtualization. Of course, sometimes the two are linked.

At this level of virtualization, it's time to consider on-demand provisioning. We have a lot of non-corporate environments that are set up for specific projects. They should be set up for build and destroy, but they're not destroyed. We're currently planning to use both server and service virtualization to set up temporary development environments or system test environments specifically for certain new projects. These would be partially virtualized servers and partially service virtualization assets … that's the next phase for us.

Think Big, But Start Small

Our main advice to anyone getting started with service virtualization is "Think big, but start small." Start using service virtualization where it delivers the greatest value. This way, it's much easier to sell the organization on the value of service virtualization. You don't want to make a tool available and have to push people to use it. It's much better to create a high demand, where you have more people than you can handle approaching you asking to use it.

Look for the "low-hanging fruits" associated with frequency and demand and make sure those are available. Sometimes, even a relatively easy service virtualization task can make a big difference when it comes to enabling testing. For instance, one rather simple virtual service allowed us to replace a very complex appointment booking system. Another virtual service that was extremely easy for us to create helps us test negative scenarios—something that was quite valuable in our test environment.

When we were debating where to start, we talked about all the different use cases for service virtualization. For example, we looked at recording behavior and having virtual assets play it back, data-driven virtual assets, algorithmically-driven virtual assets, virtual assets that simulate a system that is not yet implemented, those that simulate a system that will never be available for test because it's too expensive, and so on. For every possible use case, we implemented something to prove that it works and to demonstrate the value. We have now demonstrated the value across the board. It's not always the most complex virtual asset, but we have proven that it's possible in every use case.

Service Virtualization with API Testing

Aaron Martin, Programme Test Manager at Ignis Asset Management

Ignis Asset Management is a global asset management company, headquartered in London, with over $100 billion (USD) in assets under management. They needed to accelerate testing for parallel and Agile development. Learn how they applied service virtualization as part of a broader test automation initiative to reduce testing time for their regression test plan from 10 days to a half day...

Ignis recently embarked on a large project aimed at outsourcing the back office as well as implementing the architecture and applications required to support the outsourcing model. To meet the business's needs, a number of projects have to be developed and delivered in parallel. However, we didn't have the resources, budget, and management capacity required to create and maintain multiple test environments internally. This limited test environment access impeded our ability to validate each application under test's (AUT) integration with third-party architectures. Moreover, our third-party providers also had limited test environment access, which restricted the time and scope of their joint integration testing.

At the same time, the company was transitioning to an agile development methodology. To support this initiative, we needed to adopt an automated testing solution to provide faster feedback after each build.

It soon became apparent that the existing testing process had to be optimized in order to meet these new demands. Executing the core test plan required 10 man-days. This process involved manually entering transactions in the originating application, which wasn't the primary AUT. Moreover, we were also manually building simple stubs to simulate interactions with third-party components that were not integrated. To enable complete testing to occur in more agile, parallel development—without requiring additional test environments to be built and maintained— we needed ways to:

- Enable applications (or parts of the target architecture) to be tested against the Ignis architecture before integration into the complete Ignis system.
- More efficiently simulate the AUT's interactions with third-party systems not yet integrated into the Ignis system.

Starting Integration Testing Prior to Integration

With automated API testing and service virtualization, we were able to establish a test automation framework that not only addressed the challenges outlined above, but also helped extend test automation across the SDLC.

Our initial implementation of the API Testing solution focused on automating the generation of order management traffic at the API level. The AUT was the message architecture, which interfaces with third-party components—both existing services provided by business partners as well as services being implemented in parallel by outsourcing providers. From the application initiating the order, live trade scenarios were used to form the basic test transactions. With the API testing solution, we were able to run the full transaction test plan, generating new instances of the message from data sources.

In parallel with the functional test automation, we adopted service virtualization to simulate the expected transaction response messages from third-party components. First, we rapidly implemented simple virtual assets that provided positive responses to all generated transactions, enabling us to simulate third-party responses without manually developing and managing stubs. The virtual assets were then extended to handle more complex response scenarios.

Additionally, we implemented automated tests and virtual assets to test outsourced components fully- decoupled from the Ignis environment. We used this to establish a "quality gate" that had to be passed before progressing to the integration phase. This was quite useful, since their code quality was poor and repeated testing in our integrated environment would have impacted other deliverables.

