Platform Automation Pattern

Last Updated: January 1, 0001 |

Abstract

This document describes a pattern for platform automation pipelines that have been implemented for several customers that will reduce overall execution time of the platform, makes failures easier to find and restart, simplifies how products are configured, and works across all IaaS.

What problems are we trying to solve?

The motivation for the pattern came during an engagement where a platform team of less than 5 people were charged with maintaining approximately 15 products (tiles) for 35 foundations across 3 IaaS.

The initial state of the pipeline was a typical flow where each product would go through the Upload/Stage/Configure steps and an Apply Change would be executed. This Apply Change would be for all products and would execute all errands for the products. If all went well (it rarely did), the Apply Change would execute for about 20 hours. When the Apply Change failed, the main culprit would be something preventing the errands from executing and the solution would be to “just run it again”. Running it again meant starting from the beginning where the full Apply Change is executed and errands that had already been executed are ran again. We called this “paying the PAS and MySql tax” because it would go through those products and errands even when they had previously been completed.

The pattern was created to address this situation by implementing the following:

- Give BOSH a better chance of success by limiting the Apply Change to a specific product
- Allow for errands to be restarted at the point of failure
- Allow for errands for a product to be executed at the same time as the errands from a different product

In addition to the implementation specifics above, some other conventions were implemented in order to provide a simple pipeline that is clearly understood by the platform engineers.

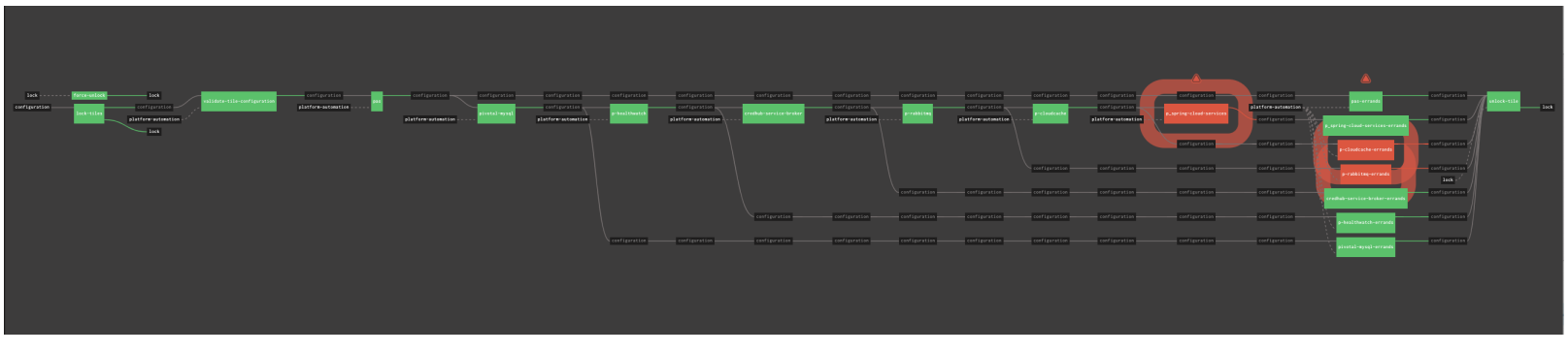
- Utilize a set of tasks for each product that are exactly the same between products
- Provide the ability to define common variables that would be used across all foundations
- Provide the ability to define only those variables that are different from the default values defined in the product
- Provide the ability to override common variables for a specific foundation
- Provide a workflow that endeavors to catch configuration errors before ever running the pipeline
- Provide a promotion path that moves changes along through a well-defined process

Starting at the end

Let’s take a look at the final pipeline and explain what it is doing. We will dig deeper into how we approach the development of the pipeline later.

