Skytap

DevOps: Lessons Learned Over Decades of Practice

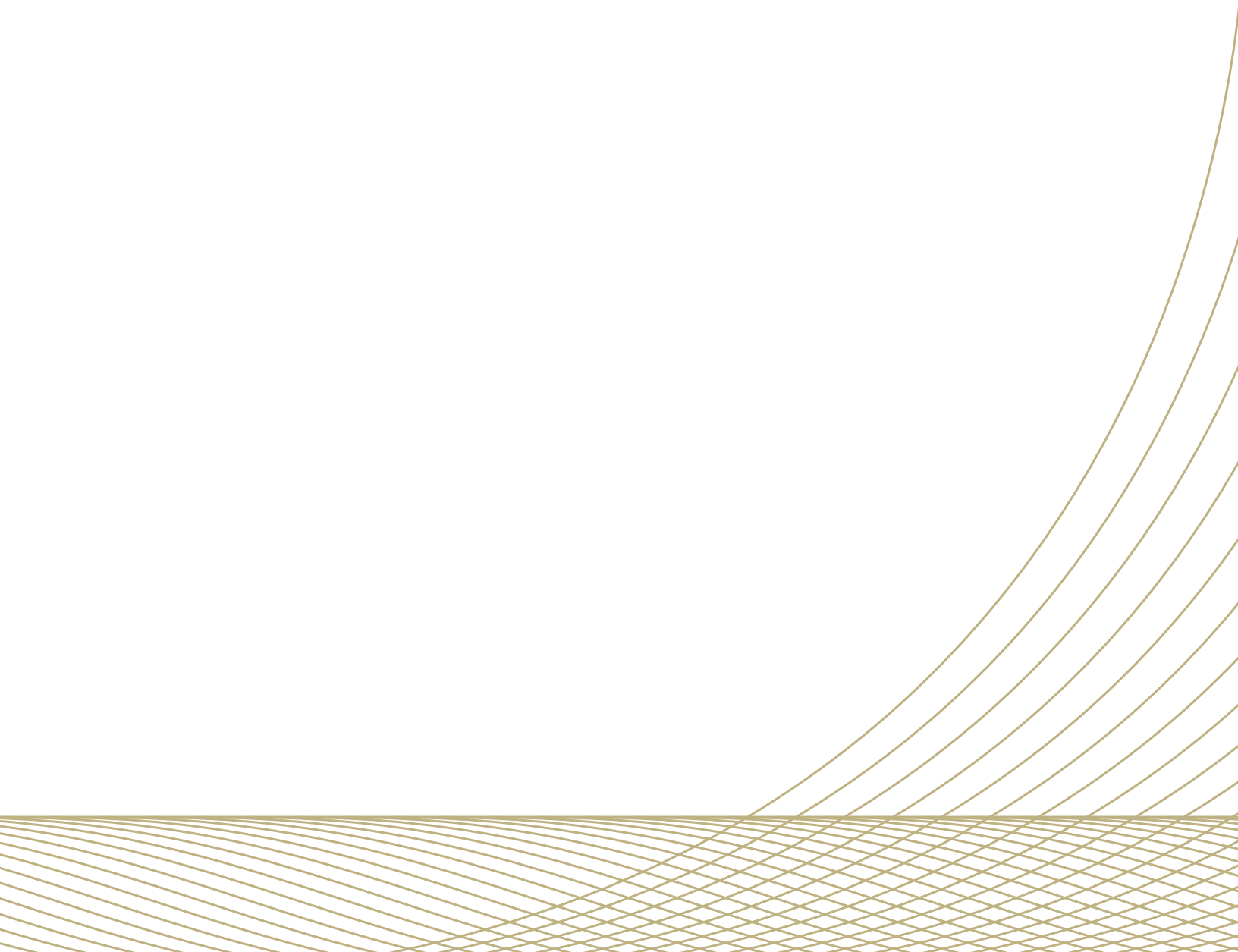


TABLE OF CONTENTS

INTRODUCTION
by Skytap Corporate Communications Manager, Noel Wurst

NOEL WURST

No matter how you define DevOps, the software community agrees that "DevOps is not about the tools; it's about the people." At Skytap, we believe that DevOps' full potential is unlocked by enabling people who are passionate about building, testing, and releasing high-quality software that delivers real business results.

Skytap's own DevOps story revolves around our people and the experiences and stories they've collected during their collective decades of experience in software delivery.

In this eBook, we've included stories from both the development and operations points of view, covering key issues that include:

- DevOps prerequisites and outcomes
- How to avoid common DevOps pitfalls
- Continuous testing, deployment, and delivery
- Eliminating bottlenecks and constraints
- Areas for increased workflow efficiency
- Getting executive buy-in

We want to thank you for the time you've taken to learn more about our point of view on this seismic shift in our industry, and we hope that these stories help you on your own DevOps journey.

HOW I LEARNED TO LOVE ENVIRONMENT PROLIFERATION

by Skytap Director of DevOps Consulting, Kelly Looney

KELLY LOONEY

Software development is a strange craft. In some ways things seem to stay the same forever; we still sit at the same Unix command line from over 20 years ago and we use usually the same editors, but the software being developed feels pretty different. For one thing, we spend much more time selecting the software we will reuse than we do writing new code. For another, we don't need actual physical servers anymore; we can just conjure up servers whenever we need them. I love that part.

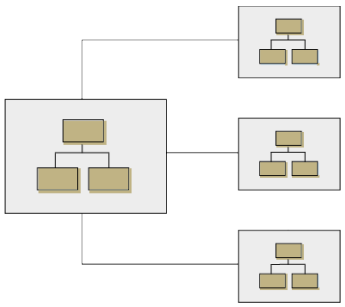
The constant theme over the last decade or more has been creating and delivering smaller and smaller chunks of new code, faster and faster. I used to develop code that wouldn't be used by actual customers until more than a year after I originally wrote it. Today, it's not unusual to get live feedback from customers for changes made just yesterday.

By necessity, the steps required to safely move code from my laptop to a production server have become small, fast, and highly consistent via automation. Agile made the argument that more releases made sense along with the techniques to iterate effectively. DevOps supplied great new automation technology for setting up my applications, and the cloud gives me whatever machines I need—quickly and cheaply.

Today's software applications often have to be usable by very large numbers of people and they don't simply run on a laptop or a single server; they require a complex conglomeration of machines, networking, and storage all load-balanced for scale and secured with firewalls. Making sure your software works in such a complex setup, known as an environment, is something pretty different from the programming experience of years past. Help is required.

The need for on-demand virtual environments

As software has become more complex, with databases, caching, message queues, and more—it has become imperative to accurately model the production environment in order to do valid testing before deployment to production. Serious organizations have done this via expensive physical replicas, or at least software-identical test environments run in labs. The problem is that as new code changes are being delivered faster, organizations have to schedule, juggle, and constantly reset their test labs.



The test facilities quickly become less like production (and therefore less valid) and more of a roadblock or impediment to real development over time. The need for multiple valid test environments quickly becomes clear as the need to update and maintain environments also becomes imperative.

These test environments need to go beyond newer-age agile shops. Even when an organization has resisted shorter release cycles, more often than not it has added technology and/or legacy applications via acquisition or merger. Getting physical test labs in place to model every production

system is becoming an increasing challenge for every type of organization. And when an organization decides to modernize and move towards the widely accepted continuous delivery (CD) model of development, the need for environments grows exponentially.

Essentially, with CD, a "delivery pipeline" is created that charts each step software must go through in order to be considered ready for "promotion". To progress from one step to the next, a clean test environment should be used to run whatever tests are required. It's not unusual for there to be four or five different quality gates per delivery pipeline, and it's also not unusual to have many pipelines in "play." The number of independent tests (with clean environments) quickly gets beyond what physical test labs can support.

The answer is on-demand virtual test environments. "Virtual" means that the whole test environment is captured in a digital form with no hardware. This includes servers, networking, firewalls, load balancers, caches, VPN connections, and so on, and means that generic data center assets can be used to create the complete environment. "On-demand" means that that you can quickly instantiate a fresh version of your environment. This typically implies a cloud (public or private) that can be used to quickly allocate resources and then discard them when testing is completed. It is interesting that the discarding of test environments ends up being a highly valuable action. This protects against environment drift (via undocumented alterations), which is one of the forces that tends to make test environments diverge from production, and then result in invalid tests. It is much better to start each test with a fresh and validated environment.

Proliferate or die

The only way to break up large-scale software development in the small chunks needed to make modern systems agile and stable is to enable lots of parallel, fully valid tests. Using flexible and automated infrastructure is the only way to support this in an economical way. Having dozens (or even hundreds) of physical test labs, and keeping them maintained is just not feasible, even for organizations that could afford to do so.

