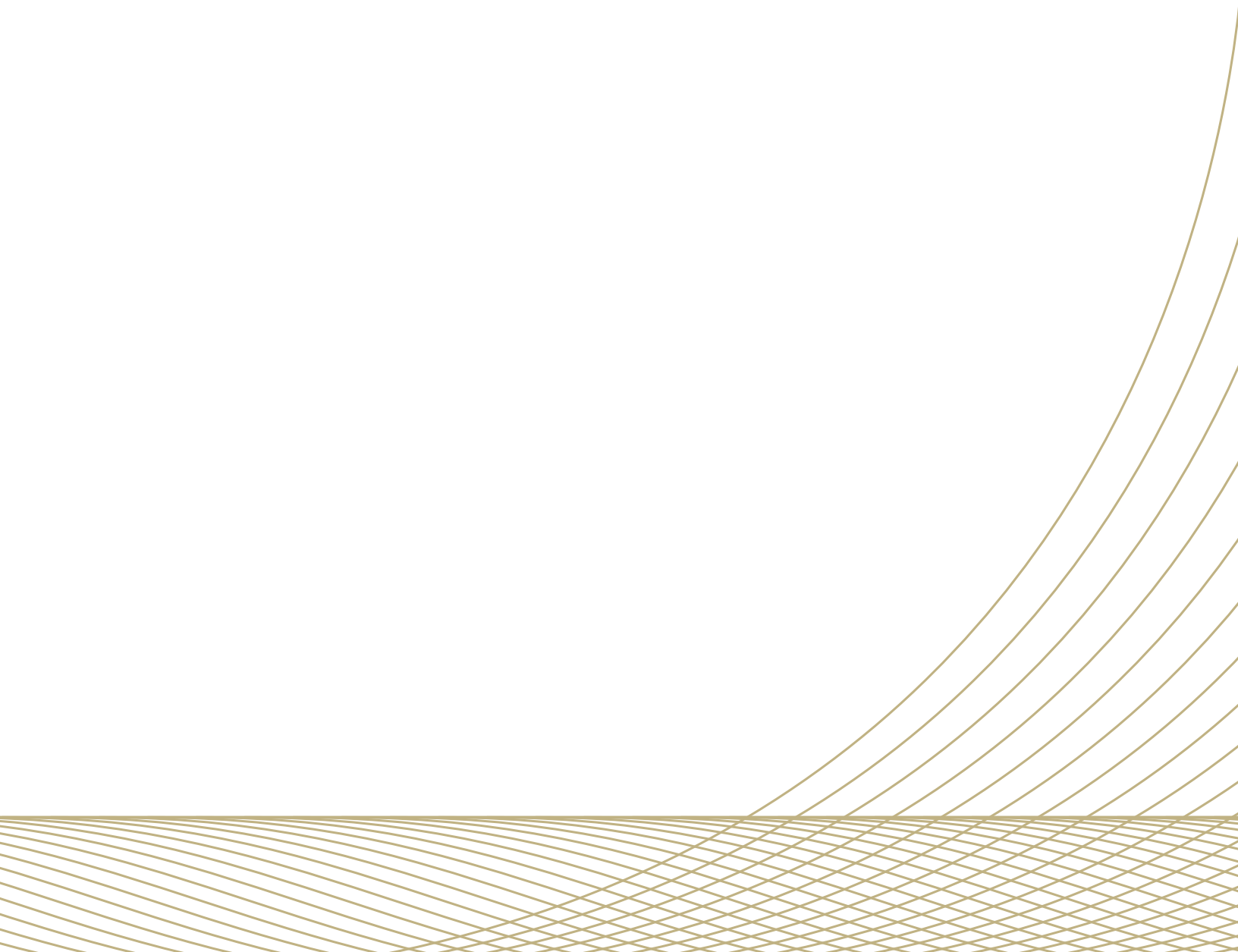
Skytap

Continuous Delivery of Fully Functional Environments

How Skytap Uses Skytap Cloud to Achieve DevOps



At Skytap, we face the same challenges as our customers in developing and releasing high-quality software quickly.

Often, the most painful stages of the software development lifecycle are the integration phase (when independently developed components run together for the first time) and the release phase (when new features face production workloads for the first time). Like other organizations, we've evolved a variety of DevOps techniques for mitigating these problems.