Reduced Test Time from 10 Days to Half a Day

We were able to reduce the execution and verification time for our transaction regression test plan from 10 days to half a day. This testing is not only automated, but also quite extensive. For example, to test the Ignis system's integration with one business partner's trading system, our fully automated regression testing now covers 300 test scenarios in a near UAT-level approach—with 12,600 validation checkpoints per test run.

Previous automation implementations focused on automating testing at the UI level—with varying levels of success. We determined that we really needed to generate transaction scenarios and traffic at the API level instead. Now, we're able to focus on the core test requirements and get more value from our investment in automation.

Bridging the Technical Gap for Executive Buy-In

Alvaro Chavez, Manager of Software Architecture at Hiper SA

Hiper SA is an independent software vendor that provides custom financial services software to clients such as Procesos MasterCard Peru, Banco de Credito del Peru, Scotiabank Peru, BBVA Peru, BAC (Bank of Central America), Visanet Peru, and many other Latin American businesses. The software they build integrates with numerous third-party systems in order to ensure secure, reliable, and compliant banking and payment transactions. They adopted service virtualization so that they could begin extensive integration testing even before the third-party systems were readily available for testing. Learn how Hiper's technical leaders gained executive buy-in for the service virtualization purchase...

What convinced our executives of the value of service virtualization was its potential for us to increase our competitiveness while isolating business risk. Our software supports financial transactions where even a "minor" glitch could have consequences for the business. To make sure the software we build is very reliable and does not bring risk to our customers, we need to extensively test how it works with the customer systems we are integrating with. However, we cannot access the actual customer systems for testing until we are ready to bring our completed software into the customer environment (we cannot remotely connect to the customer's system from our site). This means that integration testing cannot start until late in the process and it's difficult to complete all the testing we want to perform.

The proof of concept with service virtualization convinced us that we could simulate the type of complex transactions we work with—for example, transactions that directly hit 15 systems (and indirectly involve hundreds of systems) interacting over many different protocols and formats such as ISO 8583 and FIX. Once we saw this, we were confident that service virtualization would help us start testing how our software integrates with customer systems as soon as we completed each "component" internally. We knew this would help our team do the same (or greater) amount of testing a lot faster.

When we approached management for approval, the main selling point was that service virtualization would enable us to complete each project faster without any additional business risk. To the company, this meant that we could propose shorter timelines than our competitors when we bid for a project. It also meant that we could allocate our own resources more efficiently. Instead of having testers wait until late in each cycle to be able to test, they could start testing earlier, finish testing earlier, then move on to other projects.

As a result, we could get payment for each project sooner as well as increase our capacity to take on more projects over time.

Now that we have service virtualization approved and we are beginning the implementation, we expect that we will shorten the length of each cycle and that fewer cycles will be needed in order to satisfy client expectations. We are also anticipating that when we finally connect to the customer system, we won't have many big issues since we will have already completed so much testing against the simulated interfaces. This will make project timelines and budgets much more predictable. With service virtualization, we know that we can complete projects faster and ensure that they have the same, if not greater, level of quality.

A Love Letter from Cloud Dev/Test Labs to Service Virtualization: "You Complete Me"

Jason English, Galactic Head, Product Marketing at Skytap, Inc.—a provider of cloud-based Environments-as-a-Service for enterprise software development and testing

Drawing on a varied background of interactive design and marketing for supply chain and enterprise development vendors including CA, ITKO, Agency.com and i2, Jason has appeared as an author and pundit in numerous technical trade publications and co-wrote the book "Service Virtualization: Reality is Overrated." The following submission takes a lighthearted look at the "relationship" between two complementary technologies: service virtualization and cloud dev/test labs...

Dear SV,

Hey, I know it's been a while since we started being "a thing." When we met, everyone said you were just mocking, and that I wasn't real enough to make a living, with my head in the clouds. Yet, here we are, a few years later.

Service Virtualization, you complete me.

As a young Dev/Test Cloud, I always wanted to try new things. And what better use for Cloud than experimenting with software for startup companies? I was flexible, I thought I had the capacity to handle anything. I'd stay up all night studying or partying, but sometimes I'd crash. So what if some college kid's cloud-based photo-sharing site experiment goes down? It wasn't going to impact anyone's life.