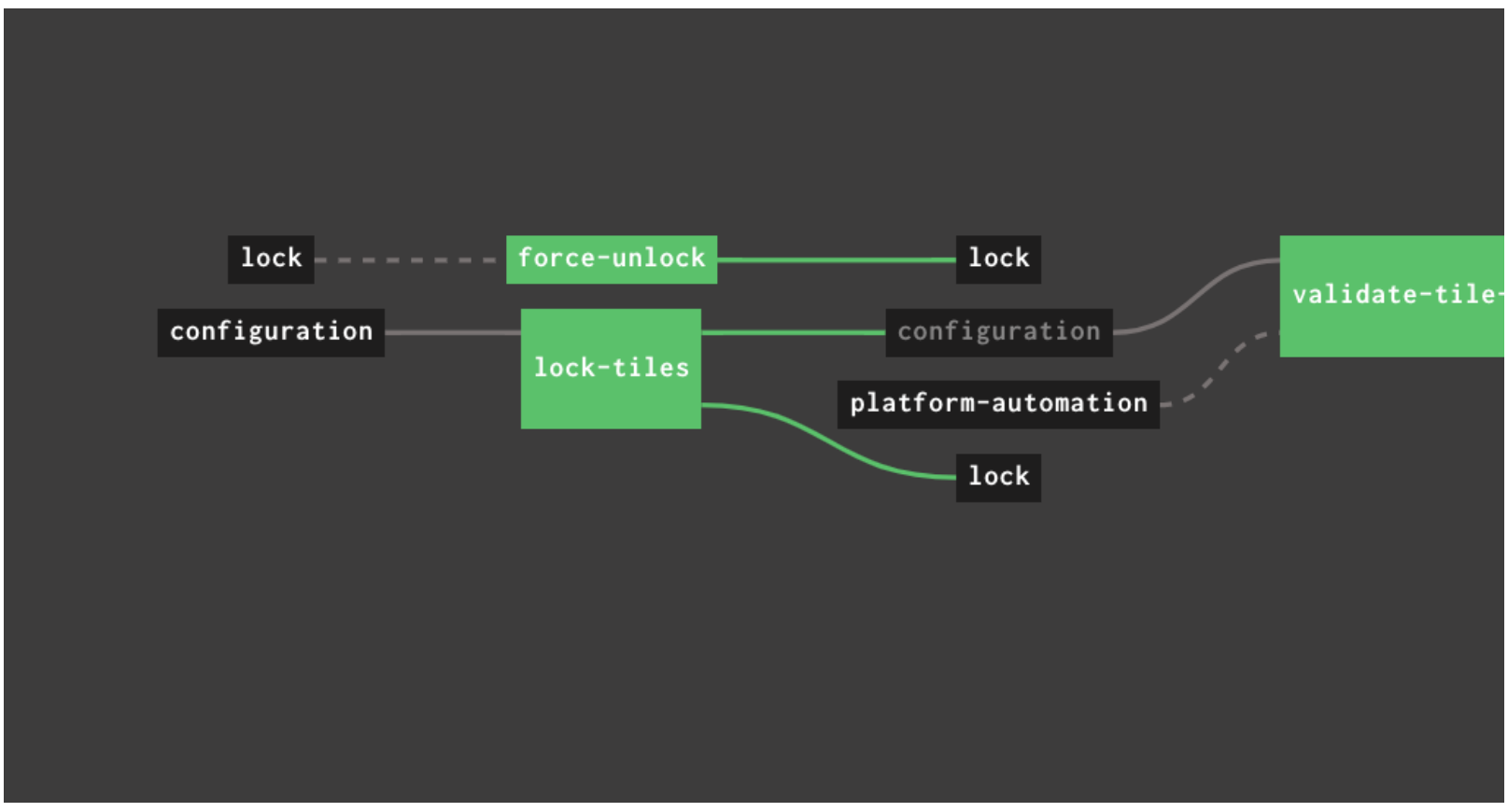
Since everything doesn’t always run as expected, let’s look at a pipeline that has failures to explain the functionality.

This is a pipeline executing against vSphere with NSX-T and deploying tiles for PAS, MySQL, Healthwatch, Credhub Service Broker, Rabbit, Cloud Cache (Gemfire), and Spring Cloud Services.



Of course the image is too small, so let’s zoom in from left to right.

