# Prodspec and Annealing: Intent-based actuation for Google Production

By: Pierre Palatin, with Betsy Beyer.

With special thanks to Niccolo' Cascarano, Adam Zalcman, Michael Wildpaner, Pim van Pelt, and Lee Verberne.

Online description: "A technical deep dive into Google's intent-based production management"

## Preface / Introduction

As organizations mature, they develop more tools. At Google, we continually create new external and internal services, as well as infrastructure to support these services. By 2013, we started outgrowing the simple automation workflows we used to update and maintain our services. Each service required complex update logic, and had to sustain infrastructure changes, frequent cluster turnups and turndowns, and more. The workflows for configuring multiple, interacting services were becoming hard to maintain— when they even existed. We needed a new solution to keep pace with the growing scale and variety of configurations involved. In response, we developed a declarative automation system that acts as a unified control plane and replaces workflows for these cases. This system consists of two main tools: **Prodspec**, a tool to describe a service's infrastructure, and **Annealing**, a tool that updates production to match the output of Prodspec. This article discusses the problems we had to solve and the architectural choices we made along the way.

Prodspec and Annealing have one fundamental aspect in common: rather than focusing on driving individual changes to production, focus on the state you want to reach. Instead of maintaining step-by-step workflows, service owners use their configuration of choice to describe what they intend their infrastructure to look like: which jobs to run, the load-balancer setup, the location of the database schema, and so on. Based on that information,  Prodspec and Annealing take over and transform the configurations into a uniform structure, which is then actuated. The actuation is safe and continuous: the automation systems repeatedly compare the intended state as expressed in the user's model to the state of production, and automatically trigger a reconciliation when safe. Service owners are freed from having to manually shepherd configuration changes to production.

Since we started developing Prodspec and Annealing around 2015, the seemingly simple concept of intent-based actuation has become a de facto standard. A large part of Google production now has some layers characterized through their intended state, rather than relying on the emergent state created by workflows—a trend that's more broadly reflected across the industry.

## Terminology

There are a lot of moving pieces necessary to run a modern service. Particularly when talking about infrastructure-as-a-service, terminology can be confusing. For example, who is the "user"? The person or service using the infrastructure? The end user? Someone or something else?

Figure 1 shows how we'll talk about the various parties involved in this article:

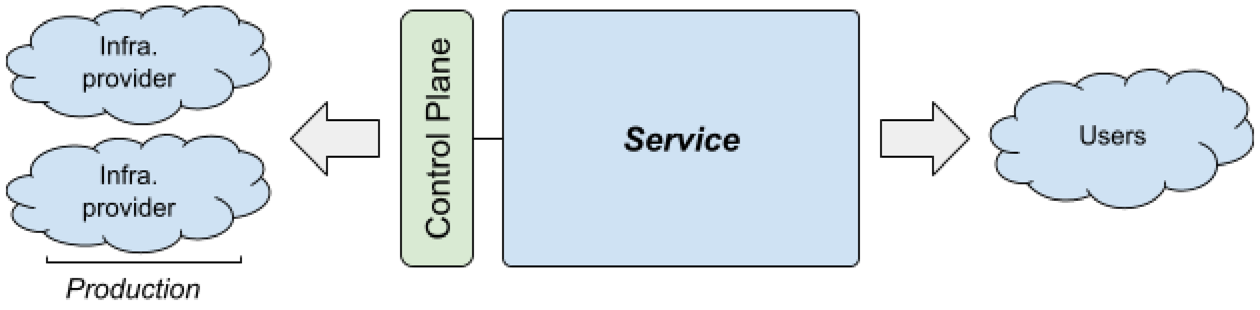

***Figure 1:*** *Terminology as centered around the relative notion of a "service".*

***Service*** is a user-facing system a team wants to run, such as Gmail or Maps. A service can be composed of multiple narrower, internal services—for example, the Gmail service for detecting spam is just one of the services that makes up Gmail.

To help illustrate the ideas in this article, we will refer to the serving side of the Shakespeare service, a hypothetical service represented in Figure 2.

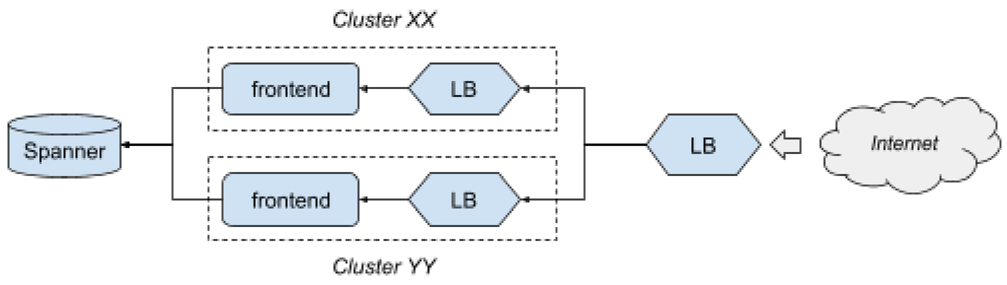

***Figure 2:*** *The simple Shakespeare service. LB stands for "load balancer".*

This service is a simplified web app, made of:

- A binary running in two clusters for redundancy, implementing the frontend logic.

- Load balancing, with both a per-cluster and a global configuration.
- A global Spanner database.

The ***production***—or ***service***—***infrastructure*** provides the building blocks necessary for a service to serve user requests—for example, the Borg cluster management system, firmware on the network switches, etc. Each of these components is an **asset**.

The Shakespeare service uses multiple *infrastructure providers*: Borg to run the binary, GSLB to manage the load balancing, and Google's shared Spanner infrastructure.

The ***control plane*** is what the service uses to manage production service infrastructure— for example, to add a VM or set up a load balancer. The control plane can range from humans ("I will copy my new binary to the server") to complex automated systems ("I'm using machine learning to control the updates"). The control plane includes ***change management***: the logic in the control plane that allows the service to progress from one state of production to another in a safe and controlled way.

In the Shakespeare service, the *control plane* for the frontend is the Borg API, which configures the corresponding jobs and runs them in each cluster.

In this article, we define a ***job*** as a set of similar tasks, similar to a Kubernetes *ReplicaSet*. A ***task*** is a single running instance, often a single process, similar to a Kubernetes *Pod*. A given job runs all its tasks in a single cluster. A ***cluster*** is a set of compute nodes able to run multiple tasks and clusters, and is usually colocated geographically.

In Shakespeare service, each frontend is a *job* in a given *cluster*.

In the next sections we focus on how we introduced an explicit control plane for our production to improve change management.

## Our challenges

Service design tends to focus on the service infrastructure, addressing important but fundamentally static questions: How are your user requests served? Which servers and databases does your service use? How much traffic should your service support?

Those are important questions, but our services are alive and ever-changing: binary versions are updated, and instances are added and removed. Architectures also often evolve: we might want to add a new cache in a specific cluster, remove some outdated logging, and so on.

In 2014, we found ourselves in the position of not adequately accounting for the live nature of our services. For most of our services, we made infrastructure changes using hand-crafted procedural workflows: push x, then y; perform the odd change manually.

But teams were commonly managing tens of services apiece, and each service had many jobs, databases, configurations, and custom management procedures. The existing solutions were not scaling for two main reasons:

- Infrastructure configurations and APIs were heterogeneous and hard to connect together— for example, different services had different configuration languages, levels of abstraction, storage and push mechanisms, etc. As a result, infrastructure was inconsistent, and establishing common change management was difficult.
- Production change management was a brittle process, with little understanding of the interactions between changes. Automated turnups were an exercise in frustration.

## The Configuration Gap

From an infrastructure provider perspective, a given service is likely to have significant redundancy in its configuration. In Shakespeare service, everything runs under the same identity, so it's redundant to specify the user for each frontend job. However, because the infrastructure can run jobs for many services, configuration must still specify the user of each job.

