O'REILLY®

Continuous Delivery with Spinnaker

Fast, Safe, Repeatable Multi-Cloud Deployments

Emily Burns, Asher Feldman, Rob Fletcher, Tomas Lin, Justin Reynolds, Chris Sanden, Lars Wander & Rob Zienert

Continuous Delivery with Spinnaker

Fast, Safe, Repeatable Multi-Cloud Deployments

Emily Burns, Asher Feldman, Rob Fletcher, Tomas Lin, Justin Reynolds, Chris Sanden, Lars Wander, and Rob Zienert

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Continuous Delivery with Spinnaker

by Emily Burns, Asher Feldman, Rob Fletcher, Tomas Lin, Justin Reynolds, Chris Sanden, Lars Wander, and Rob Zienert

Copyright © 2018 Netflix, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nikki McDonald
Editor: Virginia Wilson
Production Editor: Nan Barber
Copyeditor: Charles Roumeliotis
Proofreader: Kim Cofer

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest
Technical Reviewers: Chris Devers and Jess Males

May 2018: First Edition

Revision History for the First Edition

2018-05-11: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Continuous Delivery with Spinnaker*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Netflix. See our *statement of editorial independence*.

978-1-492-03549-7

[LSI]

Table of Contents

Preface

Many, possibly even most, companies organize software development around "big bang" releases. An application has a suite of new features and improvements developed over weeks, months, or even years, laboriously tested, then released all at once. If bugs are found post-release it may be some time before users receive fixes.

This traditional software release model is rooted in the production of physical products—cars, appliances, even software sold on physical media. But software deployed to servers, or installed by users over the internet with the ability to easily upgrade does not share the constraints of a physical product. There's no need for a product recall or aftermarket upgrades to enhance performance when a new version can be deployed over the internet as frequently as necessary.

Continuous delivery is a different model for delivering software that aims to reduce the amount of *inventory*—features and fixes developed but not yet delivered to users—by drastically cutting the time between releases. It can be seen as an outgrowth of agile software development with its aim of developing software iteratively and seeking continual validation and feedback from users in order to avoid the increased risk of redundancy, flawed analysis, or features that are not fit for the purpose associated with large, infrequent software releases.

Teams using continuous delivery push features and fixes live when they are ready without batching them into formal releases. It is not unusual for continuous delivery teams to push updates live multiple times a day.

Continuous deployment goes even further than continuous delivery, automatically pushing each change live once it has passed the automated tests, canary analysis, load testing, and other checks that are used to prove that no regressions were introduced.

Continuous delivery and continuous deployment rely on the ability to define an automated and repeatable process for releasing updates. At a cadence as high as tens of releases per week it quickly becomes untenable for each version to be

manually deployed in an ad hoc manner. What teams need are tools that can reliably deploy releases, help with monitoring and management if—let's be honest, *when*—there are problems, and otherwise stay out of the way.

Spinnaker

Spinnaker was developed at Netflix to address these issues. It enables teams to automate deployments across multiple cloud accounts and regions, and even across multiple cloud platforms, into coherent "pipelines" that are run whenever a new version is released. This enables teams to design and automate a delivery process that fits their release cadence, and the business criticality of their application.

Netflix deployed its first microservice to the cloud in 2009. By 2014, most services, with the exception of billing, ran on Amazon's cloud. In January 2016 the final data center dependency was shut down and Netflix's service was 100% run on AWS.

Spinnaker grew out of the lessons learned in this migration to the cloud and the practices developed at Netflix for delivering software to the cloud frequently, rapidly, and reliably.

Who Should Read This?

This report serves as an introduction to the issues facing a team that wants to adopt a continuous delivery process for software deployed in the cloud. This is not an exhaustive Spinnaker user guide. Spinnaker is used as an example of how to codify a release process.

If you're wondering how to get started with continuous delivery or continuous deployment in the cloud, if you want to see why Netflix and other companies think continuous delivery helps manage risk in software development, if you want to understand how codifying deployments into automated pipelines helps you innovate faster, read on…

Acknowledgements

We would like to thank our colleagues in the Spinnaker community who helped us by reviewing this report throughout the writing process: Matt Duftler, Ethan Rogers, Andrew Phillips, Gard Rimestad, Erin Kidwell, Chris Berry, Daniel Reynaud, David Dorbin, and Michael Graff.

—The authors

CHAPTER 1

Why Continuous Delivery?

Continuous delivery is the practice by which software changes can be deployed to production in a fast, safe, and automatic way.

In the continuous delivery world, releasing new functionality is not a world-shattering event where everyone in the company stops working for weeks following a code freeze and waits nervously around dashboards during the fateful minutes of deployment. Instead, releasing new software to users should be routine, boring, and so easy that it can happen many times a day.

In this chapter, we'll describe the organizational and technical practices that enable continuous delivery. We hope that it convinces you of the benefits of a shorter release cycle and helps you understand the culture and practices that inform the delivery culture at Netflix and other similar organizations.

The Problem with Long Release Cycles

Dependencies drift. As undeployed code sits longer and longer, the libraries and services it depends upon move on. When it does come time to deploy those changes, unexpected issues will arise because library versions upstream have changed, or a service it talks to no longer has that compatible API.

People also move on. Once a feature has finished development, developers will naturally gravitate to the next project or set of features to work on. Information is no longer fresh in the minds of the creators, so if a problem does arise, they need to go back and investigate ideas from a month, six months, or a year ago. Also, by having large releases, it becomes much more difficult to isolate and triage the source of issues.

So how do we make this easier? We release more often.

Benefits of Continuous Delivery

Continuous delivery removes the ceremony around the software release process. There are several benefits to this approach:

Innovation
: Continuous delivery ensures quicker time to market for new features, configuration changes, experiments, and bug fixes. An aggressive release cadence ensures that broken things get fixed quickly and new ways to delight users arrive in days, not months.

Faster feedback loops
: Smaller changes deployed frequently makes it easier to troubleshoot issues. By incorporating automated testing techniques like chaos engineering or automated canary analysis into the delivery process, problems can be detected more quickly and fixed more effectively.

Increase reliability and availability
: To release quickly, continuous delivery encourages tooling to replace manual error-prone processes with automated workflows. Continuous delivery pipelines can further be crafted to incrementally roll out changes at specific times and different cloud targets. Safe deployment practices can be built into the release process and reduce the blast radius of a bad deployment.

Developer productivity and efficiency
: A more frequent release cadence helps reduce issues such as incompatible upstream dependencies. Accelerating the time between commit and deploy allows developers to diagnose and react to issues while the change is fresh in their minds. As developers become responsible for maintaining the services they deploy, there is a greater sense of ownership and less blame game when issues do arise. Continuous delivery leads to high performing, happier developers.

Useful Practices

As systems evolve and changes are pushed, bugs and incompatibilities can be introduced that affect the availability of a system. The only way to enable more frequent changes is to invest in supporting people with better tooling, practices, and culture.

Here are some useful techniques and principles we've found that accelerate the adoption of continuous delivery practices:

Encourage self-sufficiency
: Instead of delegating the deployment process to a specialized team, allow the engineers who wrote the code to be responsible for deploying and support-

ing their own releases. By providing self-serve tools and empowering engineers to push code when they feel it is ready, engineers can quickly innovate, detect, and respond.

Automate all the things
: Fully embracing automation at every step in the build, test, release, promote cycle reduces the need to babysit the deployment process.

Make it visible
: It is difficult to improve things that cannot be observed. We found that consolidating all the cloud resources across different accounts, regions, and cloud providers into one view made it much easier to track and debug any infrastructure issues. Deployment pipelines also allowed our users to easily follow how an artifact was being promoted across different steps.

Make it easy to do
: It shouldn't require expert-level knowledge to craft a cloud deployment. We found that focusing heavily on user experience so that anyone can modify and improve their own processes had a significant impact in adopting continuous delivery.

Paved road
: It is much easier to convince a team to embrace continuous delivery when you provide them with a ready-made template they can plug into. We defined a "paved road" (sometimes called a "golden road") that encapsulates best practices for teams wishing to deploy to the cloud (Figure 1-1). As more and more teams started using the tools, any improvements we made as part of the feedback loop became readily available for other teams to use. Best practices can become contagious.

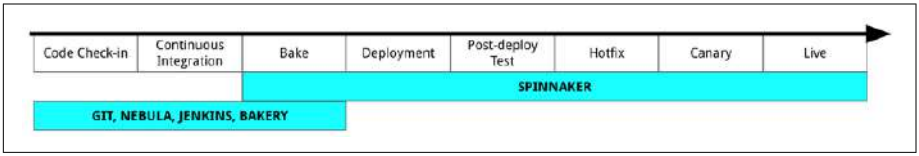

Figure 1-1. The paved road of software release at Netflix. The top row shows the steps, from code check-in to taking traffic, and the bottom rows show the tools used at Netflix for each step.

Summary

After migrating to a continuous delivery platform, we found the number of issues and outages caused by bad deployments reduced significantly. Now that we are all-in on Spinnaker, it is even easier to help push these practices further, resulting in a widespread reduction in deployment-related issues.

CHAPTER 2

Cloud Deployment Considerations

Whether starting a greenfield project or planning the migration of a complex distributed system to the cloud, choices made around how software is deployed and infrastructure architected have a material impact on an application's robustness, security, and ability to scale. Scale here refers both to the traffic handled by applications and the growing number of engineers, teams, and services in an organization.

The previous chapter covered why continuous delivery can be beneficial to organizations. It also covered some practices to keep in mind as you think about continuous delivery in your organization. In this chapter, we will discuss fundamental considerations that your organization will need to solve in order to successfully deploy software to the cloud. Each of these areas needs to have a solution in your organization before you can choose a continuous delivery strategy. For each consideration, we will demonstrate the pitfalls and present the work that has been done in the community and at Netflix as a potential solution. You'll learn what to consider before you set up a continuous delivery solution.

Credentials Management

The first thing to consider is how you will manage credentials within the cloud. As a wise meme once said, "the cloud is just someone else's computer." You should always be careful when storing sensitive data, but all the more so when using a rented slice of shared hardware.

Cloud provider identity and access management (IAM) services help, enabling the assignment of roles to compute resources, empowering them to access secured resources without statically deployed credentials, which are easily stolen and difficult to track. IAM only goes so far, though. Most likely, at least some of your services will need to talk to authenticated services operated internally or by

third-party application vendors. Database passwords, GitHub tokens, client certificates, and private keys should all be encrypted at rest and over the wire, as should sensitive customer data. Certificates should be regularly rotated and have a tested revocation method.

Google's Cloud Key Management service meets many of these needs for Google Cloud Platform (GCP) customers. Amazon's Key Management Service provides an extra layer of physical security by storing keys in hardware security modules (HSMs), but its scope is limited to the fundamentals of key storage and management. Kubernetes has a Secrets system focused on storage and distribution to containers. HashiCorp's Vault is a well regarded open source solution to secret and certificate management that is fully featured and can run in any environment.

Whether selecting or building a solution, consider how it will integrate with your software delivery process. You should deploy microservices with the minimal set of permissions required to function, and only the secrets they need.

Regional Isolation

The second thing to consider is regional isolation. Cloud providers tend to organize their infrastructure into addressable zones and regions. A zone is a physical data center; several zones in close proximity make up a region. Due to their proximity, network transit across zones within the same region should be very low latency. Regions can be continents apart and latency between them orders of magnitude greater than between neighboring zones.

The most robust applications operate in multiple regions, without shared dependencies across regions.

Simple Failure Scenario

Take an application that runs in *region-1*, *region-2*, and *region-3*. If a physical accident or software error takes *region-1* offline, the only user impact should be increased network latency for those closest to *region-1*, as their requests now route to a region further afield.

This is the ideal scenario, but is rarely as simple as duplicating services and infrastructure to multiple regions, and can be expensive. In our simple failure scenario, where the only user impact was caused by network latency, the other regions had sufficient capacity ready to handle the sudden influx of users from *region-1*. Cold caches didn't introduce additional latency or cause database brownouts, and users were mercifully spared data consistency issues, which can

occur when users are routed to a new region before the data they just saved in their original region had time to replicate.

For many organizations, that ideal isn't realistic. Accepting some availability and latency degradation for a brief time while "savior" regions autoscale services in response to a lost region can result in significant cost savings. Not all data stores are well suited for multiregion operation, with independent write masters in all regions. Many applications depend on in-memory caches to shield slower databases from load spikes, and to reduce overall latency. Let's say we have a database that typically serves 10k requests per second (RPS) of read queries behind a caching service with a 90% hit rate. How will the system behave if there is an influx of 100k RPS from users of the failed region, all resulting in cache misses and directly hitting the database? Questions like this are important to evaluate as you consider deploying more instances to help with failure scenarios.

If your company has yet to reach a scale that justifies active operation in multiple regions, deploy services to tolerate a zone failure within your chosen region. In most cases, doing so is far less complicated or costly. Due to the low latency across zones, storage systems that support synchronous replication or quorum-based operations can be evenly distributed across three or more zones within a region, transparently tolerating a zone failure without sacrificing strong consistency. Autoscalers support automatic instance balancing across zones, which works seamlessly for stateless services. Pick a consistent set of zones to use, and ensure the minimum instance count for each critical service is a multiple of the number of chosen zones. If you are using multiple cloud provider accounts for isolation purposes, keep in mind that some cloud providers randomize which physical data center a zone identifier maps to within each account.

Once your organization has extensive experience with regional redundancy, zone-level redundancy within regions becomes less important and may no longer be of concern. A region impacted by a zone failure may not be capable of serving the influx of traffic from a concurrent regional failure. Evacuating traffic from a degraded region may make follow-on issues easier to respond to.

Autoscaling

The third thing to consider is autoscaling. Autoscaling, or dynamic orchestration, is a fundamental of cloud-native computing. If the physical server behind a Kubernetes pod or AWS instance fails, the pod or instance should be replaced without intervention. By ensuring that each resource is correctly scaled for its current workload, an autoscaler is as invaluable at maintaining availability under a steady workload as it is in scaling a service up or down as workloads vary. This is far more cost-effective than constantly dedicating the resources required to handle peak traffic or potential spikes.

