CONTINUOUS DELIVERY AND IMMUTABLE INFRASTRUCTURE AT WHISTLE LABS

The Challenge

With Whistle's 2015 acquisition of Snaptracs, its solution topology grew both in size and complexity. What began as a single hardware device and a Ruby on Rails application hosted in Heroku had become several devices and applications distributed across three unique infrastructures.

The Whistle team wanted an updated strategy for maintaining their development agility and the reliability of their services in this constantly evolving environment.

Unif.io wanted to extend the workflow that the team was already comfortable and successful with to encompass the entire environment.

The Solution

Through strong collaboration between Unif.io and Whistle, a strategy of immutable, codified infrastructure with full deployment automation has been achieved. Unif.io wanted to extend the workflow that the team was already comfortable and successful with to encompass the entire environment. That meant keeping the focus application–centric, such as the Platform–as–a–Service (PaaS) use case, while adding support for different target infrastructures. The answer was to treat the infrastructure like an application.

Technology and services utilized to achieve the infrastructure included:

- HashiCorp Atlas, Vagrant, Packer, Terraform and Consul
- AWS Virtual Private Cloud (VPC), Identity & Access Management (IAM), Elastic Compute Cloud (EC2) and AutoScaling
- Atlassian Bamboo Cloud
- Ruby / Rake
- Docker
- Puppet

The Deployment Pipeline

First, we chose a technology for modeling the deployment pipelines of the various services.

The deployment pipeline is central to the methodology of Continuous Delivery (CD). Establishing the deployment pipeline early in a project, regardless of the level of automation, is valuable for creating feedback loops and revealing the phases needing improvement.

Pipelining takes the comprehensive steps required to move from committed code to production deployment and breaks them into phases. Phases need to be structured with an emphasis on speed for earlier stages with time consuming steps being carried out in later stages. The objective is to provide feedback as quickly as possible and ensure that all changes pass the comprehensive gates for deployment as frequently as possible, maintaining confidence that the code is releasable.

A good CD tool makes the pipeline easy to visualize while promoting communication and collaboration. Whistle was already using Atlassian Bamboo Cloud as well as the integrated build capabilities of Heroku. This was working well for them for continuous integration (CI) and because both are Software-as-a-Service (SaaS) offerings, they introduced little additional overhead. At Unif.io, we believe the right solution for our clients is the one that works best for them. For that reason, Whistle elected to continue with its investment in the Bamboo Cloud service.

Leveraging prior experience with the Bamboo product, we were able to quickly highlight the advantages and trade-offs of this approach.

Positives of Bamboo Cloud:

+ **Fully managed SaaS solution.** There would be no need to maintain software or infrastructure.
+ **Dedicated facility for deployment jobs.** More than just a build tool in that it includes special projects and jobs designed with CD pipelines in mind.
+ **Easy service integration.** Bamboo jobs were easy to integrate with other services in use by the Whistle team such as New Relic and Slack.
+ **Ability to run build slaves within a Whistle VPC.** Infrastructure jobs required very broad access to the AWS API as well as sensitive data stored within the environment. This hybrid deployment model allowed build slaves to be run in Whistle VPC networks and made the use of AWS IAM roles for credential management possible.

Negatives of Bamboo Cloud:

- **No configuration API.** An inability to fully codify the configuration of the build service. This is a concern at scale as well as a deviation from the ideals for the overall project.
- **No ability to load or dump configuration.** An inability to bulk export or import configuration. A concern in that a significant amount of business logic would only exist at this layer. A concern with regards to potential recovery time due again to the lack of a configuration API.
- **Minimal support for third party plug-ins.** Many features, extensions and community knowledge for the on-premises version were not applicable to the Cloud edition.

The inability to configure Bamboo Cloud programmatically was not ideal, but we were able to work around it by codifying the build logic as a data-driven Rake build. By leveraging environment variables exposed by Bamboo, it was possible to maintain the flexibility of managing data in the Bamboo service while vastly simplifying the steps for creating or copying jobs. This approach also kept the build logic portable, making it less of a burden to switch build solutions later if desired as well as ensuring that the logic used would be easy to understand and leverage from a developer's workspace.

