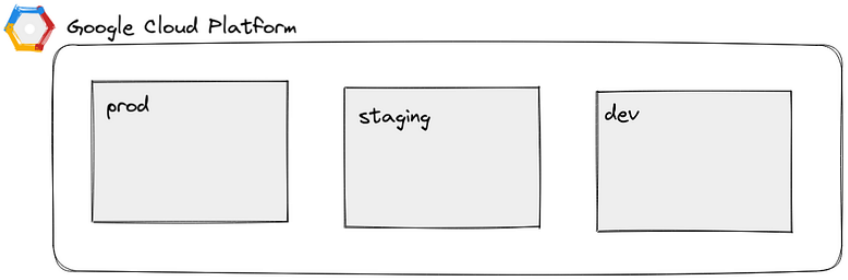
The Cloud Architecture at Oda

A new team member recently asked about any existing diagrams or schematics of our technical architecture. The truth is, we often don't have such documentation. Our tech stack is ever-evolving, making yesterday's documents obsolete today. Recognizing the importance of having a current snapshot, however, I took it upon myself to create a quick diagram. For those who haven't joined yet but have the same question, this blog post is for you!

You cannot see the boundaries in the sky

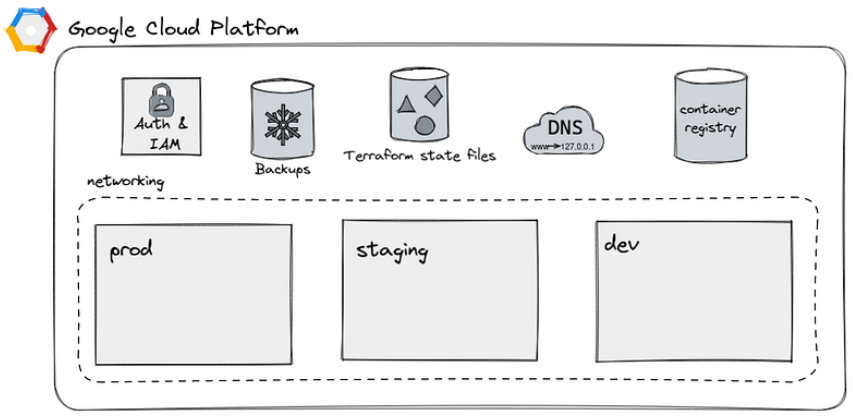


We use Google Cloud Platform (GCP). If i recall correctly, we ended up there because that was what the majority of the engineers at that time knew best. Our organizational GCP account is roughly split into three; `prod`, `staging` and `dev`. These are conceptual properties and does not hold much value on their own – but we use them to apply boundaries where we can.

The lines between these environments can blur quickly, raising immediate questions; what do you do with networking? What about the container registry? And DNS? And then you might have an internal monitoring system to keep an eye on them all, or integrations with external vendors of which you, surprise, get no `dev` environment. Do you duplicate everything?

The pragmatic approach

If you aim for the highest possible parity between your environments, you might as well build your data-centers yourselves, and you need at least three of them! So, we don't. At least, not yet.

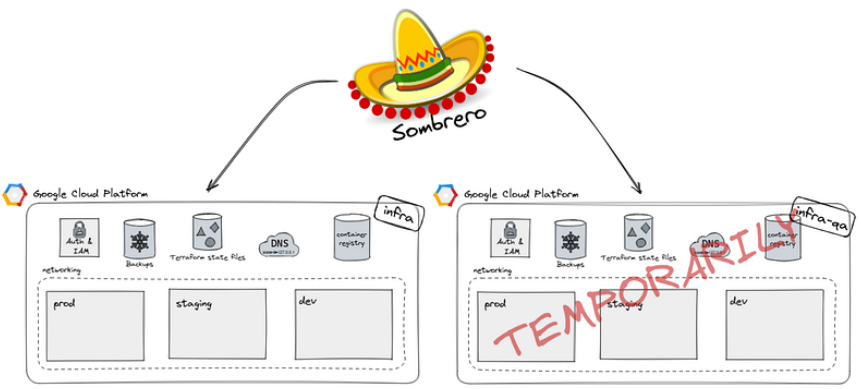


In the realm of infrastructure engineering, there are no perfect solutions—only trade-offs. Valuing simplicity and informed risk-taking, we opt for a set of common services that aren't tied to a specific «environment».

