Halyard

Code/Design Walkthrough

Before we begin...

1. Halyard’s grown quite large (~30K lines of Java); it’s time to document & explain how it works
2. These slides are meant to help new contributors & interested team members understand the codebase well enough to make core changes if needed
3. Most interfaces/classes shown are abbreviated for the sake of clarity

Walkthrough & Takeaways

1. High-level architecture
2. Config validation
3. Config generation
4. Deployments

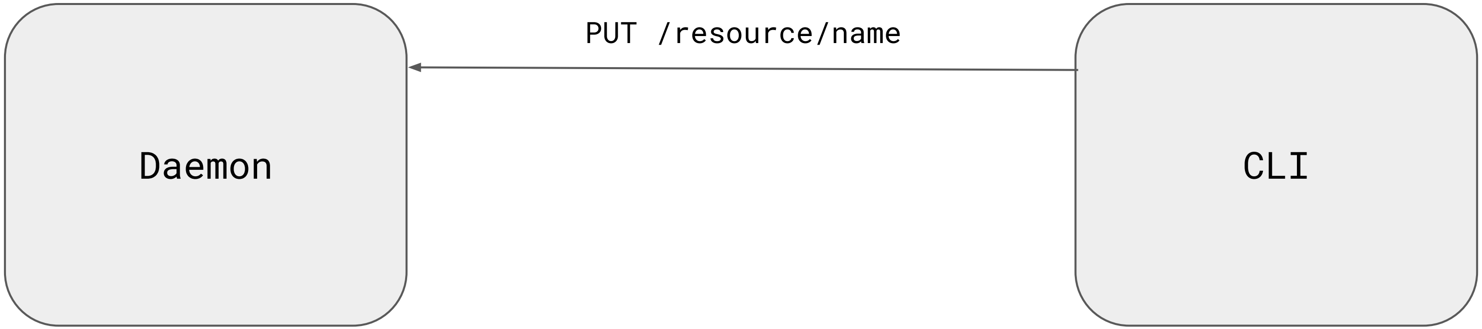
1. High-level Architecture

Request Flow

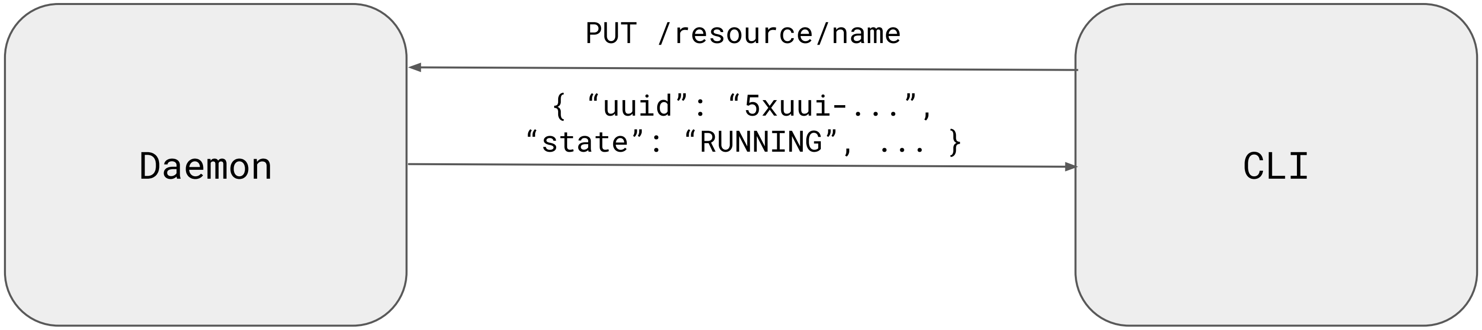
Daemon

CLI

Request Flow



Request Flow



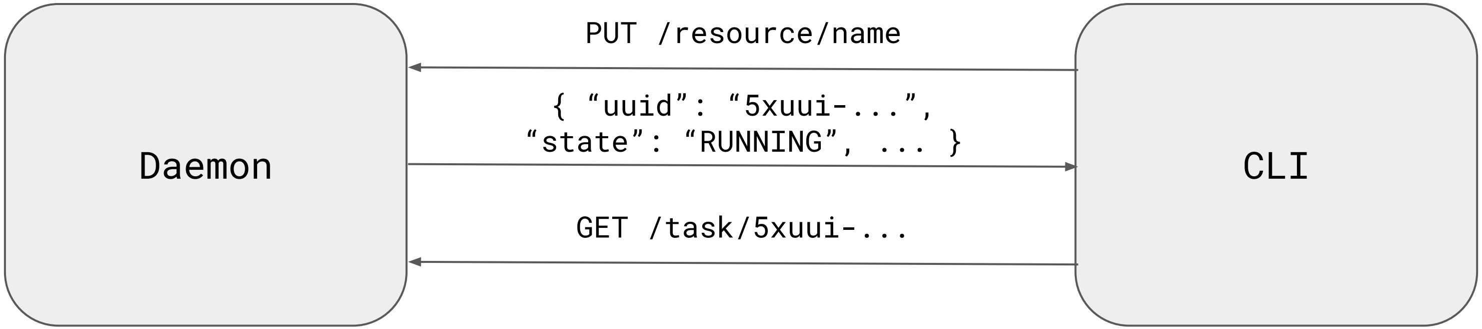
Daemon

PUT /resource/name

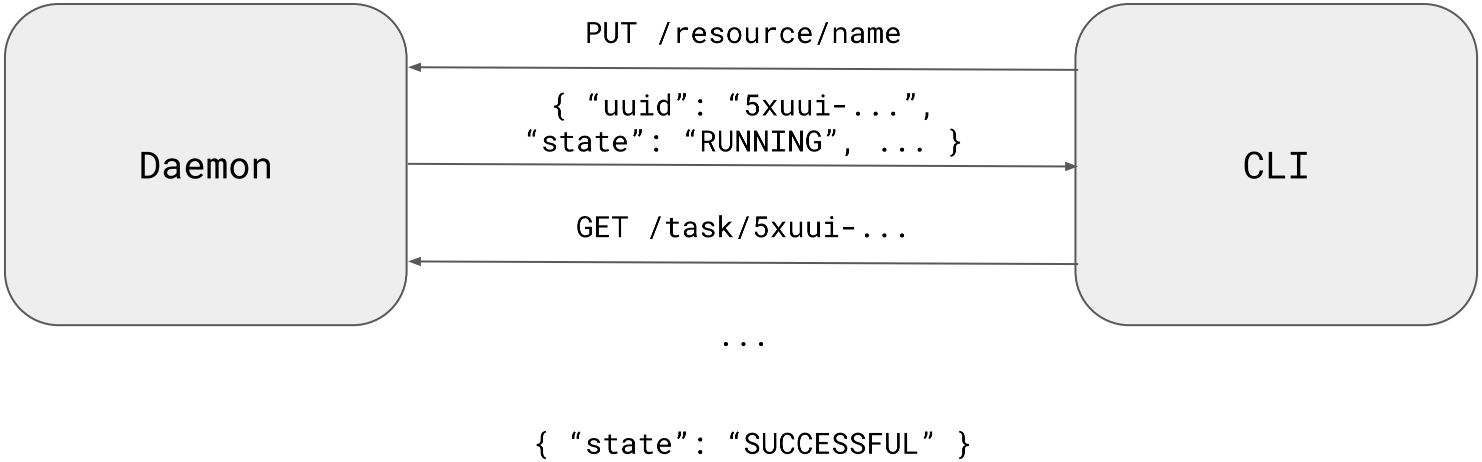
{ “uuid”: “5xuui-...”,
“state”: “RUNNING”, ... }

CLI

Request Flow



Request Flow



Daemon

CLI

PUT /resource/name

{ “uuid”: “5xuui-...”,
“state”: “RUNNING”, ... }

GET /task/5xuui-...

...