Immutable Infrastructure

At the time of engagement, Whistle's solution was distributed across several environments:

- Primary application stacks were being managed on Heroku
- Data processing and warehousing were on native AWS
- Entire Snaptracs solution was running on physical hardware at Rackspace

Whistle wanted to continue to consolidate services to native AWS for capability, flexibility and economy. While they already had a presence in native AWS, the team had only moderate experience with those services. This was due to the additional domain knowledge required to manage infrastructure as well as a separate and infrequently exercised development workflow from the one used with Heroku.

Prior investments were made in configuration management tooling yielding sound processes for managing capacity and ensuring that configurations were repeatable. The weakness of the paradigm however, was that it was based on managing change in the running environment. While not an uncommon approach, it led to infrequent use of the automation which added to the stress of updating those components. This result continued to isolate that part of the environment and reinforce the notion of siloed ownership.

Unif.io facilitated further consolidation of infrastructure and a move to a paradigm of immutability. The main difference was that instead of trying to manage change directly in the solution environment, dependency resolution along with change would be shifted up the pipeline to the build phase and locked in place as versioned artifacts. This would guarantee parity across environments and significantly reduce the complexity to rolling out (as well as rolling back) changes at the edge. These infrastructure artifacts could also follow a workflow similar to that of managing applications in a PaaS environment.

Successful implementation of an immutable infrastructure strategy requires the deployment pipeline and automation to be dialed in. Speed is critical for generating artifacts for all changes becomes a bottleneck as opposed to an improvement. As such, the HashiCorp suite of tools, integrated by the Atlas service, was the best-in-class open source tooling for management of the immutable datacenter and had strong alignment with our values. As Whistle was already using Vagrant and familiar with Terraform, this was a clear path forward.

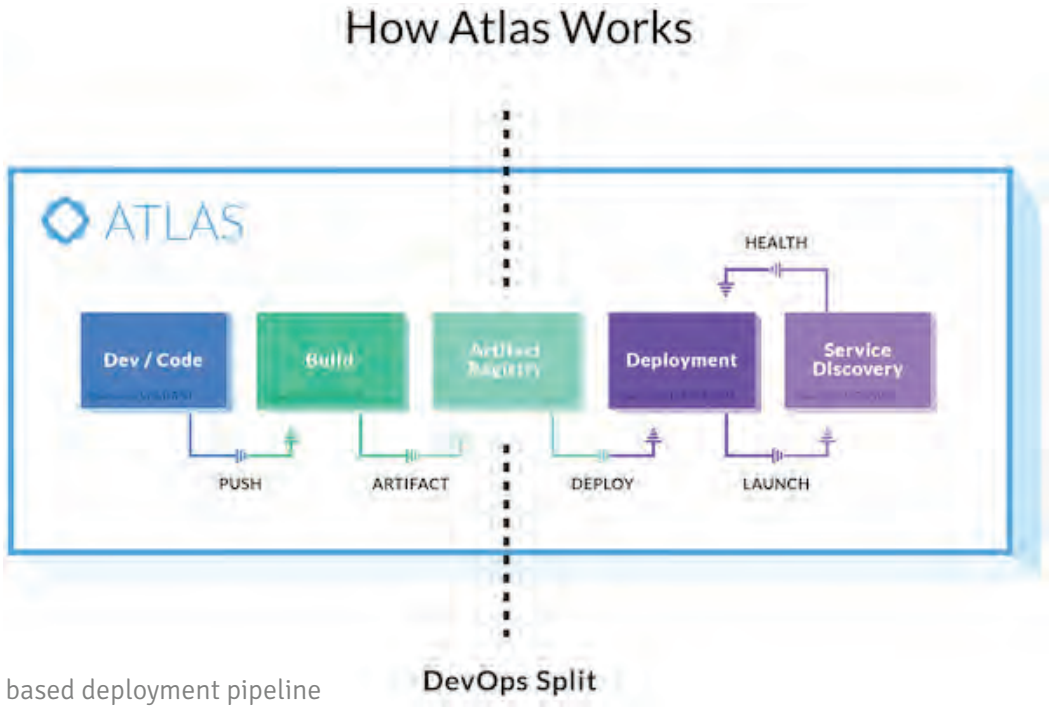

Fig 1. HashiCorp based deployment pipeline

Build / Artifacts

The HashiCorp Packer tool was utilized for creation of immutable artifacts. It was an ideal choice as it allowed for leveraging existing configuration management investments; it provided builders for both AWS Elastic Block Storage (EBS) backed instances as well as Docker containers; and it allowed for creation, management and coordination of generated artifacts using Atlas.

Using Packer, builds were created for the following set of artifacts:

- **Base image** [amazon-ebs] A base Operating System (OS) image for our AWS instances; used to seed all downstream EBS Packer builds. The primary objective was the versioning of OS and package updates. The build was also used to generate a Vagrant box to provide developers with a clean environment identical to that used in building all production nodes. Levering Vagrant metadata in Atlas, we could ensure that developers were always utilizing the latest image generated by the pipeline.
- **Third party application stacks** [amazon-ebs] Application stacks built on the base image. Targeted backing services and leveraging existing Configuration Management (CM) code.
- **Docker hosts** [amazon-ebs] Docker engine as well as other tooling and configuration in support of container based process management.
- **Whistle application stacks** [docker] Containerized application stack environments. The primary objective being to provide developers with flexibility and control at the OS layer not offered in the managed PaaS environment.

As with the base image, Vagrant box artifact creation was added to all Packer builds. While the base image allowed for development and test of the logic to be used by the Packer builders, the pre-built images could be used for verification of behavior on initial boot as well as deployment logic to be discussed in the following section.

Deployment

Unif.io believes in keeping things simple where possible. This view makes it tempting to try to avoid the addition of new technology stacks when one already in use could potentially address a need.

Most tool vendors at this point have realized that a "cloud" strategy is key to their future success. This has led to the cloud provisioning and management space becoming fairly crowded and overlapping. The Whistle team had been leveraging some of the provisioning capabilities of their CM tools already, however, we found that the capabilities of those tools were somewhat narrow in scope and none felt like a natural fit for the immutable use case.

When it was clear a more targeted solution was needed, the next avenue considered was AWS CloudFormation. Unif.io had used CloudFormation in previous solutions and found it to be extremely capable, as Amazon does well in keeping it up-to-date with their continuously expanding catalog of services. While very effective in the creation and initial orchestration of resources, the Unif.io team found it inadequate as a solution for the management of existing resources.

There were two primary deficiencies behind that assessment of the CloudFormation tool:

1. **The tool was only aware of deltas between the data passed in and the data from the previous run; not the actual state of the resources provisioned.** Unlike a traditional CM package that uses a declarative syntax to define a desired state that is then enforced, CloudFormation would not reconcile the state of existing resources with the desired state when calculating an execution plan and could not effectively be used for detection of configuration drift or true enforcement of state.
2. **The tool did not provide a planning or no-operation capability.** This could be very disconcerting when updating production services and any resources containing application data, as many types of updates can be destructive and force new resources. While risks associated

with this can certainly be mitigated with good deployment architecture and pipeline design, mistakes do occur and can cause significant harm.

Terraform was a fairly young project at the time, but promised to address many of the gaps with other tools in the space and was open source and pluggable by design. Additionally, Terraform utilized a Domain Specific Language (DSL) that was far more expressive than JavaScript Object Notation (JSON), allowing for more readable and reusable code. We collectively agreed that making an investment in Terraform would be a move in the right direction long–term.