A simple solution for the infrastructure provider is to provide a more advanced configuration surface—for example, a templating language. However, from the service perspective, this approach doesn't necessarily eliminate redundancy. The service has to configure multiple providers, and some information is usually common across those providers. For example, the name of a database is relevant both for the jobs using it (compute provider) and for configuring the database (database provider).

A configuration system is useful in these types of scenarios. Whoever maintains the service can create a high-level description of their service (e.g., *run N tasks in clusters x, y, and z*), and the configuration system expands that description into a suitable format for each infrastructure provider.

While no two services are the same, services tend to fall into broad categories, sometimes known as **service models**. A service model might target simple services like the Shakespeare service: a setup with a frontend, a load balancer, and a database. The service model just needs to know which binary to run, how many clusters to run it in, and the database schema. The logic of the service model then expands that service configuration. For example, the service model would transform the list of clusters into a configuration for the load balancer infrastructure, with specific capacity targets.

In practice, a service model can be as simple as a script that expands a config file into whatever the infrastructure providers need; a service model might also be arbitrarily complex pipelines.

If you're dealing with multiple service models and multiple infrastructure providers, you have to maintain integrations between N service models and M infrastructure providers. This integration includes both generating each provider's specific configuration and also deploying it— a process which can vary significantly across providers.

We call this the NxM problem, as illustrated in Figure 3.

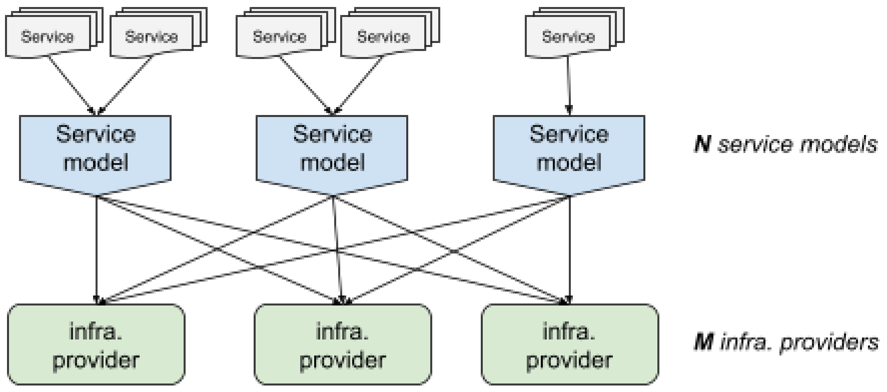

***Figure 3****: Each service model must integrate with each infrastructure provider.*

As long as you're only dealing with a few infrastructure providers, the NxM problem is manageable. As the number of infrastructure providers increases (look at the number of products available on Google Cloud, AWS or Microsoft Azure!), the quality and existence of these integrations starts to suffer.

Reducing the number of infrastructure providers is not really an option. Each provides a different service (for example, a key-value store, a pubsub system, etc.), and while some redundancy might exist, this is the exception more than the rule. It's also not feasible to have a single service model—that model would have to expose every single feature of each infrastructure provider, defeating the purpose of having a service model in the first place.

Google engineers were spending a significant amount of time connecting their preferred service model to the infrastructure they wanted to use. We clearly needed to approach configuration differently.

### Deploying at scale

Before we started addressing this problem space, we relied on traditional workflow engines. For example, a workflow outlined the steps to deploy a new binary: canary in cluster X, then deploy N at a time in clusters Y and Z, run a test, and so on. Creating a workflow required quite a bit of work, so we only had workflows for frequent cases like binary version updates. We often handled less common cases, such as turnups, manually. When a script existed, it was frequently outdated.

Those workflows are easy to conceptualize for the operator: do X, then Y. You want to add a load test? Just add a step. As appealing as these workflows are in many aspects, we were running into problems when scaling our usage of workflows.

First, scaling workflows led to a lot of duplication. Each service had a good reason to have some subtly different logic, which meant it needed a custom implementation. There were a few attempts at homogenizing workflow implementations, but only services fitting a very specific mold would fit. The resulting duplication made services across our fleet inconsistent. Best practices, such as progressive rollouts across clusters, were hard to enshrine in common parts.

Workflows were also fragile. Naive workflows make many implicit assumptions about the state of production, leading to unexpected failure modes. To prevent mistakes, we were adding preconditions: is the canary cluster actually serving live traffic? Is there an outage currently underway? But as we scaled up to hundreds of services with tens of infra providers, this web of preconditions becomes tricky. Each workflow needs to be aware of the status of the others.

The interactions became a $N^2$ problem, where N is the number of assets that comprise your service infrastructure. When you update a single asset, you have to consider the impact it will have on all the other assets or workflows. For example, is it fine to restart this cache now, or should it wait on other caches to be stable? And if I had to manually change one live workflow, which other workflows am I supposed to also nudge one way or another?

## Our solutions: Prodspec and Annealing

It was becoming impossible to design each workflow by hand. Something had to change to reduce the number of interactions and expectations. Our solutions were Prodspec and Annealing. Faced with a proliferation of service models and deployment scaling pains, we moved toward declarative management of our production through intent, managed by Prodspec, with continuous enforcement of that intent, performed by a system called Annealing.

In a workflow-oriented production management model, a large part of the state of production exists only in production. For example, your frontend runs version X because a few days ago, you started a rollout of this specific version.

In contrast, declarative production means that you write the **intent** of your production state—that your production is supposed to run version X—in a configuration file or a database. The production state is now derived from this intent. Paired with continuous enforcement, this ensures that production matches what people expect.

In our experience, the notion of intent is often seen as intuitive and rarely causes confusion. However, maintaining intent can be difficult, and require extensive logic. Prodspec (which we discuss further in the following section) is our solution to this problem of modeling and generation.

Intent-based actuation required a large shift in perception of how production was meant to be modified. To phrase the problem in the "pet vs cattle" metaphor: previously, we treated workflows like pets: we kept a running inventory of individual workflows, hand-tuned them, and interacted with them individually. Intent-based actuation instead uses an enforcement system that treats production assets as cattle: special cases become rare, and scaling becomes much easier. To this end, we created a continuous enforcement system called Annealing (described below).

A large part of our production management is now intent-based and actuates hundreds of changes per second to our infrastructure—where a single change is pushing a whole job, updating a configuration, applying a new database schema, and so on. Of course, workflows still exist in some spaces. In many cases, this is largely because our tooling does not yet cover a particular space (e.g., batch pipelines), but there are also cases where specialized requirements make workflows a better fit.

## Sidebar: A note on other solutions

Many of the concepts described in this article might be familiar from the broader market of service deployment and management. Declarative automation has become more common in the last few years, so you may find parallels to Google's declarative automation systems, Prodspec and Annealing, in your own production management environment.

**Kubernetes** is probably the closest similar system. Kubernetes was independently started by Google around the same time as Prodspec and Annealing. Initially, Kubernetes aimed to be a container orchestration system built upon a framework for declarative service management. It was a greenfield project targeted for use by a variety of services outside of Google. The nature of infrastructure in the open world requires Kubernetes to be flexible to accommodate many different styles of production management.

In contrast, Prodspec and Annealing had to work with existing infrastructure and configurations, while initially targeting very large services. But these projects only dealt with internal Google infrastructure, which has some uniformity (albeit relative) compared to the outside world.

So while the problem spaces were similar, the goal and constraints of Kubernetes vs. Prodspec and Annealing were different. As a result, their prioritizations and solutions were different. This is a case of coevolution, and it is clear that many of the projects' foundational concepts ended up being similar— at least on the surface. For example, Kubernetes CRDs and Prodspec assets fill a similar need. Prodspec has quite strict parameters for creating an asset, while Kubernetes provides more flexibility by allowing partial mutation of resources.