Smooth autoscaling requires knowledge of how each of your services behaves under load, their startup characteristics, and the resource demands they place on downstream services. For example, if you have a small MySQL cluster capable of accepting 2,000 concurrent connections and the service calling it uses a pool of 30 connections per instance, take care not to allow that service to scale beyond 66 instances. In complex distributed systems, such limits can be more difficult to ascertain.

A simple scaling policy reacts to a single system-level metric, such as average CPU utilization across instances. How should the upper and lower bounds be set? The level of CPU utilization at which service performance degrades will vary from service to service and can be workload dependent. Historical metrics can help (i.e., "When CPU utilization hit 70% last Sunday, 99th percentile latency spiked 1500 ms"), but factors other than user requests can impact CPU utilization. At Netflix, we prefer to answer this question through a form of production experimentation we call *squeeze testing*. It works by gradually increasing the percentage of requests that are routed to an individual instance as it is closely monitored.

It helps to run such tests regularly and at different times of the day. Perhaps a batch job that populates a data store periodically reduces the maximum throughput of some user-facing microservices for the duration? Globally distributed applications should also be tested independently across regions. User behavior may differ from country to country in impactful ways.

> The metric we all use for CPU utilization is deeply misleading, and getting worse every year.
>
> —Brendan Gregg, "CPU Utilization is Wrong" [1]

Scaling based on CPU utilization may not always behave as intended. Application-specific metrics can result in better performing and more consistent scaling policies, such as the number of requests in a backlog queue, the duration requests spend queued, or overall request latency. But no matter how well tuned a scaling policy is, autoscaling provides little relief for sudden load spikes (think breaking news) if parts of your application are slow to launch due to lengthy warmup periods or other complications. If a production service takes 15 minutes to start, reactive autoscaling is of little help in the case of a sudden traffic spike. At Netflix, we built our own predictive autoscaler that uses recent traffic patterns and seasonality to predict when critical but slow-to-scale-up services will need additional capacity.

1 *http://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html*

Immutable Infrastructure and Data Persistence

The fourth thing to consider is immutable infrastructure and data persistence. Public clouds made the Immutable Server pattern widely accessible for the first time, which Netflix quickly embraced. Instead of coordinating servers to install the latest application deployment or OS updates in place, new machine images are built from a base image (containing the latest OS patches and foundational elements), upon which is added the version of an application to be deployed. Deploying new code? Build a new image.

We strongly recommend the Immutable Server pattern for cloud-deployed microservices, and it comes naturally when running on a container platform. Since Docker containers can be viewed as the new package format in lieu of RPM or dpkg, they are typically immutable by default.

The question then becomes: when should this pattern be avoided? Immutability can be a challenge for persistent services such as databases. Does the system support multiple write masters or zero downtime master failovers? What is the dataset size and how quickly can it be replicated to a new instance? Network block storage enables taking online snapshots that can be attached to new instances, potentially cutting down replication time, but local NVMe storage may make more sense for latency-sensitive datastores. Some persistent services do offer a straightforward path toward the immutable replacement of instances, yet taking advantage of this could be cost-prohibitive for very large datasets.

Service Discovery

The fifth thing to consider is service discovery. Service discovery is how cloud microservices typically find each other across ever-changing topologies. There are many approaches to this problem, varying in features and complexity. When Netflix first moved into AWS, solutions to this problem were lacking, which led to the development of the Eureka service registry, open sourced in 2012. Eureka is still at the heart of the Netflix environment, closely integrated with our chosen microservice RPC and load-balancing solutions. While third-party Eureka clients exist for many languages, Eureka itself is written in Java and integrates best with services running on the JVM. Netflix is a polyglot environment where non-JVM services typically run alongside a Java sidecar that talks to Eureka and load-balances requests to other services.

The simplest service discovery solution is to use what's already at hand. Kubernetes provides everything needed for services it manages via its concept of Services and Endpoints. Amazon's Application Load Balancer (ALB) is better suited for mid-tier load balancing than its original Elastic Load Balancer offering. If your deployment system manages ALB registration (which Spinnaker can do) and Route53 is used to provide consistent names for ALB addresses, you may not

need an additional service discovery mechanism, but you might want one anyway.

Netflix's Eureka works best in concert with the rest of the Netflix runtime platform (also primarily targeting the JVM), integrating service discovery, RPC transport and load balancing, circuit breaking, fallbacks, rate limiting and load shedding, dynamically customizable request routing for canaries and squeeze testing, metrics collection and event publication, and fault injection. We find all of these essential to building and operating robust, business-critical cloud services.

A number of newer open source service mesh projects, such as Linkerd and Envoy, both hosted by the CNCF, provide developers with similar features to the Netflix runtime platform. The service mesh combines service discovery with the advanced RPC features just mentioned, while being language and environment agnostic.

Using Multiple Clouds

The sixth thing to consider is multi-cloud strategy. Organizations take advantage of multiple cloud providers for a number of reasons. Service offerings or the global distribution of compute regions may complement each other. It may be in pursuit of enhanced redundancy or business continuity planning. Or it may come about organically after empowering different business units to use whichever solutions best fit their unique needs. When deploying to different clouds you should understand how features like identity management and virtual private cloud (VPC) networking differ between providers.

Abstracting Cloud Operations from Users

The final thing to consider is how your users will interact with the cloud(s) you've chosen. Solving the previous considerations for your organization provides the groundwork for enabling teams to move quickly and deploy often. In order to enforce the choices that you've made, or provide a "paved"/"golden" path for other teams, many organizations provide a custom view of the cloud providers they have. This custom view provides abstractions and can handle organizational needs like audit logging, integration with other internal tools, best practices in the form of codified deployment strategies, and a helpful customized view of the infrastructure.

For Netflix, that custom view of the cloud is called Spinnaker (see Figure 2-1). Over the years we've built Spinnaker to be flexible, extensible, resilient, and highly available. We have learned from our internal users that the tools we build need to make best practices simple, invisible, and opt-out. There are many built-in features to make best practices happen that will be discussed in this report. For

example, Spinnaker will always consider the combined health of a load balancer and service discovery before allowing a previous server group to be disabled during a deployment using a red/black strategy (discussed in detail in "Deploying and Rolling Back" on page 15). By enforcing this, we can ensure that if there is a bug in the new code, the previous server group is still active and taking traffic.

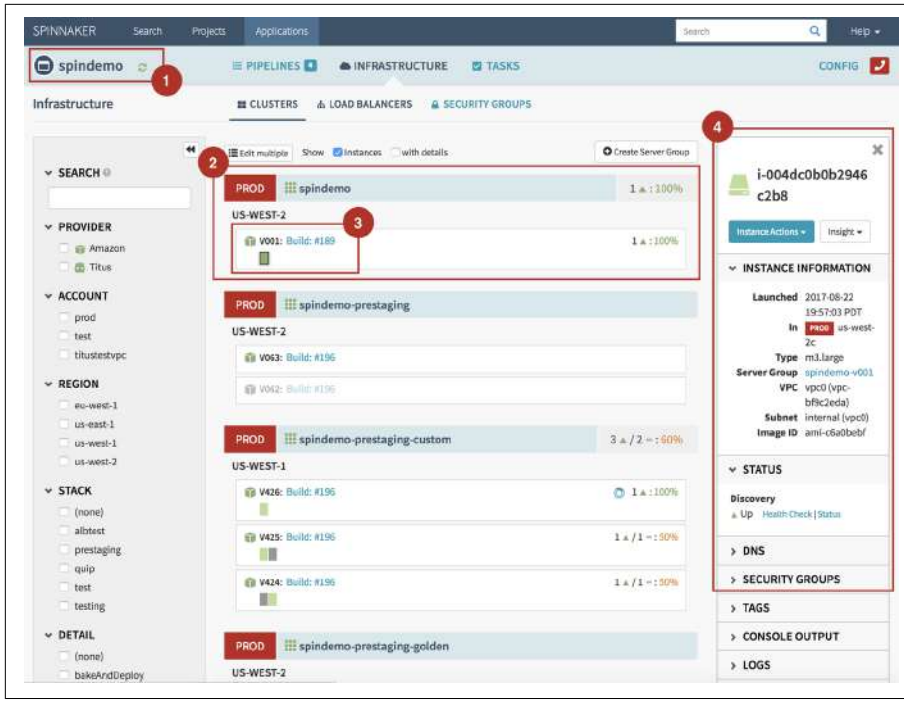

Figure 2-1. This is the main screen of Spinnaker. This view (the Infrastructure Clusters view) shows the resources in the application.

The Infrastructure Clusters view, shown in Figure 2-1, is just one screen of Spinnaker. This view nicely demonstrates how we abstract the two clouds (Amazon and Titus) away from our users. Box 1 shows the application name. Box 2 shows a cluster—a grouping of identically named server groups in an account (PROD), and the health of the cluster (100% healthy). Box 3 shows a single server group with one healthy instance running in US-WEST-2, running version v001, which corresponds to Jenkins build #189. Box 4 shows details for that single running instance, such as launch time, status, logs, and other relevant information.

Over the course of this report we will continue to show screenshots of the Spinnaker UI to demonstrate how Netflix has codified continuous delivery.

Summary

In this chapter, you have learned the fundamental parts of a cloud environment that must be considered in order to successfully deploy to the cloud. You learned about how Netflix approaches these problems as well as open source solutions that can help manage parts of these challenges. Once you've solved these problems within your cloud environment, you're ready to enable teams to deploy early and often into this environment. You will empower your teams to deploy their software without each team having to solve the problems covered in this chapter for themselves. Additionally, providing a custom view of the cloud that enforces best practices will help your teams draw from the lessons codified in that tool.

CHAPTER 3

Managing Cloud Infrastructure

Whether you are creating a cloud strategy for your organization or starting at a new company that has begun moving to the cloud, there are many challenges. Just understanding the scope of the resources, components, and conventions your company relies on is a daunting prospect. If it's a company that has a centralized infrastructure team, your team might even be responsible for multiple teams' cloud footprints and deployments.

Chapter 2 set the stage for this transition by describing the fundamental pieces of a cloud environment. In this chapter, you'll learn about some of the challenges found in modern multi-cloud deployments and how approaches like naming conventions can help in adding consistency and discoverability to your deployment process.

Organizing Cloud Resources

When thinking about how to manage the different resources that need to be deployed in the cloud, there are many questions that need to be asked about how those resources should to be organized:

- Do teams manage their own infrastructure or is it centralized?
- Do different teams have different conventions and approaches?
- Is everything in one account or split across many accounts?
- Do applications have dedicated server groups?
- Do resource names indicate their role in the cloud ecosystem?
- Are the instances or containers within a server group homogeneous?
- How are security and load balancing handled for internal-facing and external-facing services?

Only when these questions are answered can the teams working on deployments work out how to lay out and organize the resources.

Ad Hoc Cloud Infrastructure

Because most cloud platforms are quite unopinionated about the organization of resources, a company's cloud fleet might have been assembled in an ad hoc manner. Different application teams define their own conventions and thus the cloud ecosystem as a whole is riddled with inconsistencies. Each approach will surely have its justifications, but the lack of standardization makes it hard for someone to understand the bigger picture.

This will frequently happen where a company's use of the cloud has evolved over time. Best practices were likely undefined at first and only emerged over time.

Shared Cloud Resources

Sharing resources, such as security groups, between applications can make it hard to determine what is a vital infrastructure component and what is cruft. Cloud resources consume budget. Good conventions that help you keep track of whether resources are still used can make it easier to streamline your cloud footprint and save money.

The Netflix Cloud Model

Netflix's approach to cloud infrastructure revolves around naming conventions, immutable artifacts, and homogeneous server groups. Each application is composed of one or more server groups and all instances within that server group run an identical version of the application.

Naming Conventions

Server groups are named according to a convention that helps organize them into *clusters*:

<name>-<stack>-<detail>-v<version>

- The *name* is the name of the application or service.
- The (optional) *stack* is typically used to differentiate production, staging, and test server groups.
- The (optional) *detail* is used to differentiate special-purpose server groups. For example, an application may run a Redis cache or a group of instances dedicated to background work.
- The *version* is simply a sequential version number.

A server group at Netflix consists of one or more homogeneous instances. A cluster consists of one or more server groups that share the same *name*, *stack*, and *detail*. A cluster is a Spinnaker concept derived from the naming convention applied to the server groups within it.

Versioning

Each server group within a cluster typically has a different version of the application on it, and all instances within the server group are homogenous—that is, they are configured identically and have the same machine image.

Instances within a server group are interchangeable and disposable. Server groups can be resized up or down to accommodate spikes and troughs in traffic. Instances that fail can be automatically replaced with new ones.

Usually, only one server group in a cluster is active and serving traffic at any given time. Others may exist in a disabled state to allow for quick rollbacks if a problem is detected.

Deploying and Rolling Back

The typical deployment procedure in the Netflix cloud model is a "red/black" deployment (sometimes known elsewhere as a "blue/green").

In a red/black deployment, a new server group is added to the cluster, deploying a newer version of the application. The new server group keeps the name, stack, and detail elements, but increments the version number (Figure 3-1).

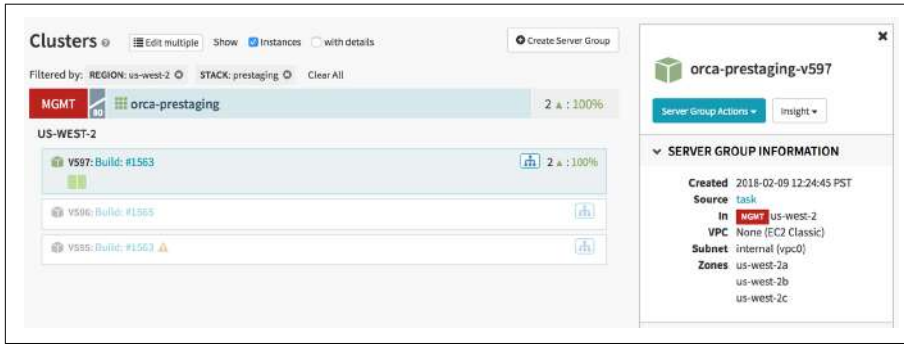

Figure 3-1. A cluster containing three server groups, two of which are disabled. Note the full server group name in the panel on the right, along with details about that server group.