But when it came to serious business, there was always something missing. What was I going to make of myself? Who could trust their future to me, and develop things that really matter in the cloud? Clearly I didn't have everything I needed – I was lacking certain critical systems and data, and it was preventing me from maturing. But you came along and together, we changed all that.

One thing I've learned is that I don't always have to handle everything by myself. A dev/test cloud environment is not just a place to store and run VMs for application work—it needs the same clustering, network settings, load balancers, security and domain/IP control as you have in production. I can handle a lot, for sure.

But there are certain items developers and testers need that don't image so well. Like a secure data source that should be obscured due to HIPAA regulations, or a mainframe system the app needs to talk to, but would be unwieldy to represent in a Cloud like me. That's when I say Service Virtualization makes every day a great day.

We've come a long way since then, and we've handled increasingly serious challenges. Simulating some very complex interaction models between systems, and deploying those into a robust cloud environment of real VMs and virtual services that can be copied, shared and stamped out at will across teams. We work together so well, we can practically finish each other's sentences.

Hard to believe all this started less than 10 years ago. Here's to us, Dev/Test Cloud and Service Virtualization standing the test of time. Now let's go make some history together.

Yours faithfully,

Cloudy

How Service Virtualization Impacts Business Strategy

Christine D. Eliseev, Founder and Managing Partner at QMAT Solutions (a staffing, recruiting and training firm that specializes in Quality Management and Testing), Big 4 alumni, and national speaker on current testing trends

Founder and former Director of the Quality Management & Testing consulting practice at PricewaterhouseCoopers LLP, Christine grew PWC's practice into a global network, offering traditional project-based software test strategy and management, as well as enterprise-wide Quality Transformations. Christine has been in systems integration consulting for 20 years, leading enterprise-level transformation projects for many Fortune 100 clients, as well as defining the test strategy and approach for some of the most complex mission critical systems. Draw from her experience as she shares her insights on how service virtualization influences business strategy...

Service Virtualization's Impact Beyond IT

Service virtualization has historically been viewed as a tactical solution to an immediate problem: we had a lot of development and testing work to do, and we were looking for ways to complete it more efficiently. Service virtualization definitely helped us achieve that—by now, it's quite clear that service virtualization reduces development cycles and exposes more defects earlier.

To date, service virtualization conversations have primarily focused on IT strategy: "How do we implement service virtualization tools and processes within our IT organization so that we can deliver higher quality software faster?" I think the next logical conversation we need to have as an industry is "How does service virtualization influence business strategy?"

Service virtualization's impact on the business extends far beyond IT. Since testing has traditionally created such a formidable process bottleneck, removing this constraint may bring changes and opportunities throughout the business. For example, when we're planning how long it will take us to bring a new product/campaign/service to the market, the process by which we perform estimations has to change because our variables have changed. Assume that we're launching a new retail campaign. We would have to understand how quickly we could get all the different components mobilized: updating the web sites and mobile apps, adjusting online pricing, updating in-store registers, and so on. If the organization has now successfully deployed service virtualization, this affects estimations for all those aspects of the campaign. It could also enable more "just in time" activities, since the release cycle is

significantly shorter. In terms of our campaign example, this could give you the freedom to defer certain promotion details until you're closer to the planned campaign start date—for instance, to better align the campaign with your current inventory, to take advantage of any unexpected trends that recently surfaced, etc. The cross-corporate impacts outside of IT can even extend beyond marketing and sales. For example, if you know you'll be putting out a product faster, you might need to ramp up hiring.

We're undeniably in a blended business enablement environment. Yes, the enabling technology is software and infrastructure, but as we bring those two things together it's not just about IT strategy—it's really a matter of business strategy. Since service virtualization can bring such impressive efficiencies, it really is significant enough to impact the business as a whole.

Disruptors as game-changing as service virtualization influence how people approach business strategy: given this "gift" of extra time and the ability to move faster, what should we do next in order to provide the greatest value to the business...and how long will it take us to do it? From sales, to marketing, to human resources and beyond, the business is now remarkably impacted by the fact that not only can we enable so much of the business with technology, but we can enable it so much faster than we could previously.

Time to Raise Your Definition of Quality?

Another interesting subject is the potential of service virtualization to fundamentally change an organization's definition of quality. It enables us to shift from an inside-out approach to a business-driven approach. In other words, rather than basing targets on what we're currently achieving, we now can (and should) aim for a higher level that's more closely aligned with business risks.