{ “state”: “SUCCESSFUL” }

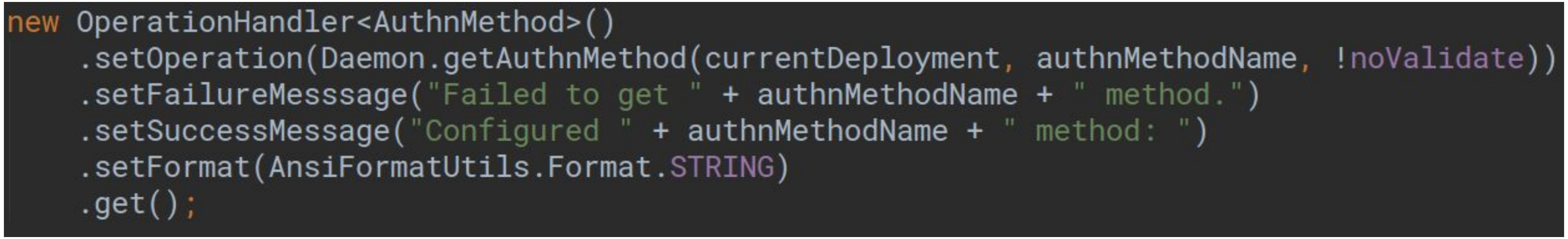
CLI

1. Does next-to-no validation
2. Every command extends `NestableCommand`

```
abstract class NestableCommand {

     abstract void executeThis();

     abstract String commandName();

     abstract String description();

     protected void registerSubcommand(NestableCommand c) {}

}
```

CLI

1. Auto-generates docs (whenever CLI is built)
2. Auto-generates command-completion (whenever Halyard is installed)
3. Request flow & output formatting is wrapped by `OperationHandler<T>`



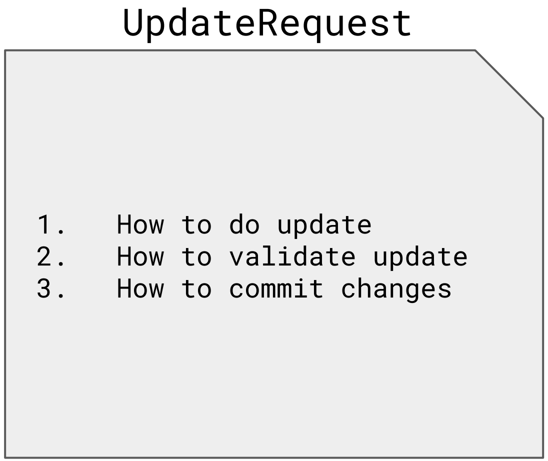
```
new OperationHandler<AuthnMethod>()
    .setOperation(Daemon.getAuthnMethod(currentDeployment, authnMethodName, !noValidate))
    .setFailureMesssage("Failed to get " + authnMethodName + " method.")
    .setSuccessMessage("Configured " + authnMethodName + " method: ")
    .setFormat(AnsiFormatUtils.Format.STRING)
    .get();
```

Daemon

`PUT /resource/name`

```
[18] 0:bash*                              "lwander.nyc.corp.googl" 17:32 29-Mar-17
```