**Terraform** is an open-source tool for defining and providing datacenter infrastructure using a declarative configuration language. With a first release in 2014, its timeline is similar to Kubernetes and Annealing. There are many similarities between Annealing and Terraform, despite the fact that they were developed entirely independently. For example, Terraform providers are similar to Annealing plugins, both enabling integration with arbitrary infrastructure. Terraform works toward updating production state to match a user-provided intent—the same as Annealing. Nevertheless, there are still a few major differences. For example:

- Annealing is built toward continuous enforcement. Annealing applies updated configurations only when it's safe to do so, and without requiring human interactions. Annealing also monitors the health of the service after applying a change.
- Terraform has a uniform configuration surface through HCL. Conversely, Prodspec directly consumes existing configuration sources.
- Prodspec forces hermeticity, allowing it to generate configuration data without access to the production it describes. This configuration data can be compared across versions and read by any tool— not only the actuation layers.
- Prodspec is authoritative and Annealing is built to recover state as needed from production. This avoids the need of a state file like Terraform, but requires clean resource naming from the infrastructure provider and impacts turndown management.
- Prodspec and Annealing scale to the millions of resources Google has to manage.

As an interesting side note, Annealing integrates many other tools to actuate production, and this includes Terraform in some cases.

The rest of the article focuses only on how we applied Prodspec and Annealing to our internal services.

## Intent management: Prodspec

To solve the NxM problem (see The Configuration Gap), we introduced an explicit unified model of production called "Intent".

As shown in Figure 4, the intent model fits between the service models and the infrastructure providers. Rather than having to deal with many different service models, infrastructure providers feed from a unified intent. And instead of having to adapt to the idiosyncrasies of each infrastructure provider, service models can target a standardized representation (the intent model).

To return to the Shakespeare service: it would have a configuration driving its service model, which in turn would generate the intent necessary to configure Borg, load balancing, and Spanner.

This way, we separate the generation of the intent from the actuation of intent, shifting the NxM problem to a N+M situation, making the diversity of configurations and service models manageable.

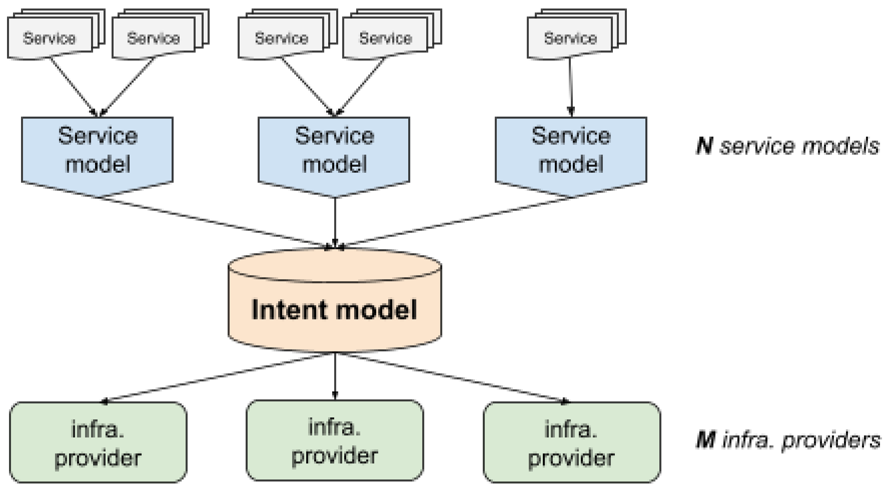

***Figure 4**: Moving to a N+M problem. Actuation layer is not represented.*

This explicit intent also allows us to introspect the configuration, which makes troubleshooting complex setups easier. For example, with a templating system directly integrated into a provider, it's hard to tell whether problems that arise are caused by your template logic, or by the provider's application of that logic. Explicit intent makes it easier to determine which layer (see Figure 4) is causing an issue. Is the intent what you expect? If yes, the problem is on the provider side. If not, the problem is with your service model.

Our intent-based production model and its tooling is called Prodspec. It is the evolution of ideas that started with Gmail in the mid 2000s. It has several components:

- A data model that structures the content.
- A pipeline to generate the content.
- A serving infrastructure.

The following sections discuss the details of the data model and how we generate the content. We don't go into details about how we serve the content, as we use a fairly standard setup.

## Data Model

Prodspec's declarative intent is modeled through a few abstractions, as shown in Figures 5 and 6:

- **Assets**: Each asset describes a specific aspect of production.
- **Partitions**: Organize the assets in administrative domains.
- **Incarnations***:* Represent snapshots of partitions at a given point in time.

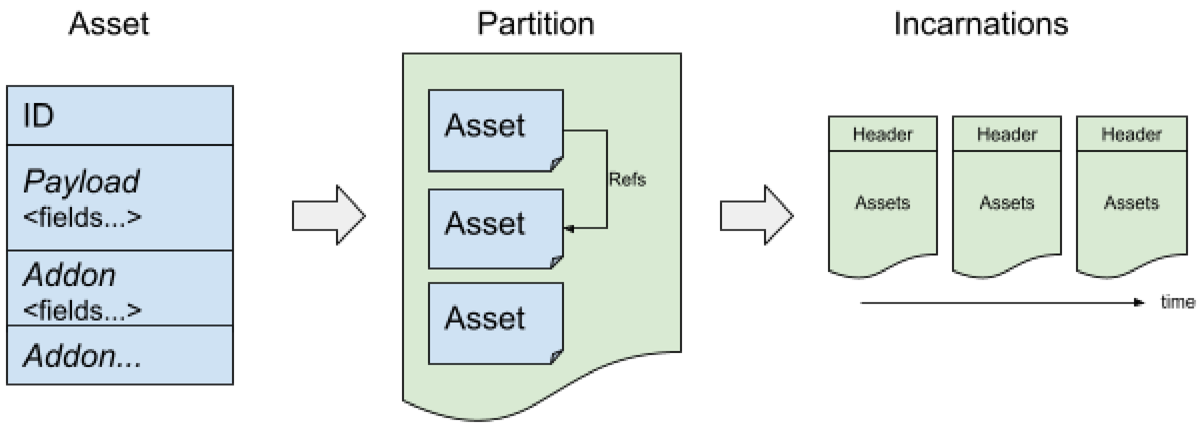

***Figure 5****: Prodspec's data model.*

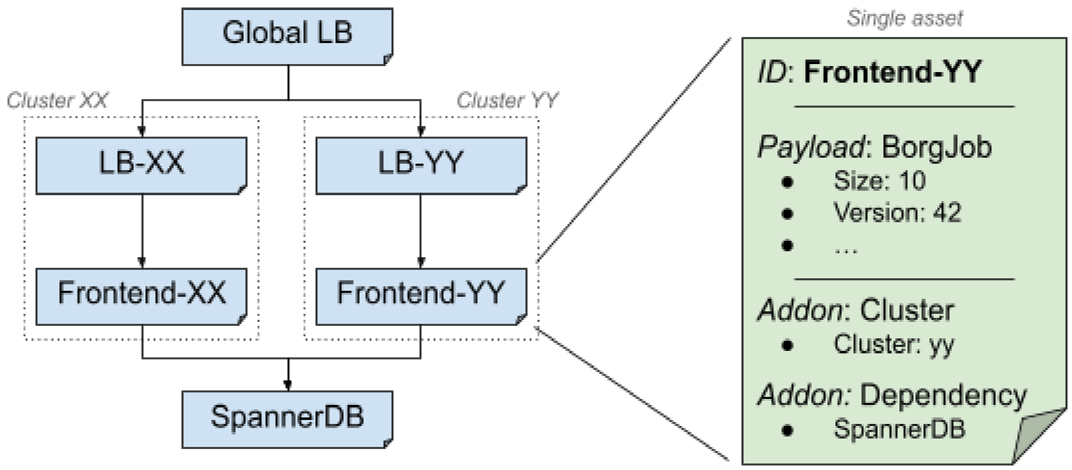

***Figure 6****: A simplified view of the Shakespeare service in Prodspec. LB stands for "load balancer".*

## Assets

Assets are the core of Prodspec's data model. An asset contains the configuration of one specific resource of a given infrastructure provider. For example, an asset can represent the resources in a given cluster or instructions for how to configure a specific job.