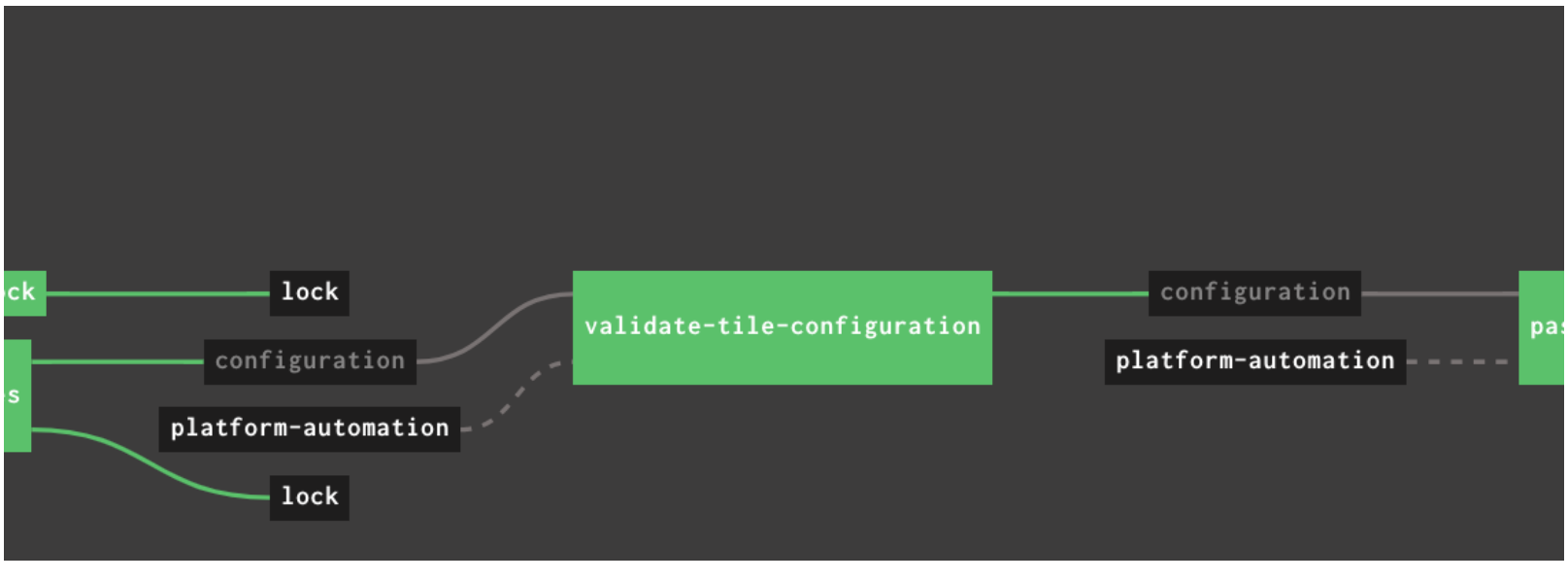
Locks



We first implement a locking mechanism to ensure that a single instance of the pipeline is executing at any given time. We will talk later about how the pipeline is triggered, but if the pipeline was allowed to execute multiple instances at the same time, you would receive the error stating that only a single apply change can be running.

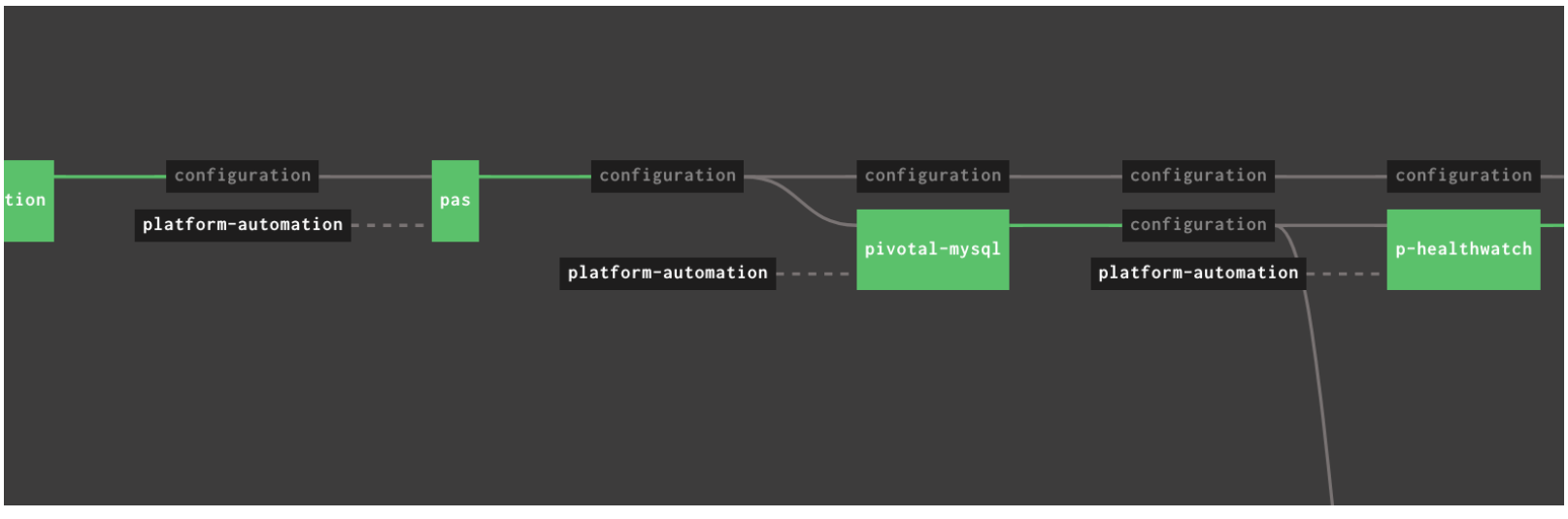
The key about the lock is that the only time the lock is released is when the entire pipeline successfully completes. This is intentional as we want to “lock” the pipeline until the problem has been investigated and resolved. However, if the solution is to modify some configuration and start the pipeline from the beginning, there is a “force-unlock” job that would be manually executed to release the lock and allow the pipeline to execute.

Validate-tile-configuration



The first job in the pipeline is to validate that all of the secrets that will be required are available in the secret store. In this case, we are using credhub. The details of where secrets are defined will be described in a later section. If any secrets are missing from credhub, this job will fail.

The Products



We now begin processing each product or tile. The pattern is the same for each product, with only a small adjustment for NSX-T.