The best answer is to turn our infrastructure into code, use cloud services where possible, and stop making testing a bottleneck in our development processes. This approach certainly introduces some new problems, like, "how do we manage and maintain all these environments?" but it solves core problems and allows our products and services to evolve as they must in today's changing markets.

Too often, tests get regulated to the hard problem that much of the organization wants to avoid or pushed to an under-resourced quality assurance organization. But testing is too important of an activity for this to really work. Testing today has to start shortly after development begins, and it changes quickly, adding more fidelity, as each change nears production. It is absolutely worth the investment to make sure that tests have a valid place to "live" that makes them easy to run, easy to build, and easy to maintain. It takes almost every part of the organization working together to get a good suite of tests in place, and every organization that has such a resource swears by it.

Twenty years ago, I wrote a bunch of code, compiled, and built it into an executable (the "environment" was usually just a desktop computer), and then either began painstakingly testing the result myself, or I gave my program to someone else who started the test process from scratch. If

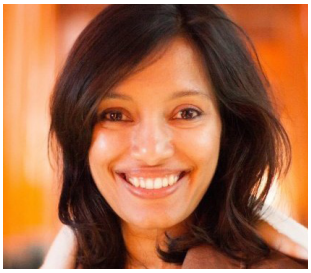
multiple people's code went into the test, figuring out what went wrong (and who was to blame) took a tremendous amount of effort.

Modern developers make small iterative changes many times a day, and each one is checked and tested via continuous integration. These small changes are combined with dozens of open source packages, servers are provisioned programmatically to accept these changes, and then updated servers are placed into an environment that allows them to work and scale traffic as required. Ideally, tests get run (automated ones are nice...) that only try to focus on a single source of change. That makes integration pretty easy.

The rub is that we may have tens or even hundreds of small changes to check— that's where lots of environments pay for themselves. It makes programming easy and accessible again, and I love that part, too.

IS AGILE A PREREQUISITE FOR BEGINNING DEVOPS INITIATIVES?

An Interview with Skytap Senior Director, Client Engagement and Services Delivery, Raksha Balasubramanyam



RAKSHA BALASUBRAMANYAM

DevOps is often described as an extension or an evolution of agile software development, but is it required to first "be" agile in order to begin your DevOps journey? We sat down with Skytap Senior Director, Client Engagement and Services Delivery Raksha Balasubramanyam to get her thoughts, advice for how to achieve early-stage DevOps benefits, and the executive buy-in needed to scale success.

Is "being agile" a prerequisite for beginning a DevOps initiative?

It is not a prerequisite, but it can be a key DevOps enabler. A successful DevOps implementation requires an organization-wide focus on delivering innovative features that provide constant support for the business. Consequently, less time is spent on re-work, production issues, firefighting, and other wasteful activities.

Whether an organization adopts agile development or lean methodologies to help move toward this end-state is entirely up to what is conducive to their style. DevOps goes beyond agile or lean methods of delivery to encompass an organization's culture, collaboration, and employee engagement

If you're still in the process of "going agile" but are far from it – can executive level buy-in for DevOps initiatives be difficult to get to begin another project/ transition?

As with everything new and unproven within an organization, DevOps initiatives can be difficult to get full executive buy-in at the enterprise-level in the early days. However, implementing DevOps is not an all-or-nothing proposition.

A team can start small, focusing on gaining efficiencies and showing value within a single product or area. As you implement agile practices, also working towards improved automation, increased collaboration between stakeholders, eliminating waste, and establishing metrics and dashboards for visibility across the organization can help you prove gains from DevOps. Such visibility, in turn, helps with championing the cause and opening doors to other areas. Start small, aim big, and focus on continuous improvement.

What are some constraints and bottlenecks that agility alone isn't designed to remove, and where DevOps can help?

We should recognize the difference between adopting agile methodologies for development and test and gaining overall agility. Agile development and test methodologies help teams get a better product developed and tested according to business requirements, with necessary demos and sign-offs from all involved stakeholders. They also allow for product changes to be implemented faster to meet customer demands.

However, to truly benefit the business, there is still a need to be able to consistently and predictably take a high-quality product to production in a timely manner. This is where agility comes into play.

Today, there are tools and accelerators that were not available a few years ago that can

help organizations consistently deliver quality releases to production. Support for continuous integration, on-demand and self-service development and test environments, test automation, and deployment tools all help in removing bottlenecks in tooling and transformation. Besides using the right tools, leadership prioritization of such initiatives along with the allocation of the necessary budget is a requirement.

Once such prioritization is made, it is imperative to create and maintain capacity (slack) in the system. Adding more to an already full plate will only result in a lack of throughput, under performance, and low morale and retention—something that being agile in development and testing alone will not solve.

For those who've struggled to see agile as possible in their organization, what are the benefits, ROI, and rewards that a DevOps transition brings that can make it worth fighting for?

I like to think about this on the lines of the "20-mile march," a key concept from Jim Collins and his book, Great by Choice. It includes an analysis of the success of several companies and what differentiates them from others that were less successful. The 20-mile march refers to Roald Amundsen and Robert Falcon Scott's journey to the south pole and their preparation, leadership, and discipline.

Where Amundsen was consistent with his pre-planned regimen of approximately 20 miles of progress a day, no matter the circumstance and conditions, Scott was making decisions driven by weather changes and pushing (or holding back) his team's progress depending on the same. Amundsen and team made it to the south pole right on pace, having averaged approximately 15.5 miles a day, while Scott's team unfortunately perished.

The consistency, progress, and discipline of Amundsen's team is akin to what we seek for software delivery in today's technologically advanced, customer-driven world. The ability to work as a single organization, deliver value to the business constantly in a predictable manner, and be adaptable to changing environments are becoming table stakes for progress and survival. And at the core of these goals lies the culture and leadership of an organization. Implementing DevOps practices within a team, department, and organization can help the journey to their south pole become a successful one.

TIPPING THE DEVOPS SCALE

An Interview with Gary Gruver

Gary Gruver and Tommy Mouser published Leading the Transformation: Applying Agile and DevOps Principles at Scale in late 2015, and we were lucky enough to interview Gary to learn more about how enterprises can set themselves up for success as they expand their initiatives across multiple services in their business.

We highly recommend reading Gary and Tommy's book, and we hope you enjoy the interview below.

*Note: This is an abridged interview; for access to our full conversation with Gary, including a recorded podcast, please visit the **Skytap blog**.*

Early in the book you say, "Continuous delivery tends to cover all of the technical approaches for improving releases, and DevOps tends to be focused on cultural changes."

There was so much confusion as to what DevOps was even just a year ago, but I think that separating the culture from the technology is a great approach to eliminating that confusion and knowing what's going to be required to implement DevOps successfully.

I've evolved a little bit in my thinking. When I've heard Gene speaking lately, he's really trying to say that DevOps is all the technical, cultural, and practices that it would take to enable a company to release code on a more frequent basis while maintaining quality, stability, reliability—all the "ilities." That captures it all because you can't do one without the other. Even if you do all the technical changes, but you don't get the cultural changes, your benefits are going to be fairly limited.

In the book, you state that applying DevOps principles at scale for enterprise solutions will require implementing continuous delivery. Why is that?

As you're trying to release on a more frequent basis, what you're really trying to do is integrate all the work in the organization and put it together and make sure that it's working on an ongoing basis.

Continuous delivery is the technical approaches that make that happen. It starts with continuously integrating all your code, testing it, and then it dealing with all the issues that you need to resolve to make sure your environments are consistent and giving you the same answers as code moves down your deployment pipeline from a developer's desktop to end of production.

The thinking is that, if you don't do the scripted environment and scripted deployment part of the equation, the feedback that you're giving to the developers is not going to be the same when they're checking in their code as it is when you're trying to get it into production.

You're just trying to resolve and deal with all those differences and make sure that you're consistent throughout the path. What you're trying to do is make sure that when a developer checks in their code, and it passes that first gate, that, that code will work all the way out in production without finding new and unique issues—which tends to happen if you let environment or deployment differences creep into your system.

What are some of the differences between continuous delivery and continuous deployment, and what are some of the benefits they provide?

Jez Humble makes a very clear distinction between these two. Continuous delivery is all the discipline and rigor around putting everything under version control whether it's the code, the tests, the environment definition, the deployment process. That rigor of putting those things under revision control and automating them to where they're reliable and consistent across all the different stages of your deployment pipeline, that is what I think of as continuous delivery.

Continuous deployment is a business decision that says, "Okay, I've got all that structure in place. I'm no longer constrained by the ability to deploy code in the production by my development process. Now I need to decide whether I want to do it with every check-in or not." In a lot of cases, some businesses, that wouldn't make sense and it wouldn't be a value. Especially if you're doing embedded things, where you're not in control of the deployment.

