

Google Anthos IN ACTION

Edited by
Antonio Gulli
Michael Madison
Scott Surovich



MANNING



MEAP Edition
Manning Early Access Program
Google Anthos in Action
Version 6

Copyright 2022 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP edition of *Google Anthos in Action*. I hope that what you'll get access to will be of immediate use to you for developing cloud software and, with your help, the final book will be amazing!

To get the most benefit from this book, you need to have some understanding of cloud computing. Don't worry too much because we will review basic concepts such as microservice and service meshes during the initial chapters of the book.

Whether you are a software developer, a businessperson interested in technology, a technology researcher, or simply a person who wants to know more about where the world of computers and people is going, it is essential to know more about Anthos. This book is an excellent way to get an understanding not just of a critical technology, but of the trends and thinking that led us to create Anthos. As such, it will also give you a good sense of the future.

When we started to look at Anthos in Google, we were inspired by one single principle: Let's abstract the less-productive majority of the effort in running computing infrastructure operations and remove habitual constraints that are not really core to building and running software services. Kubernetes and the SRE principles for running managed services with DevOps were fundamental concepts used to build the foundations for our new product. Anthos is the additional major step towards making developers' work easier thanks to the creation of a customer-focused, run-anywhere distributed development platform that can automate utilization, can reduce expensive dependencies and facilitate the production state.

This is really a big book, and I hope you find it as useful to read, as the authors, the curators, and the reviewers find it interesting to produce. My suggestion is that you deep dive into the initial five or six initial chapters and then pick the relevant follow up topics in the order you like.

Please make sure to post any questions, comments, or suggestions you have about the book in the [liveBook discussion forum](#). Your feedback is essential in developing the best book possible.

—Antonio Gulli

brief contents

- 1 *Overview of Google Anthos*
- 2 *Cloud is a new computing stack*
- 3 *Anthos, the one single pane-of-glass*
- 4 *Anthos, the computing environment built on Kubernetes*
- 5 *Anthos Service Mesh: security and observability at scale*
- 6 *Operations management in Anthos*
- 7 *Bringing it all together*
- 8 *Hybrid applications in Anthos*
- 9 *Anthos, the compute environment running on VMware*
- 10 *Anthos, at the edge and telco world*
- 11 *Knative serverless extension*
- 12 *Anthos - the networking environment*
- 13 *Anthos Config Management*
- 14 *Anthos, integrations with CI/CD*
- 15 *Anthos, data and analytics*
- 16 *Using Anthos for security and policies*
- 17 *Google Cloud Marketplace*
- 18 *Migrate for Anthos and GKE*
- 19 *Anthos for breaking the monolith*
- 20 *Anthos an end-to-end example of ML application*

21 Anthos on bare metal

22 Anthos on Windows

23 Lessons from the field

1

Overview of Google Anthos

by Aparna Sinha

This chapter includes:

- Anatomy of a Modern Application
- Accelerating Software Development with Anthos
- Standardizing Operations At-Scale with Anthos
- Origins at Google
- How to read this book

1.1 Anatomy of a Modern Application

Software has been eating the world for a while. As consumers, we are used to applications that make it faster, smarter and more efficient for us to do things like calling a cab or depositing a paycheck. Increasingly, our health, education, entertainment, social life and employment are all enhanced by modern software applications. At the other end of those applications is a chain of enterprises, large and small, that deliver these improved experiences, services and products. Modern applications are deployed not just in the hands of consumers, but also at points along this enterprise supply chain. Major transactional systems in many traditional industries such as retail, media, financial services, education, logistics and more are gradually being replaced by modern micro-services that auto-update frequently, scale efficiently and incorporate more real time intelligence. New digital-first startups are using this opportunity to disrupt traditional business models, while enterprise incumbents are rushing to modernize their systems so they can compete and avoid disruption.

This begs the question: What is a Modern Application? When you think of software that has improved your life, perhaps you think of applications that are interactive, fast (low latency), connected, intelligent, context aware, reliable, secure and easy to use on any device. As technology advances, the capabilities of modern applications, such as the level of security, reliability, awareness and intelligence advance as well. For example, new development frameworks such as react and angular have greatly enhanced the level interactivity of applications, new runtimes like node.js have increased functionality. Modern applications have the property of constantly getting better through frequent updates. On the backend **these applications often comprise a number of services that are all continuously improving**. This modularity is attained by breaking the older 'monolith' pattern for writing applications, where all the various functions were tightly coupled to each other.

Applications written as a set of modular or micro-services, have several benefits: constituent services can be evolved independently or replaced with other more scalable or otherwise superior services over time. Also the modern microservices pattern is better at separating concerns and setting contracts between services making it easier to inspect and fix issues. This approach to writing, updating and deploying applications as 'microservices' that can be used together but also updated, scaled and debugged independently is at the heart of modern software development. In this book we refer to this pattern as "Modern" or "Cloud Native" application development. The term Cloud Native applies here because the microservices pattern is well suited to run on distributed infrastructure or Cloud. Microservices can be rolled out incrementally, scaled, revised, replaced, scheduled, rescheduled, and bin packed tightly on distributed servers creating a highly efficient, scalable, reliable system that is responsive and frequently updated.

Modern applications can be written 'greenfield', from scratch, or re-factored from existing 'brownfield' applications by following a set of architectural and operational principles. **The end goal of Application Modernization is typically revenue acceleration, and often this involves teams outside IT, in Line-of-Business units.** IT organizations at most traditional enterprises have historically focused on reducing costs and optimizing operations, while cost reduction and optimized operations can be by-products of application modernization they are not the most important benefit. Of course, the modernization process itself requires up front investment. Anthos is Google Cloud's platform for application modernization in hybrid and multi-cloud environments. It provides the approach and technical foundation needed to attain high ROI application modernization. An IT strategy that emphasizes modularity through APIs, micro-services and cloud portability combined with a developer platform that automates reuse, experiments, and cost efficient scaling along with secure, reliable operations are the basic critical pre-requisites for successful Application Modernization.

The first part of Anthos is a modern developer experience that accelerates Line-of-Business application development. It is optimized for re-factoring brownfield apps, and writing microservices and API based applications. It offers unified local, on-prem and cloud development with event driven automation from source to production. It gives developers the ability to write code rapidly using modern languages and frameworks with local emulation and test, integrated CI/CD, and supports rapid iteration, experimentation and

advanced roll out strategies. The Anthos developer experience emphasizes cloud APIs, containers and functions but is customizable by enterprise platform teams. A key goal of the Anthos developer experience is for teams to release code multiple times a day, thereby enhancing both velocity and reliability. Anthos features built in velocity and ROI metrics to help development teams measure and optimize their performance. Data driven benchmarks are augmented with pre-packaged best practice blueprints that can be deployed by teams to achieve the next level of performance.

The second part of Anthos is an operator experience for central IT. Anthos shines as the uniquely scalable, streamlined way to run operations across multiple clouds. This is enabled by the remarkable foundation of technology invented and honed at Google over a period of twenty years, for running services with extremely high reliability on relatively low cost infrastructure. This is achieved through standardization of the infrastructure using a layer of abstraction comprising Kubernetes, Istio, Knative and several other building blocks along with Anthos specific extensions and integrations for automated configuration, security, and operations. The operator experience on Anthos offers advanced security and policy controls, automated declarative configuration, highly scalable service visualization and operations, and automated resource and cost management. It features extensive automation, measurement and fault avoidance capabilities for high availability, secure service management across cloud, on-prem, edge, virtualized and bare metal infrastructure.

Enterprise and small companies alike find that multi-cloud and edge is their new reality either organically or through acquisitions. Regulations in many countries require proven ability to migrate applications between clouds, and demonstrate failure tolerance with support for sovereignty. Non-regulated companies find multi-cloud necessary for providing developers' choice and access to innovative services. Opportunities for running services and providing greater intelligence at the edge add further surfaces to the infrastructure footprint. Some IT organizations roll their own cross-cloud platform integrations, but this job gets harder every day. It is extremely difficult to build a cross cloud platform in a scalable, maintainable way, but more importantly, it detracts from precious developer time for product innovation.

Anthos provides a solution rooted in years of time-tested experience and technical innovation at Google in SW development and SRE operations, augmented with Google Cloud's experience managing infrastructure for modern applications across millions of enterprise customers. Anthos is unique in serving the needs of LoB developers and Central IT together and with advanced capabilities in both domains. Consistency of developer and operator experience across environments enables enterprises to obtain maximum ROI from Application Modernization with Anthos.

1.1.1 Accelerating Software Development

Software product innovation and new customer experiences are the engine of new revenue generation in the digital economy. But the innovation process is such that only a few ideas lead to successful new products, most fail and disappear. As every industry transitions to being software driven, new product innovation becomes dependent on having a highly agile and productive software development process. Developers are the new Kingmakers. Without an agile, efficient development process and platform, companies can fail to innovate or

innovate at very high costs and even negative ROI. An extensive DevOps Research Assessment¹ study (DORA) surveyed over 30,000 IT professionals over several years across a variety of IT functions. It has shown that excellence in software development is a hallmark of business success. This is not surprising given the importance of modern applications in fueling the economy.

DORA quantifies these benefits, showing that “Elite” or the highest performing software teams are 2X more effective in attaining revenue and business goals than Low performing teams. The distinguishing characteristic of Elite teams is the practice of releasing software frequently. DORA finds that four key metrics provide an accurate measurement of software development excellence. These are:

- Deployment frequency
- Lead time for changes
- Change fail rate
- Time to restore service

High performance teams release software frequently, for example several times a day. In comparison low performers release less than once a month. The study also found that teams that release frequently have a lower software defect ratio and recover from errors more rapidly than others. As a result, in addition to being more innovative and modern, their software is more reliable and secure. Year over year DORA results also show that an increasing number of enterprises are investing in the tools and practices that enable Elite performance.

Why do teams with higher Development Velocity have better business results? In general, higher velocity means that developers are able to experiment more, test more, and so they come up with a better answer in the same amount of time. But there is another reason. Teams with higher velocity have usually made writing and deploying code an automated, low effort process. This has the side effect of enabling more people to become developers, especially those who are more entrenched in the business vs. the tooling. As a result high velocity developer teams have more Line-of-Business thinking and a greater understanding of end user needs on the development team. The combination of rapid experimentation and focus on users yields better business results. Anthos is the common substrate layer that runs across clouds to provide a common developer experience for accelerating application delivery.

1.1.2 Standardizing Operations At-Scale

Developers may be the new Kingmakers, but Operations is the team that runs the kingdom day in and day out. Operations includes teams that provision, upgrade, manage, troubleshoot and scale all aspects of services, infrastructure and cloud. Typically networking, compute, storage, security, identity, asset management, billing and reliability engineering is part of the operations team of an enterprise. Traditional IT teams have anywhere from 15-30% of their staff in IT operations. This team is not always visibly engaged in new product introductions with the line-of-business, but often lays the groundwork, selecting clouds, publishing service catalogs and qualifying services for use by the business. Failing to invest

¹ <https://www.devops-research.com/research.html>

in operations automation often means that operations become the bottleneck and a source of fixed cost.

On the flip side, modernizing operations has a tremendous positive effect on velocity. Modern application development teams are typically supported by a very lean operations team, where 80%+ of staff is employed in software development vs. operations. Such a developer centric ratio is only achieved through modern infrastructure with scaled, automated operations. This means operations are extremely streamlined and leverage extensive automation to bring new services online quickly. Perhaps the greatest value of Anthos is in automating operations at scale consistently across environments. The scale and consistency of Anthos is enabled by a unique open cloud approach that has its origins in Google's own infrastructure underpinning.

1.2 Origins in Google

Google's software development process has been optimized and fine tuned over many years to maximize developer productivity and innovation. This attracts the best software developers in the world and leads to a virtuous cycle of innovation in software and software development and delivery practices. The Anthos development stack has evolved from these foundations and is built on core open source technology that Google introduced to the industry.

At the heart of Anthos is Kubernetes, the extensive orchestration and automation model for **managing infrastructure through the container abstraction** layer. The layer above Kubernetes is grounded in Google's Site Reliability Engineering or Operations practices, which standardize the control, security and management of services at scale. This layer of **Service Management** is rooted in Google's Istio-based Cloud Service Mesh. Enterprise **policy and configuration** automation is built in at this layer using Anthos Configuration Management to provide automation and security at scale. This platform can run on multiple clouds and abstracts the disparate networking, storage and compute layers underneath.

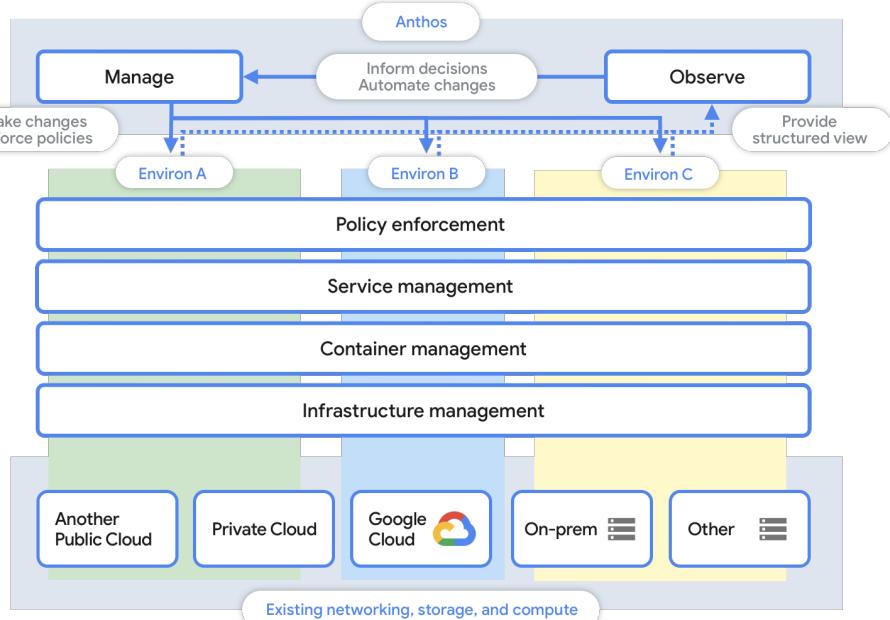


Figure 1.1. Above this Anthos stack is Developer experience and DevOps tooling including a deployment environment that uses Knative and integrated CI/CD with Tekton.

1.3 How to read this book

This book will take you through the anatomy of Anthos, the platform, the development environment, the elements of automation and scaling and the connection to patterns adapted from Google to attain excellence in modern software development in any industry. Equally importantly, each chapter has hands-on exercises to implement the techniques and practical examples on how to use the platform.

We begin with an overview of microservices and the shift to writing and running software in the cloud. We then introduce Anthos from a user's perspective and its major elements: Kubernetes, the Service Mesh, Security and Operational management and bring it all together. We then introduce Anthos as it runs on prem and in multiple-clouds, and at the edge showing the real power of a consistent platform across clouds, always with an eye to the applications that run on this platform.

The next section focuses on the operator experience with DevOps, networking, policy and walks through the benefits and transformation possible for each role and person in an enterprise as it modernizes. The final section focuses on various types of workloads run on the platform with an emphasis on data, ML, security, and an ecosystem of marketplace applications that run anywhere on Anthos. We will also talk in detail about the innovations the platform provides in migrating and modernizing existing applications. We conclude with a look at the roadmap and the upcoming capabilities from our customers and field.

In summary:

- Modern software applications provide a host of business benefits and are driving transformation in many industries.
- The backend for these applications is typically based on the cloud-native, microservices architectural pattern which allows for great scalability, modularity and a host of operational and devops benefits that are well suited running on distributed infrastructure.
- Anthos, which originated in Google Cloud is a platform for hosting cloud native applications providing both a develop and an operational benefits.

2

Cloud is a new computing stack

by Phil Taylor

The following topics will be covered in this chapter:

- Digital Velocity and The Enterprise Dilema
- Traditional models for application development and delivery
- Disrupting Application Delivery and The Birth of Cloud
- Microservices and Containers
- Software defined everything and DevOps
- Cloud is the modern computing stack

2.1 Introduction

Many technology executives and engineers talk about Cloud as a new destination. In reality it's a new state of mind. Cloud is best defined as a set of modern patterns and practices that abstract away the underlying infrastructure. On top of a new technology stack lies the evolution of new architectures for distributed systems that allow scale without requiring the developer to manage the underlying infrastructure and its design. This greatly lowers the barrier to building highly scalable, secure and distributed applications.

Most small companies or startups are already running on the Cloud and realizing these advantages. However, Enterprise computing platforms and applications are often more complex in comparison. This means Enterprises need to support their data centers and branch facilities for many years to come. Their move to the Cloud must be more purposeful and deliberate.

In the past, the upfront labor and cost of developing a new application represented the majority of a company's investment. There are three primary factors that influenced these costs:

1. The capital required to acquire the hardware and build the software for your new product.
2. The elapsed time to acquire the infrastructure and properly deploy it into a data center.
3. Quickly scaling infrastructure for anticipated and unanticipated usage spikes.

Early cloud computing capabilities solved these problems by allowing consumers to rent hardware in an Infrastructure as a Service model (IaaS). Under this model a new product team could deploy the needed infrastructure in minutes or hours and start developing their applications almost immediately. The per-minute charging models for Cloud infrastructure aided in keeping the upfront costs under control as well.

During the lifecycle of any successful product there are multiple scaling events and higher reliability needs that require you to expand your infrastructure to meet new demands. Public cloud providers make it easy for you to scale up or scale down your networking, compute and storage infrastructure to accommodate these events.

While it's more difficult than ever for enterprise companies to keep up with the pace of change, it's also easier than ever before to launch a startup and compete in their space. This creates a tremendous need for innovation within enterprise application design, compute models and delivery lifecycle..

In 2011, Marc Andreessen published an essay in The Wall Street Journal titled "Software is eating the world" (republished by his VC firm here <https://a16z.com/2011/08/20/why-software-is-eating-the-world/>). Most of his peer investors and business executives around the world probably thought Marc was nuts. After all it had been 10 few years since the last .com bubble had burst and they all thought it was coming again. The pace of change wasn't what it was today and technology visionary Marc was right again. Having invented the modern web browser and the company Netscape he had seen this type of change before.

The changes Marc was talking about are only now being applied by large enterprise companies at scale due to new capabilities like Google Cloud Platform (GCP) and Anthos. The pace of change is faster than ever. This term has become known as Digital Velocity.

2.2 Digital Velocity and The Enterprise Dilemma

At the turn of the 20th century business growth was slow, taking on average 90 years to reach 1 billion dollars (US) in annual revenues (see <https://www.inc.com/laura-montini/infographic/how-long-it-takes-to-get-to-a-1-billion-valuation.html>). This allowed companies who were first or early to their markets to effectively monopolize their industry. Innovation in manufacturing reduced this average to about 25 years in the years following World War II. Fast forward to the late 1990s and things really started heating up as companies leveraged technology to accelerate their business models. Most of the acceleration was in companies with no physical presence (true web companies), most full-stack companies failed in this era with the exception of a few shining stars like Google, Amazon and Netflix. Today, innovations in software development and infrastructure

operations have led to an extraordinary amount of acceleration. These advances in digital velocity have disrupted entire markets in a relatively short amount of time. Think of what Netflix did for streaming, Amazon for retail, AirBnB for short term rentals and Uber for taxis.

In 2021 and beyond there is not a market that is safe from disruption due to advances in Digital Velocity because of new software development patterns and more efficient infrastructure resources. Every company is finding they are rapidly becoming software companies. In the next section, you'll begin to analyze these software development patterns.

2.3 Traditional models for application development and delivery

Historically, enterprise applications were most often written using monolithic software development patterns. To help deal with monoliths at scale many developers moved from functional programming in languages like COBOL to Object Oriented Programming (OOP) in modern languages like C++ or Java and leveraged modular design patterns. This allowed developers to break up code bases into libraries that could be shared and re-used across entire application portfolios. For example, Other attempts at efficiency involved using stored procedures in relational databases to process data more rapidly. These innovations helped reduce toil, but they only rarely allowed enterprises to scale innovative ideas across more than a few teams. Instead, enterprises more often became quagmire in dealing with the legacy problems introduced by these new development paradigms.

When the IT industry began, code would be written and remain relatively static for months or even years. Most businesses did not invest in rewriting or re-platforming code. As these new OOP-based applications grew organically, supporting them became increasingly complex. Today, these messy areas of the solution or "hacks" as they are popularly called are formally referred to as technical debt. Technical debt is recognized as a first class factor that can be used to gauge the Digital Velocity of an organization or specific project. Too much technical debt results in a reduced velocity over time whereas not enough means you're not moving fast enough in most cases to keep up with business needs.

Enterprise applications were built using three popular design patterns (1) [Client-Server](#), (2) [n-tier web architecture](#) and (3) [Service Oriented Architecture](#).

2.3.1 Advantages and Pitfalls of Client / Server Architecture

The client-server architecture (see Figure 1 below) was simple and easy to build. Several popular programming languages and runtime environments helped build client-server applications faster. These Rapid Application Development (RAD) platforms like Visual Basic or Delphi were easy to use and many developers were able to get results rapidly.

The pitfalls of this application architecture included duplicated logic (business logic was duplicated in the client and server code bases), having to install the application on end-user machines, updating the application on those machines and keeping the server and its clients in sync. All of the aforementioned problems led to usability issues for the users, higher sustainability costs and longer lead times to get new features to market.

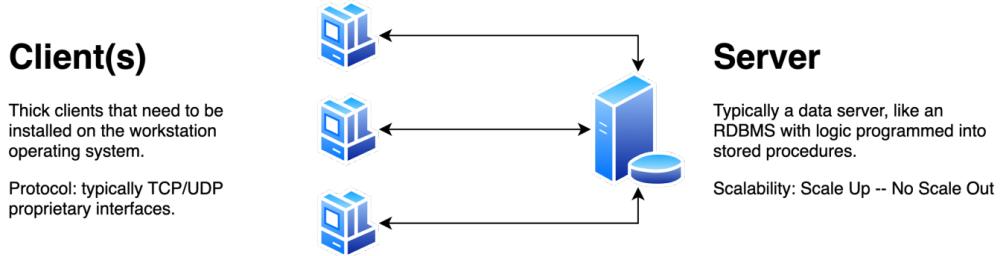


Figure 2.1 - Client Server Architecture

2.3.2 Advantages and Pitfalls of Web Architecture

A more complex architecture started to evolve with the invention of the modern web browser. This architecture is similar to the old dumb terminal days when an application was centrally deployed and managed. This new n-tier architecture (see Figure 2 below) was based on the idea that we should be generally separating our applications into three tiers: presentation, application or business logic and data.

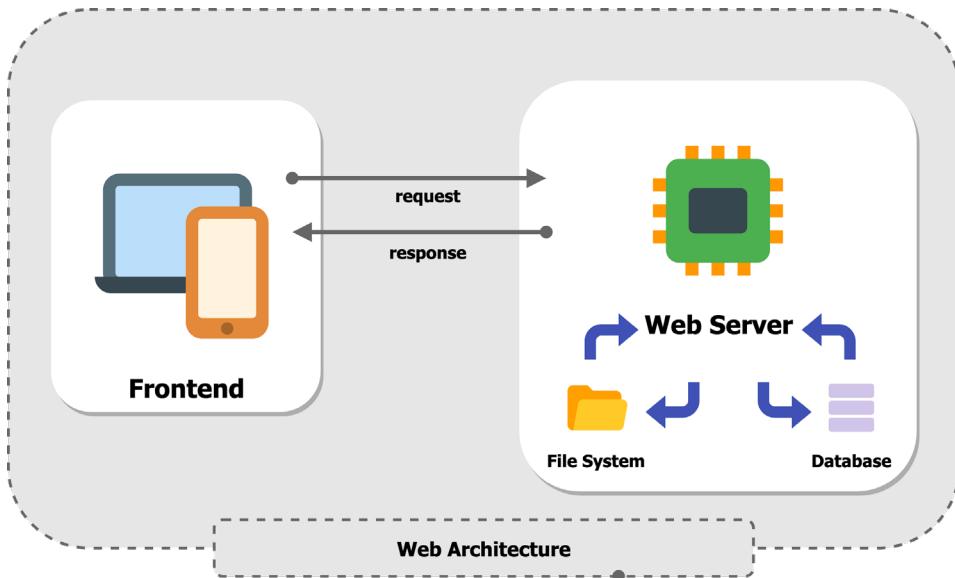


Figure 2.2 - N-Tier (web) Application Architecture

Although this architecture and deployment model fixed some pitfalls from client-server (installing and updating the application on clients), it still suffered from similar issues at scale. Most of these applications still suffer from duplicated logic (data validation, business

logic) that lead to usability issues, higher sustainability costs and longer lead times to get new features to market.

2.3.3 Service Oriented Architecture

Both client-server and n-tier architecture had another common pitfall: application crashes on the server affected every module or service and, possibly, a large majority of users. [Service Oriented Architecture \(SOA\)](#), described in Figure 3 below, helps to resolve some challenges with monolithic architectures and runtime support. The SOA model splits application business logic into well known [domains](#) and standardizes on protocols for communication between other tiers and security. This pattern provided lots of operational and development advantages including better scalability and resilience. While the API revolution still remains valid, there were a number of pitfalls with adopting SOA including cost and complexity of the underlying frameworks required by commercial products which were hard to install and maintain leading to a higher Total Cost of Ownership (TCO). However, its biggest pitfall was the learning curve to adopt the complex pattern and protocols. These limitations and high costs have resulted in a decline in popularity in recent years and a rise in the [Microservices](#) pattern. We will cover Microservices in more detail later in the book, Chapter 21, Application Modernization.

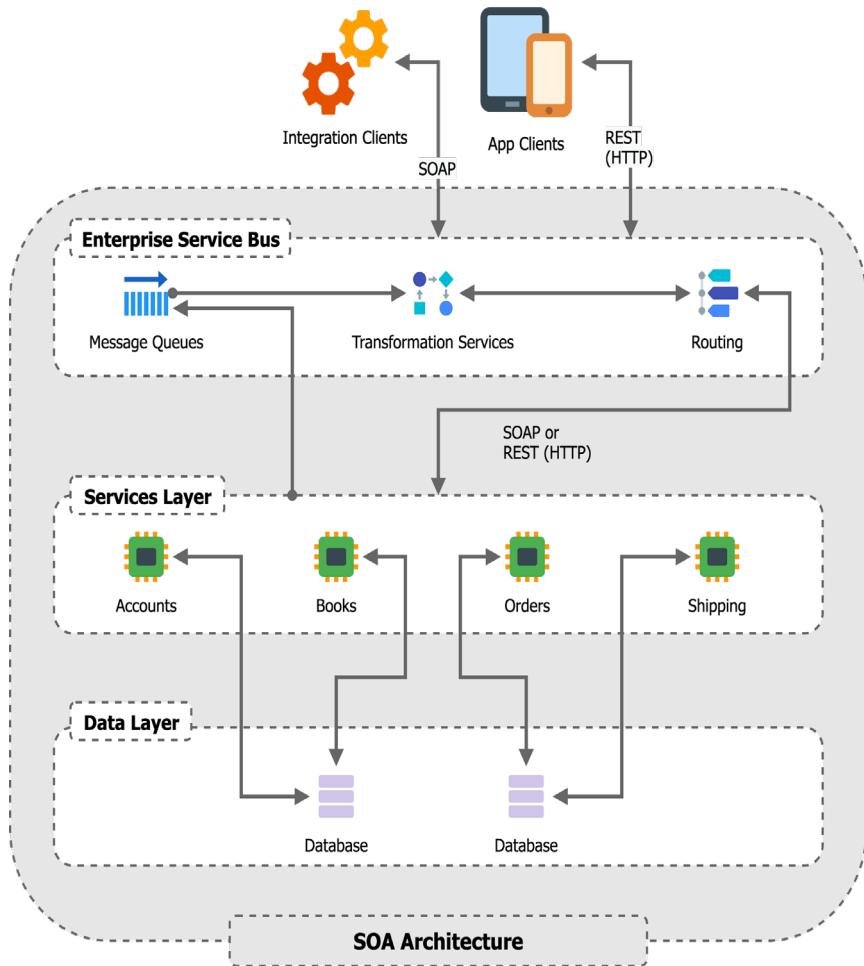


Figure 2.3 - SOA Architecture

During this evolution, data centers were also undergoing change and disruption.

2.4 Disrupting Application Delivery and The Birth of Cloud

As application development methods evolved, teams still struggled to execute successful builds of their source code and deploy them into the existing infrastructure. Processes were manual and error-prone. Developers, Operations teams, and managers all on a late-night conference call to deploy and debug an application were not uncommon. To help address this, [Continuous Integration \(CI\)](#) and [Continuous Delivery \(CD\)](#) was developed. Continuous Integration is the process of merging all developer changes once or multiple times

throughout the day to validate build integrity. Continuous Delivery is the process of validating and packaging deployment artifacts for release at any point in time. The goal of CI/CD is to create a reliable build and packaging process for repeatable success. Even with a powerful framework like CI/CD, realizing its efficiencies was difficult because the frameworks used to develop software were designed for applications that were designed for months or years and ran largely unchanged for long periods of time. One of the most popular of these project management processes is called Waterfall.

2.4.1 Disrupting How Software is Made

Waterfall is a project management process that is designed to manage a project based on milestones, timelines and resources. With Waterfall projects the team does a lot of up front work to think through all the requirements and design the solution. This often leads to problems later in the product cycle as the business requirements have changed and the design doesn't reflect the new business requirements. This inevitably leads to missed deadlines and quality issues with the finished product.

Beginning in 2001 with the release of The Agile Manifesto (<https://agilemanifesto.org/>), development teams began to evaluate their project management processes. A leading process that provides more efficient software development, Agile teams begin a project by thinking through major requirements and rough out a high level design rather than a full detailed design.

The team then works in iterations to deliver working software with more realistic designs at each iteration. Scrum is a well known framework for developing and delivering complex products using an agile mindset. Lots of teams prefer the [Scrum](#) methodology to run their Agile projects and two week iterations known as sprints. Once the teams have moved to Agile it is imperative that they implement CI/CD as they are now expected to build, test and ship the system at the end of each iteration.

2.4.2 Development Innovation at Google

At Google we leverage a number of patterns to create acceleration like [monorepo](#) and [trunk based development](#) that are described both in the popular book [Software Engineering At Google, O'Reilly](#). Many of the patterns in use at Google to create acceleration have been adopted across the industry.

2.4.3 Application Development throughout the Industry

In 2017, according to an [article published by InfoQ](#) the Facebook team was performing 50,000 builds a day just for its Android app. The statistics we just mentioned only refer to their CI process which is responsible for building the source code and running local testing to validate the build. It doesn't include the CD workflows, which would have massive infrastructure requirements to deploy 50,000 versions of the application on physical or virtual devices and run additional testing.

These infrastructure challenges aren't unique to Facebook. Years earlier, Google and Amazon ran into similar challenges building and operating their core infrastructure and applications. These challenges directly led to the cloud as we know it today. The invention of

cloud platforms were largely driven by internal IT groups inability to respond to business needs on an acceptable timeline. Shadow IT as it is commonly known is the practice of leveraging a 3rd party IT provider to deliver the business needs outside of internal IT groups. A lot of early cloud customers have started in a shadow IT mode before fully adopting new cloud platforms as a standard for infrastructure and application delivery. In 2006 when Amazon launched Amazon Elastic Compute Cloud (EC2). Heroku (2007), GCP (2008), Azure (2010) and Cloud Foundry (2011) rapidly followed EC2. Both Heroku and Cloud Foundry were cloud-based Platform as a Service (PaaS) solutions that focused on the idea of building [12 Factor Apps](#). Although all of these cloud based platforms were great at creating incremental improvements, enterprise customers struggled to create complete transformation. The public cloud needed more maturity and the early PaaS platforms like [Google App Engine](#) were narrow in scope, having limited support for languages and advanced language capabilities.

Although the aforementioned platforms all had their constraints this was the start of an era that would disrupt the data center and a number of business markets. It was the first time in history that you didn't need to build the software, acquire the hardware and build the data center to launch your new startup. You simply had to rent the infrastructure from the cloud or PaaS provider, build and deploy your app!

This new cohort of startups is often referred to as "Digital Natives"; companies born in the cloud. Their innovations continue to drive how applications are created.

2.4.4 Contract-first development, SOA and the evolution to Microservices

Around 2005 architects and developers started talking about the idea of [Test Driven Development \(TDD\)](#). It was said that by following the principles of TDD your code would be more testable and ultimately more reliable. The concept of TDD is to think through a coding problem and write the testing plan and actual tests prior to writing the code itself. Many developers found success in this practice which has evolved into the concept of [Design by contract](#). If you're writing a web service exposing an API then you should define the contract and data model first. This forces developers to think through the input/output boundaries and build a well encapsulated service. It also gives the consumer teams of your API a contract to start using in their own development process. Design by contract leverages several of the concepts from SOA. With SOA, developers build all of the subsystems using a web service pattern with messaging protocols between services. As newer ideas were applied to this concept, new technologies began to emerge.

2.5 Microservices and Containers

While enterprises were adopting SOA to build complex backend subsystems, tech startups continued to innovate. Microservices in many ways are an evolution of SOA. They share the same core idea to build all subsystems as web services that communicate with each other as illustrated in Figure 4 below.

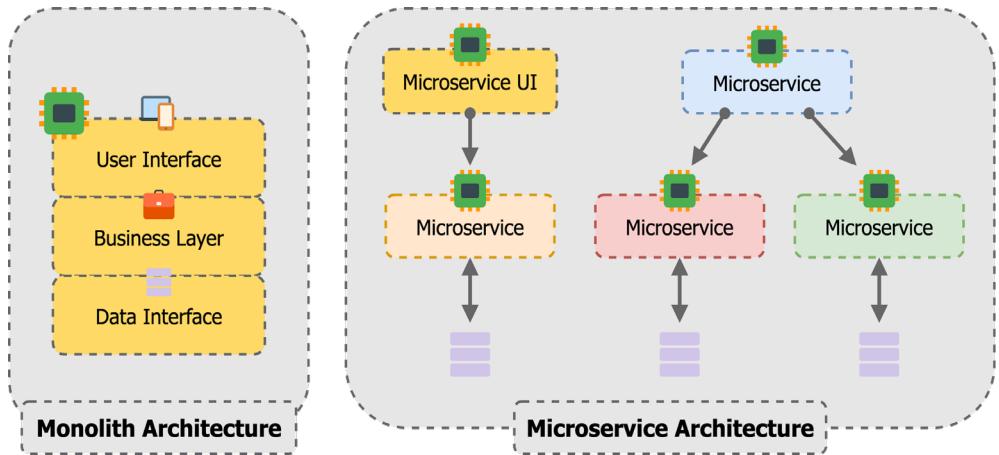


Figure 2.4 - Microservice Architecture

Microservices differ in that they favor smart endpoints with dumb protocols, as opposed to Enterprise Service Bus (ESB) and Simple Object Access Protocol (SOAP) protocols that are popular with Service Oriented Architecture (SOA). The function of the ESB is to centralize service communication, message transformation and routing. Whereas Microservices leverage Publish/Subscribe (Pub/Sub) message buses and favor decentralized communication between services.

Instead, teams build [REST](#) web services using HTTP + JSON for transport and payload protocols. Here are some other key differences:

- Microservices organize the design around business capabilities (Domain Driven Design). This allows us to write software with clear boundaries focused on solving a single business problem or a set of related problems within one microservice.
- The team chooses the programming language, database or other tech stack architecture to meet the requirements. This allows teams to choose technologies that lend well to solving the domain problem the microservice is responsible for providing.
- “Shared nothing” approach, each service has an isolated datastore. This approach allows teams to integrate at the API layer, encapsulating all functionality within their service and having clear boundaries to any dependent services.
- All communication happens through a single contract, typically exposed by a RESTful API. All communication between services (internal consumers or external consumers) access the same API interface.
- Smart endpoints, dumb pipes - simple protocols are leveraged and the services know how to handle the inputs and where to send the outputs.
- Message Queues are leveraged for asynchronous processing and throttling at scale. It’s the service’s responsibility to implement a message stack to support this capability. Not all services will need it.

- Automated testing is much easier with these cloud native design patterns, we are able to create ephemeral environments dedicated to testing a specific component or feature, before releasing our software into an integrated environment for end to end testing. In the past most teams struggled to write automated tests that would accurately validate their software and not require a lot of change between deployments. The [test pyramid](#) is a metaphor that tells us to group tests into buckets of different granularity (UI tests, service tests, unit tests). Most monolithic application development teams focused on building UI tests to validate the application. Since the UI group is known to require more integration this meant that these tests generally had to be updated regularly. This kept them from achieving advanced patterns like [canary](#) or [blue/green](#) deployments. A canary is where we would slowly roll out the tested software to a small subset of users to collect live feedback before rolling out to all users. Whereas a blue/green allows us to maintain two copies; a green or production environment and a blue or staging environment allowing us to rollback to a previous state quickly if anything goes wrong. Since most of these apps are built from scratch, the teams are able to properly apply the test pyramid, as well as advanced end-to-end deployment patterns like a canary or blue/green deployment. This leads to complete autonomous deployments and unmatched development velocity. Another big challenge for monolithic teams was data rehydration; the ability to seed the software platform with the data required for automated testing. The challenge of data rehydration is no longer an issue as these teams generally build up their required data sets as part of their automated testing code bases. Allowing them to meet the unique requirements for each use case in the test plan and avoid compliance problems created by copying live data to test systems.

Microservice architectures can be fast, efficient, and powerful. But to find widespread adoption, they needed infrastructure innovation as well. This innovation happened when Linux Containers came to the market.

2.5.1 Containers Enable Microservices

Containers have been around in some fashion since the days of Solaris Unix. The problem with containers historically is that they were hard to build and it took a series of Command Line Interface (CLI) calls to build one. In 2010, Docker revolutionized this process when they released their own open source container engine.

This new engine allowed us to document the container requirements in a source file and greatly reduced the number of CLI calls we have to make to successfully build a container. The new Dockerfiles, another Docker innovation, gave developers a way to document the requirements of a container using an Infrastructure as Code (IaC) approach instead of documenting CLI calls. By packing apps or microservices in containers, developers have a quick way to build, test and deploy their applications to a target environment. This allows developers to build software, build the target environment and then deploy and test the entire application stack as a single unit of work

Individual containers is a first step but full applications need to orchestrate multiple containers and instantiate them on an underlying infrastructure, making this more complicated.

Kubernetes solves most of the infrastructure and deployment problems (configuration, service discovery, secrets, scheduling, etc.) and provides a common orchestration interface for application infrastructure. As teams scale their solutions they will generally require capabilities that go beyond the basics that Kubernetes provides. Teams that are running large solutions will generally need more advanced capabilities and leverage other ecosystem services like Istio for service mesh or Apigee for API gateway traffic.

2.6 Software defined everything and DevOps

As you may be starting to realize, the key to velocity is through software development and independent deployments that allows teams to work in parallel with limited coordination. It's not enough to just build your application with software. The cloud has shown us that true velocity happens when the whole solution can be defined as code. This concept has been defined by the industry as DevOps which encompasses a lot more than just the technology pieces. We are not going to unpack all the DevOps concepts in this book and will focus only on the technical aspects. If you would like to learn more about DevOps we would recommend you read the [The DevOps Handbook](#).

To fully automate the build, testing and deployment of an application the team will need to become experts on CI/CD concepts, along with mastering [Infrastructure as Code \(IaC\)](#). The example below, Figure 5 shows a visualization of a CI/CD pipeline for supporting a Kubernetes application deployment.

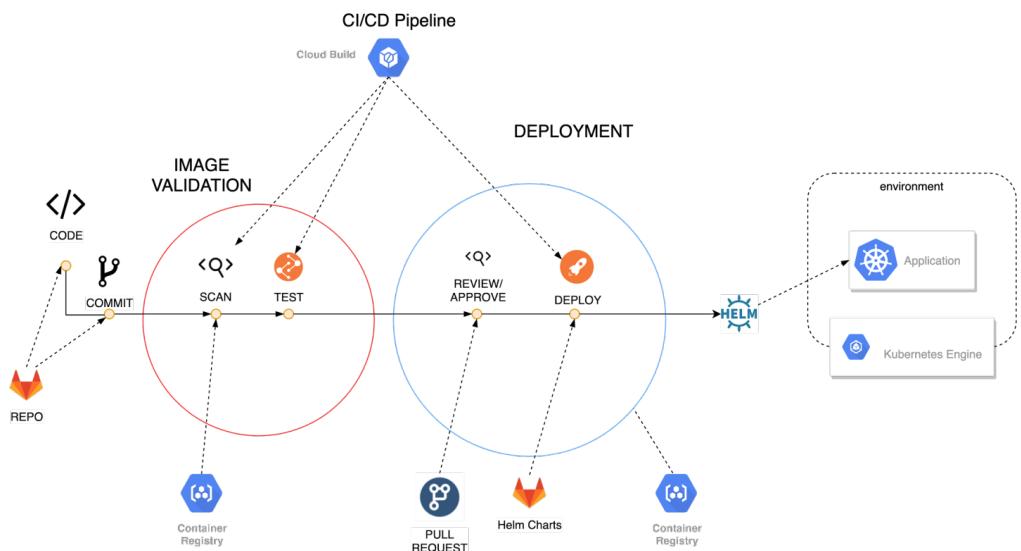


Figure 2.5 - CI/CD pipeline for Kubernetes Application

Teams that tried to deploy containers in the early days had mixed results as they needed to solve a lot of the core platform concerns required to serve these applications at scale. There is enough work to do in order to build, test and deploy the application successfully. This is the reason that Google created Kubernetes and subsequently Anthos. Anthos greatly simplifies the process of creating and managing the Kubernetes clusters and the auxiliary components and processes needed to support multiple clusters and hybrid cluster architectures. Anthos will help your teams deploy the core Kubernetes distribution, uniformly apply security policies to clusters, provide capabilities to enhance your service mesh to deal with hybrid cloud or hybrid cluster routing and more easily apply upgrades or other operational tasks. With Anthos developers can “write once, run everywhere” including across multiple clouds, the edge and on-prem environments.

2.7 Cloud is the modern computing stack

In this chapter you have come to understand the speed at which modern businesses are rising and how technology and software patterns have changed to enable this acceleration. A decade ago it would have been unthinkable to have a new business enter the Taxi market and in a few short years disrupt the entire industry. Today this is a regular occurrence due to the rapid growth of new businesses fueled by digital velocity software architectures and universal access thanks to the internet.

The concepts making this possible can be boiled down to distributed highly scalable architectures like Microservices and the use of DevOps patterns like SRE to deliver these solutions autonomously creating rapid iterative change. This rapid pace of innovation began with Cloud Native startups developing new technology. In a few short years they’re now being applied to enterprise workloads and existing applications successfully as well.

2.8 Summary

- Business are leveraging Public Cloud to meet the demands and agility their businesses require.
- Microservices are becoming more popular in software designs, allowing for parallel development to meet the software requirements without the typical drawbacks of a monolithic architecture.
- Good CI/CD capable of autonomous deployments leverganc IaC and automated testing is critical to success.
- Teams are leveraging containers and Kubernetes for the packaging and operation of these applications.
- Google Anthos is filling the void to make Kubernetes a reliable scalable platform for the enterprise.

3

Anthos, the one single pane-of-glass

by Melika Golkaram

This chapter covers:

- The advantages of having a Single Pane of Glass and its components
- How different personas can use and benefit from the above components
- Get some hands on experience configuring the UI and attaching a cluster to the Anthos UI

We live in a world where application performance is critical for success. To better serve their end-users, many organizations have pushed to distribute their workloads out of centralized data centers. Whether to be closer to their users, to enhance disaster recovery, or to leverage the benefits of cloud computing, this distribution has placed additional pressure on the tooling used to manage and support this strategy. The tools that have flourished under this new paradigm are those that have matured and become more sophisticated and scalable.

There is no one-size-fits-all tool. Likewise, there is no one person who can manage the infrastructure of even a small organization. All applications require tools to manage CI/CD, monitoring, logging, orchestration, deployments, storage, authentication/authorization, and more. In addition to the scalability and sophistication mentioned above, most of the tools in this space offer an informative and user-friendly graphical user interface (GUI). Having an easily understood GUI can help people use the tool more effectively since it lowers the bar for learning the software and increases the amount of pertinent output the user receives.

Anthos itself has the capacity to support hundreds of applications and thousands of services, so a high-quality GUI and user experience is required to leverage the ecosystem to its full potential. To this end, Google Cloud Platform has developed a rich set of dashboards and integrated tools within the Google Cloud Console to help you monitor, troubleshoot, and interact with your deployed Anthos clusters, regardless of their location or infrastructure

provider. This single pane of glass allows administrators, operations professionals, developers, and business owners to view the status of their clusters and application workloads, all while leveraging the capabilities of Google Cloud's Identity and Access Management (IAM) framework and any additional security provided on each cluster.

The Anthos single pane of glass is not the first product to attempt to centralize the visibility and operations of a fleet of clusters, but it is the one offering support for a large variety of environments. In order to fully understand the benefits of the Anthos GUI, we are going to take a look at some of the other options available to aggregate and standardize the interactions with multiple Kubernetes clusters. However, we first need to have a standard set of terminology and have a brief overview of some DevOps principles.

In this chapter, we take a journey through the offerings of the Anthos GUI, its "Single Pane of Glass".

3.1 Terminology

A single pane of glass offers 3 main pillars of characteristics that are shared across all operators, industries and operations scales. These three pillars are:

Centralization: As the name suggests, a single pane of glass should provide a central UI for resources, no matter where they run and to whom they are provided. The former aspect relates to which infrastructure and cloud provider the clusters are operating on and the latter relates to inherently multi-tenant services where one operator centrally manages multiple clients' clusters and workloads. With the benefits of a central dashboard, admins will be able to get a high level view of resources and drill down to areas of interest without switching the view.

However, a central environment might cause some concern in areas of privacy and security. Not every administrator is required to connect to all clusters and neither all admins should be able to have access to the UI. A central environment should come with its own safeguards to avoid any operational compromise with industry standards.

Consistency: Let's go back to the scenario of an operator running clusters and customers in multi-cloud or hybrid architectures. A majority of infrastructure providers, whether they offer proprietary services or run on open source, attempt to offer a solid interface for their users. However, they use different terminology, have inconsistent views on the priorities. Finally, depending on their UI philosophy and approach, they design the view and navigation differently. Remember, for a cloud provider, cluster and container management is only part of the bigger suite of services and a member of a pre-designed dashboard. While this might be a positive element in single operating environments (you can learn to navigate outside of Kubernetes dashboard into the rest of cloud services dashboard with minimum switching), it becomes a headache in multi-environment services and those who mainly only focus on Kubernetes.

Ease of use: Part of the appeal of a single pane of glass in operation is how data coming from different sources are aggregated, normalized and visualized. This brings a lot of simplicity in drilling down into performance management and triage especially if it combines a graphical interface with it.

A graphical UI has always been an important part of any online application. First, at some point in an app management cycle, there is a team who doesn't have neither the skills nor

the interest of working with remote calls. This team expects a robust, easy to navigate and a highly device agnostic UI for their day to day responsibility.

Second, regardless of the skillsets, an aggregated dashboard has so much more to offer in one concentrated view than calling service providers and perhaps clusters individually given that the UI provides lots of data fields with the right installation and readability.

When working in an IT division or department of an organization, roles can be split in any number of ways. We are going to consider a specific set of users, or personas, that are indicative of common roles and responsibilities within an organization. However, any given group may combine some of these roles together, or split them out further.

3.1.1 Personas

We start where most companies start: with the **Infrastructure Administrator**. This is the individual who is primarily responsible for the day-to-day maintenance of existing hardware, and the deployment of new hardware and core services. In the context of Anthos, this person will usually be responsible for the network backbone, as well as the physical servers the on-premise clusters run on. In general, this persona has the least interaction with the day-to-day operations and monitoring of the clusters, but will be involved in discussions around new compute or network requirements, and will be in the notification chain for outages.

A large portion of the tooling provided as part of Anthos is intended to make the day-to-day operations of clusters and workloads as streamlined as possible - this task typically falls to the Operations Team, headed by the **Operations Administrator**. For Anthos operations, this persona will typically be handling core monitoring and logging, and ensuring the health of the internal platform built on top of an organization's infrastructure and cloud environments, rather than dealing with the physical hardware itself. While Google advocates for the wider adoption of DevOps or Site Reliability Engineering (SRE) practices¹, even in a company that has fully embraced the DevOps/SRE ways, ownership and monitoring of applications is never fully decentralized. This persona will be involved in managing the health and utilization of the platform as a whole. They may drill into a specific application from time-to-time, but more often acts as a resource for the Developers (see below), rather than handling any issues directly.

As distributed operations and deployments expand, some traditional security practices such as strict release cycles, scheduled, manual regression tests, or tight lockdown of production environments have become unwieldy, slow, and frustrating. Because of this, the Security Administrator is moving to perform more monitoring and enforcement-by-default for the deployed applications and clusters. While a security team may be responsible for both physical and digital security, this book is primarily focused on the digital side. Anthos provides robust tooling, such as Anthos Configuration Management², and the Anthos dashboards to enable the Security Administrator to enforce, monitor, and audit clusters and workloads that have been deployed.

Finally, the Developers of an organization have important roles to play in deploying and managing the application their teams are directly responsible for. As we said before, we strongly encourage that Developers are given the tools and responsibility to operate and

¹ Books can be found here. <https://sre.google/books/>

² See Chapter 14: Anthos Configuration Management

maintain their own applications, while removing the need for them to manage their infrastructure directly. For the purposes of this chapter, we will consider the Developer persona to encompass all technical personnel associated with a particular application, including application developers, test and quality assurance, and DevOps/SRE experts.

3.1.2 DevOps/Site Reliability Engineering Concepts

DevOps is a method of approaching development to encourage faster delivery while improving reliability of the finished product. Site Reliability Engineering (SRE) is Google's way of implementing DevOps in the particular Google style, moving as much as possible to automation and reducing the manual work that needs to be done. An exhaustive review of the principles and practices for DevOps/SRE is far too broad for this book, let alone this chapter. However, several concepts from those principles have an outsized presence in the Anthos GUI. Specifically, we will be exploring the concepts related to automatically monitoring a workload, including how to analyze the metrics that the Anthos system captures:

- A **Service Level Indicator (SLI)** is a specific measure of how a workload is performing. Many types of SLIs can be created, either generic or specific to the workload, but the four key focus areas are LETS or Latency, Errors, Traffic, and Saturation. For Anthos, this can track the resource requests for a pod, or the time it takes to process a specific inbound HTTP request.
- With the SLIs, we can generate **Service Level Objectives (SLO)**. These are achievable targets we should expect to satisfy regularly, expressed as a measurement of the SLI. E.g., we may specify that the 95th percentile latency for a service must be under 200ms, or that the general error rate for an application must be less than one per minute.
- Usually a more generous target than the SLO, a **Service Level Agreement (SLA)** is a minimum target that must be met, or the consequence specified in the SLA will be invoked. E.g., this may be a requirement that the platform is available 99.99% of the time, or that the 95th percentile latency of a service must never exceed 2 seconds.

For a more in-depth examination of DevOps, please see the Google Cloud DevOps page at <https://cloud.google.com/devops>. SRE is used across all of Google, and more information can be found at <https://sre.google/>

3.1.3 Other Terminology

In addition to the concepts above, we have a few miscellaneous terms that we will be using throughout this chapter. For Anthos, we have two types of clusters that are visible in the GUI: **Attached** and **Anthos-managed**. Anthos-managed clusters are clusters that are deployed via Anthos utilities and are a flavor of Google Kubernetes Engine (GKE) in GCP, on-premise, or in another cloud. Attached clusters are those that are compatible with the Connect software, but are not actual GKE clusters. These can include the cloud-native Kubernetes flavors or other third-party versions of Kubernetes. There are some features that are not officially supported on Attached clusters; these features will be called out when they are covered in the chapter.

3.2 Non-Anthos Visibility and Interaction

Anthos is not the first solution to expose information about a Kubernetes cluster through a more easily digested form than the built-in APIs. While many developers and operators have used the command-line interface (CLI), kubectl, to interact with a cluster, the information presented can be very technical and does not usually help surface potential issues in a friendly way. Extensions to Kubernetes such as Istio or Anthos Configuration Management typically come with their own CLIs as well (istioctl and nomos, for example). Cross-referencing information between all the disparate tools can be a substantial exercise, even for the most experienced developer or operator.

3.3 Kubernetes Dashboard

One of the first tools developed to solve this particular issue was the Kubernetes Dashboard³. While this utility is not deployed by default on new Kubernetes clusters, it is trivial to deploy to the cluster and begin utilizing the information it provides. In addition to providing a holistic view of most of the components of a Kubernetes cluster, the Dashboard also provides users with a GUI interface to deploy new workloads into the cluster. This makes the dashboard a convenient and quick way to view the status and interact with a new cluster.

However, it only works on one cluster. While you can certainly deploy the Kubernetes Dashboard to each of your clusters, they will remain independent of each other and have no cross-connection. In addition, since the dashboard is located on the cluster itself, accessing it remotely requires a similar level of effort to using the CLI tool, requiring services, load balancing, and ingress rules to properly route and validate incoming traffic. While the dashboard can be powerful for proof-of-concept or small developer clusters, multi-user clusters need a more powerful tool.

3.3.1 Provider-specific UIs

Kubernetes was released from the beginning as an open-source project. While based on internal Google tools, the structure of Kubernetes allowed vendors and other cloud providers to easily create their own customized versions of Kubernetes, either to simplify deployment or management on their particular platform(s), or to add additional features. Many of these adaptations have customized UIs for either deployment or management operations.

For cloud providers in particular, much of the existing user interfaces for their other products already existed and followed a particular style. Each provider developed a different UI for their particular version of Kubernetes. While a portion of these UIs dealt with provisioning and maintaining the infrastructure of a cluster, some of the UI was dedicated to cluster operations and manipulation. However, each UI was implemented differently, and cannot manage clusters other than the native Kubernetes flavor for that particular cloud provider.

³ <https://github.com/kubernetes/dashboard>

3.3.2 Bespoke Software

Some companies have decided to push the boundaries themselves and develop their own custom software and UIs to visualize and manage their Kubernetes installations and operations. While always an option due to the open standards of the Kubernetes APIs, any bespoke development brings all the associated challenges that come with maintaining any custom operations software: maintaining the software for new versions, bug fixing, handling OS and package upgrades, etc... For the highest degree of customization, nothing beats bespoke software, but the cost vs. benefit calculation does not work out for most companies.

3.4 The Anthos UI

Each of the previous solutions has a fundamental flaw that prevents most companies from fully leveraging it. The Kubernetes Dashboard has no multi-cluster capability, and does not handle remote access easily. The provider-specific UIs work well for their flavor, but cannot handle clusters that are not on their network or running their version of Kubernetes. And bespoke software comes with a high cost of development and maintenance. This is where the Anthos multi-cluster single pane of glass comes into play. This single pane of glass is an extension of, and embedded in, Google Cloud Platform's already extensive Cloud Console that allows users to view, monitor, and manage their entire cloud infrastructure and workloads.

The solution Google has developed for multi-cluster visibility in Anthos depends on a new concept called Fleets (formerly referred to as Environs) (continue reading the next chapter), the Connect framework and the Anthos dashboard. The Anthos dashboard is an enhancement of the existing GKE dashboard that Google has provided for several years for its in-cloud GKE clusters. The Connect framework is new with Anthos and simplifies the communication process between Google Cloud and clusters located anywhere in the world. Fleets are methods of aggregating clusters together in order to simplify common work between them. Let's take a moment to discuss a bit more about fleets.

3.4.1 Fleets

Fleets are a Google Cloud concept for logically organizing clusters and other resources, letting you use and manage multi-cluster capabilities and apply consistent policies across your systems. Think of them as a grouping mechanism that applies several security and operation boundaries to resources within a single project⁴. They help administrators build a one-to-many relationship between a Fleet and its component clusters and resources to reduce the configuration burden of individual security and access rules. The clusters in a Fleet also exist in a higher trust relationship with each other by belonging to the same Fleet. This means that it is easier to manage traffic into and between the clusters and join their service meshes together.

An Anthos cluster will belong to one and only one Fleet and cannot join another without leaving the first. Unfortunately, this can present a small problem in complex service communications. For example, if we have an API service and a Data Processing service that

⁴A Google Cloud Platform project is a set of configuration settings that define how your app interacts with Google services and what resources it uses.

need to run in distinct fleets for security reasons, but both need to talk to a bespoke Permissioning service. The Permissioning service can be placed in one of the two fleets, but whichever service does not share its Fleets will need to talk to the Permissioning service using outside-the-cluster networking. However, this rule for fleets prevents users from accidentally merging clusters that must remain separate, as allowing the common service to exist in both fleets simultaneously would open up additional attack vectors.

When multiple clusters are in the same Fleets, many types of resources must have unique names, or they will be treated as the same resource. This obviously includes the clusters themselves, but also namespaces, services, and identities. Anthos refers to this as “sameness”. Sameness forces consistent ownership across all clusters within an Fleets, and namespaces that are defined on one cluster, but not on another, will be reserved implicitly.

An important consideration when designing the architecture of your services is that the namespace and services sameness also includes a unique internal identity across all environments in one Fleets. This means by implicitly configuring a mesh policy that allows simple access between *service A* and *service B* will apply this policy across all pods in all environments and clusters that these services can be found.

Finally, Anthos allows all services to utilize a common identity when accessing external resources such as Google Cloud services, object stores, and so on. This common identity makes it possible to give the services within a fleet access to an external resource once, rather than cluster-by-cluster. While this can be overridden and multiple identities defined, if resources are not architected carefully and configured properly, negative outcomes can occur.

3.5 Connect, How does it work?

Now that we have discussed fleets, we need to examine how the individual clusters communicate with Google Cloud. Any cluster that is part of Anthos, whether Attached⁵ or Anthos-managed, has Connect deployed to the cluster as part of the installation or registration process. This deployment establishes a persistent connection from the cluster outbound to Google Cloud that accepts traffic from the cloud to permit cloud-side operations secure access to the cluster. Since the initial connection is outbound, it does not rely on a fully routable connection from the cloud to the cluster. This greatly reduces the security considerations and does not require the cluster to be discoverable on the public internet.

Once the persistent connection is established, Anthos can proxy requests made by Google Cloud services or users using the Google Cloud UI to the cluster whether it is located within Google Cloud, in another cloud provider, at the edge, or on-premise. These requests use the user’s or the service’s credentials, maintaining the security on the cluster and allowing the existing role-based access controls (RBAC)⁶ rules to span both direct connectivity as well as connections through the proxy. A request using the Anthos UI may look like Figure 3.1:

⁵Attaching clusters lets you view your existing Kubernetes clusters in the Google Cloud Console along with your Anthos clusters, and enable a subset of Anthos features on them, including configuration with Anthos Config Management. More details can be found here: <https://cloud.google.com/anthos/docs/setup/attached-clusters#prerequisites>

⁶ Role-based access control (RBAC) is a set of permissions and an authorization component that allows/denies admin or compute objects access to a set of requesting resources.

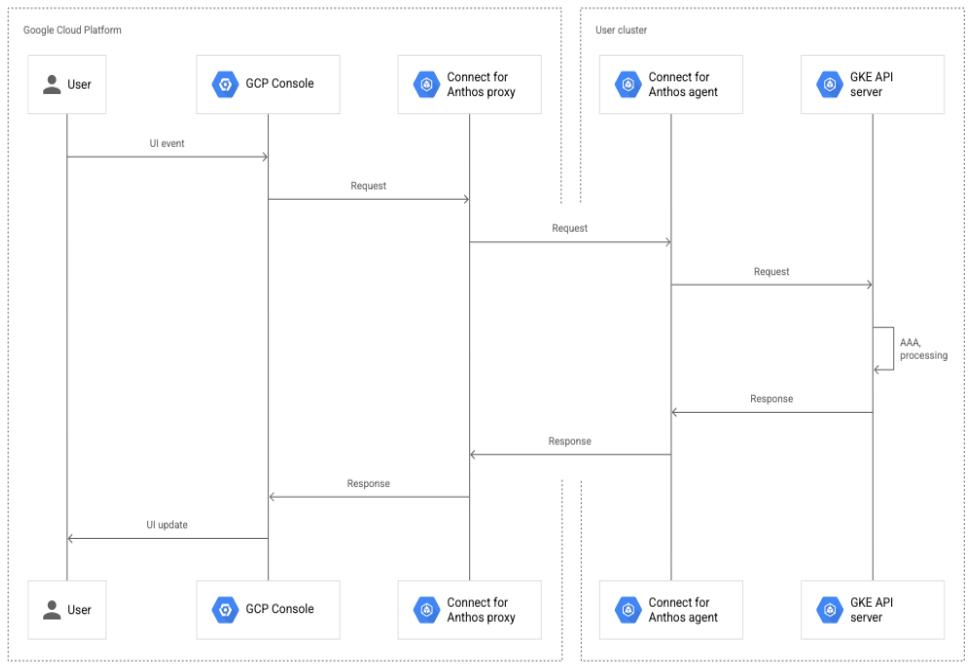


Figure 3.1. Flow of request and response from Google Cloud to Cluster and back

While the tunnel from the Connect agent to Google Cloud is persistent, each stage of each request is authenticated using various mechanisms to validate the identity of the requestor and that that particular layer is allowed to make the request. Skipping layers is not permitted and will be rejected by the next layer receiving the invalid request. An overview of the request-response authentication is seen in Figure 3.2:

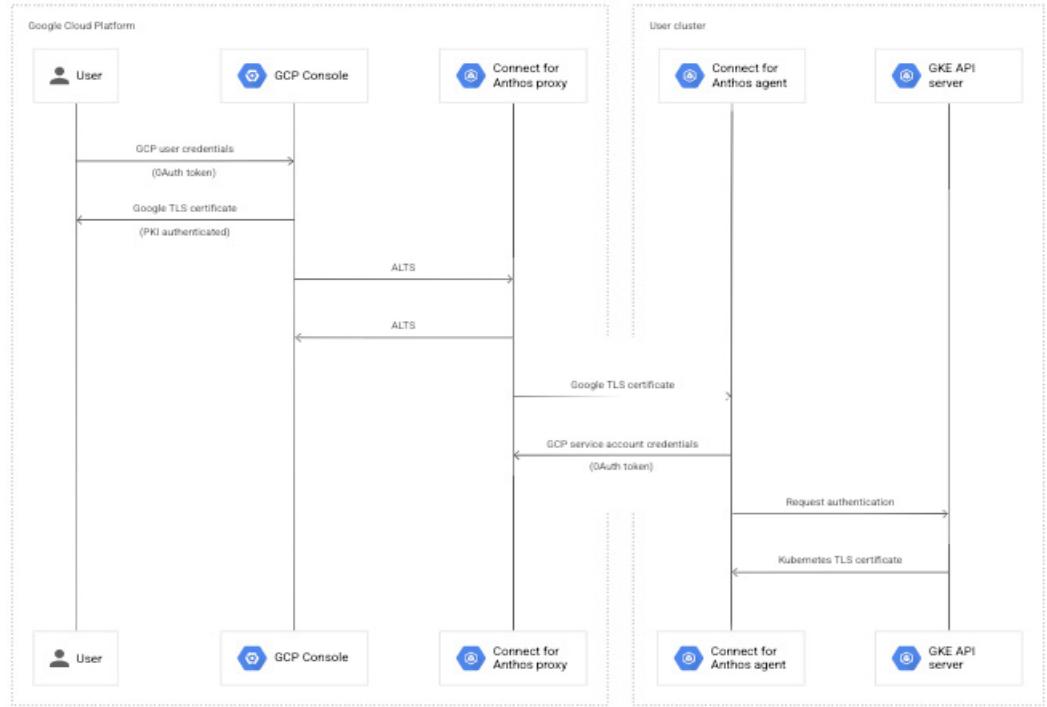


Figure 3.2. Request Validation Steps from Google Cloud to Cluster

Even if a user has the appropriate RBAC permissions to gain access to a given cluster, they must still be allowed to view the Google Cloud project the cluster is attached to in order to use the Connect functionality. This uses the standard IAM processes for a given project, but having the separate permission allows the Security team to grant a user access to a cluster through a direct connection (or some other tunnel), but not allow them remote access via Google Cloud.

Connect is compliant with Google’s Access Transparency⁷ which provides transparency to the customer in 2 main areas:

- **Access approval:** Customers can authorize Google support staff to work on certain parts of their services. Customers can view the reasons why a Google employee might need that access.
- **Activity visibility:** Customers can import access logs into their project Cloud logging to have visibility into Google employees’ actions and location and can query the logs in real-time, if necessary.

⁷ Access Transparency enabled services lets customers control access to organization’s data by Google personnel. It also provides logs that capture the actions Google personnel take when accessing customer’s content.

3.5.1 Installation and Registration

In order to leverage the Connect functionality, we obviously need to install the Connect agent on our cluster. We also need to inform Google about our cluster and determine which project, and therefore which Fleets, the cluster belongs to. Fortunately, Google has provided a streamlined utility for performing this task via the gcloud command line tool⁸. This process will utilize either Workload Identity or a Google Cloud Service Account to enroll the cluster with the project's Connect pool and install and start the Connect agent on the cluster.

While these steps have enrolled the cluster with Google and enabled most Anthos features, you still need to authenticate with the cluster from the Google Console in order to view and interact with the cluster from Google Cloud. Connect allows authentication via Cloud Identity (when using the Connect Gateway)⁹, bearer token, or OIDC, if enabled on the cluster. The easiest, and recommended, method is to use Cloud Identity, but this requires the activation and configuration of the Connect Gateway for the cluster. For more information on Connect Gateway, please see Chapter 6: Operations Management with Anthos.

3.6 The Anthos Cloud UI

Now that we've done the plumbing, we can actually walk through and show off the UI. Google provides the Anthos UI via the Cloud Console, at the project level. Since the Anthos UI is only visible at the project level, only clusters registered to that project's Fleets are visible. We will discuss other tools later in this chapter to view clusters in multiple fleets simultaneously. The Anthos UI menu contains multiple sub-pages, each providing a focus on a distinct aspect of cluster management. At the time of writing, these sections are the Dashboard, Service Mesh, Config Management, Clusters, Features, Migrate to containers, and Security. Let's take a look at each of these pages:

3.6.1 The Anthos Dashboard

The default page for the Anthos menu, and the central hub for the UI, is the Dashboard. The Dashboard is intended to give Admins a wide-angle view of the clusters in the current Fleet, while making it easy to drill down into details for the specific components. To start, go to the hamburger menu on the top left corner of the console (Figure 3.3). Find "Anthos" from the menu and click on it to enter the all-Anthos features page. Image 8 shows how to navigate to Anthos dashboard.

⁸ <https://cloud.google.com/anthos/multicluster-management/connect/registering-a-cluster>

⁹ Google Cloud Identity and Access Management (IAM), lets you give more granular access to specific Google Cloud resources and prevents unwanted access to other resources.

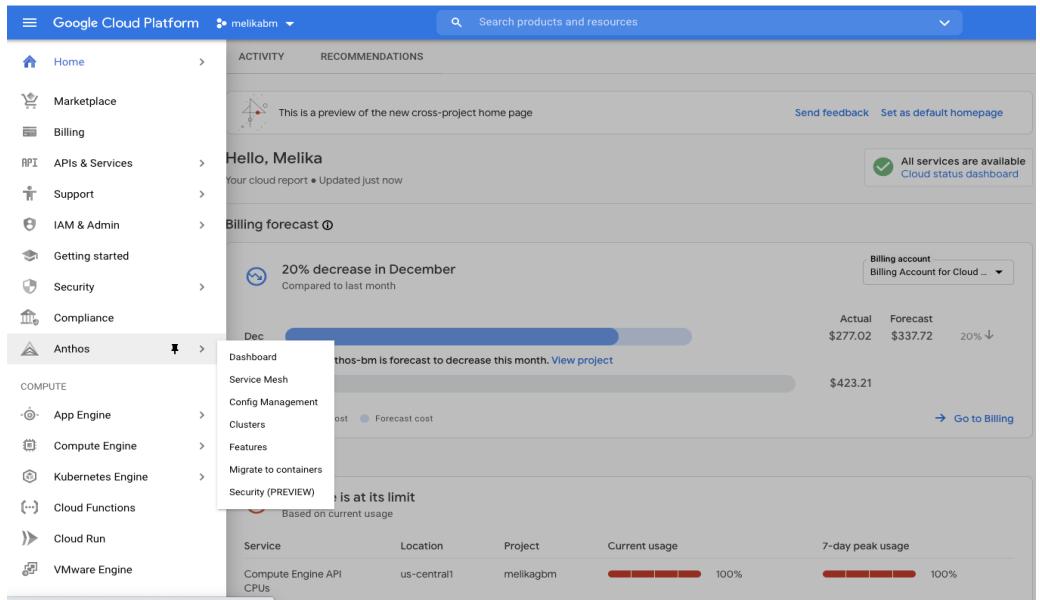


Figure 3.3: Navigation to Anthos dashboard

Figure 3.4 shows an example of the Anthos Dashboard view.

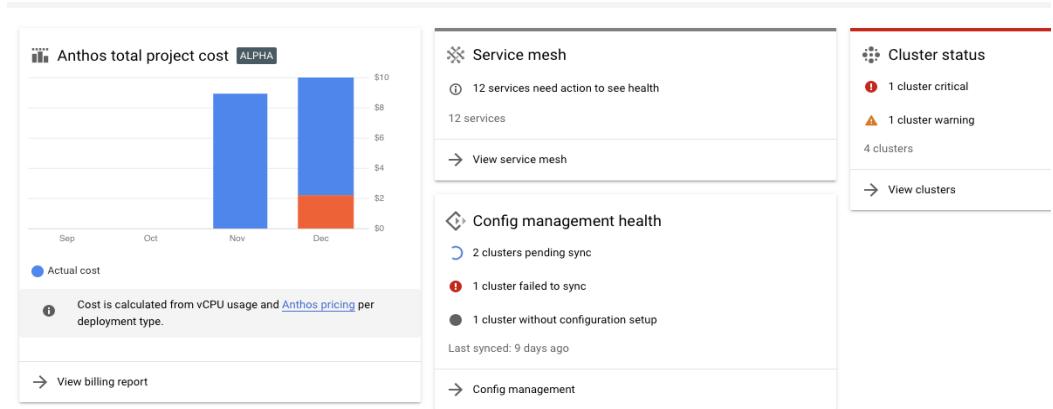


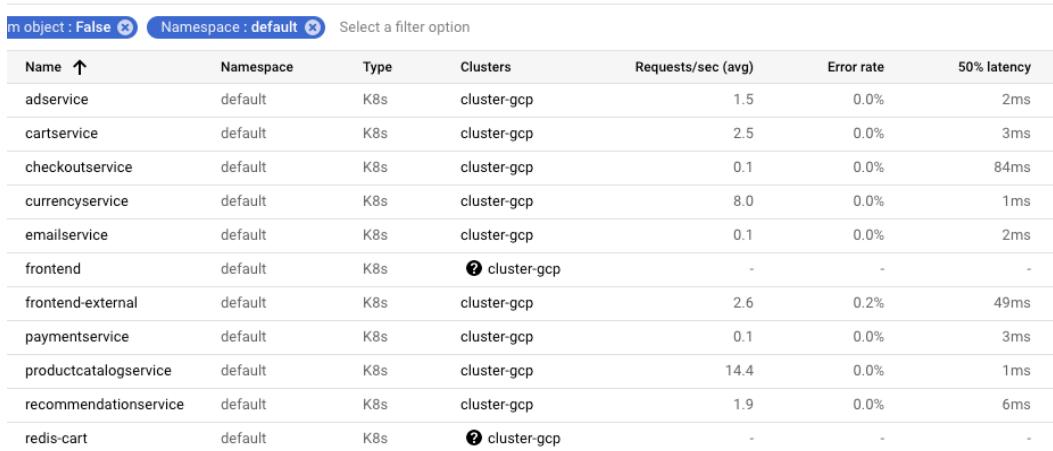
Figure 3.4. Example of an Anthos Dashboard

While this example shows the current Anthos project cost, the Dashboard still leverages Google's IAM and that information will only appear if the viewing user has the appropriate billing-related permissions. The remaining sections highlight critical errors or other user-

involved issues for that particular aspect of Anthos. Following those links takes you to the appropriate sub-page below.

3.6.2 Service Mesh

The Service Mesh page shows all services registered in any of the clusters in the current Fleet. The initial list shows the names, namespaces, and clusters of each service, as well as basic metrics such as error rate and latency at predefined thresholds. This list can also be filtered by namespace, cluster name, request per second, error rate, latency, request size and resource usage to allow admins to easily drill down for specific tasks. Figure 3.5 shows the Service Mesh screen filtered for services in the default namespace.



The screenshot shows a table of service metrics. At the top, there are three filter buttons: 'm object : False' (disabled), 'Namespace : default' (selected), and 'Select a filter option'. The table has columns: Name (sorted ascending), Namespace, Type, Clusters, Requests/sec (avg), Error rate, and 50% latency. The data rows are:

Name ↑	Namespace	Type	Clusters	Requests/sec (avg)	Error rate	50% latency
adservice	default	K8s	cluster-gcp	1.5	0.0%	2ms
cartservice	default	K8s	cluster-gcp	2.5	0.0%	3ms
checkoutservice	default	K8s	cluster-gcp	0.1	0.0%	84ms
currencyervice	default	K8s	cluster-gcp	8.0	0.0%	1ms
emailservice	default	K8s	cluster-gcp	0.1	0.0%	2ms
frontend	default	K8s	cluster-gcp	-	-	-
frontend-external	default	K8s	cluster-gcp	2.6	0.2%	49ms
paymentservice	default	K8s	cluster-gcp	0.1	0.0%	3ms
productcatalogservice	default	K8s	cluster-gcp	14.4	0.0%	1ms
recommendationservice	default	K8s	cluster-gcp	1.9	0.0%	6ms
redis-cart	default	K8s	cluster-gcp	-	-	-

Figure 3.5. Service Mesh UI with Filters.

3.6.3 Config Management

Anthos Configuration Management, explored in depth in Chapter 14, is Anthos' method of automatically adding and maintaining objects on a Kubernetes cluster. These objects can include most common Kubernetes core objects (such as Pods, Services, ServiceAccounts, etc...) as well as custom entities such as policies and cloud-configuration objects. This tab displays the list of all clusters in the current Fleet, their sync status, and which revision is currently enforced on the cluster. The table also shows whether Policy Controller¹⁰ has been enabled for the cluster.

¹⁰ Policy Controller is part of Anthos Configuration Management, allowing administrators to define customized rules to place guard-rails for security, resource management, or operational reasons. Policy Controller is explored in greater depth in Chapter 14 with Anthos Configuration Management.

The screenshot shows the 'Clusters for "melikabm"' section in the Anthos Config Management interface. At the top, there is a 'CONFIGURE' button with a gear icon. Below it, a table lists four clusters: 'azure-cluster', 'cluster-1', 'cluster-gcp', and 'externalazure'. The table has columns for 'Cluster name', 'Config sync status', 'Revision', and 'Policy controller status'. The 'Config sync status' column includes icons for 'Unreachable' (red exclamation mark), 'Pending' (blue dot), and 'Not installed' (grey circle). The 'Revision' column shows the commit hash for each cluster. The 'Policy controller status' column shows 'Disabled' for all clusters.

Cluster name	Config sync status	Revision	Policy controller status
azure-cluster	⚠️ Unreachable		Disabled
cluster-1	• Pending	master/47b472d55320c37fb8c064571c617669febd06f5	Disabled
cluster-gcp	• Pending	master/47b472d55320c37fb8c064571c617669febd06f5	Disabled
externalazure	● Not installed		Disabled

Figure 3.6. Clusters in Config Management View

Selecting a specific cluster opens up the config management cluster detail as shown in Figure 3.7. This detailed view gives further information about the configuration settings, including the location of the repo used, the cycle for syncing, and the version of ACM running on the cluster.

The screenshot shows the 'Clusters' page in the ACM interface. A cluster named 'cluster-1' is selected. The 'DETAILS' tab is active. The 'Anthos config management' section shows 'ACM' and 'Version 1.5.2'. The 'Config sync' section lists configuration details for 'cluster-1', including sync status (Pending), sync repo (ssh://melikag@google.com@source.developers.google.com:2022/p/melikabm/r/config-repo), revision (master/47b472d55320c37fb8c064571c617669febd06f5), and sync wait (disabled). The 'Policy controller' section shows audit interval, template library installed, exemptable namespaces, and referential rules status as disabled.

Figure 3.7. Cluster Detail in Configuration Management View

3.6.4 Clusters

The Clusters menu lists all clusters in the current Fleet, along with the location, type, labels, and any warnings associated with each cluster as shown below in Figure 3.8. By selecting a cluster in the list, a more detailed view of the cluster, with the current Kubernetes version, the CPU and Memory available, and the Features enabled, will be displayed in the right sidebar as shown in Figure 3.8. Below the sidebar information, a “Manage Features” button will take you to the Features tab for that cluster. In figure 3.8, below clusters are created on the project:

- GKE (cluster-gcp)
- Baremetal (cluster-1)
- Azure AKS (azure-cluster and externalazure)

Clusters BETA

[CREATE CLUSTER](#)[REGISTER EXISTING CLUSTER](#)

Status

❗ 1 cluster critical

⚠ 1 cluster warning

4 clusters total

Anthos managed clusters

Filter table

●	Name ↑	Location	Type	Labels
❗	azure-cluster	registered	Unknown	
✓	cluster-1	registered	Anthos	
✓	cluster-gcp	us-central1-c	GKE	
⚠	externalazure	registered	Unknown	

Figure 3.8. List view in the Clusters Menu

The screenshot shows the 'cluster-1' details page. At the top, there's a back button and the cluster name 'cluster-1'. Below this is a section titled 'Details' containing a table with cluster specifications. Under 'Cluster features', there's another table showing the status of various Anthos services.

Type	Anthos
Master version	v1.18.6-gke.6600
Location	registered
Cluster Size	5
Total cores	40 CPU
Total memory	157.81 GB

Feature	Status
Feature Authorizer	Enabled
Binary Authorization	Anthos Feature ⓘ
Cloud Run	Anthos Feature ⓘ
Config Management	Enabled
Ingress	Available ⓘ
Service Mesh	Anthos Feature ⓘ

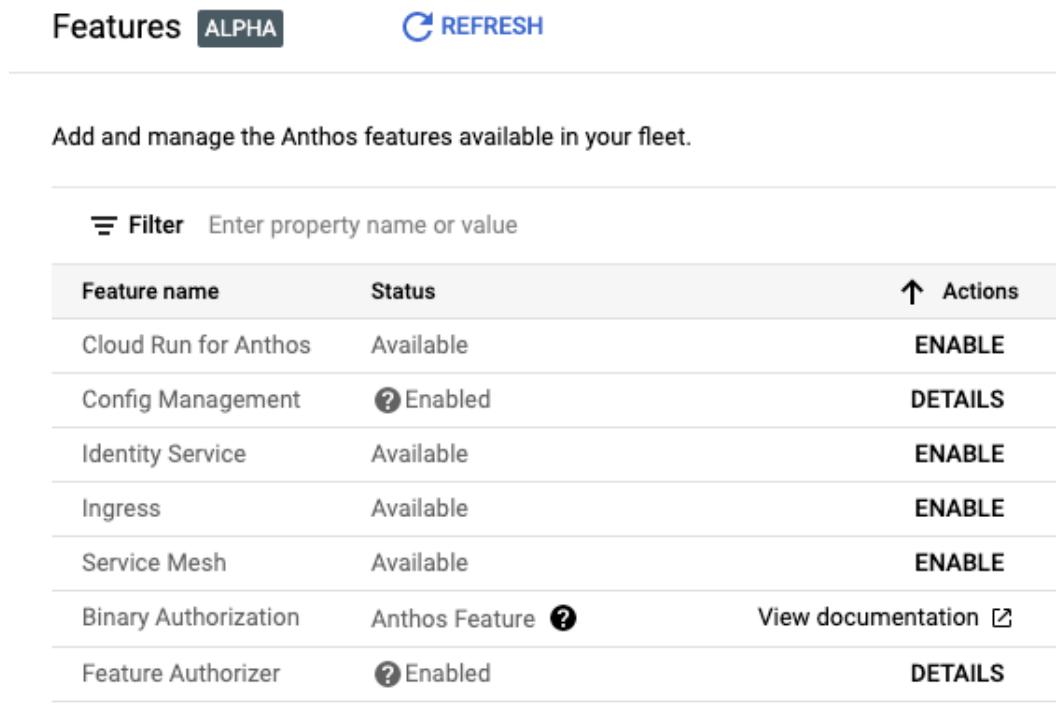
Figure 3.9. Cluster Detail view in the Clusters Menu

3.6.5 Features

The Anthos service encompasses several features, including:

- Configuration Management (Chapter 14)
- Ingress (Chapters 4 and 5)
- Binary Authorization (Chapters 15 and 17)
- Cloud Run for Anthos (Chapter 12)
- Service Mesh (Chapter 5)

The Features menu provides an easy way to enable and disable specific services for the entire Fleet. Figure 3.10 shows the list of existing features for every cluster.



The screenshot shows a user interface for managing Anthos features. At the top, there's a header with 'Features' and a button labeled 'ALPHA'. To the right of the header is a 'REFRESH' button with a circular arrow icon. Below the header, a sub-header reads 'Add and manage the Anthos features available in your fleet.' A 'Filter' input field is present, followed by a table listing seven features:

Feature name	Status	Actions
Cloud Run for Anthos	Available	ENABLE
Config Management	Enabled	DETAILS
Identity Service	Available	ENABLE
Ingress	Available	ENABLE
Service Mesh	Available	ENABLE
Binary Authorization	Anthos Feature	View documentation
Feature Authorizer	Enabled	DETAILS

Figure 3.10. Feature Menu

An admin also has the ability to disable/enable most of these features from the interface (some features are integral components of Anthos and cannot be disabled). It's worth noting that if enablement is not possible fully through the visual interface, the Console generates the right commands for the admin to seamlessly enter them into their CLI.

3.6.6 Migrate to containers

One of the major benefits to Anthos is the automatable migration of Windows and Linux VMs to containers and their deployment onto a compatible Anthos cluster. Previously, this has primarily been done via CLI and initiated from the source cluster, but this menu now provides a convenient, centralized process for shifting VMs to containers and into a different deployment scheme. This menu contains tabs for viewing and managing your Migrations, Sources, and Processing Clusters. For more information on the process of migrating your existing VMs to containers, see Chapter 21: Migrate for Anthos.

3.6.7 Security

The security menu is where we find multiple tools related to viewing, enabling, and auditing the security posture of the clusters in the current Fleet. Figure 3.11 shows the basic view when the Security menu is first selected.

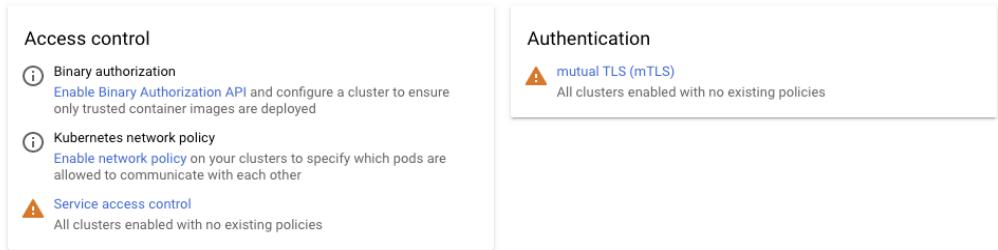


Figure 3.11. Security Menu

As you can see, we do not currently have Binary Authorization¹¹ enabled, but Anthos provides us a shortcut here to quickly turn it on. Once we do, we are presented with the configuration page for Binary Authorization (Figure 3.12) enabling us to view and edit the policy, if needed.

¹¹ Binary Authorization is explored in further detail in Chapter 15: Anthos integration with CI/CD

Binary Authorization

POLICY ATTESTORS

Binary Authorization lets you restrict which images can be deployed to a Kubernetes Engine cluster by making sure they pass through the appropriate checkpoints in your deployment workflow. [Learn more](#)

Policy deployment rules [EDIT POLICY](#)

Project default rule	Allow all images
Cluster-specific rules	us-central1-c.cluster-gcp Allow all images Dry run mode: Disabled
Dry run mode	Not enabled
Images exempt from policy	Google-provided system images VIEW DETAILS gcr.io/google_containers/* gcr.io/google-containers/* k8s.gcr.io/* gke.gcr.io/* gcr.io/stackdriver-agents/*

Figure 3.12. Binary Authorization Policy Details

3.7 Monitoring and Logging

The Anthos menu in the GCP Console is only part of the solution, however. Google also provides the Operations suite, including Cloud Monitoring and Cloud Logging, in order to help with managing the operations of applications and infrastructure. Anthos simplifies the logging of application data and metrics to the Operations suite as part of the default deployment. This can make it simple to add SLOs and SLAs based on these metrics¹². In addition, several pages within the Anthos menu include shortcuts and buttons that trigger wizards to create SLOs in a guided fashion.

3.8 GKE Dashboard

Google has provided the GKE Dashboard for several years to assist with viewing and managing your clusters for GKE in GCP. With the release of Anthos, the GKE Dashboard has been expanded to be able to view the details for Kubernetes clusters attached via GKE

¹² See Chapter 6 for details on SLIs, SLOs, and SLAs with Anthos

Connect. While the Anthos menu is focused on the clusters at a high-level and on the Anthos-specific features, such as the Service Mesh and Configuration Management, the GKE Dashboard allows an admin to drill down to specific workloads and services. Next section is a simple tutorial to register an Azure AKS cluster into Anthos dashboard.

3.9 Connecting to a Remote cluster

In this example, a cluster is already created in the Azure Kubernetes engine (AKS). Google supports below cluster types to be registered remotely, referred to as Attached Clusters:

- Amazon Elastic Kubernetes Service (Amazon EKS) on Kubernetes versions 1.18, 1.19, 1.20
- Microsoft Azure Kubernetes Service (Microsoft AKS) on Kubernetes versions 1.18, 1.19, 1.20
- Red Hat OpenShift Kubernetes Engine (OKE) version 4.6, 4.7
- Red Hat OpenShift Container Platform (OCP) version 4.6, 4.7
- Rancher Kubernetes Engine (RKE) version 1.2.4, 1.2.5, 1.2.6
- KIND version 0.10

Step 1: Use below command to register your cluster:

```
gcloud container hub memberships register MEMBERSHIP_NAME \
--context=KUBECONFIG_CONTEXT \
--kubeconfig=KUBECONFIG_PATH13 \
--service-account-key-file=SERVICE_ACCOUNT_KEY_PATH
```

service account key that you need to create beforehand with the role:

```
--role="roles/gkehub.connect"
```

This command also stores the service account key by creating a secret named “creds-gcp” in the “gke-connect” namespace.

It takes a few minutes and your cluster appears on the GCP console as displayed in Figure 3.13.

¹³ the local file path where your kubeconfig containing an entry for the cluster being registered is stored. This defaults to \$KUBECONFIG if that environment variable is set; otherwise, this defaults to \$HOME/.kube/config.

The screenshot shows the 'Clusters' page in beta. At the top, there are links for 'CREATE CLUSTER' and 'REGISTER EXISTING CLUSTER'. Below this, a red bar highlights the 'Status' section. It displays the following information:

- 1 cluster critical (indicated by a red exclamation mark icon)
- 1 cluster warning (indicated by an orange triangle icon)
- 4 clusters total

Below the status section is a heading 'Anthos managed clusters' followed by a 'Filter table' button. A table lists four clusters with columns for Name, Location, Type, and Labels. The clusters are:

Name	Location	Type	Labels
azure-cluster	registered	Unknown	
cluster-1	registered	Anthos	
cluster-gcp	us-central1-c	GKE	
externalazure	registered	Unknown	

Figure 3.13: Registered cluster view

Step 2: Authenticate to the registered cluster. As you can see, there is a warning sign next to the recently created cluster (externalazure). That is normal and a reminder to sign in to the cluster to be able to perform more operations on the cluster. Figure 3.14 shows the view of the registration status of the cluster.

The screenshot shows a dashboard titled "Kubernetes clusters". At the top, there are buttons for "CREATE", "DEPLOY", "REFRESH", "DELETE", and "REGISTER". Below this is a search bar labeled "Filter table". A table lists four clusters:

	Name	Location	Type	Number of nodes	Total vCPUs	Total memory
!	azure-cluster	registered	Unknown	unknown	unknown ?	unknown ?
✓	cluster-1	registered	Anthos	5	40	157.81 GB
✓	cluster-gcp	us-central1-c	GKE	4	16	64 GB
⚠	externalazure	registered	Unknown	unknown	unknown ?	unknown ?

Figure 3.14: View of the registration status of the cluster

By clicking on the login menu, you can see what login options are available:

- Use your Google identity to log-in
- Token
- Basic authentication
- Authenticate with Identity Provider configured for the cluster

Let's go ahead and authenticate with a token. In order to do that you need to have a KSA with the right permissions¹⁴:

```
KSA_NAME=[KSA_NAME]
kubectl create serviceaccount ${KSA_NAME}
kubectl create clusterrolebinding [VIEW_BINDING_NAME] \
--clusterrole view --serviceaccount default:${KSA_NAME}
kubectl create clusterrolebinding [CLOUD_CONSOLE_READER_BINDING_NAME] \
--clusterrole cloud-console-reader --serviceaccount default:${KSA_NAME}
```

Once you have created the KSA, acquire the KSA's bearer token:

```
SECRET_NAME=$(kubectl get serviceaccount [KSA_NAME] -o jsonpath='{$.secrets[0].name}')
kubectl get secret ${SECRET_NAME} -o jsonpath='{$.data.token}' | base64 --decode
```

Once, you have pasted the token in the login, you immediately get the same view in your AKS cluster (externalazure) that you would see in other cluster types. Figure 3.15 provides that view:

¹⁴All accounts logging in to a cluster need to hold at least the following Kubernetes RBAC roles in the cluster:

View: Kubernetes primitive role that allows read-only access to see most objects in a namespace. It does not allow viewing roles or role bindings.

Cloud-console-reader: Users who want to view your cluster's resources in the console need to have the relevant permissions to do so. You define this set of permissions by creating a ClusterRole RBAC resource, cloud-console-reader, in the cluster. cloud-console-reader grants its users the get, list, and watch permissions on the cluster's nodes, persistent volumes, and storage classes, which allow them to see details about these resources.

	Name	Location	Type	Number of nodes
!	azure-cluster	registered	Unknown	unknown
✓	cluster-1	registered	Anthos	5
✓	cluster-gcp	us-central1-c	GKE	4
✓	externalazure	registered	External	3

Figure 3.15: Anthos attached cluster authenticated

Figure 3.16 shows the nodes and their health status through the dashboard:

Kubernetes cluster details

◀ DETAILS STORAGE NODES

✓ externalazure

Nodes

Filter nodes

Name	Status	CPU requested	CPU allocatable	Memory requested
aks-agentpool-17822882-vmss000000	Ready	845 mCPU	1.9 CPU	1.13 GB
aks-agentpool-17822882-vmss000001	Ready	749 mCPU	1.9 CPU	524.29 MB
aks-agentpool-17822882-vmss000002	Ready	685 mCPU	1.9 CPU	638.58 MB

Figure 3.16: Node view on attached cluster

Several other types of Kubernetes clusters that are not managed by GCP can be attached to Anthos this way. It gives operations simplicity, consistency and access security to administrators from a single platform.

3.10 Summary

- Providing a single pane of glass to hybrid and multi-cloud Kubernetes for any organization who uses microservices is a stepping stone to a successful and global operation.
- One of the biggest benefits to a single pane of glass is that admins can use the same interface to configure service level objectives and alerts to reassure service guarantees.
- Anthos UI provides some major advantages including:
 - Central operation of services and resources
 - Consistent operation experience across multiple service providers
 - Effortless navigation and easy staff training
 - Providing a window to any organizational persona
- Anthos UI provides multiple usages including cluster management, service operation and observability using a unified interface.
- An admin can deploy applications directly from Anthos UI to any registered cluster in a few easy steps.

4

Anthos, the computing environment built on Kubernetes

by Scott Surovich

This chapter covers:

- Understanding Kubernetes architecture, components, and resources
- Declarative application management
- Understanding how workloads are scheduled
- Managing pod placement

Like many new technologies, Kubernetes can be difficult to learn and implement. Creating a cluster manually requires an extensive skill-set that includes public key infrastructure, Kubernetes, Linux, and networking. Many vendors recognized this issue and have automated cluster creation, allowing you to create Kubernetes clusters with little to no Kubernetes background. While automation allows anyone to create a cluster, it also eliminates a lot of Kubernetes knowledge that can help you to troubleshoot issues that you may encounter as a cluster administrator, or a developer, consuming the platform.

The question that comes up is, "Do you really need to know Kubernetes?". The answer is different depending on the role you will play in the cluster, but no matter what role you will have, you will need to have some understanding of how Kubernetes functions. For example, if you are a cluster admin, you should understand how all of the cluster components interact. This understanding will help you to troubleshoot cluster and workload deployment issues. As a developer, you should understand basic Kubernetes operations and the different Kubernetes resources, also referred to as Kubernetes objects, that can be used to deploy

your workloads. It's also important to understand how to force your deployment to a node or a set of nodes by utilizing options like selectors, tolerations, and affinity/anti affinity rules.

In this chapter, you will learn how each component in a Kubernetes cluster interacts with each other. Once you understand the basic interaction, you will learn about the most commonly used Kubernetes resources. Finally, to end the chapter, you will learn the details of how Kubernetes schedules workloads and how to constrain the scheduler to place workloads based on labels, selectors, and affinity/anti-affinity rules.

4.1 Why do you need to understand Kubernetes?

At the heart of Anthos is Kubernetes, which provides the compute engine for applications running in a cluster. Kubernetes is an open-source project created by Google that has been around for years. At the time of this writing, the Cloud Native Computing Foundation (CNCF) has certified 90 Kubernetes offerings. Among the certified offerings are distributions from IBM, Canonical, Suse, Mirantis, VMWare, Rancher, Amazon, Microsoft, and of course, Google.

Most vendor solutions took the most common complaint about adopting Kubernetes, that deploying it was simply "too difficult", and made it easier. While making the installation easier is a necessary step for most enterprises and frees up time to focus on more important activities, it does lead to an issue – not understanding the basic components and resources included in a cluster.

Using a different service example, assume you have an application that requires a new database. You may not have any idea how to create a new database schema, or create SQL queries, but you know that Google offers MySQL and you create a new instance for the application. The MySQL instance will be created automatically and once it has been deployed you can create a database using the GCP console.

Since you may not have a strong SQL background, you may stumble through and create a single table in the database with multiple fields that will work with the application. The database may perform well for a few days or weeks, but as it gets larger, the performance will start to slow down. A single table database, while easy to implement, is not an optimized solution. If you had a SQL background, you would have created a database with multiple tables and relationships, making the database more efficient and scalable.

This scenario is similar to understanding how Kubernetes works and the features provided by the system. In order to use Kubernetes to its full potential, you should understand the underlying architecture and the role of each component. Knowing how components integrate with one another and what resources can be used will help you to make good architectural decisions when deploying a cluster or deploying an application.

The details to cover each cluster component and the more than 60 resource types included with Kubernetes could fill a series of books. Since many of the topics in the chapter will reference resources including Pods, DaemonSets, and more, the chapter will begin with a Kubernetes resource pocket guide, providing a brief definition of the most commonly used API resources.

In this chapter, we will provide a background of Kubernetes components, resources, and commonly used add-on components which provide the compute power that powers Anthos. If you are newer to Kubernetes, there are many books on the market today that explain how to build a cluster, how to use kubectl, and entire chapters devoted to each Kubernetes

resource. This chapter should be viewed as an introduction to resources, with an in-depth focus on how to control the placement of deployments in a cluster.

Technical requirements

The hands-on portion of this chapter will require you to have access to a Kubernetes cluster running in GCP with the following deployment pattern:

- The cluster must be deployed across at least two different zones in the same region.
 - The examples shown in this chapter will be based on **us-east4** zones, across **us-east4-a**, **us-east4-b**, and **us-east4-c**, but you can use different zones for your cluster
- Each zone must contain at least one Kubernetes node

This chapter is not specific to Kubernetes on GCP; the resources and constructs used in the exercises are applicable to any Kubernetes cluster.

4.2 The History of Abstraction

You may be wondering why containers or Kubernetes came about. Questions like what limitation or issue it was designed to address are popular among senior executives in many organizations..

To understand what containers and Kubernetes provide, we need to understand the history of abstraction. Over the years, the industry has attempted to abstract the complexities of the underlying infrastructure. By abstracting certain levels of the infrastructure, like the physical server, the operating system, or even the container, we can provide the agility that today's applications and developers demand.

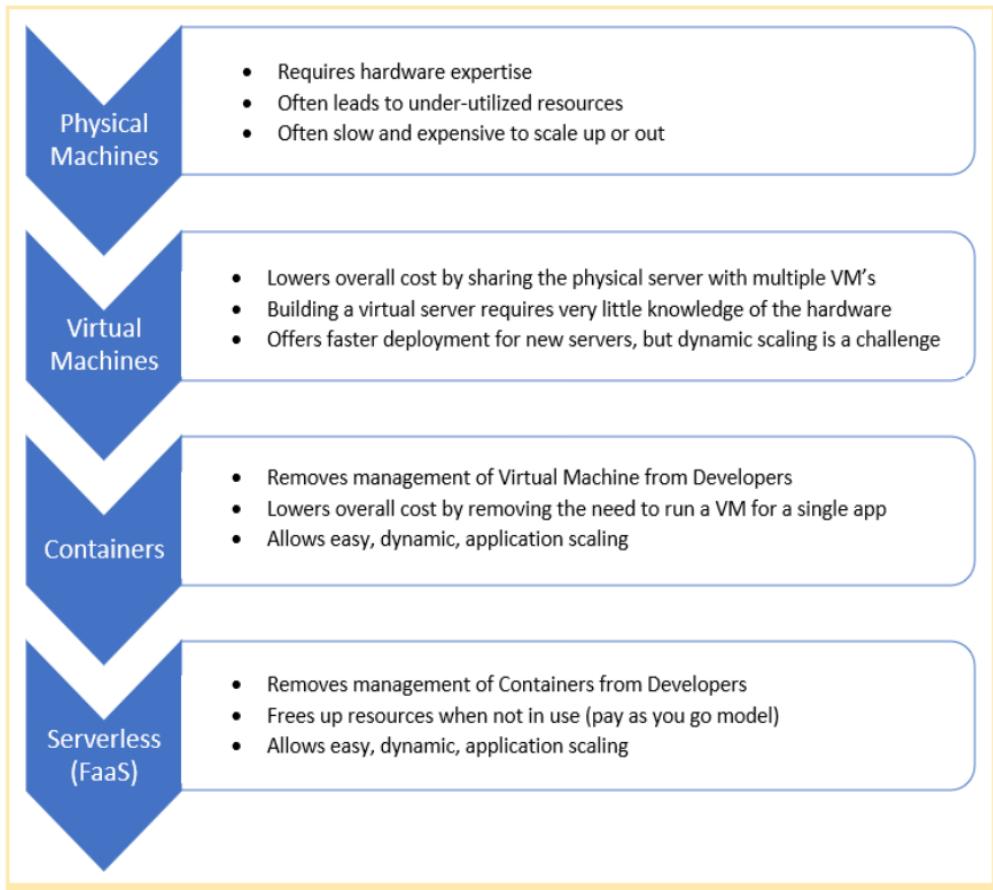


Figure 4.1 - Abstraction history

4.3 Introducing Physical Servers

Not too long ago, data centers were built using physical machines, often running a single application per server. This deployment model was very expensive and required employees with knowledge of the physical hardware, power, cooling, cabling, and more. Since a physical server is one of lowest level components in a datacenter, it didn't offer any abstraction and was difficult to adapt to any changes including adding or removing capacity.

As processors evolved it became common for a CPU to have 4,6,8 or more cores. This often led to wasted resources since the server was running a limited number of processes. To use the new amount of computing power, you needed to run multiple processes on the server, which could lead to a variety of issues including:

- Library / Dependency conflicts between applications

- Difficulty controlling access between multiple teams
- Maintenance windows become difficult to schedule

The industry needed a way to utilize the new power in servers, and virtualization came to the rescue.

4.3.1 Introducing Virtual Machines

A virtual server is unique and separate from all other virtual machines running on the physical server, but you don't own or maintain the physical server and in most cases, you don't want to.

With all of the compute power in a physical server, we needed a way to efficiently and safely utilize the resources. As mentioned in the previous section, you could attempt to run multiple processes on a single operating system to increase utilization, but since all of the applications would be running on a single operating system instance, all of the processes share a single instance of the kernel, modules, libraries, etc... This sharing often led to conflicts like Java versions, DLL versions, module versions, and more.

To address these issues, virtualization was born. A virtual machine abstracts the physical server hardware, creating a server that runs inside a Hypervisor that has a defined set of virtual hardware and drivers - abstracting the hardware that is in the physical server. The Hypervisor layer is responsible for communication between the virtual machines and the physical hardware in the server. With the introduction of virtual machines, organizations were able to:

- Increase utilization of the physical servers
- Eliminate worries about driver issues for different server models or vendors
- Easily build a new server for an application from a template
- Create separate virtual machines based on developer requirements or security requirements

Virtualization quickly became a standard in most organizations, and is still considered to be the standard for new deployments in many companies. While still popular today, it has some limitations, including:

- Additional cost of each virtual machine operating system
- Server sprawl - requiring patching and staff to manage each instance
- Difficulty of scaling an application instance out
- Installation of most software requires elevated privileges to be granted to multiple users

These limitations drove people to the next level of abstraction, containers.

4.3.2 Introducing Containers

Just like a virtual machine is nothing more than a server running on someone else's hardware, a container is just an application running on someone else's virtual or physical server. The container is isolated from all other containers running on the server, and you do not need to know details about the server or the host operating system.

The development community needed an infrastructure that offered agility and speed. Developers wanted, and needed, the ability to:

- Run multiple versions of an application, quickly and easily, without building a new server for each version
- Scale a deployment up when additional resources are required, without waiting for additional virtual machines to be provisioned
- Easily rollback a deployment
- Have more autonomy and control over their own resources

To address these requirements, the operating system was abstracted from the developer. By removing the operating system, the developer no longer needs to worry about conflicting libraries, or missing libraries on a server. Each container runs in its own namespace, keeping it separate from all other running containers, sharing nothing with other containers outside of the host kernel.

NOTE: There are projects to remove the sharing of the kernel between running containers, offering an additional security layer. Google has created gVisor, which provides an application kernel for containers. You can read more about the project on their Github page at <https://github.com/google/gvisor>

Since containers abstract the operating system, developers no longer need to worry about the host or dependency issues. This means they can spend more time developing rather than troubleshooting issues or learning different commands and utilities for each operating system that they may develop on.

This sounds very attractive to most people, and it should since it offers developers a set of features that was difficult to offer with virtual or physical servers. While the abstraction removes any operating system knowledge or requirements, it doesn't necessarily mean that a developer doesn't need to concern themselves with other requirements. Rather than learn any operating system command, utilities, etc... a developer now needs to know how to:

- Create a container image
- Maintain different container images
- Use container registries
- Understand the container runtime that is used
- Understand the container orchestrator, e.g. Kubernetes
- Create configuration files for application deployments

Containers ushered in a new age of development, and while removing one requirement in the operating system, it introduces a new collection of tools that developers need to learn and understand. This should not be viewed as a negative towards container adoption, the knowledge that is required to use containers efficiently offers advantages over learning or using legacy deployment models. Learning about containers and Kubernetes offers the following advantages:

- Portability - Containers can run on any compatible system without any changes
- Additional Security - Container can be forced to always run as a non-root account and any unnecessary modules/utilities should not be included in the image

- Common Commands - All Kubernetes commands and (most) resources are portable across different providers

Still, there are some developers that don't want to deal with anything other than their application - this introduces the next deployment type, serverless.

4.3.3 Introduction to Serverless

You can think of a serverless application as an application running in someone else's container.

Serverless abstracts the container from the developer, allowing developers to focus on their code and image, rather than trying to decide what Kubernetes resources need to be created to run the application.

NOTE: Contrary to what the term serverless implies, serverless workloads still require a server to run the application. The "serverless" application will still run in a container, running on a physical or virtual server.

This abstraction allows a developer to use all of the features that containers provide, including scaling, security, and faster development - all without **any** knowledge of the underlying infrastructure.

Google provides serverless features as part of Cloud Run for Anthos. You can read more about Cloud Run for Anthos at <https://cloud.google.com/run/docs/qke/setup>

Now that we have covered the evolution of abstraction, we can dive into Kubernetes, the container orchestration tool that Anthos uses to provide compute to applications.

4.4 Introducing Kubernetes

The classic definition of Kubernetes is "a container orchestration tool". While this definition is correct, it doesn't consider all of the features that Kubernetes provides. It has evolved beyond being a simple orchestration tool, and as it matures more people define it as an application platform, or an application infrastructure.

Kubernetes provides automation for deployment, scaling, and management of containerized applications. By taking a group of hosts and managing them as a single unit, or cluster, Kubernetes schedules workloads across all of the hosts in an efficient manner.

4.4.1 Addressing Kubernetes Gaps

A base Kubernetes cluster will provide you with a basic cluster to run applications. Most default installations will include the control plane, worker nodes, networking and DNS for cluster name resolution. These provide the minimum components that allow you to deploy applications, but they don't provide other key components to make a cluster "production like", including:

- Enterprise authentication
- Ingress controller
- Logging
- Monitoring

- Backup and restore
- Service mesh
- Configuration management
- Centralized cluster view

Organizations may decide to use multiple vendors to address the components in the list, including open source projects and vendor solutions.

For example, you have decided to use open source projects to address the features that are not part of a base cluster. To address logging and monitoring, you select the EFK stack (Elasticsearch, FluentD, and Kibana) stack. You also decide to deploy NGINX as your Ingress controller, Dex as your identity provider, Istio as your service mesh, ArgoCD for configuration management, and Velero as your backup and restore solution.

Since each solution is a unique product, you will find yourself using multiple Github repositories or Slack channels for support. This type of support model may be acceptable for some organizations, but most enterprises require faster problem resolution than what GIT issues or chats can provide.

This is where Anthos comes in! It addresses the gaps in Kubernetes by including components to address each requirement, providing a managed platform from Google. Anthos extends Google's services and engineering practices to your organization, allowing you to modernize applications faster with operational consistency across GCP services and your own environments.

Anthos Feature	Description
Anthos Connect	Connects clusters to GCP, simplifying connectivity, authentication, and authorization of clusters
Anthos Config Manager (ACM)	Provides configuration and policy management
Anthos Service Mesh (ASM)	Service mesh based on Istio
Cloud Operations for GKE	Logging and monitoring
Cloud Run	Serverless workloads
Istio-Ingress	Ingress controller

Since all of the components are included with Anthos, you don't need to use different vendors to address each component, and all features are supported by Google, providing a single point for support and escalation. Of course, since Anthos runs on a standard Kubernetes platform, you can replace or add any components that you require, for example: if you need to use an NGINX Ingress controller instead of the included Istio-Ingress, you can remove the included controller and replace it with NGINX.

Kubernetes does not include federation or consolidation of multiple clusters, which often leads to confusion as an organization's cluster count increases. Management of clusters that are not only geographically dispersed on-premise, but off-premise, provides a unique challenge to an organization. Like many other gaps that are not native to Kubernetes, Anthos provides a solution to this issue by including Anthos Connect Agent, allowing an organization to collate clusters from on-premise and off-premise into the GCP console. This is also a key component of Google's Fleet management, simplifying and enhancing multi-cluster management. You can read more about Fleet on Google's Fleet page at <https://cloud.google.com/anthos/multicluster-management>

4.4.2 Managing On-Prem and Off-Prem Clusters

When a company decides to run a Kubernetes cluster in the cloud, they often use the Cloud provider's native offering, including:

- Google Kubernetes Engine (GKE) <https://cloud.google.com/kubernetes-engine/>
- Amazon Elastic Kubernetes Service (EKS) <https://aws.amazon.com/eks/>
- Azure Kubernetes Service (AKS) <https://azure.microsoft.com/en-us/services/kubernetes-service/>

Using the native offering offers the quickest and easiest way to get a new cluster up and running, since the providers have automated the installation. To get from ground zero to a running cluster, you only need to provide a few pieces of information like the number and size of the nodes, zones, and regions. With this information and a click, or API call, you can have a cluster in a few minutes, ready to deploy your applications.

Google was the first Cloud Service Provider to offer their Kubernetes solution across both the Cloud and on-prem, without requiring any specialized hardware solution. Before Google did this, other offerings required organizations to deploy a different solution for each Cloud Provider and their on-premise clusters. Using a different solution for multiple installations often lead to a variety of different issues, including:

- Increased staff to support each deployment
- Differences in the deployment of an application for on-prem and off-prem
- Different identity management solutions
- Different Kubernetes versions
- Different security models
- Difficulty in standardizing cluster operations
- No single view for all clusters

Each of these differences makes the job of running Kubernetes more difficult and ultimately more costly for an organization.

Google recognized these issues and created Anthos, which addresses the on-prem and off-prem challenges by providing a Kubernetes installation and management solution that not only works on GCP and on-premise clusters, but in other Cloud providers like AWS and Azure running Anthos.

Using Anthos provides a common environment no matter where you deploy it. Imagine having a single support path and a common set of tools for all of your clusters in GCP, AWS, Azure, and on-prem. Anthos provides an organization with many advantages, including:

- A consolidated view of clusters inside the Anthos console
- A common service mesh offering
- Configuration management using ACM
- All options supported by Google - A single point of contact for all cluster components

Best of all, Anthos is based on the upstream Kubernetes - so you get all of the standard features but with the added tools and components that Anthos provides, making multiple cloud cluster management easier.

Next, we will jump into the architecture that makes up a Kubernetes cluster and how they communicate with each other.

4.5 Kubernetes architecture

Like any infrastructure, Kubernetes is made up of multiple components that communicate to create a cluster. The components are grouped together into two layers, the control plane and the worker nodes. The control plane is in charge of keeping cluster state, accepting incoming requests, scheduling workloads, and running controllers. While the worker nodes communicate with the control plane to report available resources, run container workloads, and maintain node network rules.

If you are running Anthos on GCP, you may not be familiar with the components of the control plane or the worker nodes, since you do not interact with them like you would with an on-premise installation. As this section will explain, Kubernetes clusters have a layer called the control plane that contains the components required to run Kubernetes. When a cluster is running in GCP the control plane is created in a Google managed project, which limits you from interacting with the admin nodes and the Kubernetes components.

All GKE clusters can be viewed in your GCP console, located under the Kubernetes Engine section. For each cluster you can view the details of the nodes by clicking on the cluster in the details pane, then selecting nodes. The node details will be displayed, as shown in the Figure 4.2 - GKE node details.

The screenshot shows the GCP Kubernetes Engine interface. On the left, there's a sidebar with icons for Clusters, Workloads, Services & Ingress, Applications, Configuration, Storage, Object Browser, and Migrate to containers. The 'Clusters' icon is selected. The main area shows a list of clusters with 'anthos-1' selected. Below the cluster list, there are tabs for Details, Storage, and Nodes, with 'Nodes' being the active tab. A 'Filter nodes' input field is present. A 'Columns' dropdown menu is also visible. The main table displays three node details:

Name	Status	CPU requested	CPU allocatable	Memory requested	Memory allocatable	Storage requested	Storage allocatable
gke-anthos-1-default-pool-bb1016e5-f166	Ready	203 mCPU	940 mCPU	262.14 MB	2.77 GB	0 B	0 B
gke-anthos-1-default-pool-bb1016e5-h2dj	Ready	463 mCPU	940 mCPU	377.49 MB	2.77 GB	0 B	0 B
gke-anthos-1-default-pool-bb1016e5-hkdp	Ready	639 mCPU	940 mCPU	730.86 MB	2.77 GB	0 B	0 B

Figure 4.2 - GKE node details

Unlike GKE on GCP, an on-prem installation of GKE provides access to the control plane nodes and Kubernetes resources for the clusters. Of course, Google still supports the on-premise control plane, but you may be asked to look at components to troubleshoot any issues or configuration changes to a cluster. If you have only deployed GKE on GCP, you may not know how all of the components of the control plane and how they interact. Understanding this interaction is vital to troubleshooting any issues and finding root causes to any issues.

NOTE: When you deploy a GKE on-prem cluster, three Kubernetes config files are created. One will be named using the user cluster's name with a suffix of -kubeconfig, one called kubeconfig, and the last one is called internal-cluster-kubeconfig-debug. The kubeconfig file is configured to target the load-balanced address of the admin cluster, while the internal-cluster-kubeconfig-debug is configured to target the admin cluster's API server directly.

To explain the multiple configuration files, reference **Figure 4.3 - GKE on-prem admin cluster architecture**.

GKE On-Prem Admin Cluster (Control Plane)

When a user cluster is added to an Admin cluster, a new namespace is created using the name that was given to the user cluster. In this example, the admin cluster is managing two user clusters, cluster-001 and cluster-002. Each namespace contains the Control Plane components for the user cluster.

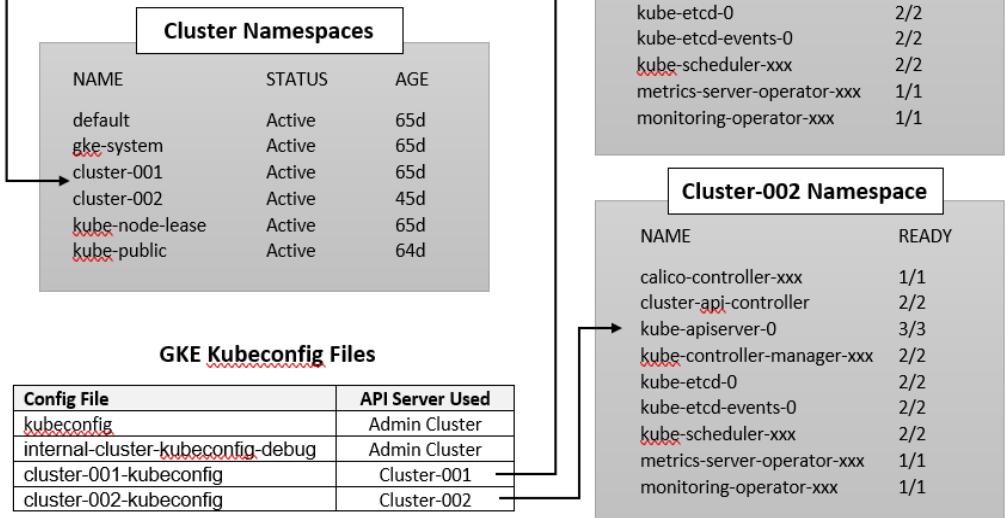


Figure 4.3 - Admin cluster and user cluster configuration files

With the importance of understanding the system as a whole, let's move on to each layer in a cluster, starting with the control plane.

4.5.1 Understanding the Cluster Layers

The first layer, the control plane, contains five or six components (in reality, the two controllers actually contain multiple components). The control plane includes the components that provide cluster management, cluster state, and scheduling features. We will detail each component in the next section, but for now, we just want to introduce the control plane components:

- ETCD
- The Kubernetes API server
- The Kubernetes scheduler
- The Kubernetes controller manager, which contains multiple controllers:
 - Node controller

- Endpoint controller
- Replication controller
- Service account / token controller
- The cloud controller manager, which contains multiple controllers:
 - Route controller
 - Service controller

To show a graphical representation of the control plane, reference the diagram below. At the end of this section we will provide a complete component diagram including how each component communicates.

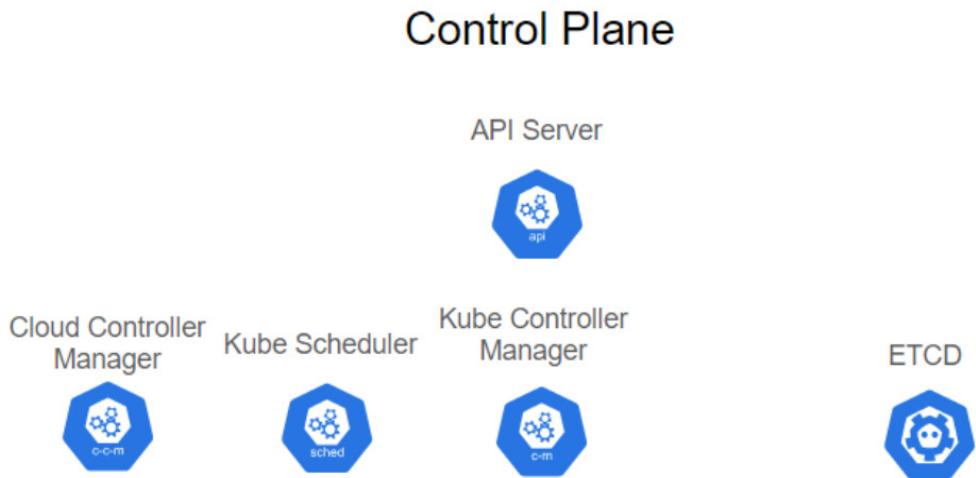


Figure 4.4 - Control plane components

The second layer in the cluster is the collection of worker nodes, which are responsible for running the cluster workloads. Each worker node has three components that work together to run applications:

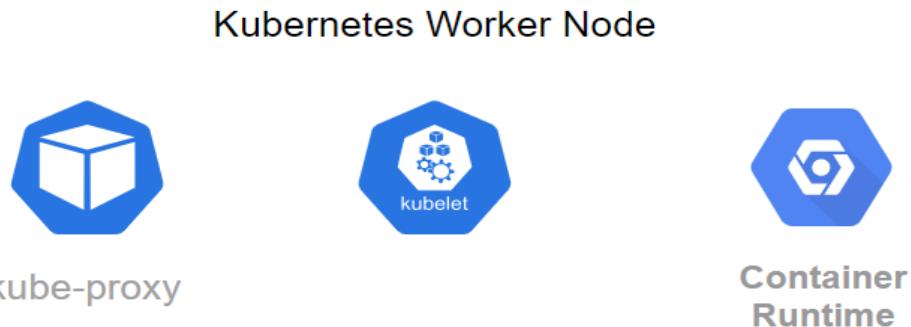


Figure 4.5 – Worker node components

Up to this point, we haven't explained how each component interacts with the others. Before we show a full diagram of cluster interactions, we need to understand each component in the cluster. In the next section, we will explain each cluster component and to close out the section, we will combine the two diagrams to show the connectivity between all components.

4.5.2 The Control Plane Components

As mentioned in the introduction, the control plane includes up to six components. Each of the components work together to provide cluster services. Understanding each component is key to delivering a robust, stable cluster.

ETCD

Every resource in the cluster and its state are maintained in the ETCD key/value database. The entire cluster state is stored inside this database, making ETCD the most important component in a cluster. Without a functioning ETCD database, you do not have a functioning cluster.

Since ETCD is so important, you should always have at least three replicas running in a cluster. Depending on the size of the cluster, you may want to have more than three – but no matter how many you decide to run, always run an odd number of replicas. Running an odd number of ETCD nodes allows the cluster to elect a majority leader, minimizing the chance of the ETCD cluster going into a split-brain state. If a cluster goes into a split-brain state, more than one node claims to be the majority leader, which leads to data inconsistencies and corruption. If you find yourself in a split-brain state, you will need to recreate the ETCD cluster from an ETCD backup.

While running multiple copies will make ETCD highly available, you also need to create a regular backup of your database and store it outside of the cluster in a safe location. If you lose your entire cluster or your ETCD database goes corrupt, you will be able to restore your backup to restore a node or the entire cluster. We will explain the process to backup ETCD later in this chapter.

The last consideration for ETCD after making it highly available and creating regular backups is to consider security. Since the ETCD database contains every Kubernetes

resource, it will contain sensitive data like secrets, which may contain data like passwords. If someone gets a copy of your ETCD database, they can easily pull any of the resources out since, by default, they are stored in clear text.

ETCD can be an entire chapter by itself, for more information on ETCD you can head over to the main ETCD site at <https://etcd.io/docs/> Google also provides the steps and a script to backup GKE on-prem clusters, you can find the documentation and the script at <https://cloud.google.com/anthos/gke/docs/on-prem/how-to/backing-up>.

THE KUBERNETES API SERVER

The API server is the front door to a cluster. All requests that come into the cluster enter through the API server which will interact with the other component to fulfill requests. These requests come from users and services from the kubectl CLI, Kubernetes dashboard, or direct JSON API calls.

It's really an event-driven hub-and-spoke model. The apiserver encapsulates etcd. All other components communicate with apiserver. The apiserver doesn't communicate with controllers directly in response to requests. Instead, the controllers watch for relevant change events.

THE KUBERNETES SCHEDULER

If the API server receives a request to create a pod, it will communicate with the Kubernetes scheduler, which decides which worker node will run the workload.

When a workload attempts to request a resource that cannot be met, or has constraints that cannot be matched, it will fail to schedule and the pod will not start. If this happens, you will need to find out why the scheduling failed and either change your deployment manifest or add resources to your nodes to fulfill the request.

THE KUBERNETES CONTROLLER MANAGER

The controller manager is often referred to as a *control-loop*. To allow Kubernetes to keep all resources in a requested, desired state, the state of each resource must be compared to its requested state. The process that makes this happen is known as a *control-loop*.

The Kubernetes controller manager consists of a single binary that runs separate threads for each "logical" controller. The bundled controllers and their role are:

Controller	Description
Node	Maintains the status of all nodes.
Replication	Maintains the number of pods for replication controllers.
Endpoint	Maintains the mapping of pods to services, creating endpoints for services.
Service accounts / Token	Creates the initial default account and API tokens for namespaces.

The main concept to take away from the table is that by using a control-loop the manager constantly checks the resource(s) that it controls to keep them in the declared state.

The Kubernetes controller manager deals with internal Kubernetes resource states. If you are using a cloud provider, your cluster will need a controller to maintain certain resources, which is the role of the cloud controller manager.

THE CLOUD CONTROLLER MANAGER

NOTE: You may not see this controller on every cluster you interact with. A cluster will only run a cloud controller if it has been configured to interface with a cloud provider.

To allow cloud providers flexibility, the cloud controller manager is separate from the standard Kubernetes controller manager. By decoupling the two controllers, each cloud provider can add features to their offering that may differ from other providers or base Kubernetes components.

Similar to the Kubernetes controller manager, the cloud controller manager uses a control-loop to maintain the desired state of resources. It is also a single binary that runs multiple controllers in their process:

Controller	Description
Node	Creates node resources and maintains the status of the nodes located in the cloud provider.
Route	Maintains network routes to provide node communication.
Service	Maintains cloud provider components like load-balancers, network filtering, and IP addresses.

Finally, when we say cloud provider, you are not limited to only public cloud service providers. At the time of this writing, Kubernetes includes support for the following cloud providers:

- Amazon AWS
- Microsoft Azure
- Google Cloud Platform (GCP)
- Openstack
- Huawei
- vSphere

Now that the control plane has been explained, let's move on to the worker node components.

4.5.3 Worker node components

From a high level, you should have a basic understanding of the components in the control plane, it's the layer that is responsible for cluster interaction and workload deployments. Alone, the control plane can't do very much, it needs to have a target that can run the actual workload once it's scheduled and that's where the worker node comes in.

THE KUBELET

The Kubelet is the component that is responsible for running a pod and for reporting the node's status to the Kubernetes scheduler.

When the scheduler decides the node that will run a workload, the Kubelet retrieves it from the apiserver, and the pod is created based on the specs that were pulled.

KUBE-PROXY

We will mention this in more detail when we discuss services in the next section, but for now you only need to understand a basic overview of kube-proxy.

Kube-proxy is responsible for creating and deleting network rules, providing the rules that allow network connectivity to a pod. If the host operating system offers a packet filter,

kube-proxy will leverage it, but if no packet filter is offered, the traffic will be managed by kube-proxy itself.

CONTAINER RUNTIME

The container runtime is the component that is responsible for running the actual container on the host.

It has become common for people to refer to the container runtime as simply, Docker. This is understandable since Docker did bring containers to the masses, but over the years other alternatives have been developed, two of the most popular alternatives are CRI-O and ContainerD.

At one time, the container runtime was integrated as part of the Kubelet, which made adding a new runtime difficult. As Kubernetes matured, the team developed the Container Runtime Interface (CRI), which provides the ability to simply “plug-in” a container runtime.

No matter which runtime is in use, its responsibility is the same, to run the actual container on the node.

Now that we have reviewed each layer and their components, let’s show the connectivity between the two layers and how the components interact.

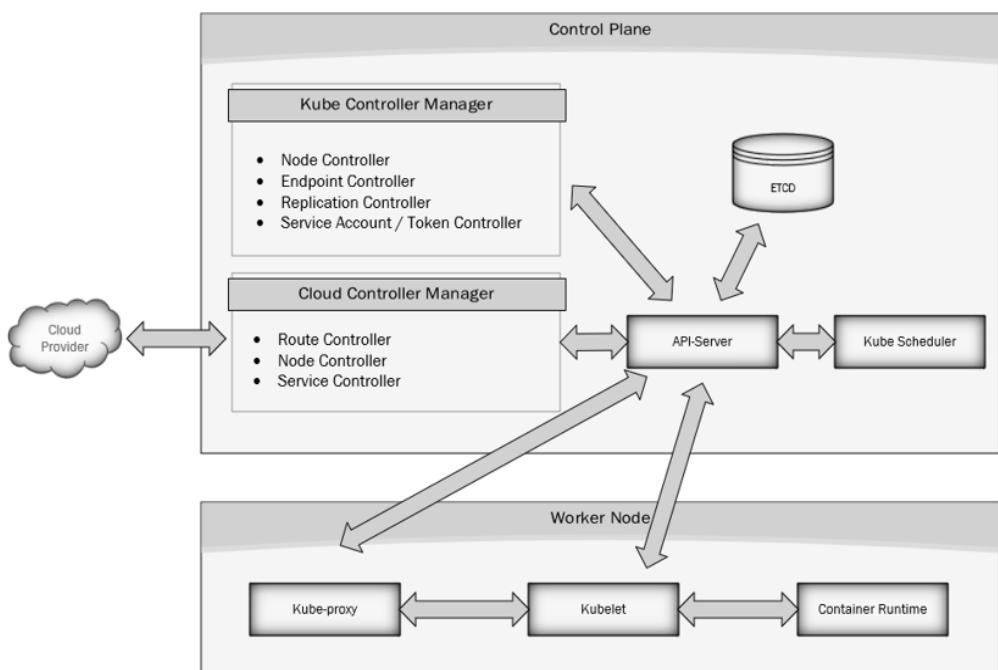


Figure 4.6 - Cluster component communications

That concludes the section on Kubernetes cluster components. Knowing how the components interact will help you to diagnose issues and to understand how the cluster interacts as a system.

Depending on your role, the cluster components and how they interact, may be less important than understanding cluster resources. Kubernetes resources are used by every user that interacts with a cluster and users should understand, at the very least, the most commonly used resources. For reference, you can read about Kubernetes resources on the Kubernetes website at <https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/>

To effectively deploy an application on Kubernetes, you need to understand the features of the infrastructure, starting with Kubernetes objects.

Next, we will move into DevOps paradigms and Kubernetes cluster components.

4.5.4 Understanding Declarative and Imperative

In DevOps, an automation framework can use two different implementation methods. These methods are referred to as DevOps paradigms and they include the declarative model and the imperative model.

Each of the paradigms will be explained in this chapter, but before diving into the differences between them, you should understand the concept of a control loop.

UNDERSTANDING CONTROL LOOPS

To maintain your desired state, Kubernetes implements a set of control loops. A control loop is an endless loop that is always checking the declared state of a resource to its current state.

If you declare that a Deployment should have 3 replicas of a pod, and one pod is deleted accidentally, Kubernetes will create a new pod to keep the states in sync.

Let's show a graphical representation of the ReplicaSet control loop and how it maintains the desired replica count:

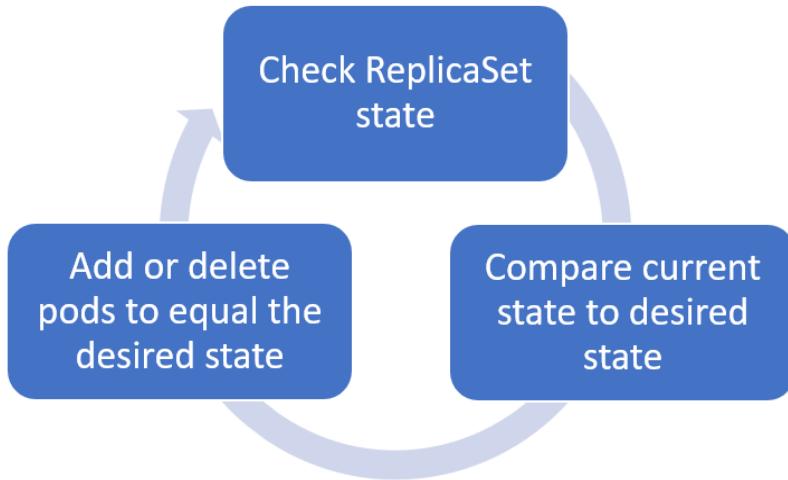


Figure 4.7 – Control loop example

As you can see in the diagram, a control loop doesn't need to be complex to maintain a desired state. The replication controller simply keeps looping through all of the ReplicaSet resources in the cluster, comparing the currently available number of pods to the desired number of pods that is declared. This means that Kubernetes will either add or delete a pod to make the current replica count equal to the count that has been set on the Deployment.

Understanding the features Anthos includes and how Kubernetes maintains the declared state of a deployment are important for any user of Kubernetes, but it's only the beginning. Since deploying a cluster has been made so simple by many vendors, developers and administrators often overlook the advantages of understanding the entire system. As mentioned earlier in the chapter, to design an effective cluster or application, you should understand the basic functionality of the cluster components. In the next section, the Kubernetes architecture will be covered, including the components of the control plane and worker nodes and how they interact with each other.

One of the first concepts to understand is the difference between the declarative and imperative models. The table below provides a brief description of each model.

Model	Description
<i>Declarative</i>	Developers declare what they would like the system to do, there is no need to tell the system how to do it. The declarative model uses Kubernetes manifests to declare the application's desired state.
<i>Imperative</i>	Developers are responsible for creating each step that is required for the desired end-state. The steps to create the deployment are completely defined by the developer. The imperative model uses kubectl commands like create, run, and delete to tell the API server what resources to manage.

In a declarative model, you can manage any number of resources in a single file. For example, if we wanted to deploy a NGINX web server that included a new namespace, the deployment, and a service, we would create a single YAML file with all of the resources. The manifest would then be deployed using the kubectl apply command, which will create each resource and add an annotation that includes the last-applied-configuration. Since Kubernetes tracks the resources and you have all of the resources in a single file, it becomes easier to manage and track changes to the deployment and resources.

In an imperative model, you must run multiple commands to create your final deployment. Using the previous example where you want to deploy a NGINX server, a service, and an Ingress rule, you would need to execute three different kubectl commands.

```
kubectl create ns web-example
kubectl run nginx-web --image=nginx:v1 -n web-example
kubectl create service clusterip nginx-web -tcp=80:80
```

While this would accomplish the same deployment as our declarative example, it has some limitations that are not immediately noticeable using our simple example. One limitation is that the kubectl command does not allow you to configure every option that may be available for each resource. In the example, we deploy a pod with a single container running NGINX. If we needed to add a second container to perform a specialized task, like logging, we wouldn't be able to add it imperatively since the kubectl command does not have the option to launch two containers in a pod.

It is a good practice to avoid using imperative deployments unless you are attempting to resolve an issue quickly. If you find yourself using imperative commands for any reason, you should keep track of your changes so that you can change your declarative manifests to keep them in sync with any changes.

To understand how Kubernetes uses the declarative model, you need to understand how the system maintains the declared state with the currently running state for a deployment by using control loop

4.5.5 Understanding Kubernetes resources

Throughout this book, you will see references to multiple Kubernetes resources. As mentioned earlier in the chapter, there are more than 60 resource types included with a new cluster, and this doesn't include any custom resources that may be added through CRD's (Custom Resource Definitions). Since there are multiple Kubernetes books available, this chapter will only provide an introduction to each resource to provide a base knowledge that will be used in most of the chapters.

It's challenging to remember all of the base resources and you may not always have a pocket guide available to you. Luckily, there are a few commands that you can use to look up resources and the options that are available for each.

The first command will list all of the api resources available on a cluster.

kubectl api-resources				
NAME	KIND	SHORTNAMES	APIVERSION	NAMESPACED
bindings	Binding		v1	true
componentstatuses	ComponentStatus	cs	v1	false
configmaps	ConfigMap	cm	v1	true
endpoints	Endpoints	ep	v1	true
events	Event	ev	v1	true
limitranges	LimitRange	limits	v1	true
namespaces	Namespace	ns	v1	false
nodes	Node	no	v1	false
persistentvolumeclaims	PersistentVolumeClaim	pvc	v1	true
persistentvolumes	PersistentVolume	pv	v1	false
pods	Pod	po	v1	true
podtemplates	PodTemplate		v1	true
replicationcontrollers	ReplicationController	rc	v1	true
resourcequotas	ResourceQuota	quota	v1	true
secrets	Secret		v1	
serviceaccounts	ServiceAccount	sa	v1	true
services	Service	svc	v1	true

The output provides the name of the resource, any short name, if it can be used at a namespace level, and the kind of resource. This is helpful if you know what each one does and you only forgot the name of a resource or if it can be set at a namespace level. If you

need additional information for any resource, Kubernetes provides another command that provides the details for each one.

```
kubectl explain <resource name>
```

The explain command will provide a short description of the resource and all of the fields that can be used in a manifest. For example, in Figure 4.2, you will see a brief description of what a pod is, and some of the fields that can be used when creating the resource.

```
KIND: Pod
VERSION: v1

DESCRIPTION:
Pod is a collection of containers that can run on a host. This resource is created by clients and scheduled onto hosts.

FIELDS:
apiVersion <string>
APIVersion defines the versioned schema of this representation of an object. Servers should convert recognized schemas to the latest internal value, and may reject unrecognized values. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources

kind <string>
Kind is a string value representing the REST resource this object represents. Servers may infer this from the endpoint the client submits requests to. Cannot be updated. In CamelCase. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds

metadata <Object>
Standard object's metadata. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata
```

As you can see from the output, each field has a detailed explanation and a link to provide additional detailed information, when applicable.

You may not have access to a system with kubectl installed all the time, so the table below provides a short description of most of the common resources you will use in a cluster.

Kubernetes Resource	Description
ConfigMap	Holds configuration data for pods.
EndpointSlice	A collection of pods that are used as targets by services.
Namespace	Used to divide clusters between multiple developers or applications.
Node	Provides the compute power to a Kubernetes cluster.

PersistentVolumeClaim	Allows an application to claim a persistent volume.
PersistentVolume	A storage resource provisioned at the cluster layer. Claims to PersistentVolume are provided by a PersistentVolumeClaim.
Pod	A container or a collection of containers.
ResourceQuota	Sets quota restrictions, enforced per namespace.
Secret	Holds secret data of a certain type. The total bytes of the values in the Data field must be less than MaxSecretSize bytes configuration value.
ServiceAccount	Provides an identity that can be authenticated and authorized to resources in a cluster.
Service	Provides a named abstraction of software service consisting of a local port that the proxy listens on, and the selector that determines which pods will answer requests sent through the proxy.
CustomResourceDefinition	Represents a resource that should be exposed on the API server.
DaemonSet	Used to deploy a container to all nodes, or a subset of nodes, in the cluster. This includes any new nodes that may be added after the initial deployment.
Deployment	Enables declarative updates for Pods and ReplicaSets.
ReplicaSet	Ensures that a specified number of pod replicas are running at any given time.
StatefulSet	StatefulSet represents a set of pods with consistent identities and controlled pod starting and stopping.
Ingress	A collection of rules that direct inbound connections to reach the pod endpoints.
NetworkPolicy	Defines what network traffic is allowed for a set of Pods.
PodSecurityPolicy	Controls the ability to make requests that affect the Security Context that will be applied to a pod and container.

ClusterRole	A cluster-level, logical grouping of PolicyRules that can be referenced as a unit by a RoleBinding or ClusterRoleBinding.
ClusterRoleBinding	Assigns the permissions defined in a ClusterRole to a user, group, or service account. The scope of a ClusterRoleBinding is cluster wide.
Role	A namespaced, logical grouping of PolicyRules that can be referenced as a unit by a RoleBinding.
RoleBinding	Assigns the permissions defined in a Role to a user, group, or service account. It can reference a Role in the same namespace or a ClusterRole in the global namespace. The scope of a RoleBinding is only to the namespace it is defined in.
StorageClass	Describes the parameters for a class of storage for which PersistentVolumes can be dynamically provisioned.

Understanding the resources that are available is one of the keys to creating the best application deployments, and to helping troubleshoot cluster or deployment issues. Without an understanding of these resources, you may not know what to look at if an Ingress rule isn't working as expected. Using the resources in the table, you can find three resources that are required for an Ingress rule. The first is the Ingress itself, the second is the Service, and the last is the Endpoints/EndpointSlices.

Looking at the flow between resources for Ingress, an incoming request is evaluated by the Ingress controller and a matching Ingress resource is found. Ingress rules route traffic based on the Service name defined in the Ingress rule, and finally the request is sent to a pod from the EndPoints created by the Service.

4.5.6 Kubernetes Resources In Depth

A brief overview of resources and what they are used for is a great refresher, if you have experience with resources already. We realize that not every reader will have years of experience interacting with Kubernetes resources, so in this section you will find additional details on some of the most commonly used cluster resources.

One thing that all GKE Kubernetes clusters have in common, on-premise or off-premise, is that they are built on the upstream Kubernetes code and they all contain the base set of Kubernetes resources. Interacting with these base types is something you are likely to do on a daily basis, and having a strong understanding of each component, its function, and use-case examples is important.

NAMESPACES

Namespaces provide a scope for names - Names of resources need to be unique within a namespace, but not across namespaces.

Namespaces create a logical separation between tenants in the cluster, providing a cluster with multi-tenancy. As defined by Gartner, "Multitenancy is a reference to the mode of operation of software where multiple independent instances of one or multiple applications operate in a shared environment. The instances (tenants) are logically isolated, but physically integrated" (<https://www.gartner.com/en/information-technology/glossary/multitenancy#:~:text=Multitenancy%20is%20a%20reference%20to,logically%20isolated%2C%20but%20physically%20integrated>)

Kubernetes resources that are created at a namespace level, are referred to as being **namespaced**. If you read that a resource is namespaced, it means that the resource is managed at a namespace level, rather than a cluster level.

In a namespace you can create resources that will provide security and resource limits. To provide a safe multi-tenant cluster, you can use the following categories of Kubernetes resources:

- RBAC
- Resource quotas
- Network policies
- Namespace security resources (previously Pod Security Policies)

We will discuss each of the resources in more detail in this section, but for now you only need to understand that a namespace is a logical partition of a cluster.

Namespaces are also used when you create a service. When you create a service, which we will cover in the services section, it is assigned a DNS name that includes the service name and the namespace. For example, if you created two services called myweb1 and myweb2 in a namespace called sales, in a cluster named cluster.local, the assigned DNS names would be:

- myweb1.sales.svc.cluster.local
- myweb2.sales.svc.cluster.local

Pods

A Pod is the smallest deployable unit that Kubernetes can manage, and may contain one or more containers. If a pod has multiple containers running, they all share a common networking stack, allowing each container to communicate with the other containers in the pod using localhost or 127.0.0.1. They also share any volumes that are mounted to the pod, allowing each container access to a shared file location.

When a pod is created, it is assigned an IP address and the assigned address should be considered ephemeral. You should never target the IP address of a pod since it will likely change at some point when the pod is replaced. To target an application that is running in a pod, you should target a service name which will use endpoints to direct traffic to the correct pod where the application is running. We will discuss endpoints and services in their respective topics in this section.

While there is no standard for how many containers should be in a single pod, the best practice is to add containers that should be scheduled and managed together. Considerations including scaling and pod restarts should be considered when deciding to add

multiple containers to a pod. Events like these are handled at a pod level, not a container level, so these actions will affect all containers in the pod.

EXAMPLE: You create a pod with a web server and a database. You decide that you need to scale the web server out to handle the current traffic load. When you scale the pod, it will scale both the web server and the database server.

To scale only the web server, you should deploy a pod with the web server and a second pod with the database server, allowing you to scale each application independently.

Many design patterns will utilize multiple containers in a pod. A common use case for multiple containers in a pod is a term referred to as a sidecar. A sidecar is a container that runs with the main container in your pod, usually to add some functionality to the main container without requiring any changes to it.

Some of the most commonly used examples that use sidecars to handle tasks include:

- Logging
- Monitoring
- Istio sidecar
- Backup sidecar (ie: Veritas Netbackup)

You can look at other examples on the Kubernetes site at:
<https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/>

Understanding pods is a key point to understanding Kubernetes deployments, it will be the most common resource that you will interact with.

LABELS AND SELECTORS

Kubernetes uses labels to identify, organize, and link resources, allowing you to identify attributes. When you create a resource in Kubernetes, you can supply one or more **key:value** pair labels like **app:frontend-webserver** or **lob=sales**.

Selectors are used to reference a set of resources, allowing you to select the resource(s) you want to link, or select, using the assigned labels. You can think of selectors as a dynamic grouping mechanism - any label that matches the selector will be added as a target. This will be shown in the next resource, services, which uses selectors to link the service to the pods that are running the application.

SERVICES

We can use many of the previous resources to provide a full picture of how they connect to create an application.

The last piece of the puzzle is the service resource, which exposes an application to allow it to accept requests using a defined DNS name.

Remember that when you create a pod with your application, it is assigned an IP address. This IP address will change when the pod is replaced, which is why you never want to configure a connection to the pods using an IP address.

Unlike pods, which are ephemeral by nature, a service is stable once created and is rarely deleted and recreated, providing a stable IP address and DNS name. Even if a service is

deleted and recreated, the DNS name will remain the same, providing a stable name that you can target to access the application.

First, there are a few service types that can be created in Kubernetes:

Service name	Description	Network Scope
ClusterIP	Exposes the service internally to the cluster.	Internal External by using an Ingress rule
NodePort	Exposes the service internally to the cluster. Exposes the service to external clients using the assigned NodePort. Using the NodePort with any worker node DNS/IP address will provide a connection to the pod(s).	Internal and external
LoadBalancer	Exposes the service internally to the cluster. Exposes the service externally to the cluster using an external load-balancer service.	Internal and external

Now, let's use an example to explain how Kubernetes uses services to expose an application in a namespace called sales in a cluster using the name cluster.local.

1. A deployment is created for a NGINX server.
 - The deployment name is nginx-frontend.
 - The deployment has been labeled with **app: frontend-web**.
 - Three replicas have been created.The three running pods have been assigned IP addresses 192.10.1.105, 192.10.3.107, and 192.10.4.108.
2. To provide access to the server, a new service is deployed called frontend-web. In the manifest to create the service, a *label selector* is used to select any pods that match **app: frontend-web**.
3. Kubernetes will use the service request and the selector to create matching endpoints.

Since the selector matches the label that was used in the deployment for the NGINX server, Kubernetes will create an endpoint that links to the three pod IP's, 192.10.1.105, 192.10.3.107, and 192.10.4.108.

- The service will receive an IP address from the cluster's service IP pool, and a DNS name that is created using the `<service name>.<namespace>.svc.<cluster domain>`

Since the application name is `nginx-frontend`, the DNS name will be `nginx-frontend.sales.svc.cluster.local`.

If any of the pod IP's change due to a restart, the endpoints will be updated by the kube-controller-manager, providing you a stable endpoint to the pods, even when a pod IP address changes.

EndpointSlices

EndpointSlices map Kubernetes services to pod(s) that are running the application, linked by matching labels between the service selector and the pod(s) with a matching label. A graphical representation is shown in **Figure 4.8 - Kubernetes endpoints**.

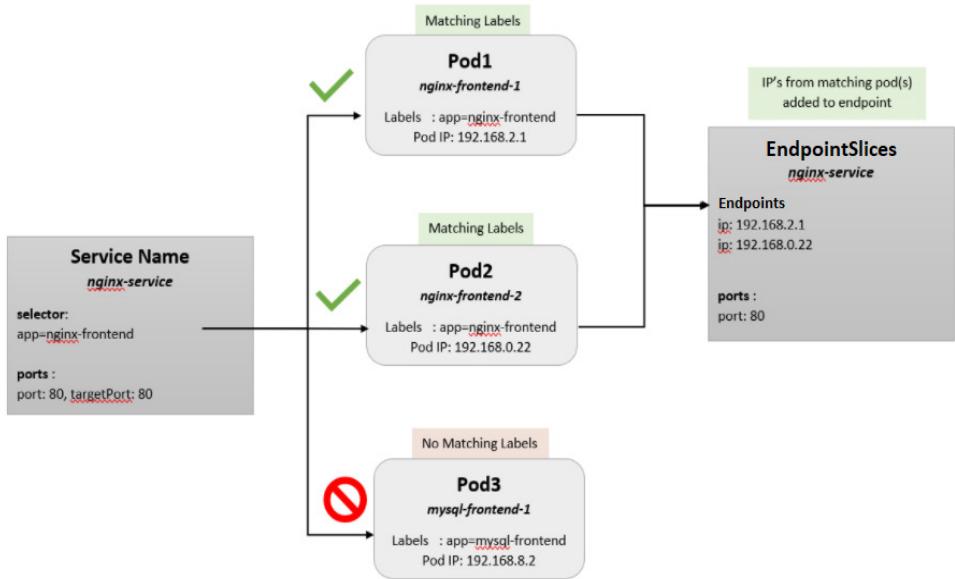


Figure 4.8 - Kubernetes endpoints

In the above example, a service named `nginx-service` has been created in a namespace. The service is using a selector for the key `app`, equal to the value `nginx-frontend`. Using the selector, Kubernetes will look for any matching labels in the namespace equal to `app=nginx-frontend`. The namespace has three running pods, two of the pods have been

labeled with **app=nginx-frontend**, and since the selector matches, all matching pod IP addresses are added to the EndpointSlices.

ANNOTATIONS

Annotations may look similar to selectors at a first glance. They are **key:value** pairs, just like labels are, but unlike labels, they are not used by selectors to create a collection of services.

You can use annotations to create records in a resource, like Git branch information, image hashes, support contacts, and more.

CONFIGMAPS

Configmaps are used to store information that is not confidential. They are used to store application configuration information separate from the container image.

While you could store a configuration directly in your container image, it would make your deployment too rigid, any configuration change would require you to create your new image. This would lead to maintaining multiple images, one for each configuration.

A better method would be to store the configuration in a configmap that is read in by your pod when it is started. Configmaps can be mounted as a file in the container, or as environment variables, depending on the application requirements. To deploy the image with a different configuration only requires a different configmap, rather than an entire image build.

For example, you have a web server image that requires a different configuration based on deployment location. You want to use the same image across your entire organization, regardless of the location of where the container will run. To accomplish this, you create a web container image that is configured to use a Configmap for the web server configuration.

By using an external configuration, you are making your image portable by allowing a configuration outside of the container itself.

SECRETS

Secrets are similar to configmaps, since they contain external information that will be used by pods. Unlike configmaps, secrets are not stored in cleartext, they are stored using base64 encoding.

If you have worked with base64 encoding before, you are probably thinking that it's not very different, or secure, from clear-text - and you would be right. Secrets in Kubernetes do not use base64 encoded to hide the secret, they are base64 encoded to allow secrets to store binary information. If a person has access to view the secret, it is trivial to decode the information to retrieve the information. Because of this, it is suggested to encrypt your secrets using an external secret manager like Vault, or Google's secret manager.

NOTE: You can also encrypt secrets when they are stored in ETCD, but this only encrypts the value in the database, not in Kubernetes. If you enable this feature, you are protecting the secrets in the ETCD database only. To secure your secrets you should use both encryption methods – this will protect your secrets both in the cluster and in ETCD.

ETCD will be discussed in the Understanding the Cluster Layers section of this chapter.

RESOURCEQUOTAS

Remember that namespaces are used to provide a logical separation for applications or teams. Since a cluster may be shared with multiple applications, we need to have a way to control any impact a single namespace may have on the other cluster resources. Luckily, Kubernetes includes the ResourceQuotas resource to provide resource controls.

A quota can be set on any standard Kubernetes resource that is namespaced, therefore, ResourceQuota's are set at a namespace level and control the resources that the namespace can consume, including:

- CPU
- Memory
- Storage
- Pods
- Services
- Etc...

Quotas allow you to control the resources that a namespace can consume, allowing you to share a cluster with multiple namespaces while providing a "guarantee" to cluster resources.

RBAC

Role-based access control, or RBAC, is used to control what users are able to do within a cluster. Roles are created and assigned permissions, which are then assigned to users or groups, providing permissions to the cluster.

To provide RBAC, Kubernetes uses roles and binding resources. Roles are used to create a set of permissions to a resource or resources, while bindings are used to assign the permission set to a user or service.

ROLES AND CLUSTERROLES

A role creates a set of permissions for a resource or resources. There are two different types of roles in Kubernetes that are used to define the scope of the permissions:

Role Type	Scope	Description
Role	Namespace	Permissions in a Role can only be used in the namespace that it was created in.
ClusterRole	Cluster	Permissions in a ClusterRole can be used cluster wide.

The scope of roles can be confusing for people that are newer to Kubernetes. The Role resource is more straightforward than the ClusterRole resource. When you create a Role, it must contain a namespace value, which creates the role in the assigned namespace. Since a Role only exists in the namespace, it can only be used to assign permissions in the namespace itself, it cannot be used anywhere else in the cluster.

A ClusterRole is created at the cluster level and can be used anywhere in the cluster to assign permissions. When assigned to the cluster level, the permissions that are granted in the ClusterRole will be assigned to all defined resources in the cluster. However, if you use a ClusterRole at a Namespace level, the permissions will only be available in the assigned namespace.

Two of the most commonly used ClusterRoles are the built-in admin and view.

By themselves, roles and ClusterRoles, do not assign a set of permissions to any user. To assign a role to a user, you need to bind the role by using a RoleBinding or a ClusterRoleBinding.

ROLEBINDING AND CLUSTERROLEBINDING

Roles simply define the set of permissions that will be allowed for resources, they do not assign the granted permissions to any user or service. To grant the permissions that are defined in a role, you need to create a binding.

Similar to Roles and ClusterRoles, there are two scopes to bindings, each is described in the table below.

Binding Type	Scope	Description
RoleBinding	Namespace	Can be used to assign permissions only in the namespace that it was created in.
ClusterRoleBinding	Cluster	Can be used to assign permissions cluster wide.

Now that we have discussed Kubernetes resources, let's move on to how Kubernetes schedules workload on clusters.

4.5.7 Understanding the Kubernetes scheduler

Earlier in the chapter, we mentioned that the kube-scheduler was responsible for finding a node that meets the pod resource requirements, and any custom scheduling properties that may have been declared.

To understand how a node is selected to run a workload, we need to understand how the scheduler uses filters and scoring to select the best node for the pod.

STEP 1: NODE FILTERING

The first step of the scheduling process is to remove any nodes that do not meet the minimum requirements for the requested resources. Filters are based on predicates, and the default scheduler includes a set of predicates including:

Predicate	Description
PodFitsHost	If the pod has a selector for a node name, check the node's name and compare it to the pod selector.
PodFitsResource	Check the node for free CPU and Memory.
PodMatchNodeSelector	If a pod has a selector(s) set, checks the value of the node's labels against the pod selector list.
PodToleratesNodeTaints	If a node has any taints set, checks to verify if the pod can tolerate the node taints.

For example, let's show how the PodFitsResource predicate works, the below image shows an overview of how the scheduler filters the list of potential nodes.

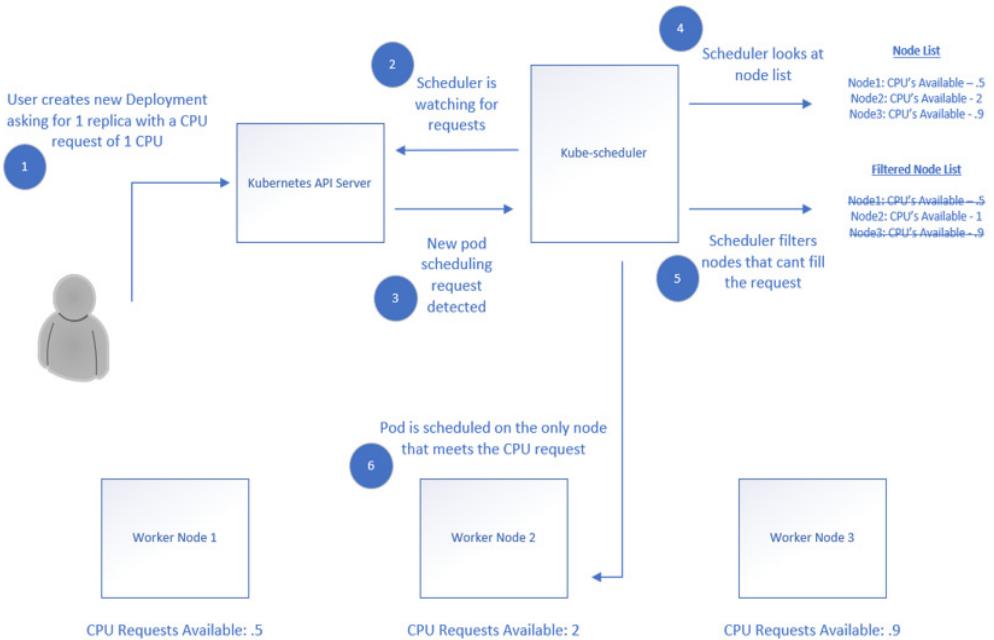


Figure 4.9 – Kubernetes scheduler node filtering

1. A user deploys a manifest that creates a Deployment with a single replica and a CPU request of 1 CPU.
2. The scheduler is always watching the API server for any unscheduled pods.
3. When the scheduler sees a pod that needs to be scheduled it starts the task of scheduling the pod on a node.
4. Using the node list, the scheduler starts to filter the list.
5. Since the Deployment has set 1 CPU request of 1, any nodes that do not have 1 CPU available will be removed from the list of nodes.
6. In our example, only a single node is left after the filtering so the pod will be scheduled on node 2 since it has 2 CPU's available.

This example only considers a CPU request to filter the nodes, but the scheduler will also filter on any requirement that is included in a manifest. If the scheduler cannot find a suitable node after all filters have been evaluated, the pod will fail to start, since the cluster does not have a node that meets the manifest requirements.

If we were to change the CPU request in Figure 1.5 – Kubernetes scheduler node filtering, to 3 CPU's, the scheduling request would fail. With the increased CPU request, the scheduler would filter out node 2 since it only has 3 available cores, resulting in no suitable node to schedule the pod on.

STEP 2: NODE SCORING

The previous scheduling example resulted in a single node that could schedule the pod, but what would have happened if the filter resulted in more than a single node in the filtered list? When more than one node is available for scheduling, the scheduler will use a scoring system to rank the nodes, ultimately scheduling the pod on the node that scores the highest.

Like filtering, which uses predicates to filter out nodes, scoring uses priorities to calculate a node's score. Some examples of priorities are shown in the table below.

Priority	Description
BalancedResourceAllocation	Calculates the CPU and memory balancing of the node if the pod is scheduled to run on the node.
SelectorSpreadPriority	Attempts to minimize the number of pods from the same application (ie: Service, ReplicaSet, ReplicationController) running on the same node.
ImageLocalityPriority	Scoring is based on the size of images that already exist on the host that the pod requires - Larger image totals will receive a higher score.

Each priority receives a score between 0 and 10, with 0 being the lowest value and 10 is the highest value. Once all priorities have been evaluated, the score from each of them is added up and the node with the highest value will be selected to run the pod. In the event that more than one node has the same score, Kubernetes will randomly select one of them to run the workload on.

4.5.8 Controlling pod scheduling

As Kubernetes has gained popularity, the use-cases have grown and become more complex. You may run into deployments that require special scheduling, including:

- A pod that requires a GPU, or other specialized hardware
- Forcing pods to run on the same node
- Forcing pods to run on different nodes
- Specific local storage requirements
- Use locally installed NVMe disks

If you simply deploy a manifest to your cluster, the scheduler does not take any "special" considerations into account when selecting a node. If you deployed a pod that required

CUDA and the pod was scheduled to run on a node that did not have a GPU the application would fail to start, since the required hardware would not be available to the application.

Kubernetes provides the ability to force a pod to run on a particular node, or set of nodes, using advanced scheduling options that are set on the node level, and in your deployment. At the node level we use node labels and node taints to group nodes, and at the deployment level we use node selectors, affinity/anti-affinity rules, and taints / tolerations to decide on pod placement.

USING NODE LABELS AND TAINTS

At the node level, there are two methods you can use to control the pods that will be scheduled on the node(s). The first method is by labeling node(s), and the second is by tainting the node(s). While both methods allow you to control if a pod will be scheduled on the node(s), they have different use-cases by either attracting the pods or repelling the pods.

ATTRACTING VERSUS REPELLING

Labels are used to group a set of nodes that you can use to target in your deployment, forcing the pod(s) to run on that particular set of nodes. When you label a node, you are not rejecting any workloads. A label is an optional value that can be used by a deployment, if a value is set in the deployment to use a label, or labels. In this way, we are setting an attraction for pods that may have a requirement that a label will provide, for example, **gpu=true**.

To label a node using kubectl, you use the label option:

```
kubectl label nodes node1 gpu=true
```

If a deployment requires a GPU, it uses a selector that tells the scheduler that it needs to be scheduled on a node with a label **gpu=true**. The scheduler will look for nodes with a matching label and then schedule the pod to run on one of the nodes with a matching label. If a matching label cannot be found, the pod will fail to schedule and the pod will not start.

The use of a label is completely optional. Using the example label above, if you create a deployment that does not select the **gpu=true** label, your pod will not be excluded from nodes that contain the label.

Taints work differently, rather than create a key:value that invites the pods to be run on it, a taint is used to repel any scheduling request that cannot tolerate the value set by the taint. To create a taint, you need to supply a key:value and an effect, which controls if a pod is scheduled.

For example, if you wanted to control the nodes that have a GPU, you could set a taint on a node using kubectl:

```
kubectl taint nodes node1 gpu=true:NoSchedule
```

This would taint node1 with the key:value of **gpu=true**, and the effect of NoSchedule, which tells the scheduler to repel all scheduling requests that do not contain a toleration of **gpu=true**. Unlike a label, which would allow pods that do not specify a label to be scheduled, a taint setting with the effect **NoSchedule** will deny any pod that does not “tolerate” **gpu=true** to be scheduled.

Taints have three effects that can be set, *NoSchedule*, *PreferNoSchedule*, and *NoExecute*. Each one sets a control on how the taint will be applied:

- **NoSchedule**: This is a “hard” setting that will deny any scheduling request that does not tolerate the taint.
- **PreferNoSchedule**: This is a “soft” setting that will only attempt to avoid scheduling a pod that does not tolerate the taint.
- **NoExecute**: Effects already running pods on a node, it is not used for scheduling a pod.

Now that we have explained how to create labels and taints on nodes, we need to understand how a deployment is configured to control pod placement.

USING NODE NODESELECTORS

When a pod is created, you can add a nodeSelector to your manifest to control the node(s) that the pod will be scheduled on. By using any label that is assigned to a node, you can force the scheduler to schedule the pod on a node or a set of nodes.

You may not know all of the labels that are available on a cluster. If you have access to list the nodes, you can use kubectl to get a list of the nodes and all labels by using the get nodes command with the --show-labels option.

```
kubectl get nodes --show-labels
```

This will list each node and the labels that have been assigned.

NAME	STATUS	ROLES	AGE	VERSION	LABELS
gke-cluster-1-default-pool-77fd9484-7fd6	Ready	<none>	3m28s	v1.16.11-gke.5	beta.kubernetes.io/arch=amd64,beta.kubernetes.io/instance-type=e2-medium,beta.kubernetes.io/os=linux,cloud.google.com/gke-nodepool=default-pool,cloud.google.com/gke-os-distribution=cos,cloud.google.com/gke-preemptible=true,failure-domain.beta.kubernetes.io/region=us-central1,failure-domain.beta.kubernetes.io/zone=us-central1-a,kubernetes.io/arch=amd64,kubernetes.io/hostname=gke-cluster-1-default-pool-77fd9484-7fd6,kubernetes.io/os=linux
gke-cluster-1-default-pool-ca0442ad-hqk5	Ready	<none>	3m18s	v1.16.11-gke.5	beta.kubernetes.io/arch=amd64,beta.kubernetes.io/instance-type=e2-medium,beta.kubernetes.io/os=linux,cloud.google.com/gke-nodepool=default-pool,cloud.google.com/gke-os-distribution=cos,cloud.google.com/gke-preemptible=true,failure-domain.beta.kubernetes.io/region=us-central1,failure-domain.beta.kubernetes.io/zone=us-central1-b,kubernetes.io/arch=amd64,kubernetes.io/hostname=gke-cluster-1-default-pool-ca0442ad-hqk5,kubernetes.io/os=linux
gke-cluster-1-default-pool-ead436da-8j7k	Ready	<none>	3m19s	v1.16.11-gke.5	beta.kubernetes.io/arch=amd64,beta.kubernetes.io/instance-type=e2-medium,beta.kubernetes.io/os=linux,cloud.google.com/gke-nodepool=default-pool,cloud.google.com/gke-os-distribution=cos,cloud.google.com/gke-preemptible=true,failure-domain.beta.kubernetes.io/region=us-central1,failure-domain.beta.kubernetes.io/zone=us-central1-c,kubernetes.io/arch=amd64,kubernetes.io/hostname=gke-cluster-1-default-pool-ead436da-8j7k,kubernetes.io/os=linux

Figure 4.10

You can also see the node labels in the GCP console, by clicking on the details for a node.

The screenshot shows the GCP Kubernetes Engine interface. On the left, there's a sidebar with icons for Clusters, Workloads, Services & Ingress, Applications, Configuration, Storage, Object Browser, and Migrate to containers. The main area is titled 'Node details' for a node named 'gke-cluster-1-default-pool-77fd9484-7fd6'. Below the title are tabs for Summary, Details (which is selected), YAML, and Events. Under the 'Details' tab, there are sections for Cluster (cluster-1), Node pool (default-pool), Zone (us-central1-a), Created (Aug 8, 2020, 12:06:18 PM), Labels (beta.kubernetes.io/arch: amd64, beta.kubernetes.io/instance-type: e2-medium, beta.kubernetes.io/os: linux, cloud.google.com/gke-nodepool: default-pool, cloud.google.com/gke-os-distribution: cos, cloud.google.com/gke-preemptible: true, failure-domain.../region: us-central1, failure-domain.../zone: us-central1-a, kubernetes.io/arch: amd64, kubernetes.io/hostname: gke-cluster-1..., kubernetes.io/os: linux), and Annotations (containing container.googleapis.com/instance_id: 70454018393632200, node.alpha.kubernetes.io/ttl: 0, node.gke.io/last-applied-node-labels: cloud.google.com/gke-nodepool=default-pool, cloud.google.com/gke-os-distribution:cos, cloud.google.com/gke-preemptible=true, volumes.kubernetes.io/controller-managed-attach-detach: true).

Figure 4.11 - GCP node console view

Using a label from the cluster in the images, we can create a manifest that will deploy Nginx on the third node by using a nodeSelector.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  Labels:
    run: nginx-test
  name: nginx-test
spec:
  replicas: 1
  selector:
    matchLabels:
      run: nginx-test
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: nginx-test
    spec:
      containers:
        - image: bitnami/nginx
          name: nginx-test
      nodeSelector:
        kubernetes.io/hostname: gke-cluster-1-default-pool-ead436da-8j7k
```

Using the value ***kubernetes.io/hostname:gke-cluster-1-default-pool-ead436da-8j7k*** in a nodeSelector, we forced the pod to run the third node in the cluster. To verify the pod did schedule on the correct node, we can use ***kubectl*** to get the pods using the **-o wide** option.

kubectl get pods -o wide

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
nginx-test-765fb5b456-nd66w	1/1	Running	0	9s	10.4.0.3	gke-cluster-1-default-pool-ead436da-8j7k

Figure 4.12 - Get pods with wide output

The nodeSelector option allows you to use any label(s) to control what nodes will be used to schedule your pods. If the nodeSelector value does not match any nodes, the pod will fail to schedule and will remain in a pending state until it is deleted, or a label is updated on a node that matches the selector. In the below example, we tried to force a deployment to a nodeSelector that had a value of a host that does not exist in the cluster.

First, we can look at all of the pods to check the status using **kubectl get pods**.

NAME	READY	STATUS	RESTARTS	AGE
nginx-test-765fb5b456-nd66w	1/1	Running	0	3h25m
nginx-test2-6dccc98749-xng5h	0/1	Pending	0	2m55s

Figure 4.13 - Gets pods output

Notice that the nginx-test2 pod is in a pending state. The next step to check why the pod fails to start is to describe the pod.

kubectl describe pod nginx-test2-6dccc98749-xng5h

A description of the pod will be displayed, including the current status at the bottom of the output.

Events:				
Type	Reason	Age	From	Message
Warning	FailedScheduling	18s (x5 over 4m23s)	default-scheduler	0/3 nodes are available: 3 node(s) didn't match node selector.

Figure 4.14 - Kubectl describe output

In the message area, the status shows **0/3 nodes are available: 3 node(s) didn't match node selector**. Since our nodeSelector did not match any existing label, the pod failed to start. To resolve this, you should verify that the nodeSelector is correct and if it is, verify that a node has the same label set.

USING AFFINITY RULES

Node affinity is another way to control which node(s) your pods will run on, but unlike a nodeSelector, an affinity rule can:

- Contain additional syntax beyond a simple matching label.
- Schedule based on an affinity rule match, but if a match is not found, the pod will schedule on any node.

Unlike a nodeSelector which has a single value, a node affinity rule can contain operators , allowing for more complex selections. The table below contains a list of operators and a description of how they are evaluated.

Operator	Description
In	Checks the label against a list, if any value is in the list, it is considered to be a match.
NotIn	Checks the label against a list, and if the value is not in the list, it is considered to be a match.
Exists	Checks if the label exists, if it does, it is considered to be a match. Note: The value of the label does not matter and is not evaluated in the match.
DoesNotExist	Checks if the label does not exist, if the label does not match any in the list, it is considered to be a match. Note: The value of the label does not matter and is not evaluated in the match.
Gt	Used to compare numeric values in a label, if a value is Greater than (Gt) the label, it is considered to be a match.. Note: Only works with a single number.
Lt	Used to compare numeric values in a label, if a value is Less than (Lt) the label, it is considered to be a match.. Note: Only works with a single number.

Using a node affinity rule, you can choose a soft or hard affinity based on your requirements. Affinity rules can be created using two preferences, **RequiredDuringSchedulingIgnoredDuringExecution**, also known as a hard affinity and **preferredDuringSchedulingIgnoredDuringExecution**, also known as a soft affinity. If you use a hard affinity, the affinity must match or the pod will fail to schedule. However, if you use a soft affinity, the affinity rule will be used if it matches, but if a match is not found the pod will schedule on any node in the cluster.

Creating node affinity rules

Node affinity is set in a manifest in the PodSpec, under the affinity field as nodeAffinity. To better explain how to use a node affinity rule, let's use an example cluster to create a manifest that uses an affinity rule.

The cluster has three nodes, the label we will use in the rule is in **bold**.

Node	Node Labels
Node 1	beta.kubernetes.io/arch=amd64,beta.kubernetes.io/instance-type=e2-medium,beta.kubernetes.io/os=linux,cloud.google.com/gke-nodepool=default-pool,cloud.google.com/gke-os-distribution=cos,cloud.google.com/gke-preemptible=true,failure-domain.beta.kubernetes.io/region=us-central1,failure-domain.beta.kubernetes.io/zone=us-central1-a,kubernetes.io/arch=amd64,kubernetes.io/hostname=gke-cluster-1-default-pool-77fd9484-7fd6,kubernetes.io/os=linux
Node 2	beta.kubernetes.io/arch=amd64,beta.kubernetes.io/instance-type=e2-medium,beta.kubernetes.io/os=linux,cloud.google.com/gke-nodepool=default-pool,cloud.google.com/gke-os-distribution=cos,cloud.google.com/gke-preemptible=true,failure-domain.beta.kubernetes.io/region=us-central1,failure-domain.beta.kubernetes.io/zone=us-central1-b,kubernetes.io/arch=amd64,kubernetes.io/hostname=gke-cluster-1-default-pool-ca0442ad-hqk5,kubernetes.io/os=linux
Node 3	beta.kubernetes.io/arch=amd64,beta.kubernetes.io/instance-type=e2-medium,beta.kubernetes.io/os=linux,cloud.google.com/gke-nodepool=default-pool,cloud.google.com/gke-os-distribution=cos,cloud.google.com/gke-preemptible=true,failure-domain.beta.kubernetes.io/region=us-central1,failure-domain.beta.kubernetes.io/zone=us-central1-c,kubernetes.io/arch=amd64,kubernetes.io/hostname=gke-cluster-1-default-pool-ead436da-8j7k,kubernetes.io/os=linux

We want to create a deployment that will create a Nginx server in either the **us-central1-a**, or **us-central1-c** zones. Using the manifest below, we can create a pod in either of the zones using an affinity rule based on the **failure-domain.beta.kubernetes.io/zone** key.

```

apiVersion: v1
kind: Pod
metadata:
  name: nginx-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: failure-domain.beta.kubernetes.io/zone
                operator: In
                values:
                  - us-central1-a
                  - us-central1-c
  containers:
    - name: nginx-affinity
      image: bitnami/nginx

```

By using the key **`failure-domain.beta.kubernetes.io/zone`** in the **`matchExpressions`** we set the affinity to evaluate to true if the node label matches either **`us-central1-a` or `us-central1-c`**. Since the second node in the cluster has a label value of `us-central2-b`, it will evaluate as false and will not be selected to run the pod.

USING POD AFFINITY AND ANTI-AFFINITY RULES

A pod affinity rule will ensure that deployed pods are running on the same set of nodes as a matching label, and an anti-affinity rule is used to ensure that pods will **not run** on the same nodes as a matching label. Pod affinity rules are used for different use-cases than node affinity use-cases. While node affinity allows you to select a node based on the cluster node labels, pod affinity and anti-affinity rules use the labels of pods that are already running in the cluster.

Creating pod affinity Rules

When you create an affinity rule, you are telling the scheduler to place your pod on a node that has an existing pod that matches the selected value in the affinity rule. Pod affinity rules, like node affinity rules, can be created as soft or hard affinity rules. They also use operators like node affinity rules, including **`In`, `NotIn`, `Exists`, `DoesNotExist`**, but they **do not** support the `Gt` or `Lt` operators.

Pod affinity rules are specified in the **`PodSpec`**, under the **`affinity`**, and **`podAffinity`** fields. They require an additional parameter that node affinity rules do not use, the **`topologyKey`**. The `topologyKey` is used by Kubernetes to create a list of nodes that will be checked against the affinity rule. Using a `topologyKey`, you can decide to look for matches based on different filters like zones, or nodes.

For example, you have a software package that is licensed per node and each time a pod that runs a portion of the software runs on another node, you need to purchase an additional license. To lower costs, you decide to create an affinity rule that will force the pods to run where an existing licensed pod is running. The existing pods run using a label called **`license`** with a value of **`widgets`**. Below is an example manifest that will create a pod on a node with an existing pod with a label **`license=widgets`**.

Since we need to be on the same node, to maintain licensing, we will use a topologyKey that will filter by kubernetes.io/hostname.

```
apiVersion: v1
kind: Pod
metadata:
  name: widgets-license-example
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: license
                operator: In
                values:
                  - widgets
      topologyKey: kubernetes.io/hostname
  containers:
    - name: nginx-widgets
      image: nginx-widgets
```

The manifest tells Kubernetes to create the pod nginx-widgets, running an image called nginx-widgets on a host that already has a pod running using the **label license** with the **value of widgets**.

Creating pod anti-affinity rules

Anti-affinity rules will do the opposite of affinity rules. Where affinity rules will group pods based on a set of rules, anti-affinity rules are used to run pods on different nodes. When you use an anti-affinity rule, you are telling Kubernetes that you **do not** want the pod to run on another node that has an existing pod with the values declared in the rule.

Some common examples that use anti-affinity rules include forcing pods to avoid other running pods or spreading pods across availability zones.

Pod anti-affinity rules are specified in the PodSpec, under the affinity, and podAntiAffinity fields. They also require the **topologyKey** parameter to filter the list of nodes that will be used to compare the affinity rules.

In our affinity example, we used a topologyKey that used the hostname of the node. If we use the same key for the deployment, zones wouldn't be considered, it would only avoid placing the pod on the same node as another running pod. While the pods would spread across nodes, the selected nodes could all be in the same zone, failing to spread the pods across zones.

To spread the pods across zones, we will use the label **failure-domain.beta.kubernetes.io/zone**, and we will use the operator **In** to compare the label app for the value of **nginx-frontend**. We will also use a soft anti-affinity rule, rather than a hard rule, allowing Kubernetes to use the same zone, if there is no other choice.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-frontend-antiaffinity-example
spec:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
      podAffinityTerm:
        labelSelector:
          matchExpressions:
            - key: app
              operator: In
              values:
                - nginx-frontend
        topologyKey: failure-domain.beta.kubernetes.io/zone
  containers:
    - name: nginx-frontned
      image: bitnami/nginx
```

By using the `failure-domain.beta.kubernetes.io/zone` as the **topologyKey**, we are telling Kubernetes that we want to avoid placing any pod that has a **label** of `app=nginx-frontend` in the same zone.

USING TAINTS AND TOLERATIONS

While you are more likely to require scheduling a pod to a specific node, there will be use cases where you will want to reserve specific nodes to only certain workloads, essentially disabling default scheduling. Unlike NodeSelector and affinity rules, taints are used to automatically repel, rather than attract a pod, to a node. It's useful when you want the default action for a node to reject any scheduling attempts, unless a deployment specifically states the correct "tolerations" to be scheduled on the node. For example, you have a cluster that has a few hundred nodes and a few nodes have GPU's available. Since GPU's are expensive, we want to restrict pods on these nodes to only applications that require a GPU, rejecting any standard scheduling requests.

Using controls like NodeSelector or affinity rules will not tell the Kubernetes scheduler to avoid using a node. These provide a developer the ability to control how pods will be deployed, and if they don't provide either of these, the scheduler will attempt to use any node in the cluster. Since GPU's are expensive, we want to reject any scheduling attempt to run a pod on a node with a GPU that doesn't require using a GPU.

Create a Node Taint

To stop the scheduler from scheduling pods on a node, you need to "taint" the node with a value. To create a taint, use kubectl with the taint command and the node(s) you want to taint, the key:value, and the effect. The key:value can be any value that you want to assign and the effect can be one of three values, `NoSchedule`, `PreferNoSchedule`, or `NoExecute`.

Effect	Description
NoSchedule	If a pod does not specify a toleration that matches the node taint(s), it will not be scheduled to run on the node.
PreferNoSchedule	If a pod does not specify a toleration that matches the node taint(s), the scheduler will attempt to avoid scheduling a pod on the node.
NoExecute	If a pod is already running on a node and a taint is added, if the pod does not match the taint, it will be evicted from the node.

For example, if we had a GPU in a node named node1, we would taint the node using the below command.

```
kubectl taint nodes node1 gpu=:NoSchedule
```

The key in the taint command tells the scheduler what taint must be matched to allow a pod to schedule on the node. If the taint is not matched by a pod request, using a toleration, the scheduler will not schedule a pod on the node, based on the **effect NoSchedule**.

Creating pods with Tolerations

By default, once a node has a taint set, the scheduler will not attempt to run any pod on the tainted node. By design, you set a taint to tell the scheduler to avoid using the node in any scheduling, unless the deployment specifically requests running on the node. To allow a pod to run on a node that has a tainted, you need to supply a **toleration** in the deployment. A toleration is used to tell the scheduler that the pod can “tolerate” the taint on the node, which will allow the scheduler to use a node that matches the toleration with an assigned taint.

NOTE: Taints will not attract a pod request - They only reject any pod that does not have a toleration set. As such, to direct a pod to run on a node with a taint, you need to set a toleration and a node selection, or a node affinity. The selector will tell the scheduler to use a node with a matching label, then the toleration tells the scheduler that the pod can tolerate the taint set on the node. Since tolerations tell the scheduler to “prefer” a node with a matching taint, if one cannot be found, the scheduler will use any node in the cluster with a matching label.

KEY TAKEAWAY: Toleration and node selectors/affinity rules combine to select the node that the pod will run on.

Tolerations are created in the pod.spec section of your manifest by assigning one or more tolerations that include the key to match, an operator, an optional value, and the taint effect.

The key must be assigned to the key that matches the node you want to schedule the pod on. The operator value tells the scheduler to simply look for the key (**Exists**) or to match a key value (**Equals**). If you use the Equals operator, your toleration must contain a

value field. Finally, the effect is the effect that needs to be matched for the pod to be scheduled on the node.

To schedule a pod that can tolerate the GPU taint for node1, you would add the following to your PodSpec:

```
spec:  
  tolerations:  
    - key: "gpu"  
      operator: "Exists"  
      effect: "NoSchedule"
```

Adding the toleration tells the scheduler that the pod should be assigned to a node that has a taint key of **gpu** with an effect of **NoSchedule**.

Controlling where pods will be scheduled is a key point to understand to ensure that your application deployments can meet your assigned SLA/SLO objectives.

4.6 Advanced topics

This section contains a few advanced topics that we wanted to include in this chapter. We thought these were important topics, but they weren't required to understand the main topics in the chapter.

4.7 Aggregate ClusterRoles

When a new component is added to the cluster, a new ClusterRole is often created that can be assigned to users to manage the service. There are times that a role may be created and you may notice that a user assigned the ClusterRole of admin has permissions to the new components by default. Other times, you may notice that a newly added component, like Istio, does not allow the built-in admin role to use any Istio resources.

It may sound odd that a role like admin would not have permissions to every resource by default. Kubernetes includes two ClusterRoles that provide some form of admin access, the first is the admin ClusterRole and the other is the cluster-admin ClusterRole. They may sound similar, but how permissions are assigned are very different between them.

The cluster-admin role is straight-forward since it is assigned wildcards for all permissions, providing access to every resource, including new resources. The admin role is very different since it is not assigned wildcard permissions. Each permission that is assigned to the admin role is usually explicitly assigned. Since the role does not use wildcards, any new permissions need to be assigned for new resources.

To make this process easier, Kubernetes has a concept called Aggregated ClusterRoles. When a new ClusterRole is created, it can be aggregated to any other ClusterRole by assigning an aggregationRule. Let's see an example to help explain how aggregation works:

The default admin ClusterRole looks similar to the below example.

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: admin
...
aggregationRule:
  clusterRoleSelectors:
    - matchLabels:
        rbac.authorization.k8s.io/aggregate-to-admin: 'true'
```

In the bolded text above, you can see that the admin ClusterRole has an aggregationRule that contains **rbac.authorization.k8s.io/aggregate-to-admin: 'true'**. When a new ClusterRole is created, it can be automatically aggregated with the built-in admin ClusterRole if it uses the same aggregationRule. For example, a new CRD has been deployed to the cluster that creates a new ClusterRole. Since the permissions for the new ClusterRole should be assigned to admins, it has been created with an aggregationRule that matches **rbac.authorization.k8s.io/aggregate-to-admin: 'true'**:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: aggregate-example-admin
  labels:
    rbac.authorization.k8s.io/aggregate-to-admin: "true"
rules:
- apiGroups: ["newapi"]
  resources: ["newresource"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

This will create a new ClusterRole named aggregated-example-admin that assigns the actions get, list, watch, create, patch and delete to the resource newresource in the newapi apiGroup. This new new ClusterRole can be bound to any user that you want to assign permissions to, but since the permission is required by admins, it also has a label assigned of rbac.authorization.k8s.io/aggregate-to-admin: "true", which matches the aggregationRule that is assigned in the admin ClusterRole. Since the labels match, a controller on the API server will notice the matching labels and "merge" the permissions from the new ClusterRole with the admin ClusterRole.

4.8 Custom schedulers

One of the most misunderstood concepts in Kubernetes is how the cluster schedules workloads. You will often hear that applications deployed on a Kubernetes cluster are highly available, and they are, when deployed correctly. To deploy a highly available application, it's beneficial to understand how the kube-scheduler makes decisions and the options available to your deployments to help influence the decisions that it will make.

The default Kubernetes scheduler, kube-scheduler, has the job of scheduling pods to worker nodes based on a set of criteria that include node affinity, taints and tolerations, and node selectors. While Kubernetes includes the base scheduler, you are not stuck using only a single scheduler for all deployments. You can create custom schedulers if you need

special scheduling considerations that the base scheduler does not include, by specifying a scheduler in the manifest, and if one is not provided, the default scheduler will be used. Creating a custom scheduler is beyond the scope of this book, but you can read more about custom schedulers on the Kubernetes site at <https://kubernetes.io/docs/tasks/extend-kubernetes/configure-multiple-schedulers/>

Below is an example of a pod that sets the scheduler to a custom scheduler named custom-scheduler1.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-web
spec:
  containers:
  - name: nginx
    image: bitnami/nginx
  schedulerName: custom-scheduler1
```

The Kubernetes scheduler watches the API server for pods that require scheduling. Once it determines that a pod needs to be scheduled it will determine the most appropriate node by going through a multi-stage decision process that will filter out nodes and then assign a score to nodes that have not been filtered out.

4.9 Summary

In this chapter you learned about how Google has taken out many of the complexities of Kubernetes with Anthos. We explained a number of topics that will help you work with Anthos and Kubernetes that included:

- How the components of a cluster work together and the role each one plays in the Kubernetes architecture.
- How declarative application and imperative application management work and how control loops factor into how the desired state of a resource is maintained.
- How the Kubernetes scheduler decides which node to schedule a workload on.
- How to control the placement of your workloads using options like selectors, affinity/anti-affinity rules, and tolerations.

4.10 Examples and Case Studies

Using the knowledge from the chapter, address each of the requirements in the case study found below. Remember that if you deployed your GKE cluster across different regions to replace any commands with your regions, replacing the example regions in the exercises. To save on any potential cost, the examples only require a single node in each region.

4.10.1 FooWidgets Industries

You have been asked to assist FooWidgets Industries with a new GKE cluster that they have deployed. They quickly discovered that they did not have the internal skills to complete their

deployment and therefore, the current state of the cluster is a simple, new cluster, across three GCP zones.

CLUSTER OVERVIEW AND REQUIREMENTS

FooWidgets Industries has a GKE cluster that has been deployed across three zones, **us-east4-a**, **use-east4-b**, and **us-east4-c**. FooWidgets has various requirements for pod placement based on internal standards and specialized hardware use. They have included a breakdown of the desired placement of workloads and the labels that should be assigned to nodes, outlined in the table below:

Zone	Desired Workloads	Label/Taint	Node Name
us-east4-a	Any workload Fast disk access	disk=fast	gke-cls1-pool1-1d704097-4361
us-east4-b	Only workloads that require a GPU	workload=gpu	gke-cls1-pool1-52bcec35-tf0q
us-east4-c	Any workload		gke-cls1-pool1-e327d1de-6ts3

The statement of work requires you to provide the requirements in the table. The cluster has not been configured past the initial deployment stage and will require you to complete the following configuration:

- Create any node labels or taints that are required to achieve workload placement based on the supported workloads documented in the table
- Create example deployment using an NGINX image to demonstrate successful placement of workloads based on the requirements provided by FooWidgetsIndustries

The next section contains the solution to address FooWidgets requirements. You can follow along with the solution, or if you are comfortable, configure your cluster to address the requirements and use the solution to verify your results.

FooWidgets Industries Solution - Labels and Taints

The first requirement is to create any labels or taints that may be required. Using the requirements table, we can tell that we need to label the nodes in **us-east4-a** with **disk=fast**. A label will allow a deployment to force scheduling on a node that has the required fast disks for the application. The second requirement is to limit any running workloads in the us-east4-b zone to only applications that require a GPU. For this requirement, we have decided to taint all nodes in the **us-east4-b** zone with **workload=gpu**.

Why is a label used for one solution and a taint for the other solution?

You may recall that labels and taints are used to accomplish different scheduling requirements - we use labels to attract workloads, while we use taints to repel workloads. In the requirements, FooWidgets clearly states that **us-east4-a** and **us-east4-c** can run any type of workload, but **us-east4-b** must only run workloads that require a GPU. If a

deployment is created that does not specify a label on a node, the scheduler will still consider that node as a potential node for scheduling. Labels are used to force a deployment to a particular node, but they do not reject workloads that do not contain a label request. This behaviour is far different than a node that has been assigned a taint. When a node is tainted, it will repel any workloads that do not contain a toleration for the assigned node taint. If a deployment is created without any tolerations for the node taint, the scheduler will automatically exclude the tainted nodes from scheduling the workload.

CREATING THE LABELS AND TAINTS

We need to label the node in us-east4-a with disk=fast. To label the node we use the kubectl label command, supplying the node and the label.

```
kubectl label node gke-cls1-pool1-1d704097-4361 disk=fast
```

Next, we need to add a taint to the nodes in the us-east4-b zone with workload=gpu. Remember that a taint will repel any request that does not tolerate the assigned node taint(s), but it doesn't attract a workload. This means that you also need to add a label to direct the GPU pods to the correct node. To taint the node, we use the kubectl taint command, supplying the node name and the taint.

```
kubectl taint node gke-cls1-pool1-52bcec35-tf0q workload=gpu:NoSchedule
```

And, label the node to attract the GPU pods.

```
kubectl label node gke-cls1-pool1-52bcec35-tf0q workload=gpu
```

Notice that we did not add a label or a taint to the node in us-east4-c since that zone can run any workload.

Now that the nodes are labeled, you need to create example deployments to verify that workload placement matches the requirements from the table.

CREATING A DEPLOYMENT THAT REQUIRES FAST DISK

To force a deployment that requires fast disk access to the **us-east4-a** zone, you need to add a node selector to the deployment. The below manifest creates a NGINX server that contains a nodeSelector using the label disk=fast, forcing the workload to run on a node in the **us-east4-a** zone.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: nginx-fast
    name: nginx-fast
spec:
  containers:
  - image: nginx
    name: nginx-fast
  restartPolicy: Always
  nodeSelector:
    disk: fast
```

When you create and execute the manifest, the nodeSelector tells the scheduler to use a node with the label **disk:fast**. To verify the selector is working correctly, we can list the

pods with `-o wide` to list the node that the pod is running on. In **us-east4-a** we have a single node, `gke-cls1-pool1-1d704097-436` - the abbreviated output from our `kubectl get pods` confirms that the pod was scheduled correctly is shown below.

NAME	READY	STATUS	AGE	IP	NODE
nginxx-fast	1/1	Running	4m49s	10.8.0.4	gke-cls1-pool1-1d704097-4361

Now that you have confirmed that pods requiring fast disk access can be scheduled correctly, you need to create a deployment to test workloads that require GPU's.

CREATING A DEPLOYMENT THAT REQUIRE A GPU

Any workload that requires a GPU needs to be scheduled on a node in **us-east4-b**. We already tainted the node in that zone, and to confirm that a workload requiring a GPU will be scheduled correctly, we need to create a test deployment with a toleration using the manifest below.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: nginx-gpu
    name: nginx-gpu
spec:
  containers:
    - image: nginx
      name: nginx-gpu
  restartPolicy: Always
  nodeSelector:
    workload: gpu
  tolerations:
    - key: "workload"
      operator: "Equal"
      value: "gpu"
      effect: "NoSchedule"
```

When the manifest is applied, you can verify that the pod is running on the correct node in **us-east4-b** using `kubectl get pods -o wide`.

NAME	READY	STATUS	AGE	IP	NODE
nginxx-gpu	1/1	Running	3s	10.8.2.6	gke-cls1-pool1-52bcec35-tf0q

Comparing the output with the table that lists the nodes in each zone verifies that the pod has been scheduled on a node in the **us-east4-b** zone.

Congratulations! You have successfully addressed the workload requirements, and have proven that you understand how to schedule workloads based on node labels and taints.

4.11 References

4.11.1 The EFK stack

A collection of tools to enable logging. The stack includes three tools that must be deployed:

- ElasticSearch - <https://github.com/elastic/elasticsearch>

- FluentD - <https://github.com/fluent/fluentd>
- Kibana - <https://github.com/elastic/kibana>

The Github repositories for each component contain the details to download and install each component, providing an open-source logging system.

4.11.2 Backing up ETCD on GKE on-prem Clusters

<https://cloud.google.com/anthos/gke/docs/on-prem/how-to/backing-up>

5

Anthos Service Mesh: security and observability at scale

by Onofrio Petragallo

This chapter covers:

- Sidecar proxy and proxyless architectures
- An introduction and the main features of Istio
- Security and observability with Istio
- What is Anthos Service Mesh
- Practical example with code

One of the key components to being cloud native is to break up your application into microservices. This means that an application that may have run on a single server, now has multiple services, backed by multiple pods, that are separate components. As applications scale out their services, it becomes difficult to troubleshoot issues that you may encounter with the application. WIth this added complexity, we needed a tool that would help organize, secure, and add resilience to the expanding complexities that microservices introduced. Another important problem that a service mesh could fix is the fact that enterprises often have a huge number of microservices and aren't always able to control, manage and observe them.

In this chapter, we will discuss Anthos Service Mesh (ASM), and the features that ASM inherits from Istio⁴, a popular open source framework for creating, managing and implementing a service mesh.

⁴<https://istio.io/>

The implementation of a service mesh not only facilitates communication between microservices using a dedicated communication management control plane, it also includes tools to observe communication between services, aka observability, enhance security, control application traffic flow, and simulate faults in an application.

Anthos Service Mesh is a Google managed service that allows enterprise management of all the service meshes present in a hybrid cloud or multi cloud architecture from a single point, providing complete and in-depth visibility of all microservices. The visualization of the topography of the service mesh and the complete integration with Cloud Monitoring, provides users the tools to identify failing workloads or other problems, making problem resolution faster.

The security features made available by Anthos Service Mesh allow you to manage the authentication, authorization and encryption of communications through mTLS² mutual authentication to secure and ensure trust in both directions for the communication between microservices; mTLS ensures a high level of security, minimizing the related risks.

Finally, the traffic management features of Istio provide users the tools to manipulate traffic using request routing, fault injection, request timeouts, circuit breaking and mirroring.

As you can see, Istio includes a number of features that are complex and may be difficult to troubleshoot. There are few offerings in the market today that include support for Istio , leaving you to support your service mesh on your own. Google has addressed this by including ASM in Anthos, providing a single support point for your Kubernetes clusters and Istio.

Before talking about the Anthos Service Mesh features and Istio in detail, let's start by explaining what a service mesh actually is.

5.1 Technical Requirements

The hands-on portion of this chapter will require you to have access to a Kubernetes cluster running on GCP with the following deployment pattern:

- A GKE cluster with at least 3 nodes with 4 CPU's and 16GB of RAM

5.2 What is a service mesh?

To understand how a service mesh works, and why it's becoming a standard tool in the microservices toolbox, you need to understand what a service mesh provides.

The main advantages of adopting a service mesh are the ability to:

1. Observe and monitor all communications between the individual microservices
2. Secure connections between the available microservices
3. Ability to deliver resilient services (Distributed services) through multi-cluster and multi-cloud architecture patterns
4. Advanced traffic management: A/B testing, traffic splitting and canary rollouts

²https://en.wikipedia.org/wiki/Mutual_authentication

A service mesh is an infrastructure layer in a microservices architecture that controls the communication between services. It is possible to create a mesh of services within a microservices architecture that not only runs in a Kubernetes cluster, but it is also possible to create a single service mesh that spans multiple clusters, or even non-microservice services running on virtual machines.

A service mesh manages all the ingress, or inbound traffic, and egress, or outbound traffic for each microservice. Traffic management is a complex topic, and something that most users do not want to deal with. To remove this burden, Istio doesn't require the developer to make any changes in their application logic - instead, the service mesh handles all of this by using a sidecar proxy approach or a proxyless approach.

The sidecar proxy is one of the main components of a service mesh that manages the ingress and egress traffic for each microservice, abstracting itself from the application logic of the microservices. Since all the traffic flows through the sidecar proxy, it can monitor all the traffic to send metrics and logs to the centralized control plane.

While the sidecar proxy is the most common approach to create a service mesh, it's not the only approach that is available. There are three approaches to create a service mesh:

- Sidecar Proxy: in a microservices architecture where a proxy is connected to each microservice, with the same life cycle as the microservice itself, but executes as a separate process.

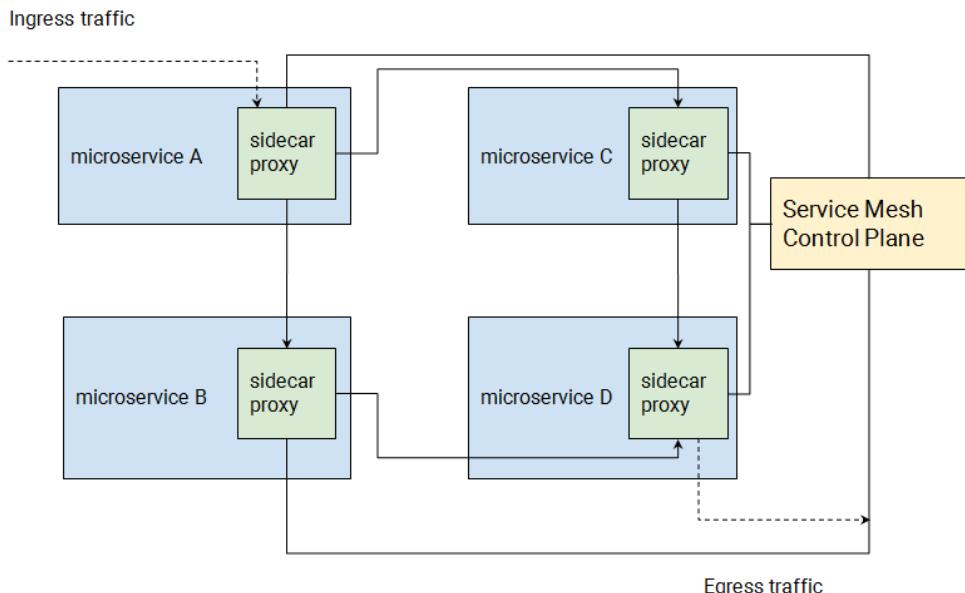


Fig 5.1. Sidecar proxy architecture

- Proxyless: in a microservices architecture where the microservice can send the telemetry directly to the control plane by using gRPC, a remote procedure call (RPC) system developed by Google.
- Proxy inside a VM: an L7 proxy runs inside VMs as a process or an agent which can be added to the service mesh as if it were a sidecar proxy.

After seeing what a service mesh is and what the approaches are to create a service mesh, let's review how Anthos Service Mesh uses each of them.

To monitor in real time the telemetry of all inbound and outbound communications between the microservices of the various service mesh networks, ASM uses two approaches:

- Anthos Service Mesh can use the sidecar proxy approach, using Envoy³ proxies, an open source service proxy attached on each pod to get the real time telemetry;

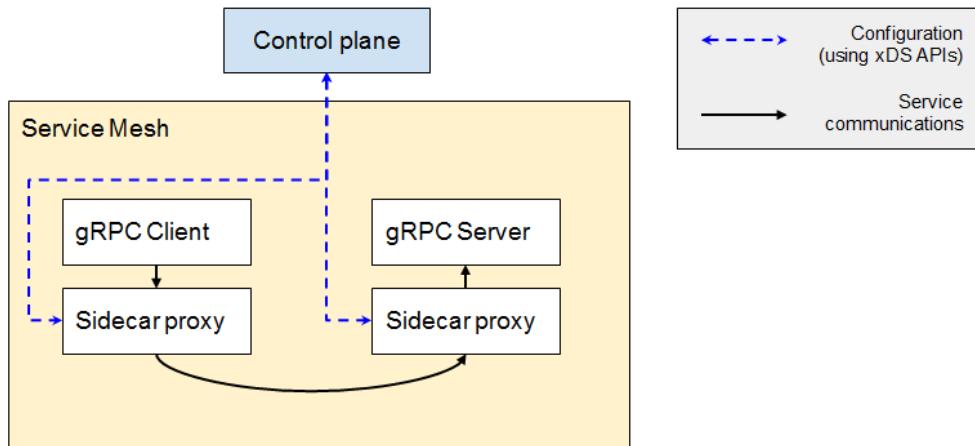


Fig 5.2. Sidecar proxy approach

- Anthos Service Mesh can use a proxyless approach using Google Cloud Traffic Director⁴ that is able to use gRPC, with xDS API⁵, the same technology used by Envoy to communicate with the control plane.

³ <https://www.envoyproxy.io/>

⁴ <https://cloud.google.com/traffic-director>

⁵ <https://github.com/envoyproxy/data-plane-api>

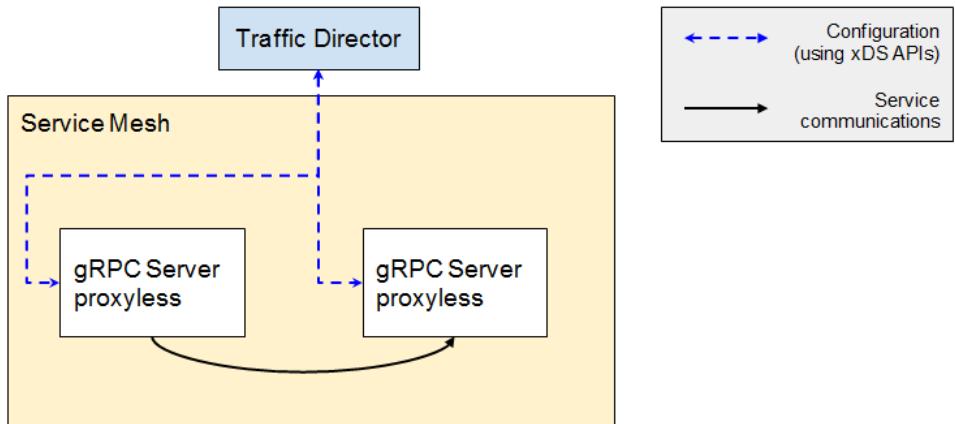


Fig 5.3. Proxyless approach

As shown in figure 5.3, the proxyless approach removes the Istio sidecar from your deployments. By removing the sidecar, we are removing an extra hop in the network traffic to the service, leading to a reduction in network latency, enhancing the service overall response time.

A single service mesh can have services that both use the standard Istio sidecar and the proxyless approach. This allows you to use the correct approach for different applications, including gRPC using a proxyless approach, sidecars for services that do not use gRPC and sidecars for services that use gRPC.

As discussed in previous chapters, Anthos is a complete platform from Google to build applications running on hybrid or multi-cloud platforms. Anthos Service Mesh is the main component that provides service management to developers and cluster administrators.

In the next section, we will examine the features of Istio, which is the basis for the Anthos Service Mesh.

5.3 An Introduction to Istio

Istio is an open platform for providing a uniform way to integrate microservices, manage traffic flow across microservices, enforce policies and aggregate telemetry data. Istio's control plane provides an abstraction layer over the underlying cluster management platform, such as Kubernetes.

The main features⁶ of Istio are:

- Automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic
- Fine-grained control of traffic behavior with robust routing rules, retries, failover, and fault injection

⁶ <https://istio.io/latest/docs/concepts/what-is-istio/#why-use-istio>

- A pluggable policy layer and configuration API supporting access controls, rate limits and quotas
- Automatic metrics, logs, and traces for all traffic within a cluster, including cluster ingress and egress
- Secure service-to-service communication in a cluster with strong identity-based authentication and authorization

Given its open source nature, Istio is extensible and usable in various environments: for example you can execute it on-prem, in the cloud, inside a VM or with microservices and more allowing you to customize the service mesh to your security and monitoring requirements.

To understand Istio, you need to understand the underlying architecture of the system. In the next section, we will explain the components of Istio and the features they provide..

5.3.1 Istio architecture

Istio's flexible architecture allows you to implement a service mesh from scratch. You do not have to have Istio installed before developers start using the cluster, a service mesh can be deployed before or after the developers have implemented deployed their services. Remember, Istio uses the sidecar proxy injection to intercept the ingress and egress traffic from inside and outside the network of microservices, so there are no dependencies on the developers.

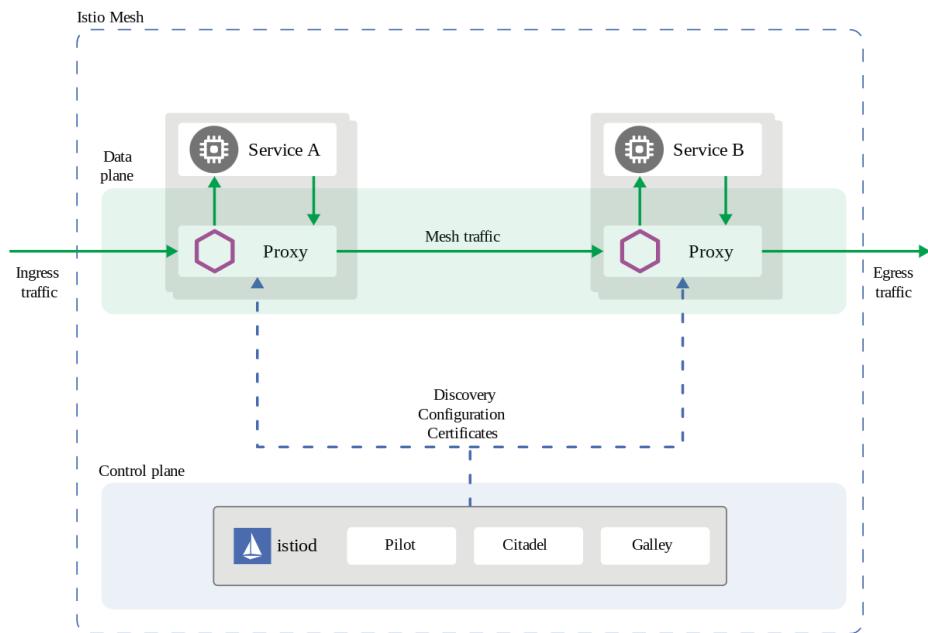


Fig 5.4. Detailed Istio Architecture

Starting with Istio version 1.5, the main components of Istio's architecture are the sidecar proxy and Istiod, which contains three sub-components: *Pilot*, *Citadel* and *Galley*.

Istiod provides service discovery, configuration and certificate management. It can provide high level routing rules that control traffic behavior into a specific configuration for the Envoy proxy, injecting them into the sidecars at runtime. Istiod also acts as a Certificate Authority (CA) that generates certificates to allow secure mTLS communication in the data plane.

Istiod contains three processes to provide its services:

- **Pilot:** Responsible for the lifecycle of Envoy sidecar proxy instances deployed across the service mesh. Pilot abstracts platform-specific service discovery mechanisms and is conformant with any sidecar that can consume Envoy API.
- **Galley:** Interprets the YAML files for Kubernetes and transforms them into a format that Istio understands. Galley makes it possible for Istio to work with other environments than Kubernetes since it translates various configuration data into the common format that Istio understands.
- **Citadel:** Enables robust service-to-service and end-user authentication with built-in identity and credential management.

So far, we have covered Istio at a high level, but now let's go through the features of Istio in detail. We can divide the features into three main categories: traffic management, security and observability.

5.3.2 Istio Traffic Management⁷

Istio provides powerful traffic management that allows users to control ingress and egress traffic. This control is not limited to simply routing traffic to a specific service, it also provides the ability to split traffic between different versions and simulate failures and timeouts in applications. Traffic management consists of the following features:

⁷ <https://istio.io/latest/docs/tasks/traffic-management/>

Table 5.1 - Istio Traffic Management features

Feature	Description
Ingress	Controls the ingress traffic for the service mesh, to expose a service outside the service mesh over TLS or mTLS using the Istio gateway. In the chapter, Hybrid Applications in Anthos: The Edge and Beyond, we will deep dive into Ingress for Anthos.
Egress	Controls the egress traffic from the service mesh, route traffic to an external system, perform TLS for outbound traffic, configure the outbound gateway to use an HTTPS proxy.
Request routing and traffic splitting:	Dynamically route the traffic to multiple versions of the microservice or migrate traffic from one version to another version gradually and in a controlled way.
Fault Injection	Provides configurable HTTP delays and fault injection, allowing developers to discover problems before they would occur in production..

Traffic management is a powerful feature of Istio, allowing developers to completely control traffic, down to the level where a single user could be directed to a new version of an application, while all other requests are directed to the current version. The fault injection feature provides developers the ability to cause a delay between services, simulating an HTTP delay or faults to verify how the application will react to unexpected issues. All of these features can be used by users without any code changes to their application, providing a big advantage over the old "legacy" development days.

Security is everyone's job, but not everyone has a background in security to create the code to enhance application security. Just like the traffic management features, Istio provides extra security features, all without developers needing to create any code.

In the next section, we will explain the security features included with Istio, and ASM.

5.3.3 Istio Security⁸

Since services in the mesh need to communicate with each other over a network connection, you need to consider additional security to defend against various attacks, including man-in-the-middle and unknown service communication. Istio includes components to enhance your application security, ranging from an included certificate authority, peer authentication and authorization.

The first component that Istio provides to increase your application security is the handling of certificate management, including an Istio's certificate authority (CA) with an existing root certificate. In cryptography, a certificate authority⁹ or certification authority is

⁸ <https://istio.io/latest/docs/tasks/security/>

⁹ https://en.wikipedia.org/wiki/Certificate_authority

an entity that issues digital certificates. A digital certificate certifies the ownership of a public key by the named subject of the certificate. This allows others to rely upon signatures or on assertions made about the private key that corresponds to the certified public key - proving the identity of the certificate owner. A CA acts as a trusted third party—trusted both by the owner of the certificate and by the party relying upon the certificate.

Certificates follow a format known as the X.509¹⁰ or EMV standard. In an organization, the root certificate signatory may want to remain responsible for signing all certificates that are issued for all entities in the organization. It is also possible that whoever has the responsibility of signing the root certificate, wants to delegate the responsibility of signing the certificates of a subordinate entity, in this case we refer to the subordinate entity as a "delegate CA".

Istio's certificate authority can be configured in multiple ways, by default, the CA generates a self-signed root certificate and key, which is used to sign the certificates for microservices. Istio's CA can also sign certificates using an administrator-specified certificate and key, and with an administrator-specified root certificate. The last configuration is most common in enterprise environments, where the CA is configured with an existing root certificate or delegated CA signing certificate and key.

The CA in Istio is used to securely provision strong identities to every workload in the mesh. Certificates are issued using the X.509 certificates, which is a standard that defines the format of public key certificates. X.509 certificates are used in many Internet protocols, including TLS/SSL, which is the basis for HTTPS, the secure protocol for browsing the web.

Istio agents, running alongside each Envoy proxy, work together with the Istio CA component of Istiod to automate key and certificate rotation at scale. Since the rotation and distribution of certificates is automated, once configured, there is little overhead for operators or users of the cluster. This is a powerful feature of Istio to secure communications between services.

Certificates are the building block for additional security in our service mesh. In the next section we will discuss authentication and mutual TLS encryption - a security layer that relies on the issued certificates to secure the mesh workloads.

ISTIO AUTHENTICATION

Istio uses peer authentication for service-to-service authentication and to verify the client initiating the connection. Istio also makes mutual TLS (mTLS) available as a full stack solution for transport authentication, which can be enabled without requiring changes to any application code. Peer authentication provides:

- Each service with a strong identity that represents its role to enable interoperability inside the clusters.
- Protection of all communication between services.
- A key management system to automate the generation, distribution and rotation of keys and certificates.

¹⁰ <https://en.wikipedia.org/wiki/X.509>

Istio allows request-level authentication with JSON Web Token (JWT) validation with many authentication providers (eg. Google Auth¹¹)

ISTIO AUTHORIZATION

Istio provides a mechanism for operators to define authorization policies to control access to the service mesh, namespace and workloads within the service mesh. Traffic can be restricted by type, such as TCP or HTTP, and the identity of the requestor.

The advantages that authorization provides are:

- Authorization between workloads and from user to workload;
- Flexible semantics: operators can define custom conditions on Istio attributes and use the DENY and ALLOW actions to tune the policies to suit their needs.
- High Performance: Istio authorization is applied natively on the Envoy proxy.
- Flexibility: Supports gRPC, HTTP, HTTPS and HTTP2 natively as well as all regular TCP protocols.
- Distributed: each Envoy proxy runs its own authorization engine that authorizes each request to be executed.

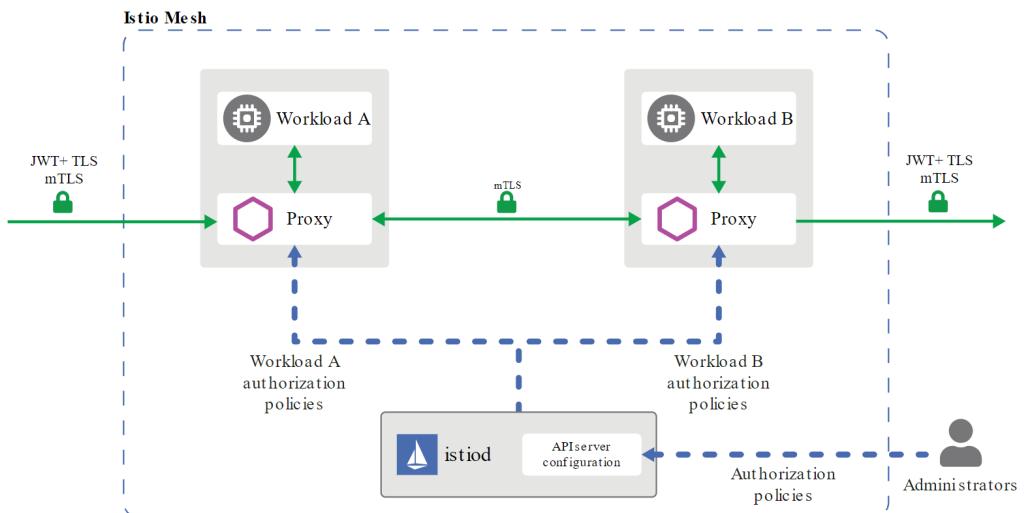


Fig 5.5. Istio Authorization architecture

At the beginning of the chapter we mentioned that one advantage of a service mesh was the ability to organize microservices. As your number of services grows, so does the complexity and the only way to maintain health and to troubleshoot issues in services is to have a powerful set of tools that allow you to look deep into the mesh activities. In the next

¹¹ <https://developers.google.com/identity>

section, we will go over the tools that you can use with Istio to view the activities in the mesh.

5.3.4 Istio Observability¹²

To offer a view into the service mesh, Istio has multiple add-on components that provide distributed tracing, metrics and logging, and a dashboard.

Istio offers a few options that provide a mesh with distributed tracing. Distributed tracing allows you to track the user through all the services and understand request latency, serialization and parallelism. Istio can be configured to send distributed metrics to different systems including Jaeger¹³, Zipkin¹⁴ or Lightstep¹⁵.

Jaeger is a distributed tracing system released as open source by Uber Technologies. It is used for monitoring and troubleshooting microservices-based distributed systems, including those features: distributed context propagation, distributed transaction monitoring, root cause analysis, service dependency analysis and performance optimization.

Zipkin and Lightstep are other distributed tracing systems. They help gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data.

Metrics and Logs: collect all the metrics and logs from Envoy proxy and TCP session, customize the metrics using Istio Metrics making available all the data via Kiali¹⁶, Prometheus¹⁷ or Grafana¹⁸.

Kiali is a management console for Istio-based service meshes.

¹² <https://istio.io/latest/docs/tasks/observability/>

¹³ <https://www.jaegertracing.io/>

¹⁴ <https://zipkin.io/>

¹⁵ <https://lightstep.com/>

¹⁶ <https://kiali.io/>

¹⁷ <https://prometheus.io/>

¹⁸ <https://grafana.com/>

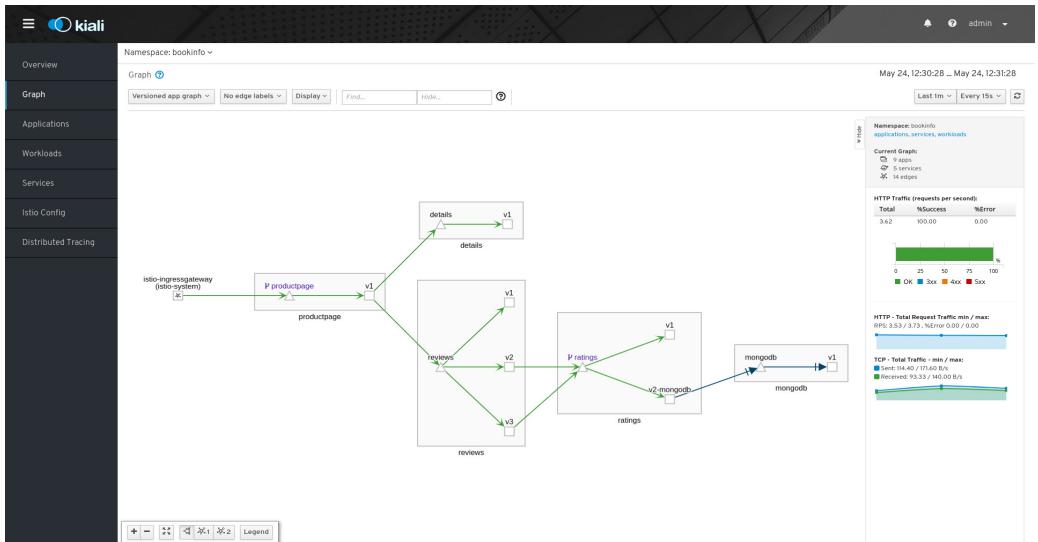


Fig 5.6. Kiali Console UI

It provides dashboards, observability and lets you operate your mesh with robust configuration and validation capabilities. It shows the structure of your service mesh by inferring traffic topology and displays the health of your mesh. Kiali provides detailed metrics, powerful validation, Grafana access, and strong integration for distributed tracing with Jaeger. Kiali allows you to use Kubernetes JWT tokens to provide native RBAC permission. The JWT presented by the user allows access to all namespaces they have access to in the cluster while denying all users that do not have permissions to the namespaces - all without any configuration by cluster admins.

Prometheus is an open-source systems monitoring and alerting that has those principal features: a multi-dimensional data model with time series data identified by metric name and key/value pairs; a flexible query language to leverage this dimensionality named PromQL. The time series collection happens via a pull model over HTTP and pushing time series is supported via an intermediary gateway.

Grafana is open source visualization and analytics software. It allows you to query, visualize, alert on, and explore your metrics no matter where they are stored. It provides you with tools to turn your time-series database (TSDB), data into beautiful graphs and visualizations. Grafana can connect with Prometheus and Kiali.

Now that we have taken a look at Istio, let's see what features and advantages of Istio are when it is used by Anthos Service Mesh, managed by Google Cloud.

5.4 What is Anthos Service Mesh?

Anthos Service Mesh has a suite of features and tools that help you observe and manage secure, reliable services in a unified way. With Anthos Service Mesh, you get an Anthos

tested and supported distribution of Istio, managed by Google, letting you create and deploy a service mesh on GKE on Google Cloud and other platforms with full Google support.

The use of Istio features, in Anthos Service Mesh, varies according to the architecture you want to design and implement - *full cloud, multi cloud, hybrid cloud, or edge*. Each implementation will have different available features, therefore, it is necessary to check the availability of the supported features¹⁹ for the various scenarios.

Before installing Anthos Service Mesh, always check the documentation and choose the most suitable and updated configuration profile. Configuration profiles are YAML files that are used by the IstioOperator API, defining and configuring the features that are installed with Anthos Service Mesh.

At time of writing you can install ASM in different scenarios:

- Anthos cluster (GKE) on Google Cloud in a single project
- Anthos cluster (GKE) on Google Cloud between different projects
- Anthos cluster (GKE) on VMware
- Anthos cluster (GKE) on bare metal
- Anthos cluster (GKE) on AWS
- Attached cluster Amazon Elastic Kubernetes Service (Amazon EKS)
- Attached cluster Microsoft Azure Kubernetes Service (Microsoft AKS)

5.5 Installing ASM

ASM installs differently on GKE clusters on GCP and on-premise. You can view the most current installation procedures on the ASM site at <https://cloud.google.com/service-mesh/docs/unified-install/install-anthos-service-mesh>. Explaining each option of the installation is beyond the scope of a single chapter, but the steps to deploy ASM on a GKE cluster with all components for testing only requires a few steps - The steps below will install ASM 1.12 in a cluster:

1. Downloading the ASM installation scripts

```
curl https://storage.googleapis.com/csm-artifacts/asm/asmcli_1.12 > asmcli
```

2. Make the script executable

```
chmod +x asmcli
```

3. Install ASM using the install_asm script

```
./asmcli install --project_id PROJECT_ID --cluster_name CLUSTER_NAME --  
cluster_location CLUSTER_LOCATION --output_dir ./asm-downloads --enable_all
```

After Istio is deployed into the Kubernetes cluster, it is possible to start configuring and using it right away. One of the first things to do is to define which approach we want to follow to make proxies communicate within the service mesh.

In the next section, we will define how Istio will handle the sidecar proxy injection.

¹⁹ <https://cloud.google.com/service-mesh/docs/supported-features>

5.5.1 Sidecar proxy injection

Activating Anthos Service Mesh features is an easy, transparent process, thanks to the possibility of injecting a sidecar proxy next to each workload or microservice.

You can inject a sidecar proxy manually by updating your Pods' Kubernetes manifest or you can use automatic sidecar injection. By default, sidecar auto-injection is disabled for all namespaces. To enable auto-injection for a single namespace, you can execute the command below:

```
kubectl label namespace NAMESPACE istio.io/rev=asm-managed --overwrite
```

where `NAMESPACE` is the name of the [namespace](#) for your application's services and "rev=asm-managed" is the release channel²⁰.

All channels are based on a generally available (GA) release (although individual features may not always be GA, as marked). New Anthos Service Mesh versions are first released to the Rapid channel, and over time are promoted to the Regular, and Stable channel. This allows you to select a channel that meets your business, stability, and functionality needs.

After you execute the command, because sidecars are injected when Pods are created, you must restart any running Pods for the change to take effect. When Kubernetes invokes the webhook, the admissionregistration.k8s.io/v1beta1#MutatingWebhookConfiguration configuration is applied. The default configuration injects the sidecar into pods in any namespace with the `istio-injection=enabled` label. The label should be consistent with the previous command. The `istio-sidecar-injector` configuration map specifies the configuration for the injected sidecar.

The way you restart Pods depends very much on the way they were created such as:

1. If you used a Deployment, you should update or recreate the Deployment first, which will restart all Pods, adding the sidecar proxies:

```
kubectl rollout restart deployment -n YOUR_NAMESPACE
```

2. If you didn't use a Deployment, you should delete the Pods, and they will be automatically recreated with sidecars:

```
kubectl delete pod -n YOUR_NAMESPACE --all
```

3. Check that all the Pods in the namespace have sidecars injected:

```
kubectl get pod -n YOUR_NAMESPACE
```

4. In the following example output from the previous command, you will notice that the `READY` column indicates there are two containers for each of your workloads: the **primary container** and the container for the **sidecar proxy**.

NAME	READY	STATUS	RESTARTS	AGE
YOUR_WORKLOAD	2/2	Running	0	20s

²⁰ https://cloud.google.com/service-mesh/docs/release-channels-managed-service-mesh#available_release_channels

We have seen how to install Anthos Service Mesh, using the approach with a sidecar proxy and how important it is to choose the right profile. Now let's see what the other features of Anthos Service Mesh are and what the advantages are of using them.

5.5.2 Uniform Observability

One of the most important and useful features of the Anthos Service Mesh is observability. Implementing a service mesh through the proxy architecture and taking advantage of the Google Cloud Monitoring services ensures an in-depth visibility of what is happening among the various microservices that are present in the mesh.

Through the proxy, each microservice can send telemetry automatically, without the developers adding any code in their application. All traffic is intercepted by proxies and the telemetry data is sent to Anthos Service Mesh. Each proxy sends the data to Google Cloud Monitoring and Google Cloud Logging without any extra development using the APIs that Google makes available.

As we have seen in the chapter, *Anthos, the one single glass-of-pane UX*, the Anthos Service Mesh control plane provides two main dashboards: table view and topology view.

In the “table view” you have a complete view and intuitive look at all the services that are deployed in the cluster. You can see all the metrics and you can add SLI and SLO to better monitor your services.

In the “topology” view, service meshes are represented as a graphical map. All the services such as workloads, pod, systems services and relative owners are connected as a network of nodes. This enables a comprehensive look at the overall performance of the entire service mesh and an inside look into each node with detailed information.

5.5.3 Operational agility

If observability is one of the most “visible” features to manage a service mesh, then the management of traffic “*to / from*” within a microservices architecture is another fundamental asset to manage operations easily.

Since Anthos Service Mesh is based on Istio, it inherits most²¹ of the traffic and network management features that Istio provides, so let's look at what these features are.

REQUEST ROUTING AND TRAFFIC SPLITTING

Istio provides the ability to redirect traffic to multiple versions of the same microservice deployed in the cluster; the ability to safely (under possible quota) control part of the traffic from an old version of the microservice to a newly installed version. Both of these options allow you to be agile in the deployment of new features or to fix bugs that may impact the business.

For example, let's imagine we need to urgently fix a microservice. After the deployment of the new version, we can redirect a small part of the incoming traffic, verify that the fix works correctly and then completely redirect the traffic to the new version without any downtime. If the fix is carried out only for a specific case, it is possible to keep both versions

²¹ <https://cloud.google.com/service-mesh/docs/supported-features#networking>

of the microservice active, redirecting traffic to both versions based on pre-established rules, without necessarily having to delete the old version that is working well for most cases.

Through these traffic management features, Anthos Service Mesh can manage A/B testing²², allowing you to direct a particular percentage of traffic to a new version of a service. This is a good idea when you want to introduce a new version of a service by first testing it using a small percentage of user traffic, and if all goes well, increase the percentage while simultaneously phasing out the old version.

Thanks to these functionalities, it is possible to implement canary deployments or progressive rollout strategies, directly testing the new versions of the services that are released. If the new version of the service experiences no issues with a small percentage of the traffic directed towards it, you could move all traffic to the new version, discarding the old one.

In the following figure a canary deployment with traffic splitting is used to redirect the 5% of traffic to test the new release of the Service A.

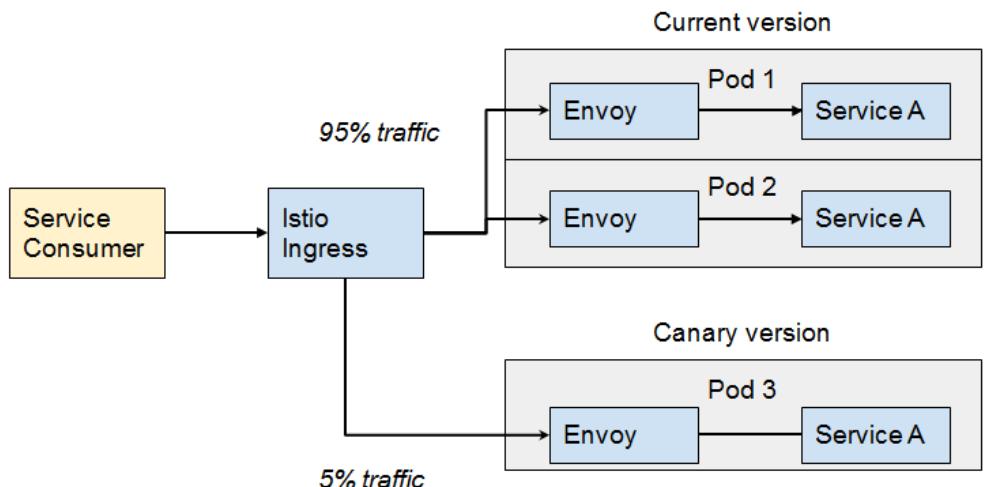


Fig 5.7. Istio Canary Deployment feature based on traffic

In the following figure a canary deployment with traffic splitting is used to redirect the iPhone's traffic to test the new release of the Service A.

²² https://en.wikipedia.org/wiki/A/B_testing

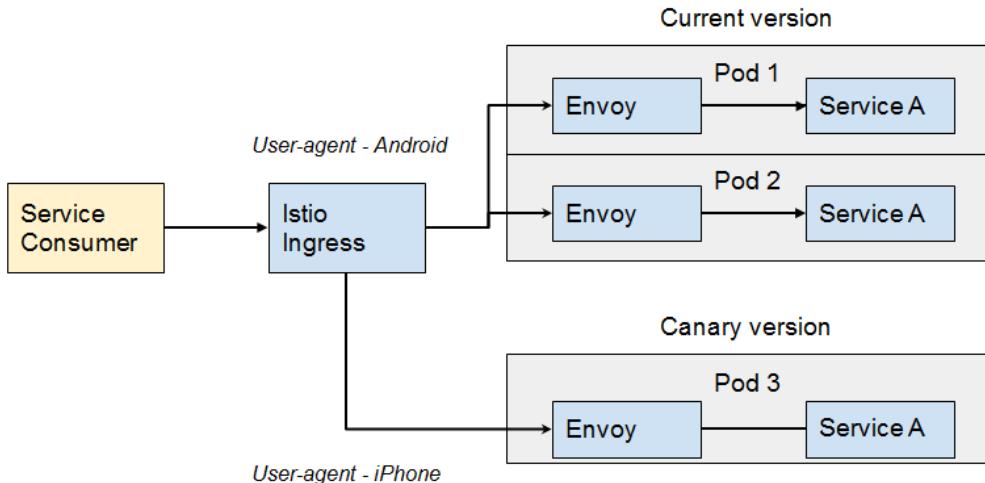


Fig 5.8. Istio Canary Deployment feature based on User-agent

The operations departments that manage the production environment take advantage of these features as they release plans for fixes and new versions of microservices. Each of these scenarios can be executed on the fly without disrupting the end users.

CIRCUIT BREAKING²³

Microservices architectures were born to be scalable, agile and resilient, but it is not always easy to design and implement these architectures to manage high workloads, manage integration with external services with consequent possible downtime or timeout.

As already mentioned above, the service mesh is independent of the application code and the programming language used and consequently this facilitates the adoption of functions dedicated to the management of the office.

The functionality that allows you to manage timeouts, failures and loads on the architecture is called circuit breaking functionality: when you design a microservices architecture you must always put yourself in the condition of being able to manage faults correctly. These faults may have been caused not only by bugs in the application code, but also from external factors. In the event of a fault or failure to reach the SLA (on availability and/or performance of a given service), this feature automatically allows you to redirect traffic to another microservice or external service to limit downtime or to limit the loss of functionality by the final user.

Let's see an example. In figure 5.7, the service consumer is invoking Istio Ingress to call service A that is distributed in two pods. Let's assume that service A inside pod 1 has a load problem and becomes unreachable: thanks to the circuit breaking feature, Istio will close the

²³ <https://istio.io/latest/docs/tasks/traffic-management/circuit-breaking/>

connection of the proxy to service A inside pod 1, redirecting all traffic to the proxy inside the pod 2, until service A in pod 1 works properly again.

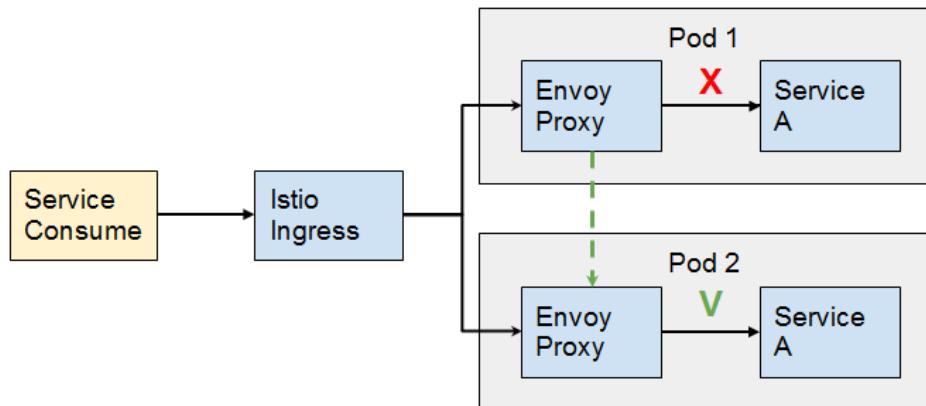


Fig 5.9. Istio Circuit Breaking feature example

EGRESS CONTROL

Usually when it comes to service mesh, a very important feature is the control of the Ingress Gateway to ensure absolute security to the network.

However, if we find ourselves in situations for which the regulations (for example PCI²⁴) or the customer's requirements require us to also control the outgoing traffic from the service mesh, then thanks to Istio and thanks to the egress gateway control it is possible to cover the security required.

With Anthos Service Mesh, it is possible to configure the routing of the traffic from the service mesh to the external services (HTTP or HTTPS), using a dedicated egress gateway or an external HTTPS proxy if necessary; to perform TLS origination (SDS or File mount) from the service mesh for the connection on those external services.

²⁴ https://www.pcisecuritystandards.org/pcl_security/

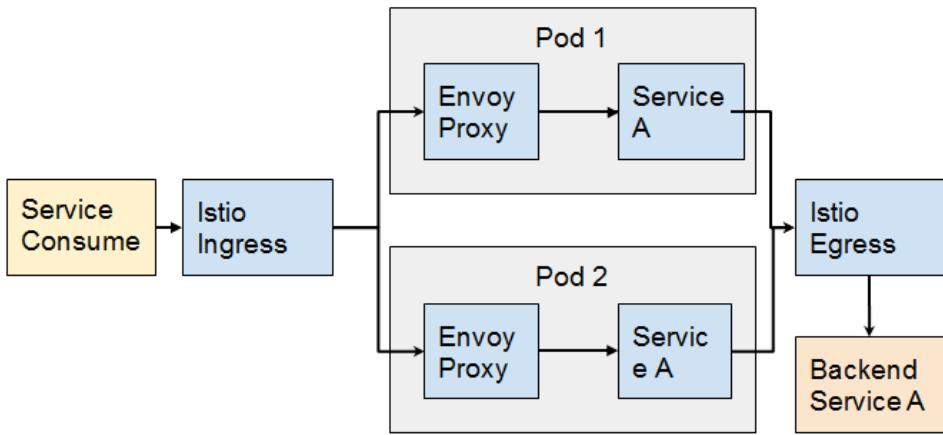


Fig 5.10. Istio Egress control example

5.5.4 Policy-Driven security²⁵

Since the adoption of containers, shared services and distributed architectures, it has become more difficult to mitigate insider threats and minimize and limit data breaches, ensuring that communications between workloads remain encrypted, authenticated and authorized. Anthos Service Mesh security features help mitigate these threats, configuring context-sensitive service levels or context-sensitive networks.

Before offerings like Istio, the requirements to secure an application was the responsibility of the application developer, many of the tasks were complex and time consuming and included:

- Encrypting the communications between the application and other services required certificate management and maintenance
- Creating modules that were language specific to authenticate access based on open identity standards like JSON web tokens (JWT's)
- Implementing a complex authorization system to limit the permissions allowed using the assertions on the presented JWT

Instead of a developer taking time to create and manage this security, they can leverage the features of Istio, which addresses each of these without any additional code required.

With Anthos Service Mesh it is possible to adopt a defense in-depth posture consistent with Zero Trust security principles. It allows you to reach this position via declarative criteria and without modifying any application code. The principal security features in ASM are the managed private certificate authority, identity-aware access controls, Request claims-aware access control policies, User authentication with Identity-Aware proxy and access logging and monitoring.

²⁵ <https://cloud.google.com/service-mesh/docs/security-overview>

The first feature, the managed private certificate authority (Mesh CA) includes a Google-managed multi-regional private certificate authority, for issuing certificates for mTLS. The Mesh CA is a highly reliable and scalable service, managed by Google, that is optimized for dynamically scaled workloads on a cloud platform. Mesh CA lets you rely on a single root of trust across Anthos clusters. When using Mesh CA, you can rely on workload identity pools to provide coarse-grained isolation. By default, authentication fails if the client and the server are not in the same workload identity pool.

The next feature, the Identity-aware access control (firewall) policies allows you to configure network security policies that are based on the mTLS identity versus the IP address of the peer. This lets you create policies that are independent of the network location of the workload. Only communications across clusters in the same Google Cloud project are currently supported.

The third feature, request claims-aware access control (firewall) policies, allow you to grant access to services based on request claims in the JWT header of HTTP or gRPC requests. Anthos Service Mesh lets you assert that a JWT is signed by a trusted entity. This means that you can configure policies that allow access from certain clients only if a request claim exists or matches a specified value.

The fourth feature, user authentication with Identity-Aware Proxy, provides authentication of users that are accessing any service exposed on an Anthos Service Mesh Ingress Gateway by using Identity-Aware Proxy (IAP). IAP can authenticate users that log in from a browser, integrate with custom identity providers, and issue a short-lived JWT token or an RCToken that can then be used to grant access at the Ingress Gateway or a downstream service (by using a sidecar).

The final features, access logging and monitoring, ensures that access logs and metrics are available in Google Cloud's operations suite, and provides an integrated dashboard to understand access patterns for a service or workload based on this data. You can also choose to configure a private destination. Anthos Service Mesh lets you reduce noise in access logs by only logging successful accesses once in a configurable time window. Requests that are denied by a security policy or result in an error are always logged. This lets you significantly reduce the costs associated with ingesting, storing, and processing logs, without the loss of key security signals.

5.6 Conclusion

In this chapter, we have seen what a service mesh is, what are the advantages of implementing it and how Anthos Service Mesh exploits the potential of Istio to manage the entire service mesh.

Thanks to Anthos Service Mesh, the developers are more agile in implementing microservices architectures and don't need to worry about implementing monitoring probes within the application code, but taking advantage of sidecar proxy and proxyless approaches.

The operations structures are able to monitor everything that happens within the service mesh in real time , guaranteeing the required service levels. The traffic splitting and rolling release features allow you to efficiently release new versions of the services, ensuring that everything works correctly. Thanks to the security features, the service mesh is protected

from risks that can come from outside or inside the network, implementing effective authentication and authorization policies.

5.7 Examples and Case Studies

Using the knowledge from the chapter, address each of the requirements in the case study found below.

5.7.1 Evermore Industries

You have been asked by Evermore Industries to enable ASM on a GKE cluster running in GCP. The cluster will be used for initial service mesh testing and should be installed with all features to allow the developers to test any feature. They have also asked you to deploy Online Boutique application to prove that the service mesh is up and running as expected.

Since they are new to Istio and the advantages of using a service mesh, they do not have any special requirements outside of deploying ASM and the Boutique demo application. The only additional requirement is to provide proof that the Boutique application is running in the mesh as expected from the GCP console.

The next section contains the solution to address Evermore's requirements. You can follow along with the solution, or if you are comfortable, configure your cluster to address the requirements and use the solution to verify your results.

EVERMORE INDUSTRIES SOLUTION - INSTALLING ASM

To install ASM, you can download the ASM installation script to deploy ASM with all components installed. You can follow the steps below to install ASM with all components on your GKE cluster running in GCP.

1. Download the ASM installation script

```
curl https://storage.googleapis.com/csm-artifacts/asm/asmcli_1.12 > asmcli
```

2. Make the installer executable

```
chmod +x asmcli
```

You will need the following information from your project and GKE cluster to execute the installation script: Project ID, GKE cluster name, and the GKE cluster location.

Execute the installation script with the information from your cluster to install ASM.

```
./asmcli install --project_id gke-test1-123456 --cluster_name gke-dev-001 --  
cluster_location us-central1-c --output_dir ./asm-downloads --enable_all
```

The installation will take a few minutes. Once the installation is complete, you will see a message similar to the one shown below.

```
asmcli: Successfully installed ASM.
```

3. Verify that the istio-system namespace has healthy pods that have been started successfully.

```
kubectl get pods -n istio-system
```

This should show you that there are four running pods, (2) istio-ingressgateway pods and (2) istiod pods, an example output is shown below.

NAME	READY	STATUS
istio-ingressgateway-68fb877774-9tm8j	1/1	Running istio-ingressgateway-68fb877774-qf5dp
istiod-asm-1124-2-78fb6c7f98-n4xpp	1/1	Running
istiod-asm-1124-2-78fb6c7f98-sgttk	1/1	Running

Now that Istio has been deployed, you need to create a namespace and enable Istio for sidecar injection.

EVERMORE INDUSTRIES SOLUTION - ENABLING SIDECAR INJECTION

To enable the namespace for sidecar injection, follow the steps below:

1. Create a namespace that will be used to deploy the Boutique application. In our example, we will use a namespace called demo.

```
kubectl create namespace demo
```

2. Next, we need to label the namespace with the correct label to enable sidecar injection. Starting with Istio 1.7, the label used to enable sidecar injection changed from a generic istio-injection, to using the value of the control plane version.

To find the control plane version we will use in the label, retrieve the labels in the istio-system namespace.

```
kubectl -n istio-system get pods -l app=istiod --show-labels
```

This will return the labels of the istiod pods, which will contain the value we need. The output will look similar to the below example. (Note: the label value we will need has been highlighted in bold)

NAME	READY	STATUS	RESTARTS	AGE	LABELS
istiod-asm-1124-2-78fb6c7f98-n4xpp	1/1	Running	0	44m	app=istiod,install.operator.istio.io/owning-resource=unknown, istio.io/rev=asm-1124-2 ,istio=istiod,operator.istio.io/component=Pilot,pod-template-hash=78fb6c7f98,sidecar.istio.io/inject=false
istiod-asm-1124-2-78fb6c7f98-sgttk	1/1	Running	0	44m	app=istiod,install.operator.istio.io/owning-resource=unknown, istio.io/rev=asm-1124-2 ,istio=istiod,operator.istio.io/component=Pilot,pod-template-hash=78fb6c7f98,sidecar.istio.io/inject=false

3. Using the `istio.io` value, label the demo namespace to enable sidecar injection.

```
kubectl label namespace demo istio.io/rev=asm-1124-2
```

EVERMORE INDUSTRIES SOLUTION - INSTALLING THE BOUTIQUE APPLICATION

Now that ASM has been installed and we have created a new namespace with the correct label to enable sidecar injection, we can deploy the Boutique application. Follow the steps below to deploy the Google Boutique demo.

1. Download the Boutique demo application from the GIT repository. The command below will download the GIT repo into a directory called `online-boutique`.

```
kpt pkg get https://github.com/GoogleCloudPlatform/microservices-demo.git/release  
online-boutique
```

2. Deploy the application using the files in the `online-boutique` directory.

```
kubectl apply -n demo -f online-boutique
```

This will install a number of deployments and services into the demo namespace. It will take a few minutes for the pods to start up. You can watch the namespace or list the pods in the namespace to verify that each pod enters a running state, and that each pod shows 2 containers. (Remember from the chapter that each pod will have a sidecar injected for the Istio proxy)

Once all pods are running, the demo namespace output should look similar to the output below.

NAME	READY	STATUS
adservice-6b74979749-2qd77	2/2	Running
cartservice-6fc79c6d86-tvncv	2/2	Running
checkoutservice-7c95787547-8dmzw	2/2	Running
currencyervice-67674dbdf7-hkw78	2/2	Running
emailservice-799966ff9f-qcb6s	2/2	Running
frontend-597d957cdf-dmdwr	2/2	Running
loadgenerator-88f7dbff5-cn78t	2/2	Running
paymentservice-5bdd645d9f-4w9f9	2/2	Running
productcatalogservice-7ffbf4fbf5-j98sq	2/2	Running
recommendationservice-599dfdc445-gpmww	2/2	Running
redis-cart-57bd646894-tdxwb	2/2	Running
shippingservice-5f4d856dc-cwtcl	2/2	Running

As part of the deployment, a load-balancer service was created that allows connectivity to the Boutique application from the Internet. To find the IP address that has been assigned, get the service called frontend-external in the demo namespace.

```
kubectl get services frontend-external -n demo
```

This will output the service details which will contain the external address that you can use to verify the application is working.

Use the address from the output in step 3 to connect to the application from a web browser. Once you connect, you should see the main Boutique page.

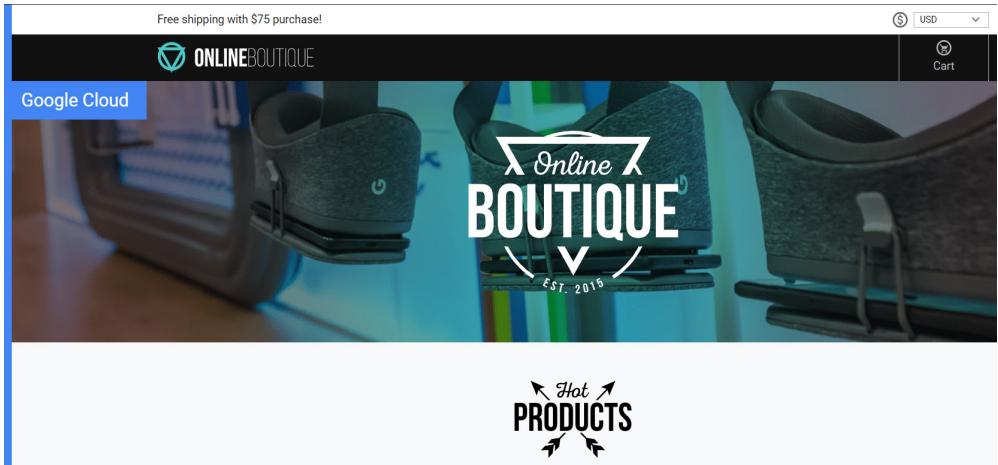


Fig 5.11. Online Boutique Web application

EVERMORE INDUSTRIES SOLUTION - OBSERVING SERVICES USING THE GCP CONSOLE

The final requirement from Evermore is to prove that the Boutique application is running in the mesh, using the GCP console. To prove this, you can use the Anthos -> Service Mesh from the GCP console.

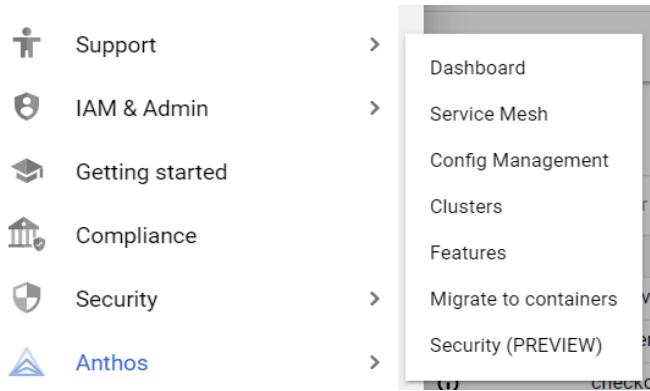


Fig 5.12. Google Cloud Console - Anthos menu

This will provide a list of all services in the service mesh. The default table view will show each service and metrics including requests, latency and failures.

Services						
Filter Select a filter option						
Status	Name	Namespace	Types	Clusters	Requests/sec (avg)	
ⓘ	adservice	demo	K8s	gke-dev-001	0.9	
ⓘ	cartservice	demo	K8s	gke-dev-001	1.5	

Fig 5.13. Anthos Service Mesh - List of services

You can change the view from the table view to a topology view by clicking the topology button in the upper right hand corner of the console. This will provide a graphical topology layout of the mesh services.

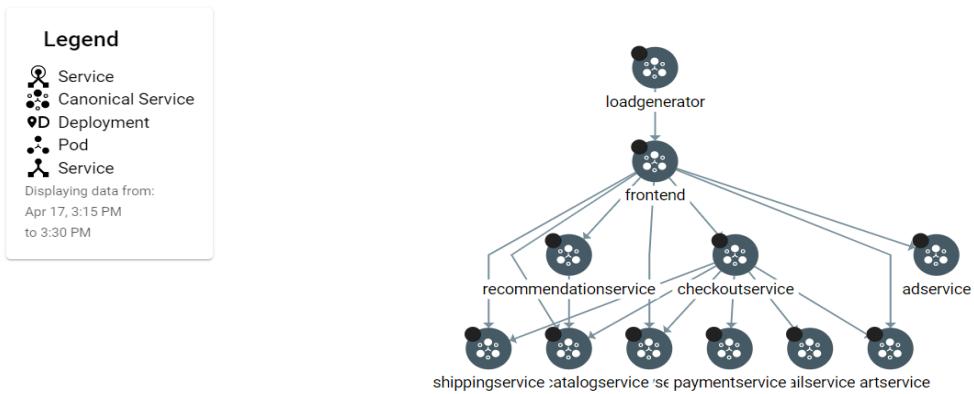


Fig 5.14. Anthos Service Mesh - Topology view

Both of these views will only show services in the service mesh. Since we see the expected services for the Boutique application, this proves that the deployment is successfully working inside of the mesh. This completes the exercise. In this exercise you deployed ASM in a GKE cluster, created a new namespace that was enabled for sidecar injection, and deployed a test application into the service mesh.

5.8 Summary

In this chapter you got to learn that a service mesh based on Anthos Service Mesh helps organizations run microservices at scale by providing:

- a more flexible release process with canary deployments and A/B testing
- availability and resilience with circuit breakers, traffic splitting and fault injection
- In-depth visibility, monitoring, security and control of the entire service mesh and between multiple environments

6

Operations management in Anthos

by Jason Quek

This chapter covers:

- Using the unified Google cloud interface to manage Kubernetes clusters
- Managing Anthos clusters
- Logging and monitoring
- Anthos deployment patterns

Operations is the act of ensuring your clusters are functioning, active, secure and able to serve the application to the users. To that end, there is one prevailing school of thought that has gained adoption and momentum in the Cloud era, an operations practice known as DevOps.

The simplest definition of DevOps is the combination of developers and IT operations. DevOps aims to address two major points, the first is to enable continuous delivery through automated testing, frequent releases and management of the entire infrastructure as code. The second, an often overlooked part of DevOps, is IT operations, which includes tasks like logging, monitoring, and using those indicators to scale and manage the system.

Google developed an approach before the development of DevOps called Site Reliability Engineering (SRE). This approach addresses the requirements that enable DevOps, which automates and codifies all tasks in operating the infrastructure to enhance reliability in the systems, and if something goes wrong, it can be repaired automatically through code. A SRE team is responsible for not only keeping the environment stable, but it should also be the team that is responsible for new operational features and improvements to the infrastructure.

Both schools of thought have different responsibilities assigned to different teams, however they have the same goal, which is to be able to implement changes rapidly and

efficiently, automate where possible and continuously monitor and enhance the system. This commonality underlies the “desire” from the engineering and operations team to break silos and take common responsibility for the system.

Either approach will deliver many of the same advantages, but they solve problems in different ways. For example, in a DevOps approach, there may be a dedicated operations team that takes care of the operations management aspect of the infrastructure, handing issues off to another development team to resolve. This differs from the SRE approach, where operations are driven by the development team and approached from a software engineering point of view, allowing for a single SRE team to address issues within their own team.

Anthos provides a path to build a strong DevOps or SRE culture by usage of the tools provided in the framework. This chapter will show product owners, developers and infrastructure teams that through using Anthos, they are able to build a DevOps / SRE culture that will help to reduce silos in their company, build reliable systems and enhance development efficiency.

In the next section, we will explain the tools that Anthos includes, starting with the Unified User Interface through Google Cloud Console, then Centralized logging and Monitoring, and Envirs, which are all key concepts that will provide the building blocks to enable an operations practice.

6.1 Unified User Interface from Google Cloud Console

With everything-as-code these days, there is a certain pride that a software engineer takes in doing everything from the command line, or, as code. However, when a production issue occurs affecting real-life services, an intuitive and assistive user interface can help an engineer identify the issue quickly.

This is where Google’s unified user interface comes in handy, as seen in Figure 6.1.

Total cores	Total memory	Notifications	Labels		
10 CPU	39.29 GB		location : hamina provider : gcp	<button>Log out</button>	
6 CPU	12.22 GB		location : paris provider : aws	<button>Log out</button>	
12 CPU	25.08 GB	Upgrade available	location : stockholm provider : onprem	<button>Log out</button>	

Fig 6.1. Multiple clusters registered to Google Cloud Console

These tools often allow you to view multiple items, like Kubernetes clusters, in a single view. Having this view available to an administrator gives an oversight of all the resources available, without having to log into three separate clusters as seen from Figure 6.1. This view also shows where they are located, which are their providers and actions required to manage the clusters.

To access this view requires that the user is already logged into Google Cloud console, which is secured by Google Cloud Identity, providing an additional layer of security to build defense in-depth against malicious actors.

Having access to this type of view fulfills one of the DevOps principles, leveraging tooling to provide observability into the system. For more details, please refer to *Chapter 3 Anthos, the one single glass-of-pane UX*.

To have a single pane of glass view, you will need to register your clusters with your GCP project. In the next section, we will cover how to register a cluster that is running Anthos on any of the major Cloud Service Providers or on-prem.

6.1.1 Registering clusters to Google Cloud Console

The component responsible for connecting clusters to Google Cloud Console is called Connect, which is often deployed as one of the last steps after a cluster is created. If an Anthos on GKE / AWS / Azure / on-prem is deployed, the Connect agent is automatically deployed at cluster creation time. However if the cluster is not deployed by Anthos, such as EKS, AKS, OpenShift, Rancher clusters, the agent will have to be deployed separately, as Anthos is not involved in the installation process. This will be covered later in this chapter on Anthos attached clusters.

As Anthos is built with the best practices from Kubernetes, the Connect Agent is represented as a Kubernetes Deployment, with the image provided by Google as part of the Anthos framework. This means the agent can also be seen from the Google Cloud Console, and can be managed like any other Kubernetes object, as shown in figure 6.2 below.

gke-connect-agent-20200515-02-00

[Overview](#) [Details](#) [Revision history](#) [Events](#) [YAML](#)

Cluster	finland-gke-cluster
Namespace	gke-connect
Labels	app: gke-connect-agent, hub.gke.io/project: anthos-sandbox-256114, version: 20200515-02-00
Replicas	1 updated, 1 ready, 1 available, 0 unavailable
Pod specification	Revision 1, containers: gke-connect-agent-20200515-02-00 , volumes: creds-gcp , http-proxy

Active revisions

Revision	Name	Status	Summary	Created on	Pods running/Pods total
1	gke-connect-agent-20200515-02-00-5865ddc9f5	✓ OK	gke-connect-agent-20200515-02-00: gcr.io/gkeconnect/gkeconnect-gce:20200515-02-00	25 May 2020, 01:32:41	1/1

Managed pods

Revision	Name	Status	Restarts	Created on
1	gke-connect-agent-20200515-02-00-5865ddc9f5-jpgfr	✓ Running	0	28 May 2020, 21:33:34

Fig 6.2. Connect agent deployed on GKE cluster

The Connect agent acts as a conduit for Google Cloud Console to issue commands to the clusters where it has been deployed on and to report vCPU usage for licensing. This brings up one important point, which is that the clusters need to be able to reach Google Cloud APIs (egress), however, the clusters do not need to be reachable by Google Cloud APIs (ingress).

So, how does Google Cloud Console issue Kubernetes API commands, such as listing pods to display on the Google Cloud Console? The answer is through the Connect Agent which establishes a persistent TLS 1.2 connection to GCP to wait for requests, which eliminates the need of having an inbound firewall rule to the user cluster.

For those who are unfamiliar with TLS (Transport Layer Security), it is a cryptographic protocol designed to provide privacy and data integrity between the sender and receiver. It utilizes symmetric encryption based on a shared secret to ensure that the message is private. Messages are signed with a public key to ensure its authenticity, and also includes a message integrity check to ensure that messages are complete. In short, the communication channel to the Connect agent over the internet is as secure as internet bank transfers. The full communication flow can be seen in *Chapter 3 Anthos, one single glass-of-pane UX*.

One important point to note is that outbound TLS encrypted connection over the internet is used for Anthos deployments to communicate with Google Cloud as shown in Figure 6.3. This simplifies things, as no inbound firewalls rules have to be added to Anthos deployments, only outbound traffic towards Google Cloud, without any virtual private networks (VPN) requiring set up.

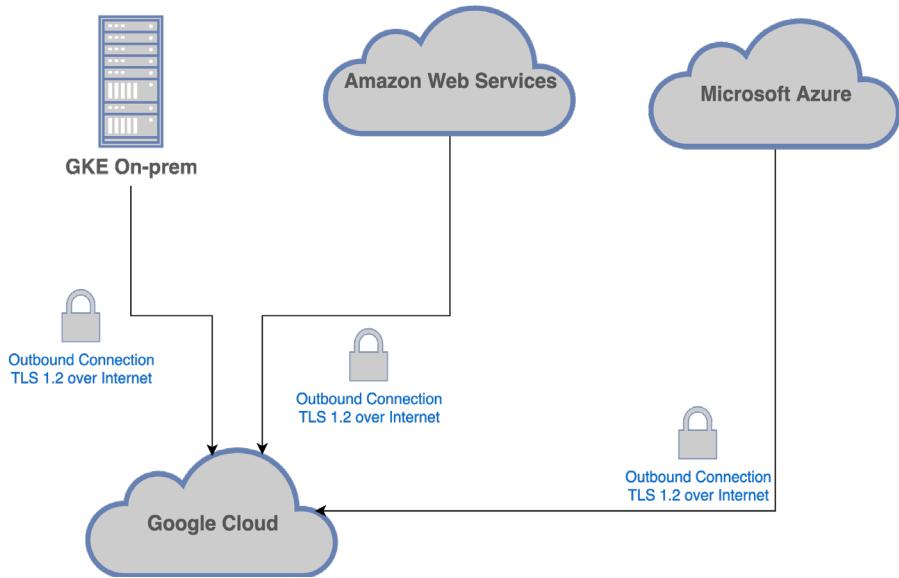


Fig 6.3. Outbound connection to Google Cloud from Anthos deployments

One great thing about Kubernetes is the standardization, which means this agent will be able to issue Kubernetes API commands on clusters created by Google or **any** provider which provides a compliant Kubernetes distribution as defined by the Cloud Native Com.

A common issue in large enterprises is that IT security would want to know what is the Connect agent sending to Google Cloud APIs, and this is a tricky issue to overcome, due to the perceived worry that if Google shares the keys to decrypt the traffic, it is effectively overriding the security put into place. More details of what information is actually sent from the Connect agent to Google Cloud can be found in this white paper by Google (<https://cloud.google.com/files/security-features-for-connect-for-anthos.pdf>). Google has also stated unequivocally that no customer data is sent via the Connect Agent, and that it is only to provide functionality to communicate with the Kubernetes API and also provide licensing metrics for Anthos billing.

6.1.2 Authentication

Authentication to Google Kubernetes Engine should utilize Identity and Access Management roles to govern access to the GKE clusters. The following section pertains to GKE On-prem, GKE on AWS, GKE on Azure and Anthos attached clusters.

To access Anthos clusters, users with access to the project will always have to provide either a Kubernetes Service Account (KSA) token, basic authentication or authenticate against an identity provider configured for the cluster.

Using a kubernetes service account token would be the easiest to set up, but requires token rotation, and a secure way to distribute tokens regularly to the users which need

access to the clusters. Using basic authentication would be the least secure due to having password management requirements, but it is still supported as an authentication method if an Identity Provider is not available. If you have the need to use basic authentication, one tip would be to implement a password rotation strategy in the event of password leaks.

The recommended practice would be to set up OpenID Connect (OIDC) with Google Cloud Identity so that users can leverage their existing security setup to manage access to their clusters as well. As of Sep 2020, OIDC is supported on GKE on-prem clusters from the command line (not from the console). So a solid KSA token rotation and distribution strategy is highly recommended, which can be as simple as utilizing Google Secret Manager, where permissions to retrieve the token can be controlled via IAM permissions, and the token can be updated every seven days using Cloud Scheduler.

Once OIDC with Google Cloud Identity has been set up, users are able to authenticate to the user clusters using the gcloud CLI or from the Google Cloud Console.

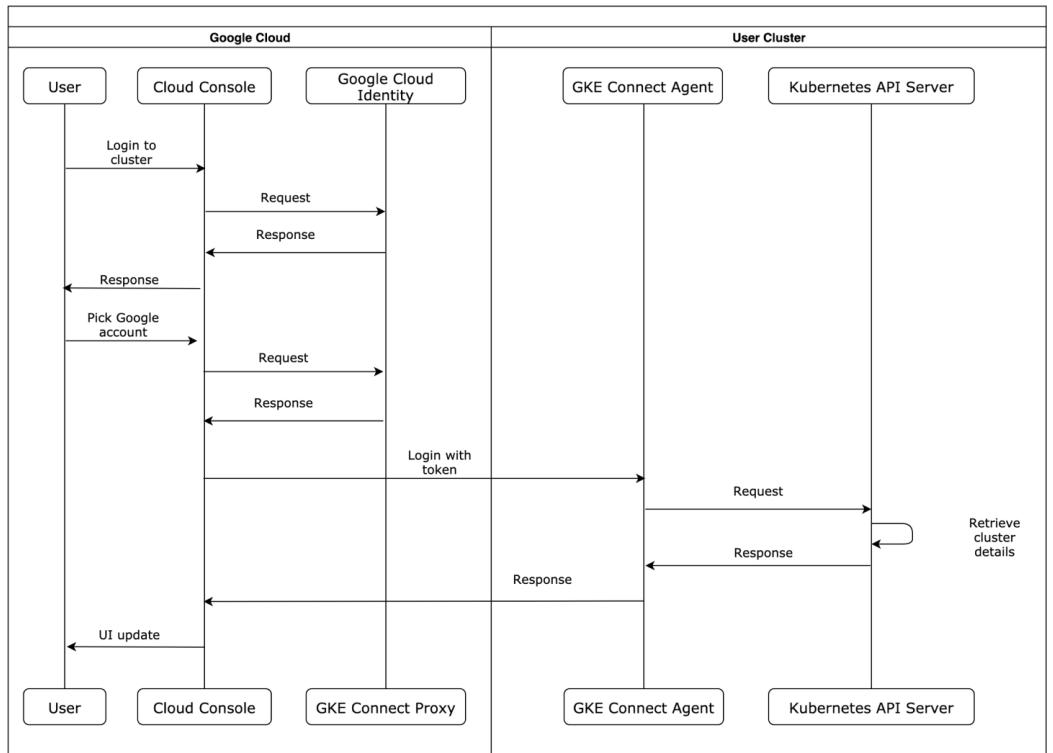


Fig 6.4. Authentication Flow for OIDC with Google Cloud Identity

In the above figure, we show the identity flow using OIDC with Google Cloud Identity. With Anthos Identity Service, other providers which follow the OIDC and Lightweight Directory Access Protocol (LDAP) protocols can provide identity. This allows a seamless user

administration with technologies such as MS Active Directory or an LDAP server, and follows the principle of having one single source of truth of identity.

6.1.3 Cluster Management

After registering the clusters and authenticating, users will be able to see Pods, Services, ConfigMaps and Persistent Volumes which are normally available from GKE native clusters. In this section, the cluster management options available via the Google Cloud Console will be covered. However, to build a good SRE practice, cluster management should be automated and scripted, but it is nice to be able to modify these from a user-friendly interface.

Administrators who have experience in Google Kubernetes Engine on GCP know how easy it is to connect to the cluster from Google Cloud Console. They just navigate to the cluster list as seen in Figure 6.5, and click on the connect button.

Name ^	Location	Cluster size	Total cores	Total memory	Notifications	Labels	Connect
✓ anthos-primary-cluster	europe-north1-a	4	10 vCPUs	37.50 GB	⚠️ Low resource requests	mesh_id:proj-710282601588	

Fig 6.5. Cluster list in GCP Console

Once the Connect button is clicked, a popup window, as shown in Figure 6.6, will provide the command to run in a Google Cloud Shell to connect to the selected cluster.

Connect to the cluster

You can connect to your cluster via command-line or using a dashboard.

Command-line access

Configure `kubectl` command-line access by running the following command:

```
$ gcloud container clusters get-credentials primary-cluster --region europe-west1 --project msp-cc-prod
```

[Run in Cloud Shell](#)

Cloud Console dashboard

You can view the workloads running in your cluster in the Cloud Console [Workloads dashboard](#).

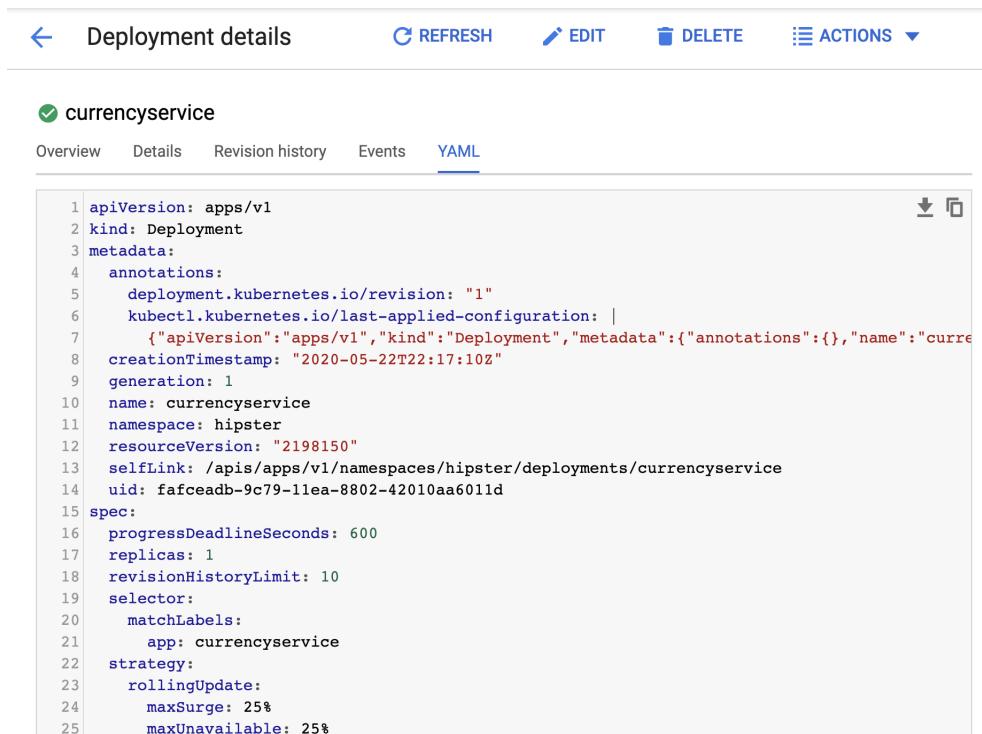
[Open Workloads dashboard](#)

OK

Fig 6.6. One of the best features in GKE, generating `kubectl` credentials via `gcloud`

For On-prem and other cloud clusters the Connect Gateway functionality further in the chapter allows operations administrators to still manage their clusters remotely but through a different command.

Google Cloud Console provides a user-friendly interface to edit and apply yaml deployments as shown in Figure 6.7. Through this interface, administrators are able to modify Kubernetes configurations without having to go through the *kubectl* command line, which can save some time in emergency situations. These actions on Google Cloud Console translate to Kubernetes API calls, or *kubectl* edit commands, and are issued via the Connect agent to the Anthos clusters. Of course this should only be used in triage or dev situations, not necessarily in production, but it shows the future possibilities of opening up access to the Connect Agent from the local command line.



```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   annotations:
5     deployment.kubernetes.io/revision: "1"
6     kubectl.kubernetes.io/last-applied-configuration: |
7       {"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{},"name":"curren
8   creationTimestamp: "2020-05-22T22:17:10Z"
9   generation: 1
10  name: currencyservice
11  namespace: hipster
12  resourceVersion: "2198150"
13  selfLink: /apis/apps/v1/namespaces/hipster/deployments/currencyservice
14  uid: fafceadb-9c79-11ea-8802-42010aa6011d
15 spec:
16   progressDeadlineSeconds: 600
17   replicas: 1
18   revisionHistoryLimit: 10
19   selector:
20     matchLabels:
21       app: currencyservice
22   strategy:
23     rollingUpdate:
24       maxSurge: 25%
25       maxUnavailable: 25%
```

Fig 6.7. Editing a YAML definition from the Google Cloud Console

Google Cloud Console also provides useful information about the underlying Docker, kubelet, and memory pressure for the nodes, as seen from Figure 6.8. This provides administrators a quick root cause analysis if there is a fault with one of the nodes and they are able to cordon and drain the node.

Node details		REFRESH	EDIT	CORDON
Huge pages 1Gi	0 B	0 B	0 B	
Huge pages 2Mi	0 B	0 B	0 B	
Memory	7.84 GB	5.92 GB	774.85 MB	
Pods	110	110	0	
Storage	0 B	0 B	0 B	

Conditions				
Condition ^	Message	Last heartbeat	Last transition	
CorruptDockerOverlay2: False	docker overlay2 is functioning properly	14 Jul 2020, 20:10:39	28 May 2020, 22:36:26	
DiskPressure: False	kubelet has no disk pressure	14 Jul 2020, 20:10:57	28 May 2020, 22:36:22	
FrequentContainerdRestart: False	containerd is functioning properly	14 Jul 2020, 20:10:39	28 May 2020, 22:36:26	
FrequentDockerRestart: False	docker is functioning properly	14 Jul 2020, 20:10:39	28 May 2020, 22:36:26	
FrequentKubeletRestart: False	kubelet is functioning properly	14 Jul 2020, 20:10:39	28 May 2020, 22:36:26	
FrequentUnregisterNetDevice: False	node is functioning properly	14 Jul 2020, 20:10:39	28 May 2020, 22:36:26	
KernelDeadlock: False	kernel has no deadlock	14 Jul 2020, 20:10:39	28 May 2020, 22:36:26	
MemoryPressure: False	kubelet has sufficient memory available	14 Jul 2020, 20:10:57	28 May 2020, 22:36:22	
NetworkUnavailable: False	RouteController created a route	28 May 2020, 22:36:33	28 May 2020, 22:36:33	
PIDPressure: False	kubelet has sufficient PID available	14 Jul 2020, 20:10:57	28 May 2020, 22:36:22	
ReadonlyFilesystem: False	Filesystem is not read-only	14 Jul 2020, 20:10:39	28 May 2020, 22:36:26	
Ready: True	kubelet is posting ready status. AppArmor enabled	14 Jul 2020, 20:10:57	28 May 2020, 22:36:22	

Fig 6.8. Node information from Google Cloud Console

When listing the workloads in Google Cloud Console, a user can see deployments across all clusters and filter them by clusters they are in. This gives the user an overview of what services are running across all clusters and indicates if there are any issues with any services and scaling limits. One of the most common issues being when the Pods cannot be provisioned due to lack of CPU or memory, and this is clear visible as a bright red error message in the console as seen from Figure 6.9.



Fig 6.9. Unschedulable pods example

Viewing a cluster interactively with tools is beneficial for real-time views of object states, node statuses, and more. While this can be helpful in the right scenario (e.g. When diagnosing a previously unknown issue which impacts production where a user-friendly interface reduces the need to remember commands in a high stress situation), you will find

yourself looking at logs and creating monitoring events more often than real-time views. In the next section we will detail the logging and monitoring features that Anthos includes.

6.2 Logging and Monitoring

In Kubernetes, there are different kinds of logs which are useful for administrators to investigate when managing the cluster. These are the system logs which Kubernetes system services such as kube-apiserver, etcd, kube-scheduler and kube-controller-manager log to. Clusters also have application logs, which contain log details for all the workloads running on the Kubernetes cluster. These logs can be accessed through the Connect Agent, which would communicate with the Kubernetes API and essentially issue a `kubectl logs` command.

These logs are not stored in the cloud, but retrieved on demand from the Kubernetes API, which translates to an increased retrieval latency but is at times necessary in case of IT security requests. These are both for application logs and for system logs as seen from Figure 6.10.

Container	Timestamp	Message
istio-proxy	14 Jul 2020, 20:42:23	[2020-07-14T18:38:12.810Z] " - " 0 " " " 1428 3415 240859 - " " " " " " 173.194.222.95:443" outbound[443]/*.googleapis.com 10.10.4.11:38954 173.194.222.95:443 10.10.4.11:38952 cloudtrace.googleapis.com -
istio-proxy	14 Jul 2020, 20:40:08	[Envoy (Epoch 1)] [2020-07-14 18:40:08.150][38][warning][filter] [src/istio/mixerclient/report_batch.cc:110] Mixer Report failed with: UNAVAILABLE:upstream connect error or disconnect/reset before headers. reset reason: connection failure
istio-proxy	14 Jul 2020, 20:29:37	[Envoy (Epoch 1)] [2020-07-14 18:29:37.449][31][warning][config] [bazel-out/k8-opt/bin/external/envoy/source/common/config/_virtual_includes/grpc_stream_lib/common/config/grpc_stream.h:91] gRPC config stream closed: 13,
istio-proxy	14 Jul 2020, 20:13:23	[2020-07-14T18:13:20.731Z] "GET /computeMetadata/v1/instance/service-accounts/default/token HTTP/1.1" 200 - " " 0 210 26 25 " " Google-HTTP-Java-Client/1.24.1 (gzip)" "8ef3aea-0512-43e3-9c79-5148e0802f7c" "169.254.169.254" "169.254.169.254:80" PassthroughCluster - 169.254.169.254:80 10.10.4.11:55300 - "169.254.169.254:80"
istio-proxy	14 Jul 2020, 20:09:07	[Envoy (Epoch 1)] [2020-07-14 18:09:07.886][38][warning][filter] [src/istio/mixerclient/report_batch.cc:110] Mixer Report failed with: UNAVAILABLE:upstream connect error or disconnect/reset before headers. reset reason: connection failure
istio-proxy	14 Jul 2020, 20:06:03	[2020-07-14T17:34:10.309Z] " - " 0 " " " 948462 7830 1904930 - " " " " " " 173.194.73.95:443" outbound[443]/*.googleapis.com 10.10.4.11:49044 173.194.73.95:443 10.10.4.11:49042 monitoring.googleapis.com -

Fig 6.10. Container logs

Logs are primarily about errors, warnings that output to the standard output stream during the execution of any kubernetes pod. These logs are written to the node itself and if the Google Cloud's operations suite (formerly Stackdriver) agent is enabled on the GKE cluster, the logs are aggregated and forwarded to the Cloud Logging API and written into the cloud.

Metrics are observations about a service, such as memory consumption, requests per second. These observations are saved in a historical trend which can be used to scale services, or to identify possible issues in implementation. Given that each service can potentially have tens of observations occurring every second or minute depending on the

business requirements, managing this data in a usable manner is non-trivial and a few solutions have been proposed in the following paragraph involving Google's Cloud Logging and Monitoring service. Partner technology such as Elastic Stack, Prometheus, Grafana or Splunk can also be used to make sense of the metrics (e.g. <https://cloud.google.com/architecture/logging-anthos-with-splunk-connect> and <https://cloud.google.com/architecture/partners/monitoring-gke-on-prem-with-the-elasticsearch>)

6.2.1 Logging and Monitoring GKE on-prem

Administrators are able to choose between a few different options for observability when installing GKE on-prem clusters.

The first option is to use Google's native Cloud Logging and Cloud Monitoring solutions. Cloud Logging and Monitoring handles infrastructure and cloud services, as well as Kubernetes Logging and Monitoring. All logged data can be displayed in hierarchical levels according to the Kubernetes object types. By default, GKE logging only collects logs and metrics from the kube-system, gke-system, gke-connect, istio-system, and config-management system namespaces, which are used to track cluster health and are sent to Cloud Logging and Monitoring in Google Cloud. This is a fully managed service and includes dashboarding and alerting capabilities to be able to build a useful monitoring control panel. Cloud Logging and Monitoring is often used to monitor Google Cloud resources and issue alerts on certain logged events and it also serves as a single pane of glass for monitoring service health.

This is the recommended option in the event the organization is open to using and learning a new logging and monitoring stack, and wants a low cost and fully managed option.

There may be certain organizations that want to disable Cloud Logging and Monitoring due to internal decisions. While this can be disabled, the Google Support SLA will be voided and Google support will only be able to help as a best-effort when resolving GKE On-prem operation issues.

The second option is to use Prometheus, AlertManager and Grafana, a popular open source collection of projects, used to collect application and system level logs, provide alerting and dashboarding capabilities. Prometheus and Grafana are deployed as Kubernetes Monitoring Add-on workloads, and as such, will benefit from the scalability and reliability of running on Kubernetes. Support from Google is limited when using this solution, limited to basic operations and basic installation/configuration. For more information on

Prometheus and AlertManager please visit <https://prometheus.io>, and for Grafana please visit <https://grafana.com>

This setup can be used across any Kubernetes setup, and there are also many grafana dashboards prebuilt which can be used to monitor kubernetes cluster health. One downside is that administrators will now have to manage Prometheus and ensure its health and manage its storage of historical metrics, as it is running as any other application workload. Other tools such as Thanos can be used to query, aggregate, downsample and manage multiple Prometheus sources, as well as store historical metrics in object storage such as Google Cloud Storage or any S3 compatible object stores.

For more information on Thanos please visit <https://thanos.io/>

This option is easy for organizations which have built logging and monitoring using open source technologies and have deployed this stack before. This also improves portability and reduces vendor lock-in due to the open source technologies used.

The third option is to use validated solutions such as Elastic Stack, Splunk or Datadog, to consume logs and metrics from Anthos clusters and make them available to the operations team.

This is an attractive option if these current logging methods are already in place, and rely on partners to manage the logging and monitoring systems uptime. Organizations who choose this option often already purchased this stack and are used for their overall organizations with many heterogeneous systems.

A fourth option is also a tiered telemetry approach, which is recommended for organizations embarking on a hybrid journey with Anthos.

The first reason is that platform and system data from Anthos Clusters are always tightly coupled with Cloud Monitoring and Logging so administrators would have to learn Cloud Monitoring and Logging to get the most up-to-date logs and metrics anyways as well as it does not have any additional costs and is part of the Anthos suite.

The second reason is that building a hybrid environment often requires migrating applications to the hybrid environment, with developers who are used to working with these partner solutions and have built debugging and operating models around that stack, making it a supported option that reduces the operational friction to moving workloads to a hybrid environment.

The third reason is to build the ability to balance points of failure among different providers and having a backup option.

6.3 Service Mesh Logging

Anthos Service Mesh is an optional component but included in the Anthos platform which is explained in depth in Chapter 5. It is an extended and supported version of open source Istio, included with Anthos and supported by Google. Part of what Google extended is to upload telemetry data from sidecar proxies injected with your pods directly to Cloud Monitoring API and Cloud Logging API on Google Cloud. These metrics are then used to visualize preconfigured dashboards in the Google Cloud Console. For more details, please refer to Chapter 3, Anthos, the one single glass-of-pane UX.

Storing these metrics on Google Cloud also allows the user to have historical information on latency, errors and traffic between their microservices, so that post-mortem can be conducted on any issues.

These metrics can be further used to drive your Service Level Indicators, pod scaling strategy and identify services for optimization.

6.4 Using Service Level Indicators and Agreements

Anthos Service Mesh Service Level Indicator (SLI), Service Level Objectives (SLO) and Service Level Agreement (SLA) are features which can be used to build an SRE practice where Anthos is deployed. These concepts have been introduced in Chapter 5, and it is

necessary to consider these concepts when designing operations management procedures in Anthos.

With Service Level Indicators, there are two indicators which are available to measure service levels: latency and availability. Latency is how long the service takes to respond, while availability is how often the service responds. When this is designed from a DevOps view, administrators have to take into consideration Anthos upgrade and scaling needs and plan accordingly so it does not affect these indicators.

For Service Level Objectives, think from the angle of the worst case scenario, and not the best case scenario, making that decision as data driven as possible. For example, if the latency is unrealistic and it does not have an impact on the user experience, there will be no way to even release the service. Find the highest latency which is acceptable according to the user experience and then work on reducing that based on business needs. Educate your business stakeholders that 100% availability is very expensive to attain and that a practical tradeoff often has to be agreed. This is an important concept mentioned as well in the SRE book by Google, to strive to make a service reliable enough but no more than it has to be. You can find more information on the SRE book by Google at <https://landing.google.com/sre/sre-book/chapters/embracing-risk/>

Understanding the procedures and risks of Anthos upgrades, rollbacks, security updates is essential input to determining if a Service Level Objective is realistic or not.

A compliance period should also be defined for the Service Level Objective to be measured on. The set SLO can be any period of measurement- a day, week or a month. This allows for the teams responsible for the service to decide when it is time to rollback, make a hotfix or slow down development to prioritize fixing bugs.

The SLI and SLO also empowers product owners to propose Service Level Agreements with users which require it, and offer a realistic latency and availability agreement.

6.5 Anthos Command Line Management

These are various command line tools that deal with cluster creation, scaling and upgrading of Anthos versions, such as `gkectl`, `gkeadmin` and `anthos-gke`. This chapter is not meant to replace the documentation on Google Cloud, but summarizes the actions and some of the gotchas to look out for.

Reminder: Admin clusters are deployed purely to monitor and administer user clusters, think of them as the invisible control plane analogous to GKE and do not deploy services which can affect it there.

TIP: You can use a kubeconfig manager like ktx from <https://github.com/heptiolabs/ktx> which allows administrators to switch between admin and user cluster contexts easily.

In the next section, the segments will be broken up into GKE on-prem and GKE on AWS as the tools and installation process differ.

6.5.1 Using CLI Tools for GKE on-prem

GKE on-prem installation uses the APIs from VMWare¹ to build an admin workstation, admin cluster nodes and user cluster nodes programmatically. Persistent volumes are powered from individual VMWare's Datastore or vSAN, and networking is provided by either distributed or standard vSphere switches. These act like the IaaS components provided by Google Cloud when building a GKE cluster, thus the name GKE on-prem. The concept of having an Admin Cluster with User Clusters and Node Pools mirror with GKE best practices.

The current installation process is to download a tool named `gkeadm`, which creates an admin workstation. It is from this admin workstation the admin cluster and user clusters are installed from. While there are versions of `gkeadm` available for Windows, Linux and Mac OS, this section will only explain an abbreviated process for Linux.

1. The first step is to download the tool from the cloud storage bucket

```
gsutil cp gs://gke-on-prem-release-public/gkeadm/<anthos version>/linux/gkeadm  
./chmod +x gkeadm
```

2. Next, create a pre-populated config file

```
./gkeadm create config
```

Fill in the vCenter credentials, GCP white listed service account key path (After purchasing Anthos, customers will be asked to provide a service account which Google will whitelist to be able to download images and other proprietary tools), vCenter Certificate Authority certs path.

```
The vCenter Certificate Authority certs can be downloaded by  
curl -k "https://[SERVER_ADDRESS]/certs/download.zip" > download.zip
```

After unzipping the `download.zip` file, the relevant certs can be found in the `certs/lin` folder. The file with `.0` suffix is the root certificate. Rename it to `vcenter.crt` and use it in the reference from the installation config file.

The vCenter and BigIP F5 credentials are saved in plain text in the config file when creating new user clusters or on installation. One way to secure the F5 credentials is through using a wrapper around Google Cloud Secrets Manager and `gcloud`.

To create a password secured by Google Secret Manager

```
echo "vcenterp45w0rd" | gcloud secrets create vcenterpass --data-file=- --replication-  
policy=user-managed --locations=us-east1
```

To retrieve a password secured by Google Secret Manager

```
gcloud secrets versions access latest --secret="vcenterpass"
```

¹In addition to VMWare it is possible to use Anthos on Bare Metal. That is the topic discussed in Chapter 22.

This secret is now protected via Google IAM policies and a wrapper script can be written to retrieve the secret, replace the placeholder in the config file, apply and then delete the file.

The process to create Anthos cluster components is quickly evolving, and it's not uncommon for a newer version to have some changes to the config file. You can follow this link for latest release procedures at <https://cloud.google.com/anthos/clusters/docs/on-prem/1.9/how-to/create-admin-workstation>

6.5.2 Cluster Management: Creating a new user Cluster

The `gkectl` command is used for this operation.

As a rule of thumb, admins should constrain the setups so that there is a ratio of one admin cluster to ten user clusters. User clusters should have a minimum of 3 nodes, with a maximum of 100 nodes. As previously mentioned, newer releases may increase these numbers. When a new Anthos release is published, you can check the new limits in the *Quotas and Limits* section of the respective release.

The general advice is leave some space for at least one cluster which can be created in your on-prem environment. This would give the operations team space to recreate clusters and move pods over when upgrading or triage.

Keep good documentation like which IP addresses have been already assigned for other user clusters, so that non-overlapping IPs can be determined easily. Take into consideration that user clusters can be resized to 100 nodes, so reserve up to 100 IP addresses per range to keep that possibility.

Source control your configuration files, but do not commit the vsphere username / passwords. Committing such sensitive information into repositories will open up security risks as anyone with access to the repository will be able to get those login details. Tools like `ytt` can be used to template configuration yaml, code reviews and repository scanners should be used to prevent such mistakes from taking place (e.g. <https://github.com/UKHomeOffice/repo-security-scanner>).

Node pools can also be created with different machine shapes, so size them correctly to accommodate your workloads. This also gives granular control over which machine types to scale and save costs. For production workloads, use three replicas for the user cluster master nodes for high availability, but for dev, one should be fine.

Validate the configuration file to make sure the file is valid. The checks are both syntactic and programmatic, such as checking for IP range clashes and IP availability using the `gkectl check-config` command.

```
gkectl check-config --kubeconfig [ADMIN_CLUSTER_KUBECONFIG] --config [CONFIG_FILE]
```

After the first few validations, most time-consuming validations can be skipped by passing the `--fast` flag.

Next, the seesaw load balancer should be created if the bundled load balancer is chosen. If you do not create the Seesaw node(s) before attempting a cluster build that has been configured with the integrated load-balancer option, you will receive an error during the

cluster pre-check. To create the Seesaw node(s) use the `gkectl create loadbalancer` command.

```
gkectl create loadbalancer --kubeconfig [ADMIN_CLUSTER_KUBECONFIG] --config [CONFIG_FILE]
```

After creation of a new user cluster do remember that for the bundled lb seesaw version, the user will then be able to create the user cluster.

```
gkectl create cluster --kubeconfig [ADMIN_CLUSTER_KUBECONFIG] --config [CONFIG_FILE]
```

You can also add the `--skip-validation-all` flag if the config file has already been validated.

The whole user cluster process can take 20 - 30 minutes depending on the hardware, which consists of starting up new VMWare virtual machines with the master and worker node images and joining them into a cluster. The administrator is also able to see the nodes being created from the VMWare vCenter console.

High availability setup

High availability is necessary for Anthos deployments in production environments. This is important as failures can occur at different parts of the stack, ranging from networking, to hardware, to the virtualization layer.

High availability for admin clusters makes use of the vSphere High Availability in a vSphere Cluster setup to protect GKE on-prem clusters from going down in the event of a host failure. This ensures that admin cluster nodes are distributed among different physical nodes in a vSphere cluster, so that in the event of a physical node failure, the admin cluster will still be available.

To enable HA user control planes, simply specify `usercluster.master.replicas: 3` in the GKE on-prem configuration file. This will create three user cluster masters for each user cluster, consuming three times the resources, but providing a high availability kubernetes setup.

6.5.3 Cluster Management: Scaling

Administrators are able to use the `gkectl` cli to scale up or down nodes as seen below. They would change the config file to set the number of expected replicas and execute the following command to update the node pool.

```
gkectl update cluster --kubeconfig [USER_CLUSTER_KUBECONFIG] --config [CONFIG_FILE]
```

6.5.4 Cluster Management: Upgrading Anthos

Like any upgrade process, there can be failures during the process. A lot of effort has been put into making the upgrade process robust, including the addition of pre-checks before executing the upgrade to catch potential issues before they occur. Each product team at

Google works closely when an upgrade is being developed to avoid any potential incompatibilities between components like Kubernetes, ACM, and ASM.

Anthos versions are quite fast moving due to industry demand for new features and this means that upgrading Anthos is a very common activity. Upgrading Anthos can also mean upgrading to a new version of Kubernetes, which also impacts Anthos Service Mesh due to Istio dependency on Kubernetes. This means the upgrade chain is complex, which is why the recommendation in the beginning is to keep some spare hardware resources that can be used to create new versions of Anthos clusters and then move workloads to the new cluster before tearing down the older version cluster. This process reduces the risk associated with upgrades by providing an easy rollback path in case of a failed upgrade. In this type of upgrade path, there should be a load-balancer in front of the microservices running in the old cluster to be upgraded which can direct traffic from the old cluster to the new cluster, as they will exist at the same time.

However if this is not an option, administrators are able to upgrade Anthos clusters in place.

Firstly, consult the upgrade paths. From GKE on-prem 1.3.2 onwards, administrators are able to upgrade directly to any version in the same minor release, otherwise sequential upgrades are required. From version 1.7, administrators can keep their admin cluster on an older version, while only upgrading the admin workstation and the user cluster. As a best practice, administrators should still schedule the admin cluster upgrades to keep up to date.

Next, download the gkeadm tool which has to be the same as the target version of your upgrade, and run gkeadm to upgrade the admin workstation and gkectl to upgrade your user cluster, and finally the admin cluster.

When upgrading in place, a new node is created with the image of the latest version and workloads are drained from the older version and shifted over to the latest version, one node after the other. This means administrators should plan for additional resources in their physical hosts to accommodate at least one user node for upgrade purposes. The full flow can be seen from Figure 6.11.

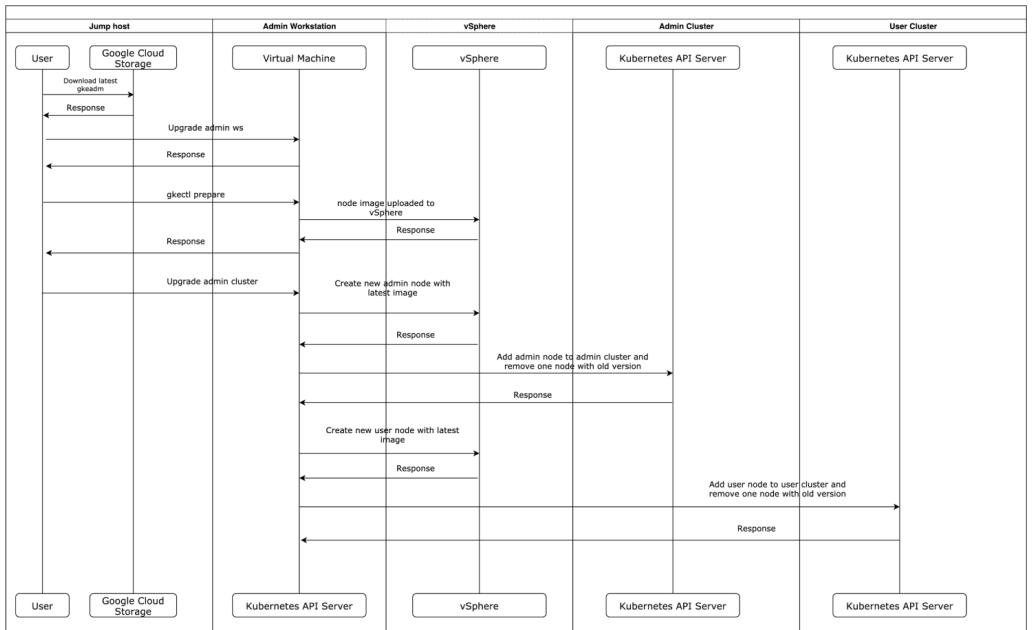


Fig 6.11. Upgrading flow

For a detailed list of commands please consult the upgrading documentation at https://cloud.google.com/anthos/gke/docs/on-prem/how-to/upgrading#upgrading_your_admin_workstation for exact details.

6.5.5 Cluster Management: Backing up Clusters

Anthos admin clusters can be backed up by following the steps found at <https://cloud.google.com/anthos/clusters/docs/on-prem/1.8/how-to/back-up-and-restore-an-admin-cluster-with-gkecli>

This is recommended to be done as part of a production Anthos environment setup to regularly schedule backups and to do on demand backups when upgrading Anthos versions.

Anthos user clusters etcd can be backed up by running a backup script which you can read more about backing up a cluster on the Anthos documentation page at <https://cloud.google.com/anthos/gke/docs/on-prem/how-to/backing-up>

Do note that this only backs up the etcd of the clusters, which means the Kubernetes configuration. Google also states this to be a last resort. Backup for GKE promises to make this simpler, and we look forward to similar functionality for Anthos clusters soon. (<https://cloud.google.com/blog/products/storage-data-transfer/google-cloud-launches-backups-for-gke>).

Any application specific data, such as persistent volumes are not backed up by this process. Those should be backed up regularly to another storage device using a number of different tools like Velero.

You should treat your cluster backups the same as any data that is backed up from a server. The recommendation is to practice restoring an admin and user cluster from backup, along with application specific data to gain confidence in the backup and recovery process.

Targeted in the roadmap for H2 2021, Google will release a feature named Anthos Enterprise Data Protection functionality to backup cluster wide config such as custom resource definitions, and namespace wide configuration and application data from Google Cloud Console into a cloud storage bucket, and be able to restore using the backup as well.

6.6 GKE on AWS

GKE on AWS uses AWS EC2 instances and other components to build GKE clusters, which means these are not EKS clusters. If a user logs into the AWS console, they will be able to see the admin cluster and user clusters nodes only as individual AWS EC2 instances. It is important to differentiate this between managing EKS clusters from Anthos as the responsibilities assigned to the different cloud providers according to each cluster type is different.

GKE on AWS installation is done via the *anthos-gke* cli, which can be run on Linux or MacOS as of the time of writing. The next release of GKE on AWS set for Q4 2021 will use the gcloud cli and follow the same architecture as GKE on Azure below with a single multi-cloud API for managing cluster life cycle operations. This will further simplify the installation process and remove the need for a bastion host and management server mentioned in the steps below.

The installation process is to first get an AWS KMS key, then use *anthos-gke* which in turn uses Terraform to generate Terraform code. Terraform is an Infrastructure as Code open source tool to define a target state of a computing environment by Hashicorp. Terraform code is declarative and utilizes Terraform providers which are often contributed by cloud providers such as Google, AWS, Microsoft to map their cloud provisioning APIs to Terraform code. The resulting Terraform code describes how the GKE on AWS infrastructure will look like. It has components which are analogous to GKE on-prem, such as a LoadBalancer, EC2 Virtual Machines, but leverage the Terraform AWS Provider work to instantiate the infrastructure on AWS.

You can learn more about Terraform at <https://www.terraform.io/>

The architecture of GKE on AWS can be seen on Figure 6.12 below which is from the Google Cloud documentation at <https://cloud.google.com/anthos/gke/docs/aws/concepts/architecture>

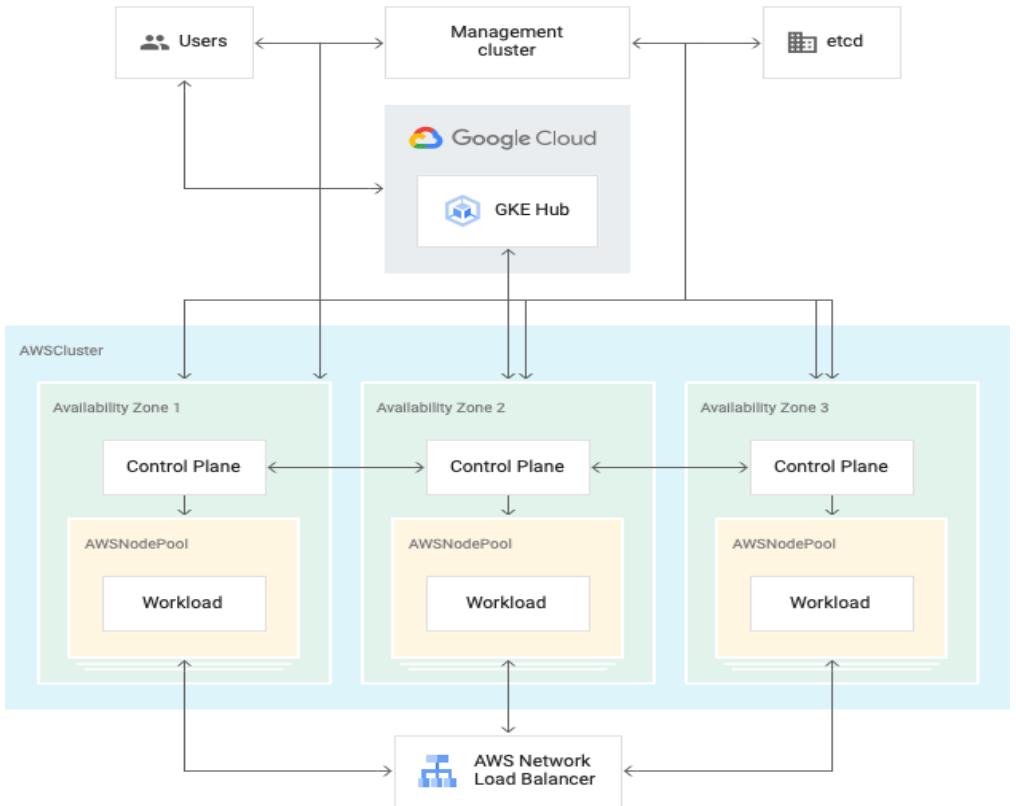


Fig 6.12. GKE on AWS architecture

The Use of node pools is similar to GKE, with the ability in a cluster to have different machine sizes.

NOTE: To do any GKE on AWS operations management, the administrator will have to log into the bastion host which is part of the management service.

6.6.1 Connecting to the management service

When doing any management operations, the administrator needs to connect to the bastion host deployed during the initial installation of the management service. This script is named `bastion-tunnel.sh` which is generated from Terraform during the management service installation.

6.6.2 Cluster Management: Creating a new user cluster

Use the `bastion-tunnel` script to connect to the management service.

After connecting to the bastion host, the administrator will use Terraform to generate a manifest configuring an example cluster in a yaml file

```
terraform output cluster_example > cluster-0.yaml
```

In this yaml file the administrator will then change the *AWSCluster* and *AWSNodePool* specifications. Be sure to save the cluster file to a code repository, this will be reused for scaling the user cluster.

Custom Resources are extensions of Kubernetes to add additional functionality, such as in the case of provisioning AWS EC2 instances. AWS Clusters and objects are represented as yaml referencing the *AWSCluster* and *AWSNodePool* Custom Resources in the management service cluster, which interpret this yaml and adjust resources in AWS accordingly.

To read more about Custom Resources please refer to <https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/>

6.6.3 Cluster Management: Scaling

You may experience a situation where a cluster requires additional compute power and you need to scale the cluster out. Luckily, an Anthos node pool has an option to scale a cluster, including a minimum and maximum node count. If you created a cluster with the same count in both the minimum and maximum nodes, you can change it at a later date to grow your cluster. To scale a cluster for GKE on AWS, you simply require the administrator to modify the YAML file by updating the `minNodeCount` while creating the user cluster, and applying it to the management service.

```
apiVersion: multicloud.cluster.gke.io/v1
kind: AWSNodePool
metadata:
  name: cluster-0-pool-0
spec:
  clusterName: cluster-0
  version: 1.20.10-gke.600
  minNodeCount: 3
  maxNodeCount: 10
```

6.6.4 Cluster Management: Upgrading

Upgrading GKE on AWS is done in two steps, with the management service handled first, and then the user clusters.

To upgrade a GKE on AWS management service, the administrator has to do this from the directory with the GKE on AWS configuration.

The user has to first download the latest version of the `anthos-gke` binary. Next the user will have to modify the `anthos-gke.yaml` file to the target version.

```
apiVersion: multicloud.cluster.gke.io/v1
kind: AWSManagementService
metadata:
```

```
name: management
spec:
version: <target_version>
```

Finally to validate and apply the version changes by running

```
anthos-gke aws management init
anthos-gke aws management apply
```

The management service will be down, so no changes to user clusters can be applied, but user clusters continue to run their workloads.

To upgrade the user cluster, the administrator switches context in the management service from the GKE on AWS directory using the following command.

```
anthos-gke aws management get-credentials
```

Then upgrading the version of the user cluster is as simple as using

```
kubectl edit awscluster <cluster_name>
```

And then editing the yaml to point to the right GKE version.

```
apiVersion: multicloud.cluster.gke.io/v1
kind: AWSCluster
metadata:
  name: cluster-0
spec:
  region: us-east-1
  controlPlane:
    version: <gke_version>
```

On submission of this change, the CRD starts to go through the nodes in the control plane one by one and start upgrading them to the latest GKE on AWS version. This upgrade process causes a downtime of the control plane, which means the cluster may be unable to report the status of the different node pools until it is completed.

Finally, the last step is to upgrade the actual Node Pool. The same procedure applies and the administrator simply edits the yaml to the version required and applies the yaml to the management service.

```
apiVersion: multicloud.cluster.gke.io/v1
kind: AWSNodePool
metadata:
  name: cluster-0-pool-0
spec:
  clusterName: cluster-0
  region: us-east-1
  version: <gke-version>
```

6.7 Anthos attached clusters

Anthos attached clusters are managed conformant Kubernetes which are provisioned and managed by Elastic Kubernetes Service (EKS) by AWS, Azure Kubernetes Service (AKS) or any conformant Kubernetes cluster. In this case, the scaling and provisioning of the clusters are done from the respective clouds. However these clusters can still be attached and managed by Anthos by registering them to Google Cloud through deployment of the Connect agent as seen from Figure 6.13. Google Kubernetes Engine is also handled in the same way and can be attached from another project into the Anthos project.



Fig 6.13. Adding an external cluster (Bring your own Kubernetes)

1. The administrator has to generate a kubeconfig to the EKS or AKS cluster, and then provide that kubeconfig in a generated cluster registration command in gcloud. Please consult documentation from AWS and Azure on how to generate a kubeconfig file for the EKS or AKS clusters. The administrator is also able to generate one manually using the template below and providing the necessary certificate, server info and service account token.

```
apiVersion: v1
kind: Config
users:
- name: svcs-acct-dply
  user:
    token: <replace this with token info>
clusters:
- cluster:
    certificate-authority-data: <replace this with certificate-authority-data info>
    server: <replace this with server info>
    name: self-hosted-cluster
contexts:
- context:
    cluster: self-hosted-cluster
    user: svcs-acct-dply
    name: svcs-acct-context
current-context: svcs-acct-context
```

2. The administrator has to create a Google service account and a service account key to provide for the registration as seen in Figure 6.14.

[←](#) Register a Kubernetes cluster

Install [Connect](#) into your cluster and register your cluster to Google Cloud Platform. GKE Connect works behind firewalls and can traverse NAT's to establish an encrypted connection to Google Cloud Platform. [Learn more.](#)

Cluster name	eks-cluster
--------------	-------------

Configure GCP labels for the cluster.

Key *	Value
location	stockholm
provider	aws
+ ADD LABEL	
CHANGE CLUSTER NAME	CANCEL

To register the cluster, run the following command:

```
$ gcloud container hub memberships register eks-cluster \  
  --context=[CLUSTER_CONTEXT] \  
  --service-account-key-file=[LOCAL_KEY_PATH] \  
  --kubeconfig=[KUBECONFIG_PATH] \  
  --project=anthos-sandbox-256114
```

 Waiting for eks-cluster GKE Connect connection with Google

 Waiting for eks-cluster Membership to be created

Fig 6.14. Generating registration command to the external cluster

3. The administrator will provide these two into the generated registration command, and after the connect agent has been deployed into the external cluster, it will be visible in the Google Cloud Console.

6.8 Anthos on Bare Metal

Operating and managing Anthos on Bare Metal often requires additional skill sets in the OS configuration space as it is based on installing Anthos on RHEL, Ubuntu or CentOS. For the

detailed steps in installation and upgrading of Anthos on Bare Metal, please consult the chapter Anthos Bare Metal.

Anthos on Bare Metal is somehow similar to Anthos on VMWare, but with more flexibility in its deployment models and no dependency on VMWare.

A few key decisions must be reached when designing the operations management procedures for Anthos on Bare Metal.

First is capacity planning and resource estimation for running Anthos on bare metal. Unlike the rest of the set ups, where new nodes need to be provisioned using either public cloud resources or a pool of VMWare resources, new bare metal nodes have to be provisioned. This requires additional capacity requirements if there is a zero-downtime requirement during upgrades of the nodes as there is always a risk of nodes failing upgrades and causing a decrease in capacity.

Second is automating as much as possible the prerequisite installation of the nodes. Many companies also require a golden image of a base operating system which has to be vetted by a security team and continuously updated with security patches and latest versions. This should be built into the Anthos on Bare Metal provisioning process to be able to verify compatibility with Anthos installation. One option is to set up PXE boot servers and have newly provisioned bare metal servers point to the PXE boot servers to install the operating system of bare metal nodes to the right configuration.

Third is determining the different deployments to run Anthos on Bare Metal, in standalone, multi-cluster or hybrid cluster deployments. Flexibility also means complexity and having to build different operational models for the different deployments. The Anthos Bare Metal goes more into detail about the differences but this chapter highlights the different operational considerations when choosing the different deployment models.

1. Standalone cluster deployment

This deployment model has the admin and user clusters in the same cluster. In such a configuration workloads run in the same nodes which have ssh credentials and Google service account keys are stored. This configuration is well suited for edge deployment and as such operational models should introduce ssh credential and service account key generation for each new standalone cluster provisioned and deployed, and a plan to decommission those credentials when a cluster is compromised or lost. There is a minimum requirement of five nodes for a high availability setup.

2. Multi-cluster deployment

This deployment model has an admin cluster and one or more user clusters, similar to Anthos on VMWare. This has many benefits such as admin - user isolation for the clusters, multi-tenanted setups (e.g. each team can have their own cluster) and a centralized plan for upgrades. The downside is the increased footprint in node requirements, and a minimum of eight nodes for a high availability setup. This would take more effort when setting up for multiple edge locations, and is more for a datacenter setup.

3. Hybrid cluster deployment

This deployment model allows for running of user workloads on the admin clusters, and managing other user clusters. This reduces the footprint required for multi-cluster deployment to five nodes for a high availability setup, but has the same security concern of running user workloads on nodes which may contain sensitive data from the stand-alone cluster deployment. This gives the flexibility to tier workloads by security levels, and introduce user clusters for workloads which require higher security.

6.9 Connect Gateway

Registering Anthos clusters allows the user to interact with them through the UI, but administrators often have a toolbox of scripts which they use to work with clusters through the *kubectl* command line. With GKE On-prem / on AWS / on Azure, these clusters often can only be accessed via either the admin workstation or a bastion host. GKE users on the other hand are able to use gcloud and generate kubeconfig details to *kubectl* to their clusters on their local machines. With Connect Gateway, this problem is solved.

Administrators are also able to connect to any Anthos registered cluster and generate kubeconfig that enables the user to use *kubectl* to those clusters via the Connect agent.

With this feature, administrators will not be required to utilize jump hosts to deploy to the GKE on X clusters, but instead run a gcloud command to generate a kubeconfig to connect via *kubectl*.

The setup requires an impersonation policy which allows the Connect agent service account to impersonate an user account to issue commands on their behalf. An example of the yaml which creates the ClusterRole and ClusterRoleBinding for impersonation can be seen below.

```
# [USER_ACCOUNT] is an email, either USER_EMAIL_ADDRESS or GCPSA_EMAIL_ADDRESS
$ USER_ACCOUNT=foo@example.com
$ cat <<EOF > /tmp/impersonate.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: gateway-impersonate
rules:
- apiGroups:
  - ""
    resourceNames:
    - ${USER_ACCOUNT}
    resources:
    - users
    verbs:
    - impersonate
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: gateway-impersonate
roleRef:
  kind: ClusterRole
  name: gateway-impersonate
```

```
apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: connect-agent-sa
  namespace: gke-connect
EOF
```

After the impersonation policy has been set up, the administrator has to run the command in Figure 6.15 below to generate a kubeconfig seen in Figure 6.16.

```
gcloud alpha container hub memberships get-credentials stockholm-gke-cluster
```

Fig 6.15. Command to get credentials to GKE On-prem cluster

```
- cluster:
  server: https://connectgateway.googleapis.com/v1alpha1/projects/74668819743/memberships/stockholm-gke-cluster
  name: connectgateway_anthos-sandbox-256114_stockholm-gke-cluster
```

Fig 6.16. kubeconfig generated via gcloud

With this kubeconfig in place, administrators are able to manage GKE on-prem workloads even from their local machine, while being secured by their Google Cloud Identity. This also opens up the possibility for building pipelines to deploy to the different Anthos clusters.

6.10 Anthos on Azure

Anthos GKE clusters can be installed on Azure with an architecture consisting of a Multi-Cloud API hosted on GCP that provides life cycle management capabilities to GKE Clusters in Azure as seen in Fig 17. Azure GKE Clusters are also accessed via the Connect Gateway mentioned in this chapter. Anthos on Azure uses Azure native technologies like the Azure Load Balancer, Azure AD and Azure Virtual Machines, but relies on Anthos via the Multi-Cloud API to manage GKE cluster life cycle operations. This creates an uniform way to deploy applications across the three major public clouds and on premise.

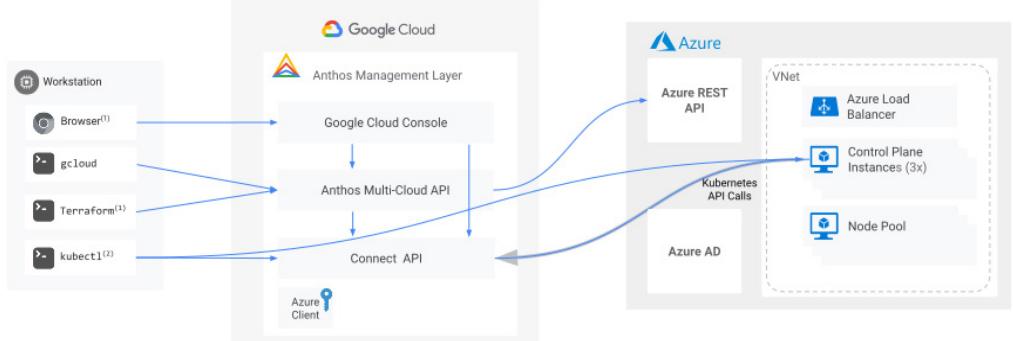


Fig 6.17. Anthos on Azure architecture

As a prerequisite, the administrator has to install the gcloud CLI.

The administrator has to have the following Azure built-in roles seen below.

Azure Built In Roles

1. Application Administrator
2. User Access Administrator
3. Contributor

The next steps would be to create an Azure Active Directory application, a virtual network, a resource group for the clusters, and grant the necessary permissions to the Azure Active Directory application. Detailed prerequisite information can be found in the [public documentation](#).

6.10.1 Cluster management: Creation

To create a new user cluster, the administrator will have to first set up an azure client with ssh key pair

```

export SUBSCRIPTION_ID=$(az account show --query "id" --output tsv)
export TENANT_ID=$(az account list \
    --query "[?id=='${SUBSCRIPTION_ID}'].{tenantId:tenantId}" --output tsv)
export APPLICATION_ID=$(az ad app list --all \
    --query "[?displayName=='APPLICATION_NAME'].appId" --output tsv)

gcloud alpha container azure clients create CLIENT_NAME \
    --location=GOOGLE_CLOUD_LOCATION \
    --tenant-id="${TENANT_ID}" \
    --application-id="${APPLICATION_ID}"

CERT=$(gcloud alpha container azure clients get-public-cert --
    location=GOOGLE_CLOUD_LOCATION \

```

```

CLIENT_NAME)

az ad app credential reset --id "${APPLICATION_ID}" --cert "${CERT}" --append
ssh-keygen -m PEM -t rsa -b 4096 -f KEY_PATH
SSH_PUBLIC_KEY=$(cat KEY_PATH.pub)
ssh-keygen -m PEM -t rsa -b 4096 -f ~/.ssh/anthos-multicloud-key
SSH_PUBLIC_KEY=$(cat ~/.ssh/anthos-multicloud-key.pub)

```

Next the administrator will need to assign Azure resource groups, VNet and subnet IDs to environment variables, add IAM permissions and run the gcloud command to create the Anthos on Azure cluster.

```

CLUSTER_RG_ID=$(az group show --resource-group=CLUSTER_RESOURCE_GROUP_NAME \
--query "id" -otsv)
VNET_ID=$(az network vnet show --resource-group=VNET_RESOURCE_GROUP_NAME \
--name=VNET_NAME --query "id" -otsv)
SUBNET_ID=$(az network vnet subnet show \
--resource-group=VNET_RESOURCE_GROUP_NAME --vnet-name=VNET_NAME \
--name default --query "id" -otsv)

PROJECT_ID="$(gcloud config get-value project)"
gcloud projects add-iam-policy-binding "$PROJECT_ID" \
--member="serviceAccount:$PROJECT_ID.svc.id.goog[gke-system/gke-m multicloud-agent]" \
--role="roles/gkehub.connect"

gcloud alpha container azure clusters create CLUSTER_NAME \
--location GOOGLE_CLOUD_LOCATION \
--client CLIENT_NAME \
--azure-region AZURE_REGION \
--pod-address-cidr-blocks POD_CIDR \
--service-address-cidr-blocks SERVICE_CIDR \
--vm-size VM_SIZE \
--cluster-version 1.19.10-gke.1000 \
--ssh-public-key "$SSH_PUBLIC_KEY" \
--resource-group-id "$CLUSTER_RG_ID" \
--vnet-id "$VNET_ID" \
--subnet-id "$SUBNET_ID"

```

This cluster should then be available on the administrator's GKE console.

Finally add a node pool to be able to deploy workloads to the cluster.

```

SUBNET_ID=$(az network vnet subnet show \
    --resource-group=VNET_RESOURCE_GROUP_NAME --vnet-name=VNET_NAME \
    --name default --query "id" -otsv)
SSH_PUBLIC_KEY=$(cat KEY_PATH.pub)

gcloud alpha container azure node-pools create NODE_POOL_NAME \
    --cluster=CLUSTER_NAME \
    --location GOOGLE_CLOUD_LOCATION \
    --node-version=1.19.10-gke.1000 \
    --vm-size=VM_SIZE \
    --max-pods-per-node=110 \
    --min-nodes=MIN_NODES \
    --max-nodes=MAX_NODES \
    --ssh-public-key="${SSH_PUBLIC_KEY}" \
    --subnet-id="${SUBNET_ID}"

```

6.10.2 Cluster management: Deletion

To delete a cluster, administrators will have to first delete all the node pools which belong to a cluster, before deleting the cluster.

```

gcloud alpha container azure node-pools delete NODE_POOL_NAME \
    --cluster CLUSTER_NAME \
    --location GOOGLE_CLOUD_LOCATION

gcloud alpha container azure clusters delete CLUSTER_NAME \
    --location GOOGLE_CLOUD_LOCATION

```

With an autoscaler ready to go with Anthos on Azure, it is easy for the administrator to control costs and manage minimum resource requirements for each cluster. It is recommended to have a security device like Hashicorp Vault to store the ssh keys for retrieval and rotation.

6.11 Summary

The best way to learn is by trying and the best advice is to try building clusters on the various providers to understand the optimizations available, and the actions that an administrator would need to do in their day to day life managing operations in Anthos. This is key to build a continuous improvement process as new features in Anthos are released to make kubernetes cluster management always easier and faster.

After reading this chapter, the reader should be able

- To utilize Google Cloud Console to operate and manage workloads of the various Anthos cluster types.
- Understand the various logging and monitoring options available with their Anthos deployments and the criteria to consider.
- Understand how to operate and manage various types of Anthos deployments through the command line and how and what kind of communication happens between Google Cloud and the deployments.

- Upgrade, scale and design operations management procedures in Anthos across an hybrid environment

In the next chapter the features described in the previous few chapters will be discussed as an overview of the Anthos product, roadmap and strategy.

8

Hybrid applications in Anthos

by Jason Quek

This chapter covers:

- Highly available applications
- Geographically distributed applications
- Applications regulated by law
- Applications which have to run on the edge

In the real world, applications are bound by rules such as data locality requirements, resource constraints, situations where a stable connection to the cloud cannot be guaranteed—such as at a baseball stadium, a construction site or on a fighter jet—or to do low latency computation locally on the edge to avoid large amounts of data transfer. That an application must be available and survive a regional disaster or cloud outage.

However, there still exists a need to run applications with the same consistency and stability that the Kubernetes platform provides. Thus solutions such as Anthos are designed to create this conformant distributed cloud platform for such applications.

In this chapter we will go over the different situations and show various architectures involving the use of Anthos and its suite of products to support these types of applications.

8.1 Highly available applications

These are a class of applications which have a need to be running 24 hours a day, 7 days a week. Financial institutions managing transactions, healthcare applications, traffic management are just some of the examples of applications which have an impact on the real world if unavailable for a short period of time.

In Google Kubernetes Engine, clusters can be created to span across availability zones within a region. This provides insurance in the event one of the availability zones within a region goes down; the other availability zones are still running the application. The Kubernetes scheduler would then spin up additional replicas on the nodes still active on the other availability zones to meet replica requirements defined in the Kubernetes configuration.

However, what if an entire region was down, or an entire cloud provider? An article found in Forbes¹ mentioned a scenario where a banking client who wanted to use a single cloud and the regulator was concerned about what would happen during an outage. The client estimated they would have a maximum of two hours downtime during the cloud outage but was rejected by the regulator as not a viable option.

In this situation, companies would have to span multiple clouds and regions, but this would introduce complexity and overhead in managing multiple cloud providers and required double the skill set across their operations, development and managing security at the same time.

With Anthos, companies would be able to standardize on Kubernetes with one unified pattern of deployment, scaling, security, development, while still being able to leverage the availability of multiple cloud providers and move workloads across regions and availability zones.

8.1.1 Architecture

The setup in Figure 8.1 is a simple setup, installing Anthos GKE, GKE on AWS and GKE on Azure and including them all into a service mesh. Such a setup would give high availability not just on the regional level but also at the cloud provider level.

¹<https://www.forbes.com/sites/tomgroenfeldt/2020/06/22/google-anthos-simplifies-synch-for-multi-cloud-implementations/?sh=68b181114c42>

Architecture: Highly available application architecture

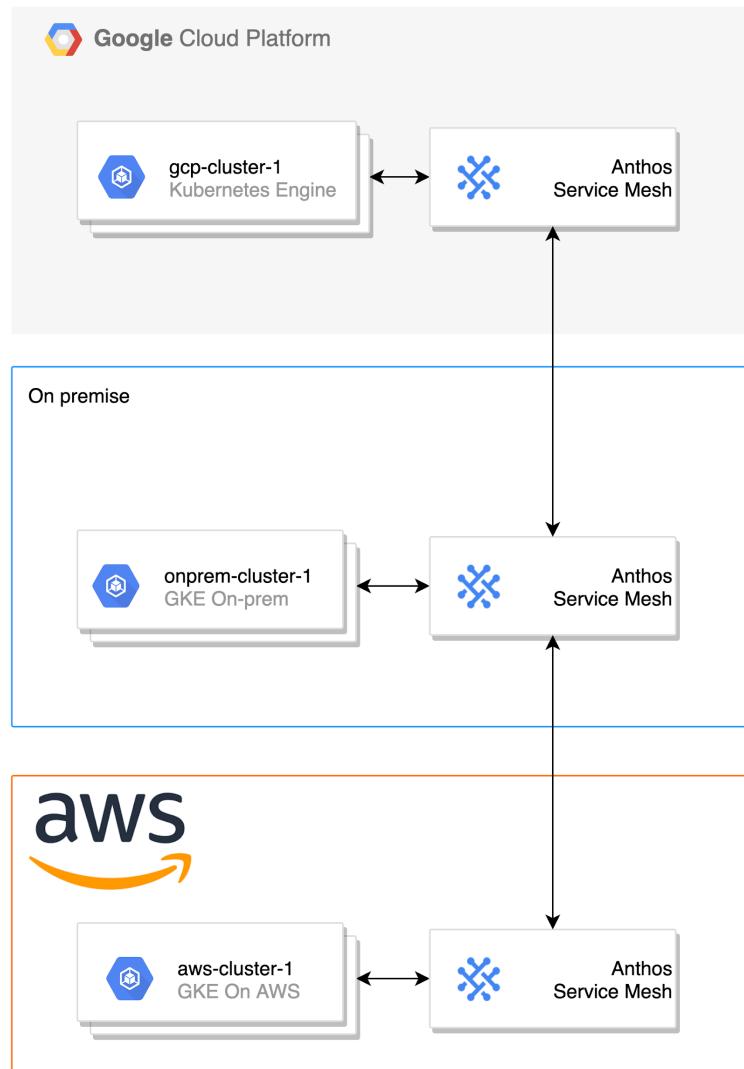


Figure 8.1. Multi Cloud availability with Service Mesh connectivity

Do note that this assumes that the application just has to be available but may not be required to be accessible from the internet. This is an important distinction as this architecture will survive even in the extremely unlikely event that one of the cloud providers

has a severe outage, as Anthos clusters can still function even if they cannot reach Google Cloud APIs for a limited amount of time.

8.1.2 Benefits

Installation of the managed Kubernetes service across all three different clouds may seem simple, but each of them have different processes, options and best practices. Versions of Kubernetes available also differ across cloud providers. By using Anthos, such issues are handled by Google engineers who work behind the scenes to make sure that it simply works on the different clouds. Solutions such as Workload Identity can be deployed across multi and hybrid clouds to provide a unified authentication framework.

8.1.3 Limitations

Such applications will normally require somewhere to persist state, which then creates a dependency on a provider unless the organizations choose to host their own data solution on the cloud. With the emergence of Kubernetes native databases, such as CockroachDB, which could be deployed across multiple clusters, or using MirrorMaker to replicate messages across Kafka deployments in multiple clusters, this issue is beginning to have robust solutions.

Anthos also does not manage the underlying networks, storage and compute used to build the clusters, and organizations would still need to manage networking and ensure storage and compute availability across hybrid and multiple clouds.

It is important to understand that such architecture might not be easy to build and maintain but for organizations which have regulatory, financial and reputational reasons to have their applications always available and disaster resistant, Anthos provides a path to that.

8.2 Geographically distributed applications

These applications have the requirement to be placed around the world due to the need to serve a worldwide audience.

Having a managed service to be able to route to the nearest cluster from one single IP from anywhere in the world, makes application scaling across the world much simpler.

The most important point for this is to be able to provide access to this application to the user from the location nearest to them, to minimize latency. The application is often an exact copy of microservices deployed to clusters in different regions, but with no requirement that it must span multiple clouds. This coupled with multi geographical databases which can be used either as a managed service from Google, (e.g. Spanner) or CockroachDB which can be deployed across multiple Kubernetes clusters spanning the globe.

However, with Anthos, the same application can be deployed in multiple regions and opens up new opportunities in various markets. One important component that ties this together is Ingress for Anthos, a cloud-hosted gateway for distributed clusters.

With Ingress for Anthos, all users of the application can access it through an anycast IP, and get routed to the cluster nearest to the user.

One use case where this is important is in the retail sector where customer churn can be proven to be directly related to latency of the e-commerce site.

8.2.1 Ingress for Anthos Architecture

Ingress for Anthos builds on top of the existing Google HTTP load balancing architecture, utilizing Google Cloud Points of Presence to route efficiently to the nearest available cluster, as can be seen from Figure 8.2. With such an architecture, applications which need to serve a new audience in a different region, can be deployed on an Anthos cluster on that specific region, and Ingress for Anthos can be configured to start sending traffic to that new region, while keeping the same IP address and domain name.

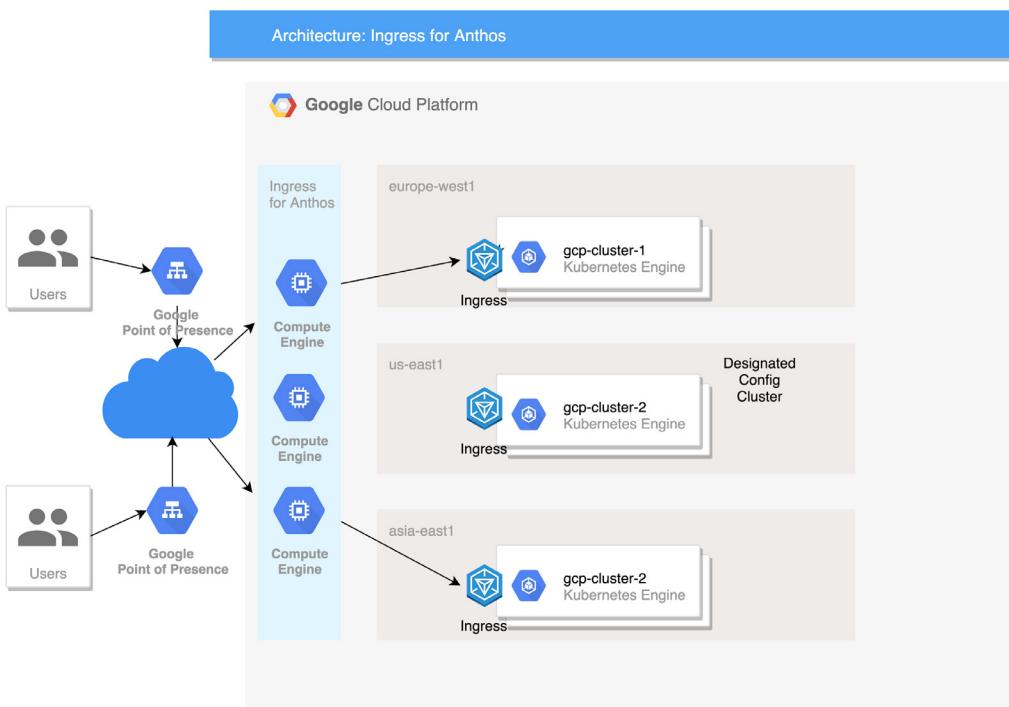


Figure 8.2. Ingress for Anthos architecture

Any cluster can be designated as a config cluster, and two custom Kubernetes objects *MultiClusterIngress* and *MultiClusterService* are deployed on this config cluster. This is just a centralized configuration storage for the Anthos Ingress Controller to read from.

The Anthos Ingress Controller is a set of Compute Engine instances which sit in multiple Google Cloud regions outside of the user's control and is managed by Google. These instances have to be placed nearest to the Google points of presence where traffic enters in

order to make routing decisions based on Anthos clusters who are members in the same Anthos environ² and pod availability.

The key concept for Ingress for Anthos is in the use of Network Endpoint Groups (NEGs). Network Endpoint Groups are groups of backend endpoints or services, which can be deployed on Anthos Clusters. The Compute Engine Instances in the Ingress for Anthos service seen in Figure 2 then route to the correct NEG based on the availability of the service.

To understand more about Ingress for Anthos networking, please refer to the chapter Anthos, the networking environment.

8.2.2 Ingress for Anthos Benefits

To create load balancers which can route traffic to clusters from points of presence, which are also aware of the state of service availability of each cluster is a complex task.

This breaks down the problem to a single cluster issue, and developers are able to concentrate on building more features and know that this can be deployed in a standardized way across the globe and accessible to their users with low latency.

8.2.3 Ingress for Anthos Limitations

Ingress of Anthos is currently restricted to only Anthos Google Kubernetes Engine clusters, and does not support GKE On-prem or GKE on AWS clusters. To have a solution which also supports GKE On-prem clusters, the Traffic Director solution would be an option.

8.3 Hybrid Multi Cloud applications with internet access

Some enterprises have invested in remote data centers, or have a multi cloud strategy, which makes Anthos a good fit as a product for them. However when exposing their services to the public internet, there is a need to protect these applications so that bad actors are unable to disrupt their availability and latency. One limitation of Ingress for Anthos is that these clusters have to be on Google Cloud, but there is also a need to route to applications deployed across private and public clouds. Thus Traffic Director, which is able to route traffic to clusters in a hybrid multi cloud architecture..

8.3.1 Traffic Director Architecture

Traffic Director architecture looks very similar to the Ingress for Anthos setup, but a key difference is that all the services have to be in a Service Mesh for Traffic Director to work. For more details on Service Mesh, please refer to the Anthos Service Mesh and Networking chapter. See Figure 3 for an overview of the use of the Traffic Director for routing of hybrid applications.

²See chapter Operation Management in Anthos for environ definition

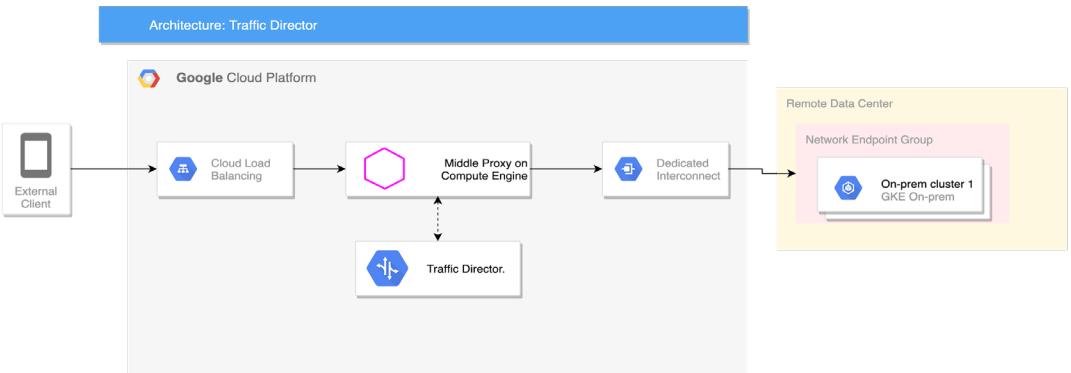


Figure 8.3. Traffic Director Architecture for Hybrid Applications

In Figure 3 the GKE On-prem cluster provides backend services to be used with the Traffic Director. Traffic Director then directs the Google Cloud load balancer to direct traffic to the GKE On-prem cluster. This paradigm works for Anthos clusters on other Clouds as well.

8.3.2 Traffic Director Benefits

Traffic Director allows the use of Google Cloud load balancers to front hybrid applications, and in doing so, able to use Cloud native services such as Google Cloud Armor³, Cloud CDN⁴, Identity-Aware Proxy⁵ and Managed Certificates⁶.

All traffic incoming to the application, regardless of where the services are hosted, will enter via Google Point of presence to a Google Cloud Balancer, which is then proxied via Envoy proxies programmed by Traffic Director to a service which is part of the service mesh. The service can be hosted on premise, across multiple clouds, but is known through the service mesh.

By using these services, hybrid applications on premise can be protected from Denial of Service (DDOS) attacks, which can be a problem for hybrid applications.

Traffic Director can also direct traffic to Google Compute Engines if the application is not running on Google Kubernetes Engine.

Traffic Director can also be used to split traffic between cloud services and on-prem services, to aid migration of services from on-prem to the cloud with no downtime.

8.3.3 Traffic Director Limitations

In Figure 3, the Middle Proxy is deployed on an managed instance group which scales according to traffic from the external load balancer before forwarding it to the GKE On-prem cluster. This is additional compute cost which is borne by the application owner.

³ <https://cloud.google.com/armor>

⁴ <https://cloud.google.com/cdn>

⁵ <https://cloud.google.com/identity-aware-proxy>

⁶ <https://cloud.google.com/load-balancing/docs/ssl-certificates/google-managed-certs>

8.4 Applications regulated by law

There is a set of applications that belong to highly regulated industries which are bound to data locality restrictions. A list of these industries include financial institutions, healthcare providers, pharmaceutical companies and government agencies.

Such applications require constant monitoring and complicated audit and security policies which are aided by the use of GKE On-prem, Anthos Config management, Anthos Service Mesh and VPC Service Controls.

8.4.1 Architecture

VPC Service Controls enable administrators to restrict access to certain Google Cloud APIs to allowed IP addresses, identities and client devices. Such APIs include `gkehub.googleapis.com`, `gkeconnect.googleapis.com`, `meshca.googleapis.com` and `containerregistry.googleapis.com` which are the GKE Hub, GKE Connect, Anthos Service Mesh and Container Registry services utilized throughout the Anthos offering. This utilizes the BeyondCorp⁷ concept made popular by Google Cloud, that trust should be built on identity not on networking, and a zero-trust policy is used within a network. This allows users to work more securely from any trusted locations without a VPN. See Figure 8.4 for a visualization of the setup.

⁷ <https://cloud.google.com/beyondcorp>

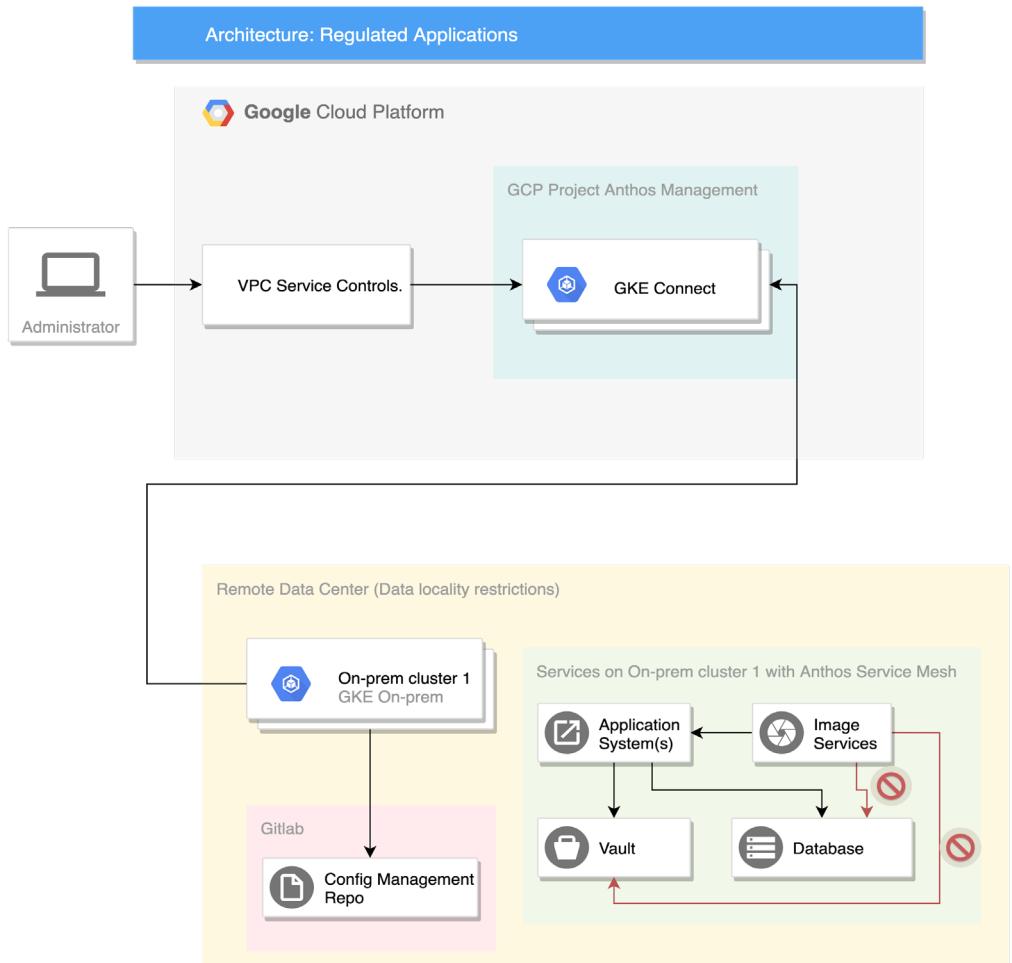


Figure 8.4. Regulated application with VPC Service Control, ACM and ASM

Enterprises are able to configure Anthos Clusters to read config management from a repository hosted on premise, and set up Role Based Access Controls in one location, and have that setup propagated through a continuous sync. With Policy Controller deployed with Anthos Config Management, administrators and security teams are able to define policies to restrict use of unapproved container registries, prevent creation of privileged pods, and define read only operating system file systems. To understand how to do this in detail, please refer to the chapter Using Anthos for Security and Policies.

With Anthos Service Mesh, administrators are able to enforce mutual TLS communication between all pods as well as define which pods are able to communicate with each other and only with each other, nothing more. This prevents unauthorized access to sensitive data and

prevent data exfiltration in the event a rogue pod is deployed on the system. For a deeper dive into this subject, please refer to the chapter “Anthos Service Mesh: security and observability at scale”.

8.4.2 Benefits

All the services mentioned above are built for purpose and have security in their highest consideration. For example if the Anthos Config Management is disconnected from the repo, the last synced policies are still in effect. All components have also been tested to work with each other and any issues is a quick ticket away from Google Support.

The above architecture is also extensible and automatable, so that new on-premise regulated clusters can be created, hooked up to the same ACM, ASM artifacts and benefit from the work already done.

8.5 Applications which have to run on the edge

The definition of edge devices have different meanings for different companies and business use case.

For the specific telecom edge use case, this would be referring to computing requirements for 5G capabilities. Please refer to chapter 11, Anthos, at the edge and telco world for a detailed look into how Anthos enables that.

Edge devices can also be used by retailers, remote manufacturing sites, where applications can be deployed closer to users to provide a low latency experience while delivering high performance compute. One example is calculating statistics of a baseball game right in the stadium while a baseball game is going on at the same time, and delivering those statistics in real time to commentators and the audience which was a driving force behind why Major League Baseball chose to use Anthos to process and analyze data in the cloud as well as on-premise at each of their ballparks.⁸

These applications have the need to run on edge appliances with low resource requirements without a dependency on a continuous internet connection. The biggest issue would relate to deploying the latest versions of your software over the air to these edge nodes. Anthos at the edge provides this while still giving developers the ability to deploy their applications uniformly in a cloud native way, and the trust that their applications would work in the same way as on the cloud due to the conformant deployment of Kubernetes.

8.5.1 Architecture

The architecture in Figure 5 shows an architecture for retail stores of the future, where shoppers can shop without any cashiers, while still being able to monitor purchases securely through video and transact via an application in store, as well as identifying hot spots of traffic, identifying when to restock good, complying with privacy regulations while providing a real time low latency experience.

⁸ <https://cloud.google.com/blog/topics/anthos/how-mlb-is-using-anthos>

Architecture: Anthos at the Edge

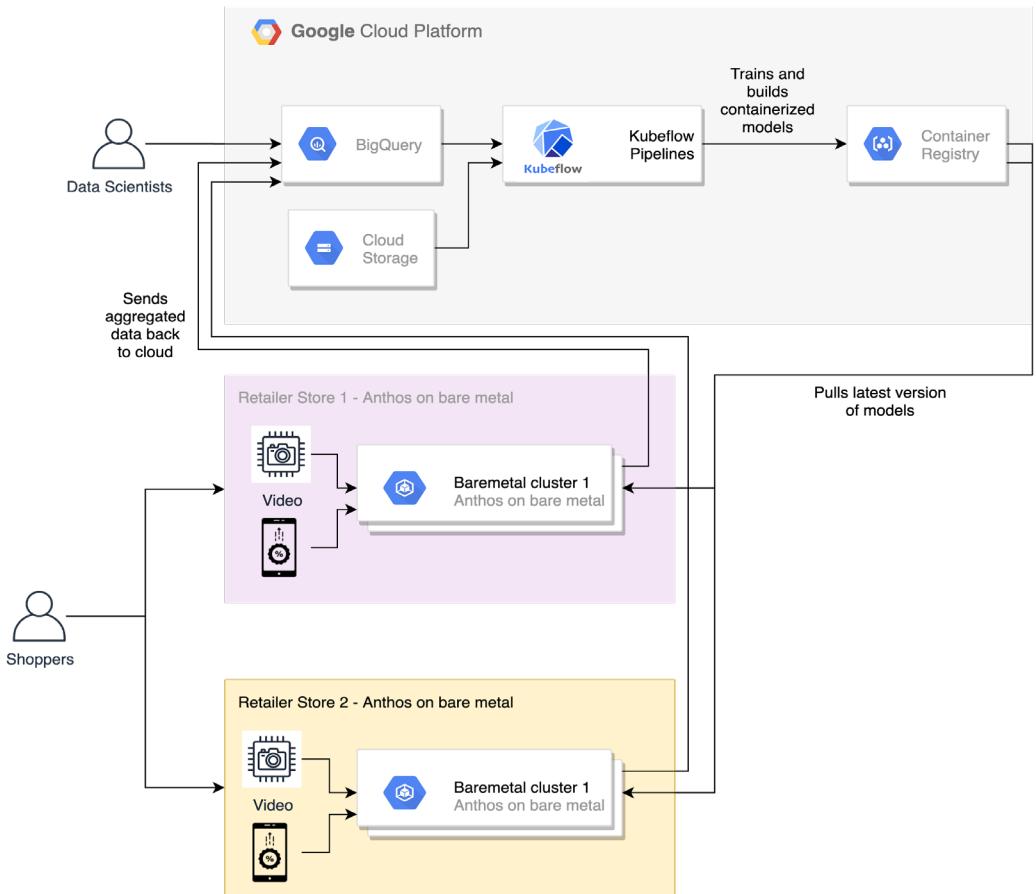


Figure 8.5. Anthos at the Edge retail architecture

For such a use case, many machine learning models would be used for detection of theft, detection of low stocks, recommendations, traffic hot spots. Models such as RetailNet can be used for people counting and hot spot detection in retail stores, reference here⁹. What is required is a way to continuously train and update the model and deploy it reliably on the edge with high performance compute for inference, which can be seen from Figure 5 will utilize Anthos on bare metal to deliver just that. Streaming video data back to the cloud for inference would require large bandwidth and cause a lag degrading the real time experience in the store.

⁹https://www.researchgate.net/publication/334782772_RetailNet_A_Deep_Learning_Approach_for_People_Counting_and_Hot_Spots_Detection_in_Retail_Stores

Aggregated and anonymized data can be then sent back to the cloud and models can be trained via Kubeflow and updated, which can be then deployed on premise again, creating a feedback loop to continuously update and enhance the accuracy of the models. To understand more in detail on how to use Kubeflow, please consult the chapter Anthos, an end-to-end example of ML application.

8.5.2 Benefits

One key point is also the improved performance of applications on bare metal due to the direct application access to hardware and skipping the cost of a virtual machine license.

Anthos on bare metal 1.9 is also able to use a private registry mirror instead of gcr.io to pull container images so that it is able to run completely air gapped from the internet.

8.5.3 Limitations

Without a virtualization layer, node scaling would require the installation of new hardware nodes and connecting them to the fleet of bare metal devices.

8.6 Summary

The different Google Cloud and Anthos services have an important place towards the different use cases of high availability, geographical spread, regulated industries and edge computing. Many enterprises will have a need to achieve these different use cases to enrich their digital offerings and can use the prescribed architectures to do this.

After reading this chapter, the reader should be able

- Identify the different types of applications which can utilize the Anthos feature set to achieve higher uptime, remove single cloud dependencies and serve a customer base globally while adhering to the terrestrial laws
- Understand the benefits and limitations of the different architectures such as having to design multi geographical state persistence, as well manual node installations for edge deployments
- Utilize different Google Cloud features in conjunction with Anthos to build out the different types of applications.

In the next chapter the reader will deep-dive into Anthos and how the compute environment looks like on VMware.

9

Anthos, the compute environment running on VMware

by Jaroslaw Gajewski

This chapter covers:

- Understanding VMware and Anthos architecture
- Deployment of the management plane
- Deploying a user cluster
- Anthos networking load-balancer options

As you already know, Google provides a managed Kubernetes service called Google Kubernetes Engine (GKE) hosted on their infrastructure. This managed solution has solved multiple problems for Kubernetes consumers including lifecycle management, security and physical resource delivery. While deploying applications in the Cloud has been growing each year, many enterprises still have a significant footprint of resources in their own data centers.

Google recognized that organizations may have various reasons to keep certain workloads on-prem, including latency concerns, data concerns, and other regulatory requirements that may limit Cloud usage. To help companies address workloads that must remain on-prem but still want to leverage Cloud native technologies like Kubernetes, on-prem GKE was born - now known as Anthos.

On-prem implementations introduce some requirements that companies must provide to allow provisioning of Anthos infrastructure components. We will not go into details of constraints that are related to a particular version of Anthos, or the VMware requirements since it's always evolving. In general there is a requirement to provide VMware vSphere

resources⁴ that can be used for hosting management control planes, Kubernetes infrastructure and application workloads.

Anthos on VMware deploys a management control plane built on Kubernetes, the admin cluster, by provisioning vSphere Virtual Machines (VMs). Every new user cluster is provisioned as a collection of dedicated VMs with a Kubernetes control plane and user nodes, fully managed by the admin cluster. Load balancing can be delivered as VMs managed by Anthos on VMware instances or external load balancers, depending on use case and implementation architecture.

In this chapter we will explain the various deployment scenarios and requirements to install Anthos on VMware in an on-prem data center, beginning with why you may want to deploy Anthos on-prem.

9.1 Why should I use Anthos on VMware?

Let's address the first question - Why did Google elect to support VMware with the initial version of Anthos? Most enterprises have an existing VMware footprint, leveraging a virtual platform providing a consistent environment and API for automated, full-stack provisioning. By supporting VMware, Google provides organizations the ability to create Kubernetes clusters leveraging existing investments, infrastructure, and scaling and node management capabilities.

In general there are multiple drivers and constraints that require businesses to keep certain data and workloads running in dedicated data centers (DCs). Most common requirements are requirements for proximity to data or users, regulatory, or the need to leverage existing on-premises investments.

Companies must increase application availability and scalability to remain competitive. Developing applications in containers will provide efficiency, agility, and portability - which provide developers the ability to increase their deployment agility and velocity. Organizations that already have DC's may be able to provide a Cloud-like experience for their developers cheaper than moving to a CSP. Leveraging on-prem capacity, you can create a cloud-like experience that feels similar to using Kubernetes on GCP, offering your developers a cohesive experience for clusters running either on or off-prem.

Kubernetes management, maintenance, integration and lifecycle management requires a significant amount of time and skills. Running clusters at a production scale increases the daily challenges, diverting business attention from applications into infrastructure management. That's where Anthos on VMware comes in, by including conformant, security tested versions of Kubernetes, bundled with integrations into a variety of network services and most importantly, enterprise support for Kubernetes and all Anthos components. Last, but not least, all clusters are unified, providing full management from the GKE console.

Based on company needs, authorization to GKE clusters on-prem can be provided by:

- OpenID Connect² based providers integrations, like
 - Google based,

⁴https://cloud.google.com/anthos/gke/docs/on-prem/how-to/vsphere-requirements-basic#resource_requirements_for_admin_workstation_admin_cluster_and_user_clusters

²<https://developers.google.com/identity/protocols/oauth2/openid-connect>

- company Active Directory Federation Services (ADFS) based
- Lightweight Directory Access Protocol (LDAP)

Anthos on VMware includes a number of addons to help administrators and security officers enable management, shift to git-based management and progressive deployments to achieve deployment velocity and agility in Enterprise. Anthos Configuration Management (ACM) allows full implementation of GitOps approach for infrastructure as a code (IaaC). It includes an Open Policy Agent based admission controller, providing security constraints and enterprise standards in a cloud native infrastructure. ACM includes multiple predefined constraints, and it can be extended to any custom policies from business or security. Both tools are described in detail in separate chapters but it's important to highlight that they are out of the box extensions available on Anthos on VMware implementation.

In the next section, we will look at the architecture requirements to successfully deploy Anthos on a VMware cluster.

9.2 Anthos on VMware Architecture

Anthos on VMware is implemented as a collection of VMs deployed on VMware vSphere clusters. In that chapter we will go into the technical requirements for an Anthos on-prem implementation.

Starting with the general requirements, Anthos on VMware must be installed on a vSphere cluster backed by a standard or distributed virtual switches or VMware NSX-T software defined networking. Since Google updates Anthos often, detailed requirements related to the most recent version is available on GCP Anthos on VMware documentation page³.

Just like the OSS Kubernetes Release, Anthos has an agile release cycle⁴, inline with the entire Anthos release, it follows monthly patch release cycles and quarter cycle for version upgrades. It's recommended to deploy version updates, we can upgrade directly to any version of same minor release or next minor release, for example we can upgrade from version 1.9.0 to 1.9.4 or directly to 1.10.1, but to upgrade from 1.9.X to 1.11.X you need to perform upgrade to 1.10.X first.

³ <https://cloud.google.com/anthos/gke/docs/on-prem/how-to/vsphere-requirements-basic>

⁴ <https://cloud.google.com/anthos/clusters/docs/on-prem/version-history>

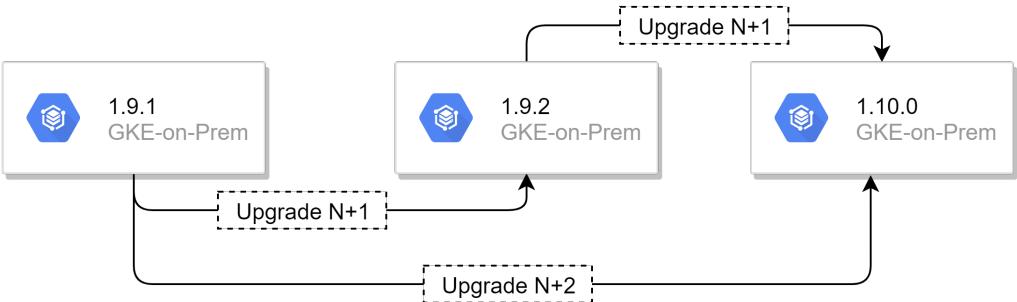


Figure 9.1. Upgrade options

To simplify the update process and to allow testing of new releases, starting with Anthos version 1.7.0, user clusters can be updated before the admin cluster upgrade. As a result, you can spin up a new cluster with a newer version of Anthos, test it, and based on the results, start a lifecycle process to upgrade the entire environment to the new version. Once the user clusters are upgraded, the admin cluster can be updated in a suitable window without pressure to do that upfront.

Now that you know a high level view of the Anthos on VMware, we can move on to details to install a new cluster.

An on-prem deployment consists of an admin workstation, equipped with all required tools for the deployment of the admin cluster and user cluster(s). The installation requires three configuration files that must be configured for the environment:

- Admin workstation configuration file
- Admin cluster configuration file
- User cluster configuration file

Each of the files are in YAML format, and contain mandatory and optional sections. By default each optional section is commented out and each configuration element is followed by short description and usage, providing an easy to follow, self-documented configuration file.

Some of the options in the configuration files include:

- Configuring the installation to use an on-prem registry for Anthos component images
- Google service accounts for GCP integration including the accounts to use for Anthos connect, and cloud logging
- OIDC configuration
- vSphere configuration
- Load-Balancer configuration

The configuration contains options for a base cluster configuration, with the one exception of enabling Cloud Run. Additional elements like Anthos Config Management, Anthos Policy Controller or Service Mesh can be installed or removed at any time during the cluster life cycle.

The overall deployment flow is presented in Figure 9.2 - Anthos on VMware Deployment flow. All options and optional steps are marked by dotted lines.

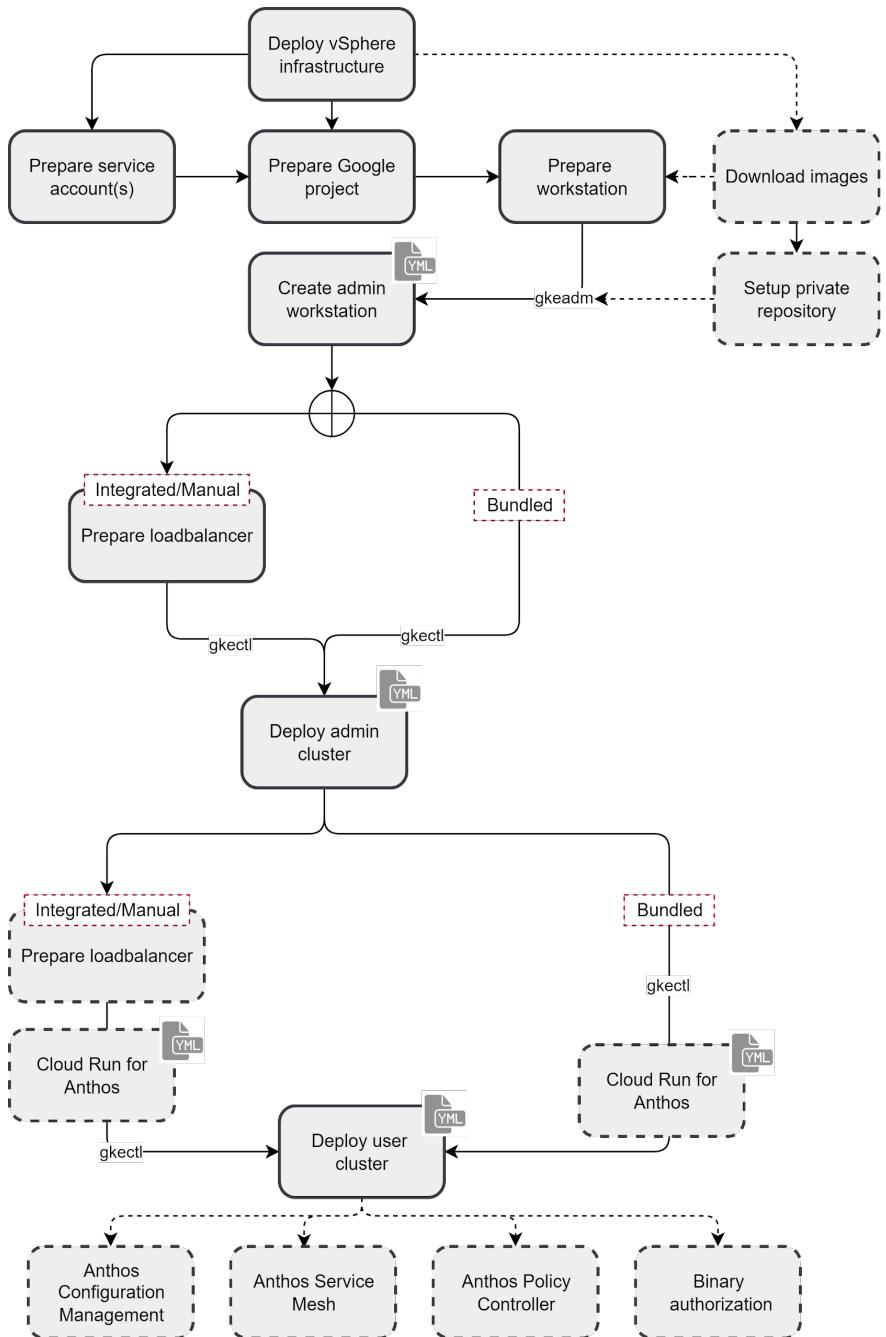


Figure 9.2: Anthos on VMware Deployment flow

In the next section we will explain how to deploy the management plane, which is the first step for a new on-prem deployment.

DEPLOYING THE MANAGEMENT PLANE

The creation of the management plane begins with the deployment of the admin workstation, providing the tools to create and provide administration for the admin and user clusters. This VM has all of the software required to create an Anthos cluster on VMware. Using an admin workstation, Google can guarantee consistency and remove the responsibility to create and maintain tools from administrators.

ADMIN WORKSTATION PREPARATION

Deployment of an admin workstation requires preparation, the graphic below shows the steps to create the admin workstation.

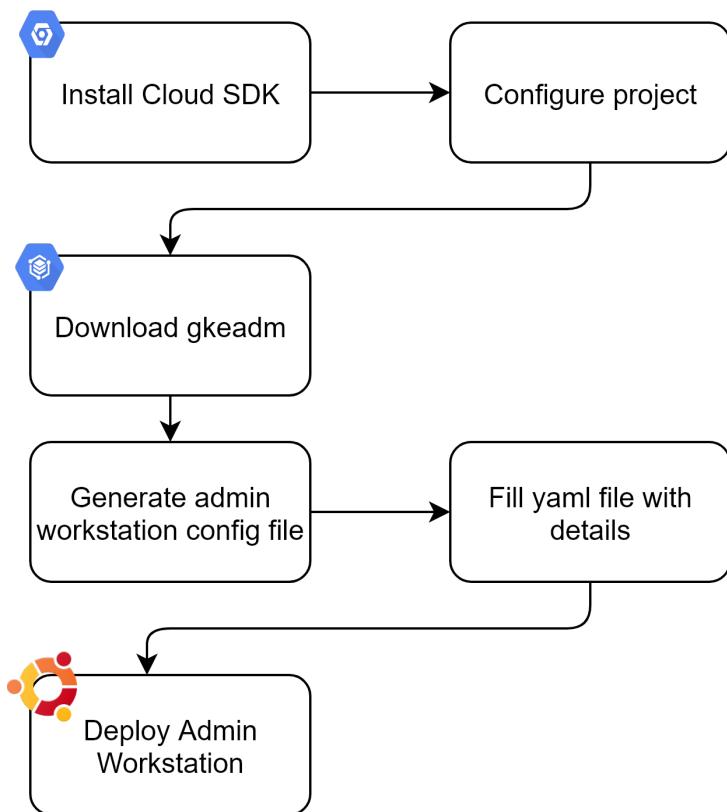


Figure 9.3. Admin workstation deployment

The first step is to install the Google Cloud SDK on your local machine. This can be any Linux, Windows (workstation or server) or MAC OS machine.

Once you have installed the SDK, you need to download the `gkeadm` utility. Using the `gsutil` utility, download the file using the command below.

```
gsutil cp gs://gke-on-prem-release-public/gkeadm/{Anthos on VMware version number}/linux/gkeadm ./
```

If you are using Linux, you will need to make the binary executable using `chmod`.

```
chmod +x gkeadm
```

Now that you have `gkeadm` downloaded, you can deploy the admin workstation.

DEPLOYING THE ADMIN WORKSTATION

Deployment of the admin workstation is based on a single YAML configuration file, which has four main sections:

- `gcp`
- `vCenter`
- `proxyUrl`
- `adminWorkstation`

The GCP section is dedicated to GCP integration configuration elements. At the time of writing this book, only one element is required, the `componentAccessServiceAccountKeyPath`, which defines a path to a component service account. This service account has rights to access Anthos on VMware binaries and consume the related APIs. If different projects are used for monitoring logging etc that account must have `serviceusage.serviceUsageViewer` and `iam.roleViewer` on each of those projects.

```
gcp:
  componentAccessServiceAccountKeyPath: "whitelisted-key.json"
```

The second section in the configuration file describes integration with a vCenter server. It covers the credentials subsection with definitions for `vCenter`, and a credentials file containing the username and password required for vSphere access. A credential file, named `credential.yaml`, is shown below.

```
apiVersion: v1
kind: CredentialFile
items:
- name: vCenter
  username: "myaccount@mydomain.local"
  password: "Th4t1$4Nth05"
```

The required vCenter information includes the datacenter name, datastore name, cluster name, network, and the vCenter root certificate. Optionally, a resource pool name and folder can be specified to place the VM in.

```

vCenter:
  credentials:
    address: "10.10.10.10"
    fileRef:
      path: credential.yaml
      entry: vCenter
  datacenter: "MY-DATACENTER"
  datastore: "MY-DATASTORE"
  cluster: "MY-CLUSTER"
  network: "MY-VM-NETWORK"
  folder: "MY-FOLDER"
  resourcePool: "MY-POOL"
  caCertPath: "vcenter-root.cert"

```

if you have a self signed certificate for your vCenter server, which needs to be added to the **caCertPath** value, you can obtain it using curl, as shown in the command below.

```
curl -k "https://{{SERVER_ADDRESS}}/certs/download.zip" > download.zip
```

Once downloaded, unzip the file, which will extract the certificate in the certs/lin folder.

If your environment requires a proxy server to access the internet, you can configure the proxyUrl section. This configuration parameter is used only by the *gkeadm* command during the VM deployment.

```
proxyUrl: "https://my-proxy.example.local:3128"
```

When a proxy is configured, you will also need to add the appropriate addresses to the OS or system *no_proxy* variable. This configuration is specific for each company and deployment - a full explanation of how proxy servers work is beyond the scope of this book. As a starting point, you may need to add your vCenter server, local registry (if configured), and the CIDR range for the ESX hosts.

The Last section is the only one that comes partly pre populated during configuration file generation:

- dataDiskName
- dataDiskMB
- Name of the VM
- Amount of cpus
- Amount of memory in MB
- Base disk size in GB

NOTE: dataDisk folder where new disk is created must exist. As result you must create it manually upfront.

The admin workstation can be assigned an IP address using a static IP assignment or by a DHCP server. Your implementation choice is defined in the network subsection of the configuration file using the *ipAllocationMode* property.

For DHCP use cases *ipAllocationMode* must be defined as DHCP, and all other child network configuration elements remain undefined.

When using a static IP assignment, the ipAllocationMode property must be set to “static”, followed by IP, gateway, netmask and DNS configuration. The DNS value can be defined as an array of properties with multiple values.

Finally, set the NTP server used by the admin workstation. It’s mandatory to use NTP that is in sync with vSphere infrastructure otherwise time differences will cause deployment failures.

Two example configuration files are shown below, the first has been configured to use a static IP and the second has been configured to use DHCP.

```
adminWorkstation:
  name: "gke-admin-ws-200617-113711"
  cpus: 4
  memoryMB: 8192
  diskGB: 50
  dataDiskName: "gke-on-prem-admin-workstation-data-disk/gke-admin-ws-data-disk.vmdk"
  dataDiskMB: 512
  network:
    ipAllocationMode: "static"
    hostConfig:
      ip: "10.20.20.10"
      gateway: "10.20.20.1"
      netmask: "255.255.255.0"
      dns:
        - "172.16.255.1"
        - "172.16.255.2"
  proxyUrl: "https://my-proxy.example.local:3128"
  ntpServer: "myntp.server.local"
```

```
adminWorkstation:
  name: "gke-admin-ws-200617-113711"
  cpus: 4
  memoryMB: 8192
  diskGB: 50
  dataDiskName: "gke-on-prem-admin-workstation-data-disk/gke-admin-ws-data-disk.vmdk"
  dataDiskMB: 512
  network:
    ipAllocationMode: "dhcp"
    hostConfig:
      ip: ""
      gateway: ""
      netmask: ""
      dns:
  proxyUrl: "https://my-proxy.example.local:3128"
  ntpServer: "myntp.server.local"
```

Now we can create the admin workstation on our vSphere infrastructure using the gkeadm utility.

```
./gkeadm create admin-workstation --auto-create-service-accounts
```

Adding the auto-create-service-accounts flag allows you to automatically create the associated service accounts in your project.

Once the admin workstation has been created, you are ready to deploy the admin cluster. In the next section, we will go through the steps to create your admin cluster.

CREATING AN ADMIN CLUSTER

The admin cluster is the key component of the Anthos control plane. It is responsible for the supervision of Anthos on VMware implementations, and the provisioning and management of user clusters. It's deployed as a Kubernetes cluster using a single control plane node and two worker nodes [Figure 9.4].

The control plane node will provide the Kubernetes API server for the admin control plane, the admin cluster scheduler, the etcd database, audit proxy and any load balancer integrated pods.

Worker nodes are providing resources for Kubernetes addons like kube-dns, cloud monitoring (former stackdriver) or vSphere pods.

In addition to the admin control plane and addons, the admin cluster hosts the user control planes. As a result, the user clusters API server, scheduler, etcd, etcd maintenance and monitoring pods are all hosted on the admin cluster.

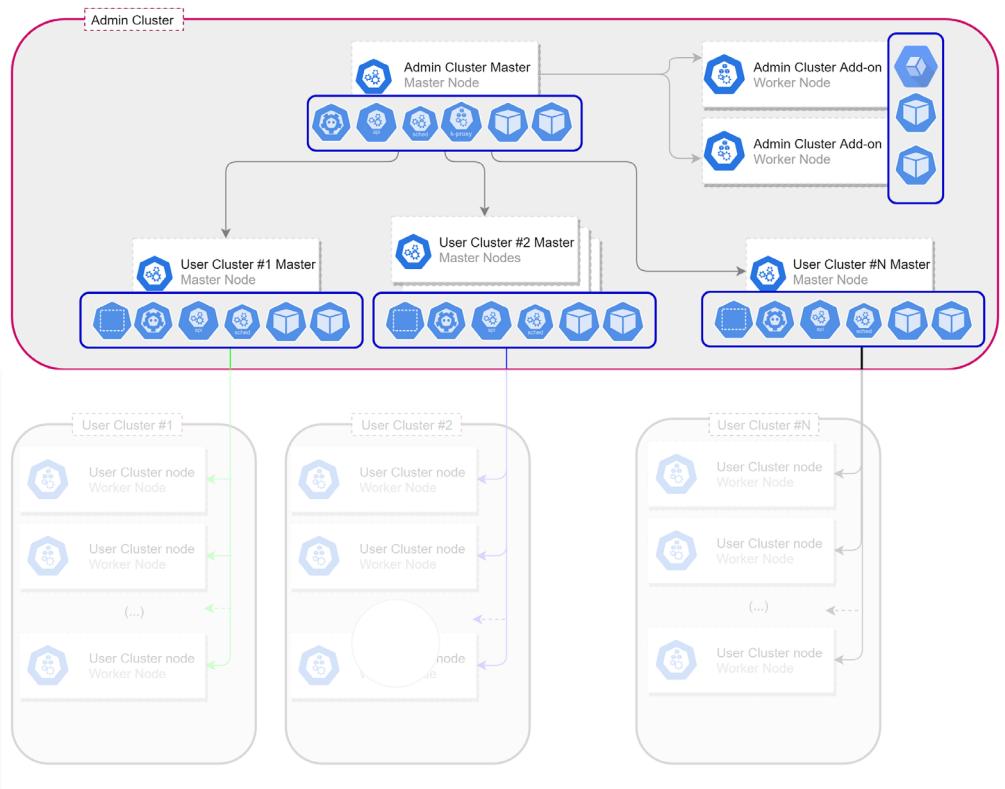


Figure 9.4. Anthos on VMware admin cluster architecture

To create an admin cluster, you will need to SSH into your admin workstation that was created in the last section. SSH into the admin workstation using the key that was created when you deployed the admin workstation, located at `.ssh/gke-admin-workstation..`

```
ssh -i /usr/local/google/home/me/.ssh/gke-admin-workstation ubuntu@{admin-workstation-IP}
```

Similar to the admin workstation creation process, the admin cluster uses a YAML file that is divided into sections. The vCenter, gkeconnect, stackdriver and gcrkeypath sections are pre populated with values gathered from the admin workstation YAML⁵ file, while all other sections must be filled in for your deployment, prior to creating the cluster.

You can use the included admin cluster configuration file, or you can generate a new admin cluster configuration file using the gkectl tool. Unlike the pre-created template file, any templates generated manually using gkectl will not contain any pre-populated values. To create a new file, use the gkectl create-config admin option.

```
gkectl create-config admin --config={{ OUTPUT_FILENAME }}
```

Both creation methods will contain the same sections, the first two sections of the configuration file must remain unchanged, defining the API version and cluster type.

```
apiVersion: v1
kind: AdminCluster
```

The next section is dedicated to the vSphere configuration, containing the requirements for the Virtual Machines and disks placement.

TIP: A good practice is to always use fully qualified domain name for vCenter and avoid usage of IP addresses in production environments.

```
vCenter:
  address: "FullyQualifiedDomainName or IP address of vCenter server"
  datacenter: "vCenter Datacenter Name"
  cluster: "vCenter Cluster name"
  resourcePool: "vCenter Resource Pool name"
  datastore: "vCenter Datastore Name for GKE VM placement"
  folder: "Optional: vCenter VM Folder"
  caCertPath: "vCenter public certificate file"
  credentials:
    fileRef:
      path: "path to credentials file"
      entry: "Name of entry in credentials file referring to username and password of
              vCenter user account"
  # Provide the name for the persistent disk to be used by the deployment (ending
  # in .vmdk). Any directory in the supplied path must be created before deployment
  dataDisk: "Path to GKE data disk"
```

TIP: A good practice is to place the data disk into a folder. The `dataDisk` property must point to the folder that exists. Anthos on VMware creates a virtual machine disk (VMDK) to hold the Kubernetes object data for the admin cluster, which is created by the installer for you, so make sure that name is unique.

⁵Sections are populated if `--auto-create-service-accounts` flag is used

TIP: If you would prefer to not use Resource Pool and place admin cluster resources directly under cluster level, provide “<clusterName>/Resources” in resoucrePool configuration.

In the next section we define the IP settings for admin cluster nodes, services and pods. These settings will also be used for the user cluster master nodes deployment.

First, we must define whether the Anthos on VMware admin cluster nodes and user cluster master nodes will use DHCP or static IP assignments. If a static option is used, an additional YAML file must be defined for IP address assignments; this file is specified in the *ipBlockFilePath* property.

The next two properties are dedicated for the Kubernetes service and pod CIDR ranges, which are detailed in Table 1 below. They are used by Kubernetes pods and services and are described in detail in chapter 4 “Anthos, the computing environment based on Kubernetes”. The assigned network ranges must not overlap between each other or with any external services that are consumed by the management plane, for example, any internet proxy used for communication with GCP.

TIP: Due to fact that Anthos on VMWare operates in Island Mode IP addresses used for Pods and Services are not routable into dataceneter network. That means you can use same IPs for every new cluster.

Finally, the last section defines the target vSphere network name that the Kubernetes Nodes will use once provisioned.

Table 9.1 - Admin Cluster Properties

Property key	Property description	
network		
ipMode	Parent key for type and ipBlockFilePath	
	type	IP mode to use ("dhcp" or "static")
	ipBlockFilePath	Path to yaml configuration file used for static IP assignment. Must be used in conjunction with type: static key value pair
serviceCIDR	Kubernetes service CIDR used for control plane deployed services. Minimal size 128 addresses	
podCIDR	Kubernetes pods CIDR used for control plane deployed services. Minimal size 2048 addresses	
vCenter	Parent key for networkName	
	networkName	vSphere portgroup name where admin cluster nodes and user cluster master nodes are assigned to.

An example configuration is shown below.

```
network:
  ipMode:
    type: dhcp
  serviceCIDR: 10.96.232.0/24
  podCIDR: 192.168.0.0/16
  vCenter:
    networkName: "My Anthos on VMware admin network"
```

As we mentioned, the Node IP assignments can be configured via a static configuration file. The path to such a file must be specified under *ipBlockFilePath* key that must be uncommented and taken into account only when *ipMode.type* key is set to static. Additionally DNS and NTP servers must be specified and search domain defined, as shown in the example below.

```

network:
  ipMode:
    type: "static"
    ipBlockFilePath: "myAdminNodeHostConfFile.yaml"
  hostConfig:
    dnsServers:
      - "8.8.8.8"
    ntpServers:
      - "myNTPServer"
    searchDomainsForDNS:
      - "myDomain.local"
  
```

The static host configuration file is built using two main configuration keys: hostconfig and blocks. The Hostconfig defines information about DNS servers, NTP servers and search domains. The blocks define netmask and gateway for Kubernetes nodes followed by an array of hostnames and corresponding IP addresses for them.

Property key	Property description
blocks	
netmask	Network netmask
gateway	Network gateway
ips	Array of <code>ip</code> and <code>hostname</code> keys with corresponding values.

```

blocks:
  - netmask: 255.255.255.128
    gateway: 10.20.0.1
    ips:
      - ip: 10.20.0.11
        hostname: admin-host1
      - ip: 10.20.0.12
        hostname: admin-host2
      - ip: 10.20.0.13
        hostname: admin-host3
      - ip: 10.20.0.14
        hostname: admin-host4
      - ip: 10.20.0.15
        hostname: admin-host5
  
```

TIP: IP addresses assigned to nodes are not assigned in the order defined in the file. They are randomly picked from the pool of available IPs during resizing and upgrade operations.

The next section of the configuration is for the cluster Load Balancing. Anthos on VMware requires a load balancer to provide a Virtual IP (VIP) to the Kubernetes API server. For your clusters you can choose between using the integrated load-balancer, based on MetalLB, using a F5, or any other load-balancer using a manual configuration. MetalLB is becoming a

popular solution for bare-metal based implementations including VMware⁶ outside of Hyperscaller build in solutions. Enablement of MetalLB on the admin cluster is limited to definition of kind: MetalLB in the admin cluster configuration file as presented below.

```
loadBalancer:
  vips:
    controlPlaneVIP: "133.23.22.100"
  kind: MetalLB
```

We will explain the options in greater detail in the Load Balancer section of this chapter.

To make sure that Kubernetes control plane nodes will be distributed across different ESXi hosts, Anthos supports vSphere anti-affinity groups. Such an implementation guarantees that a physical ESXi host failure will only impact a single Kubernetes node or addon node providing a production-grade configurations control plane. This value should be set to true to leverage anti-affinity rules, or false to disable any use of anti-affinity rules.

```
antiAffinityGroups:
  enabled: true/false
```

You can monitor the cluster using Google Cloud Logging by setting the appropriate values in the stackdriver section of the configuration file.

Logs and metrics can be sent to a dedicated GCP project, or the same project where the cluster is being created. You will need to supply the projectID that you want to use for the logs, the cluster location, VPC options, the service account key file with the appropriate permissions to the project, and your decision to enable or disable vSphere metrics.

```
stackdriver:
  projectID: "my-logs-project"
  clusterLocation: "us-central1"
  enableVPC: false
  serviceAccountKeyPath: "my-key-folder/log-mon-key.json"
  disableVsphereResourceMetrics: true
```

Moreover, you can also integrate audit logs from the cluster's API server with Cloud Audit Logs. You must specify the project that integration should target (it can be the same project used for your Cloud Operations integration), the cluster location, and a service account key with the appropriate permissions.

```
cloudAuditLogging:
  projectID: "my-audit-project"
  clusterLocation: "us-central1"
  serviceAccountKeyPath: "my-key-folder/audit-log-key.json"
```

It's important to make sure that problems on Kubernetes nodes are detected and fixed quickly, and similar to a GKE cluster, Anthos on VMware uses a Node Problem Detector. The detector watches for possible node problems and reports them as events and conditions. When any Kubelet becomes unhealthy or ContainerRuntimeUnhealthy conditions are reported by the kubelet or docker systemd service, the auto repair functionality will try to restart them automatically.

⁶<https://metallb.universe.tf/installation/clouds/>

Anthos on VMware clusters auto repair functionality allows to automatically creating Kubernetes node VMs when they are deleted by mistake or recreation of unresponsive faulty Virtual Machines. It can be enabled or disabled in the cluster deployment configuration file, by setting the autoRepair enabled option to either true or false.

```
autoRepair:  
  enabled: true/false
```

When enabled, cluster-health-controller deployment is created on the corresponding cluster in the kube-system namespace. If the node is indicated as unhealthy, it is drained and recreated, as shown in Figure 9.5.

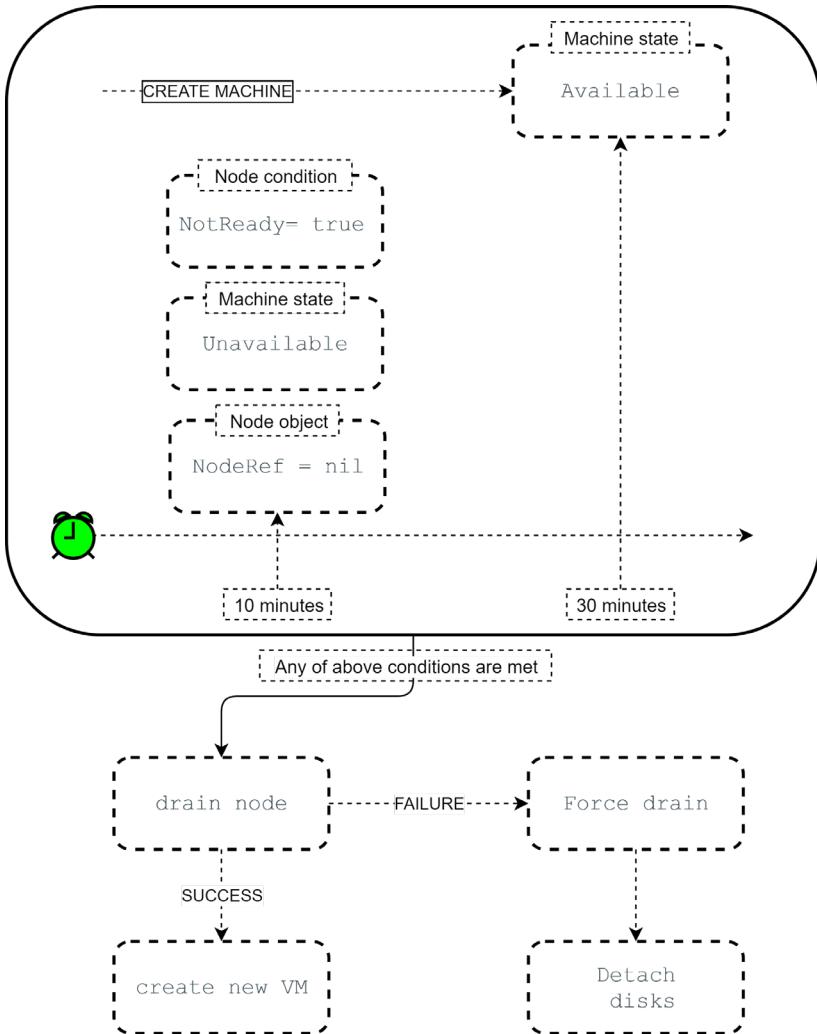


Fig 9.5. Node auto repair process

NOTE: To disable auto repair functionality on admin cluster `cluster-health-controller` deployment must be deleted from admin cluster.

It is possible to deploy Anthos on VMware from a private docker registry instead of gcr.io. To configure your deployment to use a private registry, you need to set the values in the `privateRegistry` section of the configuration file. You will need to supply values for the registry address, the CA for the registry, and the reference to the credentials to use in the credentials file.

```

privateRegistry:
  address: "{{Private_Registry_IP_address}}"
  credentials:
    fileRef:
      path: "{{my-config-folder}}/admin-creds.yaml"
      entry: "private-registry-creds"
  caCertPath: "my-cert-folder/registry-ca.crt"

```

That completes the admin cluster configuration file configuration, now let's move to the user cluster configuration.

Security is very important for Anthos based Kubernetes implementation. Anthos on VMware introduced secret encryption capability to ensure they are encrypted at rest without requirement for external Key Management Service. As a result, before a secret is stored in the etcd database, it is encrypted. To enable or disable that functionality edit the `secretsEncryption` section of the configuration file.

```

secretsEncryption:
  mode: GeneratedKey
  generatedKey:
    keyVersion: 1

```

TIP: Anytime key version is updated a new key is generated and secrets are re-encrypted using that new key.

You can enforce key rotation using `gkectl update` command, as result all existing and new secrets are encrypted in use of new key and old one is securely removed.

USER CLUSTER CREATION

Each new user cluster is required to be connected to an admin cluster, in fact, there is no way to create a workload cluster without an admin cluster. A single admin cluster can manage multiple user clusters but a single user cluster can be supervised by only one admin cluster.

Each provisioned user cluster can be deployed in two configurations, with, or without, a Highly Available (HA) production-grade management plane. As presented in the below drawing, clusters with HA enabled are built with 3 admin nodes (*User Cluster #2 in the drawing*), and without HA, using a single node (*User Cluster #1 in the drawing*). The Single node management plane consumes less compute resources but in case of a node or physical host failure, the ability to manage the cluster is lost⁷. In HA mode, a single master node failure does not impact the Kubernetes cluster configuration management ability.

⁷ vSphere High Availability feature can mitigate that behavior and decrease downtime of Kubernetes API to minutes until VM is restarted on new host.
vSphere HA will not protect against Virtual Machine corruption.

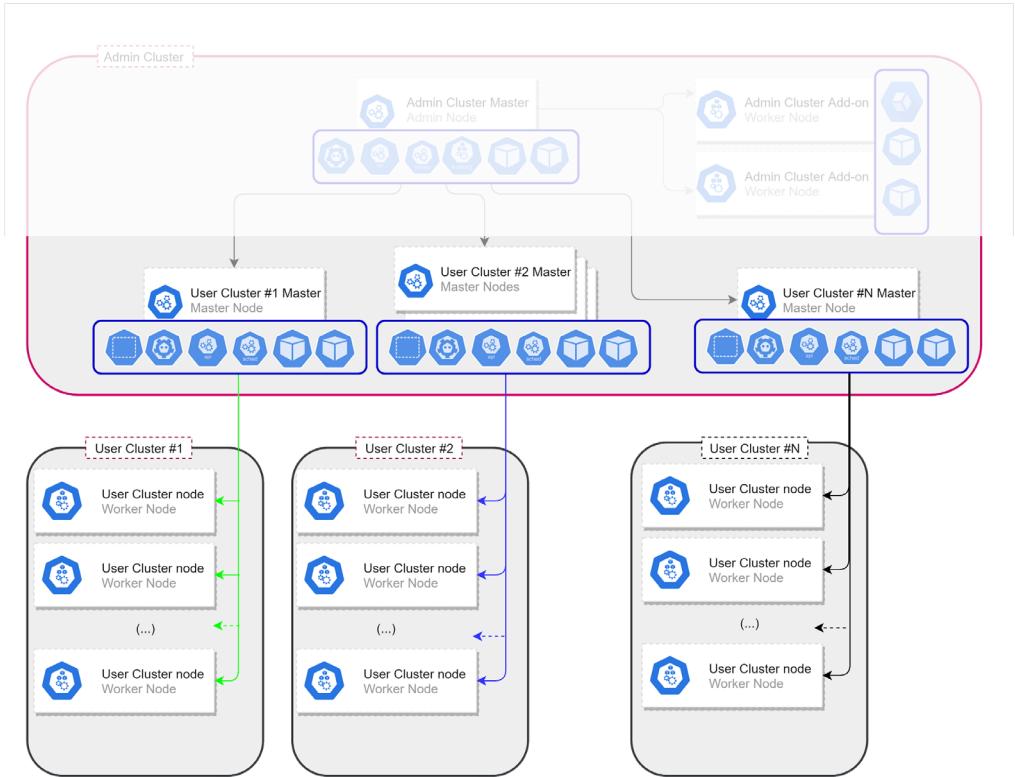


Figure 9.6. Anthos on VMware user clusters architecture

IMPORTANT: After deployment, the number of admin nodes *cannot* be changed without full cluster re-creation.

Every new user cluster is placed in a dedicated namespace in the admin cluster. It's used to host and deliver services, deployments, pods and ReplicaSets for management purposes.

The Namespace name is inline with the cluster name, allowing you to easily grab all details by referring to it via `kubectl get all -n {{ clusterName }}`. Any user cluster namespace will be hosted on dedicated nodes that are added to the admin cluster when you create the user cluster. The nodes will be labeled with the cluster name and when the cluster management pods are created, they will use a node selector to force their placement on the dedicated user cluster nodes.

Other, non-system namespaces, are created on top of workload cluster nodes as presented on below picture

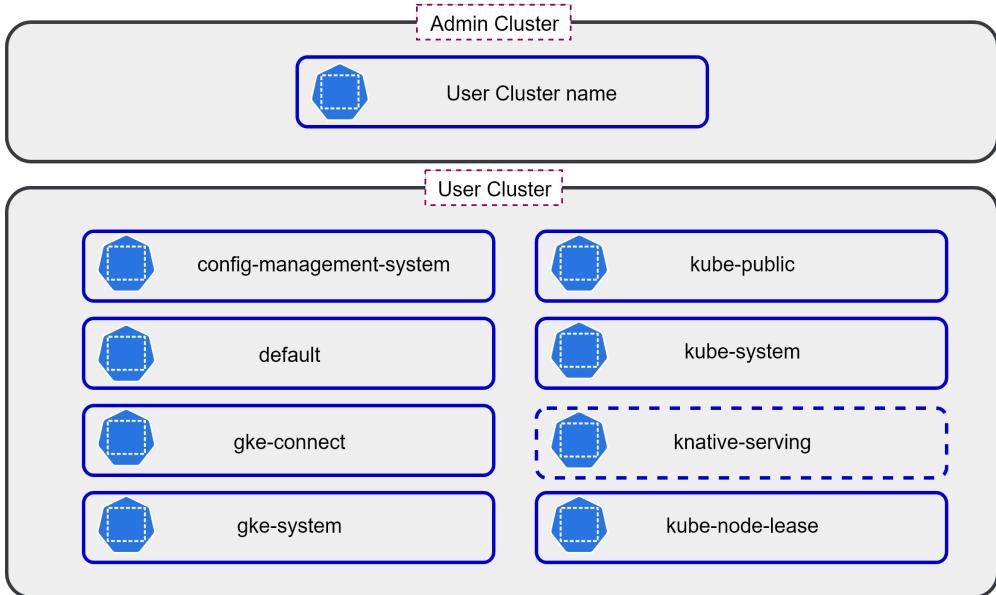


Figure 9.7. Anthos on VMware user cluster namespaces

Similar to an admin cluster, a user cluster deployment is based on a YAML configuration file. The first two sections of the configuration file must remain unchanged, defining the API and cluster type.

```
apiVersion: v1
kind: UserCluster
```

TIP: You can convert old versions of configuration files using :

```
gkectl create-config cluster --config $MyAwesomeClusterConfigFile.yaml --from
MyOldConfigFile.yaml --version v1
```

TIP: You can convert old versions of configuration files using :

```
gkectl create-config cluster --config $MyAwesomeClusterConfigFile.yaml --from
MyOldConfigFile.yaml --version v1
```

The next section is where you provide the name of the new cluster and the Anthos on VMware version. The cluster name must be unique within a GCP project and the version must be inline with the admin cluster version.

```
name: "MyAwesomeOnPremCluster"
gkeOnPremVersion: 1.10.0-gke.194
```

The next section is optional. It is used to manage vSphere integration and worker nodes placement. It is strongly recommended to separate admin and workload compute resources on vSphere level to ensure Kubernetes management plane availability. This guarantees resources for each Anthos on VMware cluster in case of resource saturation or limited access to vSphere. Following that practise your user cluster worker nodes will be placed in resource pool and datastore⁸ defined under the vCenter section. Additionally user clusters can be deployed into separate VMware datacenters if required. To make sure rights separation is properly applied for vSphere resources it's recommended to use dedicated account for user cluster vCenter communication.

```
vCenter: "MyAwsomeOnPremCluster"
  datacenter: "MyWorkloadDatacenter"
  resourcePool: "GKE-on-prem-User-workers"
  datastore: "DatastoreName"
  credentials:
    fileRef:
      path: "path to credentials file"
      Entry: "Name of entry in credentials file referring to username and password of
vCenter user account"
```

TIP: You can decide to use only a single property like vCenter.resourcePool. In such case comment other lines adding # at the beginning of line and configuration of commented property will be inherited from admin cluster configuration.

The networking section has the same structure as the admin nodes described in the admin cluster section extended with capability to define additional network interfaces that can be used for Kubernetes payload.

```
network:
  ipMode:
    type: dhcp
  serviceCIDR: 10.96.232.0/24
  podCIDR: 192.168.0.0/16
  vCenter:
    networkName: "My Anthos on VMware user network"
    additionalNodeInterfaces:
      - networkName: "My additional network"
        type: dhcp
```

Or in case of static IP assignment:

```
network:
  ipMode:
    type: "static"
    ipBlockFilePath: "myNodeHostConfFile.yaml"
  additionalNodeInterfaces:
    - networkName: "My additional network"
      type: "static"
      ipBlockFilePath: "mySecondNodeHostConfFile.yaml"
```

⁸ For vSAN based deployments all nodes must be placed on the same datastore.

At the beginning of that chapter we stated that the admin plane can be HA protected or not. Such a decision is configured under the `masterNode.replicas` section of the cluster configuration file by definition adequately of 3 or 1 replicas.

We can also scale up the cpu and memory of master nodes if required under this section.

```
masterNode:
  cpus: 4
  memoryMB: 8192
  replicas: 3
```

TIP: Configuration file is key: value based. All values defined under quotation marks "" are interpreted as strings and without quotation marks as integers. All number based configuration elements like amount of replicas, cpus or memory must be specified as integers

User cluster worker nodes are defined as pools of nodes. This allows you to have different sizes of nodes in the same Kubernetes cluster with labels and taints applied to node objects. Finally, the last configuration element of each defined node pool is the node Operating System, offering either Google's hardened Ubuntu image or Google's immutable Container-Optimized OS (COS). If using bundled loadbalancer type - MetalLB - at least one of the pools must have `enableLoadBalancer` configuration set as true.

```
nodePools:
- name: "My-1st-node-pool"
  cpus: 4
  memoryMB: 8192
  replicas: 3
  bootDiskSizeGB: 40
  labels:
    environment: "production"
    tier: "cache"
  taints:
  - key: "staging"
    value: "true"
    effect: "NoSchedule"
  vsphere:
    datastore: "my-datastore"
    tags:
    - category: "purpose"
      name: "testing"
  osImageType: "cos"
  enableLoadBalancer: false
```

NOTE: Worker node virtual machines will be named inline with defined node pool name followed by random numbers and letters i.e. `My-1st-node-pool-sxA7hs7`.

During the cluster creation, anti-affinity groups are created on vSphere with the worker nodes placed inside them. This vSphere functionality allows us to distribute the worker node VMs across different vSphere hosts in a cluster, avoiding placing too many nodes on the same physical host. As a result, in the case of a VMware ESXi host failure, only a limited number of Kubernetes nodes are impacted, decreasing impact into hosted services.

To enable antiAffinityGroups, it's mandatory to have at least 3 ESXi hosts in the vSphere cluster. You can enable or disable this feature under the configuration file `antiAffinityGroups.enabled` section by changing the default value to either true or false.

```
antiAffinityGroups:
  enabled: true
```

By default all workload clusters can be accessed in use of auto generated kubeconfig files. In such cases no additional configuration is required but access scope is not limited and significantly hard to manage. To solve this problem, Anthos on VMware clusters have an option to integrate into external identity providers via OpenID Connect (OIDC) and grant access to namespaces or clusters using Kubernetes authorization (Figure 9.8).

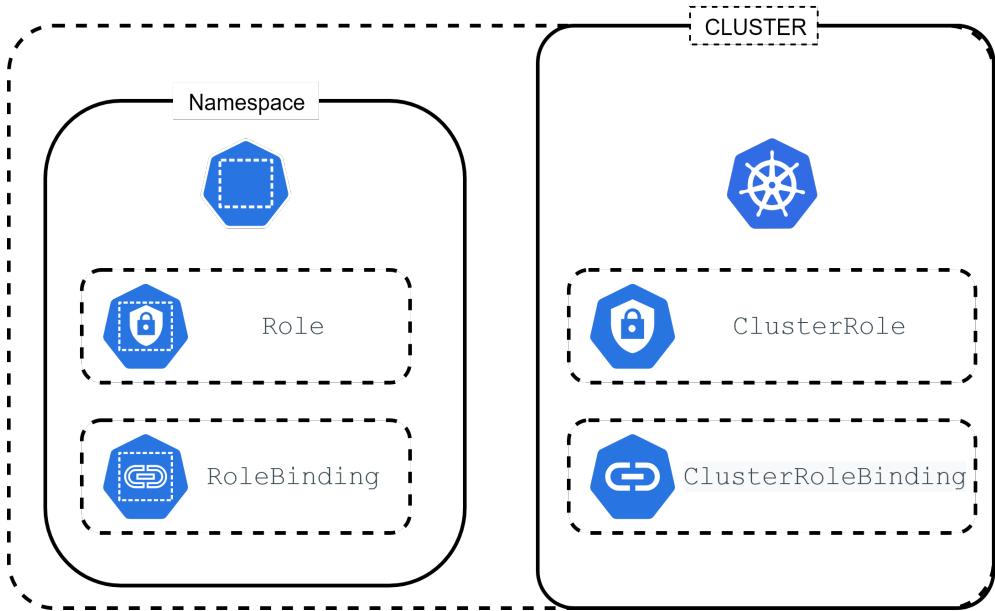


Figure 9.8. Kubernetes Cluster and namespace based access

During the user cluster deployment, you can integrate the cluster into an existing Active Directory Federation Services (ADFS), Google, Okta or any other certified OpenID provider⁹. To configure these settings, we must provide all provider specific information under the `authentication.oidc` section of the configuration file. Below is an example of using Google's OIDC integration.

INFO: Google OpenID provider returned token does not contain information about the group that user belongs to. As a result due to that limitation individual users must be provided instead.

⁹ Anthos on VMware supports all certified OpenID providers. Full list can be found under <https://openid.net/certification/>

```

authentication:
  oidc:
    issuerURL: "https://accounts.google.com"
    kubectlRedirectURL: "http://localhost:2222/callback"
    clientID: "r4nd0md4t4.apps.googleusercontent.com"
    clientSecret: "s3cr3t"
    username: "email"
    usernamePrefix: "-"
    group: ""
    groupPrefix: ""
    scopes: "email"
    extraParams: "prompt=consent,access_type=offline"
    deployCloudConsoleProxy: "false"

```

TIP: To restore user cluster kubeconfig you can trigger:

```
kubectl --kubeconfig $ADMIN_CLUSTER_KUBECONFIG get secrets -n $USER_CLUSTER_NAME
admin -o jsonpath='{.data.admin\conf}' | base64 -d > $USER_CLUSTER_NAME-kubeconfig
```

The next option, similar to admin cluster, is the option to enable or disable auto repair functionality on configuration file level

```

autoRepair:
  enabled: true/false

```

The key difference from the admin cluster configuration is that it can be easily changed by applying changes into the YAML configuration file and triggering a `gkectl update` command.

The last part to be covered before we move on to networking is storage. By default Anthos on VMware include the vSphere Kubernetes volume plugin that allows dynamically provisioning vSphere VMDK disks on top of datastores¹⁰ attached to vCenter clusters. After a new user cluster is created, it is configured with a default storage class that points to the vSphere datastore. Besides the volume connector, newly deployed clusters automatically get the vSphere Container Storage Interface (CSI). The CSI is a standard API which allows you to connect directly to compatible storage, bypassing vSphere storage. It's worth mentioning that Anthos on VMware clusters still support the use of in-tree vSphere Cloud Provider volume plugin that enables a direct connection to storage, bypassing vSphere storage as well. However, due to known limitations, like lack of dynamic provisioning support, it's not recommended to use the in-tree plugin - you should use the CSI driver instead.

We managed to define compute and storage components that are used for Anthos on VMware deployment. Let's summarize it. Our build is based on the admin workstation, admin cluster and user clusters deployed and hosted on a vSphere environment. The picture below presents resource separation based on resource pools dedicated for every user cluster and combined admin cluster resources with user master nodes.

¹⁰ vSphere datastores can be backed by any block device, vSAN or NFS storage

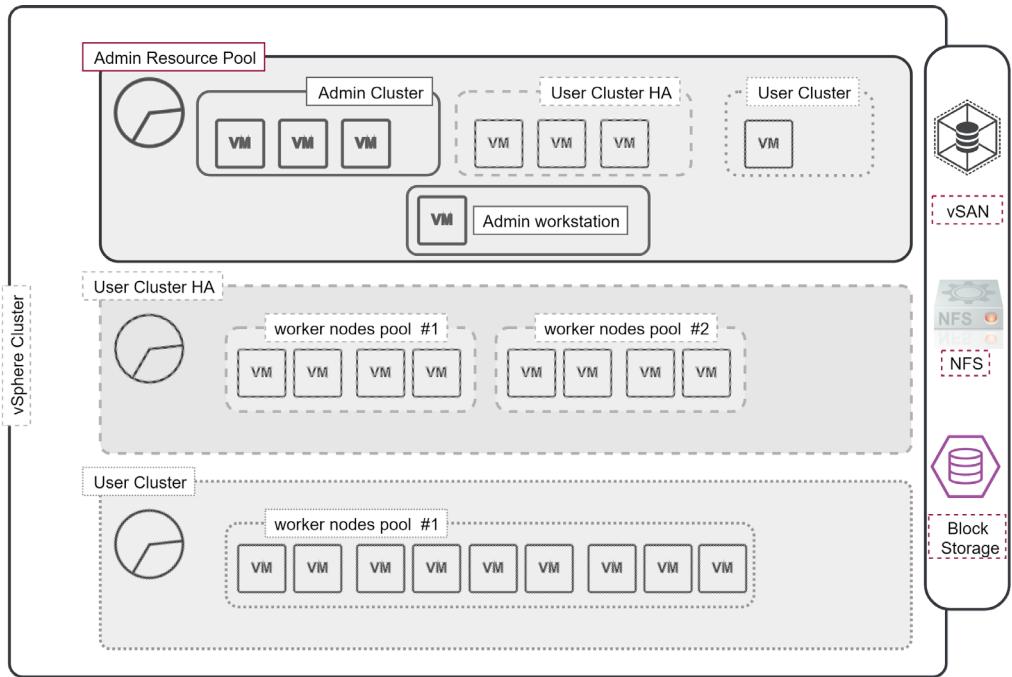


Figure 9.9. Anthos on VMware resource distribution

You have learned a lot about the compute part of Anthos on VMware implementation. In the next section, we will go through details for communication capabilities, requirements and limitations depending on network made implementation choice.

9.2.1 Anthos Networking

To understand the role that networking plays in an Anthos cluster, we need to understand that an Anthos cluster consists of two, different, networking models. The first is the vSphere network where the entire infrastructure is placed and the other is Kubernetes networking.

At the beginning of this chapter we stated that Anthos on VMware does not require any Software Defined Networking applied on top of vSphere infrastructure and can be fully VLAN based.

IP MANAGEMENT

As we already familiarized ourselves with deployment flow, let's go deeper in configuration and architecture elements.

Besides the configuration files mentioned in previous chapters, there can be additional required depending on the deployment scenario. When DHCP is used, all nodes have assigned IP addresses from it as presented on Figure 9.10.

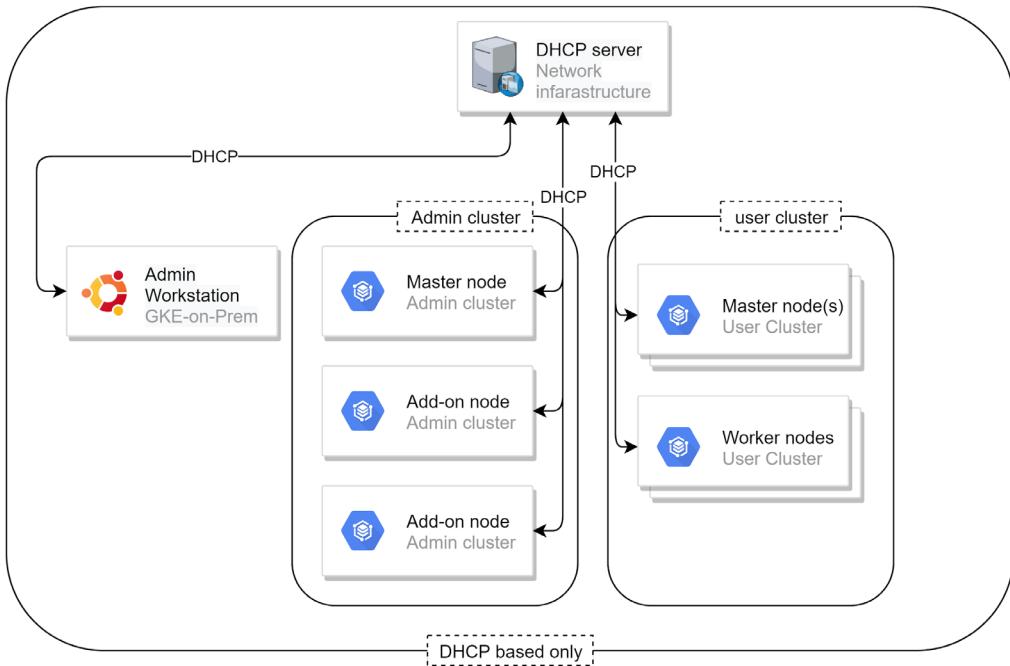


Figure 9.10. DHCP based deployment

If deployment does not utilize DHCP services for node IP allocation, additional host configuration files must be created for the admin cluster and each user cluster (Figure 9.11). Some organizations consider statically assigned addresses to be the most stable implementation since it removes any DHCP problems or lease expiration for nodes will not introduce any disturbance for node communication. However, while it may eliminate any DHCP concerns, it introduces management overhead for host configuration files preparation and scalability limitations of created clusters. The best implementation is something that you must decide for your cluster and organization.

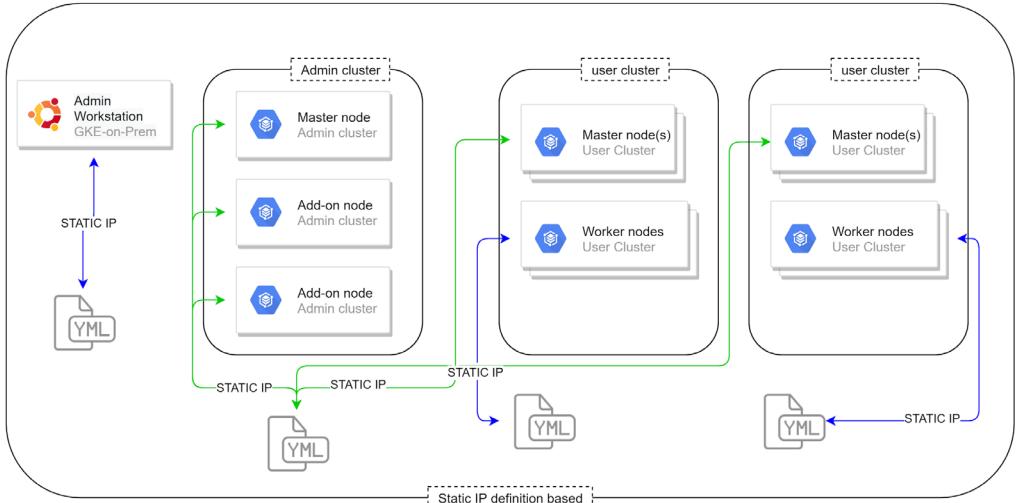


Figure 9.11. Static IP assignment scenario

It's possible to follow a mixed deployment scenario where both DHCP based and non DHCP based clusters are deployed as presented in the picture below.

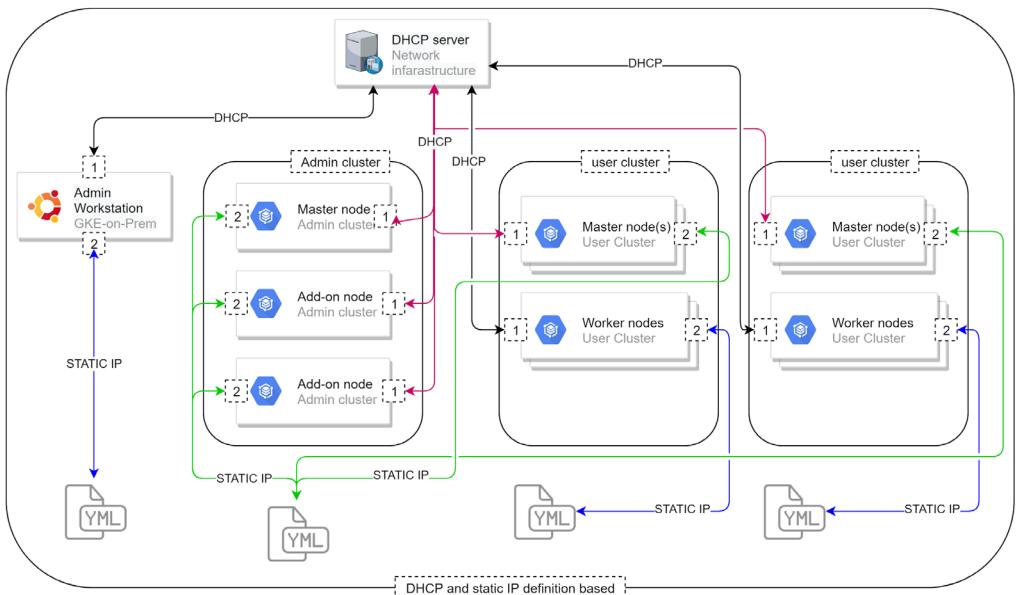


Figure 9.12. Mixed DHCP and Static IP assignment scenario

With this configuration, we can have an Admin cluster using static IP addresses for its management plane and user Kubernetes master nodes, DHCP for 1st user cluster and static IP addresses for 2nd user cluster Kubernetes worker nodes or the complete opposite. As IP address change of Kubernetes nodes introduces significant problems for storage access, it's recommended to use static IP assignment for admin cluster nodes.

There are a few constraints related to mixed deployment:

- IP assignment must be identical for the entire admin cluster and user cluster master nodes as they share the same IP address pool
- All user cluster worker node pools must use the same IP assignment method for the entire cluster
- Separate user clusters can use different IP assignment methods even if they are managed by the same admin cluster

So far, we have talked about IP assigned options and arguments for and against each of the implementations. Now let's talk about detailed network implementation configuration, good practices and recommendations for both management and workload.

MANAGEMENT PLANE

Looking deeper into the management plane network configuration there are two elements we need to communicate with, depending on activity that is intended to be performed. The first element we must deploy for Anthos on VMware is the admin workstation, which is fully preconfigured and hardened by Google.

The Second communication point is the entire admin cluster, hosting all admin nodes and user cluster master nodes. Both mandate communication with VMware infrastructure to automatically deploy Virtual Machines. There is no technical requirement to separate admin workstation, nodes and vSphere infrastructure but from a security perspective it is highly recommended to isolate those networks on layer 2 as presented in the picture below.

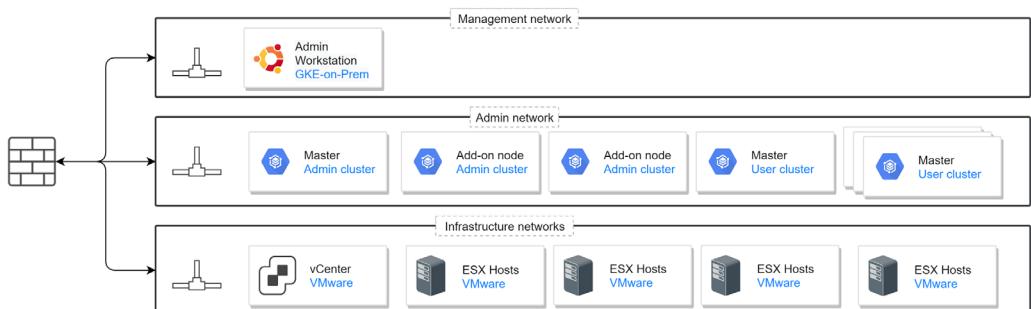


Figure 9.13. Anthos on VMware vSphere networking

As Anthos on VMware clusters are integrated into GCP console they mandate communication with the external world. This connection can be achieved using a direct internet connection or via an internet proxy.

The default networking model for a new Anthos cluster is known as island mode. This means that Pods are allowed to talk to each other but prevented by default from being reached from external networks. Another important note is that outgoing traffic from Pods to services located outside are NATed by node IP addresses.

The same applies to services. They can overlap between clusters but must not overlap with Pod subnet (Figure 9.14). Additionally Pods and Services subnets must not overlap with external services consumed by cluster i.e. internet proxy or NTP¹¹ otherwise traffic will not be routed outside the created cluster.

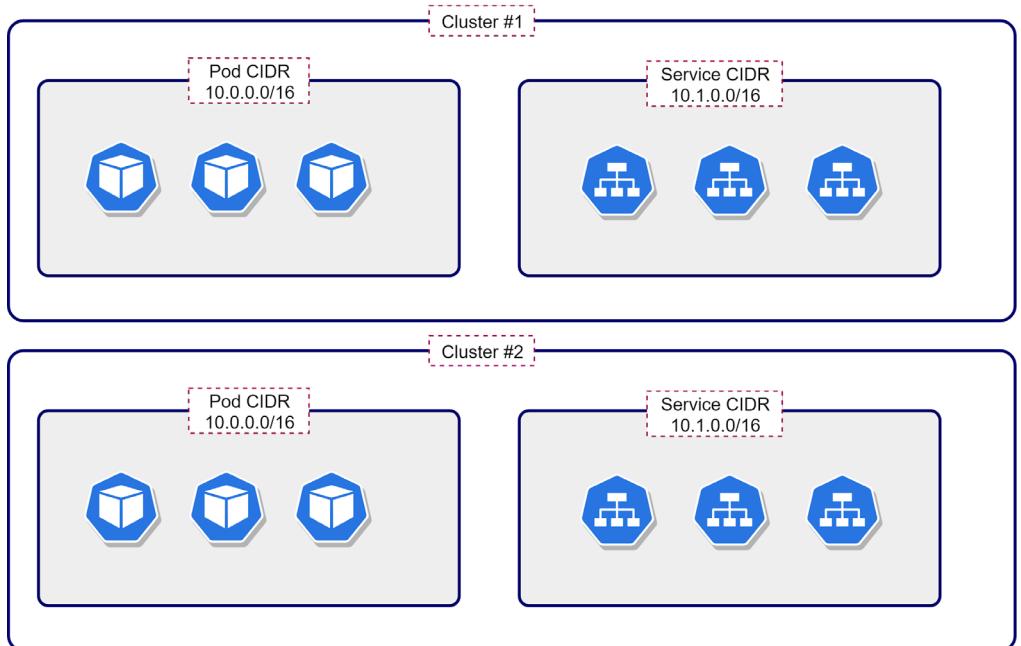


Figure 9.14. Pods and services

Both service CIDR and Pod CIDR are defined in the admin cluster configuration YAML file, and built-in preflight checks make sure that IP addresses for both do not overlap.

```
network:
  serviceCIDR: 10.96.0.0/16
  podCIDR: 10.97.0.0/16
```

¹¹ Network Time Protocol

LOAD BALANCERS

To manage a Kubernetes cluster you must reach it's Kubernetes API server. Admin cluster exposes it via LoadBalancer IP that can be configured in 3 flavours depending on type. At the time of writing this chapter, Anthos on VMware supports the following types: MetalLB, F5BigIp and ManualLB.

BUNDLED - METALLB

MetalLB is Google developed open source¹² Cloud Native Computing Foundation sandbox project network load-balancer implementation for Kubernetes clusters. It's running on bare metal implementations allowing use of LoadBalancer services inside any cluster.

MetalLB addresses two requirements that are part of hyperscaler Kubernetes implementations but lacking in on-premises ones, external announcement and address allocation. Address allocation provides you capability to automatically assign IP address to LoadBalancer service that is created without a need to specify it manually. Moreover you can create multiple IP address pools that can be used in parallel depending on your needs, for example pool for private IP addresses that are used to expose services internally and pool of IP addresses that provides external access. As soon as an IP address is assigned it must be announced on the network, and their external announcement feature is coming into play. MetalLB can be deployed in two modes: layer 2 mode and BGP mode.

Current implementation of Anthos on VMware is using layer 2 mode only. In layer 2 implementation external announcement is managed in use of standard address discovery protocols: ARP for IPv4 and NDP for IPv6. Each Kubernetes service is presented as dedicated MetalLB load balancer, as result when multiple services are created traffic is distributed across load balancer nodes. Such implementation has advantages and constraints. Key constraint is related to the fact that all traffic for service IP is always going to one node where kube-proxy spreads it to all service pods. As a result service bandwidth is always limited to single node network bandwidth. In case of node failure service is automatically failed over. Such a process should take no longer than 10 seconds. When looking at advantages for MetalLB layer 2 implementation for sure we must mention the fact that it's fully in cluster implementation without any special requirements from physical networks in the area of hardware etc. Layer 2 implementations do not introduce any limitation to the amount of load balancers created per network as soon as there are IP addresses available to be assigned. That is a consequence of using memberlist Go library to maintain cluster membership list and member failure detection using gossip based protocol instead of for example Virtual Router redundancy Protocol¹³.

¹² <https://github.com/metallb/metallb>

¹³ <https://tools.ietf.org/html/rfc3768>

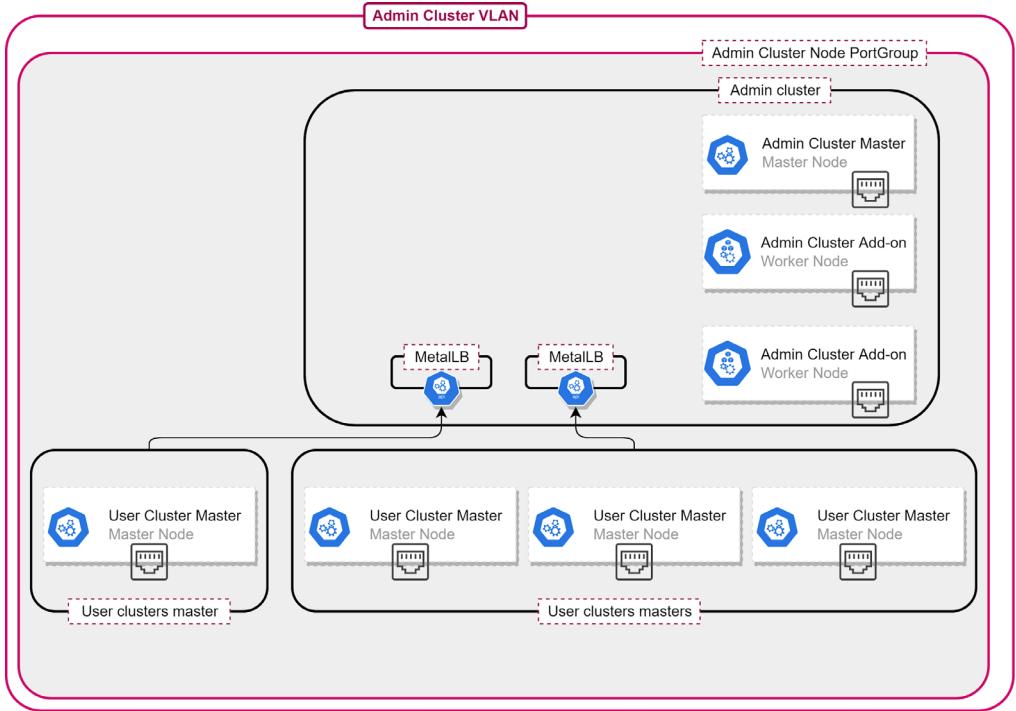


Figure 9.15. Admin cluster networking with metalLB

INFO: Admin Cluster Kubernetes VIP is not leveraging MetalLB as the admin cluster does not HA implement. All User Clusters are using MetalLB deployment for its Kubernetes VIP exposure.

MetalLB is part of Anthos on VMware covered under Anthos license and standard support inline with the chosen support model, including lifecycle management activities for each release.

INTEGRATED - F5

The second option to introduce load-balancer capabilities is integration with F5 BIG-IP load balancer - called integrated load balancer mode. Compared to Seesaw, F5 infrastructure must be prepared upfront and is not deployed automatically by Google. For Anthos on VMware, the BIG-IP provides external access and L3/4 load-balancing services. When integrated load balancer mode is defined, Anthos on VMware automatically performs preflight checks and installs single supported version of F5 BIG-IP container ingress services (CIS) controller.

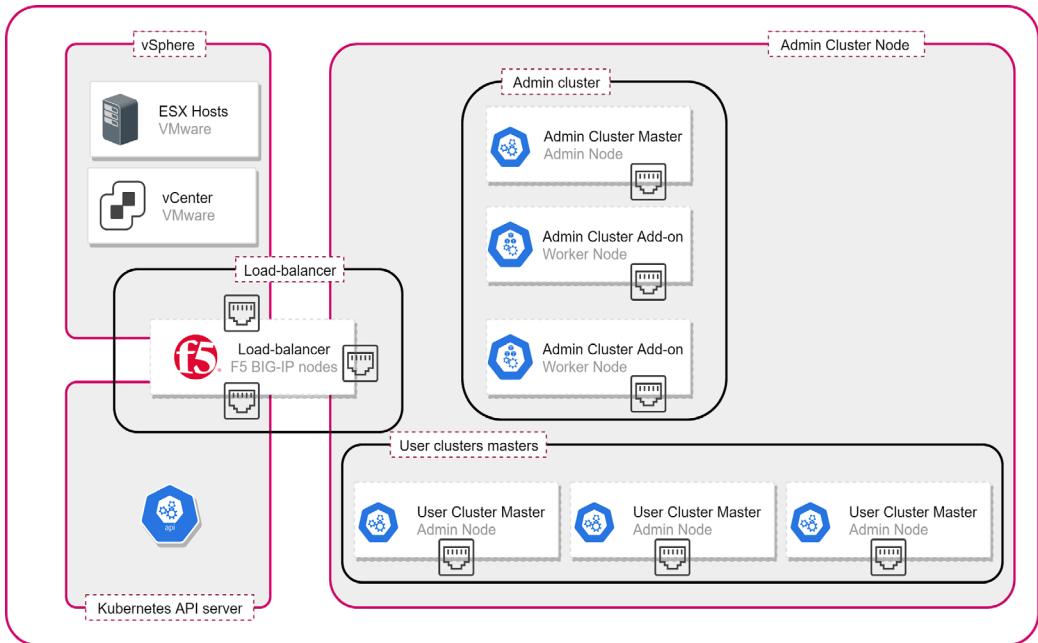


Figure 9.16. Admin Cluster networking with F5 BIG-IP

Production license provides up to 40 Gbps throughput for Anthos on VMware load balancer.

BIG-IP integration is fully supported by Google inline with the support compatibility matrix but licensed separately under F5 licensing.

MANUAL LOAD BALANCER

To allow flexibility and capability to use existing load balancing infrastructure of your choice, Anthos on VMware can be deployed in use of manual mode configuration of load balancer. In such an implementation there is a need to set up a load balancer with Kubernetes API VIP before cluster deployment starts. Configuration steps depend on the load balancer you are using. Google provides detailed documentation describing BIG-IP and Citrix configuration steps. It's important that in manual mode you cannot expose Services of type LoadBalancer to external clients.

Due to lack of automated integration with manual load balancing mode, Google does not provide support and any encounter issues with the load balancer must be managed with the load balancer's vendor.

We learned already about all three mode types. Let's have a look into configuration files. Configuration is covered under the dedicated loadBalancer section of the admin config yaml file and will vary depending on chosen option.

For MetaLB configuration in the admin cluster we must define the loadbalancer kind as MetaLB and provide Kubernetes API service VIP.

```
loadBalancer:
  vips:
    controlPlaneVIP: "Kubernetes API service VIP"
  kind: MetalLB
```

When F5 BIG-IP integrated mode is chosen, the load balancer section must be changed to kind: F5BigIP. Entire seesaw section (enabled by default on new generated config file) must be commented and f5BigIp section must be defined with credentials file and partition details.

```
loadBalancer:
  vips:
    controlPlaneVIP: "Kubernetes API service VIP"
  kind: F5BigIP
  f5BigIP:
    address: "loadbalancer-ip-or-fqdn"
    credentials:
      fileRef:
        path: "name-of-credential-file.yaml"
        entry: "name of entry section in above defined file"
    partition: "partition name"
    snatPoolName: "pool-name-if-SNAT-is-used"
```

Last use case is covering manual load balancing. In such a scenario we must define kind: ManualLB and comment out the seesaw section. Next we must manually define the NodePort configuration options.

```
loadBalancer:
  vips:
    controlPlaneVIP: "Kubernetes API service VIP"
  kind: ManualLB
  manualLB:
    controlPlaneNodePort: "NodePort-number-for-control-plane-service"
    addonsNodePort: "NodePort-number-for-addon-service"
```

USER CLUSTERS

User cluster networking is based on the same principles as for admin clusters, extended for workload deployment capabilities. Every cluster is deployed in Island Mode, where services and Pods CIDRs must not overlap in the user cluster configuration file.

```
network:
  serviceCIDR: 10.96.0.0/16
  podCIDR: 10.97.0.0/16
```

Again we have three modes of load-balancer deployment and integration: bundled, integrated and manual.

Bundled deployment similarly uses MetalLB implementation. You already learned that master nodes of the user cluster are deployed into the admin cluster node network and IPs are assigned from predefined static pool or DHCP servers. Kubernetes API service is also exposed automatically in the same network but its IP address must be defined manually in the user cluster configuration file.

User cluster worker nodes can be deployed into the same network as admin nodes or into separate dedicated network. The second option is preferred as it allows the separation of traffic from the management plane. Each new user cluster comes with a new dedicated

MetalLB load-balancer for the dataplane. Control plane VIP for Kubernetes API is always co-hosted in the admin cluster load balancer instance. User Cluster ingress VIP is automatically deployed into the cluster worker nodes pool network. As a result you can limit MetalLB to be hosted on a dedicated node pool instead of all nodes in the user cluster.

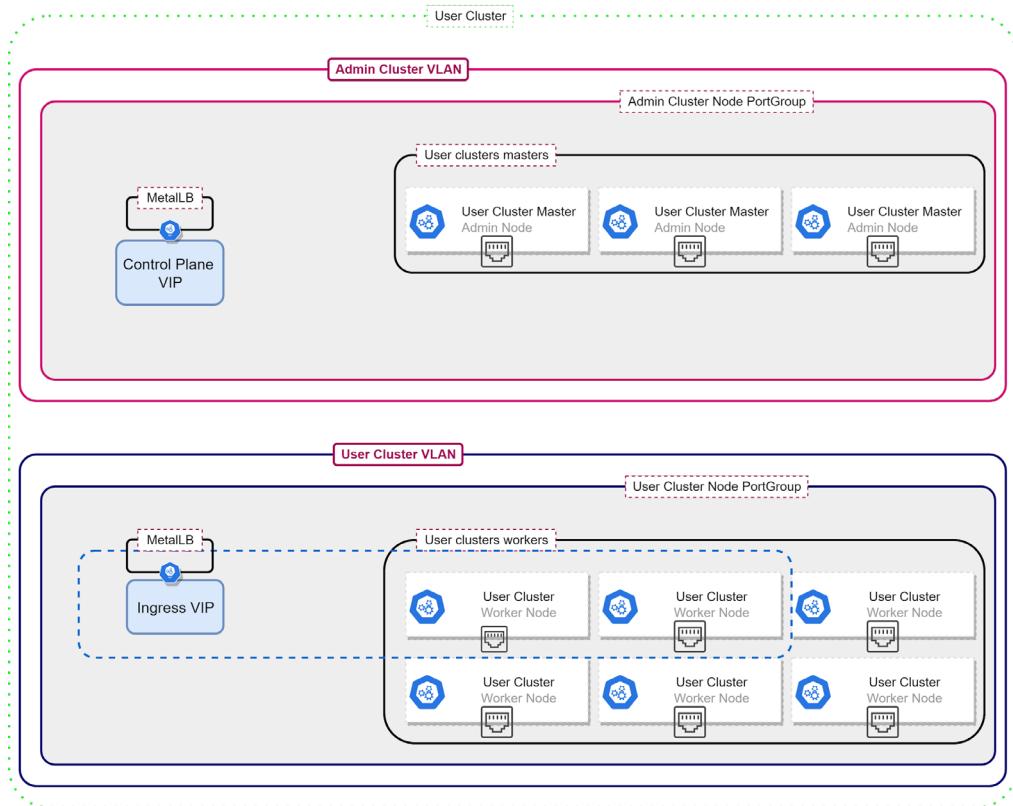


Figure 9.17. User Cluster networking with MetalLB

In multi-cluster deployment user clusters can share a single network or use a dedicated one. When using a single one, make sure that node IPs are not overlapping in configuration files.

As we already mentioned, MetalLB is a fully bundled load-balancer. That means every time when service type LoadBalancer is created VIP is automatically created on load-balancer and traffic sent to the VIP is forwarded to Service. MetalLB has IP address management (IPAM) as a result IP addresses for each service are assigned automatically. Definition of IP pools and nodes that host MetalLB VIPs is defined in user-cluster load balancer configuration file section as presented below

```

loadBalancer:
  vips:
    controlPlaneVIP: "Kubernetes API service VIP"
  kind: MetalLB
  metalLB:
    addressPools:
      - name: "name of address pool"
        addresses:
          - "address in form of X.X.X.X/subnet or range X.X.X.X-X.X.Y.Y"
# (Optional) Avoid using IPs ending in .0 or .255.
    avoidBuggyIPs: false
# (Optional) Prevent IP addresses to be automatically assigned from this pool (default:
#   false)
    manualAssign: false
  
```

Additionally we must allow MetalLB service on one (or more) Virtual Machine pools in worker pools section described already in details earlier in the book by defining configuration in form “enableLoadBalancer: true”

Integrated load balancing allows integration into F5 BIG-IP and provision LoadBalancer service type automatically similar to the way we described for admin cluster integration. In that scenario, the integration point can be the same for admin and user clusters and there is no need to use separate instances of F5s. It’s important to remember that every new cluster must be pre-configured and properly prepared on the load balancer side before deployment.

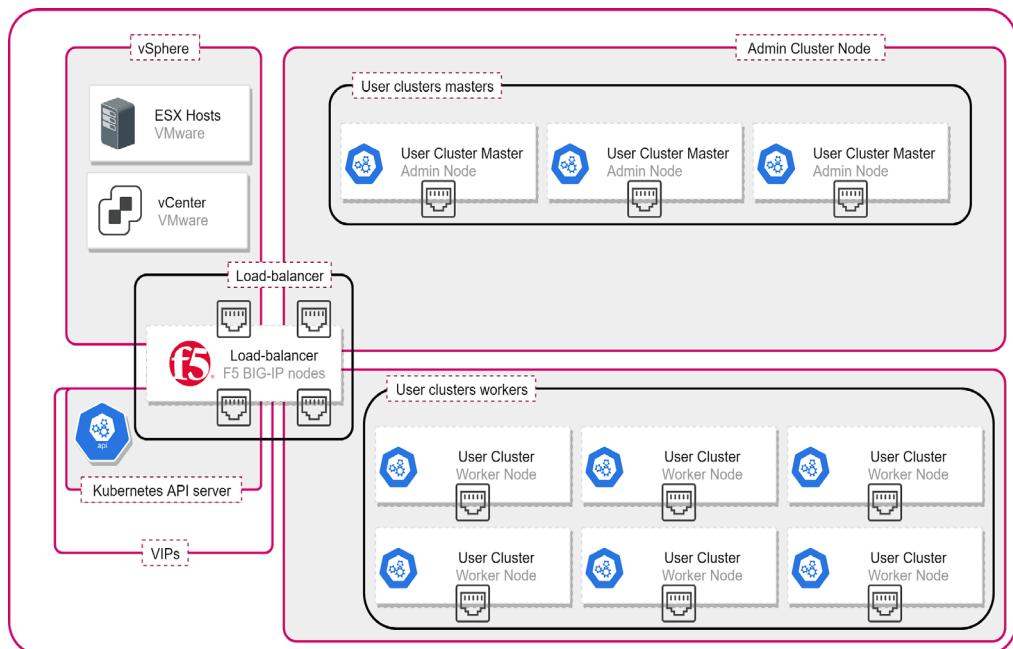


Figure 9.18. User Cluster networking with F5 BIG-IP

Manual mode implementations for user cluster load balancers are following the same rules, introducing the same constraints and limitations as for admin clusters. Every service exposure mandates contact with external teams and manual activities as described in official documentation¹⁴.

In this section of Anthos on VMware, we went through different network configuration options for management and workload clusters. It's important to remember that similarly to IP assignment, we can decide to use a single load-balancer integration mode or choose different modes for different clusters. As a result we can have a Seesaw based management cluster, one Seesaw user cluster, second F5 BIG-IP integrated user cluster and third manually integrated Citrix Netscaler load balancer.

```
loadBalancer:
  vips:
    controlPlaneVIP: "Kubernetes API service VIP"
    ingressVIP: "Ingress service VIP (must be in use node network range)"
  kind: MetalLB
  MetalLB:
    addressPools:
      - name: "my-address-pool-1"
        addresses:
          - "192.0.2.0/26"
          - "192.0.2.64-192.0.2.72"
    avoidBuggyIPs: true
```

```
loadBalancer:
  vips:
    controlPlaneVIP: "Kubernetes API service VIP"
  kind: F5BigIP
  f5BigIP:
    address: "loadbalancer-ip-or-fqdn"
    credentials:
      fileRef:
        path: "name-of-credential-file.yaml"
        entry: "name of entry section in above defined file"
    partition: "partition name"
    snatPoolName: "pool-name-if-SNAT-is-used"
```

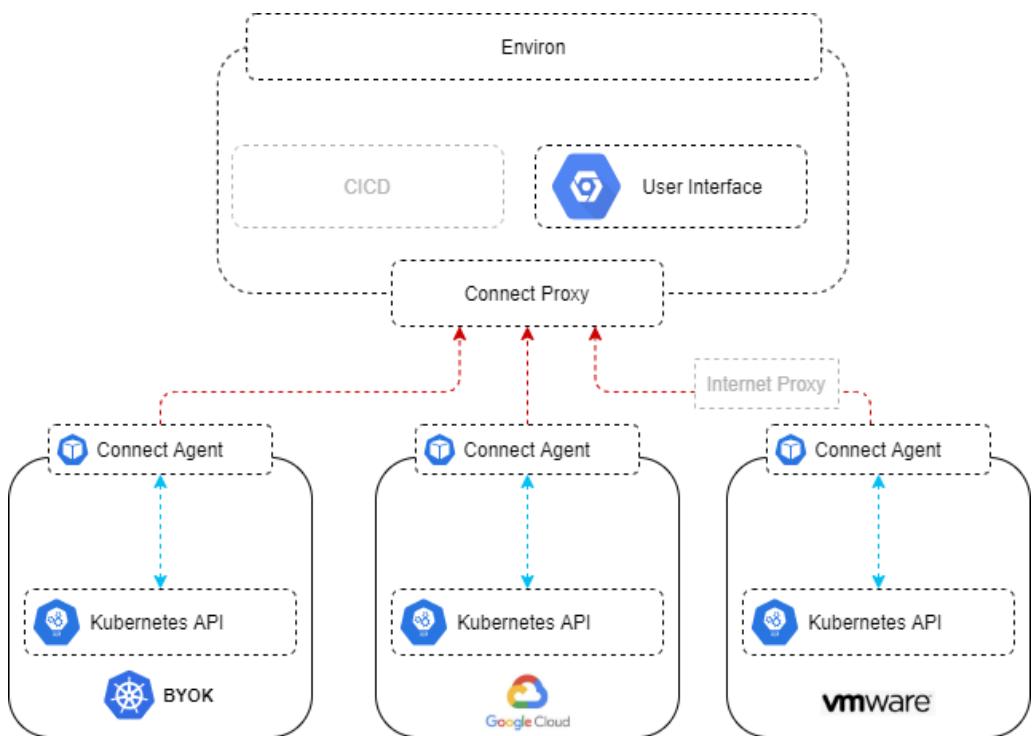
```
loadBalancer:
  vips:
    controlPlaneVIP: "Kubernetes API service VIP"
  kind: ManuallB
  manualLB:
    ingressHTTPNodePort: Ingress-port-number-for-http
    ingressHTTPSNodePort: Ingress-port-number-for-https
    controlPlaneNodePort: "NodePort-number-for-control-plane-service"
```

¹⁴ <https://cloud.google.com/anthos/clusters/docs/on-prem/how-to/manual-load-balance>

9.2.2 GCP integration capabilities

In previous sections we talked about compute and network architecture of Anthos on VMware. In that section we will cover different integration capabilities for GCP services to leverage a single panel of glass for all Anthos clusters regardless if they are deployed on premise or on other clouds .

As you already noticed during deployment of new admin and user clusters it's mandatory to define a GCP project that it will be integrated into. That enables the connect agent to properly register and establish communication with the hub as described in detail in Chapter 6 - Operations Management in Anthos. In general GKE Connect is performing two activities, enabling connectivity and authentication to register new clusters.



For that purpose two dedicated service accounts are used. That section is mandatory and must be properly defined for the user cluster. Note that due to the fact that the agent service account is using Workload Identity functionality it does not require a key file.

```
gkeConnect:
  projectId: "My-awesome-project"
  registerServiceAccountKeyPath: register-key.json
```

NOTE: GKE connect plays a significant role in on prem integration into GCP. That allows us to utilize directly from GCP console Cloud Marketplace, Cloud Run, options to integrate with CI/CD toolchain via Anthos authorization without a need to expose Kubernetes API externally.

Anthos on VMware has the ability to send infrastructure logs and metrics to GCP Cloud Monitoring. It applies for both admin and user clusters. We can choose to send only Kubernetes related metrics or include vSphere metrics as well. Metrics for each cluster can be sent to different projects.

```
stackdriver:
  projectID: "My-awesome-project"
  clusterLocation: gcp-region-name
  enableVPC: false/true
  serviceAccountKeyPath: monitoring-key.json
  disableVsphereResourceMetrics: true/false
```

Another integration feature that is applied for both admin and user clusters is the capability to send Kubernetes API server audit logs. As previously described, we can choose the project and location region where logs are stored and the service account that is used for that integration.

```
cloudAuditLogging:
  projectID: "My-awesome-project"
  clusterLocation: gcp-region-name
  serviceAccountKeyPath: audit-key.json
```

Last two integration features are applied to user clusters only. First of them is an option to consume from Google Cloud Platform console CloudRun for Anthos (described in detail in “Anthos, the serverless compute engine (Knative)” chapter) and deploy services directly into Anthos on VMware clusters. There is not much configuration there as the service itself is leveraging connect functionality and deploying Knative in a dedicated namespace of the user cluster. That means it must be enabled on the same project as Anthos on which the VMware cluster is registered. You will learn more about CloudRun for Anthos and Knative on Anthos Cloud Run chapter.

```
cloudRun:
  enabled: true/false
```

Feature list description is closed by metering. After enabling the metering feature, the user cluster sends resource usage and consumption data to Google Bigquery. It allows us to analyze it and size our clusters inline with actual demand or expose them and present them as reports for example in Data Studio.

```
usageMetering:
  bigQueryProjectID: "My-awesome-project"
  bigQueryDatasetID: dataset-name
  bigQueryServiceAccountKeyPath: metering-key.json
  enableConsumptionMetering: false/true
```

9.3 Summary

In this chapter we talked that Anthos on VMware is a great option to consume cloud native capabilities on top of existing vSphere infrastructure and learned the following:

- Architecture is composed from two elements:
 - user clusters that are responsible for delivery of resources for hosted applications
 - admin control plane that is responsible for management and control of deployed user clusters
- It can be used as an add-on to co-hosted Virtual Machines as well as dedicated on-prem Kubernetes implementations that are ready for hybrid cloud implementation of cloud native journey.
- We can leverage existing VMware skills for infrastructure management, keep full visibility of Cloud Operations and consume GCP services if needed from your own data center.
- Setup can be independent and self-contained in use of bundled features like MetallB or integrated into existing infrastructure and what automation capabilities and constraints it brings
- Cluster configurations may vary depending on purpose, size and availability requirements.

In next chapter we are going to learn how ANthos fits into telco world.

11

Knative serverless extension

by Konrad Clapa

This chapter covers:

- Introduction to serverless
- Knative Serving and Eventing Components
- Knative on Anthos

Before we get into the details let us set up the scene. What we are going to talk about in this chapter is Google Cloud Platform's managed service based on an open source project called Knative. The project was started to allow for higher development velocity of Kubernetes applications without need to understand the complex Kubernetes concepts it uses. With this service, Google installs and manages Knative serving inside your Anthos GKE cluster. One of the benefits of using Knative with Anthos instead of open source Knative is that Google's automation and Site Reliability Engineers all handle installation and maintenance. Anthos integrates with numerous GCP services like Cloud Load Balancing¹, Cloud Armor², Cloud CDN³ and many others making an enterprise-ready Knative a reality.

11.1 What is the problem we are trying to solve

We have already discussed Kubernetes in Chapter 4, *Anthos, the computing environment based on Kubernetes* so we have a feeling how complex the installation and maintenance of it can be. This problem is being solved for us with Google Kubernetes Engine. On top of Kubernetes we still need to know how to run and operate cloud native applications. What

¹ Cloud Load Balancing: <https://cloud.google.com/load-balancing>

² Cloud Armor: <https://cloud.google.com/armor>

³ Cloud CDN (Content Delivery Network): <https://cloud.google.com/cdn/docs/overview>

Knative does abstract those implementation details and allow you to serve your serverless container based workloads on any Kubernetes cluster.

In this chapter we will look at what serverless is, introduce you to Knative and finally discuss how Anthos delivers an enterprise grade container based serverless platform.

11.2 Introduction to Serverless

There is a lot of discussion about what serverless is. The most common thing to see is comparisons between serverless and Function as a Service (FaaS). This is almost an ideological dispute. To keep this simple let's have a look at the Cloud Native Computing Foundation definition:

Serverless computing refers to a new model of cloud native computing, enabled by architectures that do not require server management to build and run applications.

Google's fully managed Cloud Run service perfectly fits into this definition as it abstracts the compute layer from the developer and operator. It allows you to deploy the containers that will be serving http(s) requests. The scaling of the application is handled by the platform itself.

While Google Cloud Functions deliver similar capabilities, it is more opinionated about the runtime languages you can use. With Cloud Run you can use any language that can run a service able to answer HTTPS calls. Cloud Run does not require Anthos to be used.

When we think about Cloud Run we can think about the following set of serverless features:

- No server - developers don't need to worry about underlying compute infrastructure
- Multi-Language - the application can be written in any language
- Event-driven - the container/function is triggered by an external event
- Autoscaling - the container can automatically scale based on requests.
- Portability - your container/application should be able to run on any Kubernetes platform.

11.3 Knative

As you already know Kubernetes is a platform for building platforms. Then why not use it for building serverless platforms based on containers? Knative runs on top of Kubernetes like any other Kubernetes application. You can even see some statements from the Knative contributors that it should not be called serveless so, let's have a look at Knative as a platform to deliver serverless anywhere where you run Kubernetes. Sounds fair?

11.3.1 Introduction

Say you would like to build your own Kubernetes-based serverless platform you might come up with the following diagram showing all the required components. Clearly there is some duplication of effort related to build of the primitives like autoscalablity, observability, rollouts and many others. What Knative is doing is providing all these primitives for you so there is a common experience of running serverless workload on Kubernetes.

Kubernetes Serverless stack architecture

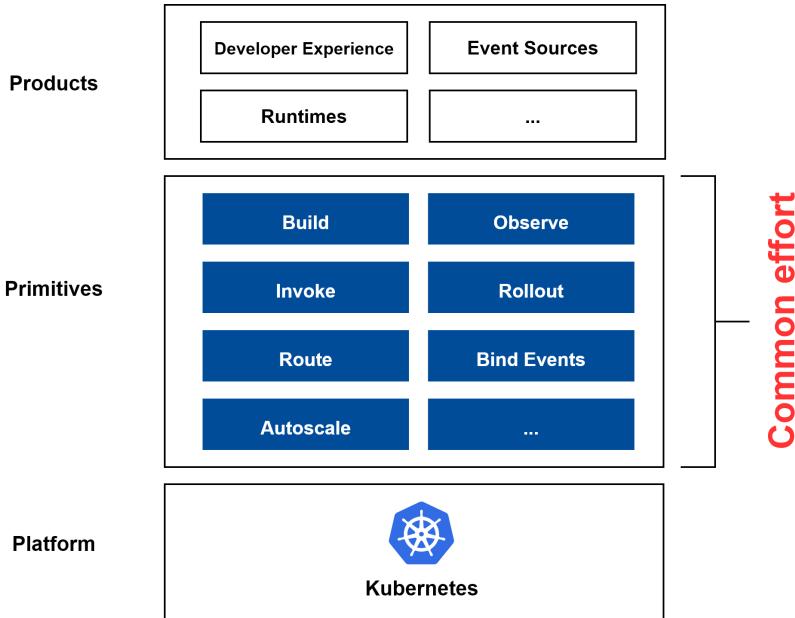


Figure 11.1. Kubernetes Serverless stack architecture

Now think of it from the developers perspective. All they have to do is to define the dependencies, write their code and put it in a container. Then they deploy the application to Knative. The details of how it is served is not of their concern. This however does not mean that they lose the capability to finetune the service. There are multiple parameters that can be set like concurrency (how many requests can be served per container), min/max instances (minimum/maximum container instances that can be provisioned for the Knative Service) and many others. Knative hides away all the complexities of Kubernetes involved with scaling and traffic management and provides a means to observe the workloads. That is what you call an easy start with development of Kubernetes applications, right?

KNATIVE VS CaaS, FaaS AND PaaS

In the second section of this chapter we learned what problems Knative is trying to solve: Enable serverless workloads to run anywhere with the flexibility of Kubernetes but hiding the complexity away. The below table presents a comparison of Knative against platform as a service (PaaS)⁴, container as a service (CaaS)⁵ and function as a service (FaaS⁶). With

⁴ PaaS: https://en.wikipedia.org/wiki/Platform_as_a_service

⁵ CaaS: <https://www.ibm.com/services/cloud/containers-as-a-service>

⁶ FaaS: https://en.wikipedia.org/wiki/Function_as_a_service

various features being supported or not. Do it yourself (DIY) means you need to put some development to be able to use that feature.

Table 11.1. Knative vs PaaS, CaaS and FaaS

Feature	Knative	PaaS	CaaS	FaaS
Simple UX/DX	Yes	Yes		Yes
Event-driven	Yes	Yes		Yes
Container based	Yes	Yes	Yes	Yes
Autoscaling	Yes	Yes	DIY	Yes
Scale resources to 0	Yes	Yes		Yes
Load Balancing	Yes	Yes	DIY	Yes
Unrestricted execution time	Yes*	Yes	Yes	
Unrestricted compute/memory limits	Yes**	Yes**	Yes**	
Variety of programming languages support	Yes	Yes	Yes	Limited

*might be restricted for Managed services like Cloud Run.

** depends on the platform

As we can see, the Knative gives you all the advantages of function as a service but also gives you the ability to run your application in almost any language. The limits on the execution time are much higher compared to FaaS. In many cases where there is a need for longer request processing time Knative is a solution. You finally have access to advanced features like volumes and networking so you can tweak your workload if needed. Like with all compute services with higher flexibility the responsibility demarcation line shifts more toward you. You need to build your own container and make sure you are able to make use of all the advanced features but let's be honest, who does not like to be in more control of your application until you get all the benefits of FaaS?

11.3.2 Knative History

Google started Knative but now has multiple companies contributing to it like IBM, RedHat, Pivotal, SAP and many others. The full documentation and source code can be found here: <https://github.com/knative>. Knative started as a set of components that allows the build and

run of stateless workloads together with subscription to events. There are currently two active projects in progress on Github those are:

- **Knative Serving** - allows to serve serverless containerised workloads
- **Knative Eventing** - allows subscription to external events.

The third project **Knative Build** that helped building containers was deprecated and turned into Tekton Pipelines⁷ project. As you will learn from the Chapter 14 *Anthos, integrations with CI/CD* it was used by Google to build Cloud Build for Anthos.

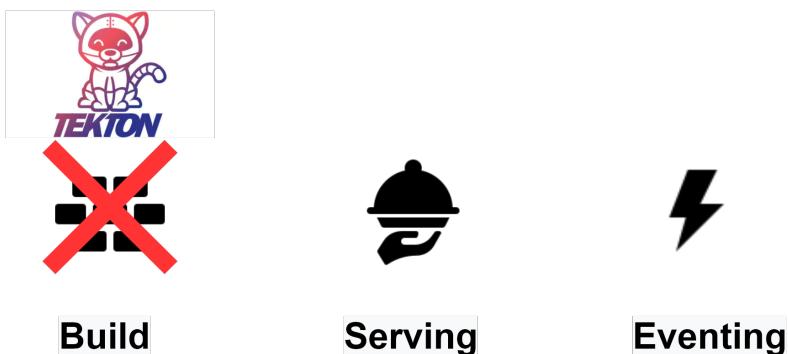


Figure 11.2. Knative components

At the time of writing this book both Knative Serving and Eventing are already in version 1.x. With Eventing being slightly behind Serving. Knative was developed with the vision to deliver the App Engine⁸ simplicity but allowing for flexibility that Kubernetes brings in. As an example with Knative you can modify the routing to different versions of the application by setting the traffic configuration on a Knative Service⁹ object rather than changing the low level network objects configuration (e.g Istio). This resembles App Engine where you simply run one command to perform this task and can be used for canary deployments and A/B testing. Knative gives you the ability to run your serverless containers anywhere, being it a cloud or on premise data center.

As you will shortly learn there are already multiple Knative-based fully managed services existing that makes it even easier to use without getting into the complexity of Kubernetes. The most interesting one for this book is of course Knative for Anthos which is one of the most advanced offering existing on the market.

⁷ Tekton Pipelines: <https://github.com/tektoncd/pipeline>

⁸ App Engine: <https://cloud.google.com/appengine>

⁹ Knative Service is explained in the next section of this chapter.

11.3.3 Knative Architecture

Let's have a look at Knative architecture presented in the below diagram. As you can already notice there are multiple layers with some of the components being plug and play or optional.

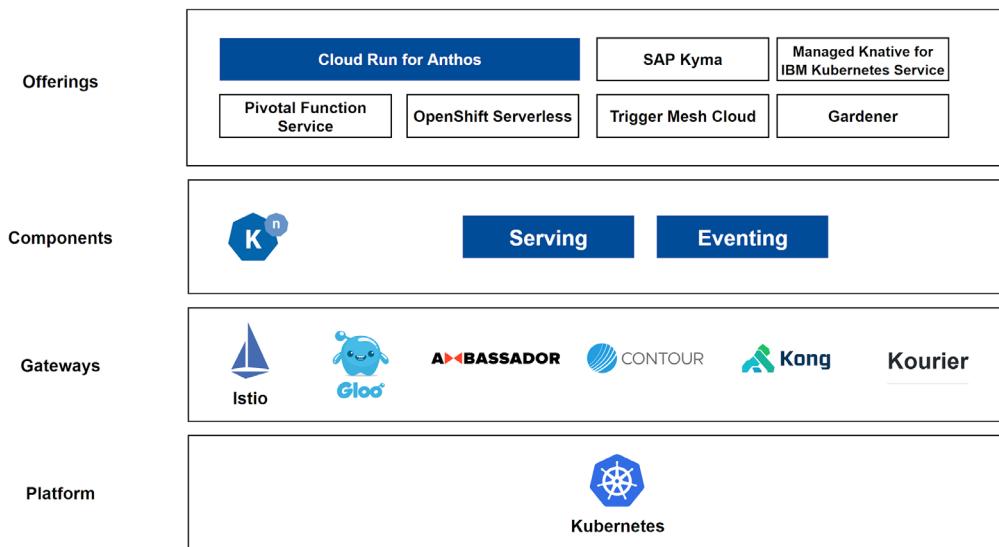


Figure 11.3. Knative Architecture

Knative can run on any compute platform that has the ability to run Kubernetes. It can be either based on virtual machines or bare-metal servers. For traffic routing, Service Mesh Gateway is used. Obviously the most popular is Istio but alternative solutions are also supported like: Gloo, Ambassador, Contour and Kourier with more to come. To learn more about Istio refer to Chapter 5 Anthos Service Mesh: security and observability at scale. On top of that we have Knative components installed as a Kubernetes application. Note that each of those components can be installed and operated separately. If you are not interested in managing the Knative installation yourself there are now multiple managed services existing already: Google Cloud Run and Google Cloud Run for Anthos, Openshift Serverless, Managed Knative for IBM Cloud Kubernetes Service etc. where both Knative and underlying Kubernetes are managed by the provider. The list of those services can be found here <https://knative.dev/docs/install/knative-offerings/>.

11.3.4 Knative Kubernetes Resources Types

Knative comes with a set of controllers and customer resource definitions (CRDs) that extend the native Kubernetes API. Therefore the integration with Knative is very much similar to interaction with Kubernetes API itself. We will look at the Knative resources in the next section.

If you think of a simple Kubernetes application you should have objects like Pods, Deployments and Services. If you include Service Mesh in the picture you will have additional resources to handle the traffic management like *VirtualServices*, *DestinationRules* etc. With Knative you control your deployment with a single resource which is Knative Service that allows you both to deploy the workload and handle the traffic. All the required Kubernetes and Service Mesh resources are created for you.

11.3.5 Knative Serving

Knative Serving allows you to easily deploy container based serverless workloads and serve them to the users via HTTP(s) requests (with gRPC recently announced). As an example you can serve the entire e-commerce website frontend using Knative Serving. It automatically scales your workload as per demand (from 0 to N) and routes or splits traffic to the version (revision) you choose. To achieve this with native Kubernetes you would need to use additional Kubernetes resources like Horizontal Pod Autoscaler HPA¹⁰. Knative Serving extends the Kubernetes API with new Custom Resource Definitions (CRDs) like Knative Serving Service, Configuration, Route and Revision. The below diagram shows how the Knative resources depend on each other.

¹⁰ Horizontal Pod Autoscaler (HPA): <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>

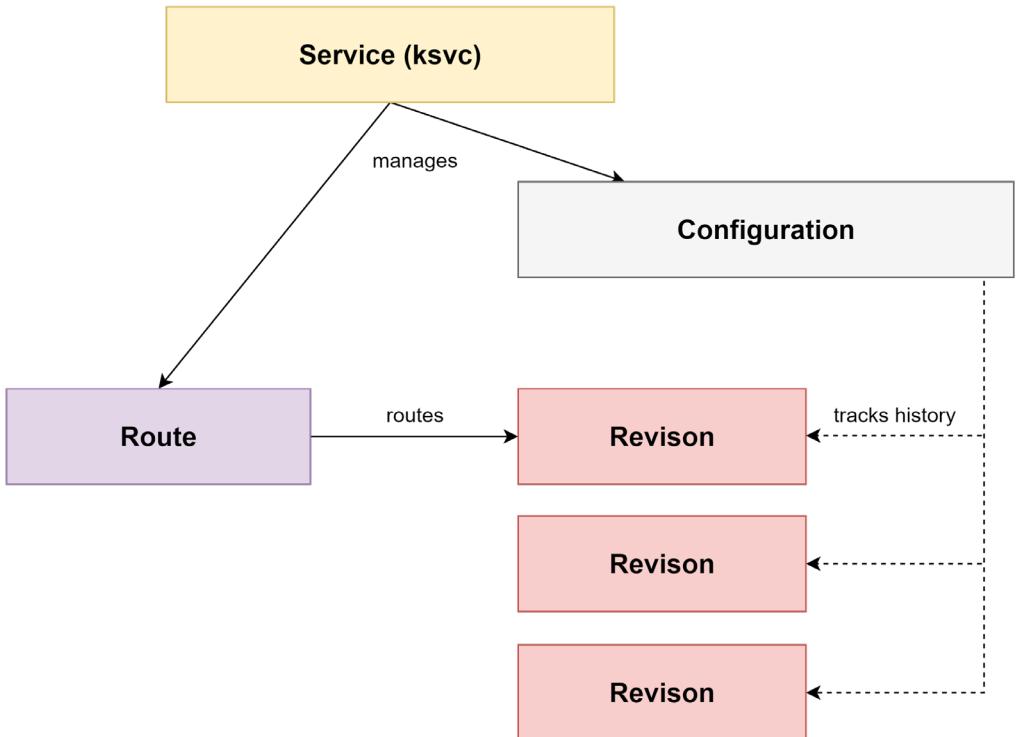


Figure11.4. Knative Serving resources Let's

have a look at each of the CRDs:

- **Service** (API path `service.serving.knative.dev`): This is the most important resource in Knative serving. It automatically creates other Knative resources that are needed for the entire lifecycle of your workload. With the update of the services a new Revision is created. Within the Knative Service you define both the container version as well traffic rules.

NOTE: This is different from the native Kubernetes Service object, which might be confusing to new users at first.

Below we can see an example of Service definition that deploys a simple hello world workload to Knative Serving. This will automatically create other resources: Revision, Configuration and Route.

Listing 11.1

```
apiVersion: serving.knative.dev/v1
kind: Service
```

```

metadata:
  name: helloworld
  namespace: default
spec:
  template:
    spec:
      containers:
        - image: docker.io/{username}/helloworld
          env:
            - name: TARGET
              value: "Python Sample v1"

```

- **Revision** (API path `revision.serving.knative.dev`): This is essentially an immutable snapshot of the container version and its configuration. It defines what is actually served to the user..
- **Configuration** (API path `configuration.serving.knative.dev`): This is the configuration part of your application that enforces the desired state of your workload. It allows you to separate your code (container) from the configuration piece. Modification of the configuration results in new revision creation.
- **Route** (API path `route.serving.knative.dev`): This maps the endpoints to one or more revisions.

When you use Knative you no longer need to worry about the native Kubernetes and Service Mesh resources like: *Deployments*, *Services*, *VirtualServices* etc. You define your application as a Knative Service and all the “backend” resources are created for you.

TRAFFIC MANAGEMENT

When you want to update your Knative Service with a new image, a new revision is created and by default the traffic is directed to the new revision. You can perform A/B¹¹ testing and Canary¹² release and by controlling how much traffic should be directed to particular revision by defining the **`metadata.spec.traffic`** attribute. You can also just tag a particular revision to be accessible by a dedicated URL.

¹¹ A/B testing: <https://martinfowler.com/articles/feature-toggles.html>

¹² Canary release: <https://martinfowler.com/bliki/CanaryRelease.html>

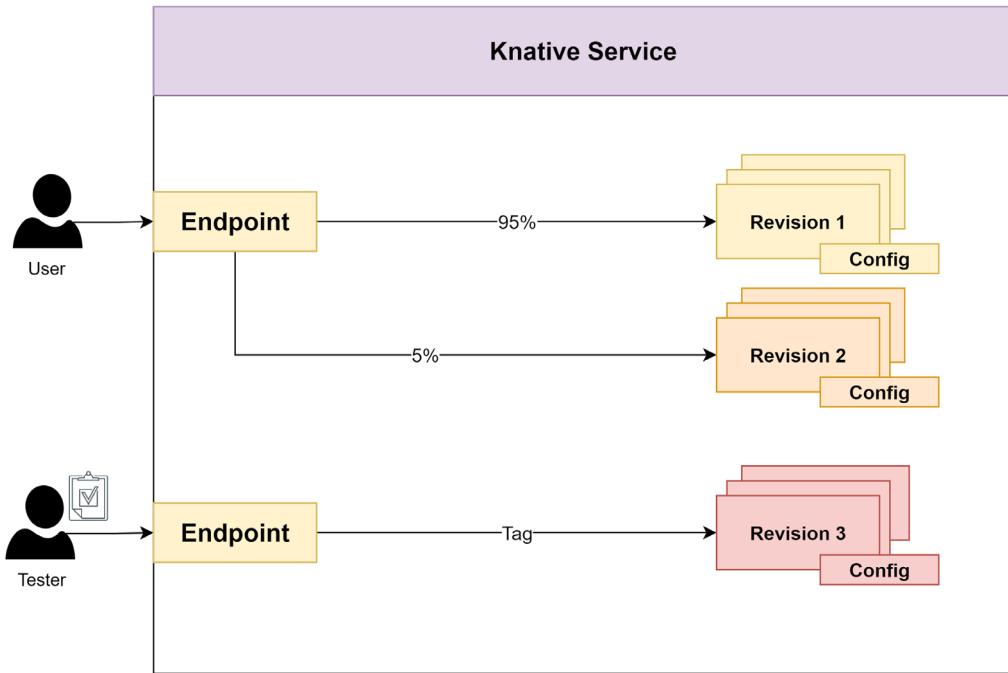


Figure 11.5. Knative Service traffic flows

To achieve the routing as per the picture above you would set up the `metadata.spec.traffic` attribute in the Knative Service as bellow:

```
traffic:
  - tag: current
    revisionName: helloworld-v1
    percent: 95%
  - tag: candidate
    revisionName: helloworld-v2
    percent: 5%
  - tag: latest
    latestRevision: true
    percent: 0
```

As you can see with a single Kubernetes object, a developer can control how the entire workload is served. There is no need to deepdive into Kubernetes backend.

11.3.6 Knative Serving Control Plane

Now let's have a look at the Knative Serving control plane that allows for all this magic to happen. As we already said Kubernetes leverages Istio or any other supported service mesh for traffic management. It also comes with a number of services that take care of running and scaling the workload.

To retrieve the list of the services in the knative-serving namespace you installed with Knative, use:

```
kubectl get services -n knative-serving
```

it will then show the following services:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
activator-service	ClusterIP	10.96.61.11	<none>	80/TCP,81/TCP,9090/TCP	1h
autoscaler	ClusterIP	10.104.217.223	<none>	8080/TCP,9090/TCP	1h
controller	ClusterIP	10.101.39.220	<none>	9090/TCP	1h
webhook	ClusterIP	10.107.144.50	<none>	443/TCP	1h

As you can see for such a complex service, there are not that many supporting services let's have a look at it one by one.

- **Activator** - receives and buffers requests for requests for inactive revisions and reporting metrics to the autoscaler. It also retries requests to a revision after the autoscaler scales the revision based on the reported metrics.
- **Autoscaler** - sets the number of pods required to handle the load based on the defined parameters.
- **Controller** - monitors and reconciles Knative objects defined in CRDs. When a new Knative Service is created it creates Configuration and Route. It will create a revision and corresponding Deployment and Knative Pod Autoscalers (KPAs).
- **Webhook** - intercepts, validates and mutates Kubernetes API calls including CRD insertions and updates. Sets default value and rejects inconsistent and invalid objects.

If you retrieve the list of the deployments in the namespace you installed Knative to

```
kubectl get deployments -n knative-serving
```

you it will see the following deployments:

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
activator	1	1	1	1	1h
autoscaler	1	1	1	1	1h
controller	1	1	1	1	1h
networking-certmanager	1	1	1	1	1h
networking-istio	1	1	1	1	1h
webhook	1	1	1	1	1h

We have already discussed four of these services but there are two deployments that we have not seen yet. This is what they are used for:

- **Networking-certmanager:** reconciles cluster ingresses into cert manager objects.
- **Networking-istio:** reconciles a cluster's ingress into an Istio virtual service.

11.3.7 Knative Eventing

Eventing is the Knative component that orchestrates events originating from various sources inside or outside of the Kubernetes cluster. This very important element of event-driven architecture allows you to trigger your service with a choice among existing event sources and also build new ones for scenarios when you have a custom source not already available.

This is very different from Function as a service where the functions are triggered only using HTTP requests or other predefined triggers like Google Cloud Storage events.

All the Knative Eventing objects are defined as CRDs. It makes sure that the events are handled as defined in the Knative objects with use of controllers. Scalability is taken care of automatically as events trigger calls to your container. It gives you scalability similar to Knative Serving, so you start with a small load of few events and can scale to handle a stream of events.

There are around 20 predefined sources for that you can use already and the list is growing. You can also develop your own source. Knative Eventing is also very pluggable so you can choose how you want to store your event while being processed. You can choose between in memory or persistent storage. Knative is using an open CNCF standard CloudEvent to parse the original events. The target for the events can be both Knative and Kubernetes Services. The Eventing pipelines can be very simple like a simple event being sent to a single Service, but can also get very complex. For the sake of understanding Cloud Run, let's concentrate on the basis first.

11.3.8 Knative Eventing Resources

The most essential component of Knative events are Brokers and Triggers. If we look at the below diagram we see that the events are generated from external sources and are captured by the Broker. There, one or more Triggers receives the events, filters them and passes the events to the Service.

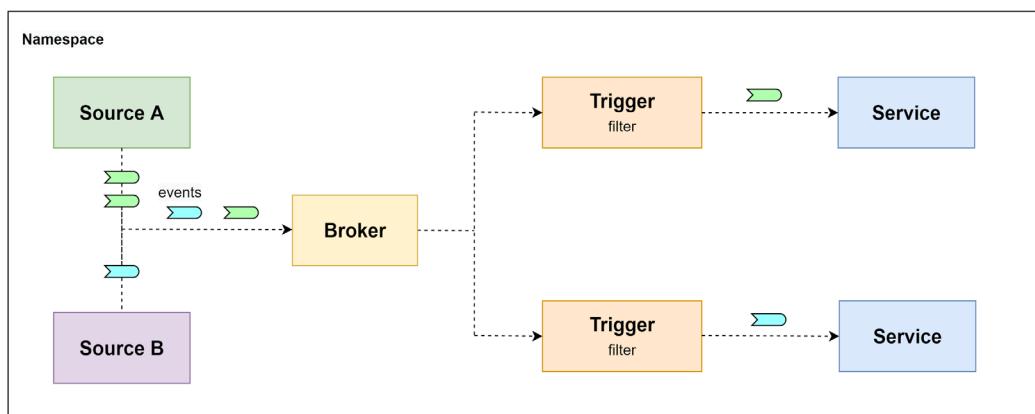


Figure 11.6. Knative eventing resources

The architecture of Knative brings a clear separation of concerns. Knative Eventing Broker is an Event Mesh that pulls or receives the events while Knative Trigger filters and routes events to the targets. Event sources are the control plane of Knative Eventing that makes sure events are sent to the broker.

Now let's have a closer look at the resources.

- **Broker API path broker.eventing.knative.dev** is essentially an addressable Event Delivery System that you install in your namespace. This gets done by setting a label on your namespace. Similar as what you do with Istio when you want to do sidecar injections into the Pods. Events are received by the Broker and then sent to subscribers. The messages are stored in a channel that is managed by the Broker.

The channel can be a simple In-memory channel or can use persistent storage for reliability purposes. Examples of those are Pub/Sub and Kafka. The configuration of the channels is stored in ConfigMaps. You can install Broker into multiple namespaces. The use case for this is if you want to have different types of messages. You can also filter which events are accepted by the Broker.

Bellow we can see an example definition of a Knative Broker:

```
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  annotations:
    eventing.knative.dev/broker.class: MTChannelBasedBroker
  name: default
  namespace: default
spec:
  config:
    apiVersion: v1
    kind: ConfigMap
    name: config-br-default-channel
    namespace: knative-eventing
```

Triggers API path trigger.eventing.knative.dev matches the event with a Service. So it is defined for which type of event (e.g Cloud Storage object created sent the event to that Service). Triggers can filter the events based on one or more attributes. If there are multiple attributes all of the attributes values need to match. It can also produce new event types out of the received event. This can be a nice use case for filtering events with sensitive data.

Bellow we can see an example definition of a Knative Trigger:

```
apiVersion: eventing.knative.dev/v1alpha1
kind: Trigger
metadata:
  name: helloworld-python
  namespace: knative-samples
spec:
  broker: default
  filter:
    attributes:
      type: dev.knative.samples.helloworld
      source: dev.knative.samples/helloworldsource
  subscriber:
    ref:
      apiVersion: v1
      kind: Service
      name: helloworld-python
```

Source API path <source_name>.eventing.knative.dev are defined as CRDs. The list is still growing and includes: AWS SQS, Google Cloud PubSub, Google Cloud Scheduler, Google

Cloud Storage, Github, Gitlab etc. To see the full list of events see : <https://knative.dev/docs/eventing/sources/>. You can either use an existing Source or create your own. The following example shows how to configure *CloudPubSubSource* event source. The event will be generated whenever a message gets published to Pub/Sub topic named "testing".

Below we can see an example definition of a Knative CloudPubSubSource:

```
apiVersion: events.cloud.google.com/v1
kind: CloudPubSubSource
metadata:
  name: cloudpubsubsource-test
spec:
  topic: testing
  sink:
    ref:
      apiVersion: v1
      kind: Service
      name: event-display
```

In the below diagram you can see how Event Source works with different types of events sources. Those are GitHub, FTP and Google Cloud Storage (GCS).

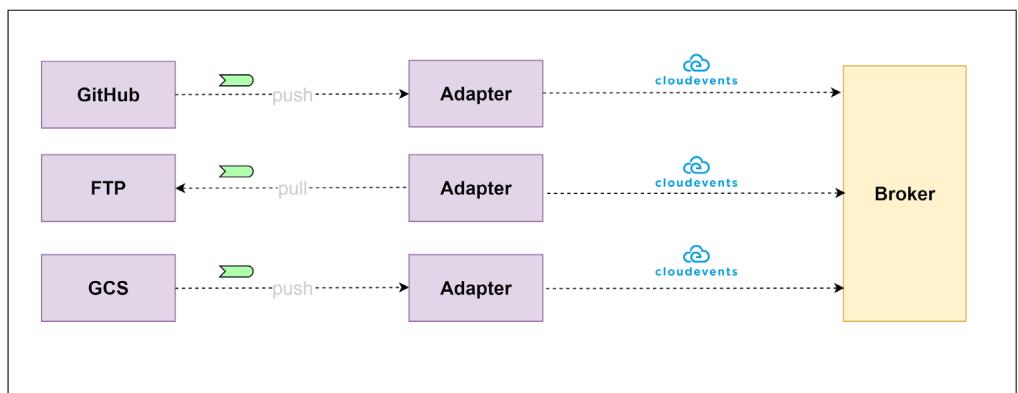


Figure 11.7.

The events are pulled or get pushed to the Adapters. This depends on whether the source is actually capable of pushing events. In case it is not the events need to be pulled. The Adapters are developed to understand the events and translate them into a common Cloud Events format. Once translated they available for the Broker to be picked up in the new format.

How EVENT SOURCES WORK

The Event Sources consist of Control and Data Plane. The control plane is responsible for configuration of the event delivery with the authoritative source, setup the data plane and cleanups. So to put it simple it creates the webhooks and subscriptions. The Data Plane performs the push/pull operations validates and converts the data into cloud events.

On top of existing sources you are able to create your own Event Sources. This can be done using Kubernetes Operators, Container sources or using an existing Services. To see how you can develop your own source refer to the knative eventing documentation (<https://knative.dev/docs/eventing/custom-event-source/custom-event-source/>)

11.3.9 Knative Use Cases

With Knative you can cover a number of use cases from a simple single service to very complex multi microservices applications. Using Knative Serving you can create both HTTP and gRPC¹³ services, webhooks and APIs. You are able to manage rollouts and rollbacks. You can easily control the traffic to your application. With cloud eventing you can create simple or very complex eventing pipelines. Combining those two services you can deliver a fully cloud-native event driven applications.

Let's have a look at a simple example of binding running services to an Cloud IoT Core.¹⁴ The messages from the IoT devices are sent to the Google Cloud IoT Core and sinked to PubSub. The Cloud Eventing service uses the PubSub Source to get the messages from the topic. The messages are sent to the Broker and converted to Cloud Events. Trigger makes sure the events are sent to the proper service that can further process, log or display them to the user.

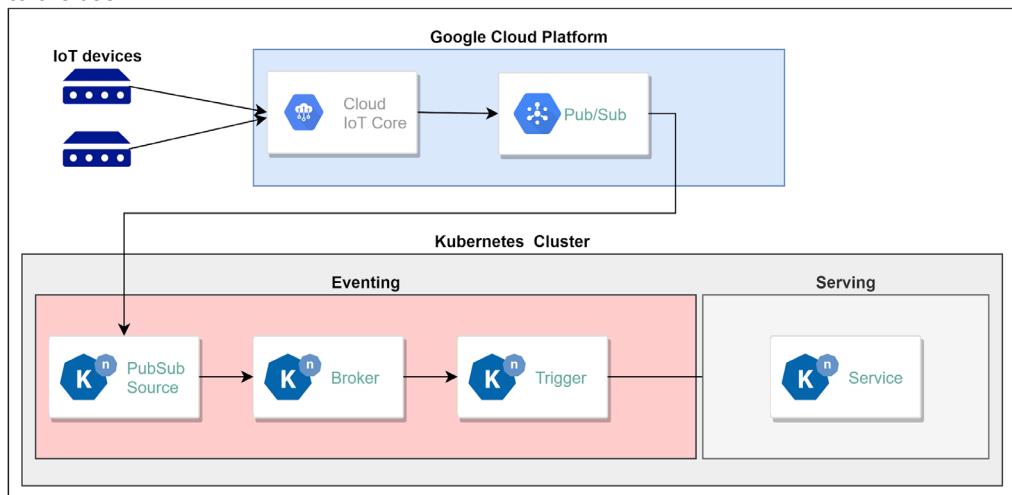


Figure 11.8.

If you would like to try Pub/Sub example yourself we encourage you to follow the step by step tutorial <https://github.com/google/knative-gcp/tree/main/docs/examples/cloudpubsubsource>

¹³ gRPC: <https://grpc.io/docs/what-is-grpc/introduction/>

¹⁴ Cloud IoT Core: <https://cloud.google.com/iot-core>

11.3.10 Observability

Knative comes with logging and tracing capabilities. The following open source software is supported:

- **Prometheus** and **Grafana** for metrics
- **ELK stack (Elasticsearch, Logstash and Kibana)** for logs
- **Jaeger** or **Zipkin** for distributed tracing

To learn more about metrics and tracing see the Chapter 5 Anthos Service Mesh: security and observability at scale.

You can also integrate with **Google Cloud Logging** (former Stackdriver Logging) for logs using the **Fluent Bit** agent. The installation procedure for each of the component is well described in the following article:
<https://knative.dev/docs/serving/observability/logging/collecting-logs/>

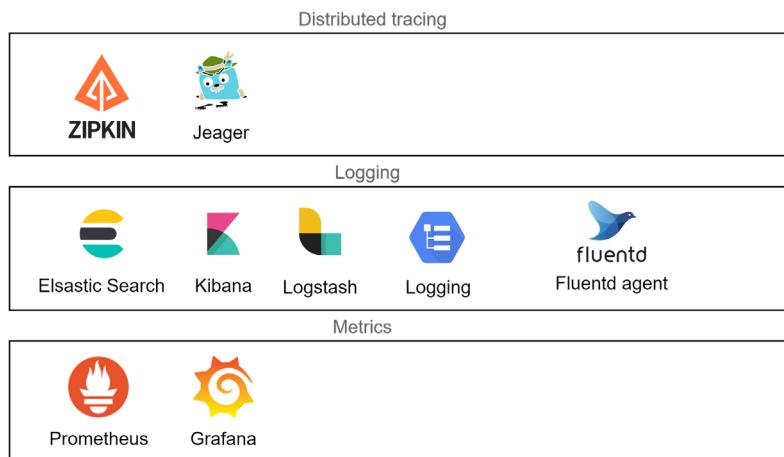


Figure 11.9. Knative observability ecosystem

As those components are deployed like any other Kubernetes application you can access them by exposing the Kubernetes service. Below see the example of pods running on your cluster after deployment.

NAME	READY	STATUS	RESTARTS	AGE
grafana-798cf569ff-v4q74	1/1	Running	0	2d
kibana-logging-7d474fbb45-6qb8x	1/1	Running	0	2d
kube-state-metrics-75bd4f5b8b-8t2h2	4/4	Running	0	2d
node-exporter-cr6bh	2/2	Running	0	2d
node-exporter-mf6k7	2/2	Running	0	2d
node-exporter-rhzr7	2/2	Running	0	2d
prometheus-system-0	1/1	Running	0	2d
prometheus-system-1	1/1	Running	0	2d

11.3.11 Installing Knative

You can install Knative on multiple cloud platforms or on premise as long as you run a Kubernetes cluster this includes but is not limited to:

- Amazon EKS
- Google GKE
- IBM IKS
- Red Hat OpenShift Cloud Platform
- Minikube

In the end Knative is nothing else but a Kubernetes application. The installation can be done either by using YAML files or Operator. To learn more about Operators see: <https://kubernetes.io/docs/concepts/extend-kubernetes/operator/>. The installation process for the **Knative Serving** component consists of:

- Installation of the Custom Resource Definitions (aka CRDs):
- Installation of the core components of Serving
- Installation of the networking layer
- Configuration of DNS
- Installation of optional Serving extensions

Installation of the **Knative Eventing** component consist of:

- Installation of the Custom Resource Definitions (aka CRDs):
- Installation of the core components of Eventing
- Installation of the default Channel (messaging) layer
- Installation a Broker (eventing) layer
- Optional Eventing extensions (sources)

Once you are done with the installation of the Serving and Eventing you can install the observability components described in the previous chapter. The well described step by step procedure for end to end installation is available here: <https://knative.dev/docs/getting-started/quickstart-install/>

11.3.12 Deploying to Knative

You can follow a simple guide to deploy your first application to Knative. It is actually as simple as applying a single Knative Service object. This is assuming you already have a containerised Python application that responds with response *Hello Python Sample v1!* Stored in Docker Hub. (See <https://github.com/knative/docs/tree/main/code-samples/eventing/helloworld-python> to check the source code for that application.

1. Run the following command to create a Knative Service:

```
kubectl apply -f service.yaml
```

Where Service is defined in the `service.yaml` file:

```
apiVersion: serving.knative.dev/v1
```

```

kind: Service
metadata:
  name: test
  namespace: default
spec:
  template:
    spec:
      containers:
        - image: docker.io/<user>/<image_name>
          env:
            - name: TARGET
              value: "v1"

```

- Once deployed multiple objects are created for you. This includes Pods, Knative Service, Configuration, Revision and Route. . You can verify them by running:

```
kubectl get pod,ksvc,configuration,revision,route
```

- To obtain IP address you can access the service and get the ip address of the Istio ingress gateway:

```
kubectl get ksvc helloworld-python --output=custom-
  columns=NAME:.metadata.name,URL:.status.url
```

This will return the URL :

NAME	URL
helloworld-python	http://helloworld-python.default.1.2.3.4.xip.io

- Now test the application by running a curl query:

```
curl http://helloworld-python.default.1.2.3.4.xip.io
```

Note that the xip.io domain is so called Magic DNS you can configure it when installing Knative (see: <https://knative.dev/docs/serving/using-a-custom-domain/>).

- You should see the following output

```
Hello Python Sample v1!
```

You have successfully deployed your first Knative application!

To get some hands-on experience with Knative we suggest you follow the below examples. They cover end to end Knative apps development and deployment scenarios with multiple language support. We especially recommend the Mete Atamel tutorial on Knative which takes you by hand from a very simple deployment to very complex ones including usage of Google Cloud services like Pub/Sub, AI APIs, BigQuery etc. We are sure you will have a lot of fun!

Table 11.2. References for deploying to Knative

Title	URL
Knative Serving Code Samples	https://knative.dev/docs/serving/samples/
Knative Eventing Code Samples	https://knative.dev/docs/eventing/samples/
Mete Atamel Knative Tutorial with multiple examples	https://github.com/meteatamel/knative-tutorial

KNATIVE SUMMARY

With Knative Service you no longer have to choose between the flexibility of Kubernetes and simplicity of Function as a Service. You get the best of two worlds. You are able to run your serverless workload anywhere. With Knative Eventing you are able to subscribe and receive events from a number of predefined sources as well as define your own source using cloud native architecture.

CLOUD RUN VS KNATIVE ON ANTHOS

Cloud Run is a fully managed serverless offering, Knative on Anthos runs on top of your Anthos clusters. You have the same way of interacting with the Cloud Run whatever version you go for. Cloud Run however runs on Google Infrastructure and you don't need to worry about the underlying platform.

Note: for the purposes of this book we will refer to Cloud Run (fully managed) as Cloud Run.

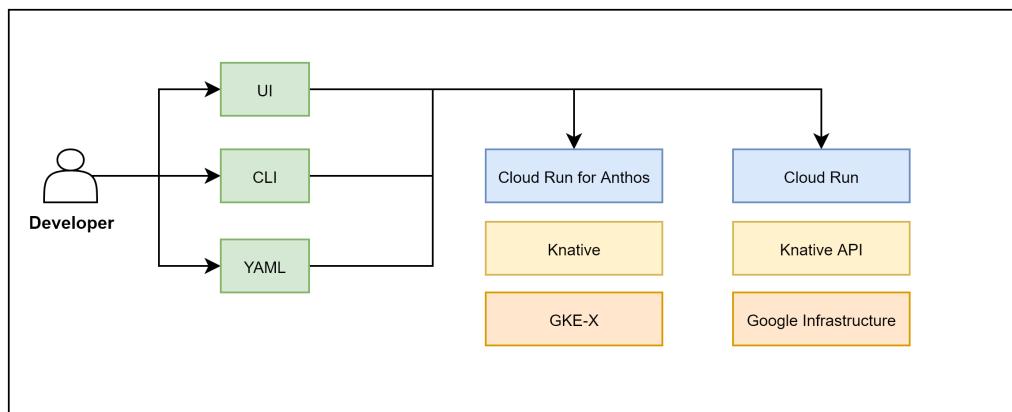


Figure 11.10. Cloud Run and Knative on Anthos architecture

Ok, so we know what the main differences are but you still might be wondering which service better fulfills your workloads needs. Have a look at the table below that shows a little bit more details on the differences:

Table 11.3. Cloud Run vs Cloud Run for Anthos

Feature	Cloud Run	Knative on Anthos
Price	Pay-per-use	GKE Anthos cost
Compute	CPU and Memory limits	As per GKE cluster nodes capabilities (includes GPU)
Isolation	Based on gVisor/other sandbox	Default GKE isolation
Scaling	1000 Container with extensible Quota	As per GKE cluster
URL/SSL	URL and SSL is autogenerated	Can configure custom domain
Domains	Custom domain can be created	
Network	Access to VPC via Serverless VPC access	Direct access to VPC
Service Mesh	Integrated with service mesh	Services connected to Istio Service Mesh
Execution Environment	Google Infra	GKE Cluster

So when to choose each of the offerings? This very much depends on how much control you want to have over your application execution and if you need a custom hardware that GKE nodes might come with. As an example you might want to leverage GPUs to boost the performance of our ML pipelines. In such a case Knative on Anthos will be the way to go.

11.4 Summary

In this chapter we have talked about how Knative fits into Anthos suite and we have learned the following:

- Knative abstracts the complexity of Kubernetes away.
- The workloads are portable to any Kubernetes cluster.
- Knative has multiple components that can address multiple use cases.
- Eventing is the component that orchestrates events originating from various sources.
- Service is the component that allows you to deploy container based serverless workloads and serve them to the users.
- Serverless Kubernetes workloads can be deployed and served using Knative on

Anthos.

- Versions of the application can be controlled using revisions.
- Traffic to the application can be managed using revision parameters.
- You can get insights into your application using a rich open source ecosystem of tools for monitoring, logging and tracing.

In the next chapter the networking concepts of Anthos networking will be discussed.

12

Anthos - the networking environment

by Ameer Abbas

This chapter covers

- Anthos cloud networking and hybrid connectivity between multiple cloud environments.
- Anthos Kubernetes and GKE networking including dataplane v2.
- Anthos multi-cluster networking including service discovery and routing.
- Service to service and client to service connectivity.

Anthos networking can be divided into four sections. Each section provides a layer of connectivity between entities such as environments (public cloud and on prem for example), Anthos GKE clusters, and service to service communications. The four layers are as follows:

1. **Cloud networking and hybrid connectivity** - This section addresses the lowest layer of networking and covers how different infrastructure environments can be interconnected.
2. **Anthos GKE networking** - Anthos GKE clusters come in a variety of implementations depending upon the infrastructure they are deployed in. This section covers Anthos GKE cluster networking including how Ingress works in various environments.
3. **Multicluster networking** - This section addresses how various Anthos GKE clusters connect to each other. Anthos GKE clusters may be deployed in a single infrastructure environment (in GCP, e.g.) or they can be deployed across multiple infrastructure environments (GCP and in an on premise data center).

- 4. Service and client connectivity** - This section addresses how applications running on Anthos connect to each other. This section also addresses how clients and services running outside of Anthos can connect to Services running inside the Anthos platform.

Networking

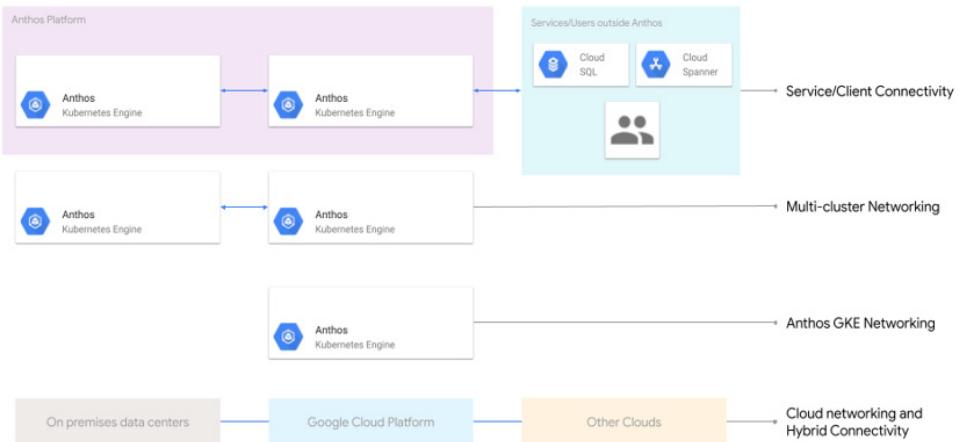


Figure 12.1 Four layers of Anthos networking

12.1 Cloud networking and hybrid connectivity

This section addresses various aspects of network connectivity at the infrastructure environment level. Anthos is a multi cloud platform and can run in one or more public and private cloud environments.

At the infrastructure layer, there are two main ways to deploy Anthos.

- 1. Anthos in a single cloud environment** - for example GCP or on premises data center or even in another public cloud.
- 2. Anthos in a multi/hybrid cloud environment** - for example a platform deployed in GCP and in one or more on premises data centers.

12.1.1 Single cloud deployment

Anthos platform can be deployed in a single cloud environment. The single cloud environment can be GCP, another public or private cloud or on premises data centers.

ANTHOS ON GCP

Anthos on GCP uses resources that are placed within a Virtual Private Cloud [VPC¹](#). There are multiple ways to configure VPCs in GCP. Depending on the needs of the company, a single VPC (in a single GCP project) might suffice. In more complex design, shared VPC, peered VPC or even multiple disparate VPCs are required. Anthos platform can work with a variety of VPC designs. Choosing the right VPC architecture up front is important as it may pose scalability and operational consequences later on. Various VPC design and decision criteria are discussed below.

SINGLE VPC

This is the simplest VPC design. For small environments where everything is contained in a single GCP project, you may choose a single VPC. A single VPC results in a flat network meaning all resources using the VPC are on the same network. Connectivity between resources can be controlled via security features at various layers in the Anthos platform. For example you can use [VPC firewall²](#) at the network layer, Kubernetes [NetworkPolicies³](#) inside Kubernetes data plane and [Anthos Service Mesh⁴](#) authentication and authorization policies at the service mesh layer. With this approach, multiple teams utilize resources in the same GCP project and same VPC. Single VPC design also simplifies network administration. All resources, whether inside or outside of the Anthos platform, reside on the same flat network and can communicate easily as allowed via security rules. No additional configuration is required to connect resources together.

¹ <https://cloud.google.com/vpc>

² <https://cloud.google.com/vpc/docs/firewalls>

³ <https://kubernetes.io/docs/concepts/services-networking/network-policies/>

⁴ <https://cloud.google.com/service-mesh/docs/security-overview>

Single VPC on GCP

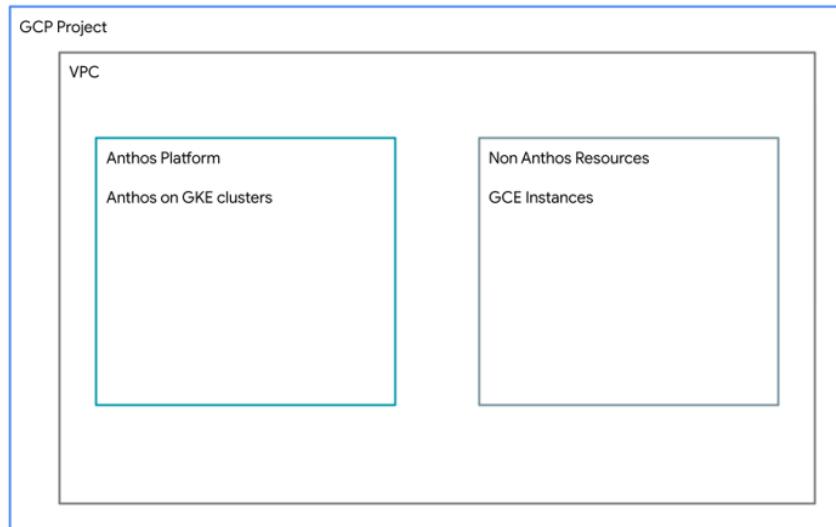


Figure 12.2 Single VPC architecture in GCP

The primary challenge with a single VPC design is scale. While a single VPC design might be sufficient for small to medium sized implementations, large implementations may not be possible as you will start to hit [VPC limits⁵](#). As the organization grows, separate projects may need to be created for separate products, teams or environments. A single VPC design does not support multi project environments. Depending upon the industry, there might be regulations that prohibit hosting all resources in a single VPC and would require some level of network or project separation.

When designing your network structure for Anthos, it is important to understand and account for the longevity of the platform up front. For example, in two to four years, how much will the platform scale and will there be any other restrictions that need to be taken into account - quota or regulation wise.

SHARED VPC

Using a [Shared VPC⁶](#) is the recommended way to provision a network on GCP for the Anthos platform.

A Shared VPC design can be used for both simple and complex (large scale and multi tenant) Anthos environments. Shared VPC allows a single VPC to be shared across multiple projects. This results in a single flat network space shared by multiple tenants each within their own project. Separating GCP projects by products/tenants allows for granular IAM permissioning

⁵ <https://cloud.google.com/vpc/docs/quota>

⁶ <https://cloud.google.com/vpc/docs/shared-vpc>

at a project level. At the same time resources in multiple projects can still connect to each other (if allowed) as if they were on a single network.

A network host project contains a centralized “shared” VPC. All network resources are located in this network host project, including subnets, firewall rules, network permissions, etc. A centralized networking team owns and controls the network host project. This ensures that the organization’s network best practices are enforced by a single qualified team of networking experts. Multiple service projects can then use network resources from the host project. Subnets are shared from the network host project to multiple service projects and used by resources inside each service project.

For Anthos on GCP, it is recommended to create a service project named such as “platform_admins”. Anthos platform (all GKE clusters) reside inside the `platform_admins` project. The `platform_admins` project (as the name suggests) is owned by the platform administrator team who manage and maintain the lifecycle of the Anthos platform. Platform administrators are one of many tenants of the network host project. Similarly, products and environments get their own service projects. Anthos is a shared multi tenant platform where each tenant gets a “landing zone” to run their services on Anthos. A landing zone is a set of resources required to run a Service on the Anthos platform. A landing zone is typically one (or more) namespaces in one (or more) Anthos GKE clusters and a set of policies required to run that Service. All non-Anthos resources (belonging to a Service) are provisioned and managed in the individual Service’s GCP project. This way multiple tenants can have their own projects for non-Anthos resources and they can all share a single Anthos GKE on GCP clusters. Using a shared VPC allows Anthos and non-Anthos resources to connect to each other.

Shared VPC on GCP

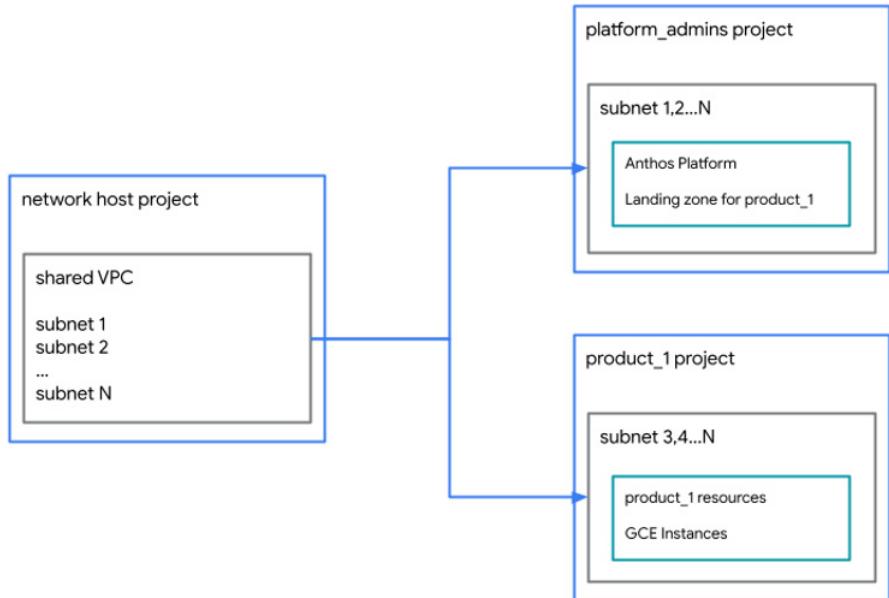


Figure 12.3 Shared VPC architecture in GCP

MULTIPLE VPC

The two previous VPC implementations result in a flat network where all resources are provisioned in a single logical VPC. In some cases, security or regulatory restrictions may require the separation of resources into multiple VPCs. The company or organization may also want each team or product to manage their own network.

Anthos GKE on GCP platform can be installed in one VPC. Multi tenancy allows you to share the Anthos GKE on GCP with multiple tenants. It may be required to connect services running on the Anthos platform to services outside of the platform. In this design, these services run in different VPCs. For example, services running on Anthos GKE in GCP may be running on a VPC called `anthos_vpc` and non-Anthos resources may be running on a VPC called `product_1_vpc`. There are a couple of ways to connect these services:

1. **IPSec VPN⁷** - An IPSEC VPN tunnel can be created between two VPCs. IPsec VPN traffic flows over the public internet in a secure manner. Traffic traveling between the two networks is encrypted by one VPN gateway and then decrypted by the other VPN gateway. This action protects your data as it travels over the internet. Flowing through the public internet may result in performance degradation. IPSEC VPN can be helpful for large scale environments.

⁷ <https://cloud.google.com/network-connectivity/docs/vpn/concepts/overview>

2. **VPC Peering⁸** - Multiple VPCs can be peered together allowing VPC inter-connectivity without having to connect VPCs via IPSEC VPN. VPC peering offers the same data plane and performance characteristics as of a single VPC, but with boundaries for administration (security and configurations).. This results in better security and performance for VPC to VPC traffic. VPC peering requires coordination between network admins of the two VPCs. VPC peering does not allow for overlapping IP addresses. Also, both VPC owners maintain separate firewalls with desired rules allowing traffic between subnets belonging to the two VPCs.
3. **Public Internet and secure Ingress** - If VPC peering or VPN is not an option, services can communicate over the public internet. In this case, higher level functionality like Anthos Service Mesh can be used to encrypt traffic between networks using [TLS⁹](#) or mTLS (mutual TLS). This method works well if only a small number of services require connectivity across VPCs because this method requires per-service (destination service) configuration as opposed to the previous two methods which connect two networks at the TCP/IP layer.

Multiple VPC on GCP

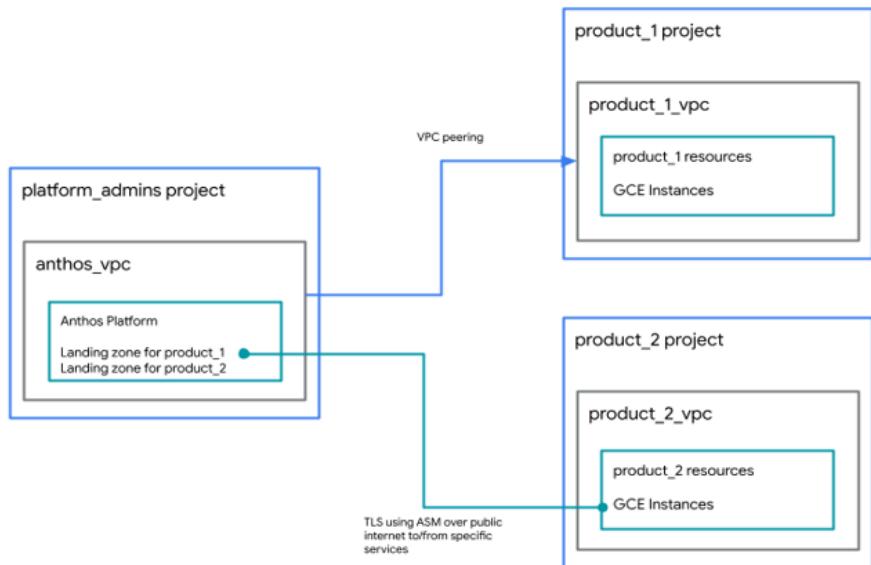


Figure 12.4 Multiple VPC architecture in GCP

⁸ <https://cloud.google.com/vpc/docs/vpc-peering>

⁹ https://en.wikipedia.org/wiki/Transport_Layer_Security

ANTHOS ON A SINGLE NON-GCP ENVIRONMENT

Anthos can be deployed in a variety of non-GCP environments. These environments include on premises data centers, public clouds and private clouds. At the infrastructure layer, there are two primary network designs to consider.

SINGLE FLAT NETWORK

As the name suggests, flat networks are composed of a single logical network space where both Anthos and non-Anthos resources (for example VMs running outside of the Anthos platform) reside on the same network. A flat network in a set of subnets connected by routers and switches where each IP endpoint can switch or route to another endpoint given the correct routing (and firewall) rules. A single GCP VPC is an example of a flat network where you may have multiple subnets and routing/firewall rules to allow routing between any two endpoints. Flat networks are easier to manage as opposed to multiple disparate networks, but they require more rigor when it comes to security since all entities are on the same logical network space. Firewall rules, NetworkPolicies and other functionality can ensure only the allowed entities have network access. Flat networks may also run into scalability issues. Typically these networks use RFC1918 address space which provides a finite number of IP addresses (just under 18 million addresses). Typically, a flat logical network requires all resources to utilize the same RFC1918 space. There are exceptions to this general rule where large organizations may use their own public IP address space for internal addressing. Regardless of the IP address usage, it is important to note that in a flat network, no two endpoints may have the same IP address.

MULTIPLE NETWORKS

Anthos can also be deployed in a multi-network environment. Anthos GKE clusters can be deployed on single or multiple networks as required. Typically, it is easier to manage network connectivity for applications running on Anthos platform if Anthos is deployed in the same network. It is possible to deploy Anthos platform across multiple disconnected networks. In some cases, it may be required to connect multiple networks. There are three main ways to connect applications running on Anthos platform across multiple networks.

1. **VPN/ISP** - Connect multiple networks together via a VPN or the chosen ISP may provide this connectivity. These are the typical choices for connecting multiple on premises data centers.
2. **VPC Peering** - VPC peering may be used if Anthos is deployed on a public cloud that offers VPC peering functionality.
3. **Gateways or mTLS** - Services may be connected securely over the public internet using TLS, mTLS or a secured API gateway. This functionality exists through service meshes like [Anthos Service Mesh](https://cloud.google.com/service-mesh/docs/overview)¹⁰ (ASM) or API gateways like [Apigee](https://cloud.google.com/apigee)¹¹. This is done on a per service level whereas the first two options are configured at the network layer.

¹⁰ <https://cloud.google.com/service-mesh/docs/overview>

¹¹ <https://cloud.google.com/apigee>

12.1.2 Multi / Hybrid Cloud Deployment

Anthos is a multicloud platform and can be deployed to multiple environments, for example public/private clouds and on premises data centers. Managing networking across multiple environments is challenging since each environment is unique and managing resources differs depending upon the provider. For example, the way a GCP VPC is provisioned is different from an AWS VPC or a data center network. Anthos provides a common interface across multiple environments.

Anthos platform can be deployed to multiple environments in three main ways.

1. **Multi cloud deployment** - Anthos platform can be deployed to multiple public cloud environments, for example GCP and one or more public clouds like AWS and Azure.
2. **Hybrid cloud deployment** - Anthos platform can be deployed to GCP and one or more on premises data centers.
3. **Multi and hybrid cloud deployment** - This deployment is a combination of the two deployments mentioned above. For example, Anthos platform can be deployed to GCP, one or more on premises data centers as well as one or more non-GCP public clouds.

MULTI/HYBRID NETWORKING

When Anthos is deployed across multiple infrastructure environments, these environments are required to have network connectivity to GCP.

There are three network connectivity options available to connect multiple infrastructure environments.

CLOUD INTERCONNECT

[Cloud Interconnect](#)¹² extends on-premises network to Google's network through a highly available, low latency connection. You can use Dedicated Interconnect to connect directly to Google or use Partner Interconnect to connect to Google through a supported service provider. Dedicated Interconnect provides direct physical connections between your on-premises network and Google's network. Dedicated Interconnect enables you to transfer large amounts of data between networks, which can be more cost-effective than purchasing additional bandwidth over the public internet.

For Dedicated Interconnect, you provision a Dedicated Interconnect connection between the Google network and your own router in a [common location](#)¹³. The following diagram shows a single Dedicated Interconnect connection between a Virtual Private Cloud (VPC) network and your on-premises network.

¹² <https://cloud.google.com/network-connectivity/docs/interconnect>

¹³ <https://cloud.google.com/network-connectivity/docs/interconnect/concepts/choosing-colocation-facilities#locations-table>

Dedicated interconnect

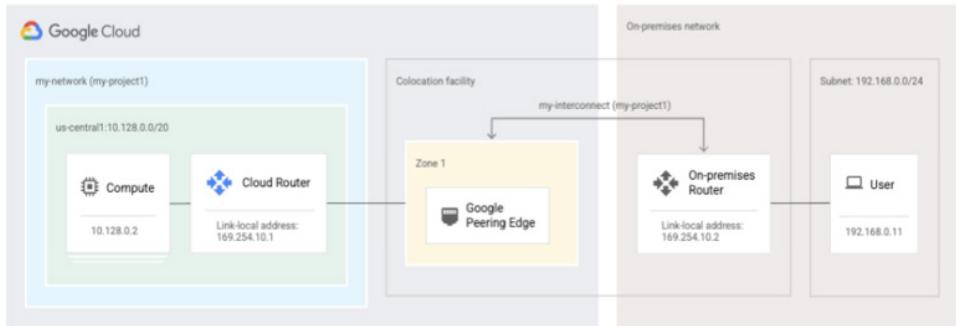


Figure 12.5 Dedicated interconnect between GCP and on-premises data center

For this basic setup, a Dedicated Interconnect connection is provisioned between the Google network and the on-premises router in a common colocation facility.

When you create a [VLAN attachment¹⁴](#), you associate it with a [Cloud Router¹⁵](#). This Cloud Router creates a BGP session for the VLAN attachment and its corresponding on-premises peer router. The Cloud Router receives the routes that your on-premises router advertises. These routes are added as custom dynamic routes in your VPC network. The Cloud Router also advertises routes for Google Cloud resources to the on-premises peer router.

Depending on your availability needs, you can configure Dedicated Interconnect to support mission-critical services or applications that can tolerate some downtime. To achieve a specific level of reliability, Google has two prescriptive configurations:

1. [Achieve 99.99% \(52.60 mins per year\) availability for Dedicated Interconnect¹⁶](#)
(recommended)
2. [Achieve 99.9% availability for Dedicated Interconnect¹⁷](#)

Cloud interconnect is the most robust and secure option to connect GCP and non-GCP environments together. This is the recommended option to connect GCP and one or more on-premises data centers.

CLOUD VPN

[Cloud VPN¹⁸](#) securely connects your [peer¹⁹](#) network to your [Virtual Private Cloud \(VPC\) network²⁰](#) through an IPSEC [VPN²¹](#) connection. Traffic traveling between the two networks is

¹⁴ <https://cloud.google.com/network-connectivity/docs/interconnect/how-to/dedicated/creating-vlan-attachments>

¹⁵ <https://cloud.google.com/network-connectivity/docs/router/concepts/overview>

¹⁶ <https://cloud.google.com/network-connectivity/docs/interconnect/tutorials/dedicated-creating-9999-availability>

¹⁷ <https://cloud.google.com/network-connectivity/docs/interconnect/tutorials/dedicated-creating-999-availability>

encrypted by one VPN gateway, and then decrypted by the other VPN gateway. This protects your data as it travels over the internet. Google Cloud offers HA VPN providing higher uptime and throughput with additional/redundant VPN connection at a higher cost.

Each Cloud VPN tunnel can support up to 3 gigabits per second (Gbps) total for ingress and egress. You can use multiple Cloud VPN tunnels to increase your ingress and egress bandwidth. The aggregate increase (using multiple Cloud VPN tunnels) does not increase per flow bandwidth.

Cloud VPN can be used between GCP and on-premises data centers as well as between GCP and other public cloud vendors. This is the easiest option to set up and can be up and running without any delays. Cloud VPN can also be used in conjunction with Cloud Interconnect as a secondary connectivity option.

PUBLIC INTERNET

Applications running on Anthos platform on multiple environments can be connected over the public internet without using Cloud Interconnect or VPN. Applications running on the platform connect over the public internet using TLS/mTLS.

Anthos Service Mesh (ASM) is a service mesh as part of the Anthos platform. ASM uses client side proxies injected in each Pod to connect services together. One of the security features of these proxies is to secure connectivity using mTLS. Using a common root Certificate Authority (CA) on multiple environments, the sidecar proxies can connect using a secure mTLS connection via gateways (for example ingress or east-west gateways) across the public internet. For details on Anthos Service Mesh, please refer to chapter "Anthos Service Mesh: security and observability at scale".

If a large number of services require connectivity between environments, then this option might not be operationally scalable. In such a case, it is recommended to use one of the network connectivity options mentioned above.

DISCONNECTED ENVIRONMENTS

In some situations, it may be required to have environments that are completely disconnected from each other. Anthos platform supports disconnected environments. The disconnected environments must have network connectivity to GCP so that the platform (i.e. Anthos clusters) can be [registered](#) to a GCP project. This is required for controlplane traffic only. For certain Anthos functionality, registering a cluster is required, for example to use [Multicluster Ingress](#) on Anthos clusters, all participating clusters must be registered to GCP. The services across disconnected environments will not be able to communicate to each other.

¹⁸ <https://cloud.google.com/network-connectivity/docs/vpn/concepts/overview>

¹⁹ <https://cloud.google.com/network-connectivity/docs/vpn/concepts/key-terms#peer-definition>

²⁰ <https://cloud.google.com/vpc/docs>

²¹ https://en.wikipedia.org/wiki/Virtual_private_network

12.2 Anthos GKE Networking

Anthos GKE clusters can be deployed to a variety of environments, for example GCP, on VMWare in an on-premises data center, on bare metal servers, and on AWS. In addition to the supported Anthos clusters, you can also register any [conformant](#) Kubernetes cluster to the Anthos platform. For example, you can register [EKS](#) clusters running in AWS and [AKS](#) clusters running in Azure to the Anthos platform.

There are currently six types of Anthos clusters.

1. Anthos clusters on GCP (GKE)
2. Anthos clusters on VMWare (GKE on-prem)
3. Anthos clusters on bare metal
4. Anthos clusters on AWS (GKE on AWS)
5. Anthos clusters on Azure (GKE on Azure)
6. Anthos attached clusters (conformant Kubernetes clusters)

12.2.1 Anthos cluster networking

CLUSTER IP ADDRESSING

All Anthos GKE clusters require three IP subnets:

1. Node and API server IP addresses
2. Pod IP addresses
3. Services or ClusterIP addresses

Node and API server IP addresses are LAN (in case of on-premises data centers) or VPC (in case of public clouds) IP addresses. Each node and API server gets a single IP address. Depending upon the number of nodes/API servers required, ensure you have the required number of IP addresses.

Pod IP addresses are assigned to every Pod in an Anthos GKE cluster. Each node in an Anthos cluster is assigned a unique IP address range which is used to assign Pod IP addresses (running inside that node). If the Pod moves from one node to another, its IP address changes based on the IP address range of the new node. The API server takes a large IP range, often called the Pod CIDR IP range, for example a /14 or a /16 (you can learn about IP subnets [here²²](#)), equally divides this range into smaller IP ranges and assigns a unique range to each node. You define the desired number of Pods per node which is used by the API server to slice the large subnet into smaller subnets per node. For example, if you want 30 Pods per node, each node requires a minimum of a /27. Your Pod IP range must be large enough to account for N subnets with 32 addresses each where N is the maximum number of nodes in the cluster.

Pod IP addresses are routable within the cluster. They may or may not be routable from outside of the cluster, depending on the type and implementation of the cluster. This is discussed in detail in the next section.

²² <https://en.wikipedia.org/wiki/Subnetwork>

Service or ClusterIP addresses are assigned to every Kubernetes [Service](#). Unlike Pod IP addresses, which may change as Pods move between nodes, [ClusterIP](#) addresses remain static and act as a load balancing virtual IP address (VIP) to multiple Pods representing a single Kubernetes Service. As the name suggests, Service IPs or ClusterIPs are locally significant to the cluster and cannot be accessed from outside of the cluster. Services inside the cluster can access services using ClusterIPs.

CLUSTER NETWORKING DATAPLANE

Anthos GKE clusters provide two options for networking dataplane.

GKE DATAPLANE V1: KUBE-PROXY AND CALICO

Kubernetes manages connectivity among Pods and Services using the [kube-proxy](#) component. This is deployed as a static Pod on each node by default. Any GKE cluster running 1.16 or later has a [kube-proxy](#) deployed as a [DaemonSet](#).

[kube-proxy](#) is not an in-line proxy, but an egress-based load-balancing controller. It watches the Kubernetes API server and continually maps the [ClusterIP](#) to healthy Pods by adding and removing destination NAT (DNAT) rules to the node's [iptables](#) subsystem. When a container running in a Pod sends traffic to a Service's [ClusterIP](#), the node selects a Pod at random and routes the traffic to that Pod.

When you configure a Service, you can optionally remap its listening port by defining values for `port` and `targetPort`. The `port` is where clients reach the application. The `targetPort` is the port where the application is actually listening for traffic within the Pod. [kube-proxy](#) manages this port remapping by adding and removing [iptables](#) rules on the node.

In GKE dataplane v1, Kubernetes [NetworkPolicies](#) are implemented today using the [Calico](#) component. Calico is an open source networking and network security solution for containers, virtual machines, and native host-based workloads.

This implementation uses components that rely heavily on [IPTables](#) functionality in the Linux kernel. [IPTables](#) provide a flexible, but not *programmable datapath* that enables K8s networking functions.

Dataplane v2 addresses and resolves some of these issues.

GKE DATAPLANE V2: EBPF AND CILIUM

GKE Dataplane V2 is an opinionated dataplane that harnesses the power of eBPF and [Cilium](#), an open source project that makes the Linux kernel Kubernetes-aware using eBPF.

Dataplane V2 addresses the observability, scalability, and functional requirements by providing a programmable datapath. [Extended Berkeley Packet Filter \(eBPF\)](#), a new Linux networking paradigm, exposes programmable hooks to the network stack inside the Linux kernel. The ability to enrich the kernel with user-space information—without jumping back and forth between user and kernel spaces—enables context-aware operations on network packets at high speeds.

The new dataplane adds two new cluster components - the `cilium-agent` DaemonSet that programs the eBPF datapath, and the `cilium-operator` Deployment that manages the Cilium-internal CRDs and helps the `cilium-agent` avoid watching every pod.

The dataplane also eliminates both calico cluster components - the `calico-node` DaemonSet and the `calico-typa` Deployment. These components provide `NetworkPolicy` enforcement, which is provided by the `cilium-agent` daemonset.

The dataplane also removes the `kube-proxy` static pod from the nodes. `Kube-proxy` provides service resolution functionality to the cluster, which is also provided by the `cilium-agent`.

In addition, the `cilium-cni` is chained to the existing GKE CNI by the CNI config written by Netd. Netd adds an option to chain to the `cilium-cni` by reading from the `netd-config` configmap.

Dataplane v2

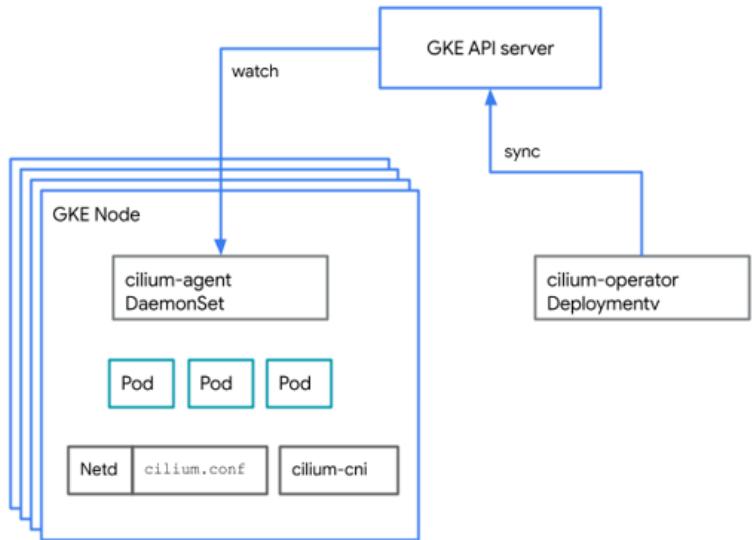


Figure 12.6 GKE Dataplane v2 architecture

Dataplane V2 offers networking programmability and scalability to Anthos clusters. One of the use cases is Network Policy flow logging. Enterprises use Kubernetes network policies to declare how pods can communicate with one another. However, there is no scalable way to troubleshoot and audit the behavior of these policies. With eBPF in GKE, you can now enforce real-time policy as well as correlate policy actions (allow/deny) to pod, namespace, and

policy names at line rate with minimal impact on the node's CPU and memory resources. As packets come into the VM, specialized eBPF programs can be installed in the kernel to decide how to route the packet. Unlike IPTables, eBPF programs have access to Kubernetes-specific metadata including network policy information. This way, they can not only allow or deny the packet, they can also report annotated actions back to user space. These events make it possible for you to generate network policy logs.

Dataplane v2: Network Policy Flow Logging

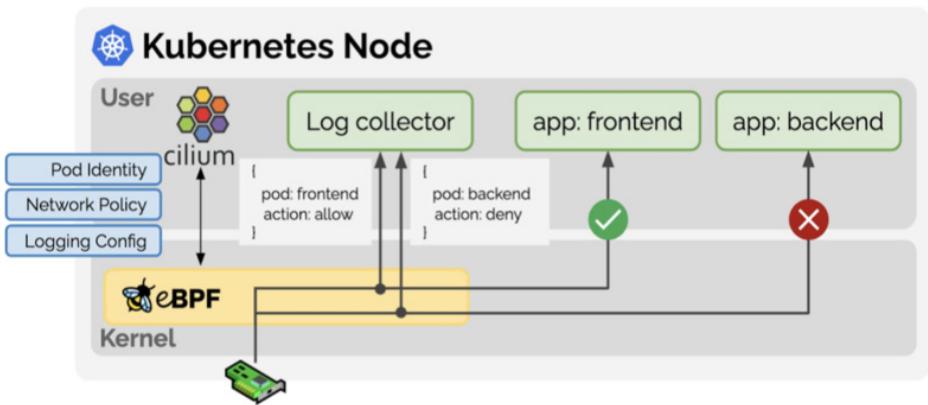


Figure 12.7 Dataplane v2: Network policy flow logging

The following table shows a comparison of networking features between Dataplane V1 and V2.

Network Feature	"Existing"	"New Data Plane"
ClusterIP service resolution	Kube-proxy using IPTables	Cilium-agent using eBPF on sockets
NodePort service resolution	Kube-proxy using IPTables	Cilium-agent using eBPF on eth0 tc hooks
LoadBalancer service resolution	Kube-proxy using IPTables redirecting to service chain	Cilium-agent using eBPF on eth0 tc hooks (same hook as above)
Network Policy enforcement	Calico using IPTables	Cilium-agent using eBPF on socket as well as eth0 tc hooks.

Depending upon the type and implementation of the Anthos cluster, networking design and requirements vary. In the next section, we look at each type of Anthos cluster in terms of networking requirements and best practices.

ANTHOS GKE ON GCP

Anthos GKE cluster runs on GCP and utilize GCP VPC functionality for Kubernetes networking. There are two types of implementations of Anthos GKE on GCP.

VPC NATIVE CLUSTERS

This is the default and recommended implementation for Anthos GKE on GCP clusters. A cluster that uses [alias IP address ranges](#) is called a VPC-native cluster. VPC native clusters use real VPC IP addresses for Pod IP ranges. This allows Pod to Pod communication within a single cluster as well as across multiple (VPC native) clusters in the same VPC. This also allows direct Pod connectivity to any routable VPC entity, for example GCE instances. VPC native clusters use [secondary IP address ranges](#) for Pod IP range and Service IP range.

VPC-native clusters have several benefits:

- [Pod](#) IP addresses are natively routable within the cluster's VPC network and other VPC networks connected to it by [VPC Network Peering](#).
- Pod IP addresses are reserved in the VPC network before the Pods are created in your cluster. This prevents conflict with other resources in the VPC network and allows you to better plan IP address allocations.
- Pod IP address ranges do not depend on custom static routes. They do not consume the [system-generated and custom static routes quota](#). Instead, automatically-generated [subnet routes](#) handle routing for VPC-native clusters.
- You can create [firewall rules](#) that apply to just Pod IP address ranges instead of any IP address on the cluster's nodes.
- Pod IP address ranges, and subnet secondary IP address ranges in general, are accessible from on-premises networks connected with Cloud VPN or Cloud Interconnect using [Cloud Routers](#).

ROUTES-BASED CLUSTERS

A cluster that uses [Google Cloud Routes](#) is called a routes-based cluster. Google Cloud routes define the paths that network traffic takes from a virtual machine (VM) instance to other destinations. The Pod IP address ranges in a routes-based cluster are not VPC IP addresses and therefore are not natively routable inside the VPC. Cloud Routes are created for each Pod IP address range so that Pods within a cluster can communicate with other Pods running on different nodes. Routes-based clusters do not provide Pod to Pod inter-cluster connectivity for multiple Anthos GKE clusters. To create a routes-based cluster, you must explicitly turn off the VPC-native option.

In a routes-based cluster, each node is allocated a /24 range of IP addresses for Pods. With a /24 range, there are 256 addresses, but the maximum number of Pods per node is 110. By having approximately twice as many available IP addresses as possible Pods, Kubernetes is able to mitigate IP address reuse as Pods are added to and removed from a node.

A routes-based cluster has a range of IP addresses that are used for Pods and Services. Even though the range is used for both Pods and Services, it is called the Pod address range. The last /20 of the Pod address range is used for Services. A /20 range has $2^{12} = 4096$ addresses. So 4096 addresses are used for Services, and the rest of the range is used for Pods.

In command output, the Pod address range is called `clusterIpv4Cidr`, and the range of addresses used for Services is called `servicesIpv4Cidr`. For example, the output of `gcloud container clusters describe` includes output similar to this:

```
clusterIpv4Cidr: 10.96.0.0/16
...
servicesIpv4Cidr: 10.96.240.0/20
```

For GKE version 1.7 and later, the Pod address range can be from any RFC 1918 block: 10.0.0.0/8, 172.16.0.0/12 or 192.168.0.0/16. For earlier versions, the Pod address range must be from 10.0.0.0/8.

The maximum number of nodes, Pods, and Services for a given GKE cluster is determined by the size of the cluster subnet and the size of the Pod address range. You cannot change the Pod address range size after you create a cluster. When you create a cluster, ensure that you choose a Pod address range large enough to accommodate the cluster's anticipated growth.

ANTHOS GKE CLUSTER IP ALLOCATION

Kubernetes uses various IP ranges to assign IP addresses to nodes, Pods, and Services.

1. **Node IP** - In Anthos GKE clusters, a node is a GCE instance. Each node has an IP address assigned from the cluster's Virtual Private Cloud (VPC) network. This node IP provides connectivity from control components like kube-proxy and the kubelet to the Kubernetes API server. This IP is the node's connection to the rest of the cluster.
2. **Pod IP CIDR** - Each node has a pool of IP addresses that GKE assigns Pods running on that node (a [/24 CIDR block by default](#)). You can optionally specify the range of IPs when you create the cluster. The [Flexible Pod CIDR range feature](#) allows you to reduce the size of the range for Pod IPs for nodes in a given node pool. Each Pod has a single IP address assigned from the Pod CIDR range of its node. This IP address is shared by all containers running within the Pod, and connects them to other Pods running in the cluster. The maximum number of Pods you can run on a node is equal to half of the Pod IP CIDR range. For example, you can run a maximum of 110 Pods on a node with a /24 range, not 256 as you might expect. This provides a buffer so that Pods don't become unschedulable due to a transient lack of IP addresses in the Pod IP range for a given node. For ranges smaller than /24, half as many Pods can be scheduled as IP addresses in the range.

3. **Service IP** - Each Service has an IP address, called the ClusterIP, assigned from the cluster's VPC network. You can optionally customize the VPC network when you create the cluster. In Kubernetes, you can assign arbitrary key-value pairs called [labels](#) to any Kubernetes resource. Kubernetes uses labels to group multiple related Pods into a logical unit called a Service. A Service has a stable IP address and ports, and provides load balancing among the set of Pods whose labels match all the labels you define in the [label selector](#) when you create the Service.

EGRESS TRAFFIC AND CONTROLS

For VPC native clusters, traffic egressing a Pod is routed using normal VPC routing functionality. Pod IP addresses are preserved in the TCP header as the source IP address. You must create the appropriate firewall rules to allow traffic between Pods and other VPC resources. You can also use `NetworkPolicy` to further control the flow of traffic between Pods within a cluster as well as traffic egressing Pods. These policies are enforced by the GKE dataplane implementation explained in the previous section. At the Service layer, you can use `EgressPolicy` through [Anthos Service Mesh](#) (ASM) to control what traffic exits the clusters. In this case, an [Envoy proxy](#) called the `istio-egressgateway` exists at the perimeter of the service mesh through which all egress traffic flows.

For routes-based clusters, all Pod egress traffic is NAT'd via the node IP address.

LOAD BALANCERS AND INGRESS

GKE provides three different types of load balancers to control access and to spread incoming traffic across your cluster as evenly as possible. You can configure one Service to use multiple types of load balancers simultaneously.

1. [External load balancers](#) manage traffic coming from outside the cluster and outside your Google Cloud Virtual Private Cloud (VPC) network. They use forwarding rules associated with the Google Cloud network to route traffic to a Kubernetes node.
2. [Internal load balancers](#) manage traffic coming from within the same VPC network. Like external load balancers, they use forwarding rules associated with the Google Cloud network to route traffic to a Kubernetes node.
3. [HTTP\(S\) load balancers](#) are specialized external load balancers used for HTTP(S) traffic. They use an Ingress resource rather than a forwarding rule to route traffic to a Kubernetes node.

The external and internal load balancers described above are TCP/L4 load balancers. If your Service needs to be reachable from outside the cluster and outside your VPC network, you can configure your Service as a [LoadBalancer](#), by setting the Service's `type` field to `LoadBalancer` when defining the Service. GKE then provisions a [Network load balancer](#) in front of the Service. The Network load balancer is aware of all nodes in your cluster and configures your VPC network's firewall rules to allow connections to the Service from outside the VPC network, using the Service's external IP address. You can assign a static external IP address to the Service.

For traffic that needs to reach your cluster from within the same VPC network, you can configure your Service to provision an [Internal load balancer](#). The Internal load balancer chooses an IP address from your cluster's VPC subnet instead of an external IP address. Applications or services within the VPC network can use this IP address to communicate with Services inside the cluster. Below is an example of a Service manifest that creates an internal loadbalancer. You can configure an external loadbalancer the same way by removing the annotation (which creates an internal loadbalancer).

```
apiVersion: v1
kind: Service
metadata:
  name: ilb-service
  annotations:
    cloud.google.com/load-balancer-type: "Internal" #A
  labels:
    app: hello
spec:
  type: LoadBalancer
  selector:
    app: hello
  ports:
  - port: 80
    targetPort: 8080
    protocol: TCP
#B
```

#A Annotation creates an internal Google load balancer
#B Creates a Google load balancer

Many applications, such as RESTful web service APIs, communicate using HTTP(S). You can allow clients external to your VPC network to access this type of application using a Kubernetes [Ingress resource](#). An Ingress resource allows you to map hostnames and URL paths to Services within the cluster. An Ingress resource is associated with one or more [Service](#) objects, each of which is associated with a set of Pods. When you create an Ingress resource, the [GKE Ingress controller](#) creates a [Google Cloud HTTP\(S\) Load Balancer](#) and configures it according to the information in the Ingress and its associated Services. To use Ingress, you must have the HTTP load balancing add-on enabled. GKE clusters have HTTP load balancing enabled by default.

GKE Ingress resources come in two types:

1. [Ingress for external HTTP\(S\) load balancer](#) deploys the [Google Cloud external HTTP\(S\) load balancer](#). This internet-facing load balancer is deployed globally across Google's edge network as a managed and scalable pool of load balancing resources.
2. [Ingress for Internal HTTP\(S\) Load Balancing](#) deploys the [Google Cloud internal HTTP\(S\) load balancer](#). These internal HTTP(S) load balancers are powered by Envoy proxy systems outside of your GKE cluster, but within your VPC network.

HTTP(S) load balancing, configured by Ingress, includes the following features:

1. Flexible configuration for Services. An Ingress defines how traffic reaches your Services and how the traffic is routed to your application. In addition, an Ingress can provide a single IP address for multiple Services in your cluster.
2. Integration with Google Cloud network services
3. Support for multiple TLS certificates. An Ingress can specify the use of multiple TLS certificates for request termination.

When you create the Ingress resource, GKE provisions an [HTTP\(S\) load balancer](#) in the Google Cloud project according to the rules in the Ingress manifest and the associated Service manifests. The load balancer sends a request to a node's IP address at the `NodePort`. After the request reaches the node, the chosen GKE dataplane routes the traffic to the appropriate Pod (for the desired Service). For dataplane v1, the node uses its `iptables` NAT table to choose a Pod. `kube-proxy` manages the `iptables` rules on the node. For dataplane v2, GKE provides this functionality using eBPF and Cilium agents.

The following example Ingress definition routes traffic for `demo.example.com` to a Service named `frontend` on port 80, and `demo-backend.example.com` to a Service named `users` on port 8080.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: demo
spec:
  rules:
    - host: demo.example.com          #A
      http:
        paths:
          - backend:
              serviceName: frontend      #A
              servicePort: 80
    - host: demo-backend.example.com   #B
      http:
        paths:
          - backend:
              serviceName: users         #B
              servicePort: 8080           #B
```

#A Requests to host `demo.example.com` forwarded to Service `frontend` on port 80

#B Requests to host `demo-backend.example.com` forwarded to Service `users` on port 8080

CONTAINER NATIVE LOAD BALANCING

[Container-native load balancing](#) is the practice of load balancing directly to Pod endpoints in GKE using [Network Endpoint Groups \(NEGs\)](#). With Ingress, Service bound traffic is sent from the HTTP loadbalancer to any of the node IP on the NodePort. After the request reaches the node, the GKE dataplane routes the traffic to the desired Pod. This results in extra hops. In some cases, the Pod may not even be running on the node and thus the node sends the request to the node where the desired Pod is running. Additional hops add latency and make the traffic path more complex.

With NEGs, traffic is load balanced from the load balancer directly to the Pod IP as opposed to traversing the nodes. In addition, [Pod readiness gates](#) are implemented to determine the health of Pods from the perspective of the load balancer and not just the Kubernetes in-cluster health probes. This improves overall traffic stability by making the load balancer infrastructure aware of lifecycle events such as Pod startup, Pod loss, or VM loss. These capabilities resolve the above limitations and result in more performant and stable networking.

Container-native load balancing is enabled by default for Services when all of the following conditions are true:

1. For Services created in GKE clusters 1.17.6-gke.7 and up
2. Using VPC-native clusters
3. Not using a Shared VPC
4. Not using GKE Network Policy

For clusters where NEGs are not the default, it is still strongly recommended to use container-native load balancing, but it must be enabled explicitly on a per-Service basis. The annotation should be applied to Services in the following manner:

```
kind: Service
...
annotations:
  cloud.google.com/neg: '{"ingress": true}'      #A
...
```

#A Annotation creates a Network Endpoint Group for Pods in the particular service

In the Service manifest, you must use `type: NodePort` unless you're using [container native load balancing](#). If using container native load balancing, use the `type: ClusterIP`.

SHARED VPC CONSIDERATIONS AND BEST PRACTICES

The GKE Ingress controllers use a Google Cloud service account to deploy and manage Google Cloud resources. When a GKE cluster resides in a service project of a Shared VPC, this service account may not have the rights to manage network resources owned by the host project. The Ingress controller actively manages firewall rules to provide access between load balancers and Pods and also between centralized health checkers and Pods.

There are two ways to manage this.

1. **Manual firewall rule provisioning** - If your security policies only allow firewall management from the host project, then you can provision these firewall rules manually. When deploying Ingress in a Shared VPC, the Ingress resource event provides the specific firewall rule you need to add necessary to provide access.

To manually provision a firewall rule, view the Ingress resource using the `describe` commands as shown below.

```
kubectl describe ingress INGRESS_NAME
```

The output of this command should have the required firewall rule that can be implemented in the host network project.

Events:				
Type	Reason	Age	From	Message
Normal	Sync	9m34s (x237 over 38h)	loadbalancer-controller	Firewall change required by network admin: `gcloud compute firewall-rules update k8s-fw-17--6048d433d4280f11 --description "GCE L7 firewall rule" --allow tcp:30000-32767,tcp:8080 --source-ranges 130.211.0.0/22,209.85.152.0/22,209.85.204.0/22,35.191.0.0/16 --target-tags gke-17-ilb-test-b3a7e0e5-node --project <project>`

2. **Automatic firewall rule provisioning** - An automated approach is to provide the GKE Ingress controller service account the permissions to update firewall rules.

You do this by creating a custom IAM role providing the ability to manage firewall rules in the host network project and then granting this role to the GKE Ingress service account.

Create a custom IAM role with the required permissions.

```
gcloud iam roles create ROLE_NAME \
    --project PROJECT_ID \
    --title ROLE_TITLE \
    --description ROLE_DESCRIPTION \
    --permissions=compute.networks.updatePolicy, compute.firewalls.* \
    --stage GA
```

Grant the custom role to the GKE Ingress Controller service account.

```
gcloud projects add-iam-policy-binding my-project \
    --member=user:SERVICE_ACCOUNT \
    --role=roles/gke-ingress-fw-management
```

MULTI-CLUSTER INGRESS

In some cases, it may be required to run the same service on multiple GKE clusters. Many factors drive multi-cluster topologies, including close user proximity for apps, cluster and regional high availability, security and organizational separation, cluster migration, and data locality. [Multi-cluster Ingress](#) (MCI) is a cloud-hosted multi-cluster Ingress controller for Anthos GKE clusters. It's a Google-hosted service that supports deploying shared load balancing resources across clusters and across regions. Multi-cluster Ingress is designed to meet the load balancing needs of multi-cluster, multi-regional environments. It's a controller for the external HTTP(S) load balancer to provide ingress for traffic coming from the internet across one or more clusters.

Multi-cluster Ingress's multi-cluster support satisfies many use cases including:

1. A single, consistent virtual IP (VIP) for an app, independent of where the app is deployed globally.
2. Multi-regional, multi-cluster availability through health checking and traffic failover.
3. Proximity-based routing through public Anycast VIPs for low client latency.

4. Transparent cluster migration for upgrades or cluster rebuilds.

Multi-cluster Ingress is an Ingress controller that programs the external HTTP(S) load balancer using [network endpoint groups](#) (NEG). When you create a MultiClusterIngress resource, GKE deploys Compute Engine load balancer resources and configures the appropriate Pods across clusters as backends. The NEG are used to track Pod endpoints dynamically so the Google load balancer has the right set of healthy backends.

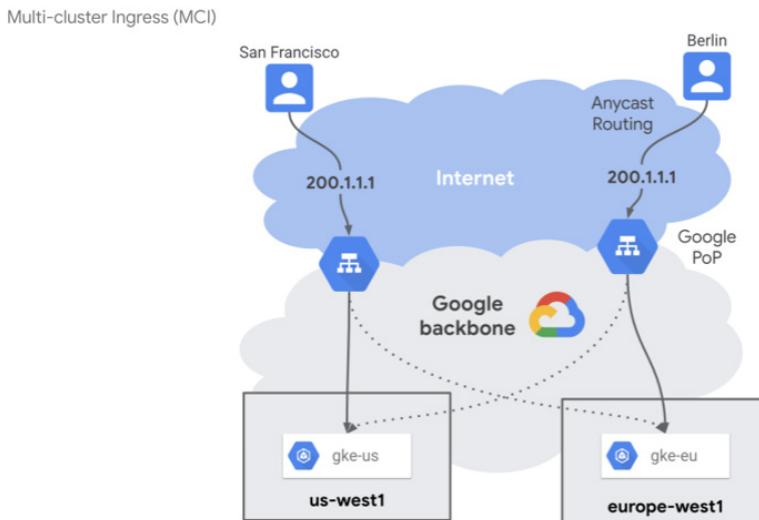


Figure 12.8 Multi-cluster ingress to multiple GKE clusters in GCP

Multi-cluster Ingress uses a centralized Kubernetes API server to deploy Ingress across multiple clusters. This centralized API server is called the `config cluster`. Any GKE cluster can act as the config cluster. The config cluster uses two custom resource types: `MultiClusterIngress` and `MultiClusterService`. By deploying these resources on the config cluster, the Anthos Ingress Controller deploys load balancers across multiple clusters.

The following concepts and components make up Multi-cluster Ingress:

1. **Anthos Ingress controller** - This is a globally distributed control plane that runs as a service outside of your clusters. This allows the lifecycle and operations of the controller to be independent of GKE clusters.
2. **Config cluster** - This is a chosen GKE cluster running on Google Cloud where the `MultiClusterIngress` and `MultiClusterService` resources are deployed. This is a centralized point of control for these multi-cluster resources. These multi-cluster resources exist in and are accessible from a single logical API to retain consistency across all clusters. The Ingress controller watches the config cluster and reconciles the load balancing infrastructure.

3. **Environ** - An [environ](#) is a domain that groups clusters and infrastructure, manages resources, and keeps a consistent policy across them (for more details about Environs, see Chapter N: Anthos, the one single glass-of-pane UX)
4. MCI uses the concept of environs for how Ingress is applied across different clusters. Clusters that you register to an environ become visible to MCI, so they can be used as backends for Ingress. Environs possess a characteristic known as *namespace sameness* which assumes that resources with the identical names and same namespace across clusters are considered to be instances of the same resource. In effect, this means that Pods in the `ns1` namespace with the labels `app: foo` across different clusters are all considered part of the same pool of application backends from the perspective of Multi-cluster Ingress. The figure below shows an example of two services `foo` and `bar` running on two clusters being loadbalanced by MCI.

Multi-cluster Ingress (MCI): Environ and namespace sameness

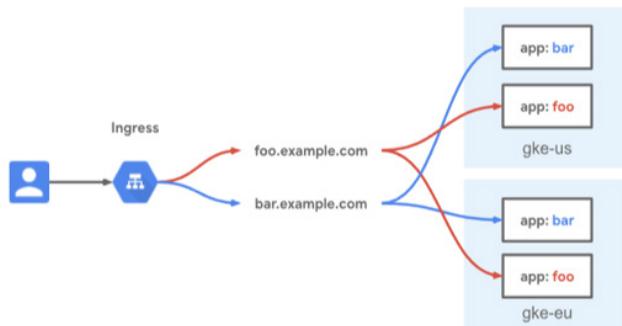


Figure 12.9 Multi-cluster ingress: Environ and namespace sameness

5. **Member cluster** - Clusters registered to an environ are called member clusters. Member clusters in the environ comprise the full scope of backends that MCI is aware of.

After the config cluster has been configured, you create the two custom resources, `MultiClusterIngress` and `MultiClusterService` for your multi-cluster Service. Note the resource names are comparatively similar to `Service` and `Ingress`, required for Ingress in a single cluster.

Below are examples of these resources which are deployed to the config cluster.

```

# MulticloudService with cluster selector
apiVersion: networking.gke.io/v1beta1
kind: MultiClusterService
metadata:
  name: foo
  namespace: blue
spec:
  template:
    spec:
      selector:
        app: foo
      ports:
        - name: web
          protocol: TCP
          port: 80
          targetPort: 80
  clusters:
    - link: "europe-west1-c/gke-eu"           #A
    - link: "asia-northeast1-a/gke-asia-1"     #B

```

#A MultiClusterService spec looks similar to Service spec with a clusters section added to define Service in multiple clusters.

#B GKE Cluster URI links for Service running in multiple clusters

MulticloudIngress

```

apiVersion: networking.gke.io/v1alpha1
kind: MultiClusterIngress
metadata:
  name: foobar-ingress
  namespace: blue
spec:
  template:
    spec:
      backend:
        serviceName: default-backend
        servicePort: 80
      rules:
        - host: foo.example.com
          backend:
            serviceName: foo           #A
            servicePort: 80
        - host: bar.example.com
          backend:
            serviceName: bar           #A
            servicePort: 80

```

#A MultiClusterIngress spec is similar to the Ingress spec except that it points to a MultiClusterService (instead of a Service)

Note that the MulticloudService includes a cluster selector stanza at the bottom. Removing this sends Ingress traffic to all member clusters. Adding a cluster selector may be useful if you want to remove MCI traffic from a specific cluster (or clusters), for example, if you are performing upgrades or maintenance to a cluster. If the clusters stanza is present in the MulticloudService resource, Ingress traffic is sent to only the clusters available in

the list. Clusters are explicitly referenced by <region | zone>/<name>. Member clusters within the same environ and region should have unique names so that there are no naming collisions. Like a Service, a MCS is a selector for Pods but it is also capable of selecting *labels* and *clusters*. The pool of clusters that it selects across are called member clusters, and these are all the clusters registered to the environ. This MCS deploys a derived Service in all member clusters with the selector `app: foo`. If `app: foo` Pods exist in that cluster then those Pod IPs will be added as backends for the MCI.

ANTHOS ON PREM (ON VMWARE)

Anthos on Prem clusters on VMware use an *Island Mode* configuration in which Pods can directly talk to each other within a cluster, but cannot be reached from outside the cluster. This configuration forms an "island" within the network that is not connected to the external network. Clusters form a full node-to-node mesh across the cluster nodes, allowing Pod to reach other Pods within the cluster directly.

NETWORKING REQUIREMENTS

Anthos on Prem clusters are installed using an Admin workstation VM which contains all the tools and configurations required to deploy Anthos on Prem clusters. The admin workstation VM deploys an [admin cluster](#). Admin cluster deploys one or more [user clusters](#). Your applications run on user clusters. The admin cluster manages the deployment and life cycle of multiple user clusters. You do not run your applications on the admin cluster.

In your initial installation of Anthos on Prem, you create these virtual machines (VMs):

1. One VM for an admin workstation
2. Four VMs for an admin cluster
3. Three VMs for a user cluster. You can create additional user clusters as well as larger user clusters if needed.
4. In your vSphere environment, you must have a network that can support the creation of these VMs. Your network must also be able to support a vCenter Server and a load balancer. Your network needs to support outbound traffic to the internet so that your admin workstation and your cluster nodes can fetch GKE on-prem components and call certain Google services. If you want external clients to call services in your GKE on-prem clusters, your network must support inbound traffic from the internet.

IP address architecture and allocation is discussed in the next section.

ANTHOS ON PREM CLUSTER IP ALLOCATION

The following IP addresses are required for Anthos on Prem on VMWare clusters.

1. **Node IP** - DHCP or statically-assigned IP addresses for the nodes (virtual machines or VMs). Must be routable within the data center. You can manually assign static IPs. Node IP addressing depends upon the implementation of Load Balancer in the Anthos on Prem cluster. If the cluster is configured with [integrated mode load balancing](#) or [bundled mode load balancing](#), you can use DHCP or statically-assigned IP addresses for the nodes. If the cluster is configured with [manual mode load balancing](#), you must use static IP addresses for nodes. In this case, ensure enough IP addresses are set aside to account for cluster growth. Load balancing modes are discussed in detail in the next section.
2. **Pod IP CIDR** - Non-routable CIDR block for all Pods in the cluster. From this range, smaller /24 ranges are assigned per node. If you need an N node cluster, ensure this block is large enough to support N x /24 blocks.
3. **Services IP** - In Island Mode, similar to Pod CIDR block, so only used within the cluster. Any private CIDR block which does not overlap with the nodes, VIPs, or Pod CIDR block. You can share the same block among multiple clusters. The size of the block determines the number of services. In addition to your Services, a block of Service IP addresses are used for cluster controlplane Services. One Service IP is needed for the ingress service, and ten or more IPs for Kubernetes services like cluster DNS, etc.

EGRESS TRAFFIC AND CONTROLS

All egress traffic from the Pod to targets outside the cluster is NAT'd by the node IP. You can use `NetworkPolicy` to further control the flow of traffic between Pods within a cluster as well as traffic egressing Pods. These policies are enforced by Calico running inside each cluster. At the Service layer, you can use `EgressPolicy` through [Anthos Service Mesh](#) (ASM) to control what traffic exits the clusters. In this case, an [Envoy proxy](#) called the `istio-egressgateway` exists at the perimeter of the service mesh through which all egress traffic flows.

LOAD BALANCERS

Anthos on Prem clusters provide two ways to access Services from outside of the cluster: load balancers and Ingress. This section addresses load balancers and different modes of implementations.

Anthos on Prem clusters can run in one of three load balancing modes: *integrated*, *manual*, or *bundled*.

1. **Integrated mode** - With integrated mode, Anthos on Prem uses an F5 BIG-IP load balancer. Customer provides the F5 BIG-IP load balancer with the appropriate licensing. You need to have a [user role](#) that has sufficient permissions to set up and manage the F5 load balancer. Either the Administrator role or the Resource Administrator role is sufficient. For more information, see [F5 BIG-IP account permissions](#). You set aside multiple [Virtual IP addresses](#) (VIPs) to be used for Services that are configured of type `Loadbalancer`. Each Service requires one VIP. The number of VIPs required depends upon the number of Service of type `Loadbalancer`. You specify these VIPs in your cluster configuration file, and Anthos on Prem automatically configures the F5 BIG-IP load balancer to use the VIPs.

The advantages of integrated mode is you get to use an enterprise grade load balancer and the configuration is mostly automated. This mode is also opinionated in that it requires an F5 load balancer which may incur additional licensing and support cost.

2. **Manual mode** - With manual mode, Anthos on Prem uses a load balancer of your choice. Manual load balancing mode requires additional configuration than with integrated mode. You need to manually configure the VIPs to be used for Services. With manual load balancing , you cannot run Services of type `LoadBalancer`. Instead, you can create Services of type `NodePort` and manually configure your load balancer to use them as backends. You must specify the `NodePort` values to be used for these Services. You can choose values in the 30000 - 32767 range. For more information, see [Setting aside nodePort values](#).

The advantages of manual mode is you get absolute freedom in terms of what load balancer you choose. On the other hand, the configuration is manual which may result in increased operational overhead.

3. **Bundled mode** - In bundled load balancing mode, Anthos on Prem provides and manages the load balancer. Unlike integrated mode, there is no license required for a load balancer, and the amount of setup that you have to do is minimal. The bundled load balancer that GKE on-prem provides is the [Seesaw load balancer](#). Seesaw load balancers run as VMs inside VMWare. We recommend that you use vSphere 6.7+ and Virtual Distributed Switch (VDS) 6.6+ for bundled load balancing mode. You can run Seesaw load balancer in HA and non-HA mode. In HA mode, two Seesaw VMs are configured. In non-HA mode, a single Seesaw VM is configured.

The advantages of bundled mode is a single team can be in charge of both cluster creation and load balancer configuration. For example, a cluster administration team would not have to rely on a separate networking team to acquire, run, and configure the load balancer ahead of time. Another advantage is that the configuration is completely automated. Anthos on Prem configures the Service VIPs automatically on the load balancer.

Anthos on VMWare clusters can run Services of type [LoadBalancer](#) as long as a `loadBalancerIP` field is configured in the Service's specification. In the `loadBalancerIP`

field, you need to provide the VIP that you want to use. This will be configured on F5, pointing to the `NodePorts` of the Service.

An example of a Service manifest is as follows. You can access a Service running inside Anthos on Prem cluster called `frontend` via the SERVICE VIP.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: guestbook
    name: frontend
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: guestbook
  type: LoadBalancer
  loadBalancerIP: [SERVICE VIP] #A
```

#A Load balancer IP address is defined in the Service spec

In addition to Service VIPs, a Controlplane VIP for the Kubernetes API server is required by the load balancer. And lastly, there is an ingress controller running inside each Anthos on Prem cluster. The ingress controller service also has a VIP called the Ingress VIP. Services exposed via ingress use the Ingress VIP to access Kubernetes Services.

The Anthos on Prem high level load balancing architecture looks as follows.

Anthos on Prem: Load balancers

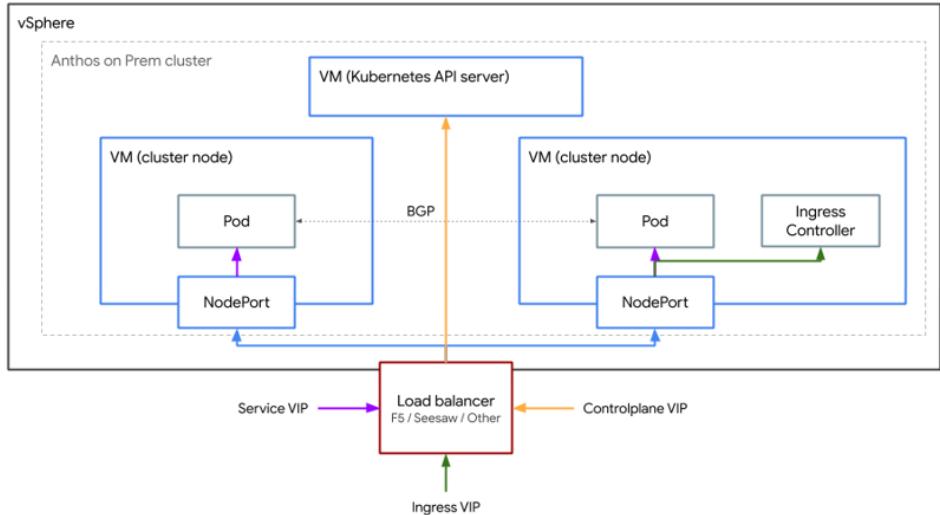


Figure 12.10 Anthos on prem: Load balancer network architecture

The following table summarizes what you must do to prepare for load balancing in various modes.

	Integrated / Bundled mode	Manual mode
Choose VIPs before you create your clusters.	Yes	Yes
Choose node IP addresses before you create your clusters.	No, if using DHCP. Yes, if using static IP addresses.	Yes
Choose nodePort values before you create your clusters.	No	Yes
Manually configure your load balancer	No	Yes

INGRESS

Anthos on Prem includes an L7 load balancer with an Envoy-based ingress controller that handles Ingress object rules for ClusterIP Services deployed within the cluster. The ingress controller itself is exposed as a `NodePort` Service in the cluster. The ingress NodePort service can be reached through a L3/L4 F5 load balancer. The installation configures a [virtual IP](#)

[address](#) (Ingress VIP) (with port 80 and 443) on the load balancer. The VIP points to the ports in the NodePort Service for the Ingress controller. This is how external clients can access services in the cluster.

In order to expose a Service via Ingress, you must create a DNS A record to point the DNS name to the Ingress VIP. Then you can create a Service and an Ingress resource for the Service. For example, let's say you want to expose a `frontend` Service of a sample guestbook application.

First, create a DNS A record for the guestbook application pointing to the Ingress VIP.

```
*.guestbook.com    A    [INGRESS_VIP]
```

Next, create a Service for the `frontend` Deployment. Note that the Service is of type ClusterIP.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: guestbook
    name: frontend
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: guestbook
  type: ClusterIP          #A
```

#A For Ingress, Service type is ClusterIP (instead of LoadBalancer)

Finally, create the Ingress rule.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: frontend
  labels:
    app: guestbook
spec:
  rules:
    - host: www.guestbook.com
      http:
        paths:
          - backend:
              serviceName: frontend    #A
              servicePort: 80           #A
```

#A Ingress rule points to the Service name and port

ANTHOS ON BARE METAL

Anthos on bare metal is a GCP supported Anthos GKE implementation that is deployed on bare metal servers. Anthos on bare metal eliminates the need for a virtualization layer or a hypervisor. All cluster nodes and API servers run directly on bare metal servers

Anthos on Bare Metal deployment models and networking architecture are described in detail in the chapter dedicated to it.

ANTHOS ON AWS

Anthos clusters on AWS (GKE on AWS) is hybrid cloud software that extends Google Kubernetes Engine (GKE) to Amazon Web Services (AWS). Anthos on AWS uses AWS resources such as [Elastic Compute Cloud \(EC2\)](#), [Elastic Block Storage \(EBS\)](#), and [Elastic Load Balancer \(ELB\)](#).

There are two components to Anthos clusters on AWS.

1. **Management service** - an environment that can install and update your user clusters, uses the AWS API to provision resources.
2. **User clusters** - Anthos on AWS clusters where you run your containerized applications.

NETWORKING REQUIREMENTS

Both the management service and the user clusters are deployed inside an AWS VPC on EC2 instances. You can create your management service in a [dedicated AWS VPC](#) or an [existing AWS VPC](#). You need a management service in every AWS Virtual Private Cloud (VPC) where you run Anthos on AWS user clusters. The management service is installed in one AWS Availability Zone. You only need one management service per VPC; a management service can manage multiple user clusters.

A user cluster consists of two components: a controlplane or the Kubernetes API server and node pools where your applications run. The management service's primary component is a Cluster Operator. The Cluster Operator is a [Kubernetes Operator](#) that creates and manages your `AWSClusters` and `AWSNodePools`. The Cluster Operator stores configuration in an etcd database with storage persisted on an AWS EBS volume.

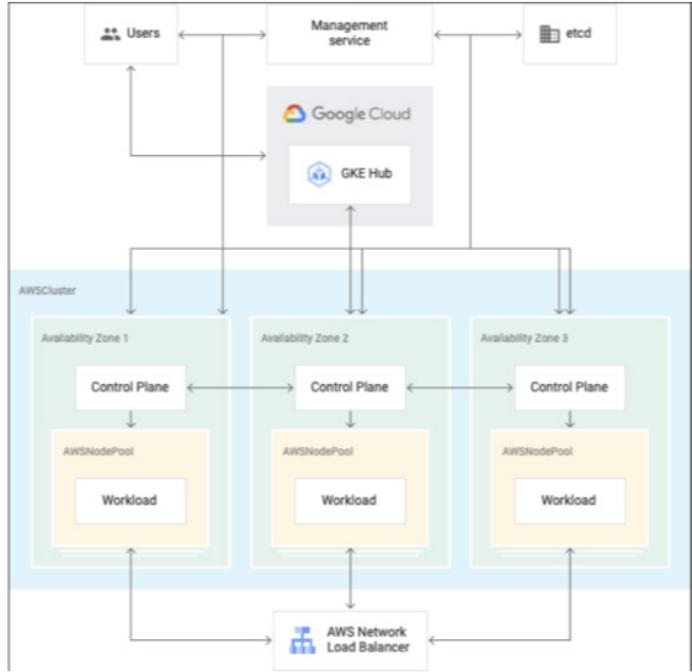
`AWSClusters` resource creates and manages the user clusters' controlplane and `AWSNodePools` resource creates and manages the user clusters' node pools.

When you install a management cluster into a [Dedicated VPC](#), Anthos on AWS creates control plane replicas in every zone you specify in `dedicatedVPC.availabilityZones`. When you install a management cluster into [existing infrastructure](#) Anthos on AWS creates an `AWSCluster` with three control plane replicas in the same availability zones. Each replica belongs to its own [AWS Auto Scaling group](#) which restarts instances when they are terminated. The management service places the control planes in a private subnet behind an AWS Network Load Balancer (NLB). The management service interacts with the control plane using NLB.

Each control plane stores configuration in a local etcd database. These databases are replicated and set up in a [stacked high availability topology](#).

One control plane manages one or more `AWSNodePools`.

Anthos on AWS Arch

**Figure 12.11** Anthos on AWS architecture

The following VPC resources are required when creating Anthos on AWS clusters in a Dedicated VPC.

- VPC CIDR Range** - the total [CIDR](#) range of IP addresses for the AWS VPC that anthos-gke creates. For example, 10.0.0.0/16.
- Availability Zones** - the AWS EC2 [availability zones](#) where you want to create nodes and control planes.
- Private CIDR Blocks** - the CIDR block for your private subnet. Anthos on AWS components such as the management service run in the private subnet. This subnet must be within the VPC's CIDR range specified in `vpcCIDRBlock`. You need one subnet for each availability zone.

4. **Public CIDR Block** - the CIDR blocks for your public subnet. You need one subnet for each availability zone. The public subnet exposes cluster services such as load balancers to the security groups and address ranges specified in AWS [network ACLs](#) and [security groups](#).
5. **SSH CIDR Block** - the CIDR block that allows inbound SSH to your [bastion host](#). You can use IP ranges for example 203.0.113.0/24. If you want to allow SSH from any IP address, use 0.0.0.0/0. When you create a management service using the default settings, the control plane has a private IP address. This IP address isn't accessible from outside the AWS VPC. You can access the management service with a [bastion host](#) or using another connection to the AWS VPC such as a VPN or [AWS Direct Connect](#).

The following VPC resources are required when creating Anthos on AWS clusters in an existing VPC.

1. At least one public subnet.
2. At least one private subnet.
3. An internet gateway with a route to the public subnet.
4. A NAT gateway with a route to the private subnet.
5. [DNS hostnames](#) enabled.
6. No custom value of `domain-name` in your DHCP options sets. Anthos on AWS does not support values other than the [default EC2 domain names](#).
7. Choose or create an AWS [security group](#) that allows SSH (port 22) inbound from the security groups or IP ranges where you will be managing your Anthos clusters on AWS installation.

ANTHOS ON AWS CLUSTER IP ALLOCATION

The management service creates user clusters. Management service uses a CLuster Operator with resources `AWSClusters` and `AWSNodePools` to create the user clusters' controlplane and node pools respectively. The IP address per user cluster is defined in the `AWSCluster` resource.

Below is an example of an `AWSCluster` resource.

```

apiVersion: multicloud.cluster.gke.io/v1
kind: AWSCluster
metadata:
  name: CLUSTER_NAME
spec:
  region: AWS_REGION
  networking:                                     #A
    vpcID: VPC_ID
    podAddressCIDRBlocks: POD_ADDRESS_CIDR_BLOCKS
    serviceAddressCIDRBlocks: SERVICE_ADDRESS_CIDR_BLOCKS
    ServiceLoadBalancerSubnetIDs: SERVICE_LOAD_BALANCER_SUBNETS
  controlPlane:                                     #B
    version: CLUSTER_VERSION
    instanceType: AWS_INSTANCE_TYPE
    keyName: SSH_KEY_NAME
    subnetIDs:
      - CONTROL_PLANE_SUBNET_IDS
    securityGroupIDs:
      - CONTROL_PLANE_SECURITY_GROUPS
    iamInstanceProfile: CONTROL_PLANE_IAM_ROLE
    rootVolume:
      sizeGiB: ROOT_VOLUME_SIZE
    etcd:
      mainVolume.sizeGiB: ETCD_VOLUME_SIZE
    databaseEncryption:
      kmsKeyARN: ARN_OF_KMS_KEY
    hub: # Optional
      membershipName: ANTHOS_CONNECT_NAME
    workloadIdentity: # Optional
      oidcDiscoveryGCSBucket: WORKLOAD_IDENTITY_BUCKET

```

#A Anthos on AWS cluster networking values are defined

#B Anthos on AWS cluster control plane values are defined

You define the required IP addresses in the `networking` section.

Anthos on AWS cluster requires the following IP addresses.

1. **Node IP** - Node IPs are assigned to the EC2 instances as they are created. Each EC2 instance is assigned a single IP from the private subnet in its availability zone. These addresses are defined in the management service spec.
2. **Pod IP CIDR** - the CIDR range of IPv4 addresses used by the cluster's Pods. . The range must be within your VPC CIDR address range, but not part of a subnet.
3. **Services IP** - the range of IPv4 addresses used by the cluster's Services. . The range must be within your VPC CIDR address range, but not part of a subnet.

EGRESS TRAFFIC AND CONTROLS

All egress traffic from the Pod to targets outside the cluster is NAT'd by the node IP. You can use `NetworkPolicy` to further control the flow of traffic between Pods within a cluster as well as traffic egressing Pods. These policies are enforced by Calico running inside each cluster. At the Service layer, you can use `EgressPolicy` through [Anthos Service Mesh](#) (ASM) to control what traffic exits the clusters. In this case, an [Envoy proxy](#) called the `istio-egressgateway` exists at the perimeter of the service mesh through which all egress traffic flows.

In addition, you can control the traffic flow at the AWSNodePools SecurityGroup layer. With SecurityGroups, you can further allow or deny traffic for both ingress and egress.

LOAD BALANCERS

When you create a Service of type LoadBalancer, a Anthos on AWS controller configures a [Classic](#) or [Network](#) ELB on AWS. Anthos on AWS requires tags on subnets that contain load balancer endpoints. Anthos on AWS automatically tags all subnets specified in the [spec.Networking.ServiceLoadBalancerSubnetIDs](#) field of the [AWSCluster](#) resource.

To create the tag, get the subnet ID of the load balancer subnets. Use the aws command line utility to create the tag on the subnets as follows. For multiple subnets, make sure the subnet IDs are separated by spaces.

```
aws ec2 create-tags \
--resources [SUBNET_IDs] \
--tags Key=kubernetes.io/cluster/$CLUSTER_ID,Value=shared
```

You need a tag for every user cluster on the subnet.

You can create internal and external load balancers. Internal load balancers are created on the private subnets while external load balancers are created on the public subnets. You can create either type of load balancers using either a Classic or a Network load balancer. For more information on the differences between load balancer types, see [Load balancer types](#) in the AWS documentation.

Different types of load balancers are created using annotations.

Consider the following Service spec.

```
apiVersion: v1
kind: Service
metadata:
  name: my-lb-service
spec:
  type: LoadBalancer
  selector:
    app: products
    department: sales
  ports:
  - protocol: TCP
    port: 6000
    targetPort: 50001
```

This resource creates a Classic public load balancer for the Service.

To create a public Network load balancer, add the following annotation to the spec above.

```
...
metadata:
  name: my-lb-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-type: nlb  #A
...
#A Annotation creates a Classic public load balancer in AWS exposing Service
```

To create a private Classic load balancer, add the following annotation to the Service spec.

```
...
metadata:
  name: my-lb-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-internal: "true" #A
...
#A Annotation creates an internal load balancer in AWS exposing Service
```

Finally to create a private Network load balancer, add both annotations to the Service spec.

```
...
metadata:
  name: my-lb-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-type: nlb          #A
    service.beta.kubernetes.io/aws-load-balancer-internal: "true"   #A
...
#A Both annotations together create an internal network load balancer in AWS
```

INGRESS

There are two ways to use Ingress on Anthos on AWS clusters.

1. **Application Load Balancer** - [Application load balancer](#) (ALB) is an AWS managed L7 HTTP load balancer. After the load balancer receives a request, it evaluates the listener rules in priority order to determine which rule to apply, and then selects a target from the target group for the rule action. This method uses an alb-ingress-controller installed in the Anthos on AWS cluster with proper permissions to create ALBs for ingress.
2. **ASM Ingress** - You can install Anthos Service Mesh (ASM) on an Anthos on AWS cluster and use ASM ingress. ASM ingress, a Service called `istio-ingressgateway`, is an L7 Envoy proxy that lives at the perimeter of the service mesh. The `istio-ingressgateway` Service itself is exposed using ELB described in the previous section. All L7 load balancing and routing is handled by the `istio-ingressgateway`.

EXPOSING SERVICES USING INGRESS

In order to use the ALB method, follow the instruction [here²³](#) and deploy the `alb-ingress-controller` to the Anthos on AWS cluster. The `alb-ingress-controller` is a Deployment that runs on the Anthos on AWS cluster with proper AWS credentials and Kubernetes RBAC permission to create the rules and resources required to create an ALB for Ingress.

You can now create an Ingress resource with proper annotations to create an ALB and the required resources for your Service.

²³ https://cloud.google.com/anthos/gke/docs/aws/how-to/loadbalancer-alb#before_you_begin

Here is an example of a Service spec. Note that the type of the Service is `NodePort`. The Service must be of type `NodePort`.

```
apiVersion: v1
kind: Service
metadata:
  name: "service-2048"
  namespace: "2048-game"
spec:
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
  type: NodePort
  selector:
    app: "2048"
```

And the Ingress resource to expose this Service using an ALB. Note the two annotations which configures an internet-facing ALB.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: "2048-ingress"
  namespace: "2048-game"
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/scheme: internet-facing #A
  labels:
    app: 2048-ingress
spec:
  rules:
    - http:
        paths:
          - path: /*
            backend:
              serviceName: "service-2048"
              servicePort: 80
```

#A Annotations create an internet facing application load balancer in AWS

You can also use ASM Ingress to expose your Services. To use, ASM, follow the steps in the footnote link²⁴ to install ASM on your Anthos on AWS cluster. Once ASM is installed you should see the `istio-ingressgateway` Deployment and Service in the `istio-system` namespace.

An example of the Service spec looks like the following. Note that the Service type is `ClusterIP` instead of `NodePort` used in the ALB method. The reason is that in the case of ASM, the L7 proxy runs inside the cluster, whereas the ALB is a managed HTTP load balancer that runs outside of the cluster.

²⁴ https://cloud.google.com/anthos/gke/docs/aws/how-to/ingress#top_of_page

```

apiVersion: v1
kind: Service
metadata:
  labels:
    app: hello-app
    name: hello-app
spec:
  type: ClusterIP
  selector:
    app: hello-app
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080

```

And the Ingress resource looks like the following. Note the annotation which uses ASM for ingress.

```

apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: istio          #A
  labels:
    app: hello-app
    name: hello-app
spec:
  rules:
    - host:
        http:
          paths:
            - backend:
                serviceName: hello-app
                servicePort: 8080

```

#A Annotation uses the Istio ingress controller

ANTHOS ATTACHED CLUSTERS

The final type of Anthos clusters are Anthos attached clusters. Attaching clusters lets you view your existing Kubernetes clusters in the Google Cloud Console along with your Anthos clusters, and enable a subset of Anthos features on them, including configuration with Anthos Config Management.

You can attach any [conformant Kubernetes cluster](#) to Anthos and view it in the Cloud Console with your Anthos clusters.

Regardless of where your clusters are, you need to register any clusters that you want to use with Anthos with your project's environ by using [Connect](#). An environ provides a unified way to view and manage multiple clusters and their workloads as part of Anthos. You can find out more about environs and the functionality that they enable in our [Environs guide](#).

NETWORKING REQUIREMENTS

To successfully register your cluster, you need to ensure that the domains below are reachable from your Kubernetes cluster:

1. `cloudresourcemanager.googleapis.com` resolves metadata regarding the Google Cloud project the cluster is being connected to.
2. `oauth2.googleapis.com` to obtain short-lived OAuth tokens for agent operations against `gkeconnect.googleapis.com`.
3. `gkeconnect.googleapis.com` to establish the channel used to receive requests from Google Cloud and issues responses.
4. `gkehub.googleapis.com` to create Google Cloud-side Hub membership resources that corresponds to the cluster you're connecting with Google Cloud.
5. `www.googleapis.com` to authenticate service tokens from incoming Google Cloud service requests.
6. `gcr.io` to pull GKE Connect Agent image.

If you're using a [proxy](#) for Connect, you must also update the proxy's allowlist with these domains.

If you use [gcloud](#) to register your Kubernetes cluster, these domains also need to be reachable in the environment where you run the gcloud commands.

You only need outbound connectivity on port 443 to these domains. No inbound connections are required to register Anthos attached clusters. You can also use VPC Service controls for additional security.

USING VPC SERVICE CONTROLS

If you use [VPC Service Controls](#) for additional data security in your application, ensure that the following services are in your service perimeter:

1. Resource Manager API (`cloudresourcemanager.googleapis.com`)
2. GKE Connect API (`gkeconnect.googleapis.com`)
3. GKE Hub API (`gkehub.googleapis.com`)

You also need to set up [private connectivity](#) for access to the relevant APIs. You can find out how to do this in [Setting up private connectivity](#).

You can learn more about GKE Connect connectivity and security features [here](#).

12.2.2 Anthos GKE IP address management

With the exception of Anthos GKE on GCP, all other Anthos clusters operate in an *Island Mode* configuration in which Pods can directly talk to each other within a cluster, but cannot be reached from outside the cluster. This configuration forms an "island" within the network that is not connected to the external network. This allows you to create multiple Anthos clusters using the same IP addressing.

For Anthos clusters in Island Mode, IP address management and IP exhaustion is not an issue. You can standardize on an IP schema and use the same schema for all clusters.

GCP recommends running Anthos GKE on GCP clusters in VPC-native mode. In VPC native mode, all IP addresses used by any cluster is a real VPC IP address. This means with VPC

native clusters, you cannot use overlapping IP addresses and you must use unique subnets for every cluster.

Recall that each Anthos GKE on GCP cluster requires three IP ranges.

1. **Node IP** - These IPs are assigned to GCE instances or nodes belonging to clusters. One IP is required per node. These IP addresses are automatically assigned using the primary subnet.
2. **Pod IP CIDR** - These IPs are assigned to every Pod that runs inside the cluster. A large subnet is assigned to the cluster. The cluster controlplane divides this large subnet into smaller subnets and each subnet (of equal size) is assigned to every node. For example, you can have a Pod IP CIDR of 10.0.0.0/16 and the cluster controlplane assigns subnets of size /24 (from the Pod IP CIDR block) to each node starting with 10.0.0.0/24 for the first node, 10.0.1.0/24 for the second node and so on.
3. **Service IP CIDR** - These IPs are assigned to Services running inside the cluster. Every Service of type ClusterIP requires one IP address.

Let's address these one at a time in a bit more detail.

Node IP

In order to determine the size of the Node IP pool, you must know the following:

1. Number of clusters in a GCP region
2. Number of maximum nodes per cluster

If you have equal sized clusters, you can simply multiply the two numbers to get the total number of maximum nodes required to run in that region.

```
total number of nodes = number of clusters x max number of nodes per cluster
```

You can then determine the host bits you need for the Node IP subnet from the table below.

Nodes required	Host bits for Nodes
1-4	3 (or /29)
5-12	4 (or /28)
13-28	5 (or /27)
29-60	6 (or /26)
61-124	7 (or /25)
125-252	8 (or /24)
253-508	9 (or /23)
509-1020	10 (or /22)
1021-2044	11 (or /21)
2045-4092	12 (or /20)
4093-8188	13 (or /19)

You can use a single subnet for multiple GKE clusters.

Pod IP CIDR

In order to determine the Pod IP CIDR, determine the maximum number of Pods per Node you need in your cluster over its lifetime. If you cannot determine the maximum number you need, use the [quota limit](#) of 110 Pods per Node as the maximum. Use the following table to determine the host bits needed for the required number of Pods.

Pods-per-Node count	Host bits for Pods
1-8	4
9-16	5
17-32	6
33-64	7
65-110	8

To calculate the Pod IP CIDR block, you need the host bits for nodes and pods and use the following formula.

```
Pod IP CIDR block netmask = 32 - (host bits for Nodes + host bits for Pods)
```

For example, let's assume that you need 110 pods per node and the total number of nodes across all GKE clusters in the region is 5000. First determine the host bits for nodes using the table in the previous section. This would be 13. Then determine the host bits for pods using the table above. This would be 8. Then using the formula, your Pod IP CIDR block netmask needs to be.

```
Pod IP CIDR block netmask = 32 - (13 + 8) = 11
```

You would need a subnet with a mask of /11.

SERVICES IP

In order to calculate the Service IP CIDR, determine the maximum number of cluster IP addresses you need in your cluster over its lifetime. Every Service requires one cluster IP. You cannot share Service IP subnets between clusters. This means you need a different Service IP subnet per cluster.

Once you know the maximum number of Services in a cluster, you can use the table below to get the subnet mask you need.

Number of cluster IP addresses	Netmask
1-32	/27
33-64	/26
65-128	/25
129-256	/24
257-512	/23
513-1,024	/22
1,025-2,048	/21
2,049-4,096	/20
4,097-8,192	/19
8,193-16,384	/18
16,385-32,768	/17
32,769-65,536	/16

CONFIGURING PRIVATELY USED PUBLIC IPs FOR ANTHOS GKE

From the previous section, you can see that in very large GKE environments, you may run into IP exhaustion. The biggest source of IP exhaustion in large GKE environments is the Pod IP CIDR block. GCP VPCs use the RFC1918 address space for networking resources. In large environments, RFC1918 space might not be sufficient to configure Anthos. This is specially a concern for managed service providers that deliver their managed services to many tenants on Anthos.

One way to mitigate address exhaustion is to use privately used public IP (PUPPI) addresses for the GKE Pod CIDR block. PUPPIs are any public IP addresses not owned by Google that a customer can use privately on Google Cloud. The customer doesn't necessarily own these addresses.

The following diagram shows a company (producer) that offers a managed service to a customer (consumer).

Anthos GKE using PUPI Addressing

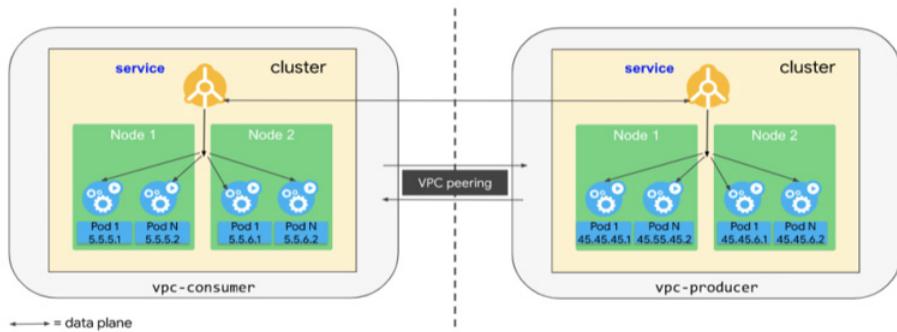


Figure 12.12 Anthos GKE using privately used public IP (PUPI) addressing

This setup involves the following considerations:

1. **Primary CIDR block:** A non-PUPI CIDR block that is used for nodes and ILB and must be non-overlapping across VPCs.
2. **Producer secondary CIDR block:** A PUPI CIDR block that is used for Pods (for example, 45.45.0.0/16).
3. **Consumer secondary CIDR block:** Any other PUPI CIDR block on the customer side (for example, 5.5/16).

The company's managed service is in the producer VPC (`vpc-producer`) and is built on an Anthos GKE deployment. The company's GKE cluster uses the PUPI 45.0.0.0/8 CIDR block for Pod addresses. The customer's applications are located in the consumer VPC (`vpc-consumer`). The customer also has an Anthos GKE installation. The GKE cluster in the consumer VPC uses the PUPI 5.0.0.0/8 CIDR block for Pod addresses. The two VPCs are peered with one another. Both VPCs use the RFC 1918 address space for node, service, and load balancing addresses.

By default, the consumer VPC (`vpc-consumer`) exports all RFC 1918 to the producer VPC (`vpc-producer`). Unlike RFC 1918 private addresses and extended private addresses (CGN, Class E), PUPIs aren't automatically advertised to VPC peers by default. If the `vpc-consumer` Pods must communicate with `vpc-producer`, the consumer must enable the VPC peering connection to export PUPI addresses. Likewise, the producer must configure the producer VPC to import PUPI routes over the VPC peering connection.

The `vpc-consumer` address space that is exported into `vpc-producer` must not overlap with any RFC 1918 or PUPI address used in `vpc-producer`. The producer must inform the consumer what PUPI CIDR blocks the managed service uses and ensure that the consumer isn't using these blocks. The producer and consumer must also agree and assign non-

overlapping address space for internal load balancing (ILB) and node addresses in vpc-producer.

PUPIs don't support service networking.

In most cases, resources in vpc-consumer communicate with services in vpc-producer through ILB addresses in the producer cluster. If the producer Pods are required to initiate communication directly with resources in vpc-consumer, and PUPI addressing doesn't overlap, then the producer must configure the producer VPC to export the PUPI routes over the VPC peering connection. Likewise, the consumer must configure the VPC peering connection to import routes into vpc-consumer. If the consumer VPC already uses the PUPI address, then the producer should instead configure the IP masquerade feature and hide the Pod IP addresses behind the producer node IP addresses.

The example above shows a more complex producer / consumer model. You can simply use this in a single project model. This would free up RFC1918 space that may otherwise be used for Pod IP CIDR.

Learn more about configuring privately used public IPs for GKE [here](#).

12.3 Anthos Multi-cluster Networking

This section addresses mechanisms for connecting Services running across multiple clusters.

Every hybrid and multi-cloud Anthos architecture, by definition, has more than one cluster. For example, you have Anthos GKE clusters running in GCP and Anthos GKE on Prem clusters running in on-premises data centers. The Services running on Anthos clusters often require network connectivity to Services running in other Anthos clusters.

For multi-cluster Service networking, let's look at the following scenarios:

- Multi-cluster networking on GCP** - In this architecture, all Services run on multiple Anthos GKE clusters in GCP.
- Multi-cluster networking in hybrid and multi-cloud environments** - In this architecture, Services run on multiple Anthos GKE clusters in hybrid and multi-cloud environments.

12.3.1 Multi-cluster networking on GCP

Cloud native enterprises can run the Anthos platform on GCP. This can be on a single cluster in a single region. Often, Anthos platform consists of multiple clusters and in multiple regions for resiliency.

In GCP, we recommend using a shared VPC model with multiple service projects. One of these service projects belongs to the `platform_admins` group and contains all of the Anthos GKE clusters that form the Anthos platform. Resources on these clusters are shared by multiple tenants. We also recommend using VPC native clusters. VPC native clusters use VPC IP addresses for Pod IPs which allow direct Pod to Pod connectivity across multiple clusters.

A typical Anthos platform architecture on GCP looks as follows.

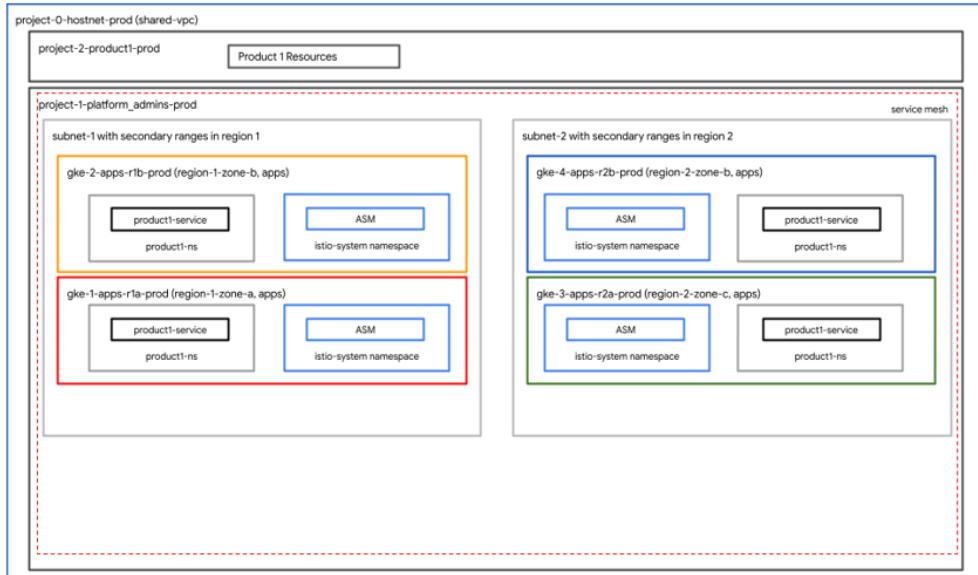


Figure 12.13 Anthos architecture: single environment

This architecture represents a single environment, for example production in this case.

There is a single network host project called `project-0-nethost-prod` that manages the shared VPC. There are two service projects, one for platform admins called `project-1-platform_admins-prod` where Anthos platform is deployed and managed by the platform administrator and one for a product called `project-2-product1-prod` where resources pertaining to `product1` reside. In this example, Anthos platform is deployed across two GCP regions to provide regional redundancy. You can create the same architecture with more than two regions or even a single region. Inside each region is a single subnet with secondary ranges. There are two zonal Anthos GKE clusters per region. Multiple clusters per region provide cluster and zone level resiliency. You can use the same design for more than two clusters per region. All clusters are VPC native clusters allowing Pod to Pod connectivity between clusters. ASM is installed on all clusters which form a multi-cluster service mesh. ASM control planes discover Services and Endpoints running in all clusters and configure the Envoy sidecar proxy running inside each Pod with routing information pertaining to all Services running inside the mesh.

Every tenant or product gets a *landing zone* in the form of a Kubernetes namespace (in all clusters inside the mesh) and a set of policies. Tenants can deploy their Services inside their own namespaces only. You can run the same Service in multiple Anthos GKE clusters for

resiliency. These Services are called [Distributed Services](#). Distributed Services act as a single logical Service from the perspective of all other entities. In the diagram above, `product1-service` is a Distributed Service with four Endpoints, where each Endpoint runs in a different cluster. ASM takes care of Service Discovery and VPC native clusters allow for L3/L4 Pod to Pod connectivity.

Anthos GKE Multi Cluster Networking

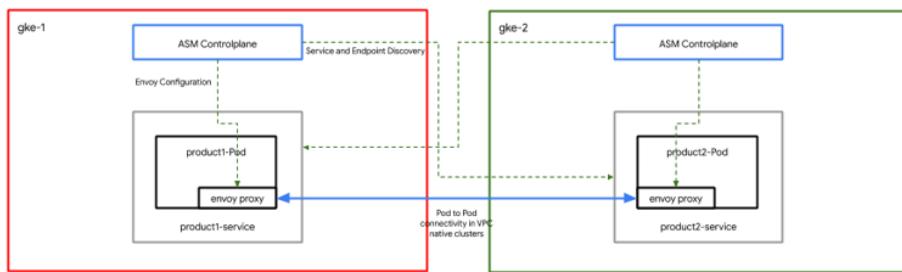


Figure 12.14 Anthos GKE multi-cluster networking

12.3.2 Multi-cluster networking in hybrid and multi-cloud environments

Apart from Anthos GKE on GCP, all other Anthos GKE clusters run in an island mode. This means that the IP addressing used for Pods and Services inside the cluster are not routable outside of the cluster. In this scenario, you can still connect Services running on multiple clusters either in the same environment, for example multiple Anthos GKE clusters running in on-premises data center, or across multiple infrastructure environments, for example Services running in Anthos GKE in GCP and in on-premises data center environments.

There are three aspects to consider when connecting multiple Anthos GKE clusters in hybrid or multi-cloud environments.

1. Network connectivity between Pods running on multiple clusters.
2. Service Discovery across multiple clusters.
3. In the case of hybrid and multi-cloud architectures, connectivity between infrastructure environments.

NETWORK CONNECTIVITY

Every Anthos GKE cluster has a load balancer. The load balancer either comes bundled during installation or can be configured manually. These load balancers allow Services to be exposed via `NodePort` to resources running outside the cluster. Each Service gets a virtual IP address (Service VIP) which is routable and reachable inside the network. Load balancer

routes the traffic to the node IP on the Service NodePort which gets forward to the Pod IP on the desired Port.

Services (and Pods) running in one cluster can access the Service VIP for a Service running in another cluster, which gets routed to the desired Pod via the load balancer.

Anthos GKE Hybrid Multi Cluster Networking

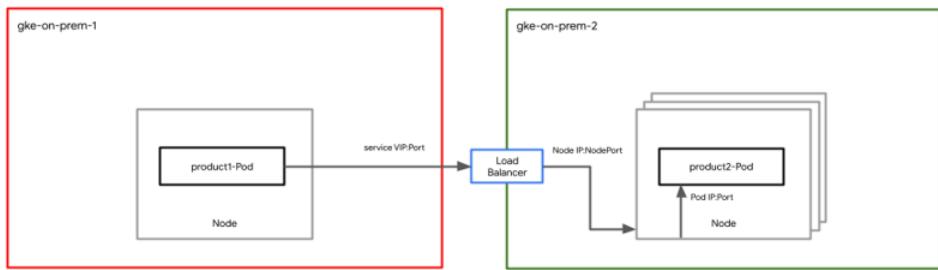


Figure 12.15 Anthos GKE hybrid multi-cluster networking

Anthos clusters can also be configured with ingress controllers. Ingress controllers are L7/HTTP load balancers that typically reside inside the cluster. The Ingress controllers themselves are exposed via an L3/L4 load balancer. This way you can use one VIP (the Ingress VIP) for multiple Services running on the same cluster. Ingress controllers act upon Ingress rules which dictate how the traffic is to be routed inside the cluster.

Anthos GKE Hybrid Multi Cluster Networking: Ingress

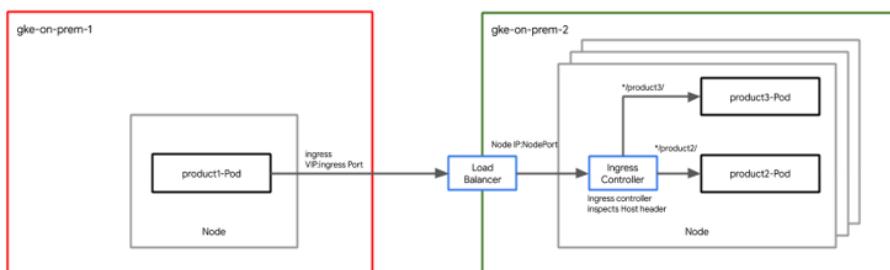


Figure 12.16 Anthos GKE hybrid multi-cluster networking: Ingress

MULTI-CLUSTER SERVICE DISCOVERY

Anthos Service Mesh (ASM) is used for multi-cluster service discovery. ASM controlplane is installed in each cluster. The ASM controlplane discovers Services and Endpoints from all clusters. This is also known as the service mesh. ASM controlplane must have network access to the Kubernetes API server of all Anthos clusters inside the service mesh. ASM creates its own service registry which is a list of Services and their associated Endpoints (or Pods).

In Anthos GKE on GCP, the Endpoints are the actual Pod IP addresses if using VPC native clusters. Traffic flows from Pod to Pod using VPC routing. In non-GCP Anthos clusters, traffic between clusters flows through an L7 Envoy proxy. This proxy runs as a Service in every Anthos cluster called the `istio-ingressgateway`. Traffic bound for all Services inside a cluster flows through the `istio-ingressgateway`. `istio-ingressgateway` is configured to inspect the Host header and route the traffic to the appropriate Service inside the cluster.

Anthos GKE Hybrid Multi Cluster Networking: ASM

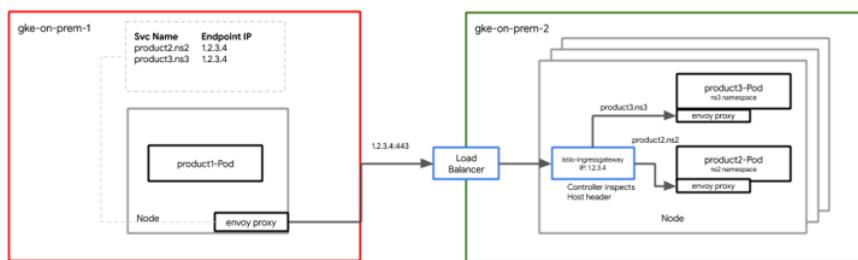


Figure 12.17 Anthos GKE hybrid multi-cluster networking: Anthos Service Mesh (ASM)

For Distributed Services, we recommend using ASM, which provides Service Discovery and traffic routing functionality across multiple clusters.

HYBRID AND MULTI-CLOUD CONNECTIVITY

You can connect Services running in multiple Anthos clusters across multiple infrastructure environments as long as you can reach the Kubernetes API server and the external public load balancer of the target Anthos cluster.

There are three main ways to connect infrastructure environments.

1. [Cloud Interconnect](#) - Cloud Interconnect extends on-premises network to Google's network through a highly available, low latency connection. You can use Dedicated Interconnect to connect directly to Google or use Partner Interconnect to connect to Google through a supported service provider. Dedicated Interconnect provides direct physical connections between your on-premises network and Google's network.
2. [Cloud VPN](#) - Cloud VPN securely connects your [peer](#) network to your [Virtual Private Cloud \(VPC\) network](#) through an [IPsec VPN](#) connection. Traffic traveling between the two networks is encrypted by one VPN gateway, and then decrypted by the other VPN gateway.
3. [Public Internet](#) - Anthos platform on multiple environments can be connected over the public internet without using Cloud Interconnect or VPN. Services running on the platform connect over the public internet using TLS/mTLS. This type of connectivity is done at a per-service level using Anthos Service Mesh (ASM) and not at the network level.

These connectivity models are explained in detail [here](#).

12.4 Services and Client Connectivity

This section addresses services and client connectivity in the Anthos platform.

This can be divided into three categories.

1. **Client to Service connectivity** - This is also sometimes referred to as north-south traffic, suggesting traffic originates from outside of the platform (north) and travels south into the platform.
2. **Service to Service connectivity** - This is also sometimes referred to as east-west traffic, suggesting traffic traverses the platform laterally (hence east-west). All traffic originates and terminates inside the Anthos platform.
3. **Service to external service connectivity** - This is traffic egressing the platform.

12.4.1 Client to Service connectivity

In this context a client is referred to an entity that resides outside of the Anthos platform and a Service runs inside the Anthos platform. There are two main ways to access Services inside Anthos platform:

1. **With Anthos Service Mesh (ASM)** - With ASM, you can use ASM ingress for HTTP(S) and TCP traffic. ASM provides additional L7 functionality for example ability to perform authentication and authorization at ingress. ASM is the recommended way of accessing web based Services in the Anthos platform. ASM can also be used for TCP based Services.
2. **Without Anthos Service Mesh** - All Anthos clusters have the option to configure a load balancer. The load balancer either comes integrated/bundled when you deploy the Anthos cluster or it can be configured manually. Any TCP based service can be exposed using a service of type `LoadBalancer` which creates a Service VIP that can be accessed by the client. In addition, all Anthos clusters can be configured with ingress. Ingress controllers typically run inside the cluster as L7 proxies (with the exception of Anthos GKE on GCP and Anthos on AWS using ALB). The ingress controllers themselves are exposed using the L3/L4 load balancer. Ingress is the recommended way to expose web based Services. Ingress rules are implemented as part of the Service deploy pipeline and ingress controllers enforce these rules which include listening and routing traffic to the appropriate service.

12.4.2 Service to Service Connectivity

Services that run inside the cluster require network connectivity. This is accomplished in two ways.

1. **With Anthos Service Mesh (ASM)** - We recommend using ASM specially in a multi-cluster Anthos platform. ASM provides Service discovery as well as routing logic between Services. ASM also configured authentication and authorization between Services. For example, you can enable mTLS at the service mesh level encrypting all Service to Service traffic. You can also configure security policies at an individual Service layer. Besides service discovery and networking, ASM provides additional features such as telemetry, quotas, rate limiting and circuit breaker. For more on the features and benefits of ASM, please see chapter xyz.
2. **Without Anthos Service Mesh** - If you choose to not use ASM, you can still configure service to service connectivity. From a networking standpoint, you can use either the load balancer or the ingress pattern to access services running inside clusters. You would have to configure service discovery yourself. You can use DNS to provide this functionality.

In either case, you can also use `NetworkPolicy` inside the cluster to control/limit the traffic between Pods and Services.

12.4.3 Service to external services connectivity

You can control egress traffic from any Anthos cluster in one of two ways.

1. **With Anthos Service Mesh (ASM)** - ASM provides both an ingress and an egress gateway. We have previously discussed how ingress works with ASM. Similarly, there is a second proxy that can be configured at the perimeter of the service mesh called the `istio-egressgateway`. You can then configure `ServiceEntries` for only the services that are allowed to be accessed from inside the cluster. You can set the `outboundTrafficPolicy` mode to `REGISTRY_ONLY`. This blocks all outbound traffic that is not destined for a service inside the mesh. You can then create individual `ServiceEntries` for access to services running outside the platform. An example of `ServiceEntry` may look like the following.

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: httpbin-ext
spec:
  hosts:
    - httpbin.org
  ports:
    - number: 80
      name: http
      protocol: HTTP
  resolution: DNS
  location: MESH_EXTERNAL           #A
```

#A Location `MESH_EXTERNAL` signifies that the Service is external to the service mesh and a DNS entry is manually added to the mesh registry

This rule allows traffic destined for `httpbin.org` on port 80. Note the location of the Service is `MESH_EXTERNAL` signifying this service is outside of the service mesh and the Anthos platform.

1. **Without Anthos Service Mesh** - You can use `NetworkPolicies` inside the cluster to control ingress and egress traffic to Pods based on label selectors. Since all Pod egress traffic exits via the node IP, you can further control egress traffic through firewall rules by limiting what destinations the nodes IP subnets can access.

12.5 Summary

Anthos networking can be divided into four layers:

- **Cloud networking and hybrid connectivity** - This is the lowest layer of Anthos networking. This layer describes how to set up networking within each cloud environment as well as options to securely connect multiple cloud environments together. Inside GCP, you can set up a single network (or VPC), a shared VPC or multiple VPCs depending upon the organizational and functional requirements. In non-GCP environments, all Anthos clusters are treated as isolated networks (or in "island mode"). There are multiple hybrid connectivity options:
 - **Dedicated interconnect** - This option provides direct physical connections between your on-premises network and Google's network. Dedicated Interconnect enables you to transfer large amounts of data between networks, which can be more cost-effective than purchasing additional bandwidth over the public internet.
 - **Cloud VPN** - This option securely extends your peer network to Google's network through an IPsec VPN tunnel. Traffic is encrypted and travels between the two networks over the public internet. Cloud VPN is useful for low-volume data connections.
 - **Public internet** - This option does not require any special software or hardware to connect disparate networks together. Instead, TLS/mTLS connection is used to secure service-to-service connections.
- **Anthos GKE networking** - This is the Kubernetes networking layer. Anthos GKE clusters can be deployed to a variety of environments, for example GCP, on VMWare in an on-premises data center, on bare metal servers, and on AWS. In addition to the supported Anthos clusters, you can also register any [conformant](#) Kubernetes cluster to the Anthos platform. For example, you can register [EKS](#) clusters running in AWS and [AKS](#) clusters running in Azure to the Anthos platform. There are currently six types of Anthos clusters.
 - Anthos clusters on GCP (GKE)
 - Anthos clusters on VMWare (GKE on-prem)
 - Anthos clusters on bare metal
 - Anthos clusters on AWS (GKE on AWS)
 - Anthos clusters on Azure (GKE on Azure)
 - Anthos attached clusters (conformant Kubernetes clusters)
- **Anthos multi-cluster networking** - Anthos multi-cluster networking deals with environments with multiple clusters where services need to communicate across cluster boundaries. This section can be divided into two subsections:

- **Anthos multi-cluster networking with GKE on GCP** - In GKE on GCP, you can either have a flat network architecture (using a single or shared VPC) or a multiple network (multiple VPC) model. In a flat network architecture using VPC-native GKE clusters, VPC networking automatically allows for Pod to Pod connectivity between multiple clusters. Clusters can be in any region. No additional configuration is required for Pod to Pod connectivity between clusters. In a multiple VPC architecture, you need additional configuration to connect Pods and Services between multiple clusters. For example, you can use special gateways or ingress model to communicate between clusters.
- **Anthos multi-cluster networking in non-GCP environments** - In all non-GCP clusters, every cluster and specifically Pod IP address range is isolated from other clusters. This means that there is no direct connectivity between Pods in multiple clusters. In order to connect multiple clusters, special gateways or ingress must be used. Anthos Service Mesh can be used to deploy such gateways. These are often called “east-west gateways” and are deployed in all clusters participating in a multi-cluster mesh. In addition Anthos Service Mesh also provides multi-cluster service discovery.
- **Service layer networking** - The top layer of Anthos networking is service layer networking. This layer addresses how services discover and communicate with one another. In the previous section, we mentioned Anthos Service Mesh. Anthos Service Mesh allows you to create a service mesh atop multiple Anthos clusters running in multiple cloud environments. This layer abstracts the complexity of lower layer networking and allows the operator to focus on the service layer. Anthos Service Mesh uses sidecar per-workload and specialized gateways to connect multiple clusters across multiple environments and networks together. By using Anthos Service Mesh, operators can focus on service layer functions for example authentication, encryption and authorization instead of managing individual workloads at the cluster level. This allows operators and administrators to operate at scale where there may be multiple clusters, in multiple environments, in multiple networks running numerous services.

13

Anthos Config Management

by Michael Madison

- This chapter covers
- Why configuration at scale is a challenge
- An overview of Anthos Configuration Management itself
- Examples and Case Studies of ACM implementations showing the utility and versatility of the solution.

In the world of application development, there's always a desire for more speed and more capability. More applications that fulfill more tasks, automate more minutiae, run faster, and in locations closer to where they are actually used. The proliferation of smartphones, tablets, and IoT-devices and the continued advancement of computers into every part of our daily lives drive the need for more compute power. Environmental factors, the availability of high-speed internet and other utilities, as well as government regulations are changing the way companies deploy resources. Depending on the circumstances, this could result in a concentration of compute resources in a few data centers, a move to mostly cloud-based compute, fragmenting to lots of "mini" data centers, or a combination of these solutions.

Organizations rarely decide to reduce the total resources it manages. While there might be short-term reductions, consolidations, or even eliminations of applications, most companies will be managing more tomorrow than today. This has been the path that application development has followed for the past 40 years or longer.

But the expanding use of Kubernetes has brought this into focus for many organizations as they come to grips with moving and managing thousands of VMs and applications in the Kubernetes landscape. Although many legacy tools would still work, leveraging them in the same way would negate most of the advantages available with Kubernetes. On top of that, legacy toolsets are often disconnected from one another, forcing managers to make firewall

changes in one tool, granting VM access in another, and setting up routing through a third. These challenges fall largely on the shoulders of IT Operations teams who are charged with implementing and maintaining all of this infrastructure.

As more companies move to Kubernetes for their daily operations, the need for security professionals and managers to be confident in their ability to configure, administer, and audit Kubernetes clusters has become critical for business success. IT Security groups are responsible for developing and implementing security controls around and within the IT infrastructure. This includes software, hardware, and physical limitations, policies, procedures, and guidance. Many companies adopt a tiered permissions model, allowing super-users a greater subset of abilities without becoming full administrators. Since much of the work in Kubernetes is driven by a common definition language, expressed in JSON or YAML, the security framework should also be familiar to IT Security teams who regularly work within Kubernetes.

To help organizations address this need, Google has created Anthos Config Management (ACM) to simplify the development, deployment, and maintenance of Kubernetes policies. In the next section, you'll examine the full scope of challenges that ACM helps to solve and the opportunities ACM provides to drive efficiencies with an organization.

13.1 What are we trying to solve?

Over time, businesses have moved processes to digital formats. Even without the internet as an engagement platform, companies have shifted their internal operations to rely on digital applications and communications. When that's added to the massive drive to engage digital customers, companies' need for compute power is greater than ever before. And it shows no signs of slowing. One of the newest aspects of computing, edge computing, is expected to be an over \$40 billion market alone by 2027, according to Grandview Research (<https://www.grandviewresearch.com/press-release/global-edge-computing-market>).

Additionally, many businesses see greater efficiency by having these systems communicate with each other to automate their processes. The proliferation of, for example, ticket-based self-service software such as ServiceNow, personnel management solutions like Workday, and combined authentication and authorization services such as Okta, encourage companies to expand their capabilities on-site and in the cloud. As companies pivot to depend more heavily on their staff's development capabilities for mission-critical solutions, the complexity involved in the deployment of these applications increases with each new vendor they bring into their ecosystem.

The needs of any company diverge from their closest competitor as they design their unique value to customers, but most businesses have a digital presence both on-premises and in the cloud. While some companies have leveraged multiple data centers or colocation facilities to provide their redundant and reliable infrastructure, many have turned to cloud providers. But cloud provider's best practices and user experiences can vary widely. Each cloud brought online by an Operations team greatly increases their technical burden and operational overhead. Even in the case of data centers designed to be identical, security controls and physical separation still impose barriers to their operation and maintainability.

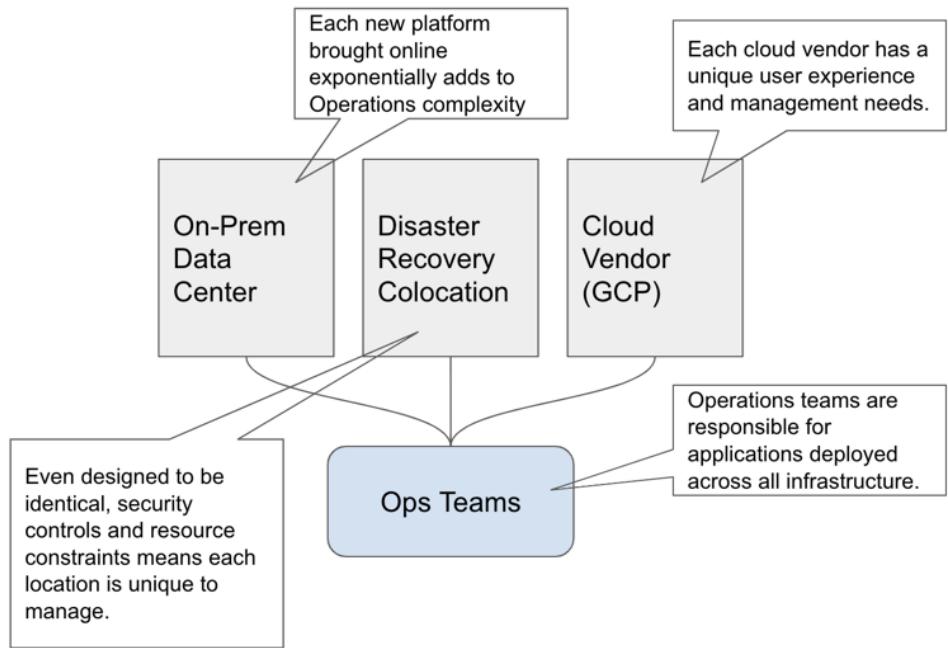


Figure 13.1. Bringing multiple infrastructure platforms online adds considerable operations overhead.

The size of a company's digital footprint also adds complexity to the configuration and operation of their systems. Multiple working locations, more data centers, and a corresponding increase to the number of people involved in the management of these systems all add their own issues. As an organization grows, introducing and utilizing systemic security and configuration controls becomes vital.

ACM addresses these challenges using three central capabilities:

1. Managing complexity
2. Workload observability and inspection
3. Remediating and preventing issues when they do occur

Next, you'll examine how ACM manages complexity in modern infrastructure.

MANAGING COMPLEXITY

There are innumerable ways to configure compute infrastructure. But most solutions do follow some general patterns. These solutions can generally be grouped by whether the system scales horizontally or vertically. For a company adopting or using a Kubernetes-based infrastructure, a similar decision must be made about the overall design of all the Kubernetes clusters at a company. A company using multiple, smaller clusters for different purposes (e.g. one or more per team) is implementing a horizontally scaling solution. A company using

a smaller number of larger clusters (e.g. one each for dev, test, and prod) is building vertical scalability into their Kubernetes infrastructure. Neither approach is better than the other, but a company should determine what fits best with their approach to deployments and software design.

Geographic limitations, edge processing needs, and telecommunications operations also impose restrictions on how clusters are delineated. Government regulations in certain countries prevent the egress of data from those regions, requiring the databases and application layers to be located local to the country. Even without government regulation, the operations of certain types of businesses, such as restaurants, retail stores, or even banks, might prefer a local processing system running a subset of applications that would benefit from, or require, a shorter communications loop.

In addition, large businesses require more staff to efficiently organize and operate the IT infrastructure. In order to mitigate single-points-of-failure among IT personnel, developing simple processes that can be scaled out to multiple people is critical to long-term success.

OBSERVABILITY AND INSPECTION

Visibility and inspection of workloads and overall health of a Kubernetes cluster is outside the purview of this chapter (being primarily covered in the Operation Management with Anthos chapter). However, the plaintext representation of policies and configurations for all clusters afforded by Anthos Config Management can do a great deal on the front-end to ensure that appropriate policies are adhered to.

The goal of Anthos Config Management is to maintain a cluster in a state specified by a policy directory stored in a git repository. This policy directory exists separate from the clusters being managed, and is stored in a text-based format. Thus, the policy directory itself can be used as a source of information about the configuration of the cluster. By using the features of the git repo itself, the IT Operations team can determine when a cluster is out of compliance, and can easily track changes to a cluster's configuration.

ACM provides a command-line utility, via the `nomos` command, to interrogate clusters directly and determine their current state. Operations teams can use `nomos` to diagnose the rollout of a change, and determine if there is an error or lag that would be causing issues. Much of the information that is provided via nomos is visible in the Anthos UI¹ within the Google Cloud Console, showing what configuration version each cluster is currently running, as well as the overall state of the ACM installation. Also, the Kubernetes operator for the system logs events and information in the same manner as other containers, and thus can be viewed in Cloud Logging and used in Cloud Monitoring alerts and metrics.

REMEDIATING AND PREVENTING ISSUES

One of the major responsibilities of an IT Operations team is to maintain system reliability and uptime. Knowing that a change has caused a disruption, being able to quickly isolate the issue, and rapidly applying a remediation, are critical tools in the team's arsenal. On all three

¹ The Anthos UI is covered primarily in Chapter 3: Anthos, the Single Pane of Glass UX

of these points, ACM brings unique features that enable the team to respond to situations as they occur.

Since ACM is driven from a git repository, you can easily tie activity on one or many branches into Cloud Monitoring or another monitoring suite and bring additional alerting to bear after a change to the policies. Using the repo as the source of truth, the team can investigate what changes were applied at what times to narrow down any problematic configurations. Existing git tooling can then be used to revert or fix the configuration. Due to ACM's design, these changes normally take effect within a minute or two of the change being pushed to the policy repo.

In addition to fixing issues that have already been deployed, existing CI/CD tooling and processes can be leveraged to verify configurations before being allowed to take effect. Other tooling around git, such as pull and merge requests, branching, and more, can also serve to allow multiple users to develop new policies, while permitting the organization the ability to approve those changes prior to deployment.

ACM includes the ability to apply a configuration to a subset of clusters. While we use this in our examples to deploy different versions of an application to clusters by region, the same functionality can be used to deploy changes in a controlled fashion, or to apply different roles per cluster. For example, a retail chain running a cluster in each store can deploy a new version of an application to a specific set of test stores before rolling it out to the entire chain. A company using different clusters for dev, test, and prod environments can grant users different permissions based on the cluster while still leveraging a single policy repository.

BRINGING IT TOGETHER

All three of these issues are well-handled with Anthos Config Management. Utilizing tools familiar to IT professionals, and with a design that thrives in a highly-distributed ecosystem, ACM can help teams manage large systems easily. In the next section, we will cover a brief overview of how ACM works, and the components that can be included with an installation.

13.2 Overview of ACM

Now that we have a good idea of the problems we are trying to use Anthos Config Management to solve, let us take a deeper look at the technical implementation of ACM.

ACM works by way of a Kubernetes operator² deployed onto each cluster to be managed. A configuration file containing the canonical name of the cluster, a git configuration, and a set of feature flags, is also applied to the cluster. The operator uses the git configuration to connect to the git repository containing the full configuration information for the cluster. The feature flags are switches to activate Config Connector, Policy Controller, or Hierarchy Controller, which will be covered later in this chapter. The operator configuration will also be expanded to automatically configure additional services and components, such as Binary Authorization, in the future. ACM uses the name of the cluster, along with the policy

² For more information on the Kubernetes operator pattern, see <https://kubernetes.io/docs/concepts/extend-kubernetes/operator/>

configurations in the git repo, to add ephemeral tags to the cluster. These tags can then be used within the policy repo to modify what resources are deployed to the specific cluster.

ACM works on a minimum-footprint mentality: it does not try to take over the entire Kubernetes cluster. Rather, the operator knows what objects are defined in the policy configuration and only works to manage those specific objects. This allows multiple deployment mechanisms to work in parallel on a single Kubernetes cluster without stepping on each other. However, using multiple tools does add an additional burden of needing to know which tools have deployed each object. ACM includes a specific annotation on objects it manages³, but that may not look the same for all tools. As we will see later, ACM repos can be configured in an unstructured mode that allows an organization to continue using existing tools which support outputting to YAML or JSON while still leveraging ACM to perform the actual deployment and management processes.

The operator syncs every few minutes, and this frequency can be adjusted in the configuration, depending on the needs of the organization. ACM is able to use both public and private repos, with appropriate credentials, as the policy repository. In addition, the git configuration can be pointed at a directory below the top-level of the repository. This can be useful if the git repository uses a templating engine, or even application code: a subdirectory can be used to store the policies without needing to sync between two multiple repos.

There are three primary ways to deploy Anthos Configuration Management, depending on the type of Kubernetes cluster you are using: A Google Kubernetes Engine (GKE) cluster deployed on GCP, a GKE on VMware cluster, or another flavor of Kubernetes. For GKE on GCP, enabling ACM is a simple matter of ticking a checkbox on the cluster configuration page. For GKE on VMware, ACM is enabled by default and cannot be disabled. Only clusters that do not fit into either category require manual configuration to install the operator: retrieve the most recent version of the operator's Custom Resource Definition file, provided by Google, and apply it to the cluster.

At this point, all flavors of Kubernetes have the operator installed and running, but ACM still needs to be enabled and told where to pull policies from. This is done by creating an operator configuration object and applying it to the cluster, along with whatever git credentials are required. Multiple methods of git authentication are supported, including public repos with no authentication, ssh keypair, personal access tokens, and Google Service Accounts. Some of these methods require specific information to be loaded into a Kubernetes secret in order for the operator to load it properly. The configuration object allows specifying proxies for the git repository, if needed. In addition to the git connection information, the configuration object can contain settings to enable or disable the individual components names above, as well as naming the cluster for use within ACM policy rules. As ACM's capabilities expand, more options will be added to this configuration object, but the current structure is:

³The annotation is `configmanagement.gke.io/managed: enabled`

```

apiVersion: configmanagement.gke.io/v1
kind: ConfigManagement
metadata:
  name: config-management
spec:
  clusterName: <name of cluster>
  configConnector:
    enabled: <true/false>
  enableMultiRepo: <true/false, enables multiple repository mode>
  enableLegacyFields: <true/false, used with multi repo, see below>
  policyController:
    enabled: <true/false>
    templateLibraryInstalled: <true/false, installs the Google-provided template library
      for policy controller>
  hierarchyController:
    enabled: <true/false>
  sourceFormat: <hierarchy or unstructured. Sets the type of policy organization>
  enableMultiRepo: <if set to true, turns on multiple repositories>
  git:
    syncRepo: <url of the git repository>
    syncBranch: <branch of the git repository to sync from>
    policyDir: <relative path in the git repository to the policy directory>
    syncWait: <number of seconds between sync attempts>
    syncRev: <git revision to sync from. Used if a specific commit or tag should be used
      instead of a branch>
    secretType: <ssh, cookiefile, token, gcnodenode, or none. Specifies the type of
      authentication to perform>
  proxy:
    httpProxy: <proxy information for http connection, styled similarly to the HTTP_PROXY
      environment variable>
    httpsProxy: <proxy information for https connection, styled similarly to the
      HTTPS_PROXY environment variable>

```

The operator on the individual clusters is the mechanism ACM uses to update the objects on each cluster. Although ACM uses a central git repository, since the individual clusters reach out to fetch the configuration, this greatly simplifies connectivity between the cluster and repository. Since the repository does not push the configs out, we do not need to introduce additional complexity or security ingress holes, nor does the central repository need to know about every individual cluster beforehand.

13.2.1 ACM Policy Structure

The ACM policy directory must be in one of two supported formats, either hierarchy or unstructured, with hierarchy as the default. This setting is also reflected in the operator configuration object referenced above, in the spec.sourceFormat key. In both cases, the policy directory defines Kubernetes objects which are then examined and applied by the ACM operator on each of the clusters connected to the git repository. Some of these objects are used by ACM itself to determine what configurations to apply to the current cluster and do not get created on the cluster itself.

In addition to the overall format of the repository, it is possible to use multiple repositories to configure clusters. When using the enableMultiRepo functionality, a single repository is used as the root repository (and may be hierarchy or unstructured), while all other repositories are used to configure objects in a single namespace.

HIERARCHY

In a hierarchy repository, there are top-level directories in the policy directory that separate configuration files based on purpose and scope: "system", "clusterregistry", "cluster", and "namespaces":

Directory	Purpose
system	Configs related to the policy repository itself, such as the version of the deployed configuration.
clusterregistry	Stores Cluster and ClusterSelector objects, which are used together to select subsets of clusters to restrict where a specific object is applied to. Cluster definitions attach specific tags to a cluster by name; ClusterSelectors can then use these tags to select a set of clusters meeting a certain set of requirements.
cluster	Contains objects that are defined for the entire cluster, except for namespaces.
namespaces	Contains objects that are assigned to one or more specific namespaces, as well the Namespace and NamespaceSelector definitions.

ACM in hierarchy mode leverages a concept of 'abstract' namespaces, a grouping of one or more actual namespaces which should share a set of Kubernetes objects. For instance, you might define a Role or ConfigMap in each namespace that a team uses. When ACM analyzes the repository, any object defined in an abstract namespace is automatically copied into every namespace underneath it. These abstract namespaces can also be nested within each other in order to have multiple layers of abstraction. For example, you may place all application development teams under an 'app-dev' abstract namespace, and then each team in a separate abstract namespace within 'app-dev'.

While ACM will copy an object in an abstract namespace to all child namespaces, you can use a NamespaceSelector to restrict what namespaces the object is applied to. Using the 'app-dev' example above, we want to deploy a ConfigMap to multiple namespaces across multiple teams, but only to the namespaces which contain finance-related applications. By applying a label to those namespaces, we can then define a NamespaceSelector to select only those namespaces, and then link the ConfigMap config object to the NamespaceSelector. While these selectors do have their purpose in a hierarchy repo, their primary use is in an unstructured repo. Further, in a hierarchy repo, a specific namespace must be a child of the folder containing the object, as well as matching the selector. A full example in an unstructured repo with the configuration objects defined can be found in the Evermore Industries example at the end of the chapter.

Once you have a specific namespace name to be created, you also should decide on which abstract namespace to inherit from. This may mean that certain objects must be defined at a higher level than would normally be done, simply to have them inherited by the individual namespaces. Once you have the name and the abstract namespaces, you create a folder with

that name at the bottom of that set of namespace directories. Inside the newly-created directory, you must also create the Kubernetes Namespace object with the same name. All the objects defined in the abstract namespaces are also created inside the leaf namespace. Let's take a look at an example:

Given the following folder structure:

```
namespaces
  └── staging
      ├── qa-rbac.yaml
      └── weather-app-staging
          └── namespace.yaml
  └── production
      ├── developer-rbac.yaml
      ├── app-service-account.yaml
      ├── weather-app-prod
          ├── namespace.yaml
          └── application.yaml
      ├── front-office-prod
          ├── namespace.yaml
          └── application.yaml
  └── marketing
      └── namespace.yaml
```

For the production abstract namespace, we are defining a role and binding for developers in the developer-rbac.yaml, and a Kubernetes service account for the applications in the app-service-account.yaml. Because weather-app-prod, front-office-prod, and marketing namespaces are under the production abstract namespace, the role, role binding, and Kubernetes service account will be created in all three namespaces. Due to how ACM analyzes the policy repository, actual namespaces cannot have sub-directories in their folder, and every leaf directory must be an actual namespace with the corresponding namespace declaration in the directory. Failing to adhere to this restriction will cause a configuration error when ACM is deployed.

In addition to abstract namespaces, objects can utilize a NamespaceSelector (which are declared in a similar manner to ClusterSelectors, covered later in this chapter), in order to only affect a subset of namespaces within the object's scope. In the example above, the app-service-account can utilize a selector to only deploy to the weather-app-prod and front-office-prod namespaces, and not the marketing namespace. However, NamespaceSelectors in hierarchy repositories only operate on namespaces in the current folder tree. For example, even if the namespace selector included weather-app-staging in its criteria, the app-service-account defined under the production abstract namespace would never be applied to the staging namespace since the weather-app-staging directory is not a child of the directory that contains the app-service-account.

PROS & CONS

A hierarchy repository simplifies the deployment of objects to a subset of namespaces, since an object can only be deployed to the namespaces at or below the level of the configuration file in the repository. With the use of NamespaceSelectors, an organization can further restrict what namespaces an object can be deployed to. This can be especially useful if there

are multiple ways to group namespaces. For example, namespaces might be grouped by development team, but may also need to be grouped by function (frontend, middleware, e.g.) or business unit. Using a hierarchy repo, you must choose one "primary" grouping strategy; if an object needs to be deployed to multiple namespaces that are not grouped together, the object would be placed at a higher level in the repo and restricted using a NamespaceSelector. This organization makes it very simple to start determining which namespaces an object deploys to, since it can only be those defined at or below the object's definition file. Cluster-level resources, and resources that are primarily used to deliver ACM also have dedicated folders where they must be located, making it easier to find a given object.

However, this rigid structure can cause difficulties when implementing ACM at your organization. Many organizations already have at least a basic familiarity with Kubernetes and utilize existing toolsets and processes to deploy Kubernetes resources and applications. Since cross-namespace objects must be configured in different folders in a hierarchy repository, this can complicate the integration of ACM into an existing CI/CD pipeline. An organization should weigh the benefits of the automatic duplication of objects afforded by an hierarchy repository with the restrictions it imposes.

UNSTRUCTURED

Unlike a hierarchy repo, an unstructured repo has no special directory structure. Teams are free to use whatever style of organization they wish, such as grouping files and objects by application or team. Using an unstructured repo, however, prevents ACM from leveraging the concept of an abstract namespace to automatically create a single object in multiple namespaces. To compensate for this restriction, an object must declare either a namespace, or a NamespaceSelector. While NamespaceSelectors behave in the same manner as in a hierarchy repository, without the restriction of only operating on namespaces in the same folder tree, greater care must be taken to make sure only the desired namespace(s) actually match the selector.

PROS & CONS

In the event that an organization is already using a templating engine to deploy objects to Kubernetes, an unstructured repo becomes even more favorable. Since most templating engines, including Helm, include the ability to export the completed Kubernetes objects to a local directory, you can use the output from those commands and simply place the generated configurations directly into the ACM policy directory. Because an unstructured ACM repo does not care about the exact placement of configurations under the policy directory, this can provide a less-stressful upgrade path when implementing ACM.

However, unstructured repositories have a couple of wrinkles when it comes to namespace assignment. Since configurations in an unstructured repository cannot infer the namespace they should be assigned to, users must explicitly assign all objects. This can result in the deployment of an object to an unintended namespace if the selector is defined or used improperly. In addition, finding a resource becomes more complicated since there is no implicit relationship between the location of the configuration file and the deployment namespace.

MULTIPLE REPOSITORY MODE⁴

Configuring ACM to pull from multiple repositories allows organizations to permit individual teams to manage their own namespaces while still leveraging many of the benefits of ACM. When the cluster configuration object is set to enable Multiple Repository mode, using the `enableMultiRepo` flag, the spec.git set of fields is not supported. Instead, you create a separate RootSync object to hold the configuration details for the root repository.

With enableMultiRepo set, an organization can define the repository to be used for each individual namespace. As with the RootSync object, these individual RepoSync objects contain the configuration for fetching from a git repository as well as the directory in that repository for the top of the policy tree. Even when using Multiple Repository mode, the root repository can still define objects to be managed in any namespace. In the case of a conflict between the root repository and the individual namespace repositories, the root repository's version is the one used.

The root repository of a Multiple Repository setup functions identically to a configuration that is not in Multiple Repository mode; only the configuration of how to fetch the repository changes. Therefore, Multiple Repository mode is an ideal solution to allow operations and security teams to impose policies, RBAC rules, configure namespaces, istio rules, etc... while enabling application teams to manage and deploy their own applications into individual namespaces. The team managing the root repository also needs to add the appropriate policies to define what objects the individual repositories can modify. This is done by defining a custom Role or ClusterRole, or using one of the built-in roles, and then using a RoleBinding to attach the namespace's worker service account to that role. This allows the operations team to offload much of the work of configuring a given application to the teams, and defining custom permissions per team if needed, rather than requiring the central team to validate or perform the work themselves.

13.2.2 ACM-specific Objects

While ACM can manage any valid Kubernetes object, there are custom objects used to adjust how the system operates and applies new configurations.

CONFIGMANAGEMENT

Used by ACM to determine how, and where, to fetch the policy configurations to be used for the cluster. Deploying a ConfigManagement object to the cluster activates ACM for that cluster. This object also defines the name of the cluster, as used inside ACM, and determines which plugins (Config Connector, Policy Controller, and Hierarchy Controller) are active for the cluster.

ROOTSYNC/REPOSYNC

When the cluster is running in Multiple Repository Mode, the configuration for fetching policies, including git urls and secrets, are not stored in the ConfigManagement object, but

⁴ <https://cloud.google.com/anthos-config-management/docs/how-to/multi-repo>

rather in either the RootSync object (for the core repository), or in RepoSync objects in each namespace.

CLUSTER

A cluster config is created in the ACM policy repo and allows users to attach labels to a specific cluster by cluster name. These labels are then used in ClusterSelectors to select specific types of clusters. In a hierarchy repo, the Cluster definitions must be in the "clusterregistry" directory.

CLUSTERSELECTOR

This object leverages the common Kubernetes labelSelectors pattern⁵ to select a subset of clusters. The ClusterSelector can then be used by an object, such as a Deployment, ConfigMap, or Secret, to only deploy that object in clusters matching the selector. In a hierarchy repo, these must be in the "clusterregistry" directory.

NAMESPACESELECTOR

Similar to the ClusterSelector, this selector also uses labelSelectors, but is used to select namespaces instead. Primarily used in unstructured repos, or in a hierarchy repo as an additional method to limit which namespaces an object is deployed to.

HIERARCHYCONFIGURATION

These objects are declared in individual namespaces and point to their parent. This sets up the hierarchical namespace relationship that the Hierarchy Controller uses. Note that using the Hierarchy Controller is not the same as a hierarchy repository, the Hierarchy Controller will be explored further later in the chapter.

13.2.3 Additional Components

Although not strictly part of ACM itself, Config Connector, Policy Controller, and Hierarchy Controller greatly enhance the functionality of ACM and your Kubernetes environments. This section only gives a short introduction to each component, but all three are demonstrated in the examples at the end of the chapter. Google is also integrating additional components as development on Anthos continues. Please refer to the online documentation at <https://cloud.google.com/anthos-config-management/docs> for the most up-to-date information on available add-ons.

CONFIG CONNECTOR⁶

Config Connector is an add-on to Kubernetes that allows you to configure GCP resources, such as SQL Instances, Storage Buckets, and Compute Engine VMs, using Kubernetes objects. A full example of the structure of one of these objects is provided in the Evermore Industries case study in this chapter. Config Connector is only usable in GKE on GCP clusters. With proper permissioning, this allows a developer proficient with Kubernetes to create several different types of GCP resources, including SQL Databases, Networks, BigQuery Datasets and Tables, Compute Instances, PubSub Topics and Subscriptions, and Storage

⁵ This is the same pattern Deployments and Jobs use and is detailed at <https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/>

⁶ <https://cloud.google.com/config-connector/docs>

Buckets. In addition, these configurations can reference each other, simplifying configuration and allowing for a single source of truth.

Users can also leverage Kubernetes secrets to store sensitive information, such as passwords, and then use that information in Config Connector resources. Each of the Config Connector objects also includes a status section, describing the current state of the resource as it is created or updated in GCP.

POLICY CONTROLLER⁷

While the Kubernetes role-based access control system (RBAC)⁸ has the ability to finely control what a specific user is permitted to do at the namespace and object-type level, it does not enforce arbitrary policies, or policies on specific objects. For example, we may want all pods deployed in a specific namespace to declare CPU limits for the containers, or require all namespaces include a custom label that indicates the cost center that should be billed for the resource usage. We may also want to protect a specific deployment and prevent modifications to that specific resource, while still allowing other resources in the same namespace to be modified. This is where Policy Controller comes into play.

Built from the open-source OPA Gatekeeper project⁹, Policy Controller is an admission controller that checks and verifies any create of, or update to, an object against the policies that have been declared and loaded to the cluster. Each policy consists of a constraint template, which is written using Rego to actually perform the test needed, and a constraint, which provides the arguments to the template for the specific policy. A set of existing templates, known as a template library, is provided by default when Policy Controller is enabled, though users can create customized constraint templates as well. Users can then create constraints that utilize these policy templates to enforce specific restrictions on the cluster.

Since we are utilizing ACM, we can pair the policy constraints with ClusterSelectors to restrict which clusters a particular policy applies to, locking some down while allowing a more relaxed set of rules on others.

HIERARCHY CONTROLLER¹⁰

Hierarchy Controller is the newest add-on to fall under the ACM umbrella, and is still in Open Beta at the time of writing. This controller substantially changes how namespaces work within Kubernetes by allowing for inheritance between namespaces and is driven from the Multitenancy Kubernetes Special Interest Group¹¹. While similar to how abstract namespaces work in a hierarchy ACM repo, this component takes it a step further by allowing objects to be actively replicated from a parent namespace to a child. This is especially useful when using an ACM repo that does not include secrets as part of the repo (due to the security considerations involved). By configuring the Hierarchy Controller to replicate a secret from a

⁷ <https://cloud.google.com/anthos-config-management/docs/concepts/policy-controller>

⁸ For more information on RBAC, see the chapter "Anthos, the computing environment based on Kubernetes"

⁹ <https://www.openpolicyagent.org/docs/latest/kubernetes-introduction/>

¹⁰ <https://cloud.google.com/anthos-config-management/docs/concepts/hierarchy-controller>

¹¹ <https://github.com/kubernetes-sigs/multi-tenancy>

parent to a child or children, a single secret can be replicated to multiple namespaces, simplifying the amount of rework or manual intervention required.

One other useful feature of Hierarchy Controller is the synergy between the controller and Cloud Logging. When the `enablePodTreeLabels` flag is set on the ACM config file, Hierarchy Controller sets flags on all pods, including those in child namespaces. This also indicates how far down the hierarchy tree the pod is located. Let's look at a quick example:

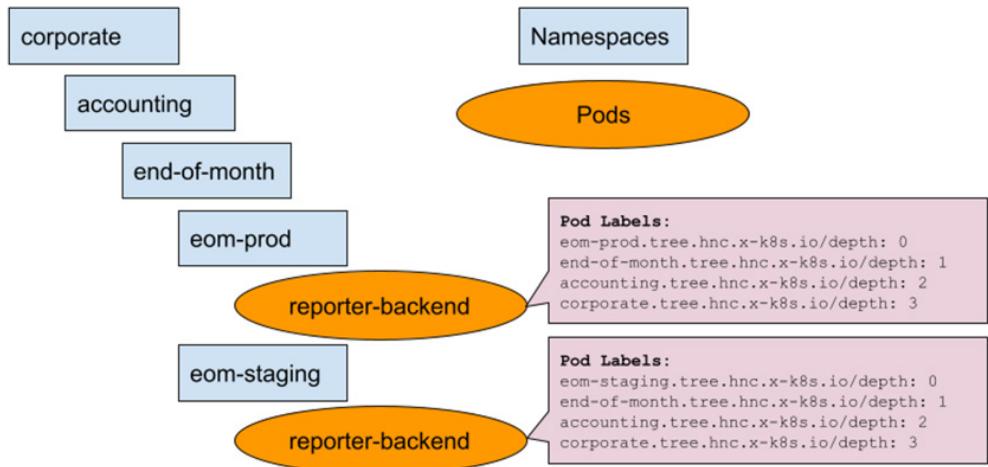


Figure 13.2 Namespaces and pods with hierarchy-related labels when `enablePodTreeLabels` is enabled.

As you can see in this example, we have the 'eom-prod' and 'eom-staging' namespaces as children of the 'end-of-month' namespace. The 'end-of-month' namespace is a child of 'accounting', which is a child of the 'corporate' namespace. As you can see in the image above, the hierarchy labels applied to the 'reporter-backend' pods correspond to the namespace hierarchy. In Kubernetes, you can query by the presence of a label, as well as by the value. So, if we wanted to see all pods under accounting and child namespaces, we can run `kubectl --all-namespaces get pods -l accounting.tree.hnc.x-k8s.io/depth` and it would fetch both instances of the reporter-backend. These labels also appear in Cloud Logging and can be used to fetch pods in multiple namespaces there.

13.3 Examples and Case Studies

ACM is built on top of Kubernetes objects, operating within the cluster lifecycle to efficiently manage the state of the cluster. So far, we have seen the components of ACM; in the next section, let us examine three case studies in greater detail. Each of these fictional companies is either using Kubernetes currently and wants to optimize their deployment or is moving to Kubernetes for the first time; each will use ACM to perform slightly different functions.

Our first company, Evermore Industries, has decided to use a single, large cluster with many nodes. All of their application teams will run their dev, qa, and production environments in parallel namespaces. Evermore wants to leverage GCP resources whenever possible, but their application developers do not have a lot of experience with Infrastructure as Code (IaC) tools. The core infrastructure team does have experience in IaC, but lacks sufficient members to provision everything the application teams desire. Management has decided to allow the application teams to manage portions of their own cloud infrastructure, but still wants to impose certain guidelines and policy rails to prevent out-of-control expenditures. Finally, due to the multitude of applications in the company, and the variable permission levels involved, a service mesh is needed to isolate and control traffic.

Village Linen, LLC was founded approximately two decades ago, and previously ran all of their infrastructure locally in two data centers near their headquarters. Partially due to a change in ownership at one of their data centers, but also due to bad results on past high-traffic shopping days, the company has decided to leverage the cloud to enable rapid scalability, while still keeping several core functions in their one remaining data center. However, corporate leadership wishes to retain the ability to run their entire application stack solely from the local data center, and has mandated that the two environments be as close to identical as possible, and that failover should be as simple and quick as possible. Village Linen also wants to allow developers the freedom to manage their own namespaces, without accidentally affecting other applications and without creating a lot of overhead to approve each change.

Our final company, Ambiguous Rock Feasting, runs several hundred restaurants across the United States and Canada, and has started expanding into Europe and Asia. Currently, their onsite applications (including inventory control, payroll, FOH systems, scheduling, and accounting) are updated via a monthly patch process that pushes the changes to the individual stores. This requires specialized networking and can be temperamental at times. The company wishes to pivot to a solution that does not require their central IT network to maintain persistent connections to the individual stores. They have also had problems in the past modifying their deployment processes and technology when adding a new application to the suite, as well as when trying to deploy targeted versions of the software to different regions.

13.3.1 Evermore Industries

For Evermore Industries, the simplicity of only having one large cluster to manage was key. However, managing the large number of namespaces, users, permissions, and GCP resources was going to prove too much for their IT operations. Thus, they turned to ACM, Policy Controller, and Config Connector to take some of the heavy load.

During a short proof-of-concept at the beginning of their migration, the IT Operations team realized that an unstructured repo would allow them to more easily attach specific policies to individual namespaces (such as applying consistent rules to production namespaces) while also allowing the use of team-based rules without requiring a large amount of duplication. The unstructured repository also permitted the IT Security team to easily restrict which users

had permission to modify specific folders in the repo. Thus, a developer on Team Griffins could not accidentally delete something from Team Unicorns, and no application team members were allowed to modify the global policies. This reduced, but did not eliminate, the amount of configuration review needed for each change.

While Evermore has been using Kubernetes for a few years, their CI/CD process¹² leverages a templating engine (Helm) to deploy directly into the cluster. This has caused a few issues in the past and management has decided to move away from users having direct access to make changes to the prod namespace directly, including the CI/CD service accounts. Since an unstructured repo does not mandate any particular organization for the config elements in the directory, Evermore has decided to continue to leverage their templating engine, but writing the configs directly to the ACM repo and creating pull requests when a new version is to be deployed. Since these can quickly be validated by the Operations team administrators, they can be quickly deployed to the active repo.

In addition, several application teams have expressed interest in using GCP resources to offload some of the workloads for their application. Primarily, these teams are interested in Cloud SQL, Pub/Sub, and Storage buckets. Since the application teams have almost no one with experience using IaC tools, or with GCP in general, Evermore will be leveraging Config Connector to allow the teams to remain in the Kubernetes space for all deployment needs, as well as removing the need to configure and train users to access both Kubernetes and GCP itself.

Let us take a look at the repo outline:

¹² For more information on CI/CD and Anthos, see the chapter "Anthos, integrations with CI/CD"

```
<Git Repo Parent>
  └── bin
  └── policies
      ├── namespace_selectors
      ├── global
      │   ├── rbac
      │   └── namespaces
      └── policy_controller
  └── teams
      ├── griffins (accounting)
      │   ├── rbac
      │   ├── namespaces
      │   └── applications
      │       ├── reconciler
      │       ├── expense_reports
      │       └── accounts_pay_recv
      ├── unicorns (public-facing stores and APIs)
      │   ├── rbac
      │   ├── namespaces
      │   └── applications
      │       ├── storefront
      │       ├── order_cap_and_ful
      │       ├── payment_auth
      │       └── inventory
      ├── dragons (internal apps)
      │   ├── rbac
      │   ├── namespaces
      │   └── applications
      │       ├── timesheets
      │       └── shipping_manager
      └── sylphs (reporting and analysis)
          ├── rbac
          ├── namespaces
          └── applications
              ├── etl
              └── reporting
```

We will not go into detail on every object and file created in this repo, but this gives us a place to start with the structure Evermore has chosen.

In an unstructured repo, the policy directory is not permitted to be at the top level of the git repository, so the company has chosen to place it in a "policies" directory one level down. Within that directory, objects can be placed at any level, so the IT Operations team has placed a set of common namespace selectors in the "namespace_selectors" directory. One of these is the production selector:

```
kind: NamespaceSelector
apiVersion: configmanagement.gke.io/v1
metadata:
  name: production
spec:
  selector:
    matchLabels:
      env: production
```

In order to simplify the work for the application teams, IT Operations also defined a selector that would normally only reference one namespace:

```

kind: NamespaceSelector
apiVersion: configmanagement.gke.io/v1
metadata:
  name: cc-project-ns
spec:
  selector:
    matchLabels:
      evermore.com/is-active-project: true

```

Due to a restriction on the creation of Config Connector objects, CC objects must be specified in a namespace with the same name as the project ID. In order to prevent massive changes in the event that a new project is created, all Config Connector objects defined by the teams use the namespace selector method to choose the appropriate namespace to deploy their objects to. For example:

```

apiVersion: storagebuckets.storage.cnrm.cloud.google.com/v1beta1
kind: StorageBucket
metadata:
  name: evermore-shipping-pk-list-archive
  annotations:
    configmanagement.gke.io/namespace-selector: cc-project-ns
spec:
  versioning:
    enabled: true
  storageClass: ARCHIVE

```

In the "global" directory, we have policies defined by the IT Operations team that apply across the cluster for both RBAC and Policy Controller. In the RBAC directory, the namespace-scoped roles for dev/qa and production are defined separately. The roles leverage NamespaceSelectors to apply to multiple namespaces with one configuration. NamespaceSelectors are utilized via an annotation, and all namespace-scoped objects in an unstructured repo must declare either a namespace directly (via metadata.namespace) or use a NamespaceSelector. Here is the basic production role most developers will receive:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: prod-developer
  annotations:
    configmanagement.gke.io/namespace-selector: production
rules:
- apiGroups: []
  resources: ["pods", "configmaps"]
  verbs: ["get", "watch", "list"]
- apiGroups: ["apps", "extensions"]
  resources: ["deployments", "replicasets"]
  verbs: ["get", "watch", "list"]
- apiGroups: []
  resources: ["secrets"]
  verbs: ["list"]

```

The "policy_controller" directory is where Evermore has decided to put all of their Policy Controller constraints. In addition to constraints requiring definition of container limits and

restrictions on the Istio¹³ service mesh, the IT Operations team has also added the following constraint to force teams to define the environment and team for a given namespace:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredLabels
metadata:
  name: ns-must-have-env-and-team
spec:
  match:
    kinds:
      - apiGroups: []
        kinds: ["Namespace"]
  parameters:
    labels:
      - key: "env"
        expectedRegex: "^(production|development|qa)$"
      - key: "team"
        expectedRegex: "^(griffins|unicorns|dragons|sylphs)$"
```

In addition to requiring the labels to be set on namespaces, this constraint also limits what the valid values are. The Policy Controller infrastructure allows for the creation of custom constraint templates, but Evermore has been able to implement all their desired policies with those from the provided library.

The "namespaces" directory in the global folder holds configurations that are managed by IT Operations and are either not used by any application team, or are used by most or all of the teams. For example, this includes the istio namespace, config connector project namespace, etc... In addition, subfolders underneath this folder contain secrets, config maps, and deployments that would be used across the system. The IT Operations staff has placed some of the general Istio configs (such as the IngressGateway) as well as configs that should be replicated to multiple namespaces (including a contact information ConfigMap that can be mounted via environment variables in each application) in this structure.

For each team's folder, the "rbac" and "namespaces" directories are handled by the individual application teams, though the changes must still be approved by a member of IT Operations via a pull request. The namespace directory holds the namespace declarations for each application and environment, while the rbac directory holds the RoleBindings and ClusterRoleBindings for each user.

Inside each of the application folders (reconciler, storefront, etl, etc...), three folders exist for each environment. These folders are tied into the CI/CD processes that already exist for each application. The CI/CD pipelines were already configured to generate the Kubernetes objects to be loaded onto a cluster, so the teams changed the destination to output to a set of files in the appropriate environment's directory. This CI/CD process also triggers an automatic pull request with the change, which can then be quickly approved and processed by a member of the IT Operations staff.

Evermore chose this configuration for their repo as it best suits their needs at the present time. However, since they are using an unstructured repo, changing the directory structure is

¹³ Istio is explored in detail in the chapter "Anthos Service Mesh: security and observability at scale"

a low-cost option, if needed. Other companies might choose to concentrate all RBAC-related objects into a single directory, or to eliminate the concept of a "team" altogether and organize everything based on the individual applications. The unstructured repo allows the freedom to organize your policies in a manner that makes the most sense for your organization, instead of being restricted to a namespace-centered structure.

13.3.2 Village Linen, LLC

Village Linen has decided to go forward with a hierarchy repo, but they are going to leverage Hierarchy Controller to help with automatic replication of some of their secrets and config maps. They are running GKE on GCP, as well as a GKE on VMware in their existing data center, and want both to operate almost identically inside the cluster.

Disaster recovery is an important issue for corporate management, but management understands that data replication can sometimes be an issue when handling failover. Therefore, the architects have developed a system that allows for users to use both the cloud and the on-premise application layer, but uses a single database cluster in the cloud. The database is replicated locally (the configuration of the replication is not included or covered here), and a configuration change directs the applications to use the standby database located in the data center.

The repo generally looks as follows:

```
<Git Repo Parent>
└── bin
└── policy_directory
    ├── namespaces
    │   ├── rbac.yaml
    │   └── central
    │       ├── namespace.yaml
    │       └── database_location_config.yaml
    ├── applications
    │   ├── service_account.yaml
    │   └── website
    │       ├── namespace.yaml
    │       ├── hierarchy.yaml
    │       └── repo-sync.yaml
    └── inventory
        ├── namespace.yaml
        ├── hierarchy.yaml
        └── repo-sync.yaml
    └── village-linen-ac15e6
        ├── namespace.yaml
        ├── hierarchy.yaml
        └── repo-sync.yaml
    └── cluster
        ├── rbac.yaml
        ├── hierarchy.yaml
        └── constraints.yaml
    └── clusterregistry
        ├── data_center.yaml
        ├── cloud.yaml
        └── selectors.yaml
    └── system
        └── repo.yaml
```

Starting from the bottom, we have a definition for the repo that contains the current version of the policies. In the clusterregistry directory, we have cluster definitions for the cluster in the local data center, as well as the cluster in GCP. We also have selectors defined for each of these clusters so that we can restrict resources in the namespaces directory. The cloud cluster declaration and selector, for example, are:

```
kind: Cluster
apiVersion: clusterregistry.k8s.io/v1alpha1
metadata:
  name: vili-cloud
  labels:
    locality: cloud
---
kind: ClusterSelector
apiVersion: configmanagement.gke.io/v1
metadata:
  name: sel-clu-cloud
spec:
  selector:
    matchLabels:
      locality: cloud
```

In the cluster directory, we have configurations that apply to the cluster as a whole. These include ClusterRoles and ClusterRoleBindings, and Policy Controller constraints. By default, Hierarchy Controller only propagates RBAC Roles and RoleBindings from parent to child namespaces. However, Village Linen wants to use Hierarchy Controller to synchronize secrets and config maps from the 'central' namespace to the 'application' and 'project' namespaces. This way, secrets can be applied directly to the central namespace and replicated automatically without needing to be checked into a git repository. The modified HierarchyConfig is:

```
apiVersion: hnc.x-k8s.io/v1alpha1
kind: HNCConfiguration
metadata:
  name: config
spec:
  types:
    - apiVersion: v1
      kind: ConfigMap
      mode: propagate
    - apiVersion: v1
      kind: Secret
      mode: propagate
```

Moving up to the namespaces directory, we have a top-level file to define a set of RBAC roles to be created in each namespace. Village Linen can then bind these roles at either the 'applications' abstract namespace (which would apply the bindings to the 'website' and 'inventory' namespaces), or to the explicitly defined namespaces to control who has access to these roles.

The directory here defines a total of four namespaces: "central", "website", "inventory", and "village-linen-ac15e6". The last namespace matches the project ID used to deploy resources for use with these clusters (also the location where the cloud GKE cluster is deployed). The two application namespaces share a service account definition, though this will create two separate service accounts, one in each namespace. In the central namespace, we declare a ConfigMap which tells the applications which database to use, either the Cloud SQL or the on-premise cluster.

In each of the "child" namespaces ("website", "inventory", and "village-linen-ac15e6"), the repository has a HierarchyConfiguration object that enables the Hierarchy Controller to propagate objects from parent to child:

```
apiVersion: hnc.x-k8s.io/v1alpha1
kind: HierarchyConfiguration
metadata:
  name: hierarchy
spec:
  parent: central
```

In this case, all the "child" namespaces inherit from a common parent, but it is possible to "stack" these namespaces into a chain. For example, we could introduce another namespace - 'applications' - which inherits from 'central' and modify 'inventory' and 'website' to inherit

from ‘applications’ instead. Performing this type of stacked hierarchy allows for a finer control of what is replicated, as well as adding additional tagging to the logs, if enabled.

When enabling Hierarchy Controller for a given cluster, an additional option can be selected which includes the tree labels on pods. These labels indicate the hierarchy relationship for the pod, and can be used both with command line tools, and in Cloud Logging, to filter for logs that descend from a given namespace.

Since Village Linen is using a hierarchy repo, explicitly defining the metadata.namespace field is not required for objects in the namespaces directory. However, the namespace itself is required to be explicitly defined; Village Linen has chosen to place these definitions in the namespace.yaml files, though that is not required. The objects defined in the website and inventory namespaces are used to enable the Multiple Repository functionality covered below. However, the Cloud SQL cluster is defined in the final folder of the namespaces directory. The cloud_sql.yaml defines the SQL Database, Instance, and User:

```
---
apiVersion: sql.cnrm.cloud.google.com/v1beta1
kind: SQLInstance
metadata:
  name: village-linen
  annotations:
    configmanagement.gke.io/cluster-selector: sel-clu-cloud
spec:
  region: us-central1
  databaseVersion: POSTGRES_9_6
  settings:
    tier: db-custom-16-61440
---
apiVersion: sql.cnrm.cloud.google.com/v1beta1
kind: SQLDatabase
metadata:
  name: village-linen-primary
  annotations:
    configmanagement.gke.io/cluster-selector: sel-clu-cloud
spec:
  charset: UTF8
  collation: en_US.UTF8
  instanceRef:
    name: village-linen
---
apiVersion: sql.cnrm.cloud.google.com/v1beta1
kind: SQLUser
metadata:
  name: village-linen-dbuser
  annotations:
    configmanagement.gke.io/cluster-selector: sel-clu-cloud
spec:
  instanceRef:
    name: village-linen
  password:
    valueFrom:
      secretKeyRef:
        name: db-creds
        key: password
```

As you can see, Config Connector allows references both to other Config Connector objects (the instanceRef declarations above), and secrets. Config Connector can also pull information from ConfigMaps. For security reasons, the db-creds Secret is not stored in the ACM repo. However, because the Hierarchy Controller is configured to replicate Secrets and Config Maps, we can manually create or update the secret in the central namespace, and the Hierarchy Controller will handle the replication to the application and project namespaces. When Config Connector reconciles the next time, the new password will be used for the user.

All of the Config Connector configurations include an annotation with a cluster-selector. This references the cluster selector defined in the clusterregistry directory for the cloud installation of GKE. Since Config Connector works to create GCP resources from Kubernetes objects, it is only active on GKE in GCP clusters. Since the company did not enable Config Connector in the local cluster, trying to deploy these resources would fail. Even if Config Connector was enabled on the local cluster, deploying the resources there should not have any effect, and would probably cause more troubleshooting issues, so we only deploy them to the cloud cluster.

With this configuration, if Village Linen needs to switch from the cloud database back to a local database, a simple change to the database_location_config should be made. After deploying and pushing the updated configuration, the individual applications would need to be restarted.

In the directory structure outlined above, each application namespace contains a repo-sync file. These objects are used to implement Multiple Repository Mode:

```
apiVersion: configsync.gke.io/v1alpha1
kind: RepoSync
metadata:
  name: repo-sync
  namespace: website
spec:
  git:
    repo: source.developers.google.com/p/village-linen-ac15e6/r/website
    branch: master
    auth: gcnod
    secretRef:
      name: acm-website-repo
```

By leveraging Multiple Repository Mode, Village Linen is able to create separate repositories for each namespace, allowing application teams a simpler experience when modifying Kubernetes objects (including the deployment, services, persistent volumes, etc...) for their application. This arrangement also restricts the teams from accidentally deploying something outside of their namespace. The Operations team is still able to add items to each namespace using the core repository, and these exist in parallel with the namespace-specific objects. In event of a conflict, the core repository's version is the one used, preventing the application teams from overriding policies, service accounts, secrets, etc... that the Operations team has already defined. In addition, since the Operations team has restricted what objects the namespace's worker can modify using RBAC, the individual repositories are

sandboxed to only control a limited set of objects, and cannot grant themselves permissions unless the Operations team allows it.

For Village Linen, ACM provides a convenient location for all core configurations to be centrally located and updated, while freeing the application teams to control their own namespaces. It also provides a convenient audit trail when configurations change. In the event that either the local data center cluster, or the cloud cluster, fail for any reason, a new cluster can be spun up and connected to the ACM repo, rapidly and automatically deploying the full operational stack.

13.3.3 Ambiguous Rock Feasting

For Ambiguous Rock Feasting (A.R. Feasting), managing their expanding set of restaurants, and the technology deployed within, has become more and more difficult over the past few years. The company now feels that the implementation time for the technology pieces, plus the additional operational overhead each new location places on their IT Operations team has become unsustainable. Therefore, they are moving to a more flexible model that will scale better.

The restaurants already ran Kubernetes clusters on their local servers, but updating the deployed applications or troubleshooting issues caused a significant time loss for each location. Therefore, A.R. Feasting has pivoted to using ACM to manage the individual clusters. In general, the repository layout matches that of Village Linen, except A.R. Feasting is not using Hierarchy Controller.

When deploying the ACM operators, each location's cluster was given a dedicated name, such as "arf-043-01a". The IT Operations team then labeled these clusters using Cluster definitions in the policy repo:

```
kind: Cluster
apiVersion: clusterregistry.k8s.io/v1alpha1
metadata:
  name: arf-043-01a
  labels:
    sales-area: us-midwest
    country: us
    region: texas
    city: austin
    location-code: alpha
```

Or this one for "arf-101-01a":

```
kind: Cluster
apiVersion: clusterregistry.k8s.io/v1alpha1
metadata:
  name: arf-101-01a
  labels:
    sales-area: emea
    country: uk
    region: england
    city: london
    location-code: alpha
    is-rollout-tester: true
```

The IT Operations team then defines selectors based on these labels, some of which are included below:

```
---
kind: ClusterSelector
apiVersion: configmanagement.gke.io/v1
metadata:
  name: rollout-testers
spec:
  selector:
    matchLabels:
      is-rollout-tester: true
---
kind: ClusterSelector
apiVersion: configmanagement.gke.io/v1
metadata:
  name: non-testers
spec:
  selector:
    matchExpressions:
    - key: is-rollout-tester
      operator: DoesNotExist
---
kind: ClusterSelector
apiVersion: configmanagement.gke.io/v1
metadata:
  name: country-us
spec:
  selector:
    matchLabels:
      country: us
```

These selectors are then used to control deployments of new versions of applications, or to only deploy certain applications in certain regions. For example, only restaurants in the United States have drive-thrus. Therefore, the drive-thru management application only needs to be deployed to the "country-us" clusters. The company has also decided to have certain selected stores be test beds of new software, as indicated by the "is-rollout-tester" flag on their cluster. An example deployment for an application is included here. However, some portions of the template have been removed as they are identical between the two examples:

```

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: foh-engine
  annotations:
    configmanagement.gke.io/cluster-selector: rollout-testers
spec:
  template:
    spec:
      containers:
        - name: engine
          image: gcr.io/ambiguous-rock/foh/engine:v2.1.0
          imagePullPolicy: IfNotPresent
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: foh-engine
  annotations:
    configmanagement.gke.io/cluster-selector: non-testers
spec:
  template:
    spec:
      containers:
        - name: engine
          image: gcr.io/ambiguous-rock/foh/engine:v2.0.3
          imagePullPolicy: IfNotPresent

```

The differences between these two deployments are the cluster-selector used and the version of the engine image. Looking back at the cluster-selectors defined, "rollout-testers" and "non-testers" do not overlap. If we had not defined a non-testers group and left the annotation off the second deployment above, we would have a collision for the rollout-testers since both deployments would have been valid.

Since ACM logs the changes it makes using the same logging standards as Kubernetes, and with A.R. Feasting restaurants forwarding their logs to Cloud Logging, the IT Operations team is able to set up monitoring using Cloud Monitoring to determine the status of specific applications and versions on the various clusters. Using this dashboard, they can quickly diagnose where potential problems might be (such as a power outage at a store) and work more efficiently.

13.4 Conclusions

Organizations face increasing complexity managing their IT environments. Leveraging Anthos Config Management with Kubernetes clusters, whether on-prem, in Google Cloud, or another cloud provider, provides administrators with a familiar, declarative method to control the foundations of their clusters. This chapter has provided a broad overview of the service, some of the reasons to adopt a strategy incorporating ACM, and a couple of examples to illustrate the power of Anthos Config Management.

However, we have not fully explored the capabilities and possibilities of Anthos Config Management: doing so would take a book all its own. Ultimately, ACM is intended to make your business more efficient and to reduce the complexity of managing your clusters. This chapter should have provided you with ideas on how to leverage ACM in your own organization, as well as some examples on how to drive adoption.

For more information and examples on various topics touched on in this chapter, see the corresponding section of this book:

- Policy Controller -- Anthos for Security/Policy
- Cloud Logging -- Operation Management with Anthos

13.5 Summary

- As the number of clusters an organization manages increases, enforcing best practices or providing core functionality becomes exponentially more difficult.
- Modern development practices, as well as the security landscape, encourages organizations to adopt technologies that can rapidly adapt and deploy changes.
- Anthos Configuration Management (ACM) is a core component of the Anthos platform, intended to provide the security and observability that infrastructure, security, and operations teams desire, while also giving development teams the ability to deploy their applications with minimal additional hurdles.
- ACM can be deployed in two different modes, both offering distinct advantages:
 - Hierarchical mode allows for easy deployment of a single resource across multiple namespaces and requires a logical collection of namespaces to be effective.
 - Unstructured mode allows for developers and administrators to more easily pick and choose which namespaces to deploy components to, with the drawback of needing to be explicit which namespace(s) to use. This mode is also compatible with many templating frameworks that may not function properly in Hierarchical mode.
- ACM includes custom resources allowing for greater control over which namespaces and which clusters to apply configuration elements to.
- ACM also brings in additional components based on open-source tools:
 - Config Connector provides Infrastructure-as-Data capability to a Kubernetes cluster, allowing for the provisioning of Google Cloud resources by declaring a Kubernetes resource.
 - Policy Controller, based on the open-source Open Policy Agent, gives administrators a convenient tool to create and enforce policies across their clusters.
 - Hierarchy Controller is the result of an initiative to provide an alternative method of replicating resources between namespaces in a cluster. Since storing secrets in a git repo is an anti-pattern, Hierarchy Controller definitions allow an organization to define a secret or configuration once and have it replicated to the descendant namespaces automatically.

- Three case studies explored different reasons for using and implementation of ACM:
 - Evermore Industries wants to reduce the number of users directly working in their Production environment, as well as allow development teams to provision certain Google Cloud resources directly. They are currently, and will continue to use, a templating engine for generating their Kubernetes configurations.
 - Village Linen, LLC is leveraging ACM both on and off-premise, as well as Hierarchy Controller to manage the replication of ConfigMaps and Secrets within both sets of clusters.
 - Ambiguous Rock Feasting is well-verses in Kubernetes, but is making specific use of the ClusterSelectors feature of ACM in order to more precisely control where their applications are deployed.

18

Migrate for Anthos and GKE

by Antonio Gulli

This chapter covers

- The benefits of using Migrate for Anthos
- Recommended workloads for migration
- Migrate for Anthos architecture
- Using Migrate for Anthos to migrate a workload
- Best practices for Migrate for Anthos

Containers provide developers multiple advantages, including speed increases when deploying and provisioning workloads, higher resource utilization, portability, and cost efficiency when compared to virtual machines (VMs).

However, many customers have several thousands of applications written over many years running on VM infrastructure using heritage frameworks. For those customers, it would be too time consuming and expensive to rewrite these applications. Therefore there is a need for tools to modernize workloads and to provide the benefits of modern cloud native environments without the cost of rewriting applications from scratch. The whole value proposition is to decrease customers' time to market during transformation projects and to augment and optimize traditional workloads with modern cloud infrastructure and services.

At Google, we believe that modernization doesn't have to be all or nothing. Microservices architectures (Chapter [Cloud is a new computing stack]) structure an application as a collection of services that are highly maintainable and testable, loosely coupled via APIs, and independently deployable. However, even if you don't use a microservice architecture from the beginning, you can convert your applications into containers and still have many of the benefits typically obtained by cloud native applications. This is what we'll see in this chapter.

Migrate for Anthos (in short, M4A) is a tool that helps extract your legacy workloads from your VMs, and automatically transforms them into containers including all you need for the

execution. This includes the runtime, system tools, libraries, code, configurations and settings. Once your application is migrated, it is possible to run it on Anthos, either on Google's Kubernetes Engine (GKE), on-prem, or on other clouds.

In this way, the management of infrastructure, the hardware and the OS/Kernel are “abstracted away” and delegated to your cloud provider(s). Furthermore, M4A generates the artifacts that enable you to switch to modern software development leveraging CI/CD pipelines and shifting from package management to container/image-based management. You can think about M4A as an accelerator to the ultimate goal of modernization, and this acceleration can happen at scale with many legacy applications migrated all together in bulk. M4A can thereby let you operate thousands of legacy applications at scale by modernizing the underlying compute infrastructure, network, storage, and management. M4A supports both Linux and Windows migration to containers. Source environments can include either a GCP Compute Engine environment, a VMware vSphere environment, a Microsoft Azure VM environment, or an Amazon Elastic Compute Cloud (Amazon EC2) environment. All workloads are directly migrated with no need of either having access to the original source code, or rewriting your workloads or manually containerizing your workloads.

Most of the migration work is done automatically, and users can modify the generated migration plans to fine tune the desired modernization. While the migration process executes, the application can continue to run uninterrupted, and you can be up and running with the containerized app in minutes. If you want to go back to the initial state, you can roll back with no data lost. Migrated workload container images that are generated by M4A are automatically deployed into Google Container Registry (GCR), or other local repositories and can run in any environment without installing any M4A component on target workload clusters. Of course, the whole M4A process can be performed and monitored via the Google Cloud Console UI.

Now that we have set up the context, let's have a look at M4A's benefits in detail.

18.1 Migrate for Anthos benefits

M4A allows us to unbundle more and more infrastructure from inside VMs and manage it with Kubernetes. This modernization unifies app management with modern IT skills. Indeed, it is possible to promote legacy applications to first class objects in a cloud-native environment with no need of changing/accessing the code. The unlocked improvements at scale include:

- Define the infrastructure with declarative APIs, dynamic scaling, self-healing and programmatic roll-out;
- Improved workload density on the datacenter allowing for better resources utilization.
- Maintain the infrastructure metrics, business metrics, network policies, and project access control;
- Integrate CI/CD pipelines and build systems.

M4A benefits are transparently gained in different classes: density, cost, security, infrastructure, automation, service management, and day 2 operations. Let's discuss each class in more detail.

Density. VMs are abstracted from the underlying physical hardware. Whereas legacy bare-metal servers can only support a single application, hypervisor¹ virtualization allows the applications of multiple VMs to run on a single bare-metal server, sharing the underlying resources. Let's have a look at utilization:

- Typically, bare-metal utilization is at 5-15%, while virtual machines can increase it up to 30%;
- Containers enhance workload density, since multiple containers are typically run on the same VM or bare-metal server. In addition, the density is increased because things like OS/kernels, networking and storage are abstracted away, as discussed earlier in the chapter.

The actual utilization gains will depend on several specific factors in your environment. Frequently, many organizations report significant gains as they move from physical servers, to VMs, to containers. For instance, the Financial Times' content platform team reported a reduction of server costs by 80% by adopting containers².

Cost. Density increase results in immediate cost savings for infrastructure. As discussed, the density increase is a result of two facts: multiple containers can be packed on the same physical machine, and many software layers are abstracted away. The increase in density means better usage of the available resources, requiring fewer servers, leading to an overall cost savings

Moreover, after migration, legacy applications are promoted to first-class citizens together with cloud native applications. As a side effect, there is no need to maintain two working environments (both the legacy and the modern one). So, the unified management of workloads allows further cost reduction at scale.

If you want to know more about cost reduction, then you can check the Google Cloud pricing calculator³ to estimate your monthly charges, including cluster management fees and worker node pricing for GKE.

Infrastructure. After moving the application to containers, it is possible to see the whole "infrastructure as code" knowing how the applications, processes and dependencies. Any change in infrastructure can be stored in a repository (in short, repo) with operations such as commit, push, pull, merging, and branching, applied to any part of your infrastructure, including config files. Traditionally, maintaining an infrastructure is a considerable cost for enterprise. Moving to Infrastructure defined as Code (IaC) allows teams to implement DevOps/SRE methodologies which leads to further cost savings at scale due to enhanced agility and reliability.

Automation. Managing virtual machines is an expensive, time consuming and error-prone process due to the need to patch and upgrade infrastructure either manually or with a plethora of third party tools which might increase the level of complexity. With Anthos this

¹ A hypervisor is software, firmware or hardware that creates and runs virtual machines.

² See <https://www.computerworld.com/article/3427686/how-containers-cut-server-costs-at-the-financial-times-by-80-percent.html>

³ See <https://cloud.google.com/products/calculator>

cost is significantly reduced via containers and automation. For instance, on GKE for Anthos many operations are run automatically on customers' behalf, including:

- node auto-repair, for keeping the nodes in your Kubernetes cluster in a healthy, running state;
- node auto-upgrade, for keeping the nodes in your Kubernetes cluster up to date with the cluster master version when your master is updated on your behalf;
- node auto-security, as security patches can be automatically deployed in a transparent way
- node auto-scale, for dynamically scaling your Kubernetes nodes up and down according to instantaneous load increase/decrease;
- node auto-provisioning, for automatically managing a set of Kubernetes node pools on the user's behalf;
- progressive roll-out/canary/a-b testing, for programmatic roll-out of applications on Kubernetes clusters, with roll-backs in case of problems.

As a rule of thumb, increasing the automation in your infrastructure will reduce the risk of accidental errors, enhance the reliability of the whole system, and reduce the cost.

Security. Once you move into containers a number of security operations are facilitated. On Anthos, these operations include:

- Security-optimized node kernel and OS updates. Anthos offers automatic operating system upgrades and kernel patches, freeing you from the burden of maintaining the OS, which is a substantial cost if your servers' fleet is large. VMs have to run a full guest operating system even if they only host a single app. Instead, containers reduce operational cost because there is no need to run an OS;
- Binary authorization and container analysis. Anthos offers a deploy-time security control that ensures only trusted container images are deployed in your environment;
- Zero trust security model. Anthos Service Mesh (ASM), covered in chapter 5: security and observability at scale, and the core capabilities of Kubernetes allow us to provide network isolation and TLS security without changing the application code;
- Identity-based security model. With ASM, you can get security insights, security policies and policy-driven security. These cases are discussed in detail in chapter 5: Anthos Service Mesh: security and observability at scale

Transparently increasing security will reduce the risk of an incident, and the associated high cost.

Service Management. After moving to the containers, it is possible to use Anthos Service Mesh (see Chapter 5: Anthos Service Mesh: security and observability at scale) to know where your services are connected, and you can get telemetry and visibility into your application without changing code. Some of the benefits transparently gained on Anthos are:

- Encryption: Applications can communicate with end-to-end encryption with no need of changing the code.
- Integrated logging and monitoring: You can get uniform metrics and traffic flow logs across your application.

- Uniform observability: It is possible to observe service dependencies and understand the critical customer journey and how they impact your service-level agreement (SLA⁴) from end to end. You do this by setting up service-level objectives (SLOs⁵) on your applications.
- Operational agility. You can dynamically migrate traffic, do circuit breaking⁶, retries within your environment, do canaries⁷ and A/B testing. Taking a canary as an example, you can move, based on a weighting, a certain amount of traffic from one service to a newer version of that service.
- Bridging. You can leverage service meshes to bridge traffic between on-prem and multiple clouds.

In general, adopting a service mesh will enhance your understanding of your infrastructure, which might become quite complex over time.

Day 2 Operations. As discussed in chapter 6: Operations Management in Anthos, Anthos relieves the burden of Day 2 operations. Once your application is migrated you can benefit from many Google Cloud Platform (GCP) capabilities. In particular:

- Cloud Logging and Cloud Operations. Cloud Logging allows you to store, search, analyze, monitor, and alert on log data and events from Google Cloud, while Cloud Monitoring provides visibility into the performance, uptime, and overall health of cloud-powered applications. After migration, both the services are available via configuration change only.
- Unified policy and integrated resource management. Anthos offers a declarative desired-state management via Anthos Config Management, which is covered in chapter 14: Anthos Config Management . Declarative means that the user defines only the desired end-state, leaving Anthos the definition of optimized steps to implement the changes. Anthos Config Management allows you to automate and standardize security policies and best practices across all of your environments with advanced tagging strategies and selector policies. As users, you will have a single UI for defining unified policy creation and enforcement (Chapter [Anthos for Security/Policy]).
- Cloud Build to implement Day 2 maintenance procedures. Cloud Build offers control over defining custom workflows for building, testing, and deploying across multiple environments and multiple languages.
- CI/CD pipelines. Anthos integrates CI/CD pipelines for enhancing the agility of your environment. This allows you to have smaller code changes, faster turn-around on feature changes, shorter release cycles, and quicker fault isolation.

⁴The service-level agreement (SLA) is the agreement that specifies what service is to be provided, how it is supported, times, locations, costs, performance, and responsibilities of the parties involved.

⁵Service-level objectives (SLO) are specific measurable metrics of an SLA such as availability, throughput, frequency, response time, or quality.

⁶Circuit breakers area design pattern used in modern software development. It is used to detect failures and to prevent a failure from constantly reappearing, during maintenance, external failures or unforeseen problems.

⁷Canary deployments are a pattern for rolling out releases to a subset of users or servers. The key idea is to first deploy the change to a small subset of servers/users, test it, and then roll the change out to the remaining servers/users.

- [GCP Marketplace](#). Anthos is integrated within GCP Marketplace, including a private catalog for deploying new applications with a single click, this is covered in detail in chapter 19: Marketplace for Anthos.

One of the most important benefits of M4A is that legacy applications are promoted to first-class objects with the same benefits in terms of Day 2 operations of what is typically expected in a cloud-native environment.

In this section, we have discussed the benefits of using Migrate for Anthos to modernize VM workloads in place and containerize your application automatically with no need to rewrite it. As discussed, containerized applications increase agility and efficiency because they require less management time when compared to VMs. Besides, they offer an increase of infrastructure density and a related reduction in cost. Finally, containerized applications can benefit from the service mesh in terms of observability, Day 2 operation streamlining, and uniform policy management and enforcement across environments.

In the next section, we will take a deep dive into what workloads are best suited for migration.

18.2 Recommended workloads for migration

Modernizing applications for the cloud can be difficult. Complex apps are often multi-tier and typically have multiple dependencies. Data has gravity and migration might depend on large volumes of data, either in files or in databases. Legacy applications might have been written in outdated code with legacy frameworks, and in many situations, the code itself might not be available for recompiling in a modern environment. In this section, we are going to discuss various types of applications that are particularly suitable for automatic migration.

Stateless web front-end. A first suitable class consists of stateless applications such as web servers and similar applications serving customer traffic. Containers are generally more lightweight than virtual machines and it is consequently easier to scale them up and down according to various load situations.

Multi-VM, multi-tier stacks, and business logic middleware. In this class there are multi-tier web service stacks such as LAMP (Linux, Apache, MySQL, PHP/Perl/Python) or WordPress, because they can be broken down into multiple independent containers. Also in this class are J2EE Middleware such as Java TomCat and other COTS (Commercial Off-The-Shelf) apps. In M4A's jargon, we typically say that the sweet-spot application categories include multi-tier web-based enterprise applications.

Medium- to large-sized databases. Databases such as Mysql, Postgres, and redis are supported; the data layer can be typically separated from the compute layer and containers can therefore help to manage lightweight computation.

Low duty-cycle and bursty workloads. Containers are the preferred solution in any situation where there are intermittent rises and decreases in compute activity. This is because they are more lightweight than virtual machines. So, Migrate for Anthos should be

considered any time when we need to rapidly set up dev/test environments, training environments, or labs.

Overall, we can say that **ideal migration candidates** include the following:

- Workloads where modernization through a complete rewrite is either impossible or too expensive
- Workloads with unknown dependencies that could break something if touched
- Workloads that are maintained, but not actively developed
- Workloads that aren't maintained anymore
- Workloads without source code access

Sometimes it might be difficult to have automatic migration. This is true in particular if there are dependencies on specific kernel drivers, specific hardware, or if software licenses need to be tied to certain hardware or virtual machines. Another relevant case are VM-based workloads that require the whole Kubernetes node capacity, including high performance and high memory databases (such as SAP HANA). Except for these very specific cases, all the other workloads should be considered for a migration from VMs to containers.

In this section we have rapidly reviewed relevant workloads suitable for migration. In the next few paragraphs, we will discuss the migration architecture and some real examples of migration.

18.3 M4A Architecture

In this section we will discuss a typical migration workflow and how a virtual machine is transformed into a number of different containers. Once the migration is finished, the generated artifacts can run anywhere.

Note: There is no longer the need to install the migration components on target clusters.

18.3.1 Migration workflow

The migration consists of three phases: the setup, actual migration, and optimization (see Fig.1). Let's take a closer look at each step.

During the **setup phase**, a processing cluster is created and the migration sources are defined. Among the supported migration sources we have VMware, AWS EC2, Azure VM, GCE VM, and local VMware. As of version 1.9 of Anthos Migrate, the operating systems supported for migrations are RHEL, CentOS, SUSE, Ubuntu, Debian and Windows. The list is, however, always expanding and it is good to check online for the latest list⁸.

During the setup phase, we need to set up the cloud landing zone, taking into account, identity, network configuration, security and billing. There are several tools that can help to make Infrastructure as a Code task more automatic, including Cloud Foundation Toolkit⁹ and Terraform¹⁰. These templates can be used off-the-shelf to rapidly build a repeatable

⁸ See <https://cloud.google.com/migrate/anthos/docs/compatible-os-versions>

⁹ See <https://cloud.google.com/foundation-toolkit>

¹⁰ See <https://www.terraform.io/>

enterprise-ready foundation, depending on your specific needs. Furthermore, during setup, you need to discover the workloads that you want to migrate. The desired workloads can be identified either manually or via discovery tools such as Stratazone¹¹ (now acquired by Google), ModelizeIT¹², Cloudomize¹³, or CloudPhysics¹⁴. Since version 1.5 of M4A, native discovery tools are included, which will be covered in the next section.

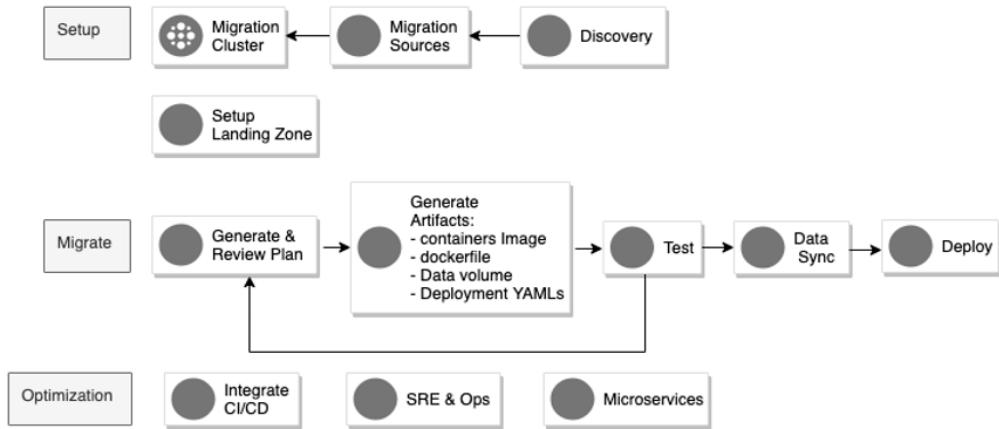


Figure 18.1: Setup and Migrate for Anthos

During the **migration phase**, M4A is run and new container images are automatically generated, together with a Dockerfile, data volumes, and new YAML files for deployment. We will see the details in the next sections, where we will cover both the Command Line Interface (CLI) and Graphical User Interface (GUI) processes. Once these artifacts are automatically generated they can be tested on GKE/Anthos or in another cloud, and if everything looks good, they can be deployed to GKE/Anthos. It's also important to note that data is automatically moved and synchronized as part of the migration process.

Anthos supports live migrations which means that applications can be migrated to modern environments without any interruptions. Behind the scenes, Migrate for Anthos creates a snapshot for the source VM, and this source VM is left running and operational with no need for downtime. In the meantime, all the storage operations are done on that snapshot of the VM. All workloads are directly migrated without requiring the original source code.

During the **optimization phase**, deployed artifacts can be integrated with CI/CD platforms such as Jenkins¹⁵, Spinnaker¹⁶, Gitlab CI/CD¹⁷ and others according to your specific preferences (see Chapter [Anthos, integrations with CI/CD]).

¹¹ See <https://www.stratozone.com/>

¹² See <https://www.modelizeit.com/>

¹³ See <https://www.cloudomize.com/en/home/>

¹⁴ See <https://www.cloudphysics.com/>

¹⁵ See <https://www.jenkins.io/>

18.3.2 From virtual machines to containers

A typical VM consists of multiple layers (see Figure 18.2, left side). At the top there are the applications run by users, together with cron jobs, config files and user data. Just below there are multiple services, including services running in the user space and SysV or Systemd¹⁸ service. Then, there is a logging and monitoring layer on the top of the OS kernel and OS drivers. At the very bottom there is the virtual hardware including networking, storage with logical volumes on various filesystems, CPUs, and memory.

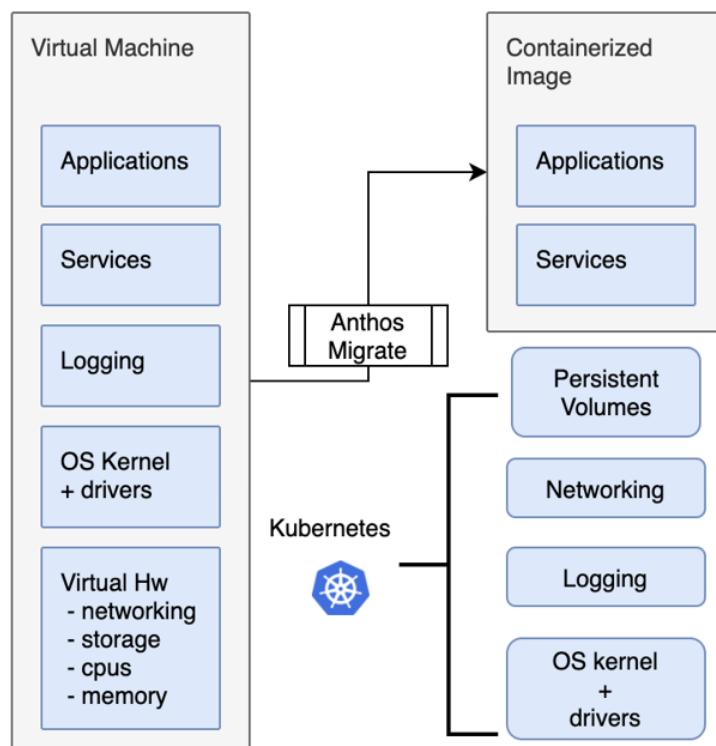


Figure 18.2: Anthos Migrate: From virtual machines to containers

For each application, M4A produces CI/CD artifacts in the form of a Docker Image, a Dockerfile and deployment YAML files. These include the applications, the user services and the persistent volumes (see Figure 18.2, right side). In particular, storage is refactored into a Kubernetes-supported persistent volume. Common functions such as networking, logging and monitoring, and OS kernel + drivers are *abstracted away* and delegated to Kubernetes

¹⁶ See <https://spinnaker.io/>

¹⁷ See <https://docs.gitlab.com/ee/ci/>

¹⁸ Two common Unix service layers, see <https://en.wikipedia.org/wiki/Systemd>.

management. A persistent volume is mounted using the Migrate for Anthos Container Storage Interface (CSI) driver (see Chapter [[Anthos, Data and Analytics]]). Then data is streamed directly from the source VM filesystem. Internally, Migrate also takes care of generating Command Line Input (CLI) and Customer Resource Definitions (CRD¹⁹). Logically, the migration produces two layers of containerized image: the first layer is the captured user-mode system, whereas the second layer is the runtime environment for migration, together with all the necessary CRDs. However, after migration there is no need to maintain the second layer and the generated artifacts can run on any Kubernetes-conformant distribution²⁰.

VM-related files and components that are not essential for the application in the Kubernetes environment are explicitly left out. In fact, this exclusion implies benefits. As discussed, containers allow higher density and cost reduction due to their lightweight nature when compared to virtual machines. The application lifecycle stays within the system container. Once ported, applications can run on either any Anthos environment (On-prem, on GCP etc) or any Kubernetes-conformant distribution independently of Migrate for Anthos deployment.

18.3.3 A look at the Windows environment

Version 1.4+ of Migrate for Anthos supports the migration of workloads from Windows servers to GKE/Windows. Similarly to Linux environments, the goal is to automate the re-platforming of the workload and then integrate it with a more modern cloud environment. At the end of 2021, all the Windows server platforms from Windows Server 2008r2 to Windows Server 2019 are available as targets. Currently, only GCE is supported as a source for Windows applications modernization, with direct support for on-prem VMWare, AWS, and Azure planned for the next version . However, you can use Migrate for Compute Engine²¹ to migrate or clone a Windows VM from other sources into Compute Engine, and then migrate the resulting VM into a container. The good news is that a migrated Windows VM does not have to be configured to run on Compute Engine.

Behind the scenes, the migration works by extracting the ASP.net application and the IIS configuration and applying them on top of the official Windows 2019 server image. M4A for Windows works well with applications developed with IIS 7+, ASP.NET, especially with Web and business logic middleware. Sweet spots for migration are stateless tiers of Windows web applications, application servers and web frontend.

18.3.4 A complete view on the modernization journey

Now that we have discussed the modernization journey with both Linux and Windows workloads, we provide a comprehensive view which also includes Mainframes (see Figure 18.3).

¹⁹ CRD is an extension of the Kubernetes API that is not necessarily available in a default K8s installation. CRD is a standard mechanism to customize Kubernetes in a modular way. See <https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/>

²⁰ As of 2020, there are over 90 certified Kubernetes-conformant distributions. See <https://www.cncf.io/certification/software-conformance/>

²¹ See <https://cloud.google.com/migrate/compute-engine>

If the source application is a modern app then it is containerized, integrated with CI/CD and can run on Anthos where integration with the whole ecosystem can facilitate further refactoring into a microservice environment.

If the source application is a traditional monolithic application in either Linux or Windows, then we can use M4A to containerize it.

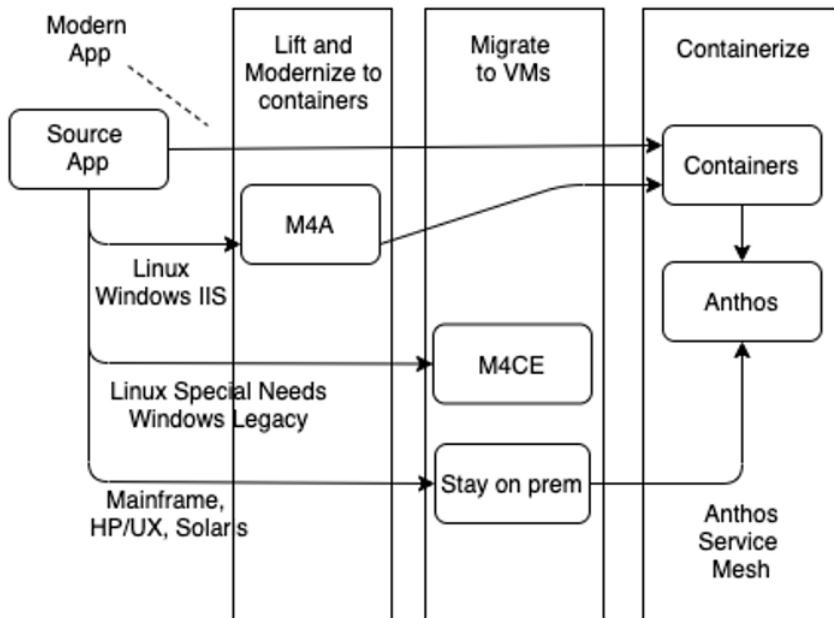


Figure 18.3: A complete modernization journey

If the source application is a Linux/Window application with particular needs either for specific drivers or legacy support then it is still possible to directly migrate the VM to GCP either in Bare Metal (BMS)²² or in GCVE (Google Cloud VMware Engine)²³ and later on manually refactor the application into microservices.

18.4 Real world scenarios

In this section, we are going to review real examples of migration with M4A. First the migration fit assessment tool²⁴ will be introduced. Then, a hands-on session on how to migrate is provided, for. both Linux and Windows .

²² See <https://cloud.google.com/bare-metal>

²³ See <https://cloud.google.com/vmware-engine>

²⁴ See <https://cloud.google.com/migrate/anthos/docs/fit-assessment>

18.4.1 Using the fit assessment tool

In this section we present a self-service fit assessment tool used to determine the workload's suitability for migration to a container. The tool consists of a utility called *mfit* - a standalone Linux CLI tool used to drive the assessment process along with dedicated Linux and Windows data collection scripts that are invoked either automatically by *mfit* or manually - depending on the scenario.

The fit assessment tool generates a report presenting pre-VM assessment results including both a score for the VM's suitability for migration to a container and recommendations on resolving various obstacles.

The table below provides a summary of possible fit scores:

Table 18.1: Fit scores produced by the fit assessment tool

Fit score	Description
Score 0	Excellent fit
Score 1	Good fit with some findings that might require attention.
Score 2	Requires minimal effort before migrating.
Score 3	Requires moderate effort before migrating.
Score 4	Requires major effort before migrating
Score 5	No fit.
Score 6	Insufficient Data collected to assess the VM.

FIT ASSESSMENT PROCESS

The fit assessment process consists of 3 distinct phases, the discovery, assessment and reporting. Each of the phases is detailed below.

1. **Discovery** - gather data about the VMs and store it in a local lightweight database for use in the next phases (located in the `~/.mfit` folder by default). There are two types of data discovery methods:
 - a) (Optional) **VM/Inventory level discovery** - use the *mfit* tool to pull inventory and configuration about VMs in one or more vCenters using the vSphere API. Future versions of the fit assessment tool *will support pulling inventory from public clouds such as GCP, AWS and Azure as well. This method is optional and fit assessment can be performed without it, albeit somewhat less thoroughly.*

- b) **Guest level data collection** - consists of running a data collection script inside the VM to be assessed. The fit assessment tool provides a bash script for Linux VMs and a powershell script for Windows VMs. Each script gathers data about the OS configuration and about running services, processes, installed packages, etc. and produces a single archive file to be imported into the database by the *mfit* tool and used later on in the assessment phase. The process of running the collection script and importing the data can be either manual or automated using the *mfit tool in the following scenarios:*
 - Linux only - run collection script and collect results via SSH
 - *Linux and Windows on vSphere - run collection script and collect results using the vSphere API.*
2. **Assessment** - use the *mfit tool to analyze the collected data, apply a set of fit assessment rules*²⁵ for each VM assessed.
 3. **Reporting** - use the *mfit* tool to produce a report to present the assessment outcomes in either CSV, HTML or JSON format. The latter can then be displayed in Google's cloud console²⁶.

Now that you know how the tool will discover and report the workloads, we can move on to how to use the tool so you can start your migration journey.

BASIC TOOL USAGE INSTRUCTIONS

Following is a basic overview for using the fit assessment tool. Full documentation is available at: <https://cloud.google.com/migrate/anthos/docs/fit-assessment>.

Note that appending `--help` to each *mfit* command will show detailed command usage, including all possible flags and sub commands.

INSTALLATION

At the time of writing, you can download the *mfit* tool (version 1.9) on your workstation used for driving the fit assessment with the following commands:

```
wget https://anthos-migrate-release.storage.googleapis.com/v1.9.0/linux/amd64/mfit
chmod +x mfit
```

INVENTORY DISCOVERY

Run the following command to perform discovery of all VMs in a vCenter

NOTE If your Virtual Center is using a certificate that is not trusted by the machine you are running *mfit* on, you can add the `-i` option to ignore SSL errors.

```
./mfit discover vsphere -u <vcenter username> --url <https://vcenter-host-name-or-ip>
```

²⁵ For a list of fit assessment rules see: https://cloud.google.com/migrate/anthos/docs/fit-assessment#assessment_operation

²⁶ Google cloud console report viewer: <https://console.cloud.google.com/anthos/migrate/report>

You will be prompted to enter the vCenter password, and once executed, you will see a summary of the discovery process:

```
Running preflight checks...
[✓] DB Readable
[✓] DB Writable
[✓] Available Disk Space
[✓] Supported VC version

[+] Found 27 VMs
Collecting data...
27 / 27 [-----] 100.00% 13 p/s
```

You may be wondering what actually happened since the output is limited from the discovery process, only telling you that the pre-checks have passed and the tool discovered 27 virtual machines. This is just the initial collection step, and once collected, you can assess and create a report of the VM's using the *mfit* tool, which we will cover after the manual collection process is explained.

MANUAL GUEST LEVEL DATA COLLECTION

LINUX

At the time of writing, you can download the Linux collection script **on the VM that you want to evaluate for migration** using the following commands:

```
wget https://anthos-migrate-release.storage.googleapis.com/v1.9.0/linux/amd64/mfit-linux-
      collect.sh
chmod +x mfit-linux-collect.sh
```

Run the collection script:

```
sudo ./mfit-linux-collect.sh
```

The script will generate a tar file named `m4a-collect-<MACHINE NAME>-<TIMESTAMP>.tar` in the current directory

WINDOWS

At the time of writing, you can download the Windows collection script **on the VM that you want to evaluate for migration** from the following URL:

```
https://anthos-migrate-release.storage.googleapis.com/v1.9.0/windows/amd64/mfit-windows-
      collect.ps1
```

Run the collection script:

```
powershell -ExecutionPolicy ByPass -File .\mfit-windows-collect.ps1
```

The script will generate a tar or zip file (depending on OS version) named `m4a-collect-<MACHINE NAME>-<TIMESTAMP>.tar/zip` in the current directory.

IMPORT COLLECTED DATA FILE

After running the collection script on the assessed VM, download it to the workstation where *mfit* was installed by any means.

Then, import it into *mfit*'s local database using the command:

```
./mfit discover import m4a-collect-<MACHINE NAME>-<TIMESTAMP>.tar/zip
```

AUTOMATIC GUEST LEVEL DATA COLLECTION

mfit contains the guest collection scripted embedded in it and can automatically run them and retrieve the results in the following scenarios.

VMWARE TOOLS

If the assessed VM is running on vSphere and has VMware tools installed, *mfit* can use vSphere APIs to automate the execution of the collection script (the one suited for the VM's OS type) and the retrieval of the results.

To run guest level collection via VMware tools, run the following command:

```
./mfit discover vsphere guest -u <vcenter username> --url <https://vcenter-host-name-or-ip>
--vm-user <vm username> <vm MoRef id or name>
```

You will be prompted to enter both the vCenter and VM/OS passwords.

SSH

If the Linux machine running *mfit* has SSH access to the assessed VM, *mfit* can leverage that to automate the execution of the collection script and the retrieval of the results.

- To run guest level collection via SSH using the ssh key of the current local user (located in `~/.ssh`), run the following command:

```
./mfit discover ssh <vm-ip-or-hostname>
```

- To run guest level collection via SSH with additional authentication options, run the following command:

```
./mfit discover ssh -i </path/to/ssh_private_key> -u <remote-username> <vm-ip-or-hostname>
```

For additional options for running guest level collection via SSH consult the official documentation or run `./mfit discover ssh -help`

ASSESSMENT

To examine the discovered VMs and collected data, run the following command:

```
./mfit discover ls
```

To perform assessment on this data, run the following command:

```
./mfit assess
```

This will create an assessment result and store it in *mfit*'s local database for use when generating reports.

REPORT GENERATION

Once assessment has been performed, we are ready to generate a report.

To generate a standalone HTML report, run the following command:

```
./mfit report --format html > REPORT_NAME.html
```

To generate a JSON report, run the following command:

```
./mfit report --format json > REPORT_NAME.json
```

The report can then be displayed using on Google's cloud console:
<https://console.cloud.google.com/anthos/migrate/report>

In this section, we discussed the fit assessment tool used to determine workloads' suitability for migration to a container. Once the assessment is made and workloads suitable for migration were chosen, we can start the migration process itself. That's the topic of the next section.

18.4.2 Basic migration example

In this basic example, we will use the Command Line Interface (CLI) to set up a Compute Engine virtual machine on GCE and then use M4A to migrate the VM to a GKE cluster. Note that we need to create another Kubernetes "processing cluster" used with the intent of driving the migration process itself. The processing cluster will do the work of pulling the application from the VM and generating all the artifacts as containers. Let's start.

First we create a source VM. In this basic example the VM will host an Apache web server. The VM can be created with this command:

```
gcloud compute instances create http-server-demo --machine-type=n1-standard-1 --  
subnet=default --scopes="cloud-platform" --tags=http-server,https-server --  
image=ubuntu-minimal-1604-xenial-v20200702 --image-project=ubuntu-os-cloud --boot-disk-size=10GB --boot-disk-type=pd-standard
```

Then, let's make sure that the VM is accessible from internet with this command:

```
gcloud compute firewall-rules create default-allow-http --direction=INGRESS --priority=1000  
--network=default --action=ALLOW --rules=tcp:80 --source-ranges=0.0.0.0/0 --target-tags=http-server
```

Now, we can now log in to the VM just configured using the GUI (see Figure 18.4) and install apache with the command:

```
sudo apt-get update && sudo apt-get install apache2 -y
```



Figure 18.4: Connecting the migration cluster

Now we have a source virtual machine with an apache web server running. The next step is to create the Kubernetes processing cluster, which we can do with the following command:

```
gcloud container clusters create migration-processing --machine-type n1-standard-4 --image-type ubuntu --num-nodes 1 --enable-stackdriver-kubernetes
```

Next, let's make sure that we give the processing cluster the correct processing rights. We need to create a specific service account for M4A, add a policy binding with "storage.admin" rights, create a json key to access the processing cluster, and get the credentials for the processing cluster. We can achieve do this with the following four commands - please note that named-tome-295414 is the name of my project and you should change it with yours:

```
gcloud iam service-accounts create m4a-install
gcloud projects add-iam-policy-binding named-tome-295414 --member="serviceAccount:m4a-install@named-tome-295414.iam.gserviceaccount.com" --role="roles/storage.admin"
gcloud iam service-accounts keys create m4a-install.json --iam-account=m4a-install@named-tome-295414.iam.gserviceaccount.com --project=named-tome-295414
gcloud container clusters get-credentials migration-processing
```

Once we have the credentials, we can log in to the processing cluster and install M4A. Let's do it with the following command - please note that m4a-install.json is the json key we have just created:

```
migctl setup install --json-key=m4a-install.json
```

We can use migctl to check whether the deployment is successful with this command:

```
migctl doctor
[✓] Deployment
```

After that, we can set up the source for migration, together with a specific service account "m4a-ce-src", "compute.viewer" and "compute.storageAdmin" policy bindings required during the migration process, and the creation of a json key "m4a-ce-src.json"

This can be done with the following commands:

```
gcloud iam service-accounts create m4a-ce-src
gcloud projects add-iam-policy-binding named-tome-295414 --member="serviceAccount:m4a-ce-src@named-tome-295414.iam.gserviceaccount.com" --role="roles/compute.viewer"
gcloud projects add-iam-policy-binding named-tome-295414 --member="serviceAccount:m4a-ce-src@named-tome-295414.iam.gserviceaccount.com" --role="roles/compute.storageAdmin"
gcloud iam service-accounts keys create m4a-ce-src.json --iam-account=m4a-ce-src@named-tome-295414.iam.gserviceaccount.com --project=named-tome-295414
```

Once we have created the credentials for the source, we can proceed to set up the source with the following command - please note that 'ce' stands for Google Compute Engine (GCE) - :

```
migctl source create ce http-source --project named-tome-295414 --json-key=m4a-ce-src.json
```

After creating a migration source, we can now create a migration plan to containerize our VM with the command:

```
migctl migration create my-migration --source http-source --vm-id http-server-demo --intent Image
```

If you want to have a look to the migration plan (for instance to modify it), you can use the following command:

```
migctl migration get my-migration
```

It is possible to start the actual migration with the command:

```
migctl migration generate-artifacts my-migration
```

As result, you should see something like this:

```
running validation checks on the Migration...
migration.anthos-migrate.cloud.google.com/my-migration created
```

Once the migration starts, it is possible to check the progress with the following command - note that the flag `-v` gives a verbose dump of the status, which is useful if something goes wrong:

```
migctl migration status my-migration
```

When the migration is concluded, you will see something similar to the output below:

NAME	CURRENT-OPERATION	PROGRESS	STEP	STATUS	AGE
my-migration	GenerateMigrationPlan	[2/2]	CreatePvcs	Completed	11m23s

The next step is to get the generated artifacts with the following command:

```
migctl migration get-artifacts my-migration
```

Once the generation has completed, you should see something like this:

```
Downloaded artifacts for Migration my-migration. The artifacts are located in
/home/a_gulli.
```

Since we want to access the web server inside the migrated containers, let's edit the `deployment_spec.yaml` and add a specific section for the task:

```

apiVersion: v1
kind: Service
metadata:
  name: hello-service
spec:
  selector:
    app: http-server-demo
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: LoadBalancer

```

We can now deploy the artifacts on our Kubernetes cluster:

```

kubectl apply -f deployment_spec.yaml

deployment.apps/app-source-vm created
service/app-source-vm created
service/my-service created

```

It might be useful to check that everything went well with the following command:

```
kubectl get service
```

As a result, you should see something like this:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
app-source-vm	ClusterIP	None	<none>	<none>	44s
kubernetes	ClusterIP	10.63.240.1	<none>	443/TCP	41m
my-service	LoadBalancer	10.63.243.209	35.232.24.49	80:32417/TCP	43s

In this case, the external IP address is 35.232.24.49. Now I can open a browser and check that everything is ok (see Figure 18.5).



Hello World

This page was created from a simple start up script!

Figure 18.5: Accessing the web server migrated from VM to containers

Congratulations! You have successfully migrated a virtual machine running on GCE into a container running on GKE using the CLI. If you want to see another example of basic

migration, I would suggest you consider the hands-on lab "Migrate for Anthos: Qwik Start" available on quiklabs²⁷.

Now that you know how to execute a migration using the CLI, we will move on to the next section where we will use the Cloud console UI to perform a migration.

18.4.3 Google Cloud Console UI migration example

In this section, we use M4A to migrate an application running as a virtual machine on Google Compute Engine to Google Kubernetes Engine. The GKE processing clusters can be located in the cloud or on-prem. The migration will be run through the Graphical User Interface (GUI) available via the Google Cloud Console. Note that the migration process is consistent between the CLI and the GUI.

A GKE cluster will be used as a "processing cluster" to control the migration. The artifacts generated during the process will be stored on Google Cloud Storage (GCS) and the final container images are pushed on Google Container Registry. Under the hood this is identical as for the CLI-driven migration.

The first step is to access Anthos Migrate from console (see Fig.6) at

<https://console.cloud.google.com/anthos/migrate>.

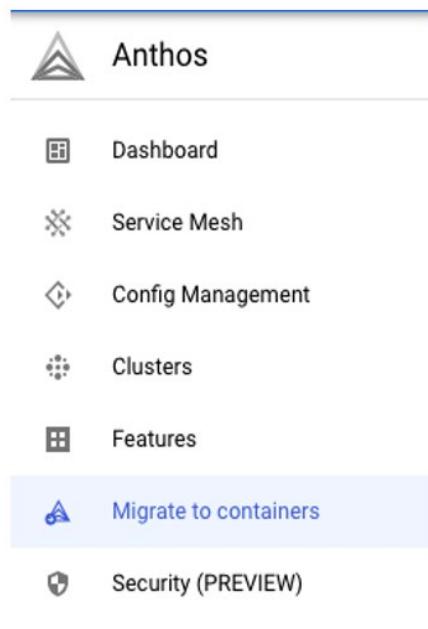


Figure 18.6: Accessing Anthos Migrate to containers

²⁷ See <https://google.qwiklabs.com/catalog?keywords=Migrate%20for%20Anthos%3A%20Qwik%20Start>

For the sake of simplicity, we will deploy a VM from Marketplace with Tomcat Server preinstalled. We will use this workload to migrate Tomcat from the VM to a container.

Let's start by accessing the Google Click-to-Deploy repository with Tomcat(see Figure 18.7). The URL is:

<https://console.cloud.google.com/marketplace/details/google-click-to-deploy-images/tomcat>.

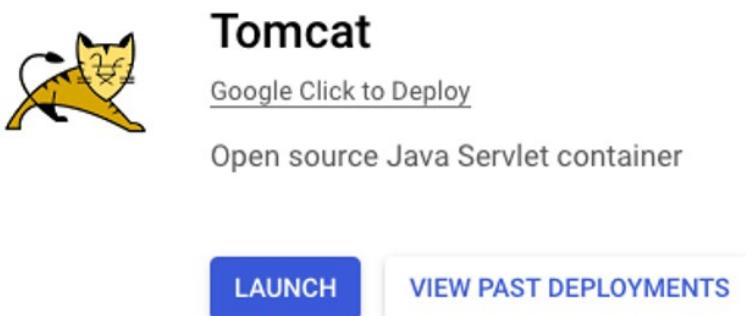


Figure 18.7: Deploying a Tomcat application to VMs

Then, let's select the zone where we want to deploy (see Figure 18.8):

A screenshot of a deployment configuration interface. It includes fields for "Deployment name" (containing "tomcat-vm-gulli"), "Zone" (set to "us-central1-c"), and "Machine type" (set to "1 vCPU" with "3.75 GB memory" and a "Customize" link).

Deployment name	tomcat-vm-gulli
Zone	us-central1-c
Machine type	1 vCPU 3.75 GB memory Customize

Figure 18.8: Deployed Tomcat application

Once the VM with Tomcat Server is deployed it is possible to access the web site from an external IP (see Fig 9).

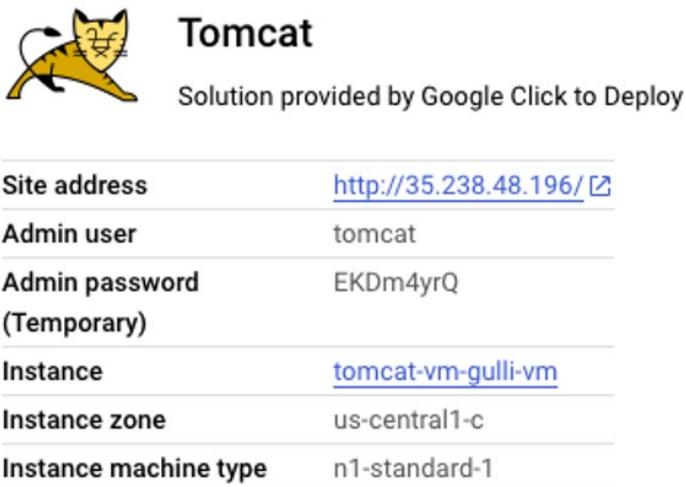


Figure 18.9: Tomcat solution deployed with Google Click-to-Deploy

In this deployment the IP address is <http://35.238.48.196/>, so if you access the web site, you should see the output shown in Figure 18.10:

It works !

Figure 18.10: Accessing Tomcat from Web

The next step is to start the proper migration process. The first thing to do is to create a "processing cluster", a cluster which will be used to control the migration of our source VM. You can perform this task by accessing the "Migrate to containers" Menu on Anthos and use the "Add Processing Cluster" option (see Figure 18.11):

ADD PROCESSING CLUSTER

Figure 18.11: Starting the migration process

It is convenient to follow the suggestion provided in the GUI and create a new cluster dedicated to processing (see Figure 18.12):

Cluster basics

The new cluster will be created with the name, version, and in the location you specify here. After the cluster is created, name and location can't be changed.

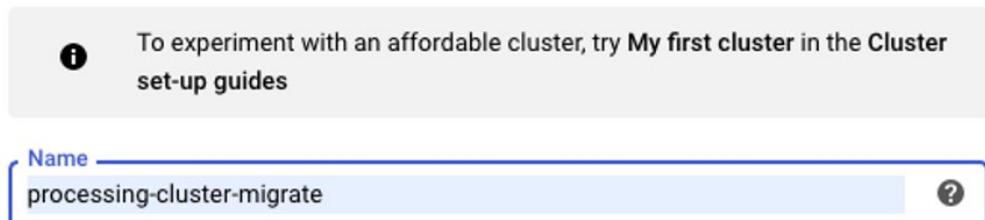


Figure 18.12: Choosing a name for the “processing cluster”

Once the cluster is ready, you should be able to see it via the Google Cloud Console in the GKE section (see Figure 18.13).

<input type="checkbox"/>	Name ^	Location	Cluster size	Total cores	Total memory	Notifications
<input type="checkbox"/>	processing-cluster-migrate	us-central1-c	3	6 vCPUs	12.00 GB	

Figure 18.13: The processing cluster is ready for use

Then, you can select the cluster (see Figure 18.14):

Start by selecting a cluster:



Figure 18.14: Selecting the processing cluster

At this point we need to make sure that the processing cluster has the proper processing rights. To this extent, the GUI suggests to run a number of commands in the cloud shell. You just need to push the suggested button (see Fig 15) "Run in Cloud Shell":

```
$ gcloud services enable servicemanagement.
```

RUN IN CLOUD SHELL

Figure 18.15: Using the UI to run the setup required by M4A

Let's see the required steps in detail.

First, we need to enable the Google Cloud APIs. Then, we need to create a service account for storing the migration artifacts in the Container Registry and Cloud Storage. Then, we need to add the permissions to access Container Registry and Cloud Storage. Then, we need to create and export a new key to a file required by M4A to use the service account.

Once these steps are done, we can install the migrate to containers software. Again, the GUI makes this step very intuitive. The last step is to check with "migctl doctor" that the deployment status is correct (see Figure 18.16):



Figure 18.16: Correct deployment M4A

Once the processing cluster is configured, you can select a migration source from where the VM will be pulled (see Figure 18.17):



Figure 18.17: Adding a source for migration

Currently, you can pull from GCE (see Figure 18.18). In addition, it is possible to use Migrate for Compute Engine²⁸ to import local vSphere environments, AWS, and Azure to GCE.

²⁸ See <https://cloud.google.com/migrate/anthos/docs/adding-a-migration-source>

2 Define source name and type

Define a name for the new source.

Name *
migrate-vm-tomcat-gulli

Select the source environment from which you want to migrate workloads.

Compute Engine ▾

Learn how to define other source types (**VMWare**, **AWS** and **Azure**) supported currently through the tool's CLI. [Learn more](#)

Figure 18.18: Adding a migration source

Once you have chosen a name, you can select the project in which the source VMs are placed (see Figure 18.19)

3 Configuration

Select the project in which the source VMs are placed at.

Project *
named-tome-295414 SELECT

Define service account for the defined source.

Select a service account

- Create new service account (recommended)
- Use existing service account

Service account name *
m4a-migrate-vm-tomcat

Figure 18.19: Selecting the project in which the source VMs are placed

Now that the processing cluster and the migration source have been created we can start the migration (see Figure 18.20):



Figure 18.20: Starting a migration process

The migration requires a name, a source, a VM OS type, the VM ID, and the Migration Intent. Let's specify them via the GUI (see Figure 18.21):

Setup migration

Migration name *	tomcat-migr
Select Source *	migrate-vm-tomcat-gulli processing-cluster-migrate
To learn how to add more sources click here .	
VM OS type *	Linux
Select the OS type (Windows or Linux) of the VM candidate for migration	
VM ID *	tomcat-vm-gulli-vm
Specify the ID of the migrated VM.	
Migration Intent *	Image

Figure 18.21: Specifying the Migration name, the Migration source, the VM OS type, the VM ID, and the Migration Intent.

Then, M4A will start generating the migration plan (see Figure 18.22):

Migration name ↑	VM ID	Status
tomcat-migr	tomcat-v...	Generating migration plan

Figure 18.22: Generating the migration plan

During the migration, we can check the progress made with the following command, which will dump an extensive debug:

```
migctl migration status my-migration -v
```

Once the migration plan is generated (see Figure 18.23), it is possible to inspect the results with the GUI:

Migration name ↑	VM ID	Status
tomcat-migr	tomcat-v...	✓ Migration plan generated

Figure 18.23: An overview of the migration plan generated by M4A

In particular, the option menu allows us to edit the generated migration plan (see Figure 18.24):

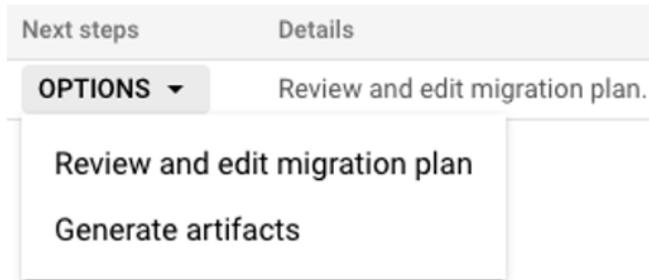


Figure 18.24: Reviewing and editing the generated migration plan

Let's have a look at what has been generated by editing the migration plan (see Figure 18.25):

Edit migration plan

```

1  apiVersion: anthos-migrate.cloud.google.com/v1beta2
2  kind: GenerateArtifactsFlow
3  metadata:
4    creationTimestamp: null
5    labels:
6      migration: kwuflyjr
7      migration-name: tomcat-migr
8      migration-namespace: v2k-system
9    name: generate-artifacts-flow-f3d851ee-2b23-48ad-88a8-6f21356d8918
10   namespace: v2k-system
11   ownerReferences:
12     - apiVersion: anthos-migrate.cloud.google.com/v1beta2
13       blockOwnerDeletion: true
14       controller: true
15       kind: Migration
16       name: tomcat-migr
17       uid: f3d851ee-2b23-48ad-88a8-6f21356d8918
18   spec:
19     # Review and set which artifacts to generate.
20

```

Figure 18.25: Editing the generated migration plan

Normally, you don't need to change the migration plan. However, this is useful if you either need to strip out unneeded VM components or you need to add some additional configuration. After checking, and if necessary, editing the migration plan, we can start generating the artifacts (see Figure 18.26):



SAVE AND GENERATE ARTIFACTS

SAVE

Figure 18.26: Generating artifacts with the edited migration plan

Once the migration plans are generated, they can be inspected by accessing the "artifacts" tab from the Google Cloud Console. This includes the Dockerfile, the Container image for deployment (that can be directly deployed), the Container image base layer (the non-

runnable image layers), the Deployment spec YAML, the Migration plan YAML and the Artifact links YAML (see Figure 18.27):

STATUS	ARTIFACTS
	 Dockerfile 
	 Container image for deployment 
	 Container image base layer 
	 Deployment spec YAML 
	 Migration plan YAML 
	 Artifact links YAML  

Figure 18.27: Generated Migration artifacts

The generated artifacts are stored in Google Cloud Storage (GCS) with separated buckets for base and image layers (see Figure 18.28):

<input type="checkbox"/>	Name 
<input type="checkbox"/>	artifacts.named-tome-295414.appspot.co...
<input type="checkbox"/>	named-tome-295414-migration-artifacts

Figure 18.28: Artifacts stored in GCS

Let's have a look at the system image with the Dockerfile, the deployment spec, the manifest and the migration YAML (see Figure 18.29):

	Name	Size	Type
<input type="checkbox"/>	Dockerfile	700 B	text/plain; charset=utf-8
<input type="checkbox"/>	deployment_spec.yaml	1.7 KB	text/plain; charset=utf-8
<input type="checkbox"/>	manifest.yaml	735 B	text/plain; charset=utf-8
<input type="checkbox"/>	migration.yaml	3.1 KB	text/plain; charset=utf-8

Figure 18.29: Dockerfile, deployment_spec, manifest, migration files

In addition, images generated for migration are automatically pushed to the Google Container Registry (GCR) and you can browse them via the console (see Figure 18.30):

Name ^	Hostname
 tomcat-vm-gulli-vm	gcr.io
 tomcat-vm-gulli-vm-non-runnable-base	gcr.io

Figure 18.30. Images generated for migration and pushed to GCR

All the generated artifacts can be downloaded via the CLI with:

```
migctl migration get-artifacts my-migration
```

Which includes:

- **deployment_spec.yaml**: Configures your workload;
- **Dockerfile**: Used to build the image for your migrated VM;
- **migration.yaml**: A copy of the migration plan.

AN OVERVIEW ON DOCKERFILE

In this section we have a deep look at the Dockerfile generated by M4A (see Listing 18.1). The file can be edited to customize your image, for instance either for installing new packages or for installing an upgraded version of M4A runtime. The file contains the original container repository for runtime, the image containing data captured from the source VM, and the initial entry point. A typical M4A Dockerfile is as follows:

Listing 18.1 Dockerfile generated by M4A

```
# Please refer to the documentation:
# https://cloud.google.com/migrate/anthos/docs/dockerfile-reference

FROM anthos-migrate.gcr.io/v2k-run-embedded:v1.5.0 as migrate-for-anthos-runtime

# Image containing data captured from the source VM
FROM gcr.io/named-tome-295414/tomcat-vm-gulli-vm-non-runnable-base:11-13-2020--15-0-39 as
source-content

# If you want to update parts of the image, add your commands here.
# For example:
# RUN apt-get update
# RUN apt-get install -y \
#     package1=version \
#     package2=version \
#     package3=version
# RUN yum update
# RUN wget http://github.com

COPY --from=migrate-for-anthos-runtime / /

# Migrate for Anthos image includes entrypoint
ENTRYPOINT [ "./v2k.go" ]
```

We will see more details on M4A generated Dockerfile later in this chapter when we discuss the post-migration integration with CI/CD pipelines. In the next section we will discuss the details of the deployment_spec.yaml file.

AN OVERVIEW OF THE DEPLOYMENT_SPEC.YAML

In this section we will discuss the deployment_spec.yaml generated by M4A. First, let's define some terminology we will use later:

- **Stateless:** An application is stateless when the server does not store any state about the client session. In other words, there is no stored knowledge of or reference to past transactions
- **Stateful:** An application is stateful when the server stores data about the client session. In other words, the current transaction may be affected by what happened during previous transactions. For this reason, a stateful application needs to use the same servers each time a request from a user is processed.

With this context in mind, let's consider deployment_spec.yaml. This file will be different according to the intent flag selected in the UI:

1. **Intent: Image.** The YAML defines a stateless application with identical Pods²⁹ managed as a service. Different parts of the YAML are:
 - a) **Deployment.** This is the set of identical Pods deployed from the image generated from your migrated VM. They are stored in GCR.

²⁹ A Pod encapsulates one or more applications and it is the smallest unit of execution in Kubernetes.

- b) **Service.** This groups Pods in your deployment into a single resource accessible from a stable IP address. By default, there is a single cluster internal IP reachable only from within the cluster with no load balancing. The Kubernetes endpoints controller will modify the DNS configuration to return records (addresses) that point to the Pods, which are labeled with "app": "app-name" where the name of the app is inferred from the "migctl migration create my-migration" command. Note that Pods will be visible only within the cluster by default. So, It might be appropriate to expose Pods outside of your cluster. We will see an example later in the Chapter.
 - c) **Logging configuration** -- Configures logging to Cloud Logging by listing many of the most common log files.
- 2. Intent: ImageAndData.** The YAML defines a stateful application with different Pods associated with persistent volumes. Different parts of the YAML are:
- a) **StatefulSet.** This is the set of Pods deployed from the image generated from your migrated VM. They are stored in GCR.
 - b) **Service.** This is similar to the one defined in the "Image" section
 - c) **PersistentVolume.** This is used to manage the durable storage.
 - d) **PersistentVolumeClaim.** This represents a request for and claim to the PersistentVolume resource (such as specific size, and access mode).
 - e) **Logging configuration.** This is similar to the one defined for stateless.
- 3. Intent: Data.** Different parts of the YAML are:
- a) **PersistentVolume.** This is similar to the one defined for stateful
 - b) **PersistentVolumeClaim.** This is similar to the one defined for stateful

A typical M4A deployment_spec.yaml is as it follows (see Listing 18.2):

Listing 18.2 Deployment_spec.yaml generated by M4A

```

# Stateless application specification
# The Deployment creates a single replicated Pod, indicated by the 'replicas' field
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: tomcat-vm-gulli-vm
    migrate-for-anthos-optimization: "true"
    migrate-for-anthos-version: v1.5.0
  name: tomcat-vm-gulli-vm
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tomcat-vm-gulli-vm
      migrate-for-anthos-optimization: "true"
      migrate-for-anthos-version: v1.5.0
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: tomcat-vm-gulli-vm
        migrate-for-anthos-optimization: "true"
        migrate-for-anthos-version: v1.5.0
    spec:
      containers:
        - image: gcr.io/named-tome-295414/tomcat-vm-gulli-vm:11-13-2020--15-0-39
          name: tomcat-vm-gulli-vm
          readinessProbe:
            exec:
              command:
                - /code/ready.sh
          resources: {}
          securityContext:
            privileged: true
          volumeMounts:
            - mountPath: /sys/fs/cgroup
              name: cgroups
          volumes:
            - hostPath:
              path: /sys/fs/cgroup
              type: Directory
              name: cgroups
      status: {}
  ---
  # Headless Service specification -
  # No load-balancing, and a single cluster internal IP, only reachable from within the
  # cluster
  # The Kubernetes endpoints controller will modify the DNS configuration to return records
  # (addresses) that point to the Pods, which are labeled with "app": "tomcat-vm-gulli-
  # vm"
apiVersion: v1
kind: Service
metadata:

```

```

creationTimestamp: null
  name: tomcat-vm-gulli-vm
spec:
  clusterIP: None
  selector:
    app: tomcat-vm-gulli-vm
  type: ClusterIP
status:
  loadBalancer: {}
---

```

DEPLOYING THE CONTAINER GENERATED BY M4A

In this section the steps needed to deploy the container generated by M4A are discussed. Deploying the `deployment_spec.yaml` is very easy. We can use the following command:

```
kubectl apply -f deployment_spec.yaml
```

As a result you should see something like this:

```
deployment.apps/tomcat-vm-gulli-vm created
service/tomcat-vm-gulli-vm created
```

If you want, you can check the status of the deployed pods with the following command:

```
kubectl get pods
```

As a result you should see something like this:

NAME	READY	STATUS	RESTARTS	AGE
tomcat-vm-gulli-vm-66b44696f-ttgq6	1/1	Running	0	21s

By default the container is deployed with no load-balancing, and a single cluster internal IP which is only reachable from within the cluster (see Figure 18.31). This means that the Kubernetes endpoints controller will modify the DNS configuration to return addresses that point to the Pods, which are labeled with "app": "tomcat-vm-gulli-vm". Of course, you can change the deployment and add a LoadBalancer. Let's do it, by adding the following in the deployment spec:

```

apiVersion: v1
kind: Service
metadata:
  name: hello-service
spec:
  selector:
    app: tomcat-vm-gulli-vm
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: LoadBalancer

```

Then, let's check that the service is indeed accessible:

```
kubectl get service hello-service
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP     PORT(S)        AGE
hello-service  LoadBalancer  10.88.10.207  34.67.239.170  80:32033/TCP  76s
```

Let's try to access from the internet (see Figure 18.31):



It works !

If you're seeing this page via a web browser, it means you've setup Tomcat successfully. Congratulations!

Figure 18.31: Accessing Tomcat container after migration with M4A

Congratulations! You now have a routable container holding the full Tomcat installation previously available in a VM! The migration happened automatically and there was no need to either recompile or access the original source code.

Before concluding, a hint on the GUI. If you need to edit multiple files, it might be very convenient to use the built in Editor which is based on Eclipse. In particular, the editor is a quick and easy way to review and change all the files generated by M4A (see Figure 18.32).

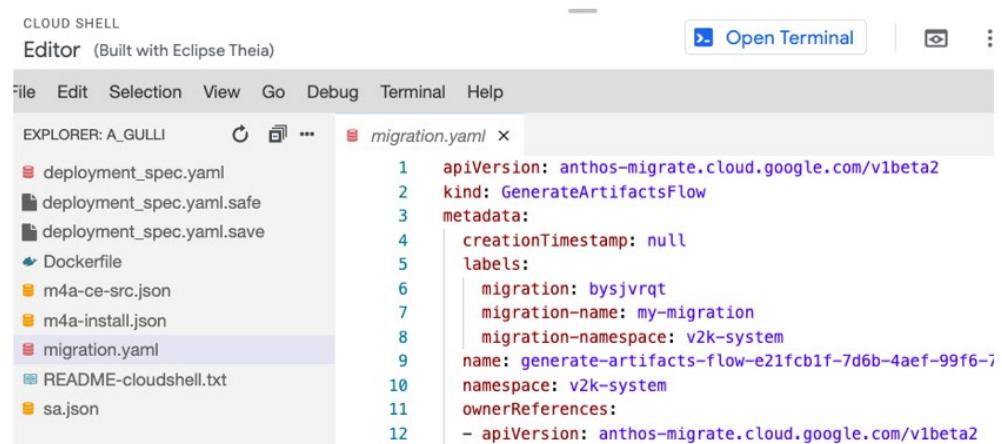


Figure 18.32: Builtin Editor used to manipulate migration configuration.

One note before concluding. In addition to Google Container Registry (GCR) and Google Cloud Storage for data repositories, M4A version 1.6 and above supports additional repositories including ECR, S3, Docker registries supporting basic authentication. Next section we are going to talk about Windows migration.

18.4.4 Windows migration

In this section we discuss how to migrate Windows VMs to GKE. Please note that Windows migration supports Compute Engine as a source. However, as discussed before it is possible to migrate a Windows VM from other sources into Compute Engine using Migrate for Compute Engine³⁰. The resulting VM can be then migrated to GKE. Unsurprisingly, Windows migration is very similar to Linux's. Indeed, behind the scenes, M4A uses a unified M4A CLI utility named `migctl`. Now, let's see a quick example using the CLI interface.

First, similarly to Linux you can use `migctl` with the following statement for adding a migration source:

```
migctl source create ce my-ce-src --project my-project --json-key=m4a-ce-src.json
```

Where `my-ce-src` is the name of the source, `my-project` is the name of the project and `m4a-ce-src.json` is the name of the JSON key file obtained after creating a service account for using Compute Engine as a migration source.

Then you can create a migration with the following command:

```
migctl migration create my-migration --source my-ce-src --vm-id my-id --intent Image --os-type=Windows
```

where `my-migration` is the name of the migration and `vm-id` is the name of the Compute Engine instance as shown in the Google Cloud Console. At the end of 2020, the only intent supported for Windows is "Image". The migration plan just created can be retrieved with the following command:

```
migctl migration get my-migration
```

If needed the migration plan can be customized by editing the file `my-migration.yaml`. When editing is completed, it is then possible to upload the edited migration plan with the following command:

```
migctl migration update my-migration
```

Next step is to execute the migration and generate artifacts with the command:

```
migctl migration generate-artifacts my-migration
```

During the migration, you can monitor the status with the following command:

```
migctl migration list
```

When the migration concludes you can access the artifacts with the command:

³⁰ See <https://cloud.google.com/migrate/compute-engine/docs/4.11/getting-started>

```
migctl migration get-artifacts my-migration
```

As a results, you should see something like:

```
Artifacts are accessible through `gsutil cp gs://PATH/artifacts.zip ./`
```

Hence you can get the artifacts with:

```
gsutil cp gs://PATH/artifacts.zip ./
```

The next step is to use the artifacts to build a Docker image. We can use Windows powershell to expand the `artifacts.zip` with the command:

```
Expand-Archive .\artifacts.zip
```

Then log in to the Container Registry with the command:

```
docker login gcr.io
```

The next step is to build the container:

```
docker build -t gcr.io/myproject/myimagename:v1.0.0 .\artifacts\
docker push gcr.io/myproject/myimagename:v1.0.0
```

Once the build is completed, the container image will be in the Container Registry, and the image can be deployed to a GKE cluster. Note that the Google Cloud Console for Anthos includes Windows workloads support since M4A v1.5. The experience is identical to the one already discussed for Linux. If you want to see another example of windows migration, I would suggest you consider the hands-on lab "Migrate for Anthos: Windows" available on quiklabs³¹.

18.4.5 Migration from other clouds

As of November 2021, migration from other clouds is based on a two-step approach. First, virtual machines are migrated (technically these are converted into GCE instances). Then, the migrated virtual machines are containerized.

M4A uses the product Migrate for Compute Engine (M4CE)³² to stream virtual machines located on other clouds or on prem, moving them onto GCE instances. Migrate for Compute Engine can be installed via the marketplace³³ and it allows the migration of thousands of applications across multiple data centers and clouds from source platforms such as VMWare, Microsoft Azure, and Amazon EC2.

To use M4CE, you need to set up a site-to-site VPN connection and firewall rules to enable communication between the VPC in which the manager is positioned and the VPC of the source VM on the other cloud. The interested reader may find the online firewall documentation useful³⁴. Dynamic routing based on BGP³⁵ protocol can be setup via GCP

³¹ See <https://google.qwiklabs.com/catalog?keywords=Migrate%20for%20Anthos%3A%20Windows>

³² See <https://cloud.google.com/migrate/compute-engine>

³³ See <https://console.cloud.google.com/marketplace/details/google-cloud-platform/migrate-for-compute-engine>

³⁴ See <https://cloud.google.com/vpc/docs/firewalls>

³⁵ See https://en.wikipedia.org/wiki/Border_Gateway_Protocol

Cloud Router³⁶ working on either Cloud VPN³⁷ or dedicated and high speed Cloud Interconnect³⁸. Cloud Interconnect extends your on-premises network to Google's network through a highly available, low latency connection.

In addition, thousands of virtual machines can be migrated in batches by aggregating them in waves according to their logical role. M4CE allows us to define Runbooks, to then decide which VM should be migrated, and in which order. A simulated testing phase can be planned before the effective migration.

The main components of M4CE are:

- A **Migration Manager** used to orchestrate migration. The manager runs on a separate Google Compute Engine VM, and it offers a migration console to manage and monitor all the system components. Note that the manager might require specific permissions to handle specific actions such as turning on and off a VM. These permissions can be defined with specific policies.
- A **Cloud Extensions** used to handle storage migrating from the source platform. An extension is a conduit for VM storage between the migration source and destination. These extensions run on separate Compute Engine VMs, and serve data to migrated workloads during the migration process itself. Note that extensions work with a dual-node active/passive configuration for high availability; each node serves workloads and, at the same time, provides backup for the other node.

Different components are then deployed on source platforms:

- On **vSphere**, a back-end component serving runtime data from VMware to extensions on Google Cloud. Data is then used by the VMs on Compute Engine. In addition, a vCenter plugin connecting vSphere to the Migration Manager. The plugin orchestrates the migration on vCenter.
- On **Amazon EC2** and **Azure**, an importer is deployed at runtime. This importer serves runtime data from the source to extensions on Google Cloud. Data is then used by the VMs on Compute Engine.

Since migrate 1.9 it is also possible to deploy containers to GKE Autopilot clusters and Cloud Run, but this topic is outside the subject of this book.

In this section we have briefly introduced M4CE used by M4A to move VMs between clouds and on-prem. Migration can happen in minutes, while data migrates transparently in the background. The interested reader can see more online³⁹. In addition, a convenient hands-on lab to migrate an EC2 instance from AWS to Compute Engine on GoogleCloud is available on quiklabs⁴⁰. Next section is about a number of Google best practices adopted for M4A.

³⁶ See <https://cloud.google.com/network-connectivity/docs/router>

³⁷ See <https://cloud.google.com/network-connectivity/docs>

³⁸ See <https://cloud.google.com/network-connectivity/docs/interconnect>

³⁹ See <https://cloud.google.com/solutions/migrating-vms-migrate-for-compute-engine-migrating-vms>

⁴⁰ See <https://google.qwiklabs.com/catalog?keywords=Migrate%20for%20Compute%20Engine>

18.5 Advanced topic: M4A best practices

In this section we discuss some best practices for Migrate for Anthos. The idea is to provide guidance on real-time scenarios frequently encountered by customers. This is a rather advanced section which assumes that you are very familiar with Kubernetes environments. Different details of Kubernetes are discussed in detail. Let's start.

VM hostnames. One convenient pattern is to transform VM hostnames into Kubernetes service names⁴¹. Note that service names are a set of Pod endpoints grouped into a single resource. So, retaining this naming convention helps with consistency.

Multiple apps/services per VM. If there are multiple applications or services per VM, it might be convenient to define a Kubernetes service for each of them. Again, this naming convention helps with consistency.

Host file customizations. If your VMs use specific customization on the host file for DNS resolution, then it is recommended to use the Kubernetes pod spec hostAliases⁴². Adding entries to a pod's /etc/hosts file provides a pod-level override of hostname resolution. Moreover, this helps to replicate multiple application environments, such as production, staging, and testing.

Multi-VM stacks. If you have a multi-VM stacks environment then it might be convenient to place co-dependent pods in the same Kubernetes namespace and use short DNS names. In addition, you should use Kubernetes NetworkPolicy to restrict access between front-end and back-end pods. This organization would help to keep your environment organized, safer and more effective.

Referring to external services. If your applications use external services then it is worth considering using the Kubernetes ExternalName service without selector⁴³a best practice in Kubernetes to abstract external back-ends.

NFS file shared. Currently, M4A does not automatically migrate NFS mounts. Therefore, you need to manually define NFS persistent volume directives and add them to the generated pod YAML. The interested reader can find more information on Mounting External Volumes online⁴⁴.

Unneeded services. Migration is a consolidation moment. Therefore, it is appropriate to double check all the services running on your virtual machine and disable those which are not needed on containers. Migrate for Anthos will automatically disable unnecessary hardware or environment-specific services, and a predefined set of additional services running on VMs. See online for a detailed list of different services automatically disabled⁴⁵.

⁴¹ See <https://cloud.google.com/kubernetes-engine/docs/concepts/service>

⁴² See <https://kubernetes.io/docs/concepts/services-networking/add-entries-to-pod-etc-hosts-with-host-aliases/>

⁴³ See <https://kubernetes.io/docs/concepts/services-networking/service/#services-without-selectors>

⁴⁴ See <https://cloud.google.com/migrate/anthos/docs/mounting-external-volumes>

⁴⁵ See https://cloud.google.com/migrate/anthos/docs/planning-best-practices#disable_unneeded_services

Environmental variables. If your application requires an environment variable, then it is a good practice to move definitions to the Kubernetes pod YAML. In this way, you will follow the best practice of having all your Infrastructure as code.

Scripts using Cloud Instance Metadata. If your scripts look up metadata, then it is worth replacing this look-up with either Kubernetes ConfigMap⁴⁶ or, again, using env variables defined in your Kubernetes Pod YAML definition.

Application logs. If you want, you can have logs that are generated by workload containers migrated with Migrate for Anthos written to Cloud Logging. By default, M4A will consider entries written to stdout of init, the parent of all Linux processes, and contents from /var/log/syslog. Adopting this strategy will enhance the level of automation in your environment and the observability of your applications.

GKE Ingress controller. If you migrate to GKE then it might be convenient to use GKE network ingress control for controlling the network traffic accessing workloads. This will eliminate the need of changing your application with additional routing rules, VPNs, filters or VLANs. For instance, if you migrate a three-tier application, you might want to split it into multiple containers. The front-end service is accessed via a GKE Google LoadBalancer⁴⁷ for load scalability. In addition, you might want to define network policies for enforcing access to the application service only by the front-end pods and not from the external world. Similarly, you might want to define policies to access the database layer from the application layer only. These choices would increase the security of your environment.

Linux specific runlevel 3. In a Linux environment, certain services are configured to start by default only at runlevel 5. Currently, M4A reaches runlevel 3 only. VMs migrated into GKE with Migrate for Anthos will be booted in the container at Linux runlevel 3. As a consequence, certain services should be configured to start automatically at runlevel 3. These might include X11, XDM and the GUI used for VNC.

In this advanced section, we have discussed a number of best practices which can be adopted for fine-tuning environments migrated with M4A. In the next section, we will discuss how to upgrade images post-migration.

18.6 Post-migration integration with CI/CD pipelines

Artifacts generated with M4A can be used for Day 2 operations such as software updates, configuration changes, security patches, and additional operations with files. These artifacts can be easily integrated with a CI/CD typical pipeline consisting of Source, Build, Test, and Deploy (see Figure 18.33)

⁴⁶ See <https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/>

⁴⁷ <https://cloud.google.com/kubernetes-engine/docs/concepts/ingress>

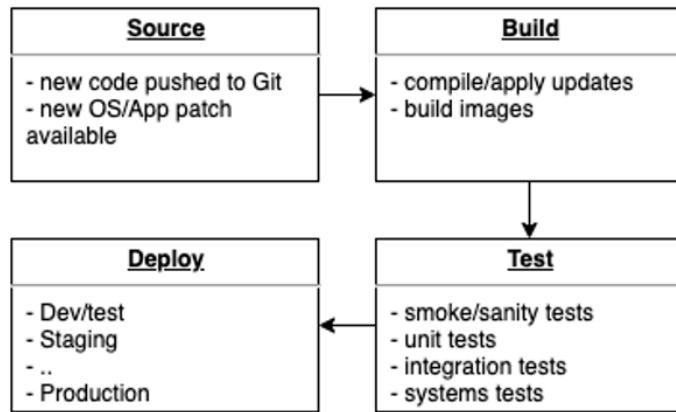


Figure 18.33: Typical CI/CD development phases

Artifacts are generated with multi-stage builds so that they can be incrementally maintained without incurring the risk of inflating the generated container image. See Figure 18.34 for an example of integration of M4A with CI/CD pipelines.

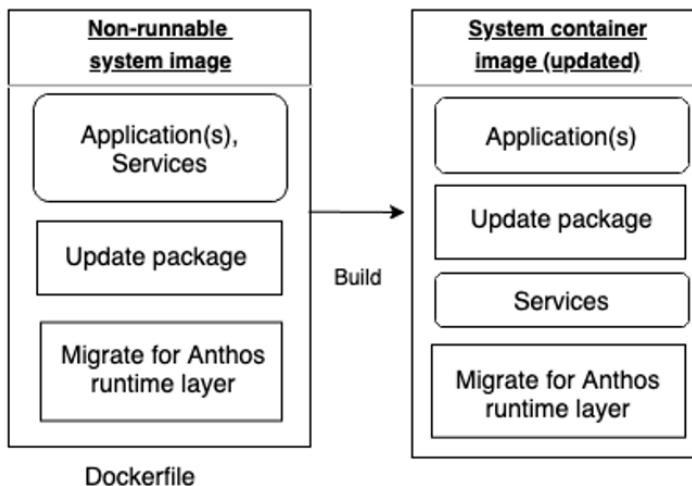


Figure 18.34: Integration of M4A with CI//CD pipelines

A typical Docker artifact is composed of two parts (see Figure 18.35). The first part is the M4A runtime, while the second part is the non-runnable base representing the capture system image layer from the migrated VM.

If you need to update the M4A runtime, then you can simply replace the first FROM directive as appropriate from the Docker file. For instance, suppose that you need to support M4A 1.8.1, then this can be easily achieved with the following new directive, which replaces the current one:

```
FROM anthos-migrate.gcr.io/v2k-run-embedded:v1.8.1 as migrate-for-anthos-runtime
```

If you need to update your application then you can change the second Docker FROM directive. In detail, you typically download the generated Docker file from your container registry, edit the Dockerfile to apply your desired changes, build a new layered image, and update the existing deployment with a rolling update. As discussed, this image layered approach is very suitable for a CI/CD-based (see Chapter [Anthos, integrations with CI/CD]) deployment environment where DevOps and Site Reliability Engineering (SRE) methodologies are the key.

```
# Please refer to the documentation:
# https://cloud.google.com/migrate/anthos/docs/dockerfile-reference

FROM anthos-migrate.gcr.io/v2k-run-embedded:v1.4.0 as migrate-for-anthos-runtime

# Image containing data captured from the source VM
FROM gcr.io/myproject/myworkload-non-runnable-base:v1.0.0 as source-content

# If you want to update parts of the image, add your commands here.
# For example:
# RUN apt-get update
# RUN apt-get install -y \
#         package1=version \
#         package2=version \
#         package3=version
# RUN yum update
# RUN wget http://github.com

COPY --from=migrate-for-anthos-runtime / /

# Migrate for Anthos image includes entrypoint
ENTRYPOINT [ "/.v2k.go" ]
```

Figure 18.35: A typical Dockerfile generated by M4A, useful for CI/CD pipelines.

In this section, we have discussed how to integrate with CI/CD pipelines for increasing your organizational agility. During the next section, we will discuss how to integrate with service meshes.

18.7 Post-migration integration with ASM

Earlier in this Chapter, we have already discussed the benefits of using a service mesh. These include transparent gains in terms of communication, policy management, observability, and agility. The key observation is the adoption of Anthos Service Mesh (Anthos Service Mesh: security and observability at scale) is another step towards the adoption of SRE and DevOps methodologies.

For instance, ASM makes it possible to check the service status together with key metrics for our applications such as error rate, latency and request rates, visualize the topology, check the estimated cost, and define service-level indicators (see Figure 18.36).

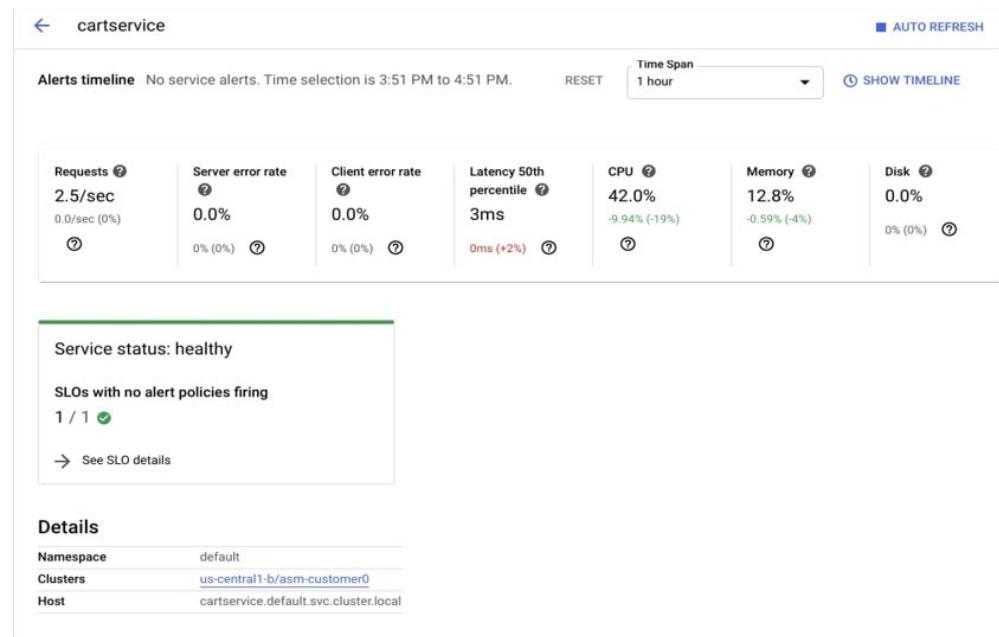


Figure 18.36: Using ASM for inspecting an application after migration

Once again, it is important to point out that these gains come for free with no need to change any code in your migrated application. Once it is containerized, your application becomes a first-class cloud-native application which can be managed with modern cloud-native methodologies and significant cost savings.

In this section we have briefly reviewed the advantages deriving from the adoption of a service mesh.

18.8 Summary

- Moving to Cloud based applications has the benefits of having a modern container based environment using infrastructure in a more efficient way than traditional virtual machines.
- We have learned how Migrate for Anthos can be used to have fully automated transformation, with no need of the original source code.
- We have learned what are the best workloads candidates for migration. These include Stateless web front-end, Multi-VM, multi-tier stacks, business logic middleware, Medium- to large-sized databases, and low duty-cycle and bursty workloads. We have reviewed the components that make up the Migrate for Anthos architecture together with Real world migration scenarios.
- We have learned some common best practices for migration using Migrate for Anthos including Post-Migration integration with CI/CD pipelines and with Anthos Service Mesh