ONE CODE TO RULE THEM ALL:

THE QUEST FOR TRUE MULTI-CLOUD IAC

Vishwas Joshi - Cloud Evangelist

Cloud computing has promised infinite choices. AWS dazzled with scale, Azure excelled at enterprise integration, and Google Cloud attracted innovators with data and AI. Enterprises looked at the menu and ordered everything. But running across multiple clouds soon felt like managing three different operating systems with one team of engineers. Every new region, every new provider, meant another set of templates, consoles, and IaC codebases to maintain.

The dream is simple to say, harder to execute: what if you could write one piece of infrastructure code, and run it on any cloud you choose? A developer shouldn't need to know the quirks of AWS IAM, Azure RBAC, or GCP IAM to get a database. A product team shouldn't worry whether their app lands in ECS, AKS, or GKE. They should ask for a service, and the platform should deliver it, whether the target is AWS, Azure, GCP, or even providers like Alibaba Cloud, Oracle Cloud, DigitalOcean, etc.

This article explores how close we are to that dream. We'll start with the foundations of Infrastructure as Code, then look at three tools that are shaping the path: Terraform, Pulumi, and Crossplane. Each takes a different route to simplify multi-cloud life. Each reveals both how far we've come and how far we still have to go.

OJASQUEST

Infrastructure as Code: The Common Ground

Before diving into the tools, it's worth grounding ourselves in what IaC is and why it matters. At its heart, IaC is a simple proposition: treat your infrastructure the same way you treat application code. That means describing servers, networks, storage, or clusters in a declarative file or program, storing it in Git, and letting automation handle the rest.

The benefits are obvious. It removes the "clickops" problem of configuring things by hand in consoles. It creates consistency every environment, from development to production, can be recreated reliably. It enables version control, rollbacks, peer reviews, and automation pipelines.

In single-cloud scenarios, IaC solved reproducibility and velocity. In multi-cloud, it solves survival. Without IaC, managing AWS, Azure, and GCP simultaneously is chaos. With it, you at least have a common discipline a chance to unify workflows, enforce policy, and abstract away differences.

The challenge is that IaC tools were born inside clouds (CloudFormation, ARM/Bicep, Deployment Manager) or built around them (Terraform, Pulumi, Crossplane). That means abstraction is always partial. You can declare a VM in all three clouds, but their shapes differ. You can request storage, but APIs disagree on defaults. The tools we'll explore show three distinct strategies to bridge this gap.

1. Terraform: The Universal Translator

HashiCorp's Terraform is the veteran in this story. Born when AWS dominated and Azure/GCP were just finding their feet, Terraform became the de facto tool for multi-cloud provisioning. Its secret sauce is the provider model: a plug-in system where every cloud or service can define resources and expose them in Terraform's declarative HCL language.

To the user, a Terraform file looks consistent whether it's creating an EC2 instance, an Azure VM, or a GCP Compute Engine. Under the hood, each provider maps those declarations into the cloud's API calls. That design allowed Terraform to scale far beyond the big three it now supports Kubernetes, SaaS services, databases, and more.

But can Terraform deliver “one code, many clouds”? Not natively. A developer can’t just declare resource "cloud_vm" "demo" and expect it to work across providers. Instead, platform teams build modules reusable wrappers that expose a generic interface and branch internally based on the chosen provider.

Here’s a taste of what that looks like:

```
variable "cloud" { default = "aws" }
variable "image" { default = "nginx:latest" }

resource "aws_ecs_service" "app" {
 count = var.cloud == "aws" ? 1 : 0
 name = "demo"
 # ...
}
resource "azurerm_container_app" "app" {
 count = var.cloud == "azure" ? 1 : 0
 name = "demo"
 # ...
}
resource "google_cloud_run_service" "app" {
 count = var.cloud == "gcp" ? 1 : 0
 name = "demo"
 # ...
}
```