As you start to automate your environments and get consistency across them, and you start to automate your deployments and get consistencies across them—as you do the same thing with testing—you are going to be solving issues that your organization has been struggling with for decades. What you're doing is increasing the frequency of those things to the point where you can no longer brute force your way through, issues that have been plaguing you for a long time, and you have to start fixing those things with automation.

You talk about how the need to determine if your organization will embrace cultural changes up-front is very important. How do you determine whether you're going to be able to get that buy-in early so that you can begin your DevOps journey?

I think one of the first places I started, if you look at my book, we talk about forming a team to lead the continuous improvement process.

Getting the executives aligned on leading the transformation. If you've got the leader of QA, and the leader of Dev, and the leader of Ops, and the leader of security all driving on different sets of objectives—it's going to be hard to get the teams to come together to align on delivering improvements and what to improve, and how to improve.

The first step is, can you get those leaders to come together and start to lead monthly checkpoints where they agree on what we're going to fix and we set that in place? These are the objectives everyone is going to drive to achieve over the next month. Then the job of the executives and the management team is to monitor the progress and play the role of an investigative reporter in the organization trying to figure out what got done, what didn't get done, and what did you learn so that they can set the objectives for the next month.

If you can't get that aligned, I think that's where you should spend your energy and your effort. Maybe you need to take it up in the organization and bring the book to a higher-level executive to see if you can get that alignment. Or bring a consultant like myself to work with the team to see what's in the way of them coming together and agreeing on those common objectives. That's one of the big cultural changes, is getting that team aligned and engaged in leading the transformation. One of the reasons I wrote the book is, I think that's one of the biggest holes in organizations is making that happen.

Number two is, can you get the development organization to respond to the feedback that you're designing and building in on the deployment pipeline? If they want to go off and just develop code and not make sure it's working in an operational environment and not prioritize making sure that's going to work, then you're going to get limited benefits.

Set up a simple CI environment and get them in the cultural process of keeping a few automated

tests running. It doesn't have to be every test automated and it doesn't have to be the entire enterprise's pipeline stood up, but can you get a simple CI environment that they're responsive to, and they're reactive to, and the work ventures out. If you can do that, then you've started down that cultural transformation.

How does establishing trust early on help avoid some of the common, early pitfalls of DevOps initiatives, and how do you get early buy-in?

I think the key part with the executives is making sure you get their fingerprints on it. That it's their plan that they believe will help meet their business objectives. If you can get them focused on their business objectives and how what you're doing and the transformations you're making is not doing DevOps to do DevOps, but using it to solve one of those problems you identified. Are you having quality problems? Are you wanting to release more frequently? Are you trying to improve your productivity? What are those things that you're trying to do?

If you can't get the executive teams to believe that it's key to their business success then you are going to struggle. You need to start by getting their fingerprints on the plans so they own its success and help lead the organizational change. Another thing that I say is, get that aligned team to go down that path together because if you leave compliance or somebody out of it, they're likely to throw rocks at the changes you're making. If you start with a set of business objectives they can all agree to and you get their fingerprints on it, then it becomes their plan and they'll make their plan successful.

WHAT ROLE DOES TRUST PLAY IN DEVOPS?

By Skytap VP of Operations, Paul Farrall

PAUL FARRALL

I've been involved in the DevOps community since before it was called DevOps. I was a pre-publication reviewer for The Phoenix Project and I was a pre-publication reviewer for The DevOps Handbook. Rather than talk to you today about DevOps at Skytap, which is a little bit of a unicorn example, as a cloud services provider who builds solutions for DevOps implementers, instead, I thought I would relate a couple of examples from my experiences prior to Skytap, when I worked in a more traditional IT shop. Before I joined Skytap, I was the VP of Operations at Big Fish Games for five years here in Seattle.

At Big Fish Games, I had overall responsibility for production operations at Big Fish's three production data centers in Seattle, Virginia, and Luxembourg. I also had responsibility for internal corporate IT and information security there. We did around a million game downloads a day. Games were localized into eleven different languages. We processed essentially every major currency, and right before I left Big Fish, we started taking Bitcoin as well. From an IT perspective, it was a global eCommerce company selling digital widgets on the internet. We had around 800 employees in our five offices in Seattle, Ireland, Oakland, Luxembourg, and Vancouver, BC.

My first attempt to introduce DevOps at Big Fish failed miserably due to cultural misunderstandings and a fundamental lack of trust between the development and operations teams.

You really need a certain minimum level of trust to get DevOps started with any chance of success. This trust is important because you're blurring traditional boundaries between development and operations responsibilities. This can be scary for operations because it's scary for ops teams to involve development in production operations. It just feels wrong from a historical ITIL-type perspective, and operations teams may also know development as a team that always breaks things. It can be scary for development to involve operations in the development process because they may know operations as the team that always says, "No."

Here's a simple, although embarrassing story, to illustrate this point. At Big Fish Games here in Seattle, application development teams equated DevOps with developers carrying pagers: an idea that they really did not like. They were highly resistant to this. Every time I brought up DevOps, they thought I was engaged in a secret plot to make the developers carry pagers. What was interesting about this is that I had no intention of asking developers to carry pagers.

The term DevOps covers a large range of continuous improvement techniques. Some organizations have developers carry pagers, and it works for them, but this is certainly not a prerequisite for DevOps. What's worse, is I had no idea they were thinking this because they were afraid to bring it up for discussion. We failed in our first attempts to implement DevOps because the trust level between development and operations teams was so low, that we couldn't even tell each other what we were thinking.

Here's another simple failure example. I noticed while managing the release efforts that release schedules for the various development teams ranged from a couple weeks to a couple months—in other words, too long for a hyper-competitive industry like gaming. I asked the development teams if they would like if their release engineering team made it possible for them to do lightweight co-releases every day, or even multiple times per day. I thought this was an innocuous question. In fact, I thought I was being a little clever by playing dumb here. I assumed I'd get an enthusiastic high-five, "Yes, we'd love that!"

Instead, I was surprised that I got a violent, "No!" response; "We don't want that!" This confused me for a while until I realized that the development team thought I was in cahoots with the product management team to put the squeeze on them. In my mind, I was proposing to them to make their release process more lightweight so they could break up their work into smaller, more frequent releases, with less stress for everyone, but I didn't understand their fears and their problems. I wasn't looking at it from their perspective. They also didn't trust operations. They didn't take us at face value.

I pictured them being able to release code whenever they want, which seemed like something they would like. In their minds, they pictured a product manager leaning over their shoulder every day, randomizing them with demands for new features to be released, like right then, while the product manager was standing there. We were coming from two different worlds. This was not the kind of problem that you could resolve with logic. The solutions I was proposing, and the working procedures I was proposing, were so far outside the development team's experience level, that they couldn't extrapolate from their current state to the world I was describing through simple logic.

It was going to require some trust to get us over this barrier, into this new proposed world order. How do we get past this impasse? To be honest, there was no silver bullet. What it required were years of conversations, education, and trustworthy actions. There were three specific strategies I employed consistently over time. These three strategies were pretty successful. In particular, number one, I somewhat sneakily stopped using the term DevOps, as soon as I discovered that this was a scary term for them. Instead, I just talked about continuous improvement and specific continuous improvement methodologies that we wanted to implement.

Number two, when deciding which of these DevOps, aka, "continuous improvement methodologies," we wanted to implement, I specifically chose ones that would benefit development teams, as opposed to benefiting operations or other teams. The development teams noticed this over time. They could see that I was doing this, and I was going out of my way to help them out, and this helped build trust. This wasn't a one-time thing. We did this consistently. The third strategy that I employed, was I took every opportunity to maximize the number of daily interactions between development and operations team members. It's easy to distrust or hate on someone that you don't know personally, or that you don't have to see every day. It's a lot harder to hate on someone that you have to work with on a daily basis.

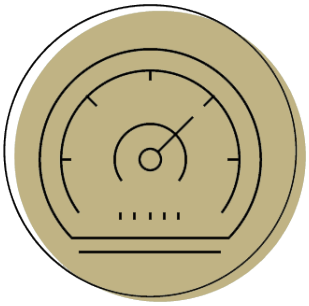
Let me give you a couple of success examples. One of the early DevOps techniques that I implemented was creating a dedicated operations liaison that was embedded into each development team. This liaison acted as an ambassador for that development team into operations, and it helped smaller development teams navigate our large, complex, and sometimes opaque operations

organization. This operations team member would attend their daily development standups, team meetings, and project meetings. This was a big success because the development teams could see that this operations team member was helping expedite their work through the operations organization.