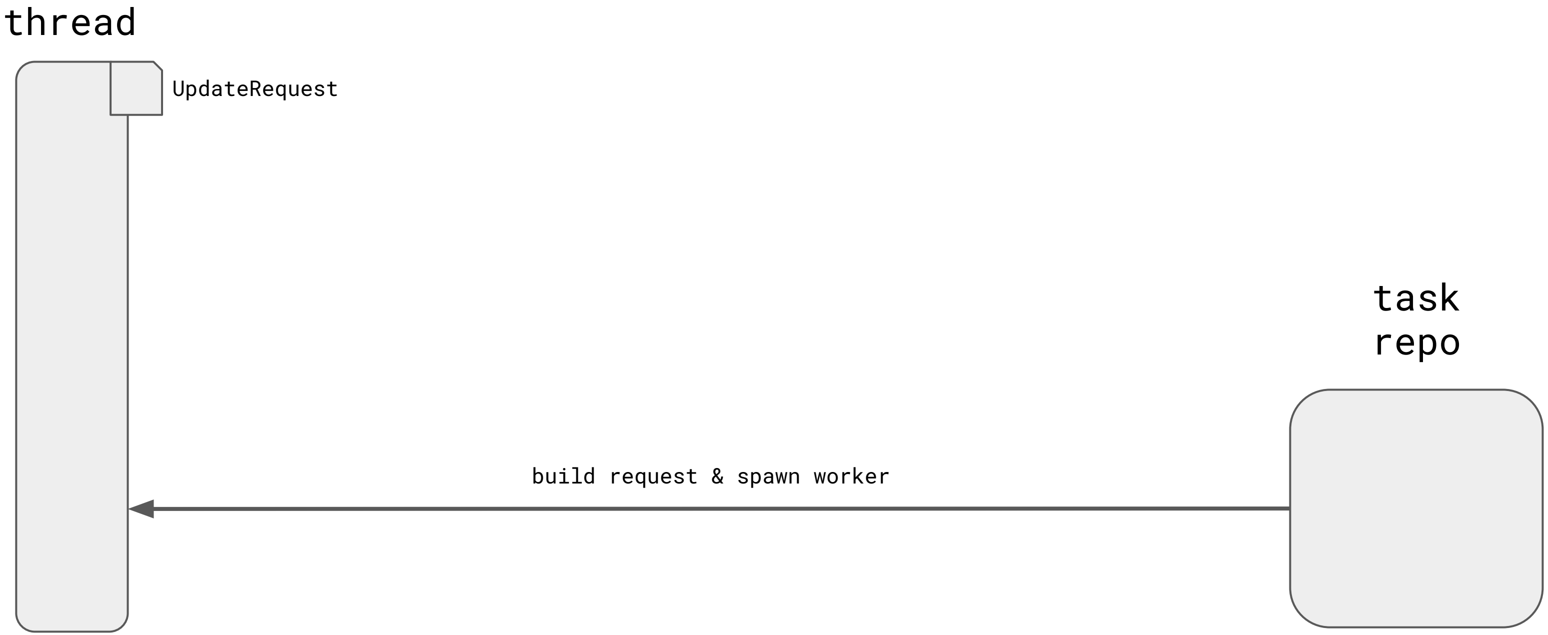
Daemon



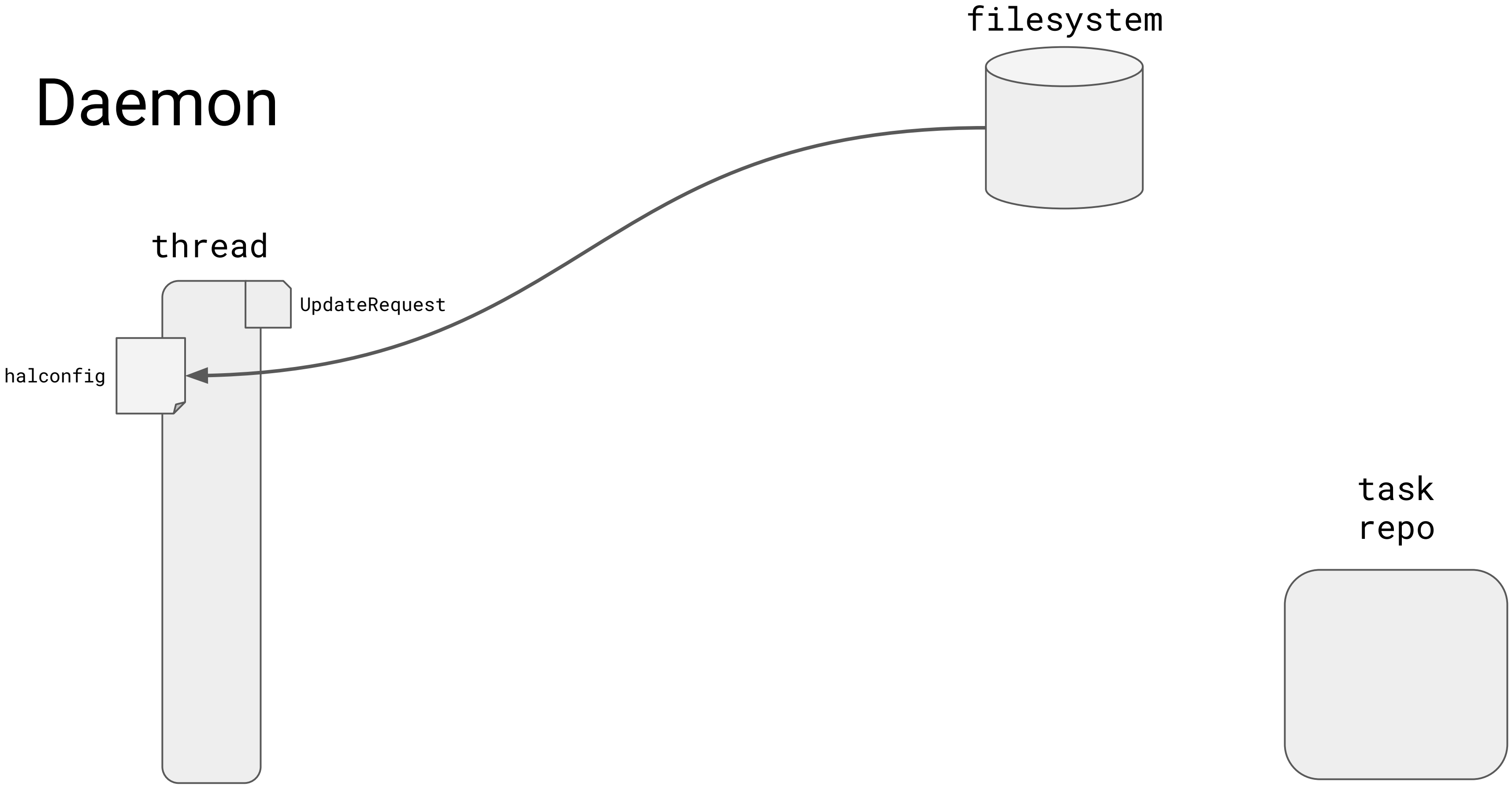
UpdateRequest

1. How to do update
2. How to validate update
3. How to commit changes

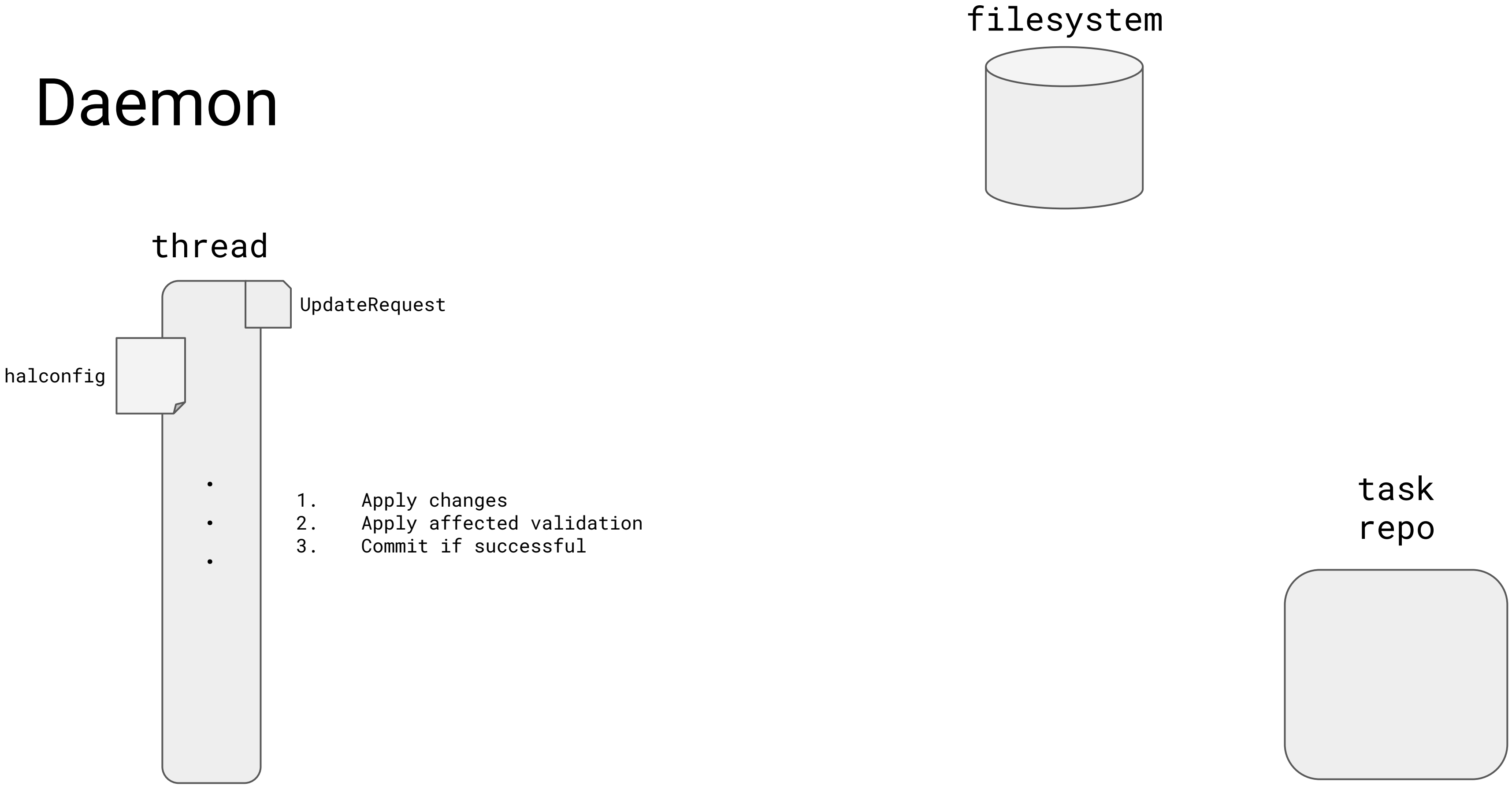
Daemon


thread
UpdateRequest
task
repo
build request & spawn worker

Daemon

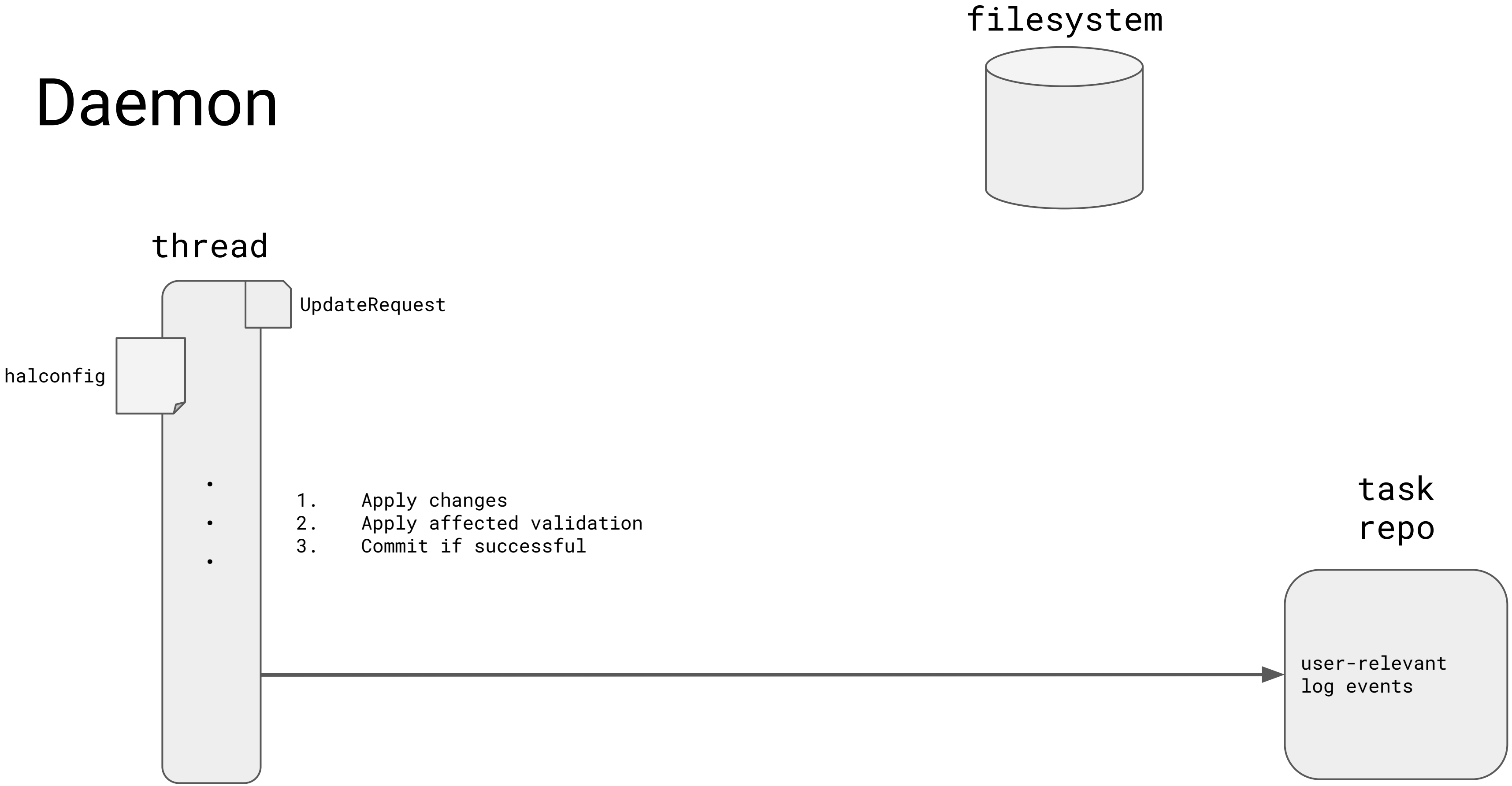


Daemon



1. Apply changes
2. Apply affected validation
3. Commit if successful

Daemon



filesystem

thread

UpdateRequest

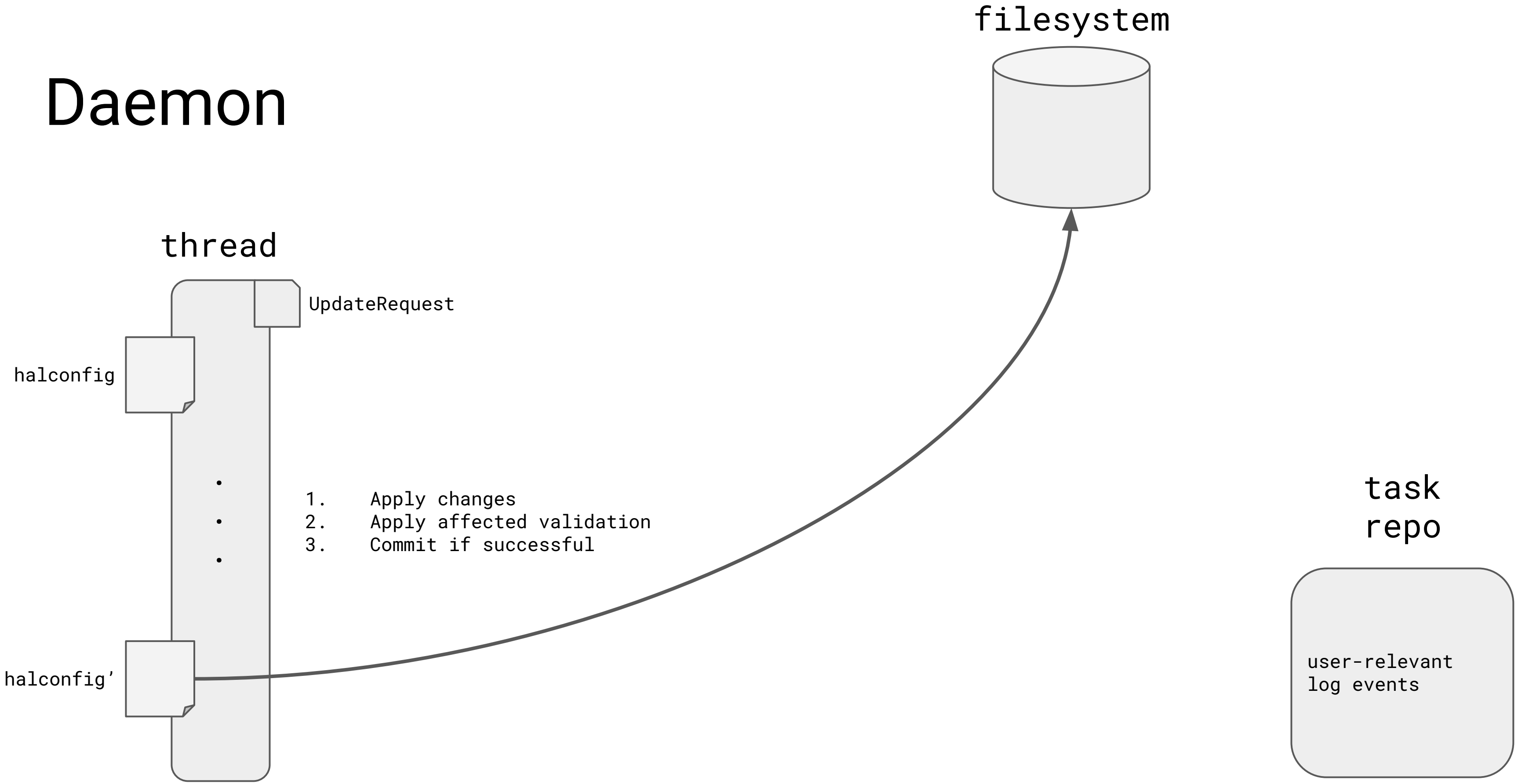
halconfig

1. Apply changes
2. Apply affected validation
3. Commit if successful

task repo

user-relevant log events

Daemon



1. Apply changes
2. Apply affected validation
3. Commit if successful

2. Config Validation

Nodes