In the Shakespeare service (see Figure 6), we would want to have an asset for each frontend job, an asset representing the per-cell load-balancer configuration, an asset for the global load balancer configuration, and an asset for the Spanner DB schema.

An asset is a very generic abstraction with the following structure:

- A string identifier, simply called "asset ID".
- A payload.
- Zero or more addons.

The payloads and the addons are arbitrary protobuf messages. The type of the payload defines the type of the asset. For example, if the payload is a protobuf message of type *prodspec.asset.BorgJob*, it means the asset represents a job running on Borg and will be referred as a BorgJob asset.

The addons make it possible to add arbitrary metadata to an asset, no matter its type. Addons are particularly useful for cross-cutting concerns. For example, the common "impacted cluster" addon gives an idea of the area of influence of an asset— if this asset is unhealthy, will it cause an outage just for one cluster, or for the service globally? Addons also provide space for user-specific data without polluting the payload definition.

Assets are often less than 1kb of data and typically less than 10kb, while we enforce a strict maximum of 150kb. The goal behind this restriction is to make assets easy to consume. This way, tooling can load assets in memory without having to worry about the cost of doing so. Our approach is inspired by Entity-Component systems.

The content of an asset provides sufficient information to construct the full state of the modeled resource. Specifically, it should be possible to recreate the production state without having to look at the current state of production—this is a one-way process.

However, all the configuration doesn't have to fit within an asset. For example, we certainly have our fair share of outliers that require megabytes of configuration. In these cases, we hold a reference to external sources of data, such as packages, databases, and version control. Those references should point to immutable external data.

In the Shakespeare service, that might mean the asset for Spanner just has a pointer to the location where the reference database schema is stored.

Assets often refer to specific resources—databases, jobs, configurations, and so on. However, it is also common to model higher level concepts. For example, an asset can describe the version of a binary we want to eventually deploy, while an individual job asset might still refer to a previous version. Assets representing the grouping (e.g., per failure domain) of other assets are also common.

### Partitions

Assets are grouped into administrative boundaries called partitions. Partitions typically match 1:1 with a service. However, there are exceptions. For example, a given user might want to have one partition for its QA environment and another for its Prod environment. Another user might use the same partition for both its QA and Prod environments. In practice, we leverage the notion of partitions to simplify many administrative aspects. To name a few aspects:

- Content generation occurs per partition.
- Many ACLs are set per partition.
- Enforcement is fully isolated per partition.
- An asset ID must be unique within a partition.

Assets within partitions are not structured, but instead are organized in a flat unordered list. Over the years, we've found that there is really no one-size-fits-all approach for structuring service assets. However, we don't entirely forgo structure: instead of structuring the assets themselves, asset fields can contain **references** to other assets. This makes it possible to create multiple hierarchies on the same assets. For example, the cluster hierarchy is different from the dependency hierarchy. Those reference fields are explicitly marked as such, allowing programmatic discovery.

### Incarnations

We regularly snapshot the content of a partition. Those snapshots are called *incarnations*, and each incarnation has a unique ID. Incarnations are the only way to access Prodspec data. Besides being a natural consequence of the generation logic (see Generation Pipeline), the incarnation model provides **immutability** and **consistency**.

***Immutability*** ensures that all actors accessing Prodspec make decisions based on the same data. For example, if a server is responsible for validating that an asset is safe to push, the server actuating the push is guaranteed to work on the same data. Caching becomes trivial to get right, and it is possible to inspect the information used for automated decisions after the fact.

***Consistency*** is important because we rarely look up assets in isolation. An asset might need to reference other assets to assemble a full configuration. If the content of those assets comes from different points in time, you might end up with broken assumptions. Imagine that in the Shakespeare service, the capacity in one cluster is reduced. This change requires decreasing both the size of the frontend job in that cluster and the corresponding load balancer configuration. Without some form of consistency, Annealing would be at risk of reducing the size of the job before updating the load balancer configuration, leading to an overload.

## Generation Pipeline

Early on in Prodspec's development, we decided to separate how assets are created from how they are consumed.

We could have taken a simpler approach by using a database to store the assets, and using that database for both updating the content (for example, changing the value of an asset field) and reading the content. This model, which Kubernetes uses, has its advantages. However, it's easy for content across fields and assets to become inconsistent when updating them directly in a database. For example, a job renaming might miss some of the jobs, so every person who writes or updates content bears the burden of keeping information consistent.

Instead, Prodspec separates content creation from content consumption. Incarnations and their assets are generated from **Sources of Truth (SoTs)**. SoTs are often simple configuration files, but can sometimes be more complex sources, such as arbitrary databases. Compared to a more traditional database model, Prodspec's approach has several advantages:

- *We can optimize the data model to make data easy to consume*. Most data models avoid redundancy, as redundancy makes it difficult for writers to edit the data consistently. Because our model doesn't permit editing data in place, we can afford to generate duplicated information. This way, the logic in consumers can stay simple— for example, they don't need special logic to find out where the default value for a field is located, and instead just rely on the data to have been expanded in place.

- *Integration with existing configuration.* This was a major driver for creating our generation pipeline. In practice, users have integrated Prodspec with a large number of configuration formats and practices.
- *Sources of Truth can be optimized for human or programmatic editing.* This is useful because we need to configure network switches quite differently than how we configure the Google Maps frontend. For example, you might want to always change the binary version of all the frontend servers together. With a simple database approach, that would require editing every single asset, leading to potential inconsistencies, especially for more complex cases. Instead, with Prodspec you can have a source of truth with only one field for that binary version and generate all the frontend assets, guaranteeing their consistency.
- *Easier backward compatibility.* We commonly add a new field to fix or expand the semantics of an existing field. When we do so, we add logic in the generation pipeline to automatically fill in the content of one field with the other. This way, producers and consumers can be updated to use the new field progressively, and we don't need to perform any risky synchronized updates.

Figure 7 shows the pipeline generating Prodspec content, which we unpack below.

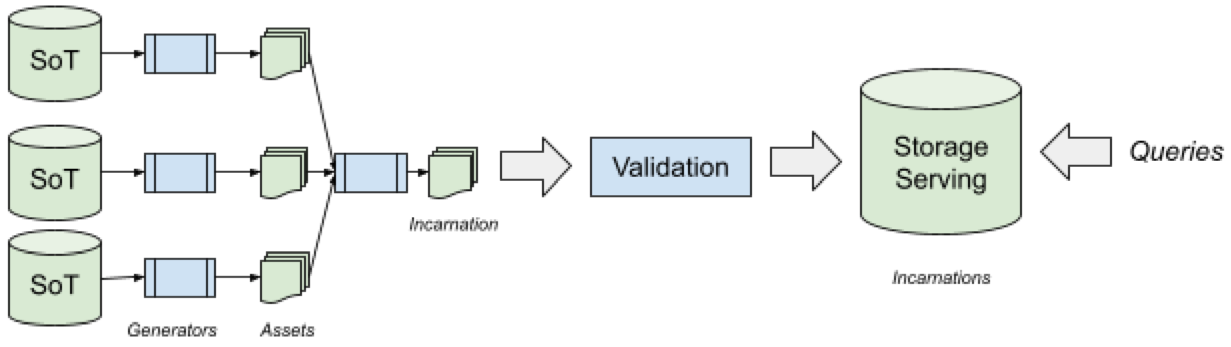

***Figure 7***: *Prodspec's pipeline.*

***Sources of Truth*** are the starting point of the pipeline. Whenever a SoT changes, we rerun the full pipeline, which creates a new incarnation. This process occurs continuously. Because some partitions undergo many changes, we coalesce regeneration. Therefore, a new incarnation might be caused by multiple SoT changes.

A new user of Prodspec must choose which sources of truth will describe their service. In practice, most of our users fall into very common patterns, and choose a ready-to-use setup. For users with more specialized needs, we allow custom SoT formats and generators.

In the Shakespeare service, we might have a simple manifest to indicate where the frontend must run, its binary version, and its sizing. Those SoTs are sufficient to configure the frontend and load balancer assets. The database schema likely comes from another SoT.

We maintain most SoTs on our version control system, and execute generation logic as part of our hermetic build. Keeping SoTs in version control allows us to observe changes over time and to trace changes. Hermetic builds allow for reproducibility, a useful tool to help guarantee stable configuration and better debuggability.

***Generators*** transform sources of truth into assets. Some generators consume the output of another generator instead of a SoT, forming a pipeline. For example, one of our generators adds to each asset an addon describing the "physical cluster" of that asset by extrapolating the payload content.

Generators allow for arbitrary logic, as expanding a service model to a prod model can be complex. While we would love to avoid the complexity, forcing simplification through some form of templating likely isn't helpful. Instead, we chose to embrace this complexity: we consider creating your service configuration to be a process like any other in your stack, which therefore deserves a regular programming language. Rather than relying on templating or a configuration language, we split the data (the config) from the transformation.

As with SoTs, most users simply reuse common generators, and only users with more specialized needs maintain custom generators.

The generator for the Shakespeare service manifest would create two assets per cluster listed in the manifest: one for the job, and one for the local load balancer configuration.

***Validation***: We run each generated incarnation through an extensive series of validators. Most validators verify self consistency of the incarnation, while some cross-check the content with external databases. It's extremely important to have a good suite of validators—corruption is a very common source of configuration issues, and validators can be quite effective at detecting corruption and preventing bad configuration from reaching production. We almost always add multiple validators when adding new asset types or addons.

***Storage***: Once an incarnation is validated, we store the result in Spanner. While a specific incarnation is rarely used for long, it's valuable to keep the results around for debugging problems after the fact.

***Serving:*** Stored incarnations are accessed through a query server. Consumers can request the latest incarnation, a specific incarnation, or a few other variations. They can also filter out the assets they want to see, which is useful when dealing with incarnations that have thousands of assets. We use Spanner indexing to filter data relatively quickly. The filtering language isn't made to be precise: its goal is to reduce the amount of data to manipulate. More complex filtering should be performed on the client side.

## Safe, Continuous Deployment: Annealing

We created Annealing to drive continuous deployment. Once intent is declared in Prodspec, we want production to match that intent. Updating the intent usually involves manual work and approval, but after that point, automation should remove the need for human interaction. Annealing takes that role.

The goal of continuous enforcement is to make production match intent as safely and quickly as possible. Annealing applies changes progressively— for example, by managing canaries or controlled rollouts. This tool provides an ideal spot to apply multiple safety policies: because Annealing sees all changes that need to be applied, it can decide which changes are safe, and when to apply those changes. Consider the case of rate limiting: Annealing can enforce a limit on the number of updates to a job, even if multiple processes or people are interacting with that job in parallel.

As a rule, we like to avoid manual approval when reconciling production, because in our experience, this type of manual intervention rarely brings anything useful. We still allow for manual approval for particularly critical changes (e.g., data deletion), in addition to all standard automated checks. We keep those occurrences to the minimum to avoid desensitization of the human approvers.

Annealing takes care of all infrastructure configuration, from binary version updates to quota management, database schemas or load balancer configurations. Over the years, we've found that infrastructure is always more dynamic than expected—for example, you might need to perform a "one-time setup" of a cluster more than once when you later need to turn up a test instance. As a result, we encourage users to encode all bits of production into Prodspec, and have them enforced by Annealing. While this approach is initially costly, it pays off quickly.

We've also found that extensive modeling in Prodspec helps with turnups specifically. Our traditional approach for turnups was a mix of documentation and custom workflows. Inevitably, those workflows broke, so using them required a lot of hand holding. Extensive continuous enforcement largely eliminates this problem. Since all infrastructure aspects of a service are continuously evaluated, issues surface immediately and not only the next time the turnup workflow is executed.

Our continuous enforcement is largely driven by two layers:

- The **enforcement layer**, driven by the Enforcer, which drives changes from the intent in Prodspec to production.
- The **strategies layer,** driven by Strategist, which schedules enforcement by updating the intent.

When possible, we prefer to express enforcement logic and constraints at the Enforcer level, which is a stateless layer that's easier to understand and manipulate. However, this approach is not always doable or desirable— for example, a week-long careful deployment of a new binary version probably warrants slowly updating the intent.

## Enforcement

Enforcer takes over the last mile of actuation. As shown in Figure 8, Enforcer tracks the intent and calls plugins to update the state of production. Enforcer has no control over the intent: its only decision is whether each specific asset can be pushed now or needs to be delayed.

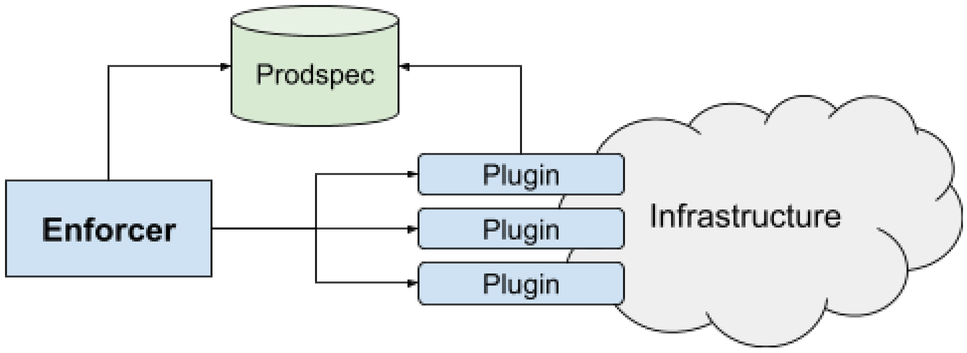

***Figure 8**: Enforcer and its plugins.*

All interaction with the infrastructure occurs through plugins, as Enforcer never touches or looks at production directly. This allows any user to add more integration points, which is useful because services often have some specific needs, such as a custom config or APIs. The plugin approach also helps with deployment: instead of rolling out a monolithic binary, we roll out many small binaries, each with its own lifecycle.

When the enforcer calls a plugin, it only provides an incarnation ID and the asset ID to operate on. Plugins contact Prodspec to get more information as needed.

Annealing has two main types of plugins:

- **Asset plugins** manage the interaction of Annealing with production for a given type of asset. Asset plugins implement two methods:

    - *Diff*, to determine whether a given incarnation matches production. It is the only mechanism to extract information about the intent or the state of production.
    - *Push,* to update production, using the configuration specified in the intent.
- **Check plugins** indicate whether an asset can be pushed now or should be delayed. All checks must pass for an asset to be pushed.

In the Shakespeare service, we need to have asset plugins for all asset types: Borg job, load-balancer resources, and the Spanner database schema. We might also want a check plugin to prevent pushes on weekends.

Annealing plugins and Kubernetes controllers are similar in the sense that they both abstract the logic needed to manipulate specific aspects of production. However, they differ in implementation:

- Kubernetes controllers watch resources for changes. Annealing plugins do nothing unless explicitly called.
- Annealing plugins are more fine-grained. A single Kubernetes controller might look up whether an action is needed (asset plugin diff), verify that the change can be pushed now (check plugin), and perform the update (asset plugin push).

Enforcer deals with each asset independently. Originally, we attempted to evaluate all assets before taking decisions, but we hit scalability issues.

As shown in Figure 9, Enforcer runs an infinite loop for each asset with the following steps:

1. **Pinning*:*** Determines which incarnation to use as intent.
2. **Diff:** If no difference exists between intent and production, the loop iteration stops.
3. **Check:** When a push is needed, verifies whether the push can proceed now. Otherwise, the loop iteration stops.
4. **Push:** Updates the infrastructure based on the intent.