We currently use continuous integration (CI) and continuous delivery (CD) techniques, along with several of the unique features of Skytap Cloud, to create entire environments that resemble the production platform. Copies of these ready-to-use environments are started on-demand and used by engineers to test code in a clone of production, and in a variety of (possibly destructive) test scenarios. This frees developers to more thoroughly test their code, which reduces the QA burden and speeds up the release process.

This paper outlines the continuous delivery workflow that we use to produce these environments, and discusses how this workflow has improved our release velocity and reliability. It also takes a deeper dive into how CI and CD are implemented with Skytap Cloud.

Our Problem Scope

The challenges that we face in scaling our software and organization aren't unique. As with many organizations, as Skytap grows, the breadth and complexity of our platform necessarily increases. Additionally, as we add more engineers, we are able to develop features faster than they can be tested and released under a strictly linear, gated process. This introduces a bottleneck, which may leave developers in a wait-state during the test and release phases. It's counter-productive to have chunks of an engineering organization idling while they wait to ship their work, and it's expensive to coordinate complex scenarios where more and more features are added to the release.

Our Solution

To remove bottlenecks and improve the flow of code, we reviewed our integration and delivery processes. We decided that ideally, developers should strive to check in code frequently, and that code should be built immediately (continuous integration). The result of that build—the artifact—should pass some test criteria and should be automatically packaged as a discrete piece of software. This packaged artifact then becomes a deployment candidate (continuous delivery).

We decided that to truly leverage the benefits of CI and CD, we needed to provide developers and testers with easy, safe access to their own clone of a production environment. We expected that this would enable engineers to:

Improve code quality

- With decreased false positive/negative results in comparative testing; Skytap Cloud environments, while virtualized, very closely mimic the behavior of physical environments,

and automated configuration management decreases drift between production and pre-prod environments

- Access and visibility into all affected portions of the stack allows engineers to better understand the system-wide impact of their changes
- Engineers are able to run more extensive automated tests, test against other features, run simulated production workloads, and perform dangerous or destructive experiments, among other things
- We check-in code to main project branches more frequently; smaller incremental changes have fewer compounding issues
- When problems do occur, they are visible earlier in the process, and are smaller in scope

Enhance cross-functional collaboration

- Test burden is shifted, in part, to the developers, who are likely to be most familiar with causes and solutions to problems that appear with their code; this frees up QA teams to spend more time developing robust test scenarios and to advise development on effective test techniques
- Frequent delivery of discrete, fully functional environments with continuously integrated changes allows teams to use each other's work quickly, instead of waiting for delivery to a shared integration environment
- The self-service nature of pre-packaged environments allows developers and operations to focus on the most important interactions between the platform and infrastructure. This helps to identify potential issues well before code reaches the production environment, and reduces the operational load inherent in maintaining multiple pre-prod environments

Increase release speed

- By front-loading the effort of addressing bugs, they're cheaper and faster to fix. This reduces the QA time spent on final integration testing and pre-release verification
- Releases are less risky, as changes have already been tested in the context of the platform at large. The development of smaller viable changes is simpler; a small change can be tested in a clone of production by smaller teams, with simpler cross-team collaboration
- Conceptually, this sounds great. To achieve this, we needed to combine a set of tools and a process that could scale with us, and this process should be largely automatic and easy to replicate

Our Tool Set

Like most software companies, we heavily leverage distributed source control (Mercurial and Git in our case) and configurable build servers (Jenkins). We manage our build jobs with configuration and

a job construction tool (Jenkins Job Builder). We make use of configuration management tools (like Ansible and Puppet), and we modularize our platform services with containerization tools (Docker, Kubernetes).

We already had many of the pieces in place to begin delivering full environments to engineers. To pull everything together, we needed to integrate these tools with our internally developed automated environment construction tool (**Jenga**) and add the real secret sauce: Skytap Cloud templates.

With these tools, and the CI/CD techniques we'll explore in-depth later, we are now able to produce several nightly caches of our full stack — including the supporting infrastructure for each — and save these as Skytap Cloud templates.