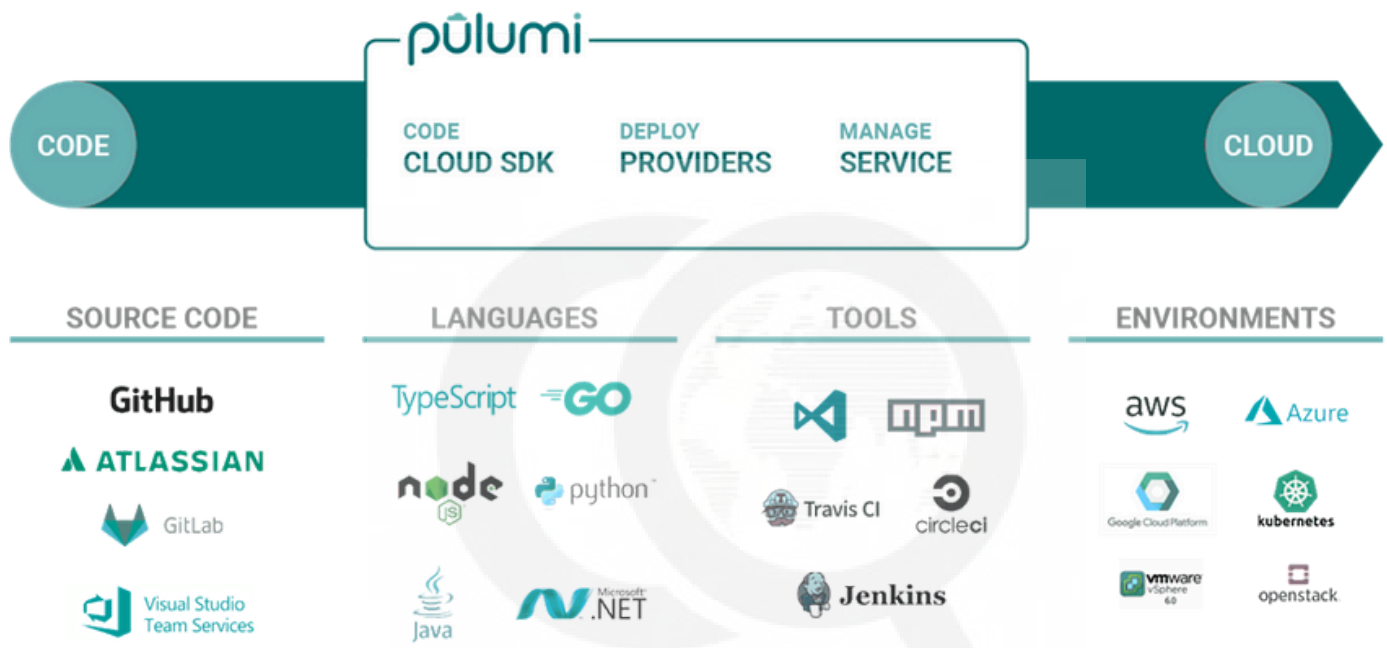
The user calls a single module, perhaps named vm or service, and passes cloud = "aws" or "azure". Inside, the provider-specific resources are hidden. From a developer’s perspective, it feels like one API.

Terraform’s strength is breadth and maturity. Its weakness is that abstraction layers must be hand-crafted. HashiCorp itself hasn’t released a vendor-maintained “universal API” for all clouds. Enterprises often buy curated libraries (from partners like Gruntwork or Cloud Posse) to avoid writing everything themselves.

So Terraform is the universal translator it speaks every cloud's language, but you still need to phrase the dictionary yourself.

2. Pulumi: One Language to Bind Them

Where Terraform uses HCL, Pulumi took a bold step: write infrastructure in real programming languages. TypeScript, Python, Go, or C#. That choice unlocked a new path for abstraction: libraries and functions



Imagine a developer writing:

```
import { createService } from "@company/platform";

const svc = createService("demo", { cloud: "aws", image: "nginx" });
export const url = svc.url;
```

Behind that innocent call, the Pulumi library decides whether to provision an ECS service, an Azure Container App, or a GCP Cloud Run service. For the developer, it's just one function.

This approach feels natural to software engineers because it mirrors how they already consume SDKs. Teams can publish internal NPM or PyPI packages that define cloud-agnostic APIs (createVM, createBucket, createCluster). The messy cloud-specific wiring is hidden in the library.

Pulumi itself has started down this road with Crosswalk for AWS a higher-level library that makes AWS services easier to consume. While Crosswalk isn't multi-cloud yet, the same principle could be extended. The platform team writes the abstraction once, and every developer enjoys the simplicity.

A short Pulumi sketch illustrates the power:

```
export function createService(name: string, opts: { cloud: string; image: string }) {
 if (opts.cloud === "aws") {
   return new awsx.ecs.FargateService(name, { /* ... */ });
 } else if (opts.cloud === "azure") {
   return new azure.app.ContainerApp(name, { /* ... */ });
 } else {
return new gcp.cloudrun.Service(name, { /* ... */ });
 }
}
```