1. `halconfig` is deserialized into a bunch of classes
2. Each class extends `Node`

```
abstract class Node {

     abstract String getName();

     abstract NodeIterator getChildren();

     boolean matchesToRoot(NodeFilter filter) { ... }

}
```

Node Iterators

1. List all nested config Nodes in a given Node
2. Auto-generated by `NodeIteratorFactory`

```
interface NodeIterator {

    Node getNext(NodeFilter filter);

}
```

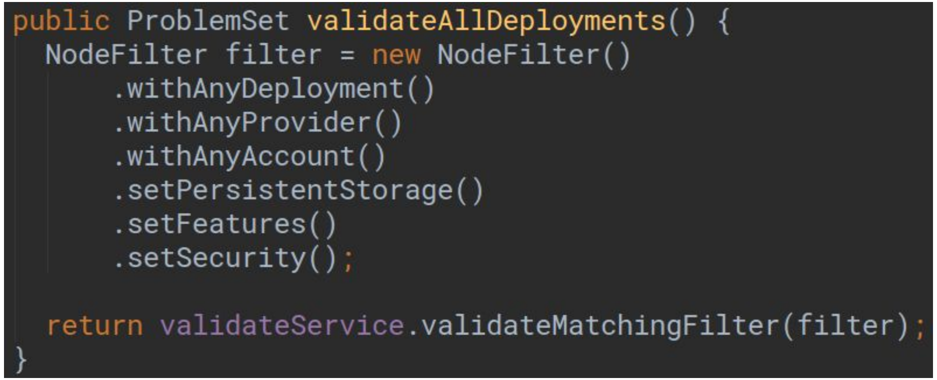
Node Filters

1. Matches a path of nodes in your `halconfig`
2. Aggregates & applies a bunch of `NodeMatcher` clauses.

```
public class NodeFilter {
     public boolean matches(Node n) {
          return matchers.stream().anyMatch(m -> m.matches(n));
     }
}
```

Why bother?