After the development teams got used to working with this person on a daily basis, we pushed a little bit farther and we got the development teams to agree to include operations officially, in architectural discussions, at the beginning of application design. Now, at first, development teams were skeptical that operations could provide useful input here, but the important thing is we had gained enough trust that they weren't fearful of including operations in the discussions. They were skeptical that it would add value, but they weren't afraid that we would cause problems. This was a big win.

It was a little bit unusual for development teams at first to include operations in design discussions, but they quickly saw how obtaining input from operations early on, before a product was built, resulted in better quality and fewer headaches for everyone when we launched into production. I also made sure that we put our most senior and diplomatic operations team members into this role. We also included InfoSec after a while, too.

A larger project we took on after we started gaining momentum with this DevOps effort, was our release engineering team built a continuous delivery framework that allowed game development teams to publish their code directly to production without direct involvement from operations. This was a big win for everyone. It wasn't something that we could have taken on immediately because we didn't have the process, and we didn't have the bi-directional trust necessary to implement this, but after we got going, we reached a state where we could deploy something like this.



We built this tool and released it to the game developers. The game development teams were very happy because they could push product code as fast as they wanted without operations being a bottleneck. The operations team was happy because they were no longer in the spotlight as a bottleneck, and also designing and building release tools was much more satisfying work for the Ops release engineering team than manually pushing code to production. The business stakeholders were happy because, in the end, game features were getting to production much faster.

It felt a little weird at first, letting developers push code straight to production, but the operations release engineering team maintained their release tools, so all the necessary controls, safeguards, audit trails, etc. were built into these tools, and we quickly got used to having the developers push the release button. Later, the operations team extended on this self-service team, and we built an internal private cloud infrastructure that allowed people to spin up their own virtual infrastructure that didn't require any involvement from operations.

In conclusion, trust is a requirement to succeed at DevOps, but we're often handicapped right out of the gate because of fuzzy DevOps definitions and the dangerous territory of mixing up boundaries between development and operations teams. Whatever your definition of DevOps may be, making sure that a high level of trust between development and operations teams is a prerequisite for your journey will ensure that even greater trust—and the positive business impact that comes with it—are your outcomes as well.

CONTINUOUS DELIVERY DECISION POINTS

An interview with Kelly Looney

Skytap director of DevOps consulting, Kelly Looney, has presented in front of—and learned from—a number of organizations around the globe, and in this interview, he shares some of his advice on how to successfully supporting a continuous delivery model.

KELLY LOONEY

You've said that DevOps is evolving from focusing on build automation to full-on continuous delivery. What's causing that evolution?

I think we have a pretty good handle and a lot of good technology that helps us automate builds through deployment, from Ant/Maven, Puppet, Ansible, Chef, and Salt, using repositories from Git to Artifactory. But automating that stuff does not make much difference if you are not delivering more often, and that is what it is all about. Small changes happening with great frequency is the future.

Continuous delivery is not the same thing as DevOps, but the two concepts tend to go hand in hand. Put together, they are a big part of the definition of "modern" software development.

During your DevOpsDays Detroit presentation, you said, "Too many organizations approach continuous delivery by automating the wrong things and just increase the amount of (often poor quality) code they have to maintain." What are the wrong things to automate, and why has that happened in so many organizations?

Well, it's OK to automate whatever you can, just bear in mind that your automations become a part of your new production code base. Throwing low-quality code in there, just because you can, may not help you. The key point I am making is that you want to carefully craft the precise tests you need to get code to production, because you will be running those thousands or even millions of times. Your delivery pipelines are a truly strategic project.

In regards to the culture required for DevOps and continuous delivery to work, you've said that asking "Are our teams empowered to deliver what's needed to meet our goals?" is important when looking for organizational issues or blockers to success. What empowers these teams?

Empowered means responsible and with the ability to make decisions. Too many organizations have developed into mazes of "gates" which are groups (also known as silos) that are empowered to say "no" to putting something into production. With enough of those gates, organizations tend to simply freeze up.

Also, when a development group isn't responsible for security, they won't necessarily think about it. With "real" DevOps, teams are responsible for security and scale, and other teams with expertise in those things act as consultants rather than gates. The team creates a product and they stand behind all of it.

Lastly, what kinds of challenges are there in trying to implement things like DevOps and continuous delivery when you're dealing with different speeds of innovation between newer, born-in-the-cloud apps as well as older, legacy applications?

We work really hard to help organizations take large and complex legacy applications and create viable test environments that allow these applications to be safely changed. This supports the goals of continuous delivery while recognizing that highly changeable, microservice-style architectures are unlikely to spring up overnight – so making cumbersome application environments quickly adaptable is key.

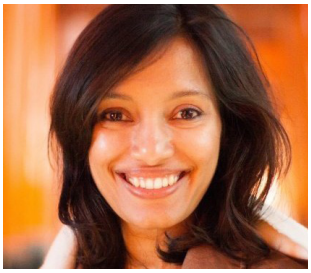
Gartner's "Bimodal IT" model has really caught hold as a way to talk about today's IT environments, and I think that there are two reasons for this: 1) Gartner is pointing out something that is obviously true. Even though we know of some better ways to build software, most of the software we already have and already depend on was not written in this way, nor was it architected with modern rates of change in mind. 2) Somewhat unfortunately, many organizations take this as license to only change their processes for "greenfield" applications, while leaving "systems of record" processes as-is. The theory is that the risks of changing systems of record are too high.

Jez Humble, one of the authors of Continuous Delivery, points out where this goes wrong in an article he also wrote, titled, "The Flaw at the Heart of Bimodal IT." He shows through solid research that high performing organizations don't take the risk of standing still with their systems of record and are able to improve BOTH agility and stability at the same time by driving smaller changes at a more frequent pace.

We agree with Jez that just "doing" new applications in new ways is not the way market leaders are going to behave. The right answer is to make all new work support a continuous delivery model regardless of what system is being changed and allow your applications to self-modernize over time. This approach pays for itself—it takes confidence to enact— and Skytap's technology can play a role in putting that confidence in place.

A DEVOPS USE CASE: INCREASING WORKFLOW EFFICIENCY TO MEET CUSTOMER DEMANDS

By Raksha Balasubramanyam, Skytap Senior Director, Client Engagement and Services Delivery



RAKSHA BALASUBRAMANYAM

One of the greatest benefits of DevOps is the automation of manual efforts that offer no advantage by being performed manually, and only delay defect resolution and time to market. One of our customers was recently able to eliminate the bottleneck around recreating and triaging issues for their clients, and before leveraging Skytap Cloud environments, they were dealing with all-too-common issues in the software industry.

Client: A global application software provider that sells an integrated suite of trading and risk applications to the capital markets function within banks and other companies participating in the world's financial markets.

Background: The client has multiple organizations servicing their customers from development, QA, and support. In a given year, there are three major releases, eleven maintenance releases, and approximately four-hundred patches are pushed to production per month on an as-needed basis.

The delivery (development, test, ops, support) teams have an established SDLC process and have a two hundred and fifty-member team located globally across the United States, India, and the UK. The client employs a mix of agile and waterfall methodologies for delivery with formal handoffs among dev, QA, support, and operations. Their goal was to improve time to market and internal workflows to efficiently meet customer demands.

Scenario one: Manual information exchange and collaboration among delivery teams

The support team, as tier one support, engages with their customers first, and downloads a customer's database and code version to triage issues in an environment created in a local machine.

Support will share details around the customer's specific configuration and data verbally, or provide documentation to help development and QA replicate client issues and then work on appropriate fixes. Once a fix becomes available, support will then recreate the client configuration in a local environment and test the fixed build prior to providing it to their customer.

Scenario two: Aspirational continuous integration

Code check-ins throughout the day trigger builds by the CI server which are then made available to QA team for testing. The global QA team of fifty engineers spends up to an hour per day setting up local environments and configuring the same for the build chosen for testing (not all builds made available are tested). Testing is a combination of manual and automated test suites being run in parallel.

Scenario one resolution: One-click collaboration

The problem with scenario one is the manual nature of information exchange between support and development teams and the effort expended to recreate the customer issue multiple times prior to solving it.

The use of Skytap Cloud environments enable support engineers to dramatically reduce the time it takes to triage customer issues. After reproducing such issues, support engineers can save the exact state of an environment, and then, with only a single click, share copies of environments with development and QA teams. This results in immediate access to the issue, inclusive of the data and configurations, so development teams can immediately move into defect resolution.

Scenario two resolution: Smarter, scalable automation

In scenario two, copious amounts of time and effort were being expended in the setup and configuration phase. This stage should ideally be made instantly, and easily repeatable in order to allow a focus on actual testing and automation improvements.

In following DevOps principles, teams need better processes, increased collaboration, the creation of a seamless, automated delivery pipeline. Removing immediate, tangible constraints so as to allow for capacity in the team to work toward these DevOps objectives is key. This is where Skytap Cloud environments brought value to this customer in particular.

