O'REILLY®

Compliments of upbound

What Is Crossplane?

A Low-Code Framework for Building Cloud Native Control Planes

Nic Cope

REPORT

upbound | Powered by Crossplane

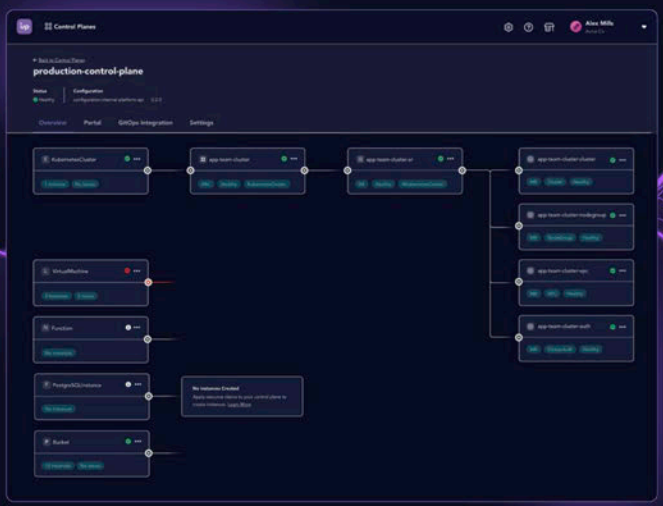
The efficiency of Crossplane. The scalability of Upbound.

With managed control planes, platform teams can scale to tens of thousands of data plane resources with confidence. **Try it for free.**

Managed Control Planes

Centralized Control

Happy Product Engineers
Productive Platform Engineers



upbound.io/freetrial

What Is Crossplane?

A Low-Code Framework for Building Cloud Native Control Planes

Nic Cope

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

What Is Crossplane?
by Nic Cope

Printed in the United States of America.

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

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

Acquisitions Editor: John Devins
Development Editor: Sarah Grey
Production Editor: Katherine Tozer
Copyeditor: Paula L. Fleming

Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Kate Dullea

June 2023: First Edition

Revision History for the First Edition
2023-05-23: First Release

978-1-098-14627-6

[LSI]

Table of Contents

CHAPTER 1
Introduction

Crossplane is a framework for building cloud native control planes. When the founders of Crossplane at Upbound told me what they were working on in late 2018, shortly before the v0.1 release, I was very excited. I'd spent a lot of my career building cloud control planes from scratch. I loved the idea of an open source framework that baked in many best practices of control plane design and could be extended to control almost anything. I immediately joined the project and have been a maintainer since Crossplane v0.1.

A cloud platform is a distributed system, and distributed systems are often architecturally separated into two parts: control planes and data planes. The *data plane* is the system under control. It's what provides direct business value. As for the *control plane*, it works like an air traffic control tower. This analogy is illustrated in Figure 1-1. The aircraft around the airport represent the data plane. The aircraft provide the business value—they move things—but the air traffic control tower is crucial to ensure they can do so safely and efficiently.[1]

1 The term *data plane* is an artifact of the concept's origins in shared-memory multiprocessing research. In this context, the *control plane* and the *data plane* were distinct physical, geometric planes of switches (see Lawrie 1973, 76–87).

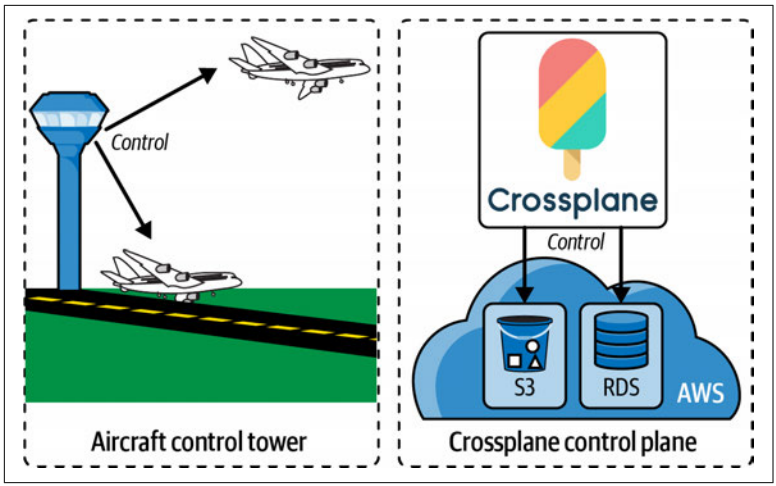


Figure 1-1. A control plane is similar to an aircraft control tower.

Amazon Web Services (AWS) is another example of control planes and data planes. AWS's data planes are the virtual machines, databases, caches, and other services folks use to build products. Each of these services has a control plane that is responsible for starting, stopping, and configuring them. When you use the AWS console or call an AWS API, you're interacting with one of AWS's control planes.

You can also think of a cloud control plane as having a backend that orchestrates the data plane and a frontend that exposes an API. That's where Crossplane comes in: it takes the Kubernetes control plane and makes both its backend *and* its frontend highly extensible. You can extend the backend to control any kind of system, and you can extend the frontend APIs to expose whatever concepts make sense for your organization. Figure 1-2 shows an engineer making an API call to the control plane's frontend API and the control plane backend orchestrating a cloud provider.