Once deployed and healthy, the new server group is enabled and starts taking traffic. Only once the new server group is fully healthy does the older server group get disabled and stop taking traffic.

This procedure means deployments can proceed without any application downtime—assuming, of course, that the application is built in such a way that it can cope with "overlapping" versions during the brief window where old and new server groups are both active and taking traffic.

If a problem is detected with the new server group, it is very straightforward to roll back. The old server group is re-enabled and the new one disabled.

Applications will frequently resize the old server group down to zero instances after a predefined duration. Rolling back from an empty server group is a little slower, but still faster than redeploying, and has the advantage of releasing idle instances, saving money and returning instances to a reservation pool where other applications can use them for their own deployments.

Alternatives to Red/Black Deployment

Variations on this deployment strategy include:

Rolling push
: The machine image associated with each instance in a server group is upgraded and then restarted in turn.

Rolling red/black
: The new server group is deployed with zero instances and gradually resized up in sync with the old server group being resized down, resulting in a gradual shift of traffic across to the new server group.

Highlander
: The old server group is immediately destroyed after being disabled. The name comes from the 1985 movie of the same name, where "There can be only one"! This strategy is usually only used for test environments.

Self-Service

Adopting consistent conventions enables teams to manage their own cloud infrastructure. At Netflix, there is no centralized team managing infrastructure. Teams deploy their own services and manage them once they go live.

Cross-Region Deployments

Deploying an application in multiple regions brings its own set of concerns. At Netflix, many externally facing applications are deployed in more than one region in order to optimize latency between the service and end users.

Reliability is another concern. The ability to reroute traffic from one region to another in the event of a regional outage is vital to maintaining uptime. Netflix even routinely practices "region evacuations" in order to ensure readiness for a catastrophic EC2 outage in an individual region.

Ensuring that applications are homogeneous between regions makes it easier to replicate an application in another region, minimizing downtime in the event of

having to switch traffic to another region or to serve traffic from more than one region at the same time.

Active/Passive

In an active/passive setup, one region is serving traffic and others are not. The inactive regions may have running instances that are not taking traffic—much like a disabled server group may have running instances in order to facilitate a quick rollback.

Persistent data may be replicated from the active region to other regions, but the data will only be flowing one way, and replication does not need to be instantaneous.

Active/Active

An active/active setup has multiple regions serving traffic concurrently and potentially sharing state via a cross-region data store. Supporting an active/active application means enabling connectivity between regions, load-balancing traffic across regions, and synchronizing persistent data.

Multi-Cloud Configurations

Increasing the level of complexity still further, more and more companies are now using more than one cloud platform concurrently. It's not unusual to have deployments in EC2 and ECS, for example. There are even companies using different platforms for their production and test environments.

Even if you're currently using only one particular cloud, there's always the potential for an executive decision to migrate from one provider to another.

The concepts used by each cloud platform have subtle differences and the tools provided by each cloud vary greatly.

The Application-Centric Control Plane

Not only do the tools vary across cloud platforms, but the way they are organized is typically resource type centric rather than application centric.

For example, in the AWS console, if you need to manage instances, server groups (autoscaling groups in EC2), security groups, and load balancers, you'll find they are organized into entirely separate areas of the console. If your application also spans multiple regions and/or accounts, you'll find that there's an awful lot of clicking around different menus to view the resources for a given application. Each account requires its own login and each region is managed by its own separate console.

That arrangement may make sense if you have a centralized infrastructure team managing the company's entire cloud fleet. However, if you're having each application team manage their own deployments and infrastructure, a single control plane that is organized around their application is much more useful. In an application-centric control plane, all the resources used by an application are accessible in one place, regardless of what region, account, or even cloud they belong to (Figure 3-2).

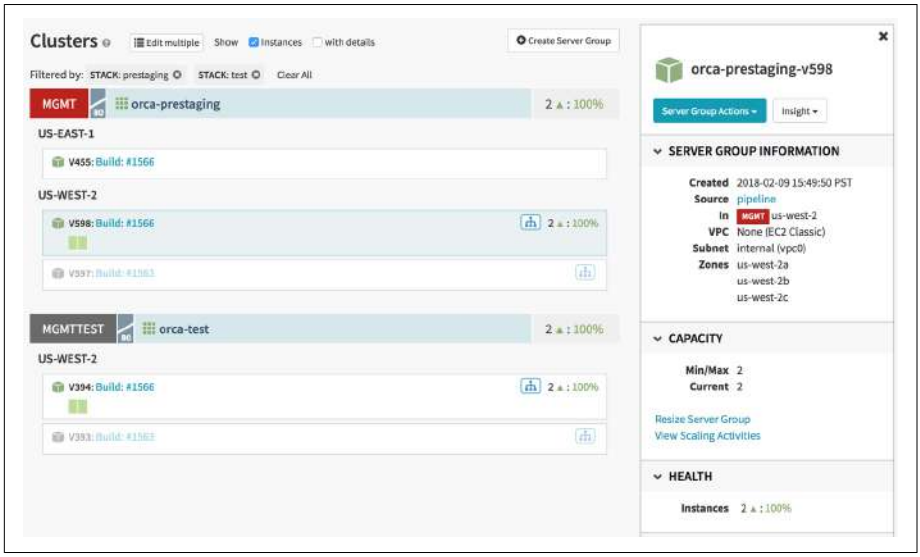

Figure 3-2. A Spinnaker view showing clusters spanning multiple EC2 accounts and regions. Load balancers, security groups, and other aspects are accessible directly from this view.

Such a control plane can link out to external systems for metrics monitoring or provide links to ssh onto individual instances.

Multi-Cloud Applications

Applications that deploy resources into multiple clouds will benefit from common abstractions, such as those that Spinnaker provides. For example, an autoscaling group in EC2 is analogous to a managed instance group in GCE or a ReplicaSet in Kubernetes, an EC2 security group is comparable to a GCE firewall, and so on.

With common abstractions, an application-centric control plane can display resources from multiple clouds alongside one another. Where differences exist they are restricted to more detailed views.

Spinnaker abstracts specific resources to facilitate multi-cloud deployment. There are many other services provided by each cloud provider that Spinnaker does not have abstractions for (and may not know about).

Summary

One of the main challenges in managing cloud resources across accounts, regions, and cloud providers is consistency. Most control planes are built with 50 or 100 resources in mind and break down when that number grows to 1,000, 10,000, or 100,000 resources that need to be tracked. Best practices in cross-region and cross-account resiliency means an explosion of resources across the typical cloud-provider account/region boundary.

By introducing an application-centered naming convention that aggressively filters the number of resources presented to maintainers, we can make it easier to notice things that are awry and manually fix them. This standardization is useful for teams managing applications as well as centralized teams working on cloud tooling.

CHAPTER 4

Structuring Deployments as Pipelines

In this chapter you'll learn about the benefits of structuring your deployments out of customizable pieces, the parts of a Spinnaker pipeline, and how codifying and iterating on your pipeline can help reduce the cognitive load of developers. At the end of this chapter, you should be able to look at a deployment process and break down different integration points into specific pipeline parts.

Benefits of Flexible User-Defined Pipelines

Most deployments consist of similar steps. In many cases, the code must be built and packaged, deployed to a test environment, tested, and then deployed to production. Each team, however, may choose to do this a little differently. Some teams conduct functional testing by hand whereas others might start with automated tests. Some environments are highly controlled and need to be gated with an approval by a person (manual judgment), whereas others can be updated automatically whenever there is a new change.

At Netflix, we've found that allowing each team to build and maintain their own deployment pipeline from the building blocks we provide lets engineers experiment freely according to their needs. Each team doesn't have to develop and maintain their own way to do common actions (e.g., triggering a CI build, figuring out which image is deployed in a test environment, or deploying a new server group) because we provide well-tested building blocks to do this. Additionally, these building blocks work for every infrastructure account and cloud provider we have. Teams can focus on iterating on their deployment strategy and building their product instead of struggling with the cloud.

Spinnaker Deployment Workflows: Pipelines

In Spinnaker, pipelines are the key workflow construct used for deployments. Each pipeline has a configuration, defining things like triggers, notifications, and a sequence of stages. When a new execution of a pipeline is started, each stage is run and actions are taken.

Pipeline executions are represented as JSON that contains all the information about the pipeline execution. Variables like time started, parameters, stage status, and server group names all appear in this JSON, which is used to render the UI.

Pipeline Stages

The work done by a pipeline can be divided into smaller, customizable blocks called stages. Stages are chained together to define the overall work done as part of the continuous delivery process. Each type of stage performs a specific operation or series of operations. Pipeline stages can fall into four broad categories.

Infrastructure Stages

Infrastructure stages operate on the underlying cloud infrastructure by creating, updating, or deleting resources.

These stages are implemented for every cloud provider where applicable. This means that if your organization leverages multiple clouds, you can deploy to each of them in a consistent way, reducing cognitive load for your engineers.

Examples of stages of this category include:

- Bake (create an AMI or Docker image)
- Tag Image
- Find Image/Container from a Cluster/Tag
- Deploy
- Disable/Enable/Resize/Shrink/Clone/Rollback a Cluster/Server Group
- Run Job (run a container in Kubernetes)

Bake stages take an artifact and turn it into an immutable infrastructure primitive like an Amazon Machine Image (AMI) or a Docker image. This action is called "baking." You do not need a bake step to create the images you will use—it is perfectly fine to ingest them into Spinnaker in another way.

Tag Image stages apply a tag to the previously baked images for categorization. Find Image stages locate a previously deployed version of your immutable infrastructure so that you can refer to that same version in later stages.

The rest of the infrastructure stages operate on your clusters/server groups in some way. These stages do the bulk of the work in your deployment pipelines.

External Systems Integrations

Spinnaker provides integrations with custom systems to allow you to chain together logic performed on systems other than Spinnaker.

Examples of this type of stage are:

- Continuous Integration: Jenkins/TravisCI
- Run Job
- Webhook

Spinnaker can interact with Continuous Integration (CI) systems such as Jenkins. Jenkins is used for running custom scripts and tests. The Jenkins stage allows existing functionality that is already built into Jenkins to be reused when migrating from Jenkins to Spinnaker.

The custom Webhook stage allows you to send an HTTP request into any other system that supports webhooks, and read the data that gets returned.

Testing

Netflix has several testing stages that teams can utilize. The stages are:

- Chaos Automation Platform (ChAP) (internal only)
- Citrus Squeeze Testing (internal only)
- Canary (open source)

The ChAP stage allows us to check that fallbacks behave as expected and to uncover systemic weaknesses that occur when latency increases.

The Citrus stage performs squeeze testing, directing increasingly more traffic toward an evaluation cluster in order to find its load limit.

The Canary stage allows you to send a small amount of production traffic to a new build and measure key metrics to determine if the new build introduces any performance degradation. This stage is also available in OSS. These stages have been contributed by other Netflix engineers to integrate with their existing tools.

Additionally, functional tests can also be run via Jenkins.

Controlling Flow

This group of stages allows you to control the flow of your pipeline, whether that is authorization, timing, or branching logic. The stages are:

- Check Preconditions
- Manual Judgment
- Wait
- Run a Pipeline

The Check Preconditions stage allows you to perform conditional logic. The Manual Judgment stage pauses your pipeline until a human gives it an OK and propagates their credentials. The Wait stage allows you to wait for a custom amount of time. The Pipeline stage allows you to run another pipeline from within your current pipeline. With these options, you can customize your pipelines extensively.

Triggers

The final core piece of building a pipeline is how the pipeline is started. This is controlled via triggers. Configuring a pipeline trigger allows you to react to events and chain steps together. We find that most Spinnaker pipelines are set up to be triggered off of events. There are several trigger types we have found important:

Time-based triggers:

- Manual
- Cron

Event-based triggers:

- Git
- Continuous Integration
- Docker
- Pipeline
- Pub/Sub

Manual triggers are an option for every pipeline and allow the pipeline to be run ad hoc. Cron triggers allow you to run pipelines on a schedule.

Most of the time you want to run a pipeline after an event happens. Git triggers allow you to run a pipeline after a git event, like a commit. Continuous Integration triggers (Jenkins, for example) allow you to run a pipeline after a CI job completes successfully. Docker triggers allow you to run a pipeline after a new Docker image is uploaded or a new Docker image tag is published. Pipeline triggers allow you to run another pipeline after a pipeline completes successfully. Pub/Sub triggers allow you to run a pipeline after a specific message is received from a Pub/Sub system (for example Google Pub/Sub, or Amazon SNS).

With this combination of triggers, it's possible to create a highly customized workflow bouncing between custom scripted logic (run in a container, or through Jenkins) and the built-in Spinnaker stages.

Notifications

Workflows that are automatically run need notifications to broadcast the status of events. Spinnaker pipelines allow you to configure notifications for pipeline start, success, and failure. Those same notification options are also available for each stage. Notifications can be sent via email, Slack, Hipchat, SMS, and Pub/Sub systems.

Expressions

Sometimes the base options aren't enough. Expressions allow you to customize your pipelines, pulling data out of the raw pipeline JSON. This is commonly used for making decisions based on parameters passed into the pipeline or data that comes from a trigger.

For example, you may want to deploy to a test environment from your Jenkins triggered pipeline when your artifact name contains "unstable," and to prod otherwise. You can use expressions to pull the artifact name that your Jenkins job produced and use the Check Preconditions stage to choose the branch of your pipeline based on the artifact name. Extensive expression documentation is available on the Spinnaker website.[1]

Exposing this flexibility to users allows them to leverage pipelines to do exactly what they want without needing to build custom stages or extend existing ones for unusual use cases, and gives engineers the power to iterate on their workflows.