Skytap Cloud environments allowed for the delivery pipeline to be automated to pick up the latest build from the CI Server, launch an automated test suite, and upon completion, create a "template" of the requisite test environment with the successful build. This template is then shared such that each test engineer could then create a test environment at the click of a button (or via API) in the exact configuration needed while also ensuring that the entire team was consuming the same version.

CONTINUOUS DELIVERY OF FULLY FUNCTIONAL ENVIRONMENTS

By Skytap software engineer Joe Burchett

JOE BURCHETT

Robust DevOps practice necessitates modular and automatic construction of software and infrastructure. At Skytap, we leverage our own customer-facing services to construct a delivery pipeline that allows engineers in operations, testing, and development to collaborate at every point of the software delivery lifecycle (SDLC). We ultimately deliver working clones of our production environment that can be launched on-demand, without incurring the cost of provisioning and deployment.

This article provides a detailed description of this continuous integration and continuous delivery (CI and CD) pipeline, with emphasis on how our process enables us to deliver fully functional pre-production environments. For a discussion of the motivation behind this effort and our observation of how it has improved software delivery at Skytap, refer back to part one of this series.

For this discussion, we'll begin with an abstract overview of CI and CD as we model them. We'll then use these models to demonstrate how we generate the toolchain itself. Finally, we'll show how these tools are combined to produce Skytap Cloud templates as packaged artifacts; engineers can use these templates to deploy their own fully-functional, pre-production environments at any time.

OVERVIEW OF CI/CD AT SKYTAP

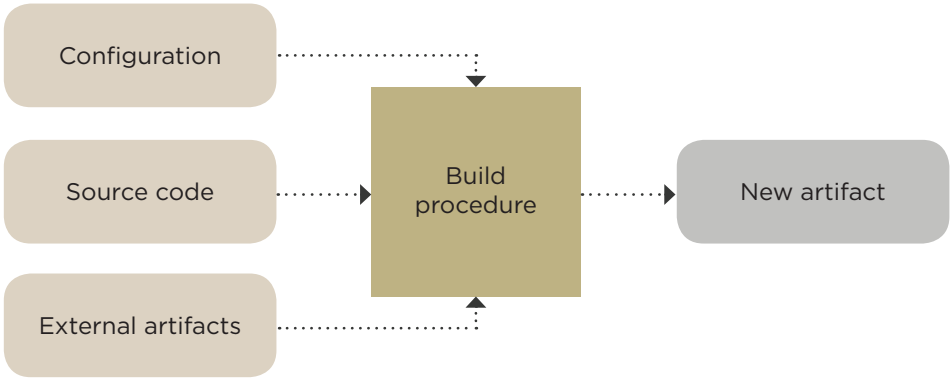
For this discussion, we consider the most basic unit of useful software to be the artifact. An artifact is the result of a build job that persists after the job ends. Examples of artifacts in our usage include packaged dependencies (libraries), Jenkins jobs, Docker images, and Skytap Cloud templates.

There are three basic abstractions that we use to model the Skytap Cloud CI/CD pipeline: the artifact production model, the continuous integration (CI) model, and an aggregate of these called the continuous delivery (CD) model.

Note that our definition of CI/CD may differ from what you're familiar with. Some organizations consider automatic deployment to be a step in the CD model, but we don't. Instead, we treat deployment as a distinct activity from delivery. We use continuous deployment in a few specialized cases (you'll see later that we automatically deploy Jenga artifacts to a Jenga service and back to our Jenkins server for later work), but we don't currently employ continuous deployment as a general principle, and won't explore it in depth in this article.

The artifact production model

Building any software artifact can be modeled as an activity in which you combine build configuration, source code, and any required external artifacts to produce a new artifact:



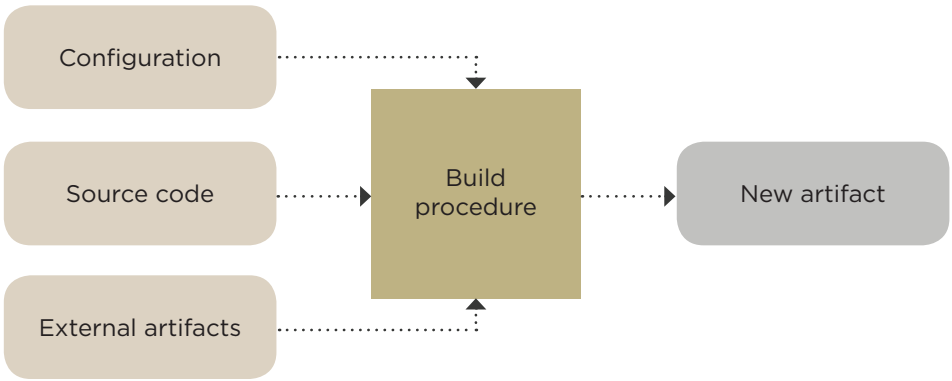
We use this simple idea to model build and delivery jobs of varying complexity. In some cases, we might not require the source code (if we're simply combining artifacts from previous builds, for example). Sometimes we don't need external artifacts (for example, if we're building a primitive that doesn't require external dependencies). Additionally, configuration may be implicit—it's always there, in some form, but sometimes an artifact-producing build includes configuration that you don't see (configuration of the build server itself, for instance, is implicit for most builds).

The continuous integration model

The goal of CI is to automatically build and validate a new artifact for every change. Builds in the artifact production model are not necessarily automatic or validated and do not necessarily occur after every check-in.

If we zoom in on a build process from the artifact production model, we can configure that build as a series of steps that automatically trigger their downstream neighbor. Some of these steps are responsible for building the software, and some of them are responsible for validating the source code or the produced artifact. This gives us a CI model that looks like this:

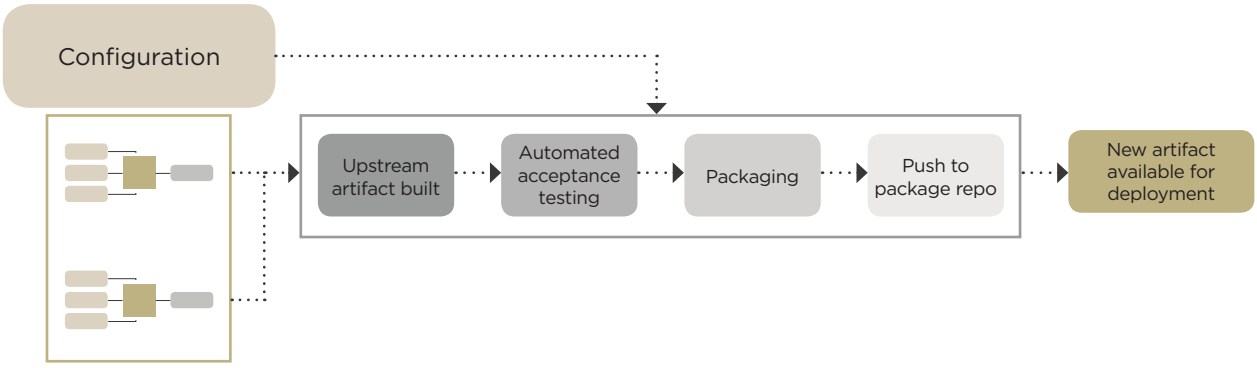
Every process using the CI model begins with a check-in (code, configuration, artifacts) and ends with either a failure in some stage or by producing a validated software artifact.



It's valuable to consider the CI build model as a sequence of steps in a single build process, rather than as a series of chained builds. It doesn't make sense to treat each step as a discrete build because there isn't a useful artifact left over after each step.

The Continuous Delivery Model

When we combine the artifact production model and the CI model, and use the construction of upstream artifacts (instead of code check-ins) to trigger new builds, we get a slightly more complex aggregate. The result is our CD model:



The CD model will automatically trigger a build when there are changes to source artifacts, and the build process will automatically produce, package, and deliver a new artifact.

Astute observers may notice the resemblance to the artifact production model—we feed configuration and artifacts into a well-defined process and deliver a packaged artifact to wherever it needs to be (package management systems, Docker image registries, etc).

You may also notice that the build process itself is very similar to the CI model, with a series of discrete steps that automatically produce and validate the desired result after each upstream change.

The CD model combines the artifact-production model and the CI model to create something that's specifically concerned with combining modular components into self-contained packages. It's distinguished from the CI and artifact-production models by these traits:

- Neither CI nor artifact production is concerned with delivery. Delivery-specific activities include: integrating external dependencies, packaging (for example, creating a Debian package or a Docker image), and shipping the resulting artifact somewhere that it can be used for deployment or downloaded directly (for example, a package manager).
- Some of its inputs come from upstream builds. Build triggers for CD are other builds, while CI builds are triggered by changes to source code or configuration.