"The definition of quality" at the level of the particular application, division, or project typically revolves around your level of failure tolerance. If you know that service virtualization enables you to find 60-90% of your defects earlier, what was considered "good" before might not be good enough now, given the new ability to test earlier, faster, and more extensively. The move to service virtualization could be a prime opportunity to re-examine your current expectations and consider whether you can now raise the bar on quality.

At a higher level, you have the organization's definition of quality. In addition to expectations for the application or project, this could also cover expectations for the people who are using service virtualization. For example, you might want to adjust KPIs like failures per LOC based on what's now possible given this new disruptive technology. Moreover, you might start measuring people on metrics that you never even considered before, such as measuring developers on how many tests were run prior to deployment. This wouldn't have been feasible 5 years ago, when the focus was on doing just enough testing to ensure something was releasable—not on achieving and logging a certain level of test coverage.

Widening the Gap Between "Verification and Validation"

From a business strategy perspective, the divide between verification (making sure that the software works) and validation (making sure we built the right thing) is actually increasing as a result of simulation technologies like service virtualization.

Previously, that line was much more blurred. You would bring in users to perform some system integration testing, let them play around with the system a little, do a final check off, then call that "user acceptance testing." However, this was hardly an ideal approach. Bringing end users into the testing cycle is costly because any amount of time users spend on testing means less time available to complete their core job responsibilities. Not surprisingly, many managers are reluctant to have their employees participate in user acceptance testing because it siphons time and resources away from achieving their main goals.

Now, that gap is increasing to a point that it offloads a lot of work from the users. IT can demonstrate to the end users the greater depth of verification that has been completed by taking advantage of service virtualization, leaving the users to indicate whether it meets their needs. This is a huge relief to most users. When they realize that the application has already been exhaustively tested by the time they see it, most are thrilled because this saves them a considerable amount of time. In many cases, users are requesting to skip actual user acceptance testing and base their acceptance decision on what they see in a demo.

What Service Virtualization Means for Testers

As service virtualization surfaced, many testers worried that it was a threat to their jobs. However, that couldn't be farther from the truth. I think it's become clear that this isn't the end of the tester; it's the beginning of the next phase of testing.

Service virtualization has emerged as an enabler that allows us to take "test" from a tactical time-boxed activity into a more strategic activity that spans the lifecycle of the release candidate and is much more closely aligned with the business risks associated with the application. Of course, to achieve this, we need to think through some things differently in terms of what to do, when to do it, and how to do it.

This is truly an evolution of the terms of testing. We can't keep doing things the same way because we just don't live in that world anymore. Service virtualization is an enabling technology, a stepping stone that will bring us to the next phase of testing…one that I expect will involve an array of different automation techniques and technologies.

About Parasoft

This resource was curated by Parasoft, industry-leader in Service Virtualization, API Testing, and Development Testing.

Parasoft researches and develops software solutions that help organizations deliver defect-free software efficiently. To combat the risk of software failure while accelerating the SDLC, Parasoft offers a Development Testing Platform and Continuous Testing Platform. Parasoft's enterprise and embedded development solutions are the industry's most comprehensive—including static analysis, unit testing, requirements traceability, coverage analysis, functional & load testing, dev/test environment management, and more. The majority of Fortune 500 companies rely on Parasoft in order to produce top-quality software consistently and efficiently as they pursue agile, lean, DevOps, compliance, and safety-critical development initiatives.

Upon entering the API Testing marketplace in 2002 with Parasoft SOAtest, Parasoft pioneered service virtualization capabilities beginning with the industry's first "stub server." Since then, Parasoft has been leading the service virtualization marketplace, as recognized by Gartner, voke, Forrester, and Bloor, as well as awards including the Jolt Grand Prize, Info-Tech: Trend Setter and Market Innovator Awards, and Wealth & Finance Award for Most Innovative Software Vendor.

Contacting Parasoft

Headquarters

101 E. Huntington Drive, 2nd Floor

Monrovia, CA 91016

Toll Free: (888) 305-0041

Tel: (626) 305-0041

Email: info@parasoft.com

Global Offices

Visit www.parasoft.com/contact for contacting Parasoft in EMEAI, APAC, and LATAM.

PARASOFT®

www.parasoft.com