1. Makes validation & node lookup a breeze



```
public ProblemSet validateAllDeployments() {
  NodeFilter filter = new NodeFilter()
      .withAnyDeployment()
      .withAnyProvider()
      .withAnyAccount()
      .setPersistentStorage()
      .setFeatures()
      .setSecurity();

  return validateService.validateMatchingFilter(filter);
}
```

Problems

1. Many things can go wrong
2. If something “expected” goes wrong, build a `Problem`

```
public class Problem {
    String message;
    String remediation;
    Severity severity;
    List<String> options;
    String location;
}
```

```
enum Severity {
    NONE,    # baseline
    WARNING, # bad practice
    ERROR,   # bad (could work?)
    FATAL,   # bad (can’t work.)
}
```

Problem Sets

1. All the `Problem`s encountered during your operation
2. User-specifiable max `Severity`

```
class ProblemSet {

     Set<Problem> problems;

     void throwIfSeverityExceeds(Severity s) { ... }

}
```

Validators

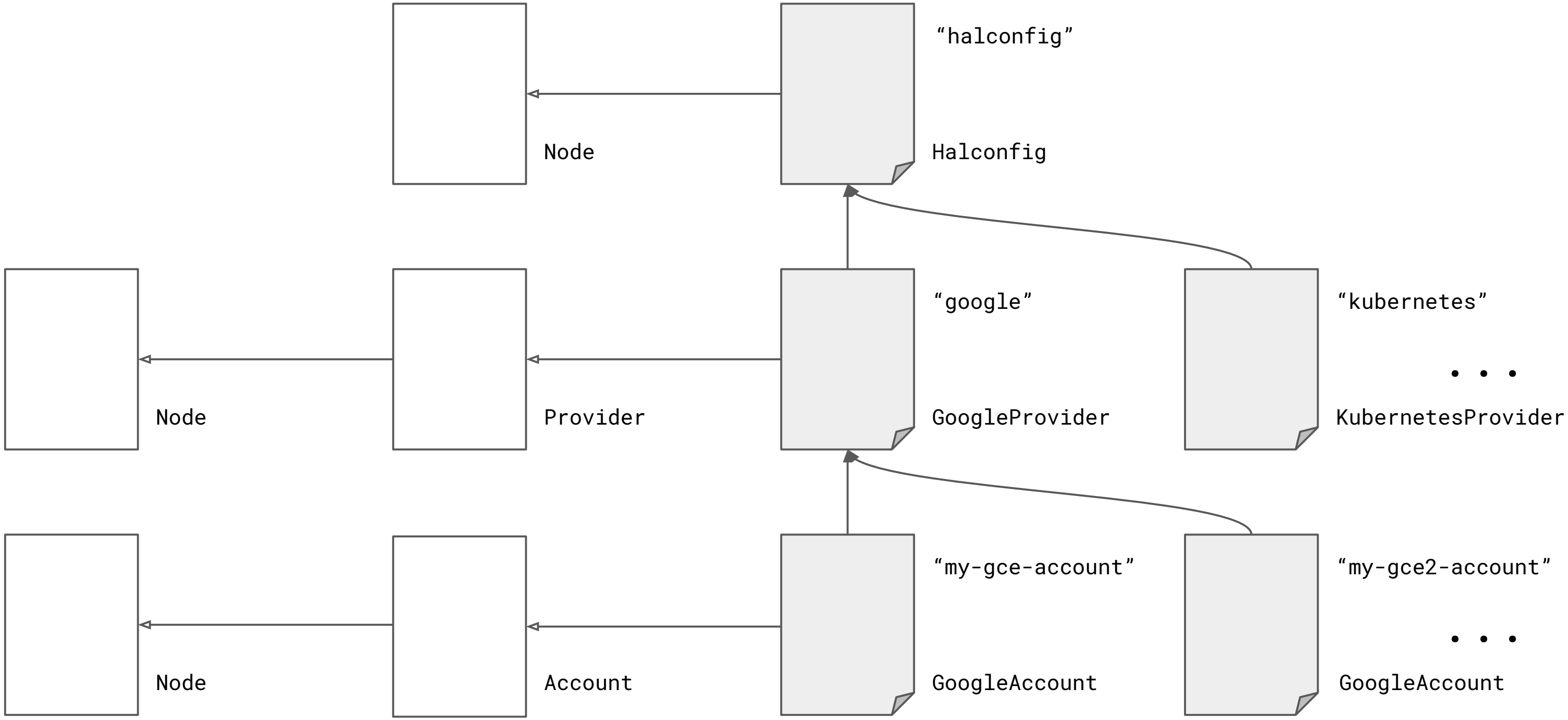
1. Visitor pattern accepting `Node`s
2. Every `Node` matching your `NodeFilter` has its class hierarchy ascended applying all matching Validators along the way

```
interface Validator<T extends Node> {

     void validate(ProblemSetBuilder p, T n);

}
```

Validators

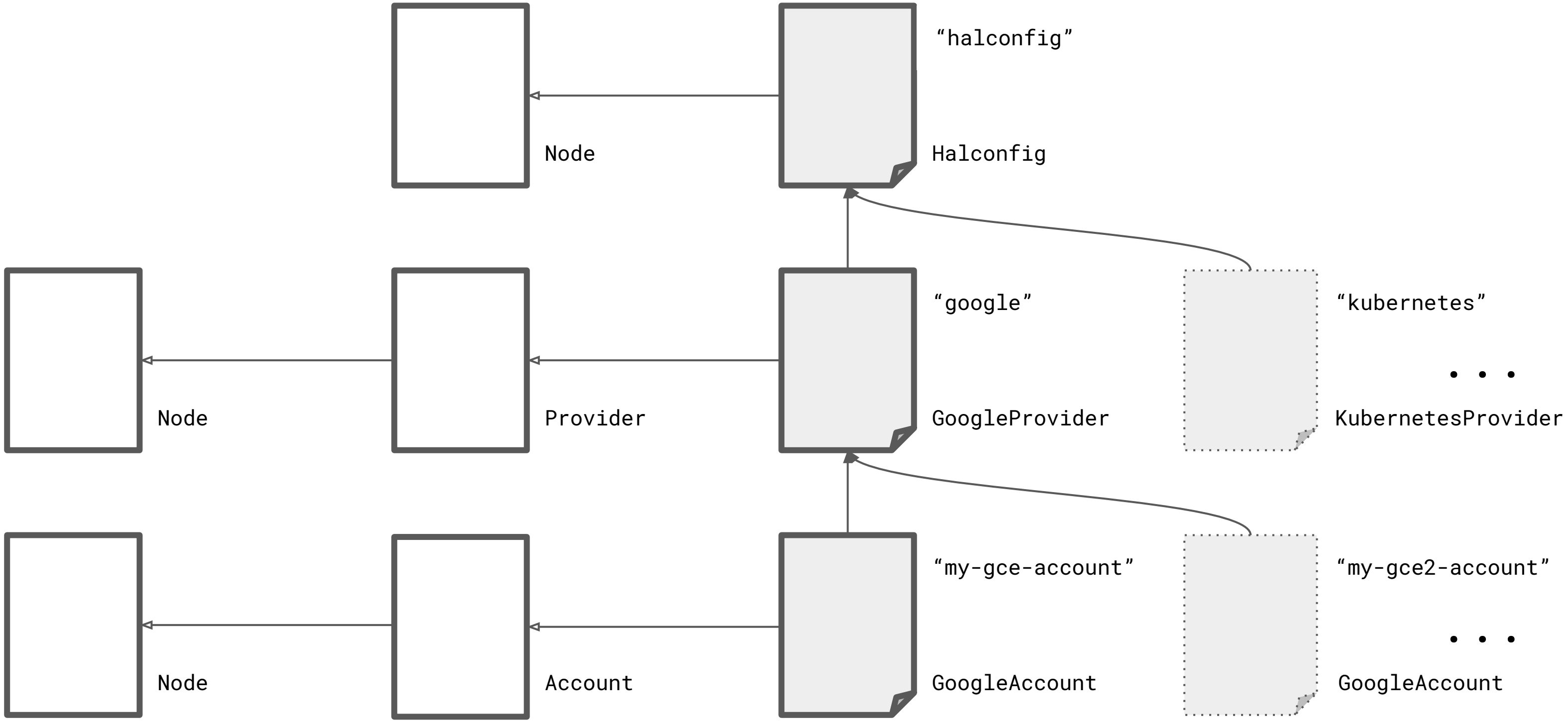
Let’s assume we’ve created a filter to validate our GCE account `“my-gce-account”`...