The process is:

- Process Secrets (again using credhub interpolate)
- Download and Stage the tile and the stemcell
- Configure the product
- Perform an apply change specifically for this product

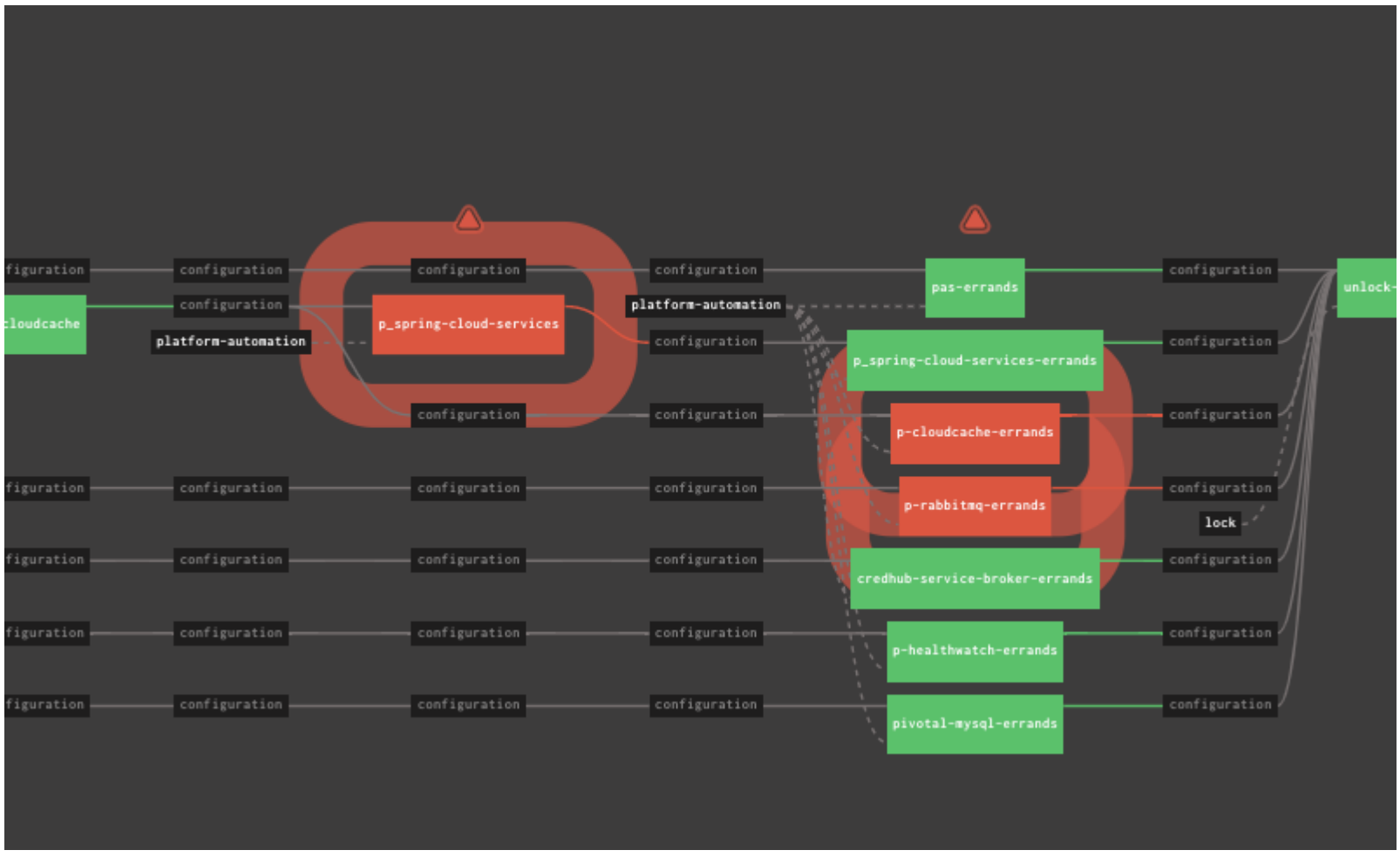
```
get: configuration

get: platform-automation-image

get: platform-automation-tasks

task: credhub-interpolate

task: download-stage-tile-stemcell

task: configure
```

```
task: apply-product-changes
```

The key to this is that all post-deploy errands are disabled.

The process is a combination of platform automation supplied tasks and custom tasks. The custom tasks are used for performance enhancements by not downloading tiles or stemcells unless they are required.