Each template contains a production-ready environment with (currently) anywhere from around 40 VM's and three networks, up to around 200 VM's with six networks, and each captures a fully functional snapshot of the Skytap Cloud (which, conveniently, also runs *on* Skytap Cloud – we're testing all the way down!) We've abstracted one step further away from continuous delivery of software artifacts; we are instead delivering *entire environments* running the full platform.

What Does All of That Get Us?

If you're a Skytap engineer, you simply need to copy one of those Jenga-constructed Skytap Cloud templates and run it. This provides an advantage over using provisioning tools to produce environments on the fly because provisioning environments is a slow, complex process, and a lot can go wrong. Engineers should have fast, easy access to production clones, and should be able to treat them as disposable when problems inevitably appear—otherwise, you're losing one of the primary benefits of virtualization.

In just a few minutes, our engineers have an environment running a fully functional instance of Skytap Cloud. This is an incredible productivity boost for many common activities:

- Comparing two releases to understand regressions
- Producing a development environment from a clone of production
- Testing release scenarios
- Destructive testing and experimentation: if your cost to produce a new environment is nearly zero, you can break whatever you want!
- Integration testing
- Platform exploration—On-boarding new engineers is simplified because they can be trained in disposable environments

We'll dive deeper into how our build system creates these nightly templates later.

The Results

Ultimately, we've been able to increase our release cadence and reliability, without sacrificing the ability of discrete teams to work independently of each other. Continuous Integration at the team level allows changes to be immediately merged into mainline development, and this, in turn, surfaces problems while the change is an active work item. Raising problems early in development simplifies resolution because the context surrounding the issue is still fresh in everyone's mind.

Decoupling code check-ins and integration from the release process has given us ancillary benefits in code reviews. Our reviews are focused on professional growth and code quality, rather than being a gate that blocks our check-in. It's always unfortunate when you make a mistake and break a build but if your builds are cheap and frequent with fast and visible feedback, developers can safely treat check-in and integration as a separate activity from review. For us, this has been a boon to our culture of collaboration: code reviews are about feedback and growth, rather than being release-oriented transactions or gates.

Automating environment construction and making access to environments a cheap self-service task has allowed us to significantly reduce the load on our operations team. Without these tools, operations might be required to service requests to create and maintain development and test environments. Additionally, it's much simpler for our dev and ops teams to collaborate.

With functional environments that can break without impacting operational integrity, operations can advise development without worry that this advice will be misapplied to the production environment, and without the burden of resolving problems in shared dev/test environments when something goes wrong (again, we can just throw away the environment and start fresh). Constant operational support of non-prod environments doesn't scale well and can lead to hostility between development and operations. DevOps is about the opposite!

Continuously delivering changes from each individual line of development into freshly constructed environments each day has helped us surface deployment and service integration issues more quickly. Your project's CI process may complete successfully and the code may pass review, but you're still likely to discover problems that only occur when you plug your service into the platform. By continuously exercising these changes together, we're now able to see both the adverse effects and the added value of current work very quickly.

Additionally, disconnecting the *running* environment from the process used to *produce* working environments has allowed us to easily clone environment state. These clones simplify A/B comparison (E.G., "did this problem exist before, or is it new?"). Guaranteeing the same state in various clones also makes it simple to do destructive testing without impacting other teams. If you've ever had dev and test stalled because a shared integration environment was down, you'll understand how much we like being able to let individuals and teams create and destroy environments at will. Plus, we're able to run automated system or acceptance testing in single-purpose environments, without tests being ruined by activity in shared environments!

Finally, by leveraging Skytap Cloud templates to continuously deliver fully functional snapshots of the current and upcoming platform, we've significantly reduced the amount of time it takes to get a functional clone of most environments. Even with powerful tools like Jenga, producing an environment from scratch often requires a lot of expertise about the tool chain (dealing with Puppet errors, for example), and a lot of knowledge about the infrastructure.

While slogging through these problems can be an instructive exercise, it's time consuming and can place a heavy support load on our infrastructure and provisioning experts. Delivering functional templates has reduced the time it takes to get a working dev stack for an individual from days or weeks, down to about an hour. Our engineers are free to spend time developing features, not wrangling their bespoke environments – and since they're using automatically constructed clones, their environmental assumptions are far more likely to match the reality of integration and production environments when it comes time to release their changes.

Building and maintaining all of this CI and CD infrastructure takes time and effort, of course, but it pays strong dividends in the form of increased individual productivity, increased ability for teams to work in parallel without creating integration headaches, and increased confidence and predictability of our releases. One of the great things about being a cloud provider is that we frequently face the same challenges as our customers. It's a constant pleasure to know that we're able to use the very tools we provide as key components in our solutions to these problems.

A Deeper Dive

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, test, 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.

The remainder of this paper provides a detailed description of this continuous integration and continuous delivery (CI/CD) pipeline, with emphasis on how our process enables us to deliver fully functional pre-production environments.

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

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

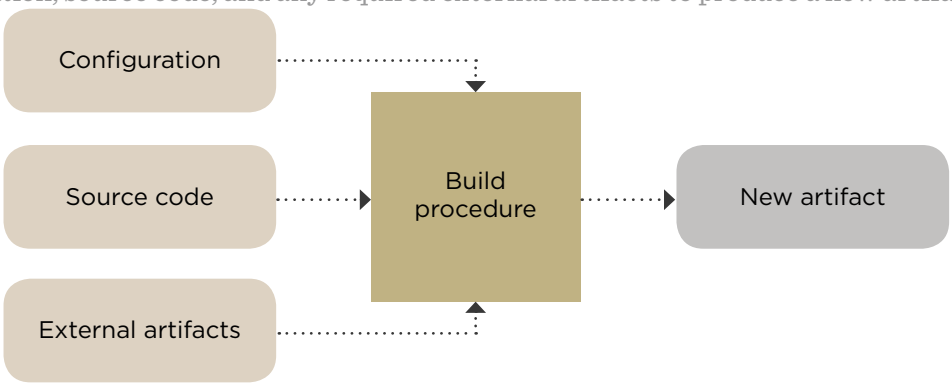
dependencies (libraries), Jenkins jobs, Docker images, and Skytap Templates.

There are three basic abstractions that we use to model the Skytap 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).

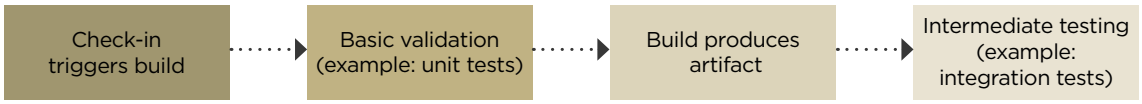
The Continuous Integration Model

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:

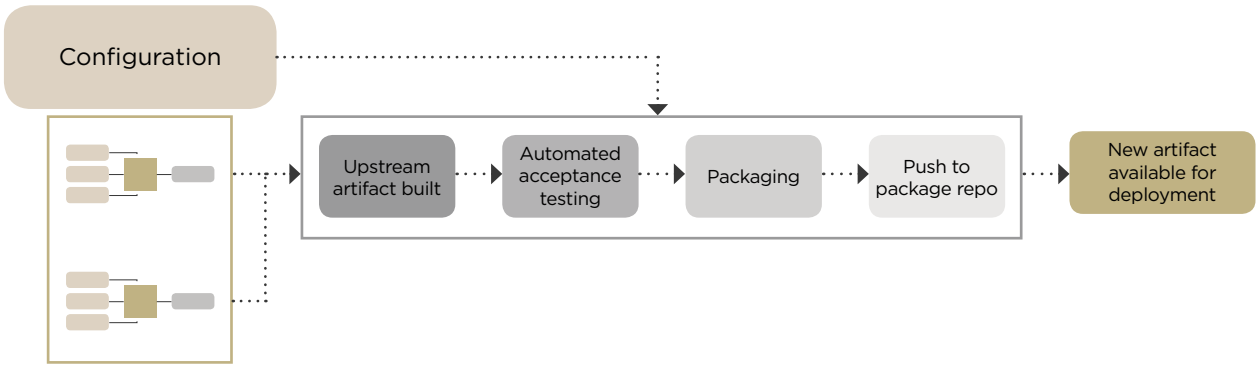
Every process using the CI model begins with a check-in (code, configuration, artifacts) and ends with either a failure in some stage or by producing a validated software artifact.



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 are 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 platform.

Our Implementation

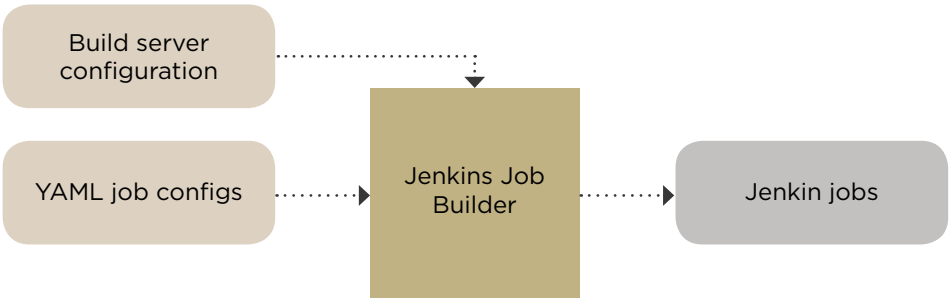
Let's take a look at how we implement these models.

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

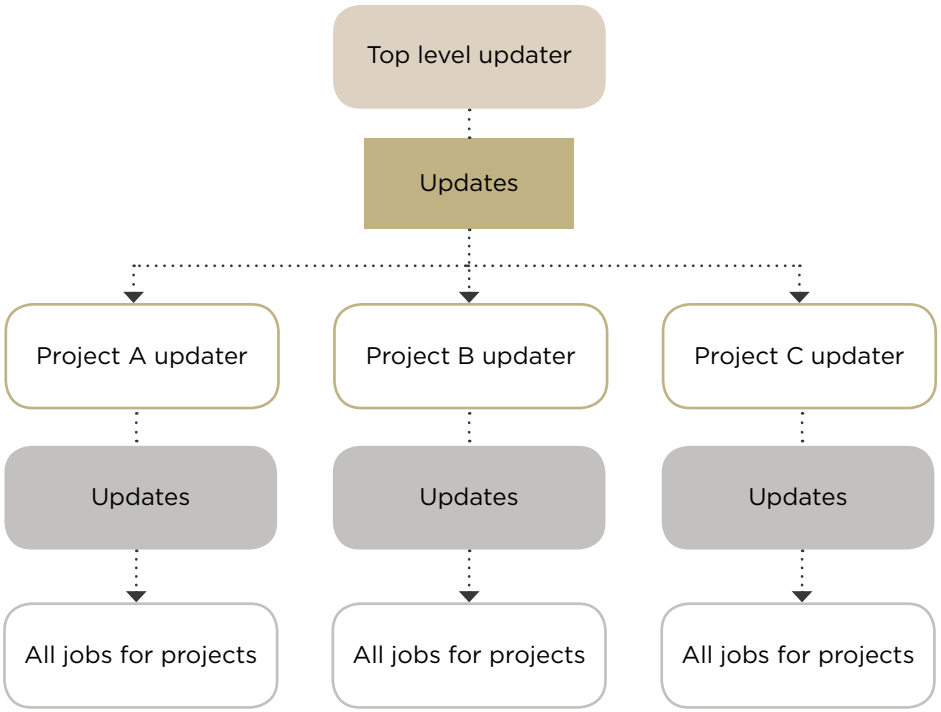
Our CI/CD workflow begins with a **Jenkins** server. The Jenkins configuration, including plugins or other server-scope items, is managed with **Puppet**. We install and configure Jenkins on a server that is launched from a Skytap Cloud template. If you're keen to apply the models outlined above, the 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 timeconsuming 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.

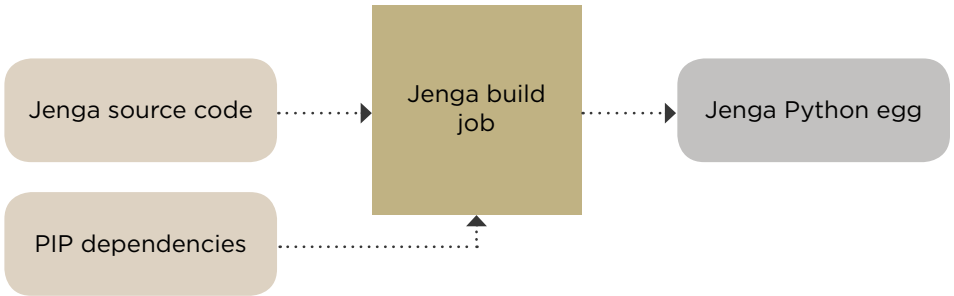
We have additional organizational constructs for maintaining build pipelines for current, previous, and upcoming releases; these all follow a similar process.

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

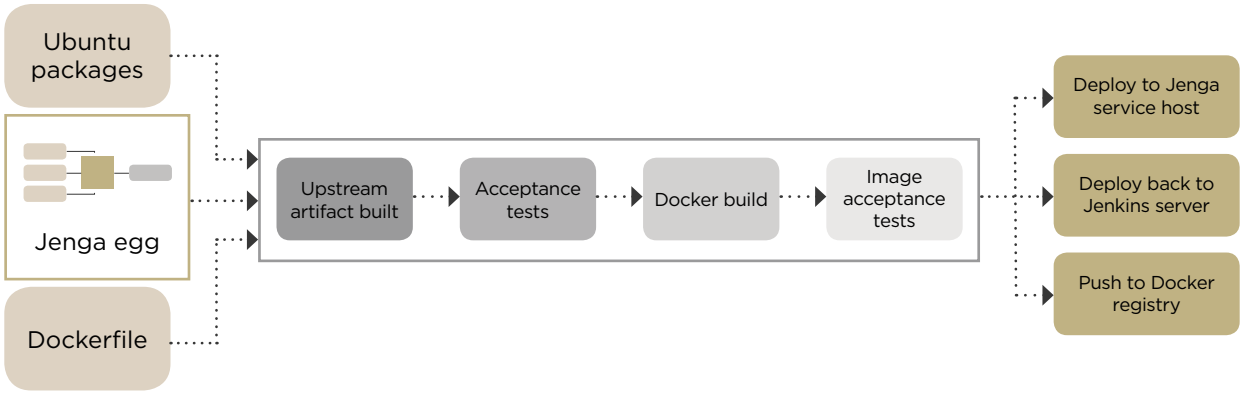
After the build server, the next component in the environment delivery pipeline is our custom-built provisioning and deployment tool, **Jenga**.

At this point, 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:

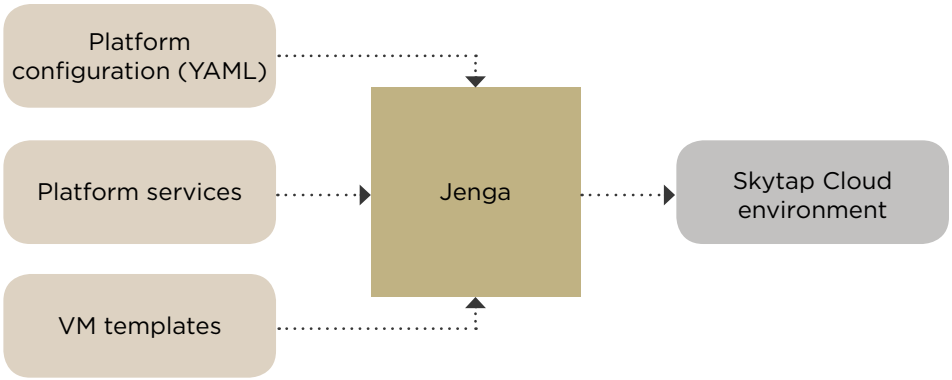
The CD model is more interesting. At this point, we package Jenga as a Docker image, and deliver it to three endpoints: a host running Jenga as a service, the Jenkins server, and our private Docker registry. Each of these endpoints uses Jenga to serve a different need; we'll discuss these three options 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.

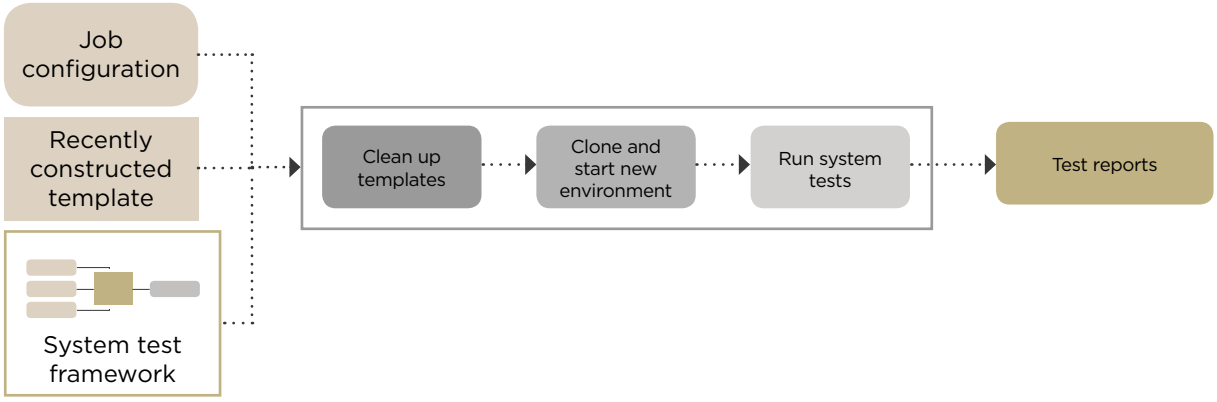
The artifact production model for Skytap Cloud environments looks like this:



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 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 Cloud, 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 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.

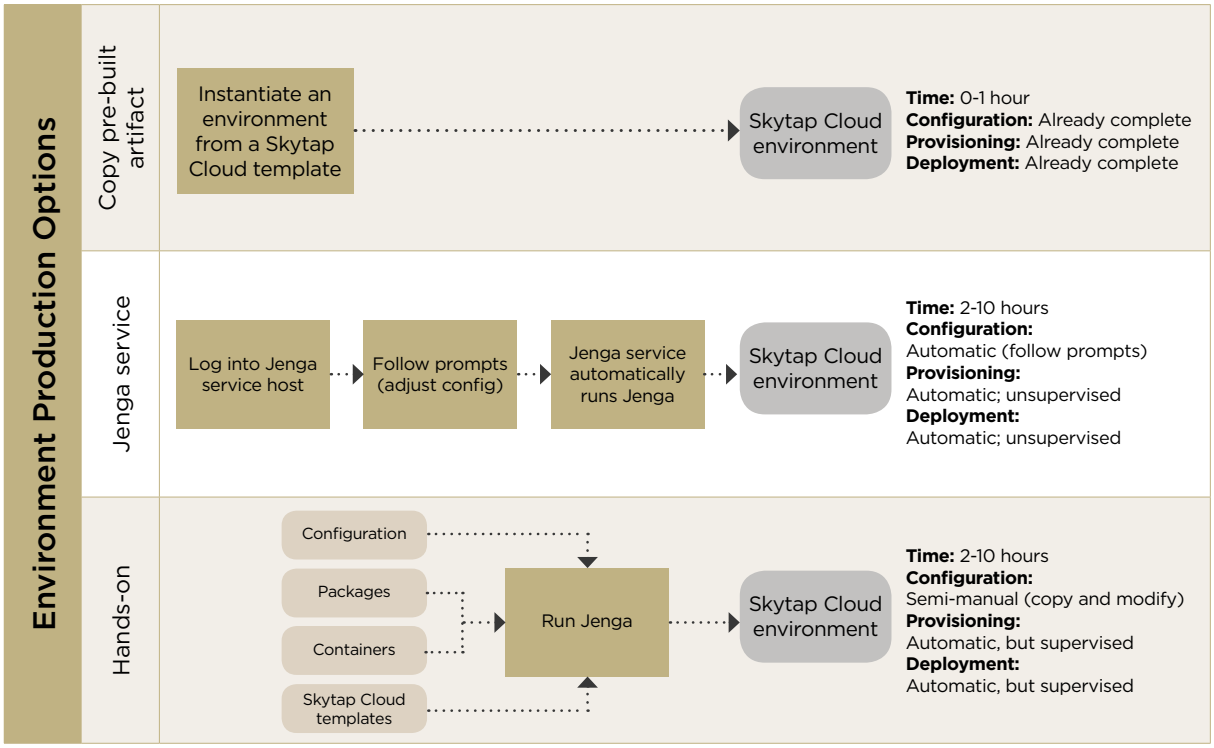
We use our CD process to deliver several "off the shelf" templates, each of which can instantiate environments for several common needs:

- Environments with the minimum set of infrastructure needed to test services
- 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 need may 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 they start 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!

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