Validators



Node
"halconfig"
Halconfig
Node
Provider
"google"
GoogleProvider
"kubernetes"
. . .
KubernetesProvider
Node
Account
"my-gce-account"
GoogleAccount
"my-gce2-account"
. . .
GoogleAccount

Validators


“halconfig”
Node
Halconfig
“google”
“kubernetes”
. . .
Node
Provider
GoogleProvider
KubernetesProvider
“my-gce-account”
“my-gce2-account”
. . .
Node
Account
GoogleAccount
GoogleAccount

Life of a Request

```
hal config provider google enable
```

```
[12] 0:bash*                                    "lwander.nyc.corp.googl" 18:07 28-Mar-17
```

Life of a Request

```
--> GET /v1/config/currentDeployment

<-- “default”

--> PUT /v1/config/deployments/default/providers/google/enabled?validate=true

    { “enabled”: “true” }

<-- { “uuid”: “16cbR-...”, “state”: “RUNNING” }

...

<-- { “uuid”: “16cbR-...”, “state”: “SUCCESSFUL” }
```

Life of a Request

```
--> GET /v1/config/currentDeployment

<-- “default”

--> PUT /v1/config/deployments/default/providers/google/enabled?validate=true

    { “enabled”: “true” }

<-- { “uuid”: “16cbR-...”, “state”: “RUNNING” }

???

<-- { “uuid”: “16cbR-...”, “state”: “SUCCESSFUL” }
```

Life of a Request

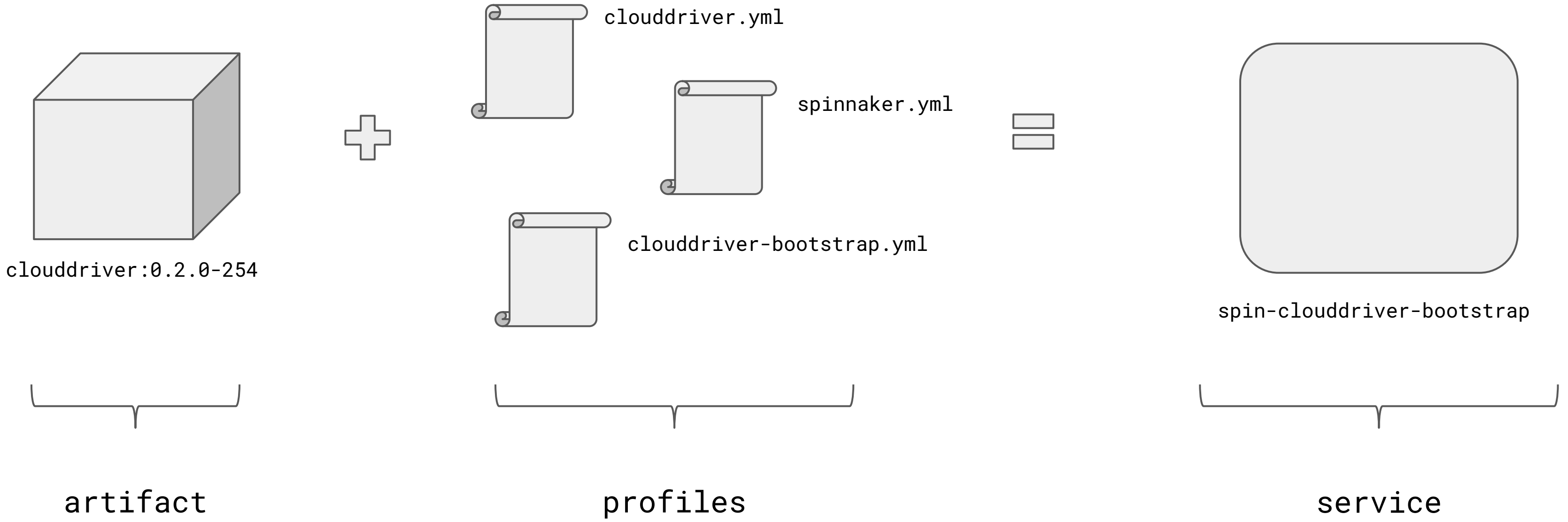
1. Enter in the `ProviderController`
2. How to modify the `Halconfig`
    a. Load the `Provider`’s `Node`
    b. Invoke the `LookupService`
    c. Parse the `Halconfig`
3. How to validate the `Halconfig`
    a. Invoke the `ValidateService`
    b. Find & Apply all matching `Validators`
    c. Run `GoogleProviderValidator`
4. Build the `DaemonResponse`

3. Config Generation

Some Terminology...

1. An `Artifact` refers to an unconfigured, deployable object at some version
   a. clouddriver:latest, deck:0.2.0-254, etc…
2. A `Profile` is a single file that can be “applied” to an `Artifact`
   a. clouddriver.yml, clouddriver-local.yml, apache2/ports.conf, etc…
3. A `Base Profile` is a single file that can be used to help generate a `Profile`
4. A `Service` is the combination of an `Artifact` and a set of `Profiles`
   a. clouddriver, clouddriver-caching, echo-cron

Some Terminology...

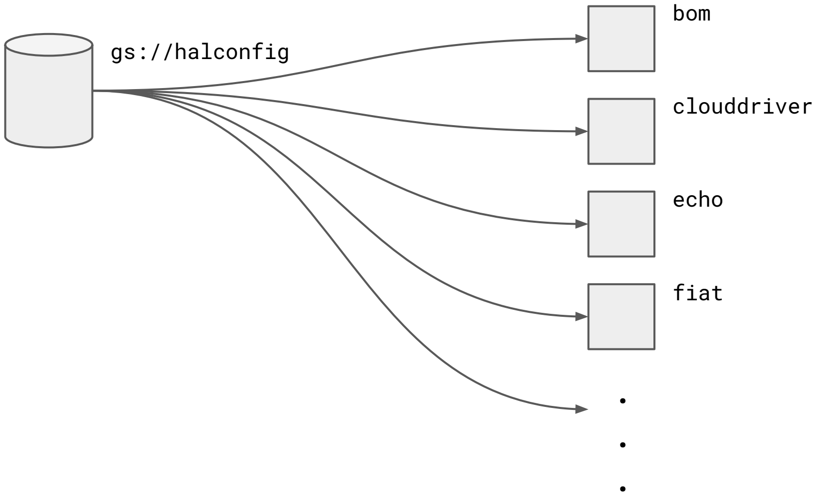


High-level process

1. Collect the set of `Service`s you need for your type of deployment
2. Have each `Service` generate all needed `Profile`s
3. Write each `Profile` to a staging directory `/home/spinnaker/.spinnaker/`
4. Copy user-provided `Profile`s into the staging directory as well

