

clearpoint.

# Applying Terraform to Platform 2.0

—  
24th November 2022



**EROAD**





# Introduction

- Product domains, regions, and environments, and systems
- The platform account/subscription and repositories
- Default infrastructure
- Scope of responsibility
- Monorepo patterns
- Terraform + infra 2.0
- Shared Terraform base images
- Shared GitHub actions
- Pipeline workflows
- Secrets



# Product Domains and Environments

# Account/Subscription split by Product Domain



---

We treat AWS accounts and Azure subscriptions as analogous.

Rather than having a single account/subscription for all ERoad infrastructure, we've opted to split things by (Super) Product Domain.

Currently, there are:

- AWS: Platform\*, Core
- Azure: Platform\*, Backoffice

Splitting infrastructure across accounts/subscriptions allows us to isolate product domains specific services, access, and resource limits

# Environment classes

Environment classes are simply a way to think about everything that's not production, and then everything that is production.

## Environment classes are:

- Nonprod
- Prod

## Within the environment classes, we have regions:

- Current regions in Nonprod:
  - APAC
  - NA (optional)
- Current regions in Prod:
  - APAC (optional)
  - NA (optional)



# Environments

**Within the environment classes, we have environments:**

- Nonprod
  - dev (APAC)
  - test (APAC)
  - apacpp (APAC)
  - napp (NA - optional)
- Prod
  - apac (APAC - optional)
  - na (NA - optional)



# Systems within a product domain

Because of the breadth of some of the (super) product domains, we support the notion of one or more **Systems** within a product domain.

You can still have all of your product domain within a single monorepo, or you might decide to split your product domains by systems. For example:

**Product domain:** Drivers

**Systems:**

- Operations
- Integrations
- Metadata



# Platform Accounts and Repositories





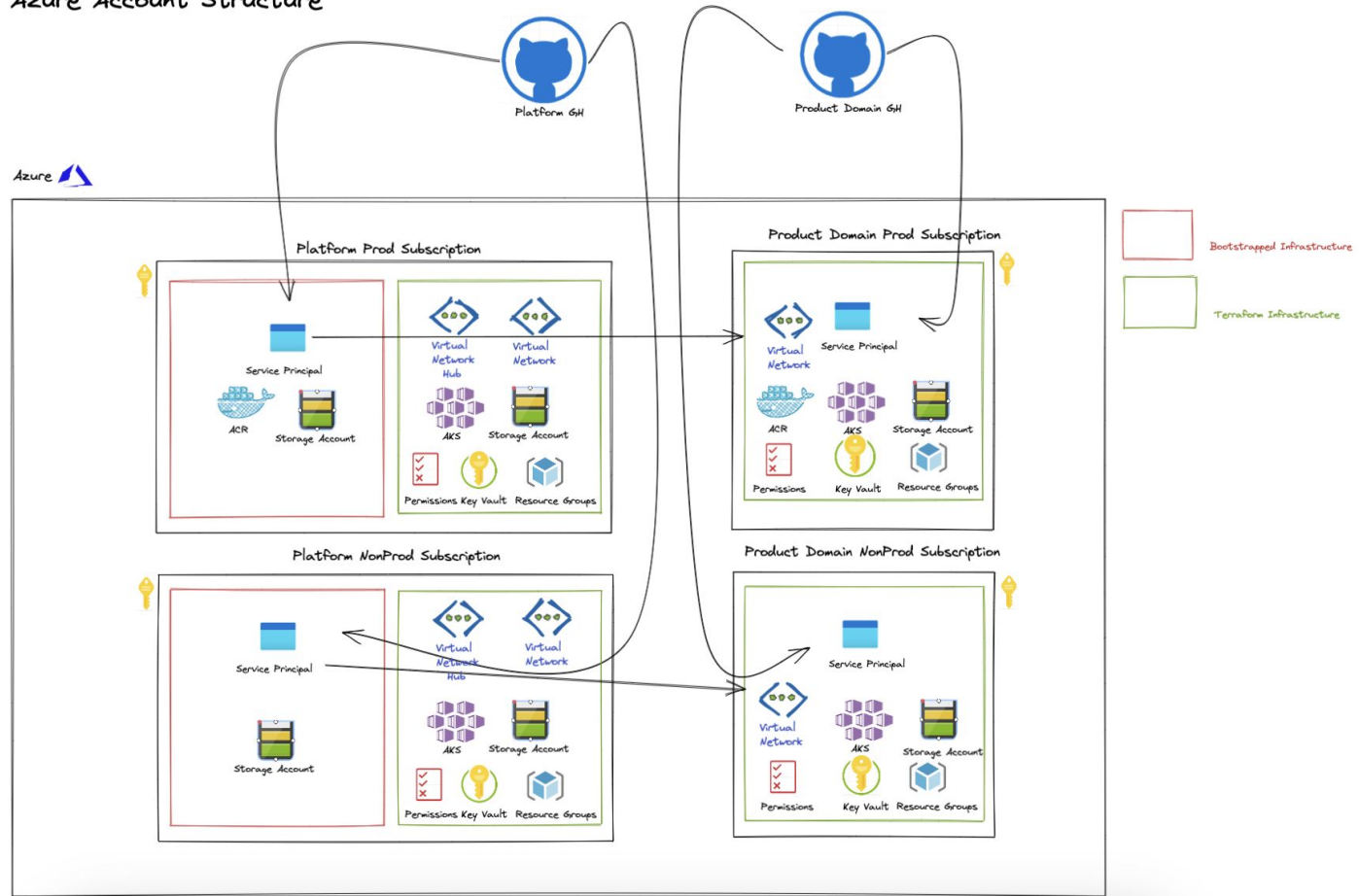
# Platform account/ subscription

The Platform account/subscription is used by the platform team for:

- Managing common infrastructure across all product domains
  - Network hubs, VPCs, VPNs, and related resources
- Shared container registries (for shared containers)
- State storage for product domain default infrastructure

Access is restricted because of the sensitive and broad nature of what's being stored.

# Azure Account Structure





# Platform team repositories

The platform team has a couple of key repos specifically for bootstrapping new product domains, and granting repositories access to these product domains.

These repositories are:

- platform-resources (AWS based product domains)
- platform-resources-azure (Azure based product domains)

And of course, the platform team uses Terraform within these repositories to manage all of the product teams repos, accounts/subscriptions, and related access.



# Default Infrastructure

# Default infrastructure

Each product domain gets default infrastructure created by the platform team (using terraform).

For AWS this includes:

- public/private subnets (a/b/c) + associated transit gateways, DNS zones, TLS certs
- S3 bucket, dynamoDB table, and KMS keys (for repo secret encryption)
- IAM roles for CICD pipelines, ECR + policies
- Datadog API keys

For Azure this includes:

- Spoke vnets, DNS zones, TLS certs
- ResourceGroups, StorageAccounts, KeyVaults + encryption keys,
- ServicePrincipals (federated with GHActions), Container registry (ACR),
- Datadog API keys



# Default optional infrastructure

In addition to the required default infrastructure, there are some additional optional resources that can also be created if you require them:

## **Kubernetes (EKS or AKS)**

Per environment class, and region

## **Regions (APAC/NA)**

Dev and test in nonprod are typically APAC

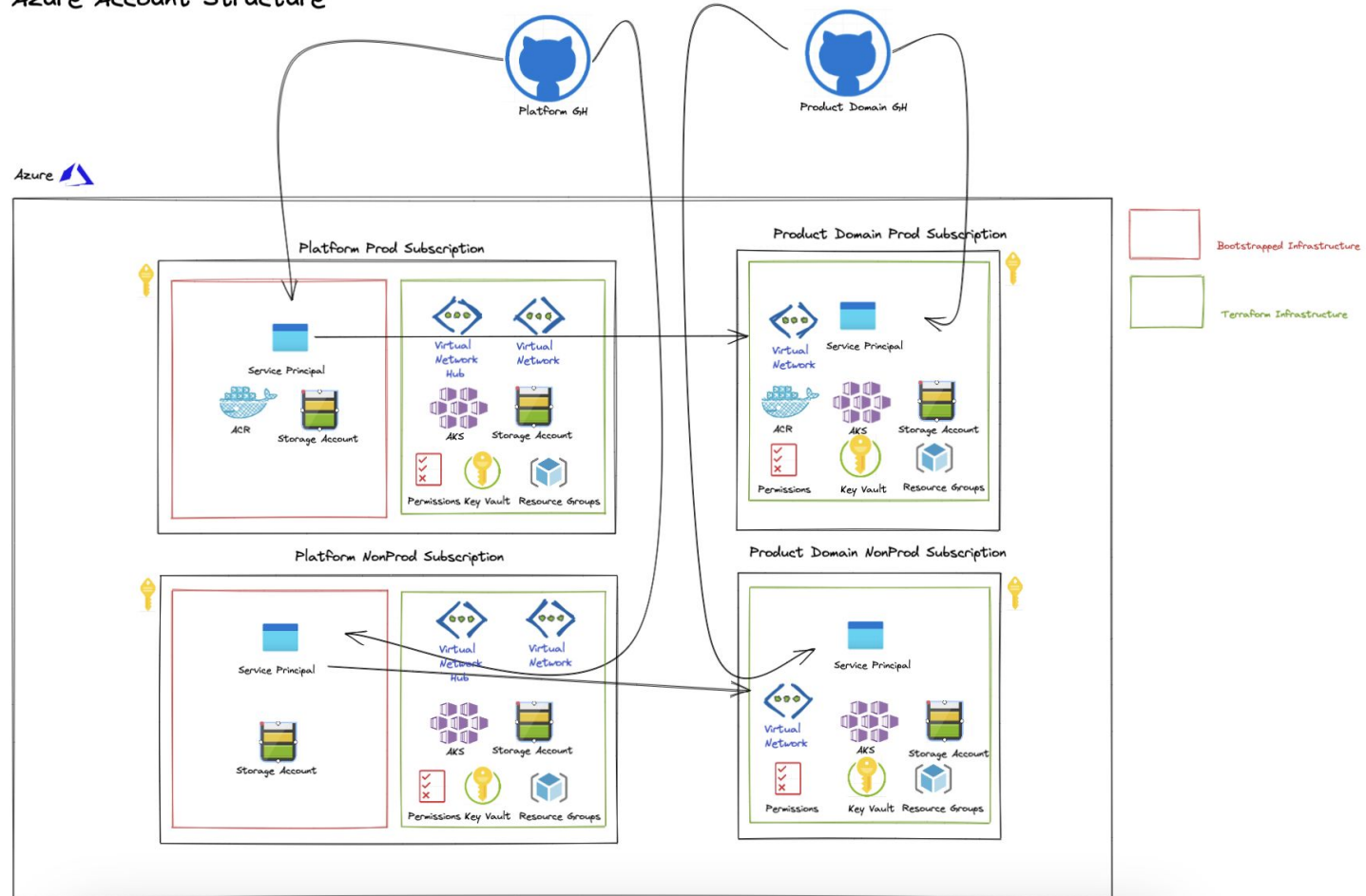
## **IPSec VPN connections (Azure only at this point)**

For access to heritage Azure infrastructure/networks

## **Other resources**

Of course, the platform team can add additional shared resources if required!

# Azure Account Structure





# Scope of Responsibility





# Platform team

Platform teams look after shared default infrastructure:

- VPCs/VNets
- EKS/AKS
- Network hubs
- State stores
- Default keyvaults
- TLS certs
- Access control
- IAM roles and ServicePrincipals

# Product squads

Product squads look after their own infrastructure such as:

- Application databases (e.g. SQL, Cosmos, Redis, etc.)
- Additional network/subnet firewall rules
- Lambdas/Function apps
- Product domain/application specific secrets
- Datadog monitors



Any questions so far?



# Monorepo Patterns



# Concept: Monorepos or Manyrepos?

When we say Monorepo, we mean that we combine all of our infrastructure code (Terraform) with our application code (Java/C#/etc.), our tests (unit, integration, e2e), our CI/CD pipeline yaml, our API specs, and any documentation and tooling.

The question is where we draw the line between:

- a single monorepo for the entire organisation, or
- individual repos for each deployable service.



# Monorepo recommendations

We recommend that you start with a repo per product domain. In this way, you can establish common patterns for your product domain in a single place and leverage all of the benefits of ERoad CICD.

If you decide to have more than one repository for your product domain, you'll need to take advantage of **Systems**.



# Terraform + Infra 2.0



# Terraform using infra 2.0 patterns




**Given you have a repository, to create infrastructure with Terraform you will need:**

- Terraform code
- Pipeline
- Terraform binary (pinned to your specific version)
- A convention to name state keys to avoid clashing with other terraform state
- Access (either IAM role - AWS ,or ServicePrincipal - Azure)
- State storage, and access to the state

**You might also need:**

- Gated production deployments (plan vs apply)
- A secure mechanism for secret storage/retrieval





# Terraform using infra 2.0 patterns

## You bring:

- Terraform code
- Pipeline
  - You provide, but using the shared GH actions to make it easier

## Everything else is provided by:

- Shared Terraform base images
- Shared GitHub Actions
- Default Infrastructure patterns



# Shared Terraform Images

# What do we need?



---

**Terraform binary (a specific version, not just latest)**

Decided when you select the shared TF base image

**A convention to name state keys to avoid clashing with other terraform state**

Provided as part of shared TF base image

# Shared Terraform base images and modules

The shared Terraform image provides a specific version of the Terraform binary, and access to some shared modules (e.g. Datadog monitors).

Out of the box, the name and location of the state storage is selected for you, although you can override this if you wish to handle your own key names.

## AWS:

- `key=${REPO_NAME}-${SYSTEM_NAME}-${ENVIRONMENT_NAME}.tfstate`
- `bucket=tfstate-${AWS_ACCOUNT_ID}`
- `region=ap-southeast-2`
- `encrypt=true`
- `dynamodb_table=terraform-locks`

## Azure:

- `key=${REPO_NAME}-${SYSTEM_NAME}-${ENVIRONMENT_NAME}.tfstate`
- `container_name=tfstate`
- `resource_group_name=${ENVIRONMENT_NAME}-${locationTLA}-${PRODUCT_DOMAIN:0:13}-rg`
- `storage_account_name=${ENVIRONMENT_NAME}${locationTLA}${PRODUCT_DOMAIN:0:13}sa`
- `subscription_id=${SUBSCRIPTION_ID}`

# Using the shared TF image

```
src > infra-aws > Dockerfile > ...
```

```
1  # Base our image on the base ERoad terraform image
2  FROM 030776936654.dkr.ecr.ap-southeast-2.amazonaws.com/platform/terraform-aws:1.1.9-4
3
4  # Set our current directory as /tf
5  WORKDIR /tf
6
7  # Copy our terraform into the tf folder:
8  COPY . .
9  |
```



# Shared GitHub Actions

# What do we need?



## Access (either IAM role - AWS, or ServicePrincipal - Azure)

Provided automatically via *init-deploy-\** step, and *cloud\_credentials: true* flag

## State storage, and access to the state

Provided as part of default infra

## Gated production deployments (plan vs apply)

Provided as part of a combination of shared TF base image + GH functionality



# Product domain and system selection

---

Our permissions and state storage location are based on product domain, environment class, and system name.

We specify all of these as part of our init-build-\* and init-deploy-\* steps in GH Actions.



# Init-build – selecting product domain and system

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: initialise for build
        uses: eroad/github-actions-pipelines/actions/dist/init-build-aws@v4.0
        with:
          product_domain: drivers
          system_name: integrations
```

# Init-deploy – selecting system and environment

```
jobs:
  deploy:
    name: Deploy to ${ inputs.environment || github.event.inputs.environment }
    runs-on: ubuntu-latest
    steps:
      - name: initialise for deploy to aws
        uses: eroad/github-actions-pipelines/actions/dist/init-deploy-aws@v4.0
        with:
          environment: ${ inputs.environment || github.event.inputs.environment }
          system_name: integrations
```



# Pipeline permissions

To deploy infrastructure, we need our terraform to run with elevated privileges.

Luckily, our shared GH actions provides this to us with a simple flag when we run our terraform container:

***cloud\_credentials: true***

This will provide either the AWS or Azure environment variables for terraform so that it executes using the specific IAM role/ServicePrincipal that has access to your product domain and environment you're deploying to.

# Running terraform with cloud\_credentials

```
- name: run aws infra
  uses: eroad/github-actions-pipelines/actions/dist/docker-run@v4.0
  with:
    image_names: example-infra-aws
    cloud_credentials: true
    args: APPLY_TERRAFORM=true
```



# IAM user/service principal access



IAM roles and ServicePrincipals are provided as part of the default infrastructure.

They are created and can be used by your pipelines to create your own infrastructure on top of the defaults.

In AWS, the IAM role is named “GitHubActions<product-domain>Role”, in Azure, the ServicePrincipal is named “<environment-class>-<product-domain>-sp”.

They are created per environment class; so you have a nonprod one, and a prod one.

This helps to ensure a clear separation of permission between nonprod and prod.

These identities allow you to run your infrastructure containers to create your own infrastructure (among other things like deploying to Kubernetes).

# A note about Azure ServicePrincipals

We use federation to permit GitHub actions to obtain a short lived token for our identities.

In the case of Azure ServicePrincipals, there is a limitation in that the subject cannot be defined with wildcards as they can with AWS IAM roles.

As such, the default subjects defined are:

- repo:eroad/<repo-name>:pull\_request
- repo:eroad/<repo-name>:ref:refs/heads/main
- repo:eroad/<repo-name>:environment:apac
- repo:eroad/<repo-name>:environment:na

This means that unless you have the platform team define additional subjects for your use case, you need to use “main” for your main branch, and must have pull requests into this main branch.

In addition, stage gates can only be used for deployments to **apac** or **na** (in the prod environment class).



# Terraform using infra 2.0 patterns – a recap

## You bring:

- Terraform code
- Pipeline
  - You provide, but using the shared GH actions to make it easier

## Everything else is provided by:

- Shared Terraform base images
- Shared GitHub Actions
- Default Infrastructure patterns



# Pipeline Workflows



# Running Plan vs Apply



Our base terraform images provide us with a simple way to instruct the terraform to run either **Plan**, or **Apply**.

You can simply specify an environment variable in your github actions as follows:

***args: APPLY\_TERRAFORM=true***

You can do more complex logic if you wish:

***args: APPLY\_TERRAFORM=***  
***\${{ endsWith(github.ref, '/main') }}***

## 2.0 CI/CD and development workflows

There are lots of options, but we recommend:

### For branches:

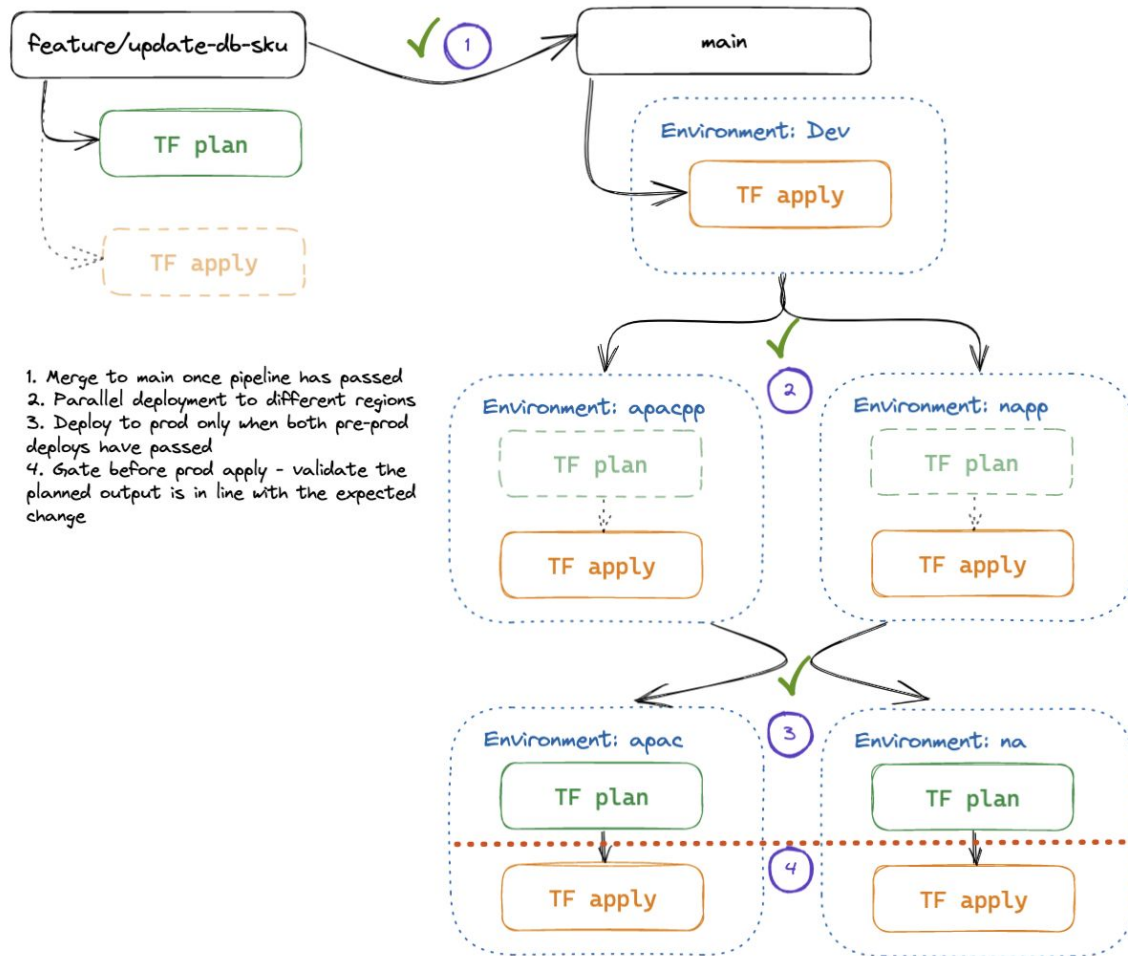
- Run plan only, inspect the planned output before merging
- In some circumstances, you might need to apply on a branch, but be careful

### For each Nonprod env:

- Plan, then Apply (or, just Apply)

### For each Prod env:

- Plan, then create a manual inspection gate
- Only after inspecting, validating, and understanding the changes, approve
- Apply
- Inspect the Apply step to ensure it did what was expected



1. Merge to main once pipeline has passed
2. Parallel deployment to different regions
3. Deploy to prod only when both pre-prod deploys have passed
4. Gate before prod apply - validate the planned output is in line with the expected change



Any questions?



# Secrets



# Different types of secrets

Broadly speaking, there are two different types sources of secrets.

The first type are secrets that our applications need to use to access cloud resources. These are things like database connection strings, storage account access keys, or kafka connection strings. Ideally, we'd use managed identities but we can also use terraform to retrieve and store secrets in SSM or KeyVaults.

We can then easily consume these by our applications (or terraform) in our GitHub pipelines.

# Adding a secret to SSM

```
resource "random_password" "this" {  
  length   = 40  
  lower    = true  
  number   = true  
  special  = false  
  upper    = true  
}  
  
resource "aws_ssm_parameter" "example_ssm_param" {  
  name     = "${var.environment_name}/${var.product_domain}/pipelines-example/password"  
  type     = "SecureString"  
  value    = random_password.this.result  
}
```

# Adding a secret to SSM

```
locals {  
  # This refers to the keyvault that is created as part of our default infrastructure  
  default_keyvault_name = "${var.environment}${var.location_short}${var.product_domain}kv"  
}  
  
# Create a reference to the keyvault that is created as part of our default infrastructure:  
data "azurerm_key_vault" "default" {  
  name           = local.default_keyvault_name  
  resource_group_name = "${var.environment}-${var.location_short}-${var.product_domain}-rg"  
}  
  
resource "random_password" "this" {  
  length   = 40  
  lower    = true  
  number   = true  
  special  = false  
  upper    = true  
}  
  
# Create an entry in the default keyvault that stores a randomly generated password:  
resource "azurerm_key_vault_secret" "tablestorageconnection" {  
  name       = "example-secret"  
  value      = random_password.this.result  
  key_vault_id = data.azurerm_key_vault.default.id  
}
```





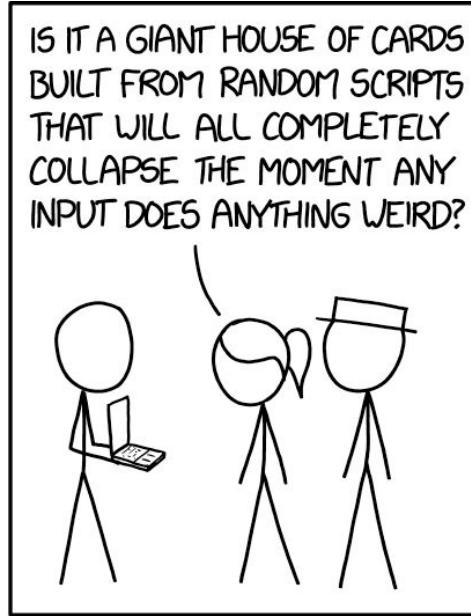
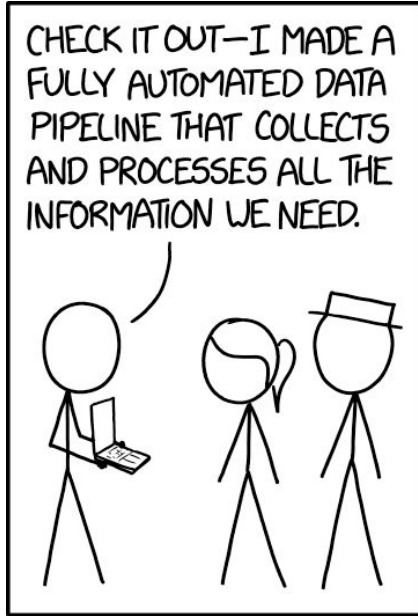
# The other type of secrets

The second type of secrets are those that are not available as outputs of infrastructure and so we cannot get them at runtime using Terraform.

These are things like client ID/client secret pairs for external APIs, or username/password pairs for legacy systems. For these, we don't want to be manually handle them/put them into SSM or keyvaults. Instead, we have a mechanism to store these as encrypted values in your code repo.

The GitHub Actions CLI we've created allows you to securely encrypt secrets for your product domain so they can be utilised by your applications/infrastructure. For AWS this uses a KMS key, and for Azure this uses a KeyVault key.

Engineers are able to encrypt/decrypt in the dev environment. All other environments are encrypt only for engineers. The pipeline IAM Role/ServicePrincipal can decrypt at deploy time.



Credit: <https://xkcd.com/2054/>

clearpoint.

| Thank you.



# Lab Repo

## Pre requisites:

- TF binary
- Cloned repo
- Sandbox AWS or Azure accounts + access
- Run script from repo to validate presence of TF binary, and access to repos

## Lab steps:

- ☒ TF basics (backend, vars, locals, providers, resources)
- ☒ Modules and outputs
- Environment specific variables & naming
- "Data sources"
- Conditionals, ternary, for\_each, count. Example would be "enabled" bool flag to switch on/off certain behaviour

~~○ Resource "depends\_on"~~

~~○ Renaming, and the "moved" block~~

## Platform v2 specific:

- Provisioning a DB for your app (Might be too slow)
- Provisioning a message queue for your app (Not attached to VPC/VNet)
- Lambda/Function app
- How to read the v2 metadata for your default infra (subnet-ids, vpc-ids etc)
- Using the shared TF modules (does EROAD platform provide shared modules)?