The CD model is, therefore, a specialized case of the artifact production model that is specifically concerned with combining the results of other builds into a new, aggregate artifact and preparing that artifact for deployment.

It's important to distinguish CD builds from simpler builds because it gives us two important

points of flexibility. First, we can more effectively distribute builds and allow individual teams the autonomy necessary to manage their builds. Second, it allows us to structure a delivery pipeline as a set of discrete stages. Discrete stages help avoid unnecessary build work (we only need to build the components which have changed, rather than the entire platform), and they help to establish a clear, modular chain of events that starts with a check-in to any one of many services before ultimately producing complete, pre-packaged environments running the entire Skytap Cloud platform.

OUR IMPLEMENTATION

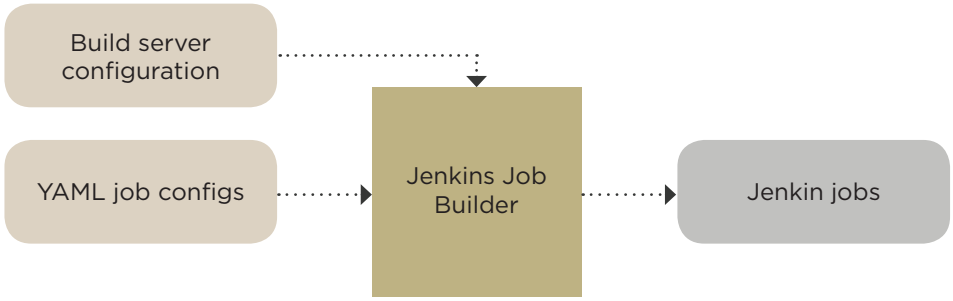
Let's take a look at how we implement these models.

We'll start with the build server and then move on to how we deliver our tools back into the build ecosystem (including, Jenga, our environment provisioning tool). Finally, we'll look at how this build process is combined with Jenga and the power of Skytap Cloud templates to produce full working environments as artifacts.

Building the build server

Our CI/CD workflow begins with a Jenkins server. The Jenkins configuration, including plugins or other server-scope items, is managed with Puppet. We install and configure Jenkins on a server that is launched from a Skytap Cloud template. If you're keen to apply the models outlined above, the artifact production model would be a good place to start: the inputs are the Skytap Cloud template, Jenkins and its plugins, and Puppet modules. The build process is running Puppet, and the artifact it produces is a functional build server.

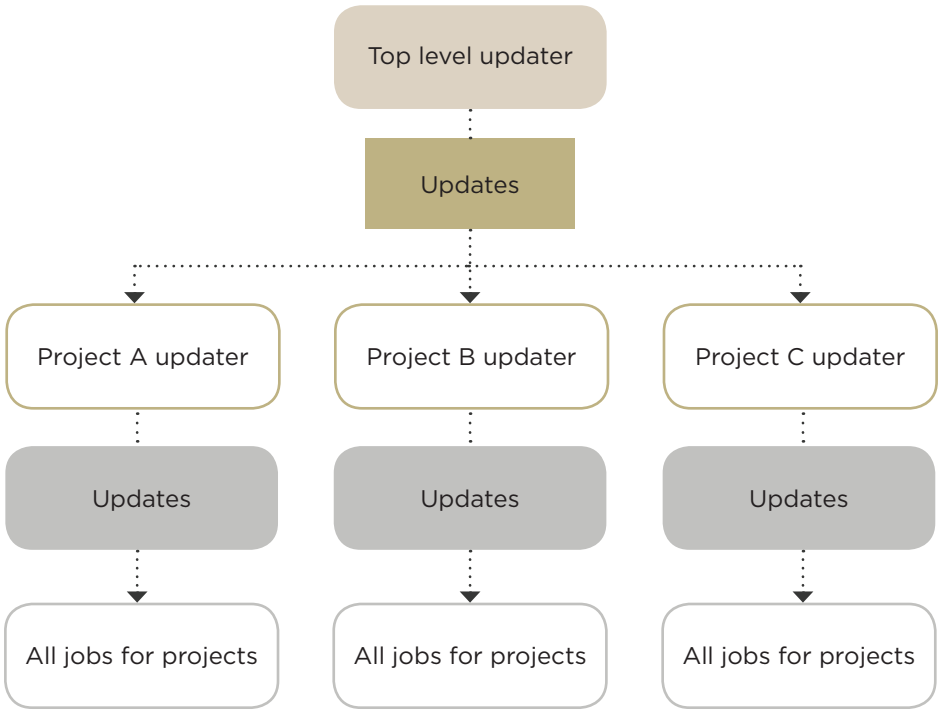
With a functional Jenkins server, we're ready to configure our build jobs. "Jobs," in this usage, are any processes that we run on the Jenkins build server—this includes build jobs that produce service artifacts, garbage collection jobs to remove old templates, packaging and delivery jobs, and upkeep jobs that keep the Jenkins server configuration up to date. We treat our Jenkins jobs as artifacts, and produce them with the artifact production model, like so:



Note that we're producing these jobs from (essentially) source code: by defining jobs with YAML, we can leverage the full power of source control to manage them. Instead of relying on finicky and time-consuming manual configuration, we can recreate all of the jobs on a new build server with minimal effort.

When we provision the build server, we use a shell script to bootstrap an initial, top-level "updater" job. This updater job has SCM hooks that trigger ***Jenkins Job Builder*** to automatically produce new

jobs each time a change is made to the job's YAML configuration. The updater will reconfigure itself in this manner, and will also configure several "child" updater jobs. The hierarchy of updaters looks roughly like this:



Generally, we'll map a Jenkins project to a Skytap Cloud service, such that one project produces artifacts for one service. Maintaining a separation between the top-level "updater" jobs (which set up the bootstrap for individual projects and the updaters for each individual project), we gain two important pieces of flexibility:

1. Teams can build a CI/CD process that makes the most sense for them, while maintaining clean separation between the build pipelines used by each team. Smaller, more modular builds make the build infrastructure less fragile.

2. Organizing jobs at the project level allows us a lot of flexibility in organizing build slaves and managing the build infrastructure without needing to understand the specifics of every project.

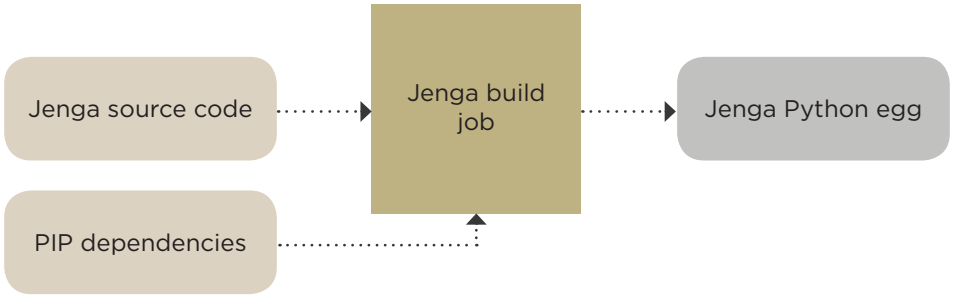
We have additional organizational constructs for maintaining build pipelines for current, previous, and upcoming releases; these all follow a similar process.

With this system, we're currently managing around 400 different jobs. We expect that the modularity of the process will allow us to trivially scale the build process as the organization continues to grow.

Building Jenga

After the build server, the next component in the environment delivery pipeline is our custom-built provisioning and deployment tool, Jenga.

