# Using Terraform/Terragrunt to Orchestrate Multiple, Regional Cloud Deployments

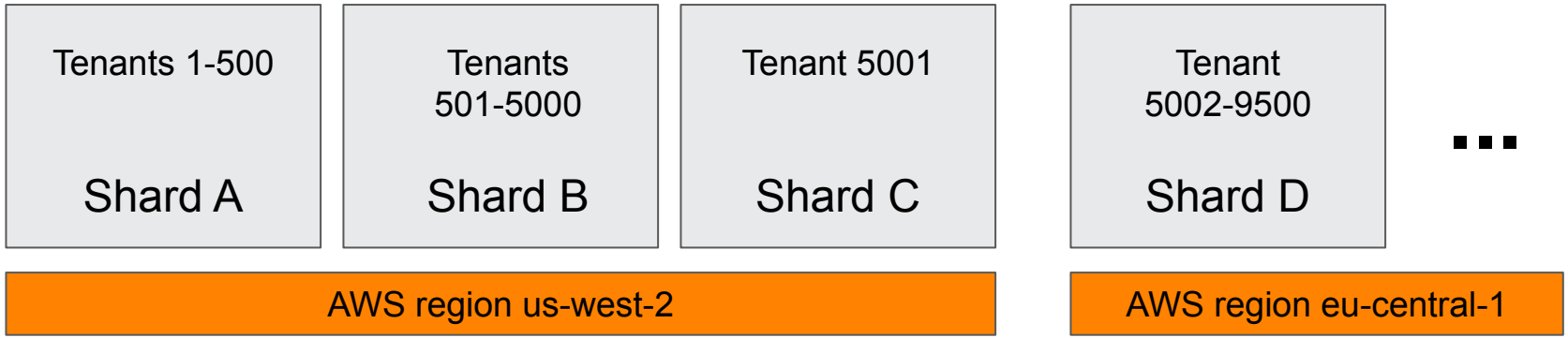
# What? Why?

# What problems did we need to solve?

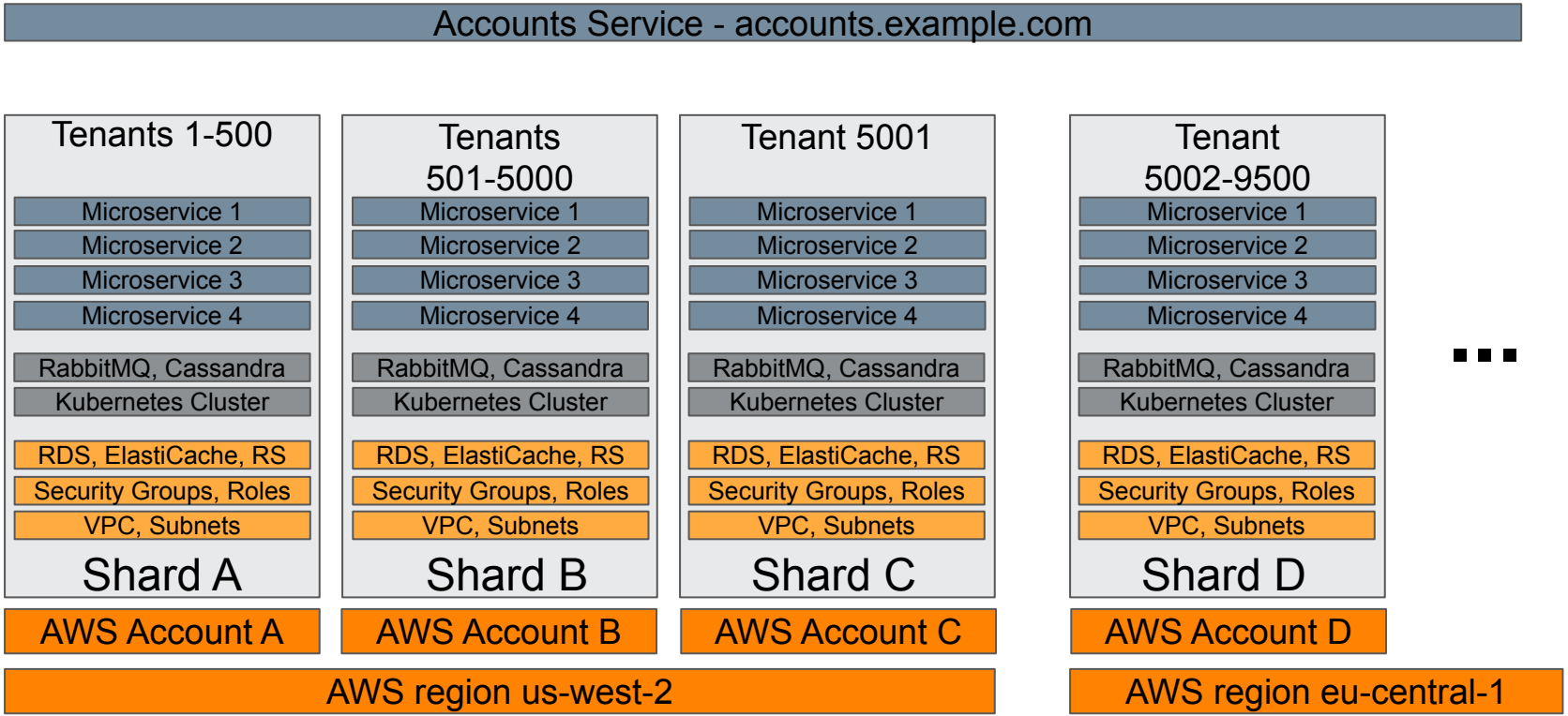
- Single-Region, Multi-AZ AWS deployment in USA
- What about non-US customers?
- What about scaling “vertical only” databases?