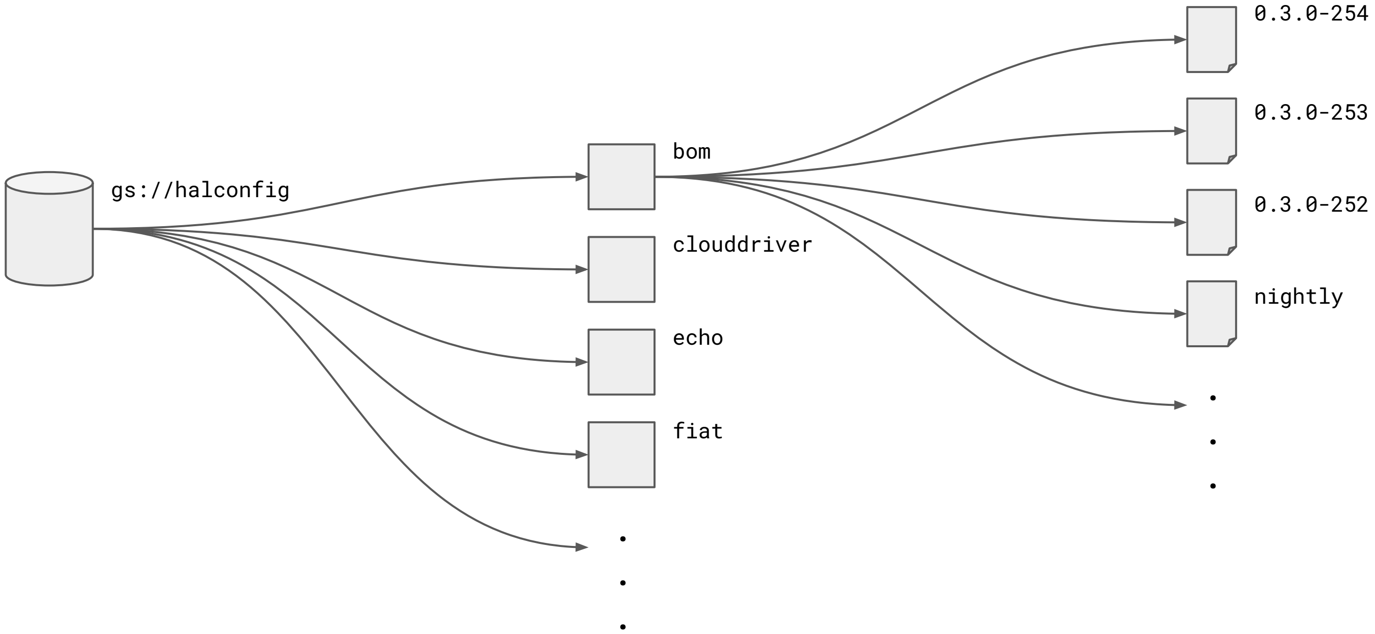
Bill of Materials

1. Maps top-level version to artifact versions & their base profiles
2. Sample



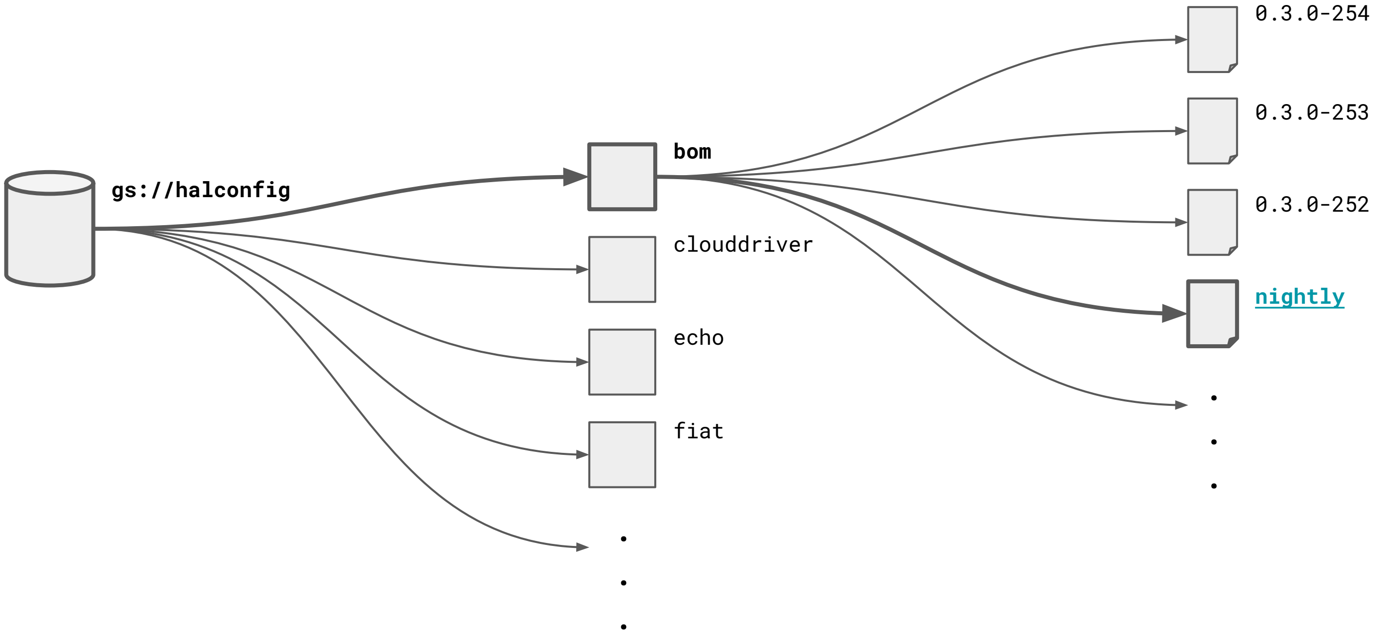
Bill of Materials

1. Maps top-level version to artifact versions & their base profiles
2. Sample



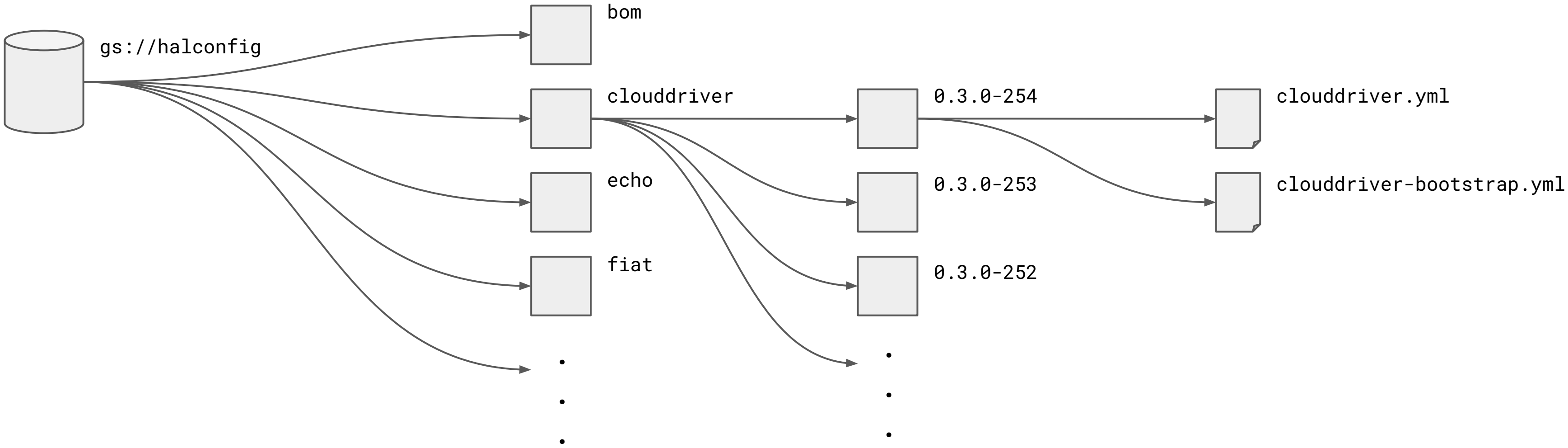
Bill of Materials

1. Maps top-level version to artifact versions & their base profiles
2. Sample



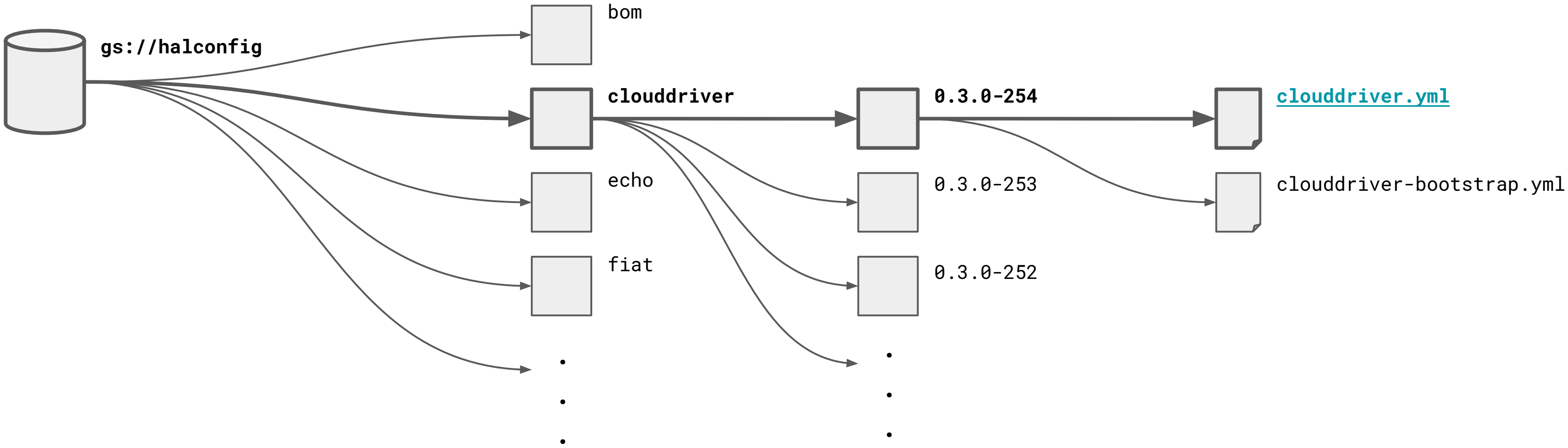