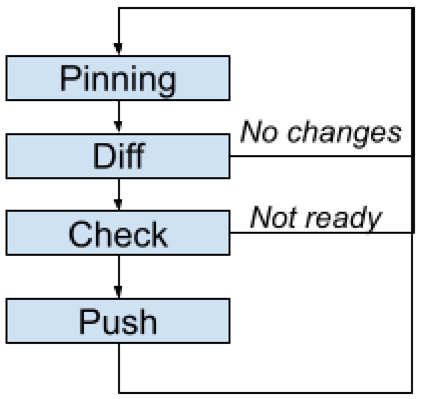

***Figure 9****: Enforcer's asset loop*

The loop does not pick up each incarnation one after the other, and there is no guarantee that each individual incarnation will be enforced. Instead, each iteration uses the latest incarnation. This approach avoids getting stuck on a broken incarnation. Instead, there's an opportunity to fix the intent, which will be automatically picked up. As a consequence, if a specific intermediate state must be reflected in production, you must wait for that state to be enforced before updating the intent further.

Each asset is independent, and the loop runs independently for each asset. However, we don't push assets at will—this is where checks are fundamental. We use checks to delay—not reject—changes to production.

Annealing has visibility and control across the whole service it enforces. This gives Annealing a unique centralized point of view, allowing checks to easily enforce invariants across production. This makes checks often conceptually simple, but powerful. For example:

- The calendar check prevents pushes on weekends and holidays.
- The monitoring check verifies that no alert is currently firing, or that the system is not currently overloaded.
- The capacity check blocks pushes that would reduce the serving capacity below the maximum recent usage.
- The dependency solver orders concurrent changes to ensure the right order of execution.

The dependency solver was the first check we introduced. It makes sure that assets are pushed in the correct order when resized. Consider Figure 6 in terms of the Shakespeare service: when reducing the footprint in a cluster, you usually want to update the load balancer configuration to reduce the maximum capacity served by that cluster before reducing the replica count of the frontend— that way, you do not end up with a frontend that's unable to cope with the load sent to it. The solver check allows you to update the capacity of both the load balancer and the frontend in one change, while Annealing will push these changes in the safe order.

The dependency solver check uses the following logic:

1. A request to push an asset *A* is made, as described in previous sections.
2. Enforcer calls the check plugins, including the solver.
3. The solver checks the diff of asset *A*. If the diff indicates a change not impacting capacity, the check passes.
4. If the check finds a change in capacity, the solver queries the dependencies of the asset. Those dependencies are explicitly listed in Prodspec with an addon and are often automatically generated in the Prodspec generation pipeline.
5. The solver requests a diff against production for each dependency *B*.

6. If the diff on asset *B* indicates that it requires a capacity change for the service to operate properly after asset *A* is pushed, then the push of *A* is blocked until the diff on asset *B* disappears. Otherwise, asset *A* can proceed and be pushed.

The exact logic to determine whether a push can proceed can be complex, but also makes this pattern generic— for example, you can specialize the solver to enforce static ordering ("always push asset *A* before asset *B*") or version ordering ("asset *A* must always run a version equal or greater than asset *B*").

## Progressive rollouts

Enforcer only takes care of the last mile: once it determines the specific intent for a given asset, Enforcer checks that it is safe to perform the update, then drives the update.

But what if you want to deploy a change progressively at a larger scale? Let's imagine that the Shakespeare service is running in 10 clusters instead of just one. You would not want to update every single cluster at the same time. Instead, you would want to set the new target state (for example, running v2 of the binary), and then update each asset in a controlled manner.

A server named Strategist performs progressive rollouts by continuously running the following three steps, shown in Figure 10:

1. **Select:** Determines which assets can be updated at this point of the rollout, based on the rollout policies and checks. Should we pick a specific cluster? Can we update more than one asset now?
2. **Update:** Changes the state of production for the selected assets to the new state.
3. **Validate:** Determines if the change was good. If not, the rollout should stop here, and potentially be followed by a rollback.

Strategist runs these three steps continuously until all assets affected by the rollout are updated or it identifies a problem.

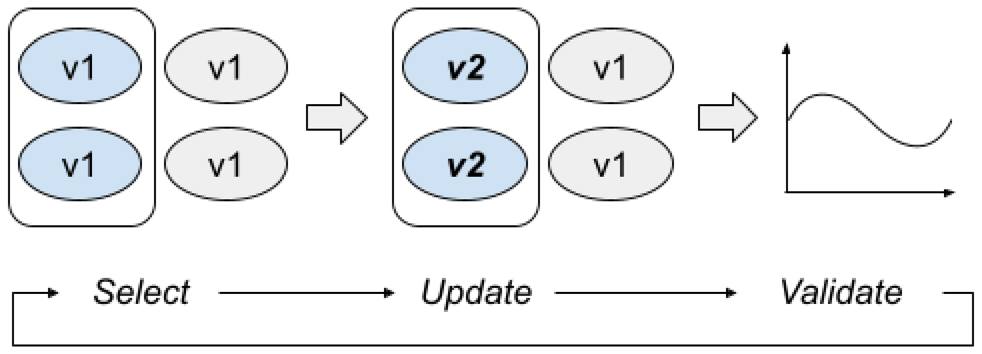

***Figure 10**: The Select-Update-Validate loop*

### Select

The goal of the Select step is to pick which assets can be updated right now. This step can return nothing if no asset is ready to be pushed. In general, rollout policies are encoded here— for example, a policy that specifies that a canary should be pushed first, followed by some period of delay, and then incrementally to the rest of production, with some further delays between each step.

The select step is implemented through a stateless service called *Target Selection*. This service offers a single RPC method, where:

- The input is the rollout's list of assets and information about which assets have already been updated.
- The output is a list of assets that are not yet updated, but ready to be updated at this moment. This list can be, and often is, empty— for example, because some delay between stages of the rollout is configured.

Target Selection is configured through Prodspec, and implements policies such as:

- Push everything at once.
- Push one cluster at a time, in a predetermined order.
- Push one asset, then all other assets.

We've found that even within the timeframe of a single rollout, we need to be able to deal with a changing environment. For example, if a rollout lasts two weeks and affects hundreds of assets, some assets will probably be added and removed over the course of the rollout. The rollout policy itself might change— either organically or because of an external constraint. For example, a critical issue might require a quick rollout to deploy a mitigation. In these scenarios, the statelessness of Target Selection is useful. Because Target Selection only cares only about what was and was not updated, it can deal with a dynamic environment.

### Update

Once assets are selected to be updated, Enforcer drives the actual production change. There are two fundamental ways to drive those changes:

- **Actuation-based** updates carefully *push* the intent. The intent represents the final desired state for all assets. The update step triggers enforcement of selected assets, while other assets are not actively enforced.
- **Intent-based** updates carefully *change* the intent. The intent represents the incremental desired state. The update step updates the SoT for the selected assets; all assets are continuously and immediately enforced in the background.

Those two approaches are found in many other systems. Each has various tradeoffs, and we use both in practice.

As shown in Figure 11, *actuation-based* rollouts start by having a change to the SoT immediately change the intent of *all* assets. This in turn generates a new Prodspec incarnation. However, Enforcer continues to use the previous incarnation, so production is not updated. Then, as assets are selected for update, Enforcer is instructed to use the new incarnation.

In the Shakespeare service, that would mean that a single version is used to configure the frontend binary in all clusters, and then each cluster is pushed only when needed.

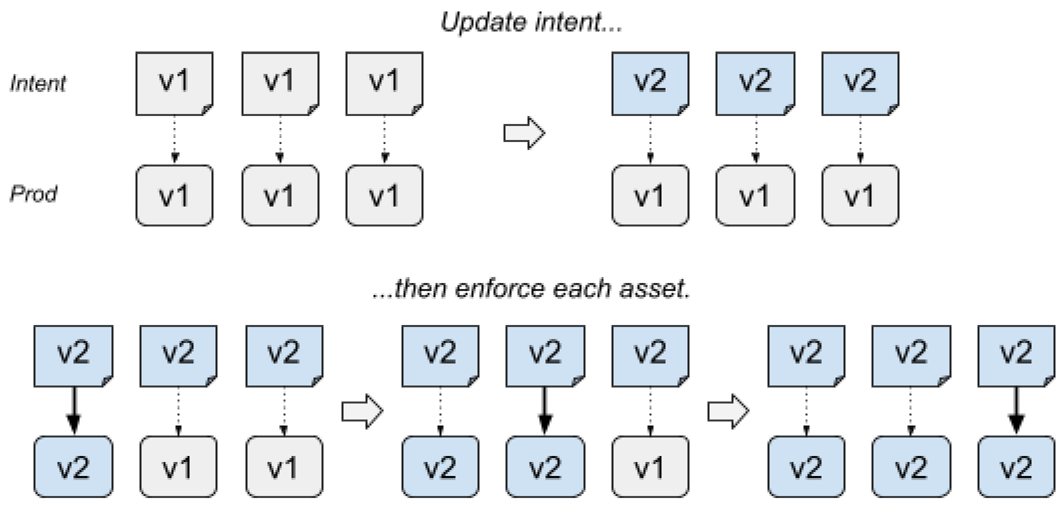

***Figure 11****: Actuation-based rollouts*

This approach has the advantage of being very simple: we just control how fast we advance along incarnations. It also acts as a catch-all mechanism: the specific changes to the SoT aren't really important because in practice, we carefully roll out the whole incarnation content.

However, this approach lacks flexibility. Consider the case in which multiple in-flight rollouts modify the same assets—for example, a week-long rollout of a new flag in parallel with a daily version update. This use case is not manageable with a pure actuation-based rollout; in practice, changes need to be batched.

As shown in Figure 12, *intent-based* rollouts update the *intent* of assets only when those particular assets are selected. The intent is then immediately enforced through Enforcer.

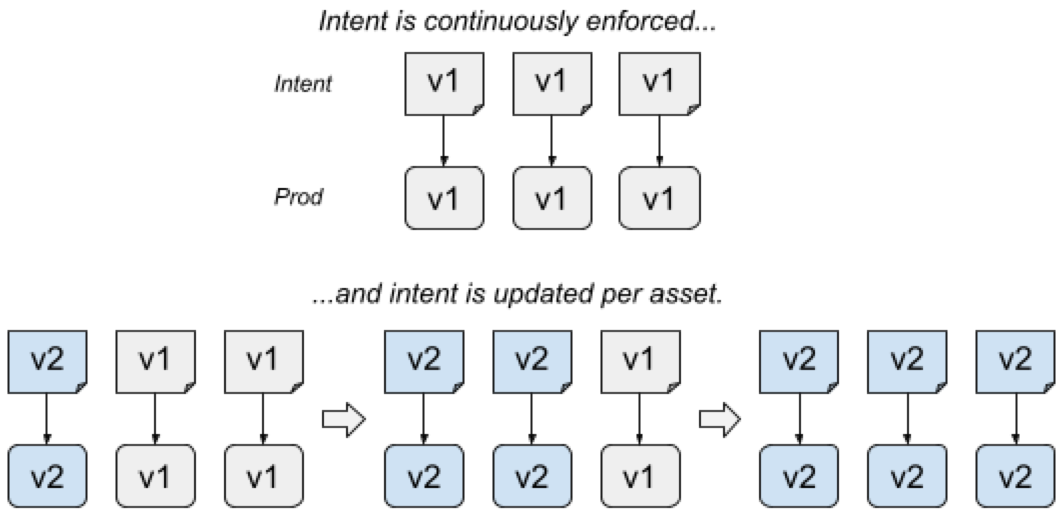
Intent is continuously enforced...
Intent
v1
v1
v1
Prod
v1
v1
v1
...and intent is updated per asset.
v2
v1
v1
v2
v1
v1
v2
v2
v1
v2
v2
v1
v2
v2
v2
v2
v2
v2

***Figure 12**: Intent-based rollouts*

Intent-based rollouts are more complex than actuation-based rollouts. They require the ability to modify the intent programmatically. In the Prodspec model, this requirement means relying on a source of truth that's programmatically editable. The source of truth should also allow sufficient granularity to match how assets are selected.

To apply this model to the Shakespeare service, we need a SoT that allows us to specify the binary version of the frontend for each cluster independently.

Intent-based rollouts allow for rollouts impacting the same assets running in parallel—something not possible with actuation-based rollouts. The only constraint is that those rollouts must modify distinct aspects of the assets— for example, one rollout might change the binary version, while another rollout updates the flags.

### Validate

Once the updates have been actuated, Annealing evaluates the impact to determine whether the rollout can continue, or if the rollout should be stopped or even rolled back. This step is automatic—we actively discourage manual validation of pushes beyond the very early stages.

We check the service health during rollouts at two levels:

- **Enforcer level:** Some Annealing plugins have built-in verification of asset-specific health. If a problem occurs, the push is signaled as failing.
- **Strategist level:** After we have selected and updated some assets, we verify the health of the service. We often introduce various waiting times to account for factors such as servers stabilizing after a push.

Health checking can include verifying the health beyond just the updated assets. In the Shakespeare service it might mean verifying the health of the frontend jobs after updating the database schema.

We analyze the monitoring data after the update to verify the health of the system. There are two broad categories of validation:

- **Absolute values:** We compare the monitoring to a configured value— for example, whether alerts are firing, or if errors are greater than 2%. This approach is robust as long as you can determine a baseline.
- **Statistical:** We compare the monitoring either against past values or against a control asset. While this approach tends to be more noisy, it requires almost no configuration and can work across many metrics. It can also catch anomalies beyond failure modes the service maintainer might have envisioned.

Automatic health assessment is a broad and complex topic with many subtleties we have learned and tuned over the years, and goes beyond the scope of this article.

## Integration with workflows

Our objective with Prodspec and Annealing is to switch more production management toward an intent-driven model. In theory, we could manage all production changes through clever use of intent. However, we explicitly do not have a goal of converting all of production to an intent-driven model. In some cases, more traditional workflows make more sense.

There is no well-defined line between when to use intent-based actuation versus traditional workflows, but we use some common considerations:

- **Size of the system:** Intent-based actuation is useful when many changes occur in parallel, as you can encode the rules in a more intuitive way. For small systems with low concurrency, workflows tend to be easier to understand.
- **Mental model:** Do you have reasons to care about the exact detailed steps and want to be able to fine tune them, or do you just need the right thing to happen, as it is a detail of your infrastructure? The latter calls for intent; the former calls for a workflow.
- **Complexity:** Does the change require a lot of odd steps—for example, a one-time run of a batch job, some user inputs, and so on? If so, a workflow might be a good fit. Are you instead looking for consistency instead of fine tuning? Intent will help.

A direct consequence is that intent-based enforcement and workflow-based enforcement must almost always be able to integrate with one another. We use two integration mechanisms.

The first mechanism lets a high-level workflow drive updates to the intent by changing the content of the sources of truth. For example, the creation of binaries for our servers are largely driven by workflows, and then Annealing and Prodspec deploy those binaries.

The other mechanism is at the other end of the stack: you can use a workflow engine to implement an Annealing plugin Push operation. For example, we update network switch firmwares using this model: a proprietary workflow system implements a lot of custom logic, then Annealing starts the workflows themselves.

Regardless of which model we use, we ensure the following:

- Only one automation system should be responsible for a given part of production. For example, a common request is to have one system responsible for turnups and another system responsible for ongoing updates. Our experience shows that this practice introduces many synchronization problems, especially on more complex assets.
- State keeping should be minimal: you should rely on the state of production and the desired end state. State of production often evolves for more reasons than expected, and the more state you keep, the more likely the automation will make the wrong choice. We are often tempted to simplify the design of some plugin or workflow by storing extra state, but doing so often makes the system more fragile and harder to debug. This is not a hard rule— sometimes it does make sense to keep extra state, but determining when this is the right choice can be difficult.

## Benefits of a control plane

Prodspec and Annealing act as a control plane between a service configuration and the infrastructure. The most obvious benefit of a centralized control plane is uniformity across how we manage production, but there are additional benefits, as well.