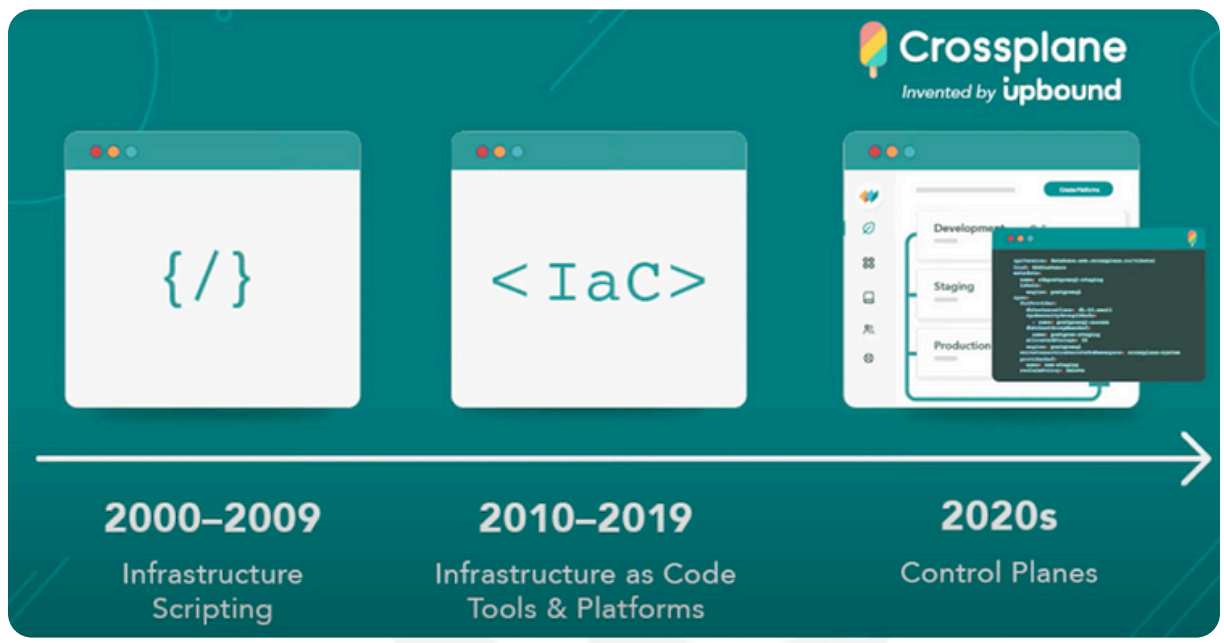
The difference from Terraform is subtle but important. In Terraform, the abstraction is still written in declarative HCL with count toggles. In Pulumi, you can use full programming constructs conditionals, loops, classes, inheritance. That makes building and maintaining unified APIs more natural.

Pulumi doesn't yet ship an official "multi-cloud Crosswalk," but its model is well suited to one. For now, organizations that invest in Pulumi often create these abstractions internally, reducing the burden on developers who just need to deploy services without thinking about providers.

3. Crossplane: Kubernetes as the Universal API

Crossplane takes a different stance altogether. Instead of writing IaC files or functions, you extend Kubernetes itself. Every cloud resource becomes a Kubernetes object. Need a database? Apply a YAML. Need a cluster? Apply a YAML.

At first glance, this looks similar to Terraform's Kubernetes provider. But Crossplane goes further: it introduces Composite Resources (XRs). An XR is a higher-level API you define once like XBucket or XCluster. Developers only interact with that XR. Behind the scenes, Crossplane maps it to provider-specific resources via Compositions.



Here's an example:

```
apiVersion: storage.example.org/v1
kind: XBucket
metadata:
 name: demo
spec:
 cloud: gcp
versioning: true
```

The developer just asks for an XBucket. Crossplane notices cloud: gcp and provisions a StorageBucket. If it were aws, it would create an S3Bucket. If azure, a StorageAccount.
From the developer's view, it's a single, generic API. No wrapper code, no conditionals. The wrapper exists, but it's maintained once in the platform as a Composition, not scattered across app teams.

Crossplane, especially with Upbound Cloud, delivers the closest thing today to a universal cloud API. You can define a portfolio of XResource types and let your developers consume them consistently. That doesn't mean every cloud service is covered edge cases and niche services still lag but for 80–90% of common workloads (compute, storage, databases, clusters), the pattern holds.

The experience is remarkably clean:

- A developer writes one YAML.
- The platform ensures it lands correctly in AWS, Azure, or GCP.
- Policies, security, and governance are enforced centrally.

Crossplane isn't as old as Terraform or as developer-friendly as Pulumi, but in terms of delivering a "single API to many clouds," it is already the most tangible realization of that vision.

Conclusion: The Road to True Portability

So where are we?

Terraform is the translator it covers everything but leaves abstraction in your hands. Pulumi is the language binder it makes writing abstractions easier and more natural but doesn't provide them ready-made. Crossplane is the API unifier it offers developers a single declarative surface, with provider specifics hidden behind the platform.
None of them have achieved a true universal IaC API where you declare cloud_vm once and it works everywhere out of the box. Cloud vendors still design their APIs to be distinct. But third-party tools are getting closer.

Today, if you want to showcase "one code, many clouds" without writing your own wrappers, Crossplane via Upbound Cloud is your best bet. If you want to craft your own abstractions in code, Pulumi gives you the most expressive toolkit. If you want mature breadth and a vast ecosystem, Terraform is still king.

The future likely belongs to a blend of these ideas:

vendor-neutral abstractions maintained by open communities, libraries that feel native to developers, and Kubernetes-style APIs that unify everything. Until then, platform teams will keep building the bridges. The good news is that the bridges are getting sturdier and for the first time, developers can cross into multi-cloud without falling into the gaps.