# What is Stack Sharding?

Stack sharding is deploying multiple, isolated, and complete instances of a multi-tenant application.



# Stack Sharding Example



# Stack Sharding Benefits

- Scalability - distribute customers across different shards reduces the load on an individual shard.
- Isolation
  - Some customers demand their own deployment of the application.
  - Isolating high-usage customers prevents them from becoming "noisy neighbors"
  - Outages in one shard generally will not affect other shards
- Location
  - satisfy in-country data sovereignty requirements
  - reduce network latency

# Stack Sharding Drawbacks

- Complexity
  - managing multiple shards creates additional operational overhead.
- Cost
  - right-sizing the shard and choosing the tenants is key to retaining the economies of scale inherent in multi-tenanted SaaS applications.
- Tenant management
  - Moving tenants between shards is complicated

# Issues with Current Deployment

- Resources were created manually (via AWS Console), with CloudFormation, with custom scripts, etc.
- Test, Stage, Prod stacks are each “snowflakes”
- How should a service team create a new AWS resource?
- How do we share “best practices” in each microservice?
- How to connect non-AWS resources: Cloudflare, DataDog, PagerDuty, etc.

# Infrastructure Requirements

- Standardize on “infrastructure as code” (IaC) toolset and design
- IaC deployment for AWS, **Kubernetes**, **Helm**, Cloudflare, DataDog, PagerDuty, etc.
- No more snowflake deployments - Test, Stage, Prod are virtually identical
- Can deploy “identical” instance to other AWS regions
- Fully automated DR restore in a new region

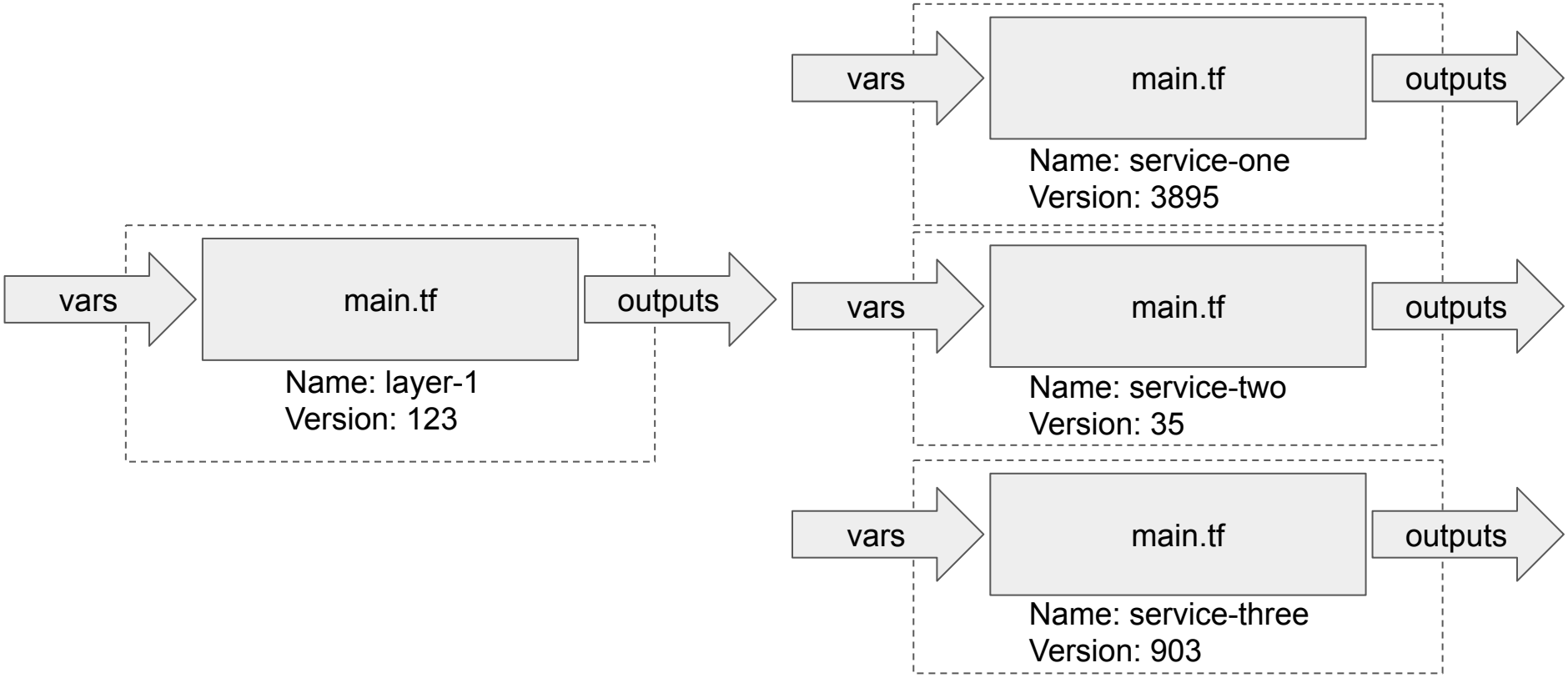
How?

# Terraform

- Infrastructure as Code created by HashiCorp
  - Allows us to define everything in code
  - Traceability, code reviews, reverts, CI pipelines, etc
- Declarative just like Kubernetes and Cloudformation
- Supports an ecosystem of Providers - AWS, Kubernetes, Helm, Cloudflare, PagerDuty, DataDog, etc.
- Lots of Hashicorp and community support that we get for free for provisioning new resources when made available from infrastructure vendors.

# Terraform Module

- HCL2 description of resources
- Inputs as variables
- Outputs as outputs
- Can be deployable “root” modules or “library” modules
- Source is Packaged:
  - Zipped
  - Versioned
  - Uploaded to S3
- Terraform apply creates resources and stores “state” encrypted in S3 (w/ DynamoDB lock)

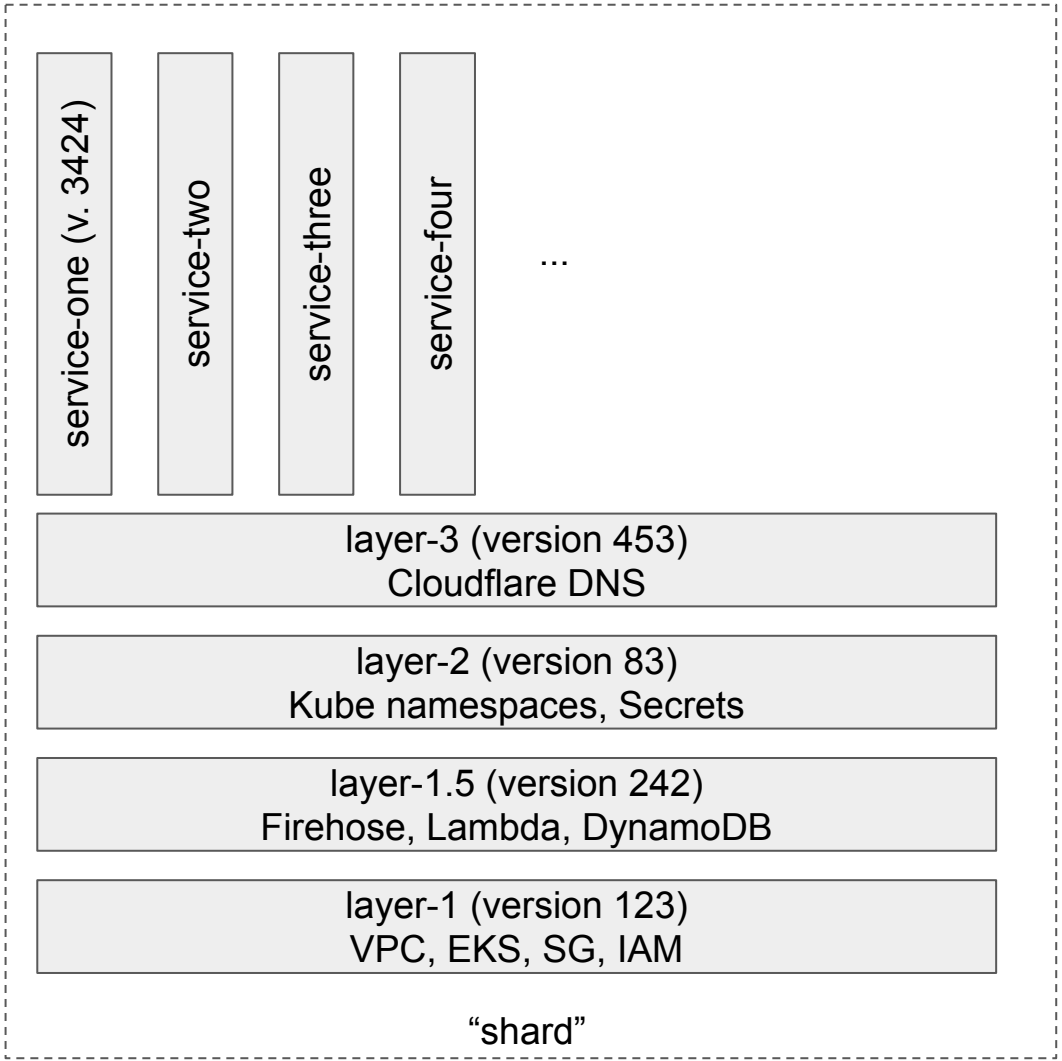

vars
main.tf
outputs
Name: layer-1
Version: 123
vars
main.tf
outputs
Name: service-one
Version: 3895
vars
main.tf
outputs
Name: service-two
Version: 35
vars
main.tf
outputs
Name: service-three
Version: 903

# How to connect all terraform modules?

- One option is to put everything into a single terraform deployment
  - Does not scale well
  - Tough to build pipelines
- Second option is writing Jenkins jobs to manage the dependency, deployment order, and configurations
  - Lots of copy/paste - for each environment
  - Quickly became unmanageable as we added more microservices
- This was a solved problem – Terragrunt!

# Terragrunt

- Terragrunt is the “glue”
- Connects module outputs to module inputs
- Versions modules
- Orchestrates apply/destroy based on dependency maps
- I think of it as “docker-compose for Terraform”
- Each “box” on the right has its own “terragrunt.hcl” file to describe version and inputs
- All shards are in `shard-manifest` git repo



# Customizations

- Use terragrunt “before_hook” for SOPS decryption
- Use Helm+Go templating to render configuration from yaml
- Wrote custom “state migrations” script
  - Import existing resources
  - Move resources from one layer or module to another

# Terraform “Library” Modules for Consistency

Examples

- RDS - common Multi-AZ, snapshot, maintenance windows, AWS Backup settings, monitoring, etc.
- Cloudflare - common WAF rules, rate-limiting, caching settings
- Helm Chart - common overrides, ingress, etc.

# Deployment Pipeline

# Pros / Cons

- Standard IaC platform for all things infrastructure and deployment
- No snowflakes - our test, staging, and production environments are practically identical
- Fully automated (and testable) DR in a backup region
- Allows us to meet scale, data, and compliance needs on a per-shard level. (Example: FedRAMP / GovCloud).

- Slow:
  - Full 90 layer “no-op” deploy takes about 30 minutes
  - Can optimize and target deploy layers to improve performance

Questions?