Bill of Materials

1. Maps top-level version to artifact versions & their base profiles
2. Sample



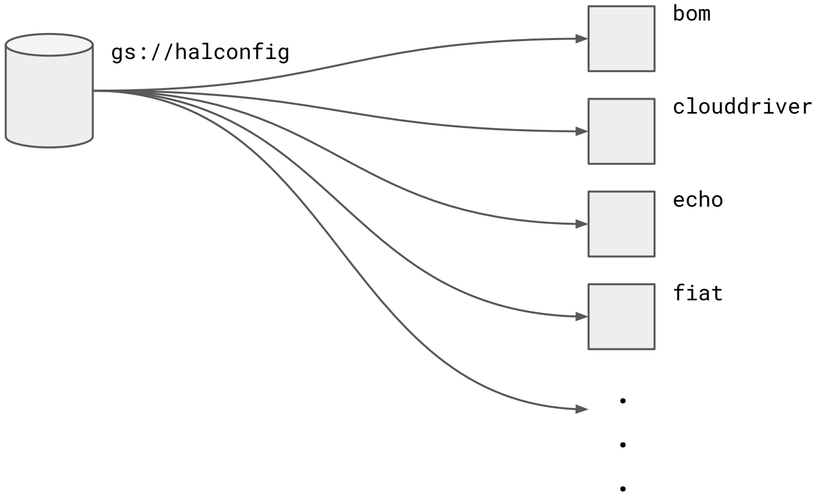
Bill of Materials

1. Maps top-level version to artifact versions & their base profiles
2. Sample

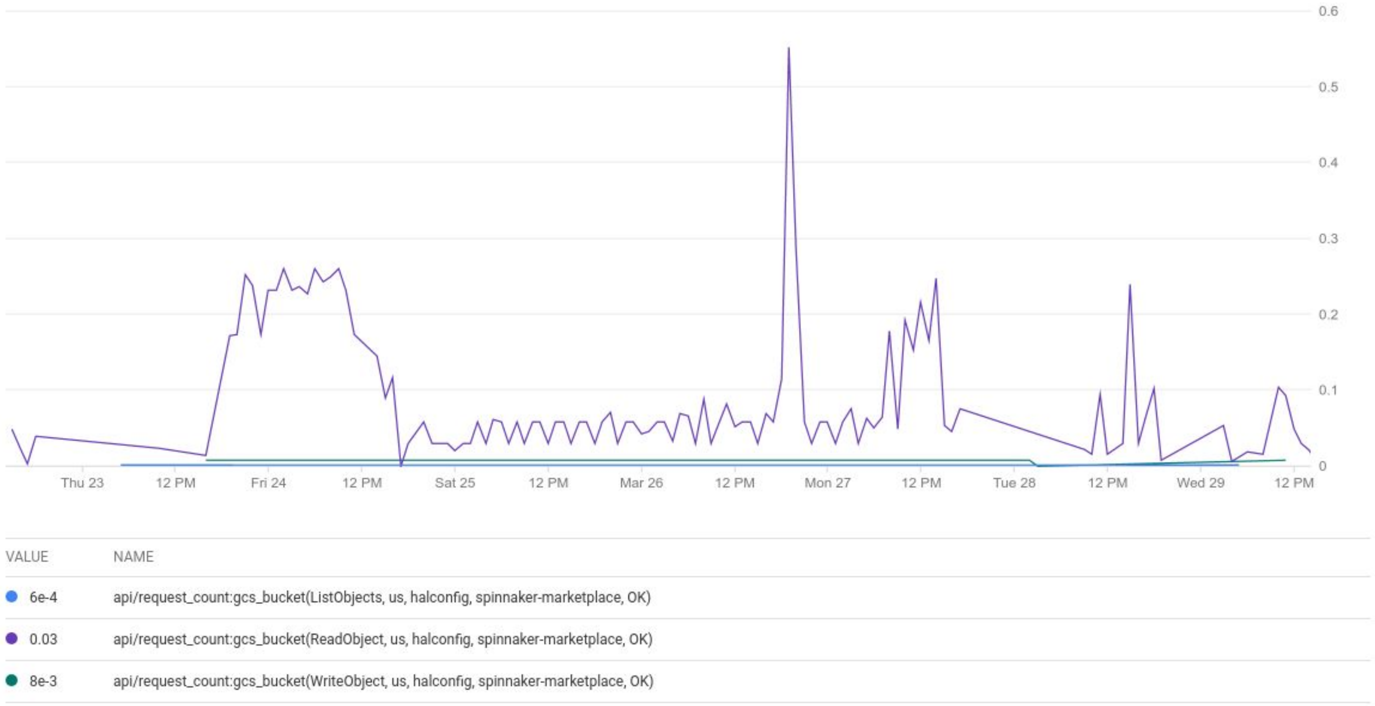


Bill of Materials

1. Maps top-level version to artifact versions & their base profiles
2. Sample
3. This is all handled by the `ProfileRegistry`



Bill of Materