SCTE CABLE-TEC EXPO 22
SEPTEMBER 19-22 • PHILADELPHIA, PA

2022 Fall Technical Forum
SCTE • CABLELABS • NCTA

Solving Automation and Orchestration for Large-Scale Hybrid Cloud Environments

A Technical Paper prepared for SCTE by

David Grizzanti
Distinguished Engineer
Comcast Cable
1800 Arch St. Philadelphia, PA
215-356-2354
david_grizzanti@comcast.com

Matthew Morrissey
Principal Engineer
Comcast Cable
1800 Arch St. Philadelphia, PA
215-435-7648
matthew_morrissey@comcast.com

Table of Contents

Title **Page Number**

List of Figures

Title **Page Number**

1. Introduction

The idea of using automation to reduce human error and speed up infrastructure deployments is not new, however, choosing the "right" tools can be challenging. Tool selection has, in some ways, become an engineering challenge on its own. Many large cable operators are not all in on a single cloud provider, automation framework, or container orchestration engine.

The authors will review a set of cloud agnostic infrastructure-as-code (IaC) modules, continuous deployment pipelines, and a GitOps/Configuration Management interface that facilitates changes. This framework can deliver an internal observability platform that hosts millions of time-series metrics, including visualization and a unified search interface.

2. Problem Statement

As cable operators expand workloads into public cloud providers, teams will be tasked with providing a solution to time series metrics collection that is cloud agnostic. Many cable operators already use some form of on-premises infrastructure and the expansion to public cloud provides an opportunity to re-think how monitoring infrastructure is built. One primary goal is to give clients a storage and query interface to their metric data, without having to manage the system on their own. This will in turn enable client teams to focus on the things that bring the most value to their customers. This platform must be robust enough to handle outage scenarios such that the clients can access their time series data to perform troubleshooting or triage for their application.

2.1. Operating in a hybrid cloud environment

A best practice for time series platforms is to collect and store samples as close to the source application as possible. Given that this team was building a platform for other teams within the company, they would need to be able to deploy wherever the clients are deployed. This meant that the platform would need to deploy into both public and private clouds. Deploying into a hybrid cloud environment can get complicated quickly if you want to maintain a consistent way to deploy across all clouds. Each cloud has its own APIs and preferred methods for interacting with the infrastructure that you're building. In addition, service offerings such as load balancing, secret management, and auto-scaling are different across each cloud.

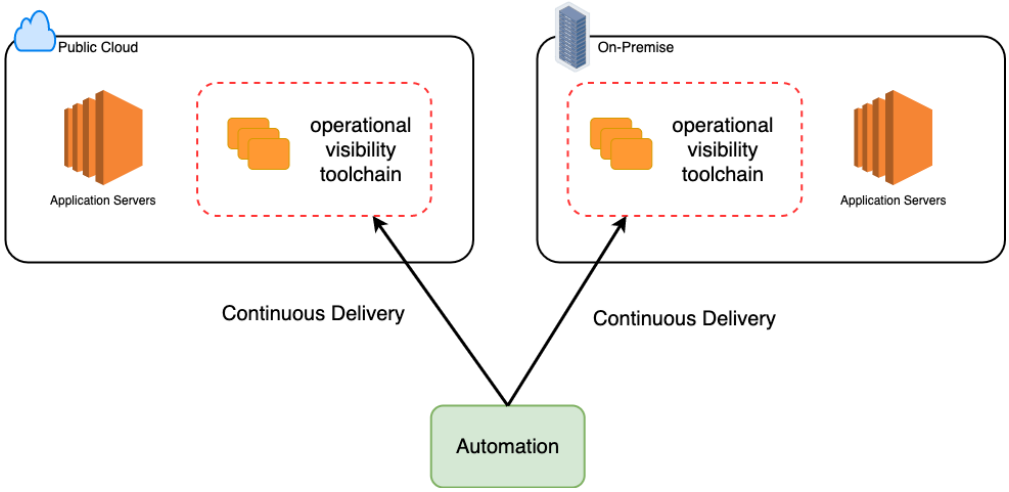

Figure 1 - Observability Toolchain

Choosing a set of abstractions can help with these challenges. The team chose to use Terraform and treat the Terraform configuration just as we would any other piece of code. This practice is commonly referred to as Infrastructure as Code. This enabled them to predictably and repeatably build infrastructure into any cloud that had a Terraform provider. This significantly decreased the complexity of deploying to multiple clouds. Once this pattern was established, they were effectively able to extract the functionality into a set of Terraform modules and fill in provider specific details.

2.2. Simplifying application developer choice

Large companies tend to have a huge breadth of choice in terms of tool sets and platforms application developers can use to deploy and monitor their applications. This can frequently lead to analysis paralysis from a developer's perspective since there can be so many options, some seemingly competing with one another. It can also be difficult as a developer to discover what tools or platforms are available to you since documentation for these tools or platforms tends to be siloed within a given team or organization. From the company's perspective this spread of similar tools and platforms doing the same or similar things, and their potential lack of discoverability, is challenging in terms of budget dollars. There could be lots of engineering hours spent developing a tool set or platform that might already exist. Teams or organizations might purchase a vendor solution for a problem that has already been solved in other parts of the company.

A goal is to avoid these pitfalls, making use of their own platform so easy that it would not even be presented as a choice. Users would be compelled to use the system because there would be little to no friction involved in getting it set up or interacting with it. This meant that not only did they need to spend a significant amount of time up front thinking about the user experience, but they also had to make sure that every decision made would not negatively impact the user experience if it could be avoided. For the team, that meant hiding as much of the complexity as they could from the user. To achieve that goal, they built a web portal for the clients to interact with that would guide them through the process of providing the required information to deploy the infrastructure into their chosen environment. Rather than require users to setup a meeting or fill out a ticket, they were given the ability to go to a web page and onboard in an entirely self-service manner.

2.3. Benefits of time series metrics over logs

Time series data is a collection of numeric measurements, made over time. There are many uses for time series data, plotting points on a graph being one of the most popular. Time series data exists in many domains outside of software systems, stock prices, heartbeats per minute, etc. For a web application, time series data may be requests per second, or for a database this might be number of active connections or active queries. [1]

Historically, applications relied heavily on application logs for observability. Application logs being text-based messages that contain informational events, warnings, and errors. Whether this was for application alerting, creating dashboards, or debugging the system, logs were the standard. The team was tasked with building a time series collection platform to reduce log ingestion and shift teams towards a "metrics first" approach. Most application teams would write verbose logs and extract metrics from those logs in downstream log analytics tools. As a result, there are often high log volumes and large storage costs associated with many applications.

Figure 2 - Time Series Data Graph

"By and large, the biggest advantage of metrics-based monitoring over logs is that unlike log generation and storage, metrics transfer and storage have a constant overhead. Unlike logs, the cost of metrics doesn't increase in lockstep with user traffic or any other system activity that could result in a sharp uptick in data.

With metrics, an increase in traffic to an application will not incur a significant increase in disk utilization, processing complexity, speed of visualization, and operational costs the way logs do. Metrics storage increases with more permutations of label values (e.g., when more hosts or containers are spun up, or when new services get added or when existing services get instrumented more), but client-side aggregation can ensure that metric traffic doesn't increase proportionally with user traffic." [2]

3. Building the platform

At the outset of the project, the team made the choice to build a metrics collection platform, instead of buying an off-the-shelf product. They also decided to avoid so called "cloud native" toolsets that existed within public cloud platforms. Each public cloud offers a suite of observability tools that integrate into their infrastructure services.

A primary goal of the project was to build an easily maintainable and scalable platform to deploy metrics infrastructure for application teams. The target for this infrastructure ranged from on-premises datacenters to public cloud providers, with the initial target being on public clouds. Many vendors offer platforms that provide metrics storage "as a Service", however, the team's research found the cost and compatibility with the various internal applications and tools incompatible. Many public cloud vendors offer their own hosted tools that provide these features; however, they wanted a single solution for all teams regardless of where they chose to host their application.

3.1. Standardized toolset over cloud native

The team chose Prometheus [1] as the time series database, primarily for its wide adoption as an open-source standard for monitoring and alerting. Using Prometheus as the base allowed them to focus their efforts on building out the platform around a common tool versus worrying about how to automate and support each cloud vendors specific product. In addition, this allowed them to offer a single solution regardless of an application teams intended deployment location.

In addition to choosing a standard tool to deploy across clouds, the team also decided to keep their deployment simple by using Virtual Machines (VM) instead of cloud specific orchestration platforms. Since each cloud platform offers its own unique orchestration service and on-premises datacenters are unique, choosing the lowest common denominator simplified the architecture. There were a few drawbacks to this, namely losing key features like auto-scaling, but they were able to work around these with proper alerting and health checks.

Moreover, this enabled the team to build upon a single common image that could be deployed to whichever cloud, be it public or private, with minimal effort to add cloud specific instruction sets for the VM. This gave the benefit of only needing to fetch a single image which has been patched or updated and build on top of it. These changes could then be easily propagated out to the client stacks without needing to remember or think about cloud VM specific patching.

3.2. Choosing the open source path

As mentioned earlier, Prometheus was selected as the base software tool for metrics collection and storage. The next focus was to choose how they wanted to manage and deploy Prometheus plus any customizations and client specific configuration that would be needed. They decided to continue with open-source, modular tools and made the following selections:

- **Terraform**: For Infrastructure as Code automation
- **Packer and Ansible**: Building Base Cloud AMIs/Images
- **Git**: Version control of Terraform modules and client configuration
- **Concourse**: Continuous Integration and Deployment

Using Terraform allowed them to take advantage of existing providers for many public clouds and, on-premises private clouds like OpenStack.

3.3. Centralized versus distributed

Another challenge they faced was whether the collection and storage of application metrics should be centralized versus distributed. More specifically, whether the collection and storage of an application's metrics should stay within the cloud and region where they originate. If they chose to distribute, then querying could be challenging, but if they chose to centralize, they'd have to deal with a centralized cost and chargeback model.

The team chose to keep collection and storage local to the application's virtual networking environment. This had a few advantages, namely:

- Prometheus uses a pull model for metrics collection, so this simplified network and access control as scrapers were local to an applications network.
- Cost was managed by deploying into a client's local network, so no chargeback or show back was necessary
- A network outage would not cause any loss in metrics collection

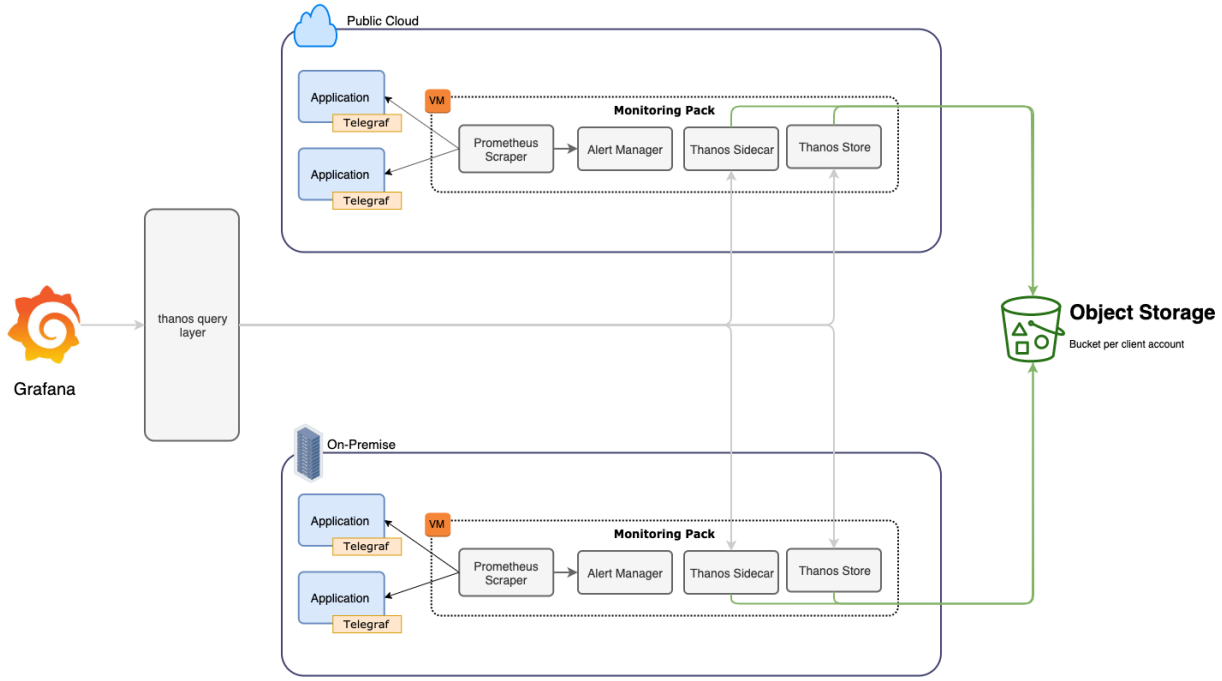

Figure 3 – Centralized Query Layer

A distributed model posed a challenge however for querying all metrics for a given client at once. This challenge was solved by deploying Thanos, an open-source Prometheus compatible tool with global querying capabilities as shown in Figure 3. The Thanos Query component, specifically, enabled deployment of a centralized set of instances that could access all regions, across all clouds and offer a single interface for client queries.

It was also decided to use the selected tool chain to deploy the monitoring as a client of infrastructure. This provided valuable insights into pain points the clients might experience, find places where there were gaps in the automation, and get comfortable with using the tools were being recommended that other teams adopt. This experience created a robust set of documentation, which in turn helped to reduce the support burden on the team.

4. Building modular components

Segmenting each concern into its own module allowed the platform to be deployable to any current or future cloud platform. The platform was broken down into Terraform modules, application container images, VM image builds, and client configuration.

4.1. Terraform Modules

The Terraform modules are primarily responsible for deploying a set of application container images, via docker, onto a virtual machine in the specified cloud environment. The logic for each cloud provider is encapsulated into each module, which includes details on instance types, disk sizes, cloud initialization instructions, etc. The set of container images can be customized per cloud and per client as needed.

4.2. Container Images

The application container images, which includes Prometheus, are the main software tools in the system. They provide the functionality for the metrics collection and storage. It's worth noting however, that the platform is agnostic to the container images deployed. Adding new container images or swapping all of them out for a different set of software tools would be trivial.

4.3. Image Builds

Since each cloud platform requires its own image type, cloud specific logic was extracted from the automation. A set of images were then built to be deployed across all clouds at once, with the same set of requirements driven by Ansible playbooks.

4.4. Client Configuration

Each client of the platform has a set of custom configurations that need to be applied at deployment time. These range from what hosts Prometheus needs to scrape, alerting rules, metric retention time, etc. This configuration is kept in version control (git) and updates to these files drive deployments to the various clouds and regions that are supported.

4.5. Client Onboarding

Clients are provided with a single URL at which they can onboard and manage their stacks. This was done to simplify how a client can start using the stack. The website has links to all the relevant documentation for any pre-requisite steps they might need to perform, as well as a simple form which takes in all the required information for the automation to build the stack for them. With a single click the client has their git directory and file structure created, has the minimal set of configurations for the containers to run, has the pipeline to deploy the stack created, and the stack is deployed. The client is presented with links to all the relevant sites and can immediately configure their stack for their use case. The whole process is entirely automated which allows the platform developers to focus on providing more value to the client and allows the client to more quickly get their stack running and configured since they don't need any touch points with the platform team.

5. Deployment & Continuous Delivery

The final step is integrating all the various components and modules into continuous delivery pipelines, using Concourse. This is where the core intelligence and logic of the platform comes together. At the outset of the project, they built each pipeline manually for each client by stringing together:

- The set of necessary Terraform modules for the target cloud platforms
- The logic for what regions were necessary
- The set of container images that matched the client configuration

This process became very tedious over time, especially when the number of clients grew beyond a handful and the number of regions for each client increased. To address to this, a templating application was created using Golang to automate the creation of each client pipeline.

5.1. Pipeline Generation

Concourse pipelines are written using YAML, which can be very verbose given its declarative nature. To simplify the management of the pipeline YAML files, which could range from 5,000 to 20,000 lines, it was decided to write an application that generate these files, given a defined set of inputs. The inputs included:

- Target cloud platform(s)
- Cloud region(s)
- Container image(s)
- Cloud platform account details

Given this set of inputs, a templated YAML file was created that was fed into the Golang program. When a client onboards or adds additional stacks to their deployment the program will re-generate the pipeline entirely rather than trying to insert just the pieces that change. This is possible because the pipeline generation is deterministic.

5.2. Concourse Interface

The generated YAML is used to create a pipeline within the Concourse platform as shown in Figure 4. This interface allows for manual triggering a region deployment if necessary, viewing logs, and tracking progress of deployments.

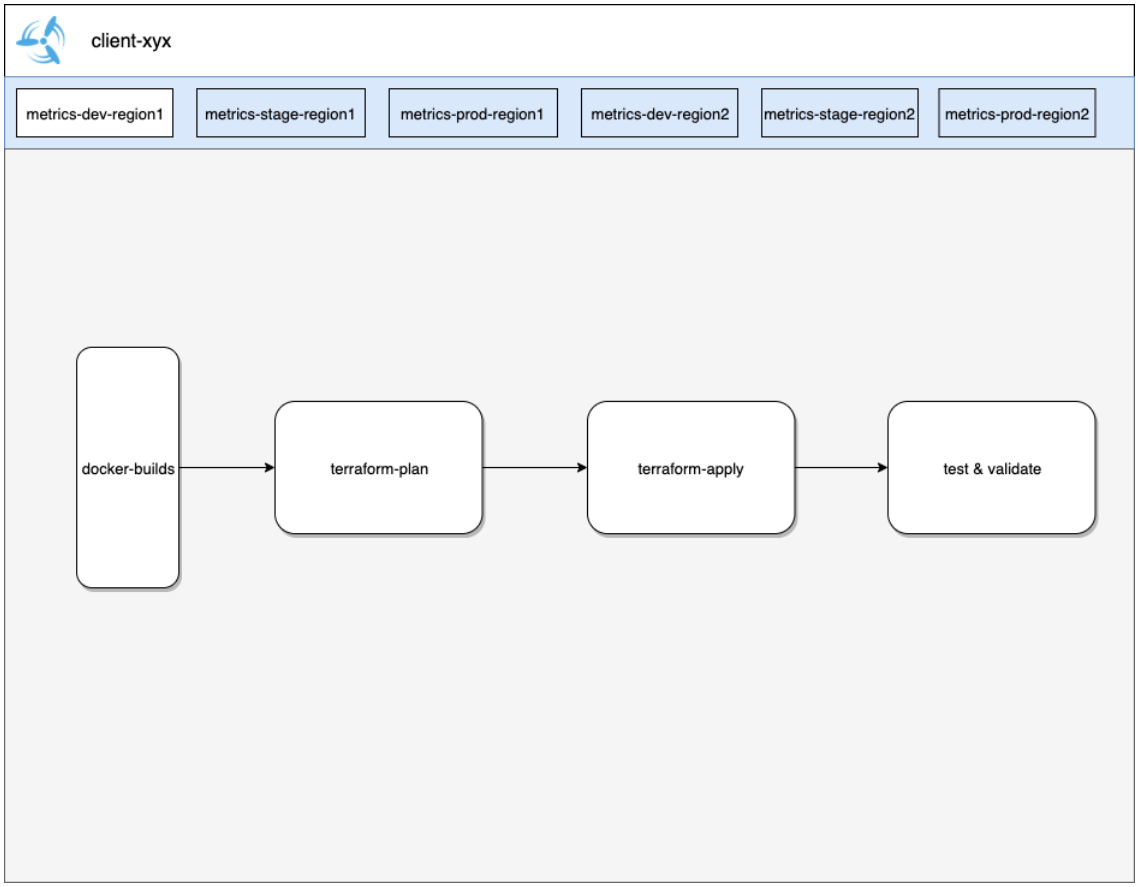

Figure 4 – Concourse User Interface

5.3. Delivery Automation

Once the Concourse pipeline is generated and applied in the Concourse platform, updates made to client configuration in git will trigger a deployment. The trigger is executed via webhooks through the version control system. A change made to a file in git will send a HTTP request to Concourse and kick off a deployment. The set of jobs run for each cloud do not need any manual intervention from the team, unless an error occurs. Any errors will trigger an alert to the team and error logs can viewed within the Concourse user interface (UI).

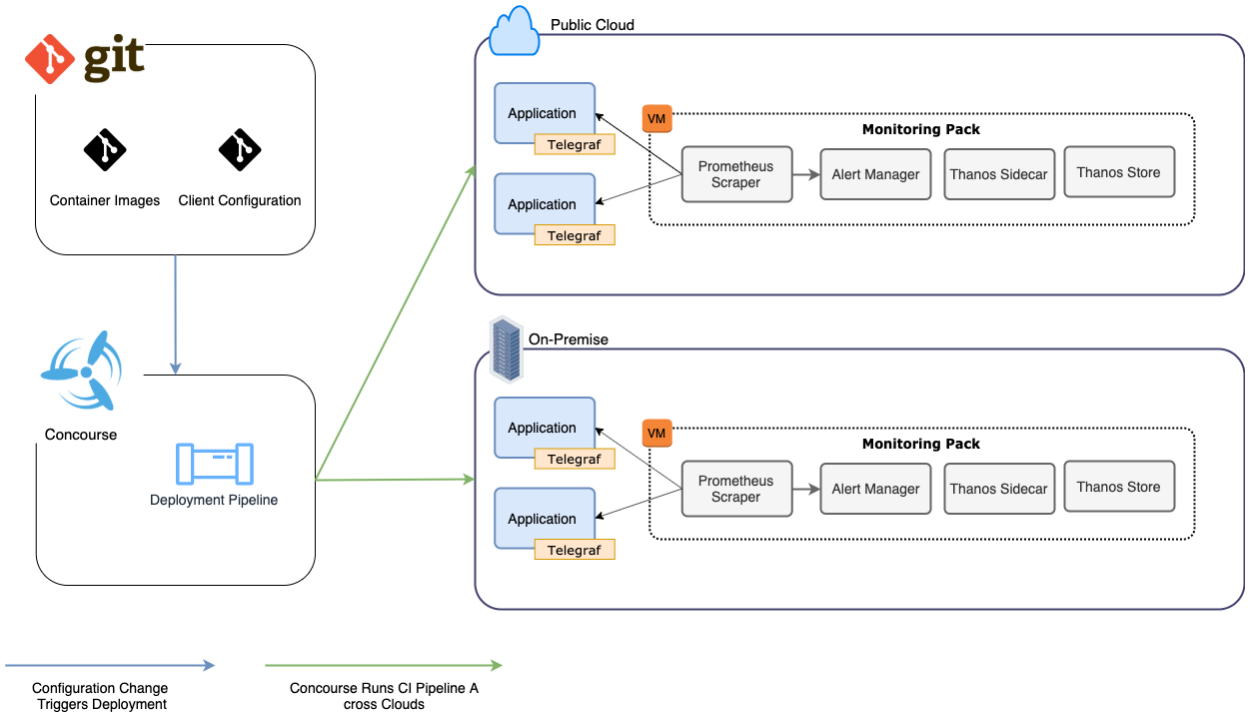

Figure 5 - Deployment Orchestration

6. Conclusion

As cable operators grow their on-premises cloud offerings and expanded into public clouds, there becomes a need for simplified abstractions to deploying infrastructure across environments. This team was tasked with providing this abstraction, specifically for observability tooling, available to all teams. They approached the problem by breaking down the areas into modularized components, capable of abstracting away the complexities of the target cloud environment, application type, and regionality. Focusing on clients' needs helped frame how to abstract the application architecture and functionality. This made the platform easy to use, quick to change and deploy seamlessly. All of this was possible because at each decision point, the software landscape was evaluated. A decision was made to go with proven, open-source tools that would meet the case. This, in turn, allowed the team to rapidly develop and scale the platform to serve any clients who need it, wherever their applications are running.

Abbreviations

API	Application Programming Interface
CI/CD	Continuous Integration and Continuous Delivery
HTTP	Hypertext Transfer Protocol
IaC	Infrastructure as Code
SCTE	Society of Cable Telecommunications Engineers
UI	User Interface
VM	Virtual Machine
YAML	Text based markup language

Bibliography & References

[1] *Prometheus Overview,* Prometheus Authors 2014-202
https://prometheus.io/docs/introduction/overview/

[2] *Distributed Systems Observability,* Cindy Sridharan; O'Reilly Media, Inc.