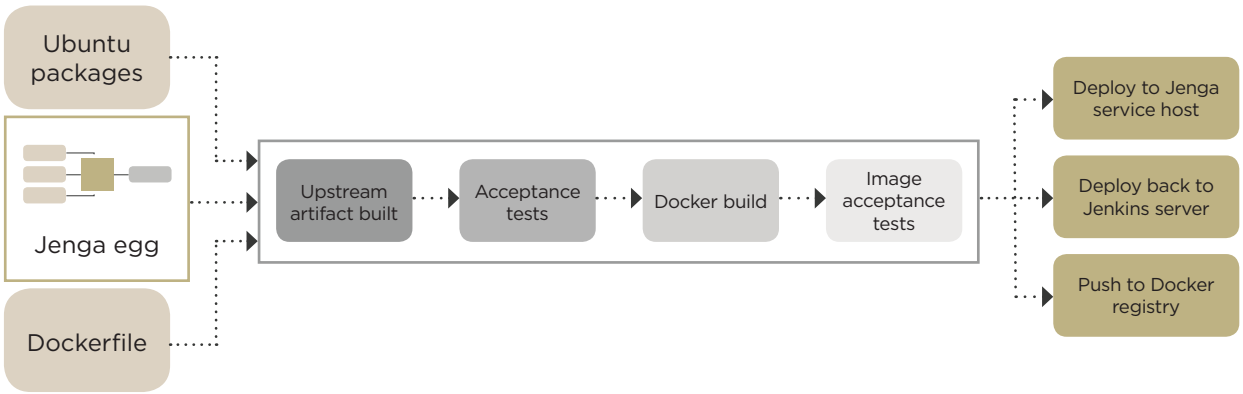
At this point, I'm sure you won't be surprised to learn that we use CI and CD to produce Jenga artifacts. The artifact production model for Jenga is straightforward:



The CI model is pretty typical for a Python project:

The CD model is more interesting. At this point, we package Jenga as a Docker image, and deliver

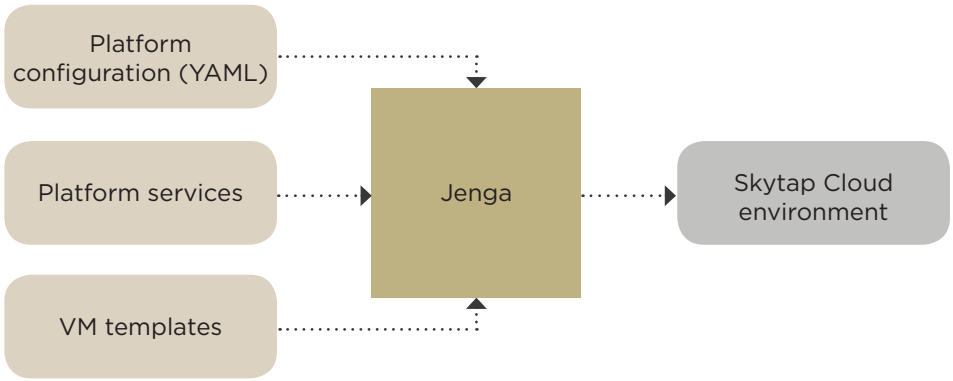
it to three endpoints: a host running Jenga as a service, the Jenkins server, and our private Docker registry. Each of these endpoints uses Jenga to serve a different need; we'll discuss these three options in the "How it Works For Us" section.



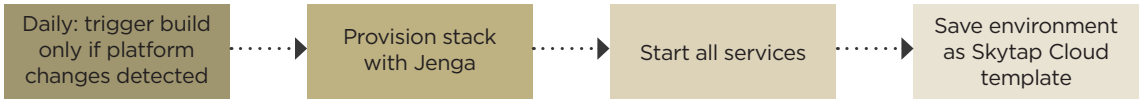
Building environments

With Jenga automatically delivered back to Jenkins, we simply need to apply the same three models outlined above one more time to build Skytap Cloud environments.

The artifact production model for Skytap Cloud environments looks like this:

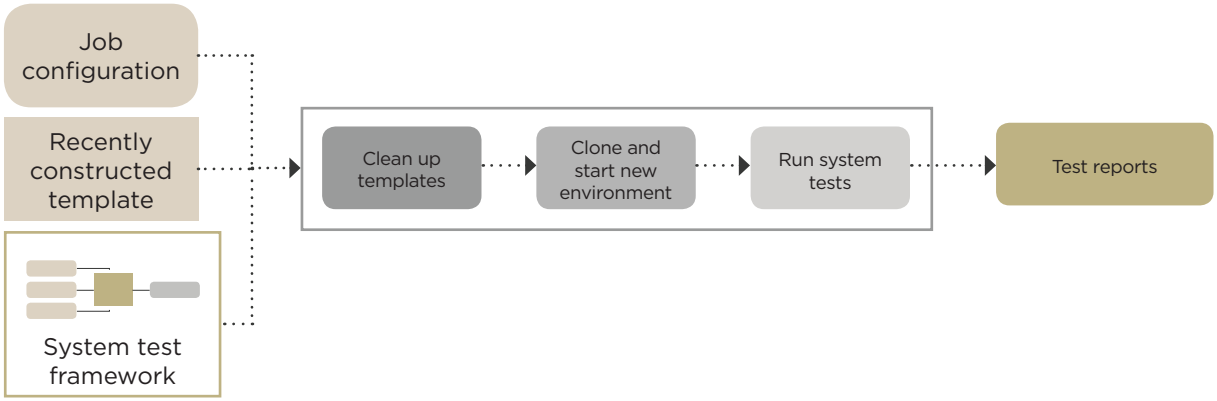


And the CI model:



At this point, we've produced a full environment as an artifact, which we can save as a golden Skytap Cloud template. That brings us to the end of the journey, right?

But wait, there's more! How about we feed those templates back into a job (as an upstream artifact), deploy an environment from the template, and perform some validation on that new environment? While we're here, let's clean up any older templates to save some resources. Adding that to a CD model is almost trivial (although fiddly bits of full system testing are another story, of course).



HOW CONTINUOUSLY-DELIVERED ENVIRONMENTS WORK FOR US

Applying CI and CD at each phase of the build and delivery pipeline allows us to apply similar abstractions to each phase of the process. This makes it easier to reason about the system, modularize it, and to build more complex artifacts atop simpler, automatically-constructed components.

CI and CD obviously aren't unique to Skytap, and neither are custom automated environment provisioning tools like Jenga. They're great tools of course, but the really spectacular thing here is being able to take working Skytap Cloud environments in their entirety and treat them like any other software artifact. Skytap Cloud templates provide us this power.

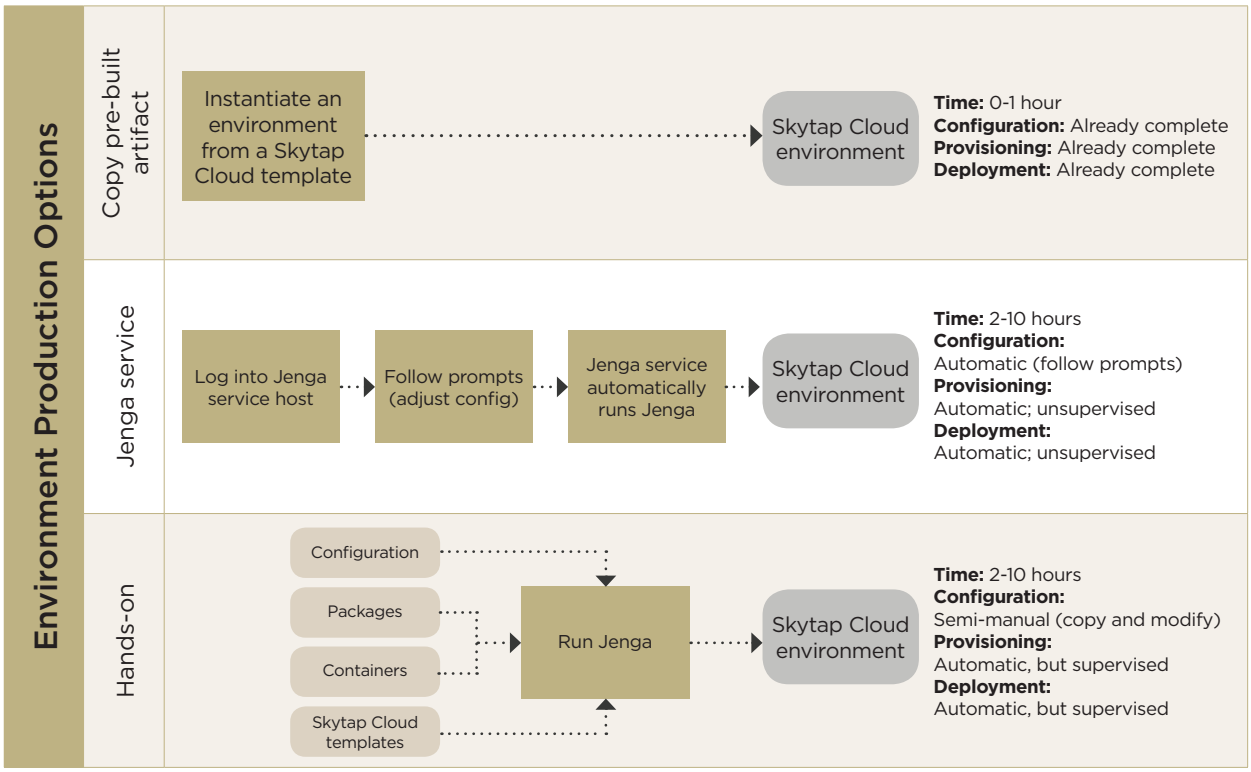
We really can't understate the benefit of deploying prepared environments from templates instead of building environments from scratch. Automatic provisioning is time-consuming, and when things go wrong, you need a fair bit of knowledge about how Puppet and the Skytap Cloud infrastructure work to resolve problems. With ready-to-go templates, we're often able to save engineers entire workdays that might otherwise be spent waiting on Jenga.