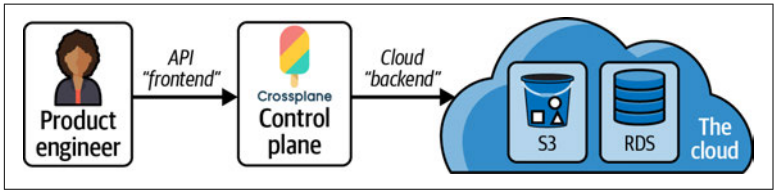


Figure 1-2. A control plane has a backend that orchestrates the data plane and a frontend that exposes an API.

Imagine you're part of the platform-engineering team at a growing technology company—say, a music streaming service. Your team supports 40 engineers who are split across 7 other teams. These teams deliver your company's core product: streaming music. Perhaps they also work on social features or recommendation engines. Let's call these folks *product engineers*. Most of your product-engineering teams want to use PostgreSQL databases to build their features, but they're not experts in securing and scaling them.

Your team's job, as platform engineers, is to make it safer and more efficient for the product engineers to do their regular tasks. This is a form of *force multiplication*: the platform engineers multiply the effectiveness of the product engineers.

How could you make it safer and more efficient for the product engineers to provision PostgreSQL databases?[2]

One option would be to have the platform team manage everyone's databases. In this scenario, product engineers raise a ticket, and the platform team provisions a database for them. You use your database expertise to ensure you're provisioning secure, scalable databases. Perhaps you use an infrastructure-as-code (IAC) tool like HashiCorp's Terraform to automate the provisioning. This option is safe but not very efficient: it doesn't scale well, because it puts the platform team in the critical path. It could be hours or days before you have time to service a request for a new database.

A second option would be to give the product engineers unfettered access to a cloud platform like AWS. This way, they can provision database instances themselves (self-service) by using the cloud provider's CLI, web console, or API and a service like Relational Database Service (RDS). However, cloud services tend to offer so many configuration knobs that you need to be an expert in PostgreSQL *and* RDS to provision secure and scalable databases. This second option may be more efficient than the first, but it's less safe—it would be easy for product engineers to make mistakes.

The sweet spot is your third option: to offer an RDS-like self-service platform of your own design. The product engineers can use a CLI,

2 Crossplane frequently uses database management as an example use case, but you can use Crossplane to build a control plane for any kind of platform. This example could just as easily be serverless functions or Kubernetes clusters.

a web console, or an API to self-service their database needs just as efficiently as if they were using AWS. Because you design the platform yourself, you can bake safety in. You can also increase efficiency by customizing your platform's concepts and configuration knobs to the product engineers. Embracing this approach is what differentiates a platform team from a more general infrastructure or devops team.

Crossplane is a framework that helps you build your self-service platform's cloud control plane. You teach Crossplane what kind of APIs you want your platform to expose and what it should do when product engineers call those APIs. For example, Figure 1-3 shows a control plane configured to expose a SimplePostgres API and to create an RDS instance when a product engineer calls that API. This way, RDS does the heavy lifting of actually running the database, but you choose what configuration knobs are exposed and how they're framed. Perhaps you don't give the product engineers a choice of whether encryption at rest is enabled: it always is. Perhaps you choose to reframe AWS's concept of instance types as a "small, medium, or large" choice. This is simpler to reason about, and it prevents product engineers from choosing unnecessarily expensive instances.

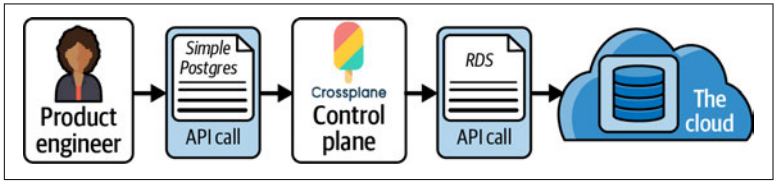


Figure 1-3. Configure Crossplane to reframe complicated cloud APIs using simpler concepts.

With Crossplane, you can build your platform's control plane without needing to code it from scratch. This means you can build a bespoke platform much faster, while still tailoring it to your organization's unique needs.

CHAPTER 2

How a Crossplane Control Plane Works

In my opinion, a good control plane should do three things:

- Drive the data plane toward the desired state
- Protect the data plane
- Be reliably available

In this chapter, I'll explore how a control plane built with Crossplane achieves these goals. The next chapter will touch on how you can use Crossplane to build your cloud control plane.

If you know how Kubernetes's control plane works, you'll notice a lot of familiar concepts in this chapter. This is because Crossplane builds on the Kubernetes control plane and enjoys many of its benefits.

Driving Toward the Desired State

A control plane's core function is to reconcile the actual state of its data plane with a desired state. A thermostat is a good example: you might desire your room to be 72°F, but its actual temperature is 65°. The thermostat can reconcile the actual state of your room (its temperature) with your desired state by running the heat until the room reaches 72°.