The Errands



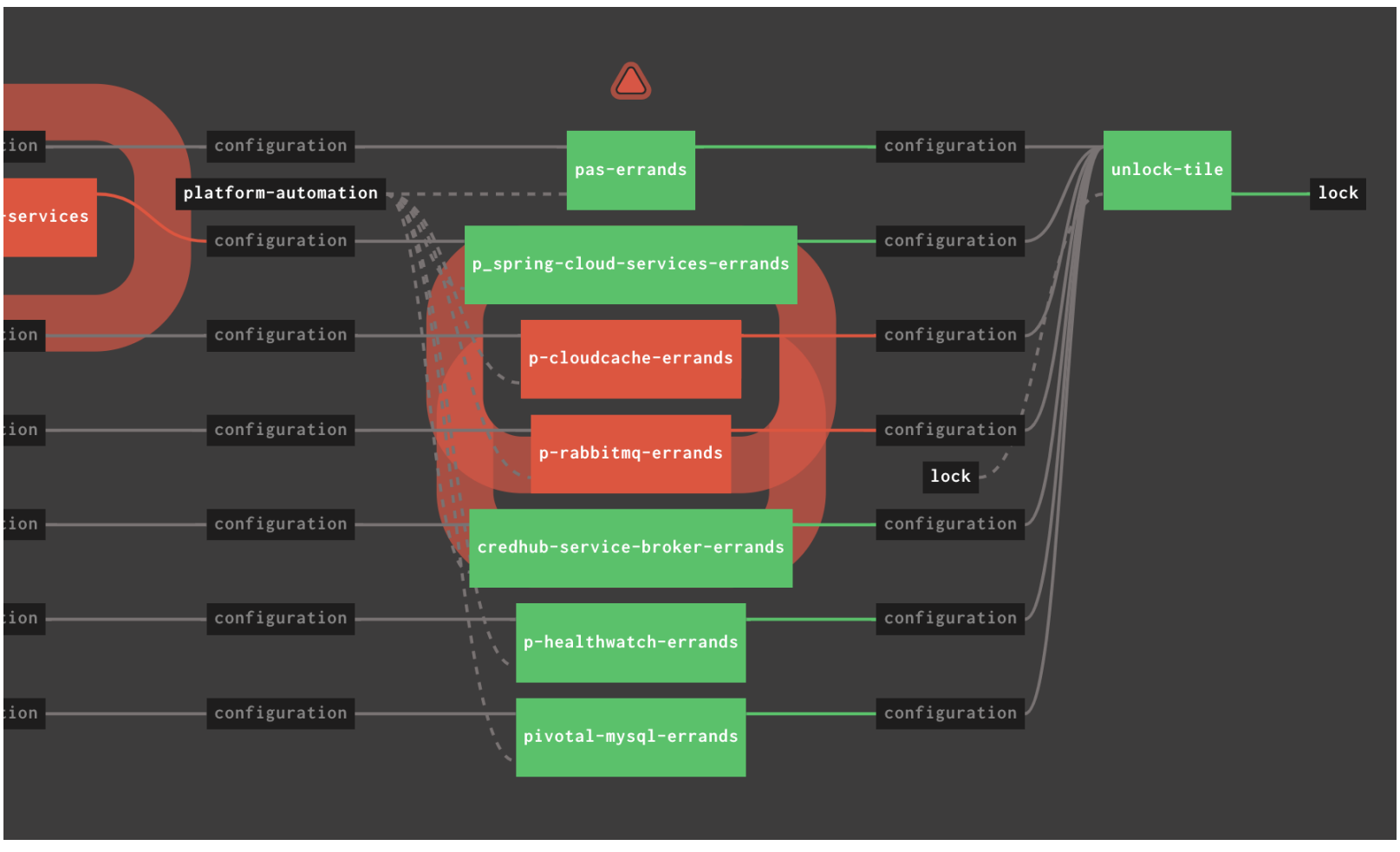
When the product apply change is complete, two tasks are triggered…

- The processing of the next product
- The errands for the product that just completed

Although opsman can only do a single apply change at a time … multiple errands can be executed simultaneously. The errands are executed by bosh and not by the opsman apply change process. Each errand is executed as a task within the job so that each can have the attempts specified and if there is a failure, the other errands do not have to be executed again. This provides the additional benefit of being able to easily determine exactly what is happening in the pipeline.

```
get: configuration

get: platform-automation-image

get: platform-automation-tasks

task: credhub-interpolate

1 2 3 4 5
task: register-broker

1 2 3 4 5
task: smoke-tests

1 2 3 4 5
task: upgrade-all-service-instances
```

The Unlock



After all errands have successfully been executed, the lock is released and the pipeline is finished.

Pipeline Structure

The pipelines are obviously designed to keep the deployed products that comprise a foundation up-to-date. Also included in the list of products that comprise the foundation are opsman and the bosh director. However, one may choose to provide a single pipeline that manages all products or a separate pipeline that maintains opsman/director from the pipeline that maintains all of the tiles. This decision is largely a personal preference to how the operator would like to maintain the pipeline.

The pipeline will *create* a foundation as well as maintain an existing foundation. Therefore, only a single pipeline is needed where in the past there was often a pipeline to “pour” a new foundation and a pipeline to “update” a foundation.

Opsman/Director

Let’s first look at the parts of the pipleine used to maintain the opsman / director.

- Resource Types
  - Pivnet
  - Any S3 compatible storage
- Resources
  - platform-automation
    - Reference to the platform-automation resource from pivnet that contains the docker image used to execute the tasks as well as custom tasks to perform the processing
  - installation
    - Reference to the S3 compatible storage which will be used to store the installation.zip file when opsman installation is exported
  - director-configuration
    - Reference to the git repository that contains the opsman/director configuration
    - Will typically define specific paths in the repository that are related to the opsman/director
  - state
    - Reference to the state file used to track the current status of opsman (this can be in the configuration git repository - but will have a separate concourse resource definition)
  - lock
    - Reference to a git repository to hold locks used during the pipeline (this can also be in the configuration git repository - but will have a separate concourse resource definition)
- Jobs
  - force-unlock
    - Used to force the lock to be released and would be manually executed on-demand
  - lock-director
    - Starting point and triggered by changes to the director-configuration resource
    - Obtains the lock for this foundation so no other instances of this pipeline or the product pipeline will execute at the same time
  - validation-director-configuration
    - Checks that all required secrets are available in the secret manager used (credhub / vault)
    - No need to continue if the pipeline is destined to fail because of missing secrets
  - install-opsman
    - If the state file is empty, this job will not do anything
    - Download opsman image
    - Create new opsman image
    - Commit the information about the opsman VM to the state file
    - Configure opsman authentication (basic / LDAP / SAML)
    - Configure Director
    - Apply Director Changes
  - export-installation
    - Export the opsman installation and store the installation.zip file in the defined resource
  - upgrade-opsman