Version Control and Auditing

All pipelines are stored in version control, backed by persistent storage. We have found it's important to have your deployments backed by version control because it allows you to easily fix things by reverting. It also gives you the confidence to make changes because you know you'll be able to revert if you cause a regression in your pipeline.

We have also found that auditing of events is important. We maintain a history of each pipeline execution and each task that is run. Spinnaker events, such as a new pipeline execution starting, can be sent to a customizable endpoint for aggregation and long-term storage. Our teams that deal with sensitive information use this feature to be compliant.

1 *www.spinnaker.io*

Example Pipeline

To tie all these concepts together we will walk through an example pipeline, pictured in Figure 4-1.

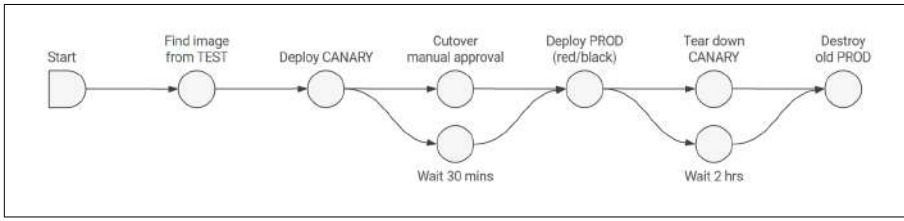

Figure 4-1. A sample Spinnaker deployment pipeline.

This pipeline interacts with two accounts, called TEST and PROD. It consists of a manual start, several infrastructure stages, and some stages that control the flow of the pipeline. This pipeline represents the typical story of taking an image that has already been deployed to TEST (and tested in that account), using a canary to test a small amount of production traffic, then deploying that image into PROD. This pipeline takes advantage of branching logic to do two things simultaneously.

This pipeline finds an image that is running in the TEST account and then deploys a canary of that image to the PROD account. The pipeline then waits for the canary to gather metrics, and also waits for manual approval. Once both of these actions complete (including a user approving that the pipeline should continue) a production deployment proceeds using the "red/black" strategy discussed in Chapter 3 (old instances are disabled as soon as new instances come up). The pipeline stops the canary after the new production server group is deployed, and waits two hours before destroying the old production infrastructure.

This is one example of constructing a pipeline from multiple stages. Structuring your deployment from stages that handle the infrastructure details for you lowers the cognitive load of the users managing the deployments and allows them to focus on other things.

NOTE

Jenkins Pipelines Versus Spinnaker Pipelines

We often receive questions about the difference between Jenkins pipelines and Spinnaker pipelines. The primary difference is what happens in each stage. Spinnaker stages, as you've just seen, have specifically defined functionality and encapsulate most cloud operations. They are opinionated and abstract away cloud specifics (like credentials and health check details). Jenkins stages have no native support for these cloud abstractions so you have to rely on plug-ins to provide it. We have seen teams use Spinnaker via its REST API to provide this functionality.

Summary

In this chapter, we have seen the value of structuring deployment pipelines out of customizable and reusable pieces. You learned the building blocks that we find valuable and how they can be composed to follow best practices for production changes at scale.

Pipelines are defined in a pipeline configuration. A pipeline execution will happen when that particular configuration is invoked either manually or via a trigger. As a pipeline runs, the pipeline will transition across the stages and do the work specified by each stage. As stages run, notifications or auditing events will be invoked depending on stages starting, finishing, or failing. When this pipeline execution finishes, it can trigger further pipelines to continue the deployment flow.

As more functionality is added into Spinnaker, new stages, triggers, or notification types can be added to support the new features. Teams can easily change and improve their deployment processes to use these new features while continuing to use best practices.

CHAPTER 5

Working with Cloud VMs: AWS EC2

Now that you have an understanding of continuous deployment and the way that Spinnaker structures deployments as pipelines, we will dive into the specifics of working with cloud VMs, using Amazon's EC2 as an example.

For continuous deployment into Amazon's EC2 virtual machine–based cloud, Spinnaker models a well-known set of operations as pipeline stages. Other VM-based cloud providers have similar functionality.

In this chapter, we will discuss how Spinnaker approaches deployments to Amazon EC2. You will learn about the distinct pipeline stages available in Spinnaker and how to use them.

Baking AMIs

Amazon Machine Images, or AMIs, can be thought of as a read-only snapshot of a server's boot volume, from which many EC2 instances can be launched.

In keeping with the immutable infrastructure pattern, every release of a service deployed via Spinnaker to EC2 first requires the creation (or baking) of a new AMI. Rosco is the Spinnaker bakery service. Under the hood, Rosco uses Packer, an extensible open source tool developed by HashiCorp, for creating machine images for all of the cloud platforms Spinnaker supports.

A Bake stage is typically the first stage in a Spinnaker pipeline triggered by an event, such as the completion of a Jenkins build or a GitHub commit Figure 5-1. Rosco is provided information about the artifact that is the subject of the bake, along with the base AMI image that forms the foundation layer of the new image. After the artifact is installed on top of a copy of the Base AMI, a new AMI is published to EC2, from which instances can be launched.

Figure 5-1. A Bake stage config for the service clouddriver at Netflix.

At Netflix, most services running in EC2 are baked on top of a common Base AMI containing an Ubuntu OS with Netflix-specific customizations. We like this approach for faster, more consistent bakes versus applying a configuration management system (such as puppet or chef) at bake time.

Tagging AMIs

By following a Bake stage with a Tag Image stage in the same pipeline, you can tag newly created AMIs for greater control over how they are deployed.

Suppose you have a CI system that builds each commit to any git branch, then triggers a Spinnaker bake pipeline. The resulting AMI can be tagged with the branch name (provided as a trigger parameter) and deployed to a server group also including the branch name (i.e., *myservice-test-mybranch-v001*) for testing. Pipelines intended to shepherd a build to the production environment can be configured to ignore branch-tagged AMIs or to look for a specific tag such as *master* or *release*.

Deploying in EC2

Setting up an EC2 deployment pipeline for the first time can seem overwhelming, due to the wealth of options available. The Basic Settings cover AWS account and region, as well as server group naming and which deployment strategy to use, both discussed in Chapter 3.

If you have more than one VPC subnet configured, you can select that here (Figure 5-2). It's good practice to separate internet-facing and internal services into different subnets, as well as production versus developer environments.

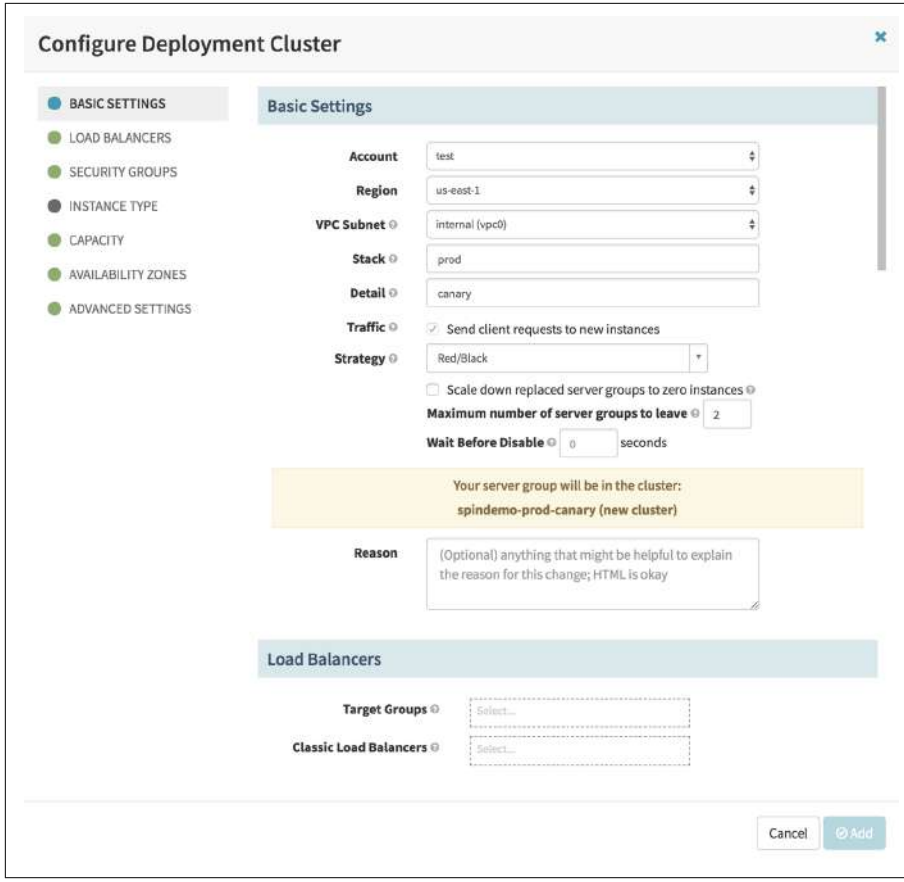

Figure 5-2. Basic EC2 deployment settings.

EC2 provides several native load-balancer options to route traffic to new instances. Spinnaker supports the newer ALB type via the Target Groups dropdown, as well as the original ELB type, labeled Classic Load Balancers. If you're starting from scratch, the ALB type provides improved performance over ELB, while also supporting Layer 7 routing features.

Add your server group to one or more security groups, and select an instance type and the number of instances to deploy (or min/max/desired if you plan to use autoscaling). That may be all the configuration required.

Availability Zones

Scrolling further down in the deployment configuration wizard reveals the ability to customize how instances are distributed across Availability Zones (AZs) within a region. By default, Spinnaker launches server groups that automatically balance instances across several AZs for greater resiliency.

By disabling AZ balancing, you can pin all instances in a server group to a single AZ. At Netflix, we typically use automatic balancing with stateless services but disable it for persistent services where AZs and region redundancy are handled more deliberately.

Health Checks

Spinnaker needs to know when a service is healthy in order to deliver safe deployments. When deploying a server group via a red/black or Highlander strategy, you don't want the original server group to be disabled or terminated before the new server group is actually healthy and taking traffic.

For EC2, it is not enough to just look at the up and down state of the instances in the server group. EC2 will mark an instance as up as soon as it boots up with a network stack; this often occurs before underlying processes like Tomcat are ready to serve requests. In order to safely orchestrate the enabling and disabling of server groups, we must rely on secondary health check mechanisms such as the ones from load balancers and service discovery systems.

For services sitting behind an ELB or ALB in EC2, Spinnaker and the EC2 autoscaler can share the load balancer's view of application health. When configuring a load-balancer health check, be sure to monitor an endpoint that is inexpensive to serve, yet accurately reflects an application's ability to serve requests.

Spinnaker also integrates with service discovery systems. Service discovery support is pluggable, but Netflix's open source Eureka platform remains the best-supported discovery platform on AWS. If Eureka integration is enabled, Spinnaker automatically sees if an application registers with it, and will check if individual instances are registered as up when determining health. Spinnaker also supports and integrates with Consul for other cloud providers.

Advanced Configuration

Spinnaker enables users to configure additional resources associated with the launch configuration in AWS (Figure 5-3). Advanced deployment settings also provide the ability to pass UserData to instances at launch time, assign custom IAM roles, customize Block Device Mappings, and more.

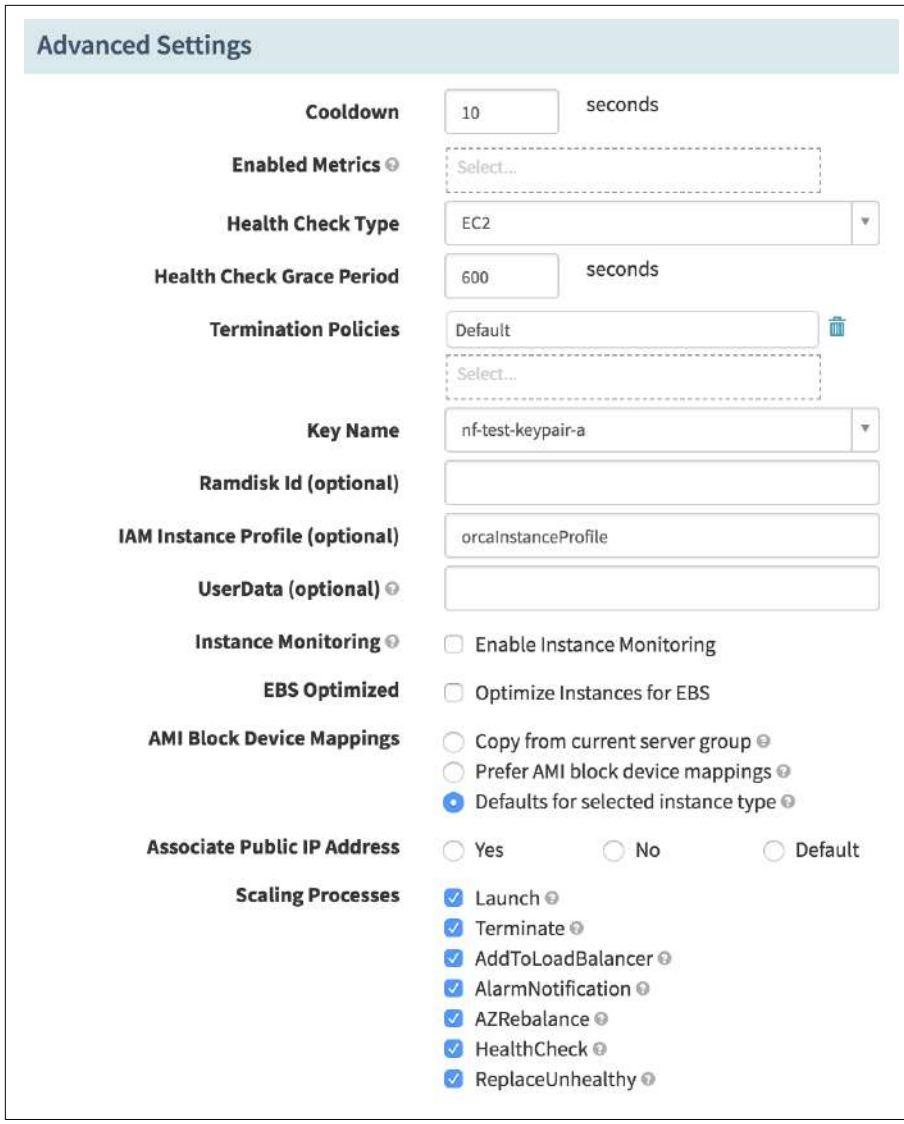

Figure 5-3. Advanced deployment settings.

Autoscaling

Spinnaker supports two types of EC2 autoscaling policies: Step and Target Tracking. Step policies are rule based, such as "If Average CPU Utilization > 50% for 2 minutes, add 1 instance; if >75%, add 3 instances."

Target Tracking policies are a more recent AWS feature (Figure 5-4). They're simpler to configure and more responsive to sudden load changes. A Target Tracking

policy is based on a single metric and its target value. The autoscaler continuously monitors the metric and scales the server group to keep the metric as close as possible to its target.

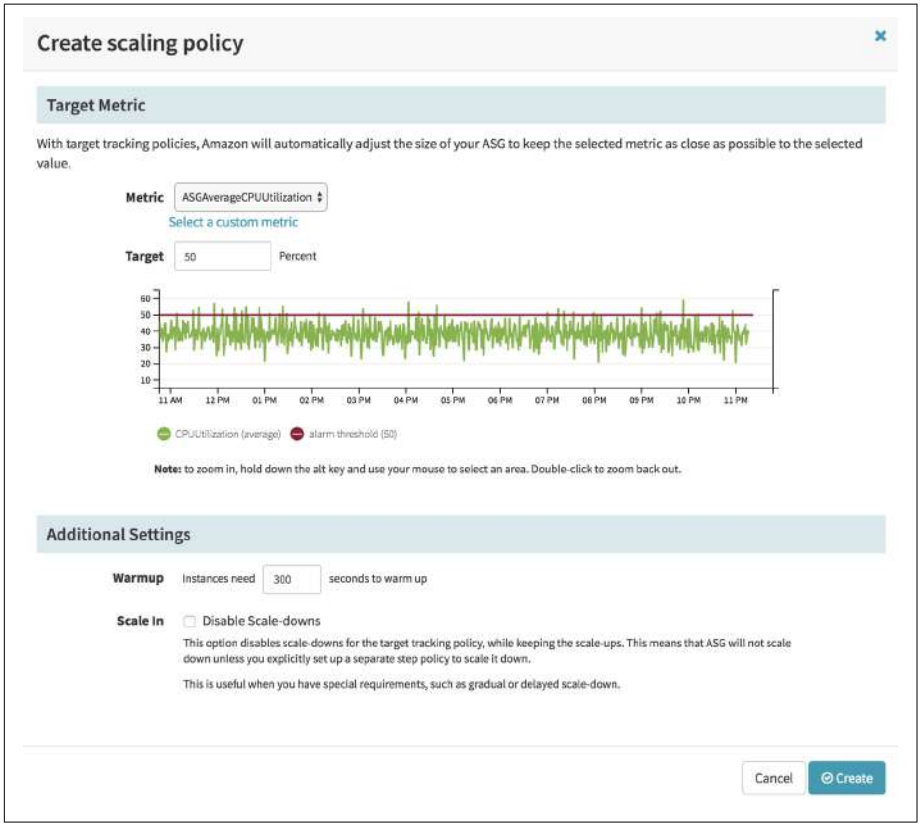

Figure 5-4. Creating an EC2 Target Tracking policy in Spinnaker.

Only the size constraints for a server group (defining minimum, maximum, and desired instance counts for the autoscaler) can be defined within a Spinnaker Deploy stage. To define an actual scaling policy, make sure you have at least one server group deployed within the cluster that you'd like to autoscale. Select the most recent server group, then expand the Scaling Policies section of the right bar to reveal the "Create a new scaling policy" link.

Once a scaling policy has been created, it will automatically be copied over to new server groups deployed within that cluster. However, if all server groups within a cluster are deleted, the scaling policy will have to be redefined. To improve that experience, EC2 scaling policy management is an initial feature of Spinnaker's nascent Declarative Delivery initiative, discussed later on in this report.

Summary

In this chapter, we discussed how Spinnaker simplifies the creation of images via a bake, tagging, and deployment to Amazon EC2. This same pattern can apply to similar virtual machine–based systems like cloud providers such as Google Compute Engine and Microsoft Azure.

Deployment to EC2 is not only restricted to creating a new server group. Spinnaker also needs to manage autoscaling and health checks during the deploy and rollback cycles in the continuous deployment process. By doing the heavy lifting around these operations, it takes the cognitive burden of managing them away from users. Spinnaker's application-centric view of infrastructure management helps group commonly needed resources in a way that simplifies interactions with the resources.

CHAPTER 6
Kubernetes

In the previous chapter you learned about the specifics of VM-based deployments using Amazon's EC2 instances as an example.

In this chapter you'll learn what makes continuous delivery (CD) pipelines to Kubernetes different from CD pipelines to VM-based clouds. You'll also learn what your organization needs to consider when designing a CD pipeline for Kubernetes, and how tooling such as Spinnaker helps you.

What Makes Kubernetes Different

Whether you are migrating workloads to Kubernetes, or Kubernetes is your first step into cloud deployments, it's good to know what makes Kubernetes different, from a CD perspective. This is especially true because most existing knowledge and tooling comes from deployments to VM-based clouds.

Faster
: Deployments to Kubernetes are generally much faster. Provisioning resources in Kubernetes takes seconds, while provisioning a VM can take minutes. This means your developers can very quickly deploy to a live cluster, and it takes less time to create testing environments and promote releases through staging environments. In short: time spent waiting for infrastructure to provision is less of a concern when deploying to Kubernetes.

Declarative
: Kubernetes uses manifest files to provide a declarative description of your infrastructure—it's central to how Kubernetes works. Everything you provision and deploy, from the containers you run to the network policies governing traffic, are described in YAML. Kubernetes always tries to reach the state you have specified using its own orchestration, rolling out binary changes or changing routing rules as needed. This provides you with easier recoverabil-

ity, the ability to code review changes to your infrastructure, and a higher level of abstraction over your underlying cloud resources.

Multi-cloud
: Whether Kubernetes is running in Google's cloud or Amazon's, in your on-premise datacenter or on your laptop, it exposes the same interface and behavior for running your workloads. This makes it trivial to deploy the same application to multiple clouds and regions, when you can treat each as being identical. This also makes it much easier to create staging and developer environments that model production.

Native deployment orchestration
: When a change is submitted to a running Kubernetes workload, it orchestrates a rollout of your change according to policies you specify. In some cases, this becomes the only deployment orchestration that you need, and can be carried out independently of the delivery platform that you choose.

Considerations

In the context of Kubernetes, it is common to ask why delivery tooling is needed on top of native Kubernetes deployment orchestration. After all, rolling out a binary or configuration change can be done with a single command using the Kubernetes command-line interface, `kubectl`. However, it can be desirable to have first-class distinctions between "production" and "staging" environments, access-control configuration per application, support for ingesting events from Docker registries, cross-cluster orchestration, and many more features you'd have to build on top of Kubernetes yourself.

There are many ways to configure your delivery pipelines to Kubernetes—too many to enumerate here. Instead, we'll list some considerations you should make when designing these pipelines, as well as ways in which tools such as Spinnaker can help enforce best practices, and automate the complicated and/or laborious parts of your delivery pipelines.

How Are You Building Your Artifacts?

Before you can deploy your code, it needs to be built into a Docker image. You might also be using a templating system, such as Helm, or Ksonnet, to create your manifests. These templates need to be hydrated before they are deployed. Generally, we consider the creation of these artifacts to be a part of your continuous integration (CI) pipeline, which will provide the artifacts it creates to your CD pipelines to deploy. Typically, this represents the separation of concerns between CI and CD: CI produces and validates artifacts, while CD deploys them. However, it's worth keeping in mind that the intersection between CI and CD is not always perfectly clear, and varies between teams and organizations.

Spinnaker provides integrations with CI systems such as Jenkins and Travis CI, as well as a wide range of Docker registries, from DockerHub to Google Container Registry (GCR), to trigger your pipelines. On top of that, more flexible payloads can be delivered to Spinnaker to trigger pipelines using webhooks or Pub/Sub. As mentioned before, the intersection between CI and CD may be different in your organization. To accommodate this, Docker images can be baked using Spinnaker's first-class Bake Image stage, and arbitrary manifest templates can be hydrated using Spinnaker's Run Job stage.

Is Your Deployed Configuration and Image Versioned?

When a team is deploying configuration and code that they own, it is important that they can identify exactly what they have running in their cluster at a certain point in time, as well as have the ability to roll back changes with confidence. This is only possible when the configuration and code that is running is uniquely identifiable each time it is deployed. The key advantage to practicing this is having repeatability of deployments, and auditability of your environment.

Docker images can be identified by a "tag" and a "digest." A tag can be applied by a user, and the image it refers to can be changed. For this reason, deploying an image by its tag is problematic—the image it refers to can change over time. The digest is a content-based hash of the image, so it will always uniquely identify an image, making it the preferred way to refer to images. Spinnaker always deploys an image by its digest if possible, but also supports ingesting events from Docker registries and build systems that describe images by both their tag and digest.

Kubernetes supports deploying configuration in a ConfigMap resource. The contents can be mounted by an application in either environment variables or a volume. By default, changes to an existing ConfigMap's contents do not change the ConfigMap's identifier, meaning each revision of the ConfigMap is not uniquely identifiable. This can be useful when this configuration needs to be hot-reloaded, or when the configuration's lifecycle is independent of the application (such as a database connection URL). However, in other cases, configuration changes need to be versioned to be rolled back. To allow for this, Spinnaker automatically assigns versions to configuration resources, and injects them into an application when deployed.

Should Kubernetes Manifests Be Abstracted from Your Users?

Kubernetes manifest files can be confusing to anyone not familiar with Kubernetes. It can be argued that while developers should control how their code reaches production, they don't need to know all the details of the manifests used by the underlying infrastructure they depend on.

While this isn't always the case, one option is to rely on templates, configured by a specialist team, that require a few variables for use in hydrating manifest files

that define an application. Another option is provided by the Kubernetes provider V1 in Spinnaker, which lays out manifest configuration options in a tooltip annotated UI, and provides context-driven dropdowns wherever possible. Both options reduce the burden on developers that are not, and do not need to be, Kubernetes experts.

When Is a Deployment "Finished"?

Most introductions to Kubernetes rely on its command-line interface `kubectl`. In embracing Kubernetes's declarative approach to infrastructure management, submitting a changed manifest using `kubectl` returns as soon as Kubernetes accepts the manifest. However, this does not imply that the desired changes have been made. Kubernetes might still be rolling out the change, waiting for health checks to pass, or attaching a disk waiting for quota to arrive. None of these are guaranteed to ever complete. It is important that the tooling you choose does not prematurely assume that a manifest change was applied.

Spinnaker has policies built in for each kind of Kubernetes resource, ranging from ensuring that all health checks pass in a deployment resource, to waiting until a deployed volume claim binds the storage it has requested. These may seem trivial, but they prevent you from having to write custom logic, and they ensure that your pipelines don't succeed until your changes have successfully been rolled out.

How Do You Handle Recoverability?

If an application is defined as a set of manifests, and the cluster it is running on is lost, one can quickly recover it by redeploying those manifests to a new cluster behind the same load balancer. However, this gets complicated when an application has dependencies on other applications in that cluster, or when manifests are stored as templates where some properties (such as the version of your Docker image) aren't known until they are deployed.

There are two options for providing recoverability:

1. Periodically snapshot the state of your cluster, capturing the definitions of all running manifests. This is conceptually simple and provides a clean state to restart your cluster from. But you might capture states where deployed manifest definitions are "bad" (health checks are about to fail, resources can't be allocated).
2. Only record changes in your backup for a manifest when its deployment finishes, as described earlier. Spinnaker makes this easy by emitting the fully hydrated manifest from every pipeline that deploys it to Spinnaker's event bus, which can forward events to a range of systems.

Summary

The considerations discussed in this chapter, read with an understanding of your organization's needs, should help clarify how to build your CD pipelines to Kubernetes. Above all, the tooling you choose to support and express these pipelines should ultimately make deployments as simple and boring to your developers as possible. However, that will take effort, careful design, and the right tools to accomplish.

CHAPTER 7

Making Deployments Safer

So far we have explored moving to the cloud, structuring deployments as pipelines, and the specifics and considerations of deployment for VMs and containers.

The ultimate goal of embracing continuous delivery is to allow users to deploy software quickly and automatically. A big part of making this practice successful is to be able to push new code without fear. Automation is great, but it is better if there are proper safeguards to ensure we never get into a state where systems are down and customers are negatively impacted.

In this chapter, we will catalog some of the techniques, actions, and practices that were added to Spinnaker with the goal of making deployments safer.

Cluster Deployments

The following types of safeguards ensure that new versions of software can be added and removed safely. In Spinnaker's cloud model, where new server groups in a cluster are mapped to software versions, we can add additional checks to ensure availability.

Deployment strategies
: Out of the box, Spinnaker comes with four deployment strategies:

  Red/black
  : Enables next server group, disables last one.

  Rolling red/black
  : Same as above, but in incremental percentages, i.e., 25%, 50%, 100%.

  Highlander
  : Destroys all server groups except current active one; there can be only one.

Custom
: User-defined.

You can also choose to not have a deployment strategy, in which case a new server group would just be created alongside the existing one. Deployment strategies increase safety by only removing server groups once the new one is active and ready.

Under the hood, deployment strategies like red/black and rolling red/black also interact with autoscalers to ensure that capacity dimensions are preserved across the deployment and server groups don't become over- or underprovisioned throughout the deployment.