You tell Crossplane what the desired state of your data plane is by making API calls. Crossplane exposes a JavaScript object notation (JSON) representational state transfer (REST) API over HTTP. JSON REST APIs are broadly supported by tools and programming languages. This makes it easier to integrate Crossplane with other systems.

Crossplane APIs look a lot like Kubernetes APIs. They're compatible with popular tools in the Kubernetes ecosystem like Helm, Argo CD, Kyverno, and kubectl. Crossplane's APIs are actually Kubernetes custom resources. This means that, like all custom resources, they're *declarative*. You tell Crossplane the desired state of the data plane—not how to achieve it.

APIs exist on a spectrum from purely declarative to purely imperative. *Imperative* APIs require you to specify how to achieve your desired state. Declarative APIs are simpler: would you rather *imperatively* tell your thermostat to run the heater for 20 minutes each time you notice that it's cold or *declaratively* tell it to keep the room at 72°?

A huge benefit of Crossplane's declarative APIs is that if you save the body of your API request to disk, you have a configuration file. This lets you keep your desired state using a revision control system like Git. Crossplane API objects use JSON, but tools like kubectl typically convert them to YAML[1] when you save them to a file. A desired-state API object saved as a YAML file is often called a *manifest*. Example 2-1 shows a Crossplane API resource in YAML form. This will look familiar to you if you've ever seen a Kubernetes Deployment or Pod manifest.

Example 2-1. A SimplePostgres YAML manifest

```
apiVersion: platform.example.org/v1
kind: SimplePostgres
metadata:
  name: product-team-db
spec:
  storageGB: 30
  instanceSize: medium
```

A sample for DSL

1 YAML originally stood for *yet another markup language*. It was later renamed with the recursive acronym YAML ain't markup language.

Each resource in Crossplane's API represents a data plane entity. You can create, read, update, or delete that entity using HTTP verbs (such as POST, GET, PUT, and DELETE). API resources represent *things*, not actions. Think back to the SimplePostgres example from the introduction: each SimplePostgres instance in your data plane would be represented by its own SimplePostgres API resource. Each resource has its own URL in the API, for example *https://api-server.example.org/apis/platform.example.org/v1/simplepostgres/product-team-db*. Note that segments of this URL appear in the YAML manifest shown in Example 2-1.

All Crossplane API resources include at least three top-level fields:

`apiVersion` *and* `kind`
: These fields specify the type of the resource. This makes it easy to identify the type of the resource without knowing its URL, for example when reading a YAML manifest.

metadata
: Metadata that is supported by all resources, including their name, creation time, and arbitrary labels and annotations.

Most Crossplane API resources contain only two other top-level fields: *spec* and *status*. The spec field specifies the desired state of the resource, while the status field reflects its observed actual state. All API resources have OpenAPI v3 schemas.

Crossplane offers a consistent, uniform API to express desired state and read actual state. This makes working with Crossplane predictable—all of its APIs look and feel similar. It reduces users' cognitive burden and helps tools integrate with Crossplane. In many places, Crossplane APIs are more uniform than Kubernetes APIs. For example, the same API field in all Crossplane resources reflects whether they are ready to use.

You might have noticed that a thermostat must measure the temperature of the room to determine whether it needs to run the heat. The thermostat uses a feedback loop to make the right corrections over time. This is called *closed-loop control*, and it's a desirable property of a control plane.[2] Crossplane does this using controllers. A *controller* subscribes to the API that it is responsible for and watches for

2 There's an entire branch of mathematics about this called *control theory*.

changes to its desired state. Each controller is responsible for a single kind of API resource. A SimplePostgres API is powered by a SimplePostgres controller.

When the desired state changes, the controller will do the following:

1. Read the desired state from its API.[3]
2. Observe the actual state of the part of the data plane its API represents.
3. Determine whether the actual state is different from the desired state.
4. Correct the actual state by creating, updating, or deleting something.

Imagine you've configured Crossplane to satisfy calls to the SimplePostgres API by creating RDS instances. Each SimplePostgres instance corresponds to an RDS instance. When the SimplePostgres API is called to create a new PostgreSQL instance, its controller will take these steps:

1. Read the desired state from the SimplePostgres API.
2. Attempt to observe the actual state of the corresponding RDS instance.
3. Find that the RDS instance does not exist.
4. Create the RDS instance, deriving its configuration from the desired state read from the SimplePostgres API.

Crossplane controllers use closed-loop control to ensure that your data plane never drifts from your desired state. Once you tell Crossplane the desired state of your data plane, it will constantly observe the data plane's actual state and quickly self-heal when that drifts.

3 This is called *level-triggered* reconciliation, a term borrowed from electrical engineering. Reading the desired state on every reconcile ensures the controller has the freshest desired state, even if it misses a change or sees changes out of order.

Protecting the Data Plane

The availability of the data plane is usually a lot more important than the availability of the control plane. Think back to our air traffic control example: if the control tower is unavailable, that's far from ideal, but it's nowhere near as bad as a plane crash.

A control plane's broad power over its data plane means that a compromised or misbehaving control plane can negatively affect data plane availability. Imagine if someone broke into the control tower and gave malicious guidance to circling aircraft, or if the control tower accidentally started emitting its signals on a 10-minute delay!

Crossplane protects your data plane in the following ways:

- Using encryption, authentication, and authorization everywhere
- Compartmentalizing responsibility across many controllers
- Limiting the rate at which it interacts with the data plane
- Making sure it's using only the freshest desired state

Encryption, authentication, and authorization make it harder for a malicious party to compromise your control plane and, in turn, to infiltrate or break your data plane. All Crossplane API calls are encrypted using Transport Layer Security (TLS). Crossplane can use any of Kubernetes's battle-hardened authorization systems, including role-based access control (RBAC), at a fine-grained level to determine who may call which API. It can also, optionally, leverage advanced policy engines such as Kyverno and Open Policy Agent's Gatekeeper. API callers can authenticate themselves using one of several methods, including X509 certificates and bearer tokens.

A Crossplane control plane consists of a handful of software components: an API server, an etcd layer, and controllers. Let's look at each more closely:

API server
: The API server is the clearinghouse for data plane state. It securely exposes APIs that you can use to modify the desired state of the data plane. Controllers use these same APIs to record the actual state of the data plane. Most Crossplane deployments use kube-apiserver, but it's compatible with alternatives like kcp.

etcd
: A reliable, distributed, key-value store used to store the desired and actual state of the data plane. As etcd is the storage layer for the API server, nothing else may communicate with it directly.

Controllers
: Controllers read the desired state by making API calls to the API server. They then use this desired state to act on the data plane.[4] Most controllers also record the actual data plane state they've observed, again by making API calls to the API server. Crossplane is made up of many controllers working together in unison, each with its own specific task and API.

Figure 2-1 shows how these components relate to each other.

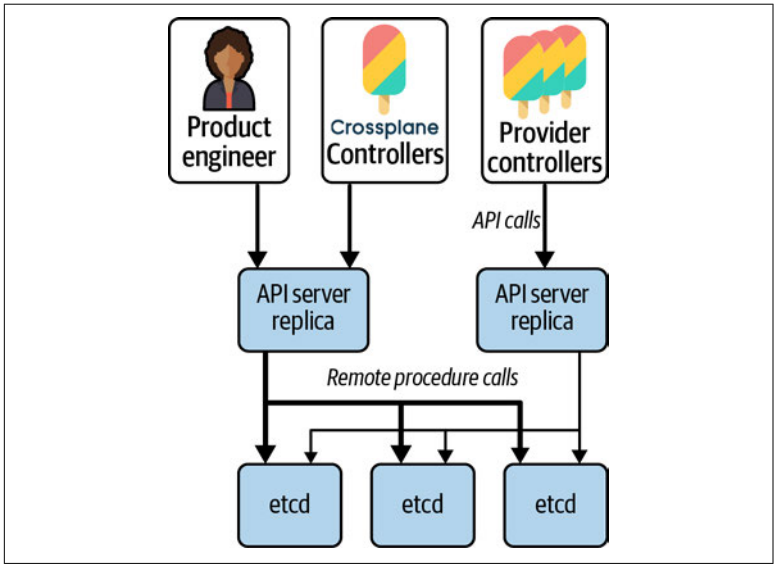


Figure 2-1. The components of a Crossplane control plane

Breaking Crossplane into several distinct components helps protect your data plane by limiting the "blast radius" of a malfunctioning or compromised controller. The API server subjects controllers to the same authentication and authorization policies (such as RBAC)

4 Actually, not all controllers operate *directly* on the data plane. Some controllers read high-level, abstract desired state from the API server and reconcile it by writing a more specific, granular desired state back to the API server for another controller to reconcile.

as any other API caller. When you configure Crossplane to control an external system like AWS, you need only give the relevant controllers access to your AWS account. Controllers can only affect the parts of your data plane that you authorize them to affect.

Finally, Crossplane protects your data plane using distributed-systems best practices like rate limiting, retries, and optimistic concurrency. *Rate limiting* ensures Crossplane doesn't overload your data plane by trying to update its state too frequently—for example by making too many API calls to AWS. *Retries* ensure Crossplane tries again when it encounters a failure. Retries back off exponentially: Crossplane waits a little longer before trying again after each failure. This gives a potentially overloaded external system a chance to recover. *Optimistic concurrency* ensures Crossplane does not accidentally operate on your data plane using out-of-date desired state.

Being Reliably Available

As I mentioned, an unavailable control plane is not ideal. The good news is that if the control plane becomes unavailable, that shouldn't in and of itself damage the data plane. The bad news is that the data plane will be frozen in place. If our example air traffic control tower's radio breaks, the aircraft around the airport would be grounded or (if in the air) stuck in a holding pattern waiting for landing guidance. It's important that your control plane be reliably available so that it can grow, update, and heal your data plane promptly.

Several of the design decisions Crossplane makes to protect the data plane also contribute to the control plane's reliability. For instance, its software components are decoupled, which means failure of one part of the control plane won't necessarily cause the entire control plane to fail. Crossplane's software components are mostly stateless: etcd is responsible for all state storage. This means that the API server can scale horizontally. It also makes it easy to restart failed API server and controller processes. Crossplane runs all of its controllers as Kubernetes Pods. This allows the Kubernetes control plane to heal Crossplane controllers in the same way Crossplane controllers heal your data plane. The control plane as a whole is self-healing.