Prodspec provides structured and accurate information to the various tools interested in the model of a given service. Tools responsible for monitoring, analysis, and auditing, and even one-off scripts have access to the same authoritative data.

A centralized control plane also allows us to encode best practices. At the configuration level, we can easily detect whether a service is configured properly, even across the multiple infrastructure providers it is built on. This significantly reduces surprises at runtime, as you won't belatedly discover that some critical infrastructure should have been configured to support your frontend.

Best practices at the actuation and enforcement level also benefit from Annealing. In a workflow model, you typically have to tailor each workflow to the service. Annealing's continuous enforcement model allows us to specify commonalities between workflows. We can use plugins to split the generic logic for driving pushes from the granular details of the actuation. For example, rather than requiring every workflow to implement its own logic, a check plugin can automatically ensure that we resize frontends and backends in the correct order.

A unified control plane also empowers infrastructure providers. In many cases, providers can change some behavior or add extra safeties by simply updating the corresponding Annealing plugin, rather than dealing with many individual customers that each use a different mechanism to configure their service.

A unified control plane also provides safety across asset types. The workflows predating Annealing often focused on a single type— for example, we used one workflow to update the binary and one workflow to update the database schema, making synchronization difficult. With Annealing, we can encode the guarantees and safeties in plugins, especially in Check plugins. A Check plugin might verify that the version of the database schema remains compatible with the version of the running job, no matter what happens through multiple pushes and rollbacks.

Ultimately, Prodspec and Annealing allow us to reliably automate— or at least streamline— complex procedures like relocating a service. The custom workflows we used before relied more on humans than automation to detect problems. Instead, our control plane does the following:

- Fully models the service: we no longer have to guess about dependencies or other considerations.
- Generates the configuration: we no longer need to edit multiple heterogeneous configuration files in subtle ways.
- Pushes changes and ensures the system is current using continuous enforcement.
- Ensures that operations happen safely and in the correct order using safety checks.

## Lessons Learned

In the years since we shifted towards intent-based production management, we've learned a lot about what works and what doesn't. Here are a few of those lessons.

### Enforcement

Having some form of continuous enforcement is fundamental for an intent-based configuration. Simply modeling intent results in stale and missing information. We've seen this problem in workflows created for managing turnups only, which have a track record of breaking when they're needed and requiring substantial effort to repair the workflow rather than executing the turnup manually. Continuous enforcement guarantees that the intent is either correct or very quickly fixed when a change impacts the intent. This in turn encourages people to manage their production via updates to intent, and to have many other tools feed from the intent. These behaviors reinforce the quality of the intent, leading to a virtuous cycle.

An early implementation mistake was to have the Enforcer work in waves. It was first diffing all the assets of a partition, then triggering the push of changes that were ready to go. Another wave could start only when the pushes were finished. That model was simple to grasp and debug. For example, it made centralized decisions— such as the dependency solver— a lot simpler compared to our current implementation. However, it was unbearably slow. We want pushes to start within minutes, at most. In this early implementation of the Enforcer, long pushes were delaying other changes for hours.

## Modeling

Not everything in production fits well in an intent model. Data pipelines and batch jobs can be such cases. For example, it is possible to construct an intent representing batch jobs— an asset can indicate "this job must have run at least once in the last 24h"— while a plugin takes care of starting the job when the constraint is no longer valid. However, this setup is a bit awkward, and we haven't yet explored proper support for data pipelines and batch jobs through Prodspec and Annealing.

Managing turndowns— removing some infrastructure from production— is also tricky. Many of our users have requested "turndown-by-absence"— i.e., for the system to consider an asset that disappears from Prodspec to be an implicit signal that the corresponding infrastructure must be removed. This approach is fine in some limited cases, but we generally won't support it, as it poses too great a risk to production at large. The most common failure mode of configuration systems is returning partial content— for example, if an error occurs in the middle of parsing and the system only sends half the configuration. In these cases, turndown-by-absence would wreak havoc on production: imagine automation deleting your database by mistake!

To deal with turndowns, we require extra safeties before implementing the code to turn down a given asset type. Then, we require explicit signals for turndown: an extra addon on the assets to turndown. While this solution isn't perfect— managing the addons is troublesome— it has avoided numerous accidental turndowns.

### Adoption

The difficulty of adopting Prodspec and Annealing is probably the largest challenge we've faced. Switching to intent-based actuation requires a change in mindset, which takes time. At this point in our evolution, adoption is less of an issue, at least at the lower layers of the stack. However, the initial modeling of an existing service can be tough. This exercise often requires adapting complex sources of truth, which collectively may not contain all of the necessary information. Some production processes might need to adapt in order to move to intent-based actuation. For example, if you need to manually silence an alert during routine maintenance, you might need to instead make that alert more precise so you don't have to silence it the next time around.

We've found that splitting the onboarding helps. Services rarely use Prodspec and Annealing directly. Instead, they use integrators, which provide opinionated services models for specific service archetypes. Those integrators typically deal with multiple types of assets and infrastructure. For example, an integrator helping with pipelines takes care of configuring temporary storage, issuing notifications, running the compute when needed, and so on. This approach allows services to easily configure their infrastructure through a model that matches their use case. Then, those integrators provide the necessary logic to generate Prodspec, thus enabling Annealing to manage the service production.

"Invisible decisions" are another challenge to adoption. One common pattern we've observed is an unacknowledged gap between the requirements for a system built on continuous enforcement versus a system where humans deal with exceptional cases. Even when planning a move towards Annealing, it's easy to miss many decisions that operators perform manually. While sometimes this gap isn't an issue, it can be a showstopper when it's difficult to automate an intuitive manual step.

Retrying a failed push is a typical example of invisible decisions. Workflow systems rarely automatically re-run a particular step, and humans commonly handle retries— estimating whether the failed push is transient and if it's reasonable to re-attempt the push. Continuous enforcement is likely more often correct than humans. However, continuous enforcement also makes those decisions more frequently and can be harder to debug. That means that for a good end user experience, automation must be an order of magnitude better than humans.

## Next steps

Today, intent-based configuration and continuous enforcement are now largely accepted and adopted at Google. As we learn from our challenges and successes, our approach to intent-based actuation continues to evolve.

Understanding the limits of your targets for intent-based actuation and continuous enforcement is important when moving to such a system. The boundary of what should be included is not fixed, and we're always exploring this limit. In general, the aspects of production that fall into intent-based and continuous enforcement grow as people become familiar with these concepts, but this process is neither instant nor simple.

In our case, driving longer rollouts for large deployments over several days and weeks is still an area where we heavily iterate. Our modeling can't yet handle this use case well— how do you express the intent of the various versions that should be deployed over multiple clusters and time windows? Our users are not comfortable (yet) with continuous enforcement at such a level, and still pay close attention to aspects like specific versions. We are exploring various options in this space, including relaxing the continuous enforcement for high-level intent.

We are also investing in improving debuggability. Intent-based production management is intrinsically at a disadvantage in this area compared to traditional workflows. Users only look at our system when something goes wrong, and typically don't have a good starting point for their investigations. We're working to better represent the state of the system, and why and how decisions are made, in order to improve introspection when something goes wrong.

Authors:

Bio: Pierre Palatin is a Staff Software Engineer in Site Reliability Engineering (SRE) at Google. During his many years as a SRE for Gmail, he learned solid production hygiene and how to manage large scale systems. Building upon that experience, he co-created the control plane now responsible for a large chunk of Google infrastructure. Navigating between modeling and automation, he aims to keep the ever growing complexity of production manageable.
He also enjoys dabbling in chocolate making, learning about the science behind food and playing board games.

Betsy Beyer is a Technical Writer for Google in NYC specializing in Site Reliability Engineering (SRE). She co-authored Site Reliability Engineering: How Google Runs Production Systems (2016), The Site Reliability Workbook: Practical Ways to Implement SRE (2018), and Building Secure and Reliable Systems (2020). She has previously written documentation for Google datacenters and hardware operations teams. She holds degrees from Stanford and Tulane.