Easy rollbacks
: It should be easy to revert changes if an issue has been encountered. Spinnaker does this by offering an action where a server group can be quickly rolled back to the previous state (Figure 7-1). As a rollback happens, it will first ensure that the previous server group is properly sized and taking traffic before disabling the bad version.

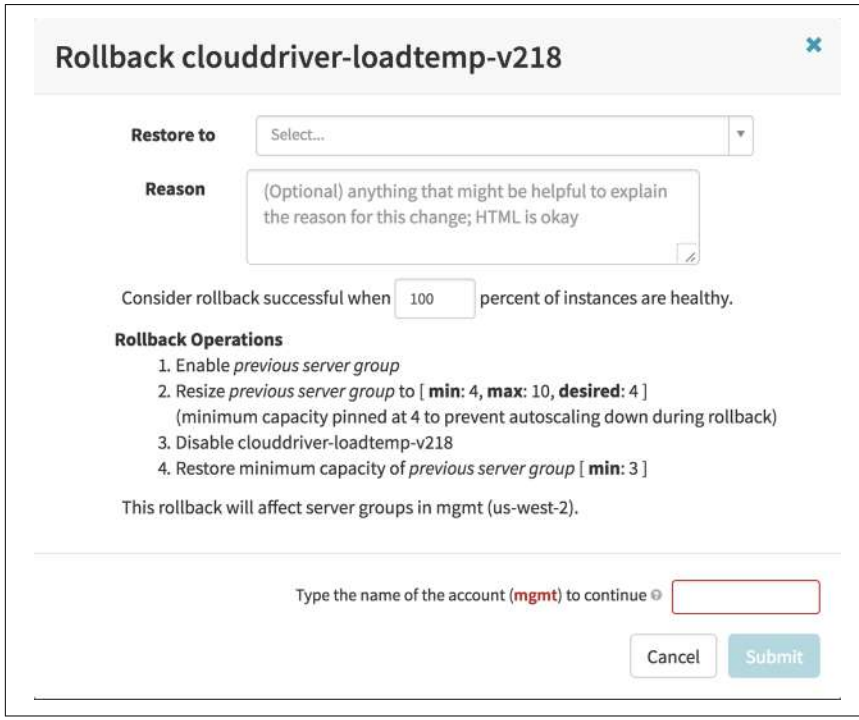

Figure 7-1. Rollback dialog.

In the dialog in Figure 7-1, you will see that all the operations that an automated rollback performs are explicitly listed for the operator under the Rollback Operations header.

Rolling back can be a long-running process and is usually performed under duress. Before Spinnaker, all those individual steps would also have been taken manually. The person rolling back would have to check that the number of instances looks OK before deactivating the old cluster. Now, they just push a button and Spinnaker does the work for them.

Cluster locking
: When a new server group is being added to a cluster, Spinnaker creates a protective bubble around it. If a conflicting server group is asked to be created at the same time in the same region and cluster, Spinnaker will wait until the first server group has finished deploying and releases its lock. This cluster locking and exclusion feature prevent automated pipelines or manual tasks from accidentally acting on the same resources.

Traffic guards
: Traffic guards ensure there is always at least one active server group. If someone tries to disable or destroy the last active server group, the traffic guard will activate and protect the server group. Instead of allowing the action to occur, Spinnaker will err on the side of safety and fail the action instead of disabling or destroying the server group, potentially causing dangerous downtime.

Deployment windows
: This is the ability to control the time of day and day of the week during which a deployment can take place (Figure 7-2). There are times of peak traffic—for example, when people are watching Netflix at home after work—when it may not make sense to push out new code. Deployment windows allow pipelines to ensure deployments happen outside of these times of peak traffic so if any errors do happen, they impact the fewest people possible. You can also use this feature to deploy only during business hours.

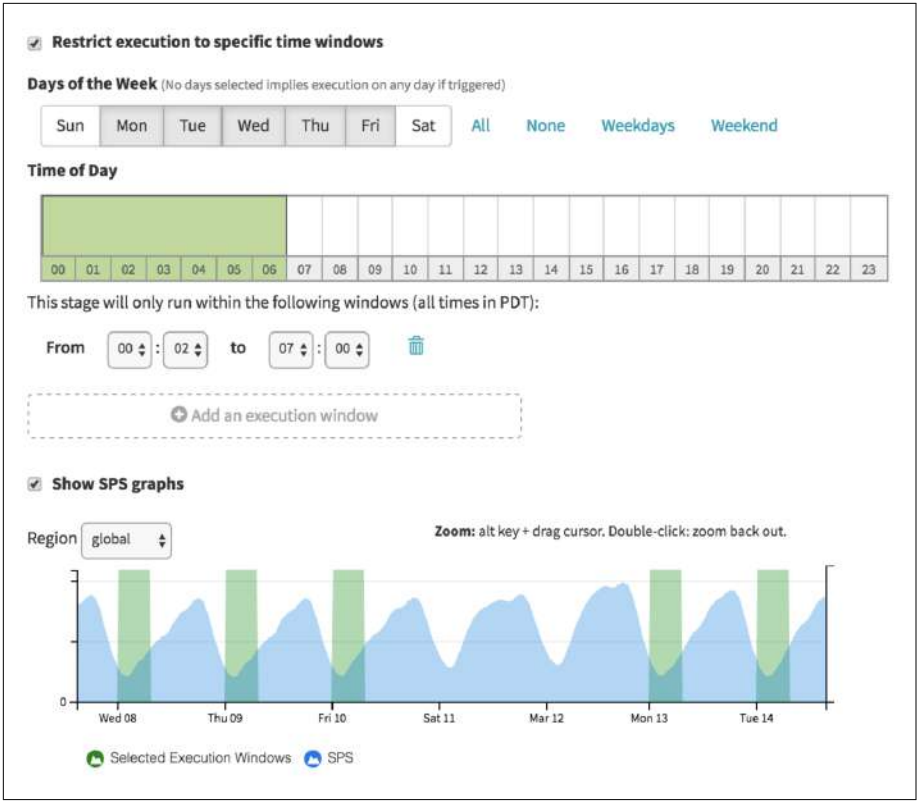

Figure 7-2. An example of a deployment window configuration in Spinnaker with corresponding traffic overlay.

Pipeline Executions

The following set of safeguards are attached to the execution of pipelines and the control flow of artifacts and behavior. Some of these safeguards were mentioned previously in Chapter 4.

Pipeline concurrency
: Having multiple executions of the same pipeline at the same time can have unintended consequences as they try to modify the same set of clusters or run the same set of downstream tests or scripts. By default, new executions will wait for the existing executions to finish before starting. This, of course, can be customized.

Locking pipelines
: Locking a pipeline prevents edits to its configuration through the Spinnaker UI. This is super useful when the pipeline itself is being managed and generated by an external system programmatically.

Disabling pipelines

You can also disable a pipeline when it is undergoing temporary maintenance, when it is unsafe to push, or because the pipeline has been decommissioned. Disabling a pipeline will also turn off all the automated triggers.

Manual judgment

This functionality will interrupt the pipeline execution and ask the user if they want to continue the execution or cancel it. It also gives an operator the ability to make a choice, for example, to run a rollback branch or run additional tests. This stage is particularly useful when there is a human process involved in the pipeline; for example, a QA person checking results from additional systems (Figure 7-3).

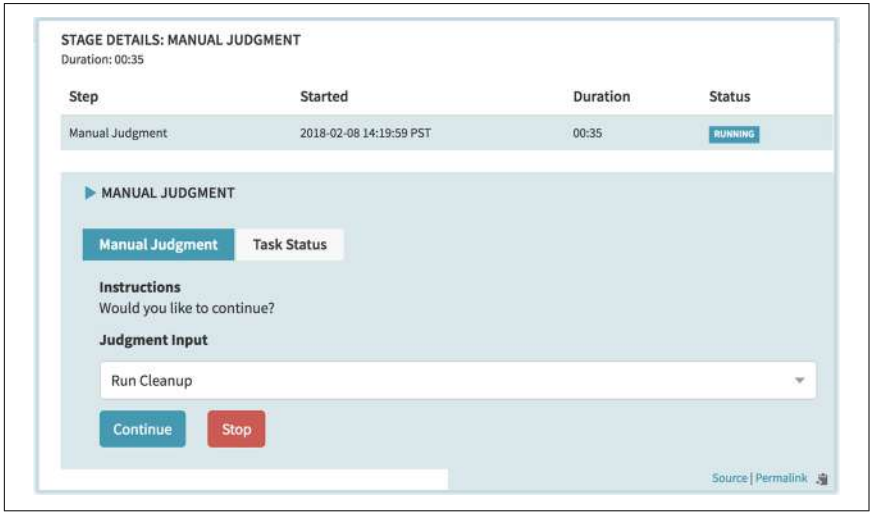

Figure 7-3. An example of a manual judgment in action. This is a detailed view of the Spinnaker Manual Judgment stage, which allows you to provide text and options around the choice you're making.

Conditional stage execution

In some cases, additional tasks need to be executed depending on the context. For example, a Jenkins job might need to be run to update a system of record only when a pipeline is triggered by a git push to the master branch. Spinnaker has a "Conditional on Expression" field in every stage that allows you to turn a stage on or off. This functionality can be used to improve safety by running additional checks or post-processing depending on the trigger or the results of intermediate stages.

Authorization propagation

For security reasons, it may be necessary to restrict deployments in sensitive accounts to only a subset of users. You might not want to let everyone deploy

to accounts containing sensitive billing or user data. In Spinnaker, you can set up accounts that only allow deployments that have been approved by a user with the right permissions. Pipelines for these type of accounts can still be automatically triggered, but they will alert the team at the point of approval, and the person with the right permission can then decide if they want it to move forward or cancel the process.

Rollback pipelines
: Pipelines can be configured to be triggered by the failure of another pipeline. In this scenario, a rollback pipeline might run additional tasks to restore the infrastructure to an expected state or do additional cleanup.

Tag image/Find image from tag
: This technique ties into the practice of immutable infrastructure. As an image (e.g., AWS AMI) is tested, promoted, and validated across different pipelines and environments, the image is tagged with a seal of approval in the form of a tag. Subsequent pipelines can then just search for the last image that passed this approval and continues the validation and promotion cycle.

Automated Validation Stages

Spinnaker also has dedicated pipeline stages to ensure that the deployed resources and infrastructure meet predefined criteria before going further.

Conditional checks
: There is a Check Preconditions stage that will ensure that downstream stages only run if a condition has been met (e.g., don't run teardown tasks if there is only one server group left).

Automated canary analysis (ACA)
: ACA is a technique to minimize the risk of deploying a new version of software into production by comparing metrics emitted from the new version with the version it intends to replace. This feature will be discussed further in the next chapter.

Chaos engineering experiments
: The idea of chaos experiments is discussed in depth in *Chaos Engineering: Building Confidence in System Behavior Through Experiments* (O'Reilly).[1] In Spinnaker, we integrate with Netflix's internal Chaos Automation Platform (ChAP) and allow experiments to run as part of pipelines.

1 *http://www.oreilly.com/webops-perf/free/chaos-engineering.csp*

Chaos Monkey

Spinnaker also has first-class integration with Netflix's Chaos Monkey,[2] which randomly terminates virtual machine instances and containers that run inside of your production environment. Exposing engineers to failures more frequently incentivizes them to build resilient services.

Auditing and Traceability

As continuous delivery pipelines can sometimes involve long-running processes, a key step for making deployments safer is the ability to observe and be notified of the change.

Notifications

Notifications allow pipelines to quickly alert users when errors occur. Spinnaker allows users to configure pipelines and applications to send emails, SMS, or Slack messages at both stage and pipeline levels (Figure 7-4). This increases productivity as developers don't need to constantly refresh their screens or wait for pipelines to complete.

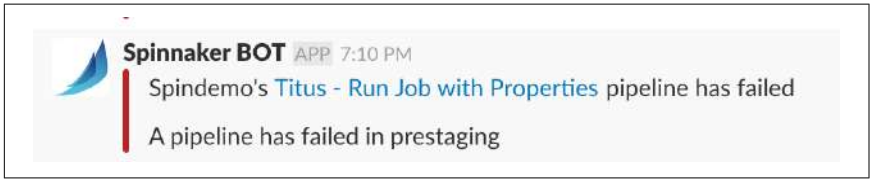

Figure 7-4. Failed pipeline notification in Spinnaker on Slack channel.

Event stream

Every event performed by the orchestration engine is logged separately from operational metrics used to power dashboards. This event stream allows for debugging and analytics. It also allows for later auditing for compliance reasons. At Netflix, the events are put into a Big Data system for long-term storage and querying.

Pipeline history

Pipelines retain an automated versioning history and capture when modifications were made and who made them. The system also allows you to restore a pipeline back to a particular version and compare changes between different pipeline versions. By making it easy and safe to revert pipeline changes, users don't need to worry about making changes.

2 *https://github.com/Netflix/chaosmonkey*

Source of server groups
: The provenance of every server group is recorded in a searchable tag system. The details view of a server group shows whether it was created manually or automatically, with a link to the task or pipeline that created it. This becomes useful when trying to debug issues days after the actual deployment pipeline or operation has ended.

Summary

When dealing with a large number of server groups and instances, it is common to prefer deployments that sacrifice speed for safety. In this chapter, we've discussed techniques available in Spinnaker to ensure safety across cluster deployments and pipeline executions.

We can take advantage of automated validation techniques and testing in production mechanics such as automated canary analysis to further ensure the correctness of the deployments, gating changes to ensure incorrect software is never deployed.

CHAPTER 8

Automated Canary Analysis

Automated canary analysis (ACA) is an example of an advanced automated testing technique available as part of the continuous deployment process. It is included in this report as an example of how many different elements of the continuous deployment puzzle—automation, insights, metrics—can be combined to validate changes in a production environment.

In this chapter, we'll describe how Spinnaker enables ACA. You'll learn in detail about how canaries are set up and supported within a continuous deployment cycle and gain a deeper understanding of how to take advantage of this technique.

Canary Release

A canary release is a technique to reduce the risk from deploying a new version of software into production. A new version of the software, referred to as the canary, is deployed to a small subset of users alongside the stable running version. Traffic is split between these two versions such that a portion of incoming requests is diverted to the canary. This approach can quickly uncover any problems with the new version without impacting the majority of users.

The quality of the canary version is assessed by comparing key metrics that describe the behavior of the old and new versions. If there is a significant degradation in these metrics, the canary is aborted and all of the traffic is routed to the stable version in an effort to minimize the impact of unexpected behavior.

A canary release should not be used to replace testing methodologies such as unit or integration tests. The purpose of a canary is to minimize the risk of unexpected behavior that may occur under operational load.

At Netflix, we augment the standard canary release process and use three different clusters:

- **The production cluster.** This cluster is unchanged and is the version of the software that is currently running. It may run any number of instances.
- **The baseline cluster.** This cluster runs the same version of code as the production cluster. Typically, three instances are created.
- **The canary cluster.** This cluster runs the proposed code changes. As in the baseline cluster, three instances are typical.

The production cluster receives the majority of traffic, while the baseline and canary clusters each receive a small amount. While it's possible to use the existing production cluster rather than creating a separate baseline cluster, comparing a newly created canary cluster to a long-lived production cluster could produce unreliable results. Creating a brand new baseline cluster ensures that the metrics produced are free of any effects caused by long-running processes.

Canary Analysis

Once a canary has been released, a decision on how to proceed needs to be made. This is often performed in a manual, ad hoc manner. For example, a team member may manually inspect logs and graphs. However, this approach requires subjective assessment and is prone to human bias and errors. In addition, manual inspection can't keep up with the speed and shorter delivery time frame of continuous delivery.

To address these issues, a more rigorous and automated approach needs to be taken. This is where ACA can help.

In this approach, key metrics are collected from both the baseline and canary cluster. These metrics are typically stored in a time-series database with a set of tags or annotations that identify if the data was collected from the baseline or the canary. These metrics are then used to determine if there is a significant difference between the canary and baseline. Based on these results, a score can be computed to represent how similar the canary is to the baseline. A decision can then be made using this score to deploy the canary version to production.

Note that a decision can also be made on any of the individual metric results. For example, the canary release could be marked as a failure if any of the individual metrics showed a significant difference between the canary and baseline clusters.

Using ACA in Spinnaker

The Canary stage in Spinnaker can be used to perform ACA. This stage does not perform any provisioning or cleanup operations for you; those must be configured elsewhere in your pipeline. The Canary stage is responsible for running one or more iterations of the canary analysis step. The results of the analysis can be used to make a decision as to whether to continue the canary, roll back, or, in some cases, prompt manual intervention to proceed.

Typically, users will set up a Canary stage before a deployment to production. If the final canary score is below an acceptable threshold, the pipeline will abort and the deployment to production will not continue.

Setting Up the Canary Stage

Once the infrastructure has been configured for the canary deployment, the Canary stage can be added to the pipeline as shown in Figure 8-1.

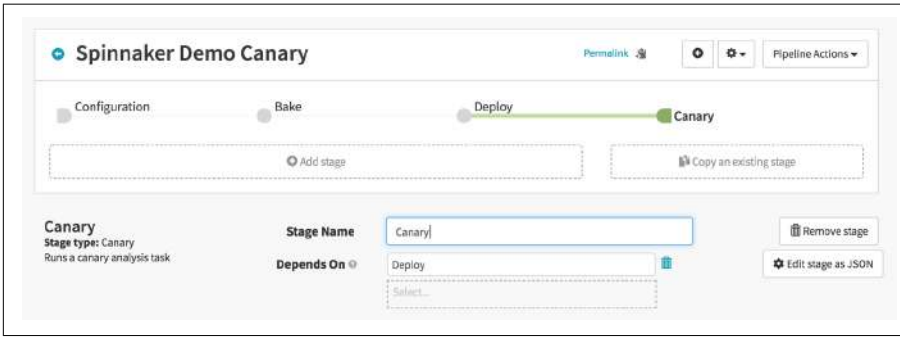

Figure 8-1. Example Spinnaker pipeline with Canary stage.

The Canary stage has a number of properties that control the behavior of the analysis as shown in Figure 8-2. For example, Config Name refers to a configuration file that defines the set of metrics to evaluate; the Interval defines how frequently to retrieve these metrics and run the analysis.

In addition to the analysis parameters, the Canary stage also defines the score thresholds, which are used to determine if a canary should pass or fail.

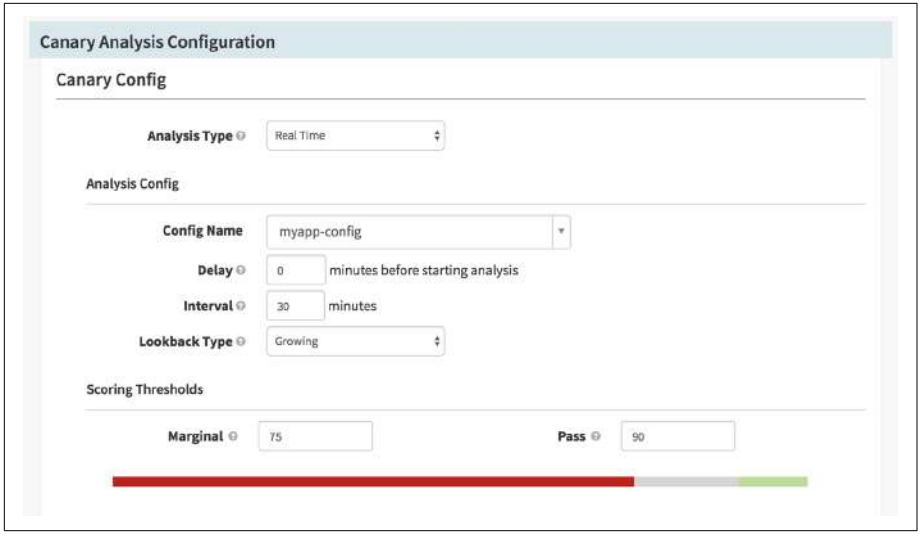

Figure 8-2. Spinnaker canary analysis configuration (ACA).

Reporting

The results of the Canary stage are displayed within the pipeline execution details as shown in Figure 8-3.

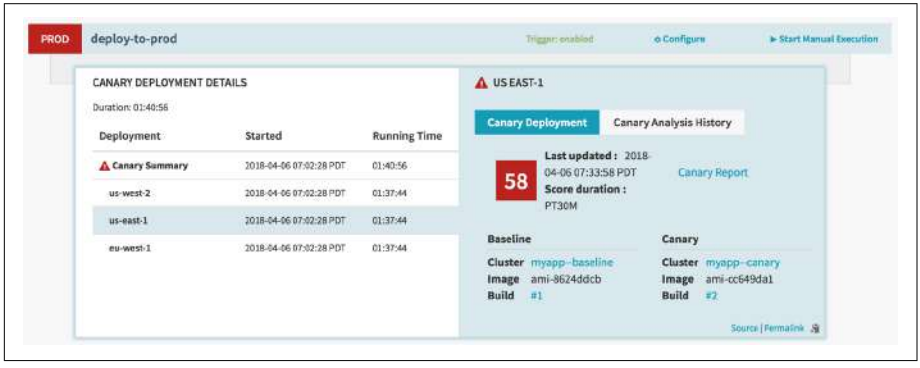

Figure 8-3. Spinnaker canary pipeline execution details.

You can drill down into the details of a canary result and view them in various ways using the Canary Report. The report gives a breakdown of the results by metric and displays the input data used for evaluation.

For example, the report in Figure 8-4 shows a canary score of 58%. A number of metrics were classified as "High" resulting in a lower score. By selecting a specific metric, you can get a view of the input data used for evaluation.

Figure 8-4. Example Canary Report.

Summary

Continuous delivery at scale necessitates having an ability to release software changes at high velocity while ensuring deployments rolled out to production are not just faster but safe as well. Automated canary analysis makes rolling out production deployments safer by reducing manual and ad hoc analysis; only the most stable releases are deployed to production.

CHAPTER 9

Declarative Continuous Delivery

Most of the topics in this book have been centered around an imperative methodology of continuous delivery: telling the system the steps to go through to reach a desired state. Declarative is another popular and powerful delivery methodology where the end state is described and the delivery tooling determines the steps to get there.

In this chapter, you'll be introduced to the pros and cons of the declarative delivery methodology, why teams are interested in its adoption, and the competitive advantage it provides for your projects, as well as declarative capabilities that will be offered through Spinnaker. Note that the declarative effort as of this writing is in development and not generally available.

Imperative Versus Declarative Methodologies

Both imperative and declarative methodologies have their own advantages and disadvantages that should be considered based on your organization.

An imperative world has a shallow learning curve and you're capable of iterating on a delivery pipeline that fits your workflow quickly. Unfortunately, this artisanal flexibility tends to break down through time and scale: as more projects and people are added, things will slowly begin to diverge and some delivery pipelines can stagnate behind the cutting-edge organizational practices. To add insult to injury, when an imperative workflow does something incorrectly, cleanup and failure recovery is often manual or imperatively defined as well, which can quickly become unwieldy.

Declarative, on the other hand, has a steeper learning curve but can scale much better as an organization grows: changes can be applied across an entire infrastructure more easily, and abstractions can be introduced transparently to make more intelligent decisions on behalf of engineering organizations. Since the

desired end state is its primary domain, reasoning about a change's happy path is greatly simplified.

Existing Declarative Systems

The devops ecosystem is dominated by the declarative tools, oftentimes referred to as infrastructure as code. Chances are, if you've been working in the delivery space for long, you're already familiar with some of the more recent, popular ones:

Ansible
: An agentless configuration management system, where you declaratively define your system configuration through YAML playbooks comprised of composable tasks and roles.

Terraform
: An infrastructure as code tool from Hashicorp that supports a wide variety of providers and a powerful plug-in system. Highly opinionated, which makes it easier to pick up and reason about than many of its predecessors.

Kubernetes
: A mixed methodology system, supporting imperative management of single resources, and declarative management of many resources through its manifest files.

These systems all have different delivery targets and were largely developed at different times, but because of the attributes of a declarative model are afforded one killer feature over many imperative systems: planning capability.

This capability isn't exclusive to declarative systems but since these systems work in desired end states, we're more capable of discerning what steps will take place if a new desired state is applied. In times of peril, having a clear vision of what will change can be a key informant in avoiding actions that may cause downtime.

In the Spinnaker ecosystem, there's already been prior work done on the declarative front, namely in GoGo's Foremast[1] project, as well as Spinnaker's Managed Pipeline Templates.

Demand for Declarative at Netflix

We've already touched on a couple reasons an organization may want to choose a declarative methodology: it's easier to manage at scale, and state changes can be reviewed ahead of their application. Aside from these, why would Netflix be

1 *https://github.com/gogoair/foremast*

investing heavily into a declarative methodology, when Spinnaker's imperative model has served us so well?

One of the objectives in a declaratively enabled Spinnaker is to reduce automation of Spinnaker itself. Over the years of Spinnaker's life at Netflix, there have been multiple, redundant efforts of building tooling to make getting started with Spinnaker faster, or to keep people in sync with evolving delivery best practices.

Some power users are responsible for dozens of applications, an aggregate of hundreds of pipelines, and thousands of servers. The imperative nature of Spinnaker works well for these teams, but maintenance may be someone's full-time job. Being able to declaratively define resources—and apply them widely—will reduce the amount of time people spend in Spinnaker, and allow more time for building and delivering direct business value.

These power users also tend to establish best practices that other teams want to adopt: the power users have already gone through the pain of operating resilient systems in production and codified their lessons into Spinnaker's pipelines. In most cases, this means that users need to copy/paste Spinnaker pipelines and slowly diverge from power-user best practices over time. A declarative management model can make it easier for users to templatize their best practices and allow teams to opt into and stay in sync with paved road best practices.

Often times after a delivery-induced incident, a team will update their pipeline to address some new failure, or we'll add guard rails to help protect users from downtime. A natural progression of thought is usually: if we're already telling Spinnaker what we want our end state to be, why can't Spinnaker just decide how to deliver code?

Intelligent Infrastructure

Consider a scenario where an engineer wants to offer an API for other applications to consume. In an effort to ship it, they set up security group rules with an ingress of *0.0.0.0/0* (allow all the things!). A concerning moment for security-minded engineers.

It's hard to expect all engineers to be security experts, so it's understandable why someone would set up security rules that are irresponsibly lenient. What if there were an abstraction available to declare the applications and clusters your app needs to talk to and let the system handle the specifics to make your desired topology reality?

This is an active effort within Netflix through Declarative Spinnaker, deferring security logic to our Cloud Security team. The obvious gains here are that teams get least-privilege security for free, but it also opens up the opportunity for networking, security, and capacity engineers to change cloud topologies, move

applications around, and iterate best practices with less (or no) cross-team synchronization.

Let's say we have an application named `bojackreference` and it needs to talk to the service `businessfactory` via Netflix-flavored gRPC (which allows us to make assumptions about ports, and so on). Such an intent could be expressed through an `ApplicationDependency` intent, which Spinnaker can send to the `Authorizer` application to inform Spinnaker what security rules need to be applied to the infrastructure to make such a link possible:

```
kind: ApplicationDependency
schema: "1"
spec:
  application: bojackreference
  dependencies:
  - kind: Application
    spec:
      application: businessfactory
      protocol:
        kind: NetflixGrpc
```

`Authorizer` would then tell Spinnaker to converge a security group.

The `businessfactory` security group must allow ingress from `bojackreference` on TCP:433 and TCP:9000. Or is it the `bojackreference` security group opens egress to `businessfactory` on TCP:443 and TCP:9000 and ensures `businessfactory` has TCP:443 and TCP:9000 open?

Or some other strategy? A service engineer shouldn't need to stay up to date with the latest, and security and networking engineers shouldn't need to cat-herd application teams: the `Authorizer` application can change its logic to migrate networking as Cloud Security best practices evolve over time transparently.

As engineering organizations grow, more teams will emerge that don't care about all the knobs and just want to deploy into production following established best practices. These teams just want to provide their application artifacts and define some dimensions of their service and have things *just work*. Teams want Spinnaker to be able to take these dimensions and make intelligent decisions on where to deploy, what cloud provider to deploy to, and when to safely deploy, all while maintaining the desired performance and cost efficiency requirements.

In a scenario where Spinnaker is making decisions for them, the desired state should be continuously maintained in the face of unintentional changes. Spinnaker will soon have the capability, at the discretion of application owners, of maintaining desired state and performance characteristics of applications even after delivering software to its target environments.

Of course, some users will want to continue to have all the knobs available to them, so this magic is optional. In order to achieve this, declarative and impera-

tive must be able to coexist side by side, and users must be given the tools to migrate between the methodologies without downtime.

It's important to understand that through declarative, Spinnaker is not looking to subvert or become devops for people. When Spinnaker makes decisions on behalf of users, they're already aware Spinnaker is configured to perform these decisions and what those decisions mean. At any point, users must have the power to suspend Spinnaker's automation should they disagree with the choices it makes. This is an important feature in building intelligent autonomous systems: the need to break the glass is inevitable, and should always be available and easy to actuate.

Summary

While we've painted a picture of what a declaratively powered Spinnaker could be, such a system is still under active development and iteration. It won't solve all problems, but it can offer powerful solutions to high-scale organizations if you want it. Just as a pipeline will work for one team and not another, imperative workflows may be a great fit over a declarative solution for some organizations.

CHAPTER 10

Extending Spinnaker

The previous sections of this report have covered built-in or planned functionality for Spinnaker. However, Spinnaker enforces a particular paved path that doesn't represent every use case.

There are four main ways to customize Spinnaker for your organization: API usage, building UI integrations, writing custom stages, and bringing your own internal logic. This chapter will dive into those scenarios with an example of each. At the end of this chapter, you should have a good understanding about how to customize Spinnaker beyond the out-of-the-box deployment experience provided.

API Usage

The first way to customize Spinnaker is by hitting the API directly. Teams at Netflix use the Spinnaker API for a variety of reasons.

Some teams want to create security groups and load balancers programmatically as part of spinning up full deployment stacks for services like Cassandra, which isn't a supported flow via the UI. Teams use scripts to create this infrastructure.

Another popular API use case is managing workloads that don't fit Spinnaker's deployment paradigm. Teams may have existing orchestrations but use Spinnaker to do the actual creation and destroying of infrastructure. Scripts are used to orchestrate deployment of multiple applications or services that depend on each other and have a more complex deployment workflow.

A third group of teams use the Spinnaker API to build their own specialized platform UI. This helps them surface only the information their team needs and reduces the cognitive load on their engineers.

UI Integrations

At Netflix, we customize the UI to show relevant information. One of the most useful customizations we've put in place is providing a shortcut to copy the ssh command to each running instance. We surface this in the instance details section as a button that copies the unique ssh command (Figure 10-1).

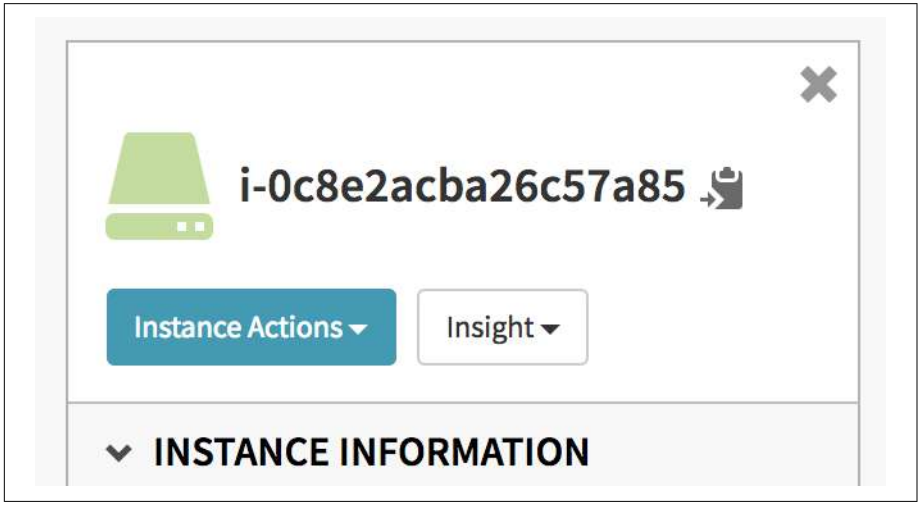

Figure 10-1. The copy ssh command button shows up next to the instance ID. This view shows up when you click a particular instance to show the details about that instance (righthand panel).

We also have an integration with PagerDuty, and add a link in each application to page the application owner (Figure 10-2).

Figure 10-2. The button to page an application owner. This integration appears on the top-right corner of the screen, next to the config tab for the application.

These are just two examples of how you can customize the UI of Spinnaker to fit the needs of your organization.

Custom Stages

Spinnaker allows you to call out to custom business processes via the Webhook stage or the Run Job stage. For more complex integrations you can build a custom stage.

Custom stages allow you to build your own UI to surface specific data. They ensure that interactions with the business process are well defined and repeatable.

Netflix has several custom stages built to integrate with internal tools. Teams have created a ChAP stage to integrate with our Chaos testing platform. Teams have also created a stage to do squeeze testing.

Internal Extensions

Spinnaker is built wholly on Spring Boot and is written to be extended with company-specific logic that isn't necessarily open sourced. By packaging your own JARs inside specific Spinnaker services, you'll be able to tune Spinnaker's internal behavior to match the unique needs of your organization.

For example, Netflix uses two cloud providers besides AWS: Titus, our container cloud (recently open sourced), and Open Connect (our content delivery network, internal only). Supporting these cloud providers requires additional custom logic in the frontend and backend services that make up Spinnaker—Deck, Clouddriver, Orca, and Gate. Furthermore, we've extended Clouddriver in the past to support early access to AWS features.

Unsurprisingly, these private cloud providers are no different than the ones that are open sourced. Any new or existing cloud provider can very well be considered an extension.

All of our services have a common platform of extensions to integrate with the "Netflix Platform," such as using an internal service called Metatron for API authentication, AWS user data signing and secrets encryption, as well as building in specific hookpoints calling out to AWS Lambda functions that are owned by our Security team.

Enumerating all of the ways we've added little bits of extension into Spinnaker would be a report in and of itself.

Summary

Spinnaker can be extended and used in multiple ways. This gives tremendous flexibility to create a system that works for your deployment.

Extensibility of a continuous delivery platform is crucial; an inflexible system won't be able to gain critical adoption. While a system like Spinnaker can address the 90% use case of Netflix, there will always be edge cases that won't be natively supported.

As Spinnaker exists today, extensions are a power-user feature: for frontend it requires TypeScript and React experience, and the backend services require JVM experience. As crucial as extensibility is, we will be continuing to focus on making pluggability and extensibility more approachable.

CHAPTER 11

Adopting Spinnaker

A question we are often asked by individuals and companies after an initial evaluation of Spinnaker is how they can effectively onboard engineering teams, especially when doing so involves reevaluating established processes for software delivery.

Over the past four years, Spinnaker adoption within Netflix has gone from zero to deploying over 95% of all cloud-based infrastructure, but that success was by no means a given. A core tenet of the Netflix culture is that teams and individuals are free to solve problems and innovate as they see fit. You can thus think of Netflix Engineering as a vast collection of small startups. Each team is responsible for the full operational lifecycle of the services they develop, which includes selecting the tools they adopt and their release cadence. We couldn't just dictate that teams had to abandon their existing deployment tooling and replace it with Spinnaker. We had to make Spinnaker irresistible.

Sharing a Continuous Delivery Platform

Here are some key features of Spinnaker that helped convince teams to try out and ultimately adopt Spinnaker:

Make best practices easy to use
: Automated canary analysis drove many teams to evaluate and ultimately adopt Spinnaker. Prior to Spinnaker, teams came up with their own methods for leveraging the canary engine, and they were responsible for every step of the process: launching clusters, running the analysis, evaluating metrics, go/no go, tearing down clusters. Spinnaker democratized automated canary analysis by making it easy to use. Teams could iterate quicker and with a higher degree of safety, without spending time dealing with infrastructure setup and teardown. By leveraging centralized platforms like Spinnaker,

complex best practices can be easily adopted and shared across the entire company.

Secure by default
: By using a centralized tool for continuous delivery, we can enforce and automatically apply good defaults. At Netflix, Spinnaker automatically enforces default security groups and IAM roles to ensure that all infrastructure launched with Spinnaker adheres to recommendations of the security team. Clusters created by Spinnaker are signed so that they can verify themselves and get credentials from Metatron, Netflix's internal credential management system. Teams get added security by deploying to the cloud with Spinnaker.

Standardize the cloud
: As mentioned in previous chapters, the consistent Netflix cloud model removes the guesswork from creating new versions of software and enforces consistency across multiple cloud providers. By having a consistent cloud, we make it easy to build additional tools that support the cloud landscape. All other tools at Netflix take advantage of this consistent naming convention. By opting into Spinnaker, Netflix teams opt in to better alerting, reporting, and monitoring.

Reuse existing deployments
: The Spinnaker API and tooling ensure that existing deployment pipelines can still be used while taking advantage of the safer, more secure Spinnaker deployment primitives. For teams that already had existing Jenkins workflows, we ensured that they could either plug Spinnaker into their jobs or helped them encapsulate their jobs as stages that are reused within Spinnaker. Using Spinnaker for these teams was not an all-or-nothing decision.

Ongoing support
: When we first started Spinnaker, we held weekly office hours and hand-held teams into migrating their existing deployments into Spinnaker. While a team might deploy only a few times a day, the aggregate knowledge of tens of teams deploying a few times a day helps build more robust and dependable systems. By having a centralized team that monitors all AWS and container deployments, we can quickly react to regional issues and help teams move to the cloud quicker.

Having a centralized team responsible for infrastructure deployments also reduces the support burden of other centralized teams. Sister teams that provide database or security services at Netflix often create guides focused on Spinnaker, as this is the preferred deployment tool.

Success Stories

As more teams adopted Spinnaker, they started using it in some ways that we did not predict. Here are a few of our favorite use cases of Spinnaker:

Spot market
: The encoding team at Netflix[1] built automation on top of Spinnaker that borrows from idle reserved EC2 instances and uses them to encode the Netflix catalog. As Spinnaker has real-time data of available idle instances, it becomes the perfect tool to build systems that optimize EC2 instance usage.

Container usage and adoption
: When the Titus team[2] started building their container scheduling engine, they delegated the orchestration of rolling updates and other CI/CD features to Spinnaker rather than implementing their own. This made deploying containers look and feel the same as deploying AWS images, speeding adoption.

Data pipeline automation
: Netflix's Keystone SPaaS[3] is a real-time stream processing as a service platform that leverages the automation offered by the Spinnaker API. Users have access to a point-and-click interface to create a stream of data, filter it, and post the results in sinks like elastic search. All the infrastructure setup and teardown is managed via Spinnaker invisible from the users of the stream.

Multi-Cloud Deployment
: Waze uses Spinnaker[4] to leverage their deployments in Google Compute Platform and Amazon Web Services. They take advantage of the fact that Spinnaker simplifies and abstracts away a lot of the details of each cloud platform. By deploying to two cloud providers, they get added resilience and reliability.

Additional Resources

If you would like to learn more about Spinnaker, check out the following resources:

- Spinnaker website
  - Getting started guides
  - Installing Spinnaker with Haylard

1 *https://medium.com/netflix-techblog/creating-your-own-ec2-spot-market-part-2-106e53be9ed7*

2 *https://medium.com/netflix-techblog/the-evolution-of-container-usage-at-netflix-3abfc096781b*

3 *https://www.youtube.com/watch?v=p8qSWE_nAAE*

4 *http://www.googblogs.com/guest-post-multi-cloud-continuous-delivery-using-spinnaker-at-waze/*

- Blog
- Slack channel
- Community forums
- Spinnaker on the Netflix Tech Blog
- Spinnaker on the Google Cloud Platform Blog

Summary

In this chapter, we shared some of the benefits of centralizing continuous delivery via a platform like Spinnaker. With Spinnaker, teams get access to best practices and a secure and consistent cloud that is well supported and always improving. They also unlock a passionate open source community dedicated to making deployment pain go away.

Continuous delivery is always evolving. New concepts, ideas, and practices are always emerging to make systems more robust, resilient, and available. Tools like Spinnaker help us quickly adopt practices that encourage productivity, safety, and joy.

About the Authors

Emily Burns is a Senior Software Engineer in the Delivery Engineering team at Netflix. She is passionate about building software that makes it easier for people to do their job.

Asher Feldman is a Senior Software Engineer in the Delivery Engineering team at Netflix. He is passionate about automation at scale and leads the effort to integrate Netflix's Open Connect CDN infrastructure with Spinnaker.

Rob Fletcher is a Senior Software Engineer in the Delivery Engineering team at Netflix. He has spoken at several conferences and is the author of *Spock: Up and Running* (O'Reilly).

Tomas Lin is a Senior Software Engineer in the Delivery Engineering team at Netflix. A founding member of the Spinnaker team, he built the original Jenkins integration and maintains the integration with the Titus container platform.

Justin Reynolds is a Senior Software Engineer in the Delivery Engineering team at Netflix.

Chris Sanden is a Senior Data Scientist in the Cloud Infrastructure Analytics team at Netflix. He is passionate about building data driven products and has contributed to efforts around automated canary analysis (ACA).

Lars Wander is a Software Engineer leading Google's Open Source Spinnaker team. He led the integration between Spinnaker and Kubernetes, and recently led the effort to write Halyard, a tool for configuring, deploying, and upgrading Spinnaker.

Rob Zienert is a Senior Software Engineer in the Delivery Engineering team at Netflix. He has contributed mostly around operations and reliability across the services and is the lead for the declarative effort within Spinnaker.