A production-grade Crossplane control plane is typically replicated across a small fleet of servers. Components like the API server and etcd scale horizontally. It's easy for a large data plane fleet to

accidentally overwhelm a small control plane fleet if the data plane is *pulling* from the control plane. Crossplane controllers call out to (or *push* configuration to) the data plane, preventing this issue.

Finally, Crossplane is built for observability, to help catch reliability issues early. All Crossplane controllers emit Prometheus metrics. Crossplane controllers also emit monitorable Kubernetes events to the API server anytime they operate on the data plane.

CHAPTER 3
Building a Control Plane with Crossplane

Now that you're familiar with the benefits of building a control plane with Crossplane, let's look at how you can do just that.

We designed Crossplane to help platform engineers build cloud control planes for the product engineers they support. As you saw in Chapter 1, platform engineers curate the control plane's APIs and capabilities, while product engineers consume them. In this section, I'll focus on the tasks being done, not roles or titles—after all, we hope Crossplane will be useful to you even if you're not a platform engineer. For that reason, I'll refer to platform engineers as control plane *curators* and product engineers as control plane *consumers*. *Consumers* specify the desired state of the data plane by making API calls to the control plane. The control plane's *curators* define the types and shapes of these APIs.

The APIs that consumers call are abstractions. When they're called, Crossplane resolves the specified desired state to a set of operations it must make on concrete data plane primitives. Think back to the thermostat analogy from Chapter 2: there, the temperature dial is the abstraction. When you set the temperature, the thermostat must resolve the desired state you've specified to a concrete operation—running the heater.

To build a control plane with Crossplane, a curator would take the following steps:

1. Choose the cloud primitives that will make up the data plane.
2. Define the API abstractions consumers will use to control the data plane.
3. Teach Crossplane how to compose their chosen cloud primitives when consumers call their defined APIs.

If you were the curator in the SimplePostgres example, this means you would execute the following steps:

1. Choose your cloud primitives (e.g., RDS or Google CloudSQL instances).
2. Define the schema of the SimplePostgres API resource.
3. Teach Crossplane how to create, update, or delete an RDS instance when a consumer creates, updates, or deletes a SimplePostgres via the API.

The rest of this chapter walks through each of these steps in detail.

Choose Your Cloud Primitives

To extend a Crossplane control plane with support for new cloud primitives, you install a *Provider*. A Crossplane Provider extends a control plane by adding new API endpoints that represent a related set of cloud primitives. Crossplane calls these APIs *managed resources* (MRs). A Provider also installs controllers that reconcile the desired state of its MRs with the data plane.

Some popular Crossplane Providers include:

provider-aws
: Extends your control plane to support controlling AWS cloud services like RDS, Elastic Kubernetes Service (EKS), and Elastic Compute Cloud (EC2). Similar providers exist for other clouds like Azure, GCP, and DigitalOcean.

provider-kubernetes
: Extends your control plane to support controlling Kubernetes API resources like Pods, Deployments, and Services.

provider-terraform
: Extends your control plane to support controlling anything Terraform supports, using your existing Terraform modules and configurations.

provider-sql
: Extends your control plane to support controlling PostgreSQL and MySQL servers, for example by creating databases, users, and roles.

An MR is a *high-fidelity* API representation of a data plane primitive. This means that each MR models a data plane primitive as faithfully as possible. For example, the RDS instance MR of provider-aws supports every field that AWS's RDS instance API supports. Crossplane avoids imposing its own opinions at the MR level, except where necessary to ensure that all MR APIs are idiomatic and consistent.

Most control plane consumers won't interact with MRs directly. Instead, they interact with higher-level API abstractions. When a consumer calls these higher-level APIs, Crossplane reconciles their desired state into one or more MRs. Crossplane calls this *composition*: the higher-level APIs are *composed of* one or more MRs.

If you're familiar with Kubernetes Pods, it might help to think of a Crossplane MR as similar. A Pod is the lowest-level container-orchestration primitive in Kubernetes, and it's an API resource in the Kubernetes API. You *can* call that API directly, but most people don't. Instead, Kubernetes control plane consumers interact with higher-level abstractions like the Deployment API. The Deployment controller reconciles calls to the Deployment API by making calls to the Pod API.[1] So a Deployment is *composed of* Pods.

Define Your Control Plane's API Abstractions

Crossplane calls a control plane's consumer-facing API resources *composite resource claims*, or just *claims*. Claims are the higher-level API abstractions that consumers call to specify the desired state of their data plane. The control plane curators define what types

1 Actually, there's even more abstraction in the Kubernetes example. The Deployment controller creates ReplicaSets, and the ReplicaSet controller creates Pods. Finally, the Pod controller (that is, the Kubelet) creates containers on a node.

of claims exist and their schemas (i.e., the fields they support). A claim is composed of one or more MRs, which is the source of terms like *composite resource* and *composition*. Figure 3-1 shows a SimplePostgres claim that is composed of three MRs, including an RDS instance.

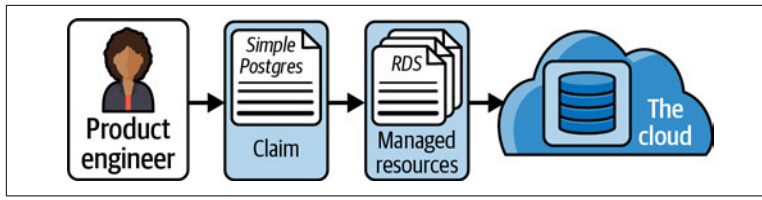


Figure 3-1. A claim is composed of one or more managed resources.

The key difference between a claim and an MR is who defines their type and schema. Provider authors set MR types and schemas—they're relatively inflexible. Control plane curators set claim types and schemas—they can be any shape you need them to be.

When a consumer makes an API call to create, update, or delete a claim, they're updating the desired state of the MRs the claim is composed of. Crossplane reconciles the new desired state with the data plane by following three steps:

1. A control plane consumer makes a claim API request to specify desired state.
2. The claim's controller reconciles the new desired state by making requests to the MR APIs.
3. MR controllers reconcile the MR API requests by controlling the data plane.

The idea of creating a database instance might sound simple. Unfortunately, the reality is usually more complicated. Cloud providers like AWS must cater to a broad consumer base with diverse infrastructure needs. This has resulted in an explosion of API types and configuration knobs. AWS's RDS instance API, for example, has *more than 50 top-level configuration fields*. Consider, also, that to create a *usable* database instance, you may also need to create firewall rules, identity and access management (IAM) bindings, parameters groups, and more.

This is where claims come in. Claim APIs reduce the cognitive load on control plane consumers. As a curator, you define which APIs your consumers interact with. You can choose which configuration fields are relevant to them and which are not. You can name fields in ways that will be easy for consumers to understand. You can even group several MRs representing several cloud primitives into a single API so that, conceptually, they form a single claim. For example, a SimplePostgres claim could be composed of an RDS-instance MR, a firewall-rule MR, and an IAM role-binding MR.

When you first install Crossplane, it won't offer any claim APIs. You must teach it what APIs you want your control plane to have. To define a new kind of claim, you create a *composite resource definition* (XRD),[2] a special kind of API that adds new APIs to Crossplane.[3]

An XRD defines two key things: the *kind* (type) of your new claim API and its *schema* (its shape and supported fields). Claim kinds are almost completely arbitrary. The SimplePostgres example I've used thus far in this report is one example. The claim API conceptually represents a "simple PostgreSQL instance," so it's kind is SimplePostgres. Claim kinds, like all Kubernetes API kinds, are spelled using CamelCase. Your claim kind could just as easily be AcmeCluster, SecureBucket, or FancyDatabase. Most curators pick a kind that succinctly captures what the claim represents to control plane consumers.

When you create an XRD, Crossplane adds a new API endpoint for your new kind of claim. It also starts a controller that reconciles claims, called a *claim controller*. The claim controller doesn't create MRs directly. Figure 3-2 shows that between a claim and its MRs lies an intermediate API resource called a composite resource, or XR. When a consumer creates a claim, Crossplane automatically creates a matching XR. The XR controller then creates MRs.

2 Crossplane often uses *X* instead of *C* to avoid confusion with similar Kubernetes acronyms like CRD.

3 This is a bit meta. You make a Crossplane API call, and Crossplane grows a new API endpoint.

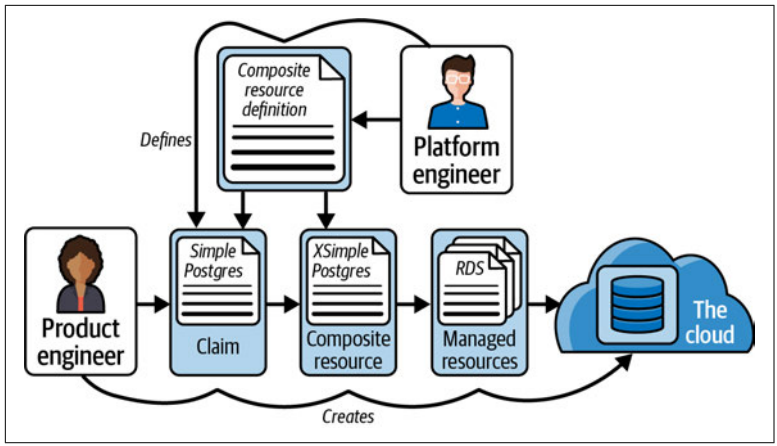


Figure 3-2. A composite resource definition defines the type and schema of a claim and a composite resource.

Most XRDs define both an XR and a claim. They're different APIs, but their schemas are identical. When a consumer calls the claim API to create, update, or delete a claim, Crossplane always calls the corresponding XR API to create, update, or delete the corresponding XR.[4]

All claims have a corresponding XR type, but not all XRs have a corresponding claim type. You can create an XRD that defines only an XR, not a claim. This allows you to define *private* API abstractions: ones that control plane consumers can't use directly. The most common reason you'd want to do this is to layer XRs. Figure 3-3 shows that an XR can be composed of other XRs, which are in turn composed of MRs. This practice is common in larger organizations with multiple, specialized platform teams. Say for example that the security team defines a standard FirewallRule XR. The database team should be allowed to compose a FirewallRule XR into a higher-level SimplePostgres claim, but the average platform consumer should not be able to create a FirewallRule claim directly.

4 The convention is for an XR kind to match its claim kind, prefixed with an *X*. For example, a claim of kind SimplePostgres would create an XR of kind XSimplePostgres.

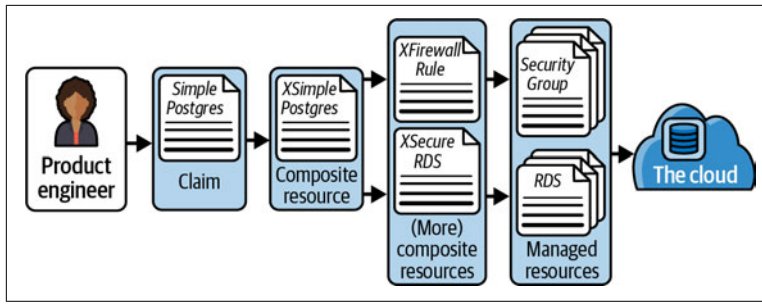


Figure 3-3. A composite resource may be composed of other composite resources.

Teach Crossplane to Compose Your Cloud Primitives

So you've chosen the cloud primitives (MRs) that will make up your data plane. You've also defined your control plane's consumer-facing APIs (claims). Now it's time to teach your control plane what to do when a consumer creates a claim; i.e., what MRs it should compose. Configure this using a (capital *C*) Composition.

Like an XRD, a Composition is an API resource. It's used to curate the control plane rather than to consume it. An XRD teaches Crossplane about a new consumer-facing API; a Composition teaches Crossplane what to do when that API is called. In its simplest form, a Composition is a list of the MRs Crossplane should create, update, or delete when a control plane consumer creates, updates, or deletes a claim. This is illustrated in Figure 3-4.

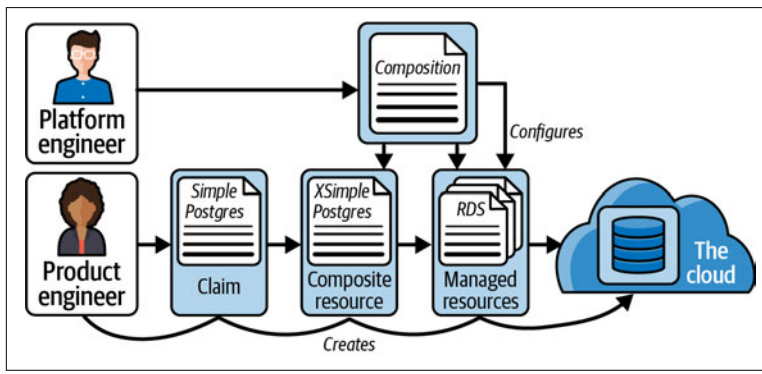


Figure 3-4. A composition configures how Crossplane composes managed resources from a given claim.

In programming terms, an XRD defines an interface, and a Composition satisfies that interface. Different Compositions can satisfy the interface defined by an XRD. You could create a Composition that satisfies a SimplePostgres claim using an AWS RDS instance and another that could satisfy the same claim using a Google CloudSQL instance. When a control plane consumer makes a claim, they can influence which Composition Crossplane will use. Control plane curators can add arbitrary sets of labels to a Composition, such as `provider: aws`, or `region: us-west-1`. The consumer can specify a set of labels as part of their claim's desired state, telling Crossplane to use only a Composition that has those same labels. This behavior is optional; a curator can instead force a particular claim API to always use a specific Composition.

Crossplane supports two styles of Composition: *patch and transform Composition* (P&T) and *Composition Functions*.[5]

P&T Composition
: When using P&T Composition, you specify a *base* list of managed resource templates. Each of these base resources can be *patched* with values copied from the claim. A value can optionally be *transformed* before it's applied to the base, for example to convert megabytes to gigabytes or a string to an integer.

Composition Functions
: When using Composition Functions, you specify a pipeline of Open Container Initiative (OCI) containers. The claim's XR is passed to these containers, which are responsible for returning managed resources for Crossplane to create, update, or delete.

P&T is ideal for simpler Compositions and for those who prefer a zero-code solution. Composition Functions are ideal for advanced Composition logic. They allow you to build containers that use your preferred CLI tool or programming language to implement Composition logic. You don't have to pick one or the other—you can mix both in one Composition.

Offering two styles of Composition allows control plane curators to choose the approaches that are best aligned with their situation, preferences, and experience level. For less complex cases, you don't

5 At the time of writing, Composition Functions are an alpha feature and thus may be subject to change.

need to learn a new programming language or tool, and there are no external dependencies: you just write familiar Kubernetes-style YAML. For advanced cases, you can leverage proven tools and languages with existing ecosystems and documentation.

Package Your Control Plane Configuration

The final piece of the control plane curation puzzle is packaging. Crossplane includes a package manager that allows curators to package their control plane configurations. This makes it easier to share configurations and to distribute them to many control planes.

There are two kinds of Crossplane packages:

Provider package
: A Provider package installs a Provider. It creates API endpoints for all of a Provider's MRs, and it installs controllers to reconcile those MRs with the data plane. Most curators won't create Provider packages but will install them.

Configuration package
: A Configuration package configures your control plane with new capabilities. Typically a Configuration package will configure your control plane to serve one or more new kinds of claim API. It does this by installing a collection of Providers, XRDs, and Compositions. Most curators do curate their own Configuration packages.

A Crossplane package is an OCI image with special Crossplane-specific metadata. This means Crossplane packages can be distributed by any kind of OCI registry, including Amazon's Elastic Container Registry (ECR), Google Container Registry (GCR), and Upbound's Marketplace, which is focused specifically on Crossplane packages.

Packages can depend on other packages. A Configuration, for example, can depend on Providers or other Configurations. This is particularly useful in larger organizations, where a control plane's configuration may be curated by multiple specialized platform teams.

CHAPTER 4

What You Can Build with Crossplane

So Crossplane is a pretty open-ended framework for building control planes. What, specifically, do we see platform engineers building control planes *for*? Let's look at three examples.

A Service Catalog

One common use case is to build a service catalog—a smorgasbord of common cloud services that product engineers can choose to use. At one Fortune 100 company, each product-engineering team has their own AWS account and a lot of freedom to pick which services they use to build their products. The platform-engineering team uses a central, Crossplane-powered control plane to create a *platform runtime* in each team's account. This runtime consists of common foundational infrastructure, like VPC networks and IAM roles. Once the platform runtime is in place, teams can choose to add services like Amazon OpenSearch, S3, and RDS, to name a few. The platform-engineering team appreciates the control plane approach because a control plane handles the entire lifecycle of the services they use—not just provisioning. They picked Crossplane specifically because it allows them to give their product engineers a lot of autonomy while enforcing their best practices using Compositions.

"Batteries Included" Kubernetes Clusters

Another problem I often see teams solving with Crossplane is "batteries included" Kubernetes cluster management. In this scenario, the platform-engineering team's goal is to empower each of the product-engineering teams they support to provision the Kubernetes cluster they'll use to run their product's workloads. We say these clusters are "batteries included" because Crossplane automatically deploys and configures supporting software that makes the clusters more useful, like Argo CD, Prometheus, Fluent Bit, and Istio.

A platform-engineering team can build a control plane for "batteries included" clusters by pairing a Crossplane provider that can create, update, or delete clusters with Crossplane providers that can manage what runs *in* those clusters. For example, one Fortune 500 company uses provider-aws to create an EKS cluster with a few node groups and provider-helm to deploy the supporting software to the EKS cluster. From the product engineer's perspective, all of this is abstracted behind an API request (i.e., a claim) for a standard Kubernetes cluster. The platform-engineering team can even update their Compositions to carefully add and tweak cluster features over time.

Your Own Application Model

Kubernetes is the most popular control plane for application code today. You package your code in an OCI container and use Kubernetes's APIs to deploy and operate it. However, not everyone loves Kubernetes's concepts and APIs. They're frequently criticized as being complex and hard for product engineers to learn. This has led to the creation of application models like Open Application Model (OAM). Most application models are an abstraction layer atop Kubernetes that attempt to reframe its concepts in a developer-friendly, application-focused way. Indeed, the core API type in OAM is called "Application."

Application models like OAM aim for more developer-friendly concepts and APIs than Kubernetes, but they're still *one-size-fits-all*. An emergent pattern in the Crossplane community is platform engineers using Crossplane to build application models that are tailored to their organizations' needs. These application models might use

provider-helm to deploy an application's code to Kubernetes and provider-aws to deploy its infrastructure dependencies such as databases and queues. The models themselves—the abstractions—are Crossplane claims.

Keep in mind that these are just three examples of control planes built with Crossplane—there are many more. Take a look at the Crossplane *ADOPTERS.md* file for more inspiration. It lists organizations that have publicly adopted Crossplane with a brief description of what they're building control planes to do.

Conclusion

I hope this report has given you a better understanding of the value of cloud control planes. I also hope you feel some of the excitement I felt when I first learned about Crossplane! Having spent much of my career as a platform engineer, I believe being able to build bespoke control planes without having to start from scratch is a game changer for engineering productivity.

If you'd like to learn more about Crossplane, their website is the best place to start. From there, you can find our "getting started" documentation and our active Slack community. Another great resource is the Upbound Marketplace. In particular, deploying one of the featured Configuration packages is a great way to try Crossplane out. If you're an AWS customer, you'll also want to check out the Blueprints for Crossplane put together by the folks at AWS.

About the Author

Nic Cope is a senior principal engineer at Upbound, founders of the Rook and Crossplane CNCF projects. Before joining Upbound to help build Crossplane, Nic spent a decade in SRE and platform engineering teams at companies large and small, including Google, Spotify, and Planet Labs. Outside of work Nic is an obsessive pinball player, amateur packrafter, and dog parent to two terriers.