From a developer's perspective, these services function as if they were externalized, not bound to any specific environment within our architecture. If we had to assign environment labels to these services, `infra-prod` would be the closest fit. However, we strive to keep environment-specific tags to a minimum to avoid unnecessary complexity.

Shadow infra

At the infrastructure level, we maintain a best-effort clone designated as `infra-qa` . Instead of it being a fourth enviromnet, try instead to think of it as a second dimension. This is specifically for quality assurance testing of high-risk changes related to networking, IAM rules, or Kubernetes clusters. All of this is provisioned from a decentralized project we call **Sombrero**, which use CDKTF, a Python-based implementation of Terraform. As a measure of its scope, Sombrero's terraform state file currently manages approximately 3,600 cloud resources, all orchestrated by about 3,000 lines of Python code. It has served us extremely well.

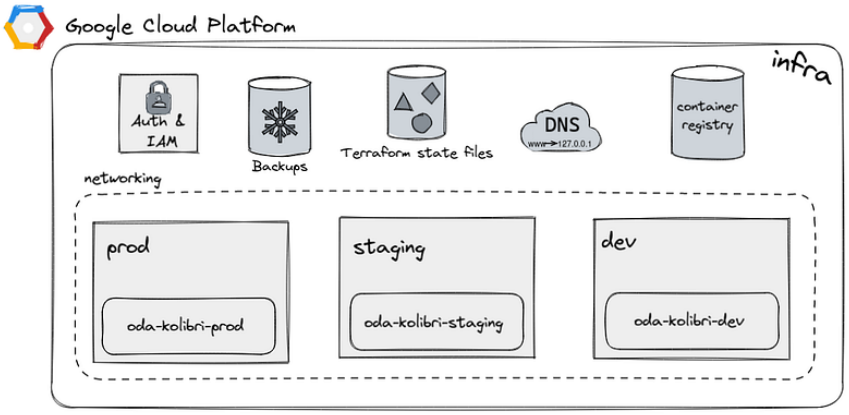


A developers perspective

Our developers don't really see all of this, and that's good. It is abstracted away. Their view is is only `prod`, `staging` and `dev`.

```
 settings['shop'].projects = [
     ...
+    *create_project_envs(name="kolibri"),
     ...
 ]
```

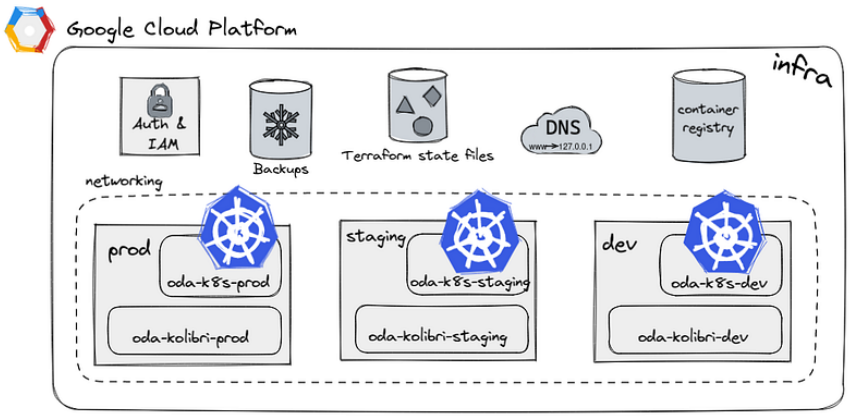
I.e, when our «Shop» developers wanted a dedicated corner in our Oda cloud for their design system Kolibri, all that is needed is this one-liner above. This yields three Google Cloud Projects, fully plumbed with networking, Google APIs, IAM rules and other sane defaults.



Currently, Sombrero manages around 60 projects, equating to approximately 180 Google Cloud Platform (GCP) projects when considering all associated environments. Each GCP project encapsulates various resources—ranging from service accounts and IAM rules to managed services like Redis, CloudSQL, Cloud Storage, Secrets and alike.

Puzzling pieces

The infrastructure team utilizes Sombrero as well. For instance, we manage a project specifically dedicated to Kubernetes, aptly named `k8s`.



Kubernetes is probably the biggest and heaviest piece in this puzzle. Ironically, even the Kubernetes' *logo* was too big to fit the diagram boxes. Within these environments, we strive to maintain as high a level of parity as possible. The developers should feel *very* confident that, if they've played in `dev`, deployed to `staging`, and then pushed to `prod` – even on a Friday afternoon – it will work.

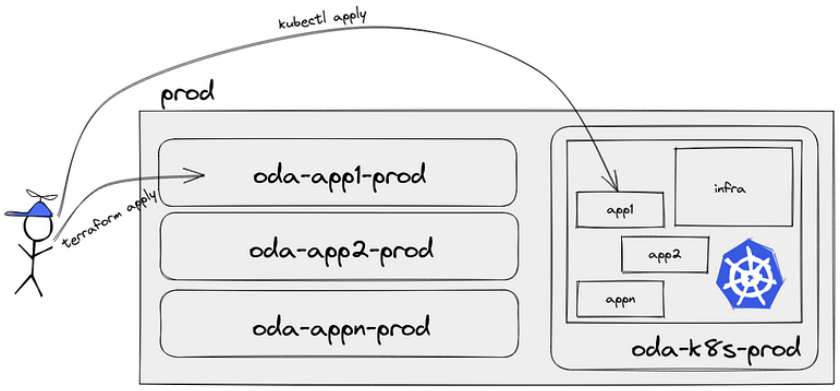
Starting with a single production cluster is acceptable. An old adage in physical machine management says, «You can have two once you've learned how to handle one», and this wisdom holds true here as well. While a single cluster offers ease of management, it does come at the cost of a wider blast radius and potentially weaker security boundaries. Fortunately, these drawbacks can be effectively mitigated. If you wonder what would be a good fit for you, this article does a good job in describing the tradeoffs involved.

Design for failure

A few plausible risks this model faces are:

> *A Kubernetes upgrade produces unexpected side effects*
>
> *An cluster-wide component (such as a CNI plugin) doesn't work as expected*
>
> *An erroneous configuration is made to one of the cluster components*
>
> *An outage occurs in the underlying infrastructure*

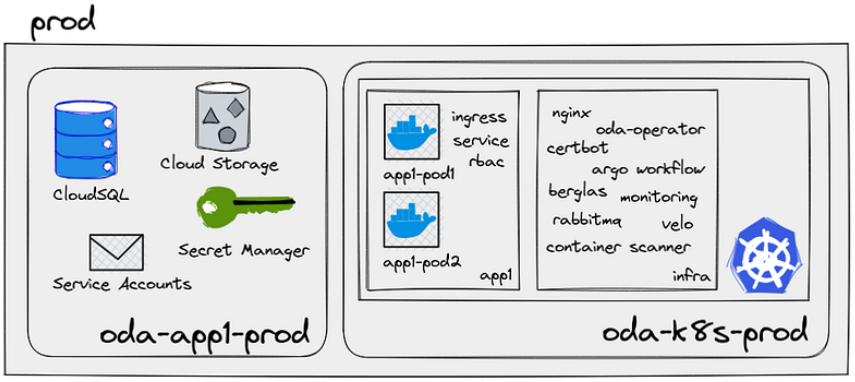
For mitigation, we now and then rebuild our `dev` environment. This practice not only ensures quick cluster recovery but also helps identify configuration drift. And by maintaining high environmental parity, we are able to detect most issues before they reach the production environment.



Speaking of risk mitigation, the above drawing is actually a bit deceiving; developers don't usually deploy directly like this. Developers don't even have access to our prod environment. Not staging either! We have a nice homegrown deploy system for that. We've iterated through various deployment systems—Bash scripts, ArgoCD, Spinnaker, and Octopus—to ultimately build our own. It's a Python wrapper around `terraform apply` and `kubectl apply`, enhanced with some Jinja templating. This space is so full of all-singing, all-dancing software. Be critical!

Let's zoom in some more!

«Enhance on that reflection!»



This design allow our developers to own and maintain most of the often-changed stuff, both in GCP land and in their own namespace in Kubernetes. Among many others, the infrastructure team handles:

> **nginx-ingress**, including automatically provisioned DNS, TLS and authentication services.
> *Why?* We used nginx in our onprem days. Sometimes the best tool is the tool you know.
>
> **Velo,** in-house tool that handles the deployments of Terraform and Kubernetes manifests.

Why? As mentioned, everything else was a bit 💩.

Berglas, that injects secrets from the team's Secret Manager and into their pods.
Why? Hashicorp Vault is too complicated. Kubernetes Secrets too simple (an insecure). Berglas has some issues, but not enough for us to change it.

Argo Workflows, a better cronjob system.
Why? It is very much optimized for all of our data pipelines that is more then just plain «cronjobs», but they work for small jobs as well. So we try to avoid Kubernetes Cronjobs to keep variance low.

RabbitMQ, as our shared messaging broker between apps.
Why? Like with nginx, we knew it quite well from earlier. Most times, the best tool for the job is the tool you know (but you probably don't want Kafka).

oda-operator, the lightest of operators that so far only validates labels (using Kopf in Python. We like Python).
Why? Seems like a good idea to play with the operator pattern. It has already gotten in handy. But more likely, we'll move over to using Kyverno instead due to ease of maintenance.

We require our developers to understand a sizeable part of both Terraform and the Kubernetes API, but we try to limit the number of components we collectively use to a minimum. If two pieces of tech does somewhat the same, we try to kill one or the other. Variance is the enemy ⚔️. For all the Terraform stuff, we try hard to provide our teams with Terraform modules that has defaults that make sense in Oda space. Most prominent are

Google Storage Buckets

Google Secrets

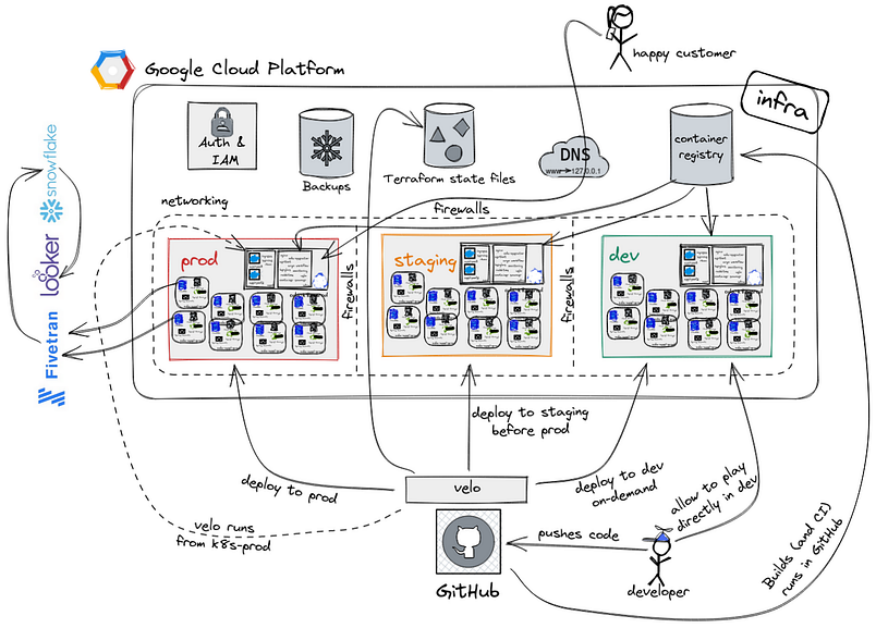
Google Cloud SQL

Google Redis

Google Kubernetes Engine (only infra uses this)

«Do you have a high level diagram of your infrastructure?»

Okay, enough details, let's zoom all the way back! And add arrows so that it looks like we're doing stuff too.



There is somewhat a lot going on – when you add some customer traffic to it, some external services, some deployments here and there—but you should be able to follow understand the gist of it.

I haven't mentioned IAM much, but from a birds eye view, that is not super complicated either. We heavily leverage GCP's resource hierarchy, which allow you to lay out all your projects in folders. Most IAM rules tied to read-only permissions are inherited through folder structures, while write access is either bounded directly to the projects with the Access Elevator (for prod and staging) or via Sombrero (for dev and edge-cases).

Wrapping up

So there you have it—a high-level snapshot, much like the one I shared with our newest team member. While there are nuances and exceptions that I've left out for brevity, our guiding principles remain the same:

> Emphasize simplicity
>
> Ensure reproducibility
>
> Maintain high parity between environments
>
> Prioritize transparent configurations
>
> Phase out legacy systems promptly
>
> Implement robust risk mitigations

These principles have been instrumental in shaping our technical decisions and will likely continue to guide us in the future.