            - If the requested version of opsman/director is the currently installed version, this job will not do anything
            - Download opsman image
            - Create new opsman image
            - Import installation.zip file from export-installation job
            - Apply Director Changes
        - unlock-director
            - Releases the lock for the foundation to allow other pipelines to run

Products

Now for the structure of the pipeline used to maintain the products (tiles)

- Resource Types
    - Pivnet
- Resources
    - platform-automation
        - Reference to the platform-automation resource from pivnet that contains the docker image used to execute the tasks as well as custom tasks to perform the processing * configuration
    - Reference to the git repository that contains the opsman/director configuration
        - Will typically define specific paths in the repository that are related to the opsman/director
    - lock
        - Reference to a git repository to hold locks used during the pipeline (this can also be in the configuration git repository - but will have a separate concourse resource definition)
- Jobs
    - force-unlock
        - Used to force the lock to be released and would be manually executed on-demand
    - lock-tiles
        - Starting point and triggered by changes to the configuration resource
        - Obtains the lock for this foundation so no other instances of this pipeline or the director pipeline will execute at the same time
    - validation-configuration
        - Checks that all required secrets are available in the secret manager used (credhub / vault)
        - No need to continue if the pipeline is destined to fail because of missing secrets
    - (specific-product) (i.e. tas / mysql / etc…)
        - download-stage-tile-and-stemcell
        - configure
        - apply-product-change
    - (specific-product)-errands (i.e. tas-errands / mysql-errands / etc…)
        - run-(errand) (i.e. run-smoke_tests / run-push-usage-service / etc…)
        - This task is repeated for each errands
    - unlock-director
        - Releases the lock for the foundation to allow other pipelines to run

Alternatives

Single task for all errands

An alternative strategy for running errands has been implemented where there is a single task that will run all of the errands instead of a task for each errand.

The advantages of having a single task for all errands are:

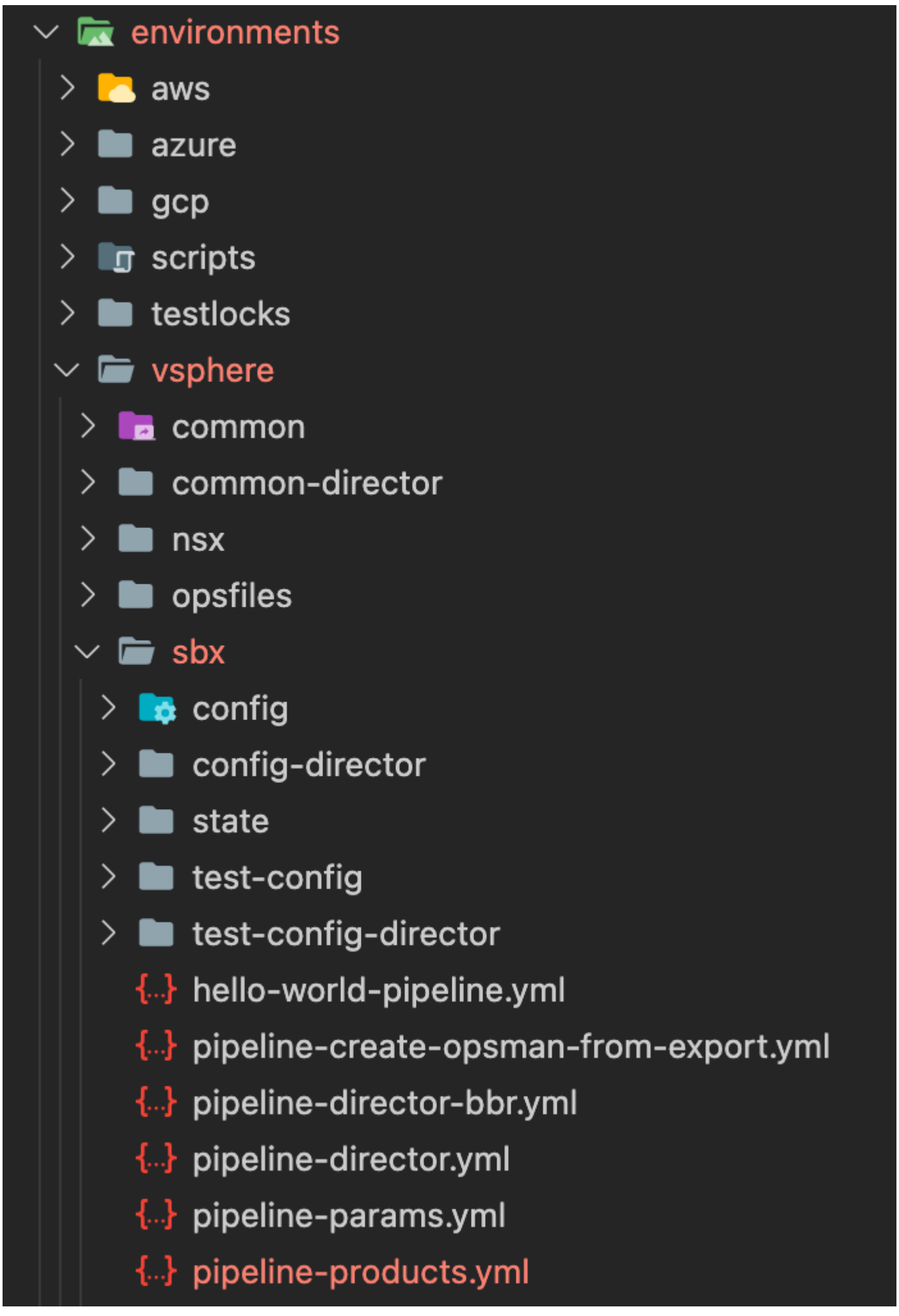
- The pipeline itself is easier since there is a single task
- If the tasks need to be maintained there is no need to re-fly the pipeline.