Using Terraform, the Whistle environments were modeled in layers. This was made feasible through a Terraform mechanism of sharing state data between stacks (groups of related resources). The foundation was the codification of the Virtual Private Cloud (VPC) topology. Stacks to be deployed in those VPC environments would source required parameters (i.e. subnets, etc.) from the VPC stacks directly. This allowed for effective decoupling without sacrificing automated propagation of data between resources collectively under management.

The following are the types of stacks used in the Whistle solution:

- **VPC** – Stack for the management of all VPC specific features (i.e. default gateways, DNS, DHCP, subnets, routing, NATs, etc.)
- **VPN** – Stack for the management of VPC site–to–site Virtual Private Networks (VPN), VPC peering and Virtual Private Gateways (VPG). These resources are managed separately from the VPC stack as their lifecycle tends to differ in that they are less disposable.
- **Container cluster** – Stack for the management of instances configured for the purpose of running container based processes as well as supporting resources (i.e. IAM roles, security groups, AutoScaling, monitoring, notifications, etc.)
- **Service nodes** – Stacks for the management of various backing and supporting services, which are at present not container based.

Lines of stack separation were driven mostly by application life cycle and audience. For example, application containers are likely to be updated more frequently than the hosts they run on and far more frequently than the VPC in which the cluster is deployed. These components may be managed by different teams as well or be shared resources, so exposing too much surface area at the application stack layer often introduces more variables than are necessary by default in that context (i.e. some developers may never have a need to change the container environment or underlying host environment, etc.).

It is important to note that continuous execution of stack automation is critical to regression detection and verification that stacks are in a releasable state. This can be accomplished with pipeline automation and without the need to generate unnecessary churn in live environments.

Most tool vendors at this point have realized that a "cloud" strategy is key to their future success.

Context Management & Service Discovery

One of the areas of complexity that comes into play when implementing an immutable strategy is how to handle context.

For instance, take a solution which is comprised of a staging and production environment. The pipeline calls for changes to go through the staging environment prior to promotion to the production environment. Perhaps the application connects to a database, has monitoring instrumentation requiring a license key or is a member of a named cluster. How does one keep the environments separated when the configuration is immutable and set at build time?

The answer is separation of data from code. This same consideration exists in the PaaS environment as well.

Applications written to run on Heroku must adhere to the principles of "The Twelve–Factor App". A twelve–factor application sources configuration from environment variables, which allows for a generic application artifact that is portable between contexts. The same concept lends itself to immutable infrastructure, but there were still a few items to work out for Whistle.

- Where is the data going to be stored and how is it be accessed at provision time?
- How would non–Whistle software, that was never intended for use as a twelve factor application, be handled?
- How is sensitive data (i.e. passwords) be handled?

The HashiCorp answer to these questions was Consul*. Consul filled several roles including service discovery, health monitoring and key/value (K/V) storage. Consul would be the environment management framework and would be utilized in facilitating contextual orchestration.

To implement, the Consul agent was added to the configurations maintained by Packer to ensure inclusion in AWS artifacts. Also, a Terraform stack for a highly available (HA) Consul server tier was created and introduced into the VPC environments. Custom solutions were implemented for the management of the K/V data store (including encryption management for secure data) as well as the sourcing of data by instances at provision time.

Whistle is currently using Consul in conjunction with AWS native capabilities as well as 3rd party services for both service discovery and monitoring.

*HashiCorp has since released the Vault project. Vault specifically targets the use case of secrets management and is now positioned as their best practice for that aspect.

How does one keep the environments separated when the configuration is immutable and set at build time? The answer is separation of data from code.

Bringing It All Together

With all the components in place:

- **Vagrant** as an interface into the environment for consistent development capacity;
- **Packer** to produce immutable artifacts when changes were made;
- **Atlas** to manage artifacts and coordinate the phases of the pipeline;
- **Terraform** to provision and maintain the environments as well as deploy the immutable artifacts; and
- **Consul** to manage a dynamic landscape;

Whistle was able to establish the pipelines for their environment.

Pipelines were modeled as layers of the environment along the same lines of separation used in the Terraform stacks. These layers roughly fell into the following groups:

- **Base infrastructure** – VPC, VPNs, etc.
- **Service infrastructure** – Application clusters, etc.
- **Applications** – Container deployment, etc.

This decoupling was designed to effectively model both stack lifecycle and stack audience and to keep each pipeline as performant as possible.

The development workflow for all components, infrastructure and applications, was now consistent.

Results & Benefits

The transition to an immutable infrastructure has resulted in several assets for the Whistle team:

- The team has a single dashboard for visibility into their pipelines with push–button deployment capability.
- The process of pushing applications to VPC environments is comparable to the PaaS process.
- Developers have the flexibility to both access and configure the container environments in which applications will be deployed.
- The process of making infrastructure updates is far more consumable and is consistent with the process for application updates.
- The Whistle team has a framework in place which facilitates continuous improvement and experimentation while ensuring the highest quality and reliability of their services.

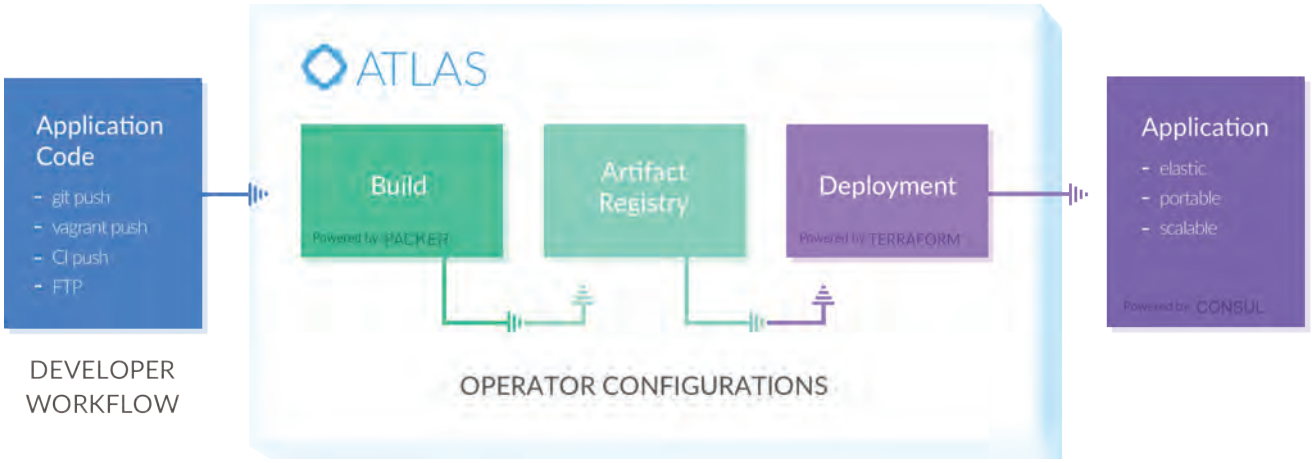

Fig 2. HashiCorp based application deployment pipeline

About Unif.io

Unif.io is a consulting services organization which uses a full-stack, holistic approach to solution design and automation. Unif.io's deep partnership with their clients guides the achievement of business goals. The How changes quickly so Unif.io values the Why.
Visit us online at unif.io, drop us a note at info@unif.io, or call us at (855) 925-0010.

About Whistle Labs

Whistle is the world-leader in pet technology, creating smart products and a mobile platform to help pets live healthier and happier lives. Whistle has raised the standard of pet care through intuitive devices, like Whistle GPS Pet Tracker, Whistle Activity Monitor and the largest comparative database of pet health information. From providing peace of mind for an escape artist pet, through organizing a pet's caretakers in one place, to giving a voice to a pet's unique needs, Whistle enhances the special bond people share with their animals.
For more information visit www.Whistle.com.

Contact us: 1-855-925-0010
General Inquiries: contact@unif.io
Partnerships: partnerships@unif.io