We use our CD process to deliver several "off the shelf" templates, each of which can instantiate environments for several common needs:

- Environments with the minimum set of infrastructure needed to test services
- Small environments that reproduce all of the components in production, such as software-defined networking and VM hosting infrastructure
- Larger "almost production" environments with all of the infrastructure and redundancies required to emulate production behavior.

This is a huge timesaver for individual engineers. If you want one of the standard "off the shelf" variants of the environment, you don't even need to build it yourself; you can just start an environment from the template (at the specific version you want). That's it!

"Off the shelf" templates won't serve every purpose, which is why we continue to vend Jenga as a Docker image and continue to maintain a Jenga service. Here's a breakdown of the options available when engineers need a pre-production environment:



For most needs, engineers are interested in testing their service on a recent pre-production environment; they don't need to customize the infrastructure or change much related to which services are deployed. In this case, the first option (copying a pre-built artifact) is the most appropriate.

An intermediate may need to customize the environment. For instance, the engineer might need to add or remove hosts, networks, or services. In this case, it's usually appropriate to run the actual provisioning process to build a custom environment. The Jenga service provides a command line interface, which simplifies copying an existing configuration and starting the Jenga process. The engineer can modify this configuration as needed before he or she starts provisioning. The service handles the details of running Jenga and notifies the engineer when it's complete.

Finally, some custom needs require that you run Jenga directly. This is common if we're developing new features in Jenga or creating a new type of environment template.

FINAL THOUGHTS

DevOps is about ownership of a product throughout the SDLC; our tools and the consistent application of similar practices across engineering teams make this broad ownership possible.

Automation is essential to this practice, and having the ability to provision and configure an environment on-demand is a great milestone in establishing DevOps practices in your organization. However, even in the best cases, the complexity and time commitment of ad-hoc provisioning can be burdensome. Building a new environment from scratch on each change, or each time you need an instance of the environment, involves a lot of wasted effort. It's like installing software—sometimes you need to build it from scratch, but most of the time, you just need to install a package that does something useful.

By delivering templates as standalone artifacts, we've been able to mitigate much of the pain inherent in provisioning. Engineers don't have to wait for environments or manage environment builds; this empowers them to focus on whichever aspect of the SDLC is most meaningful to them, without building silos around the components that other engineering specialties will find more meaningful.

We're proud of the platform that we've built, and we're proud of the DevOps culture we're empowering with it. But most of all, we're excited to see the awesome things our customers build with Skytap Cloud. Go forth, and embrace the power of DevOps today!

THREE ISSUES TO ADDRESS AT THE START OF YOUR DEVOPS INITIATIVES

By Kelly Looney

KELLY LOONEY

DevOps is a way for an entire software-producing organization to work; DevOps is not, as many see it, only automating the final few steps of delivery. If you don't deliver often, then automating delivery really has limited impact. More and more, we understand that software is like a living breathing thing; it requires constant attention and care to stay relevant. The traditional means of creating software via ephemeral "projects" then having a completely separate organization charged with tending (maintaining and operating) the software is becoming more unworkable every day.

DevOps is about creating and sustaining teams that take software all the way from ideas, to customer use, to end of life. These teams absolutely change in size and makeup over time, but it's important that it feels like one collective team is fully responsible and empowered to deliver a successful offering. When we look at companies trying to do DevOps (and there really is no alternative, because the results have been so compelling for so many), we see three areas that need to be considered:

1. Organizational Issues: Are cross-functional teams possible? Can product teams be empowered to deliver with no relaxation of requirements?

1. Cultural Issues: Can we foster teams that are willing to tackle whatever challenges are presented? Can we have deep expertise and still have people that work to the overarching goals of the business?

1. Technology Issues: Can we reduce delivery 'friction" to almost zero? Can we test and ask questions safely in an on-demand way? (This is where Skytap comes in...)

In our experience, success with DevOps requires attention to all three of these areas.

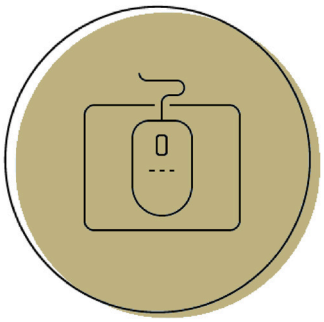
Having a Jenkins server somewhere does not mean you practice continuous integration, and using a Puppet or Chef script somewhere does not mean you are a "DevOps shop". If you have those things in place, great, but keep going, and let's see where real DevOps can truly take the successes of both you and your customers.

ABOUT SKYTAP AND CLOUD-ENABLED DEVOPS

We want to thank you for taking the time to learn more about why we feel that DevOps provides organizations with the ability to release more code with higher quality than ever before. We now invite you to learn how Skytap Cloud enables organizations to remove environment availability and management bottlenecks so teams can achieve even better results.

Self-Service, On-Demand Environments

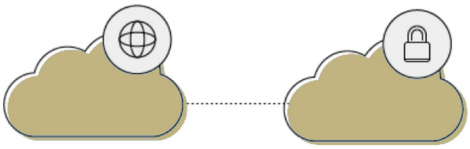
Many organizations attempt to build one-click, self-service environments by buying multiple tools and using vendor-specific scripting. Skytap Cloud delivers self-service environments out of the box along with additional features to enhance DevOps best practices, including:



- Simple, intuitive user interface
- Powerful REST API and pre-built integration points with existing DevOps tooling
- Full-stack or partial-stack deployments
- User quotas to prevent environment sprawl
- Template-based approach and environment cloning ensure zero configuration drift

Skytap Hybrid Cloud

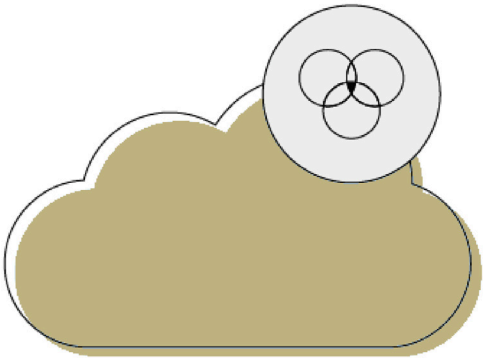
Enterprise environments are complex and consist of a mix of technologies. Skytap Cloud allows for the migration of traditional applications—x86/VMware and POWER/AIX components—while non-virtualizable systems remain on- premises. Skytap Cloud offers software-defined networking technologies, including:



- IPsec site-to-site VPN connectivity to ensure secure communications
- Direct connect availability for extra security when needed
- VPN NAT, allowing for multiple copies of a Skytap Cloud environment to be connected to on-premises systems at the same time

Integration with Existing Toolchains

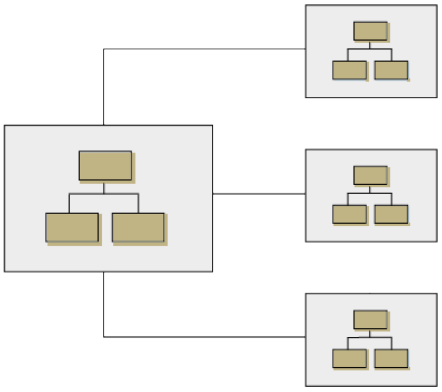
Skytap Cloud complements DevOps toolchains by making them faster and more production-ready. By adding Skytap Cloud environments to their existing toolchain, customers:



- Increase agile velocity by reducing environment build-outs to a few minutes with Skytap's template-based approach
- Continue to use infrastructure-as-code tooling to build recipe caches
- Build faster and more often with integrations to Microsoft TFS, Jenkins, IBM Rational Team Concert, and IBM UrbanCode Build
- Provision complex, production-ready cloud environments with simple Vagrant commands
- Practice deployments into production environment replicas down to the IP and MAC addresses with Skytap Cloud integrations with CA Release Automation and IBM UrbanCode Deploy

Full-Stack Environment Replication

With Skytap Cloud, developers and testers gain end-to-end, production-ready environments that enable them to:



- Collaborate on defect triage by easily sharing environments using Skytap Projects and Sharing Portals
- Develop and test against actual production configuration sets with Skytap Cloud environment cloning
- Eliminate configuration drift since each environment is copied all the way down to the IP and MAC addresses

Skytap

Skytap Headquarters
719 2nd Ave., Suite 800
Seattle, WA 98104
1-888-759-8278

Skytap UK
30 Stamford Street
London SE1 9LQ
+44 (0) 203-790-0962

Skytap Canada
240 Richmond Street W
Toronto ON M5V 2C5
+1 (426) 562-0201