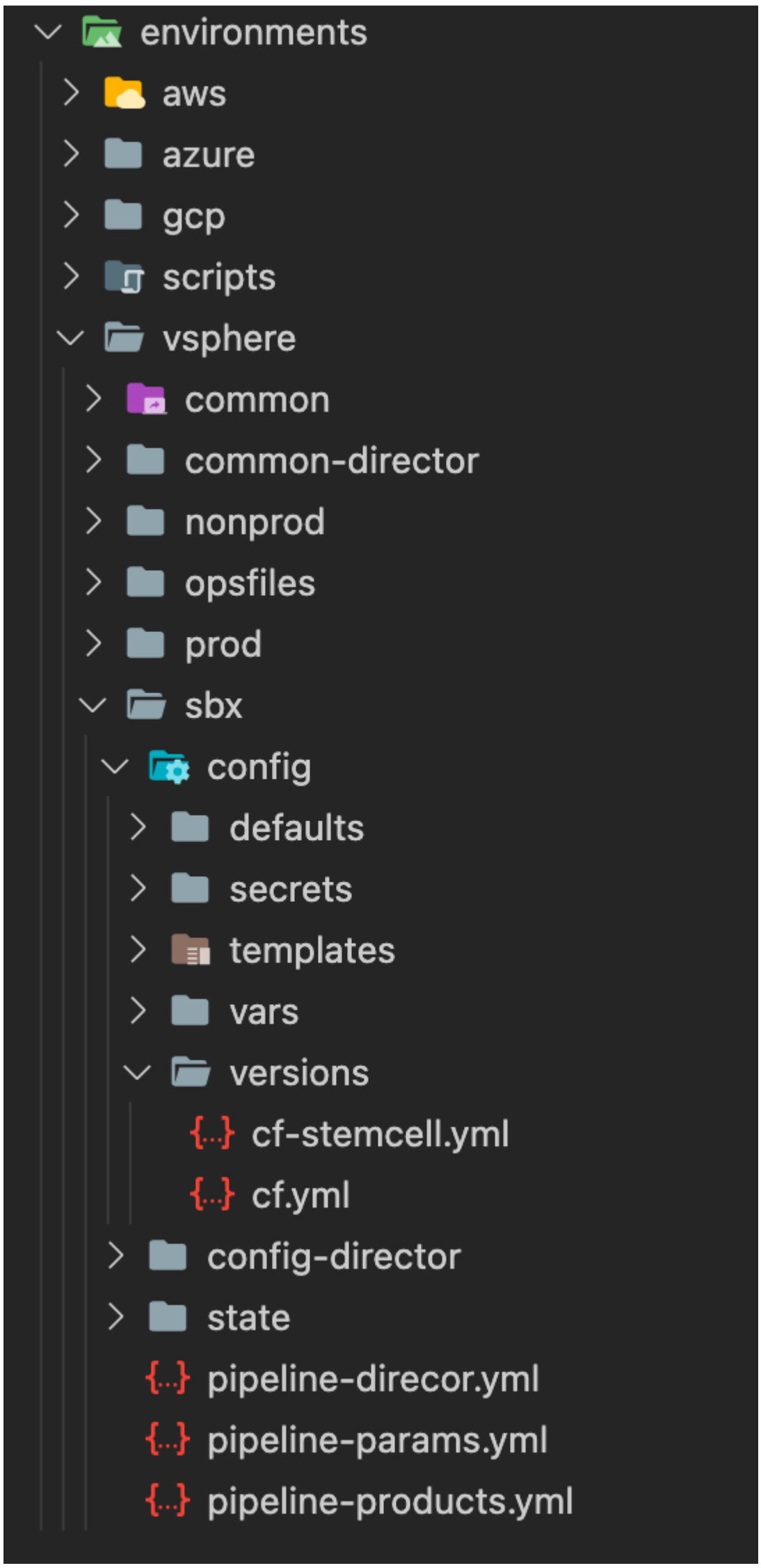
However, the advantages of having each errand run in a separate task are:

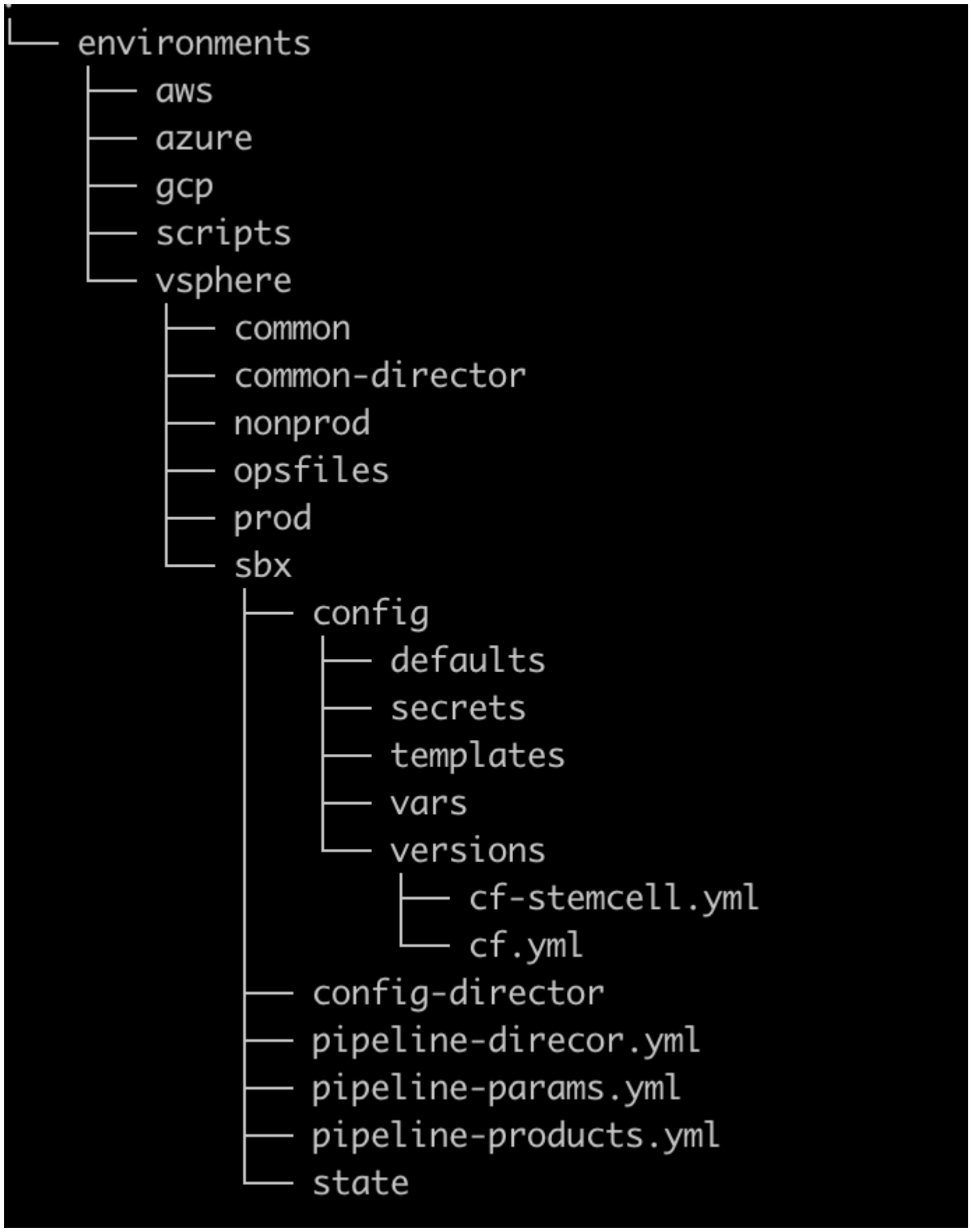
- Each task can have an “attempts” which will allow any task failure to be retried without running all preceding errands
- Visibility of failed errands

Directory Structure

Which of these makes the most sense?




environments
aws
azure
gcp
scripts
vsphere
common
common-director
nonprod
opsfiles
prod
sbx
config
defaults
secrets
templates
vars
versions
cf-stemcell.yml
cf.yml
config-director
state
pipeline-direcor.yml
pipeline-params.yml
pipeline-products.yml



Define the workflow / Products

The general workflow is the same when you are adding or updating a product.

- Start with a “version” file.
  - This contains the product information including the slug, glob, and version number for the product
- Run generateconfig.sh script
  - This will pull product information from pivnet and generate
- Run validateconfig.sh
- Working with opsfiles (regenerate / validate)
- Promotion

It all starts with the version file. But, where should the version file reside? Given that several files are going to be generated, we take an opinionated view of the repository structure. To get started, we will need to create the following structure:

```
repository-home/
|   environments
|   |   scripts
|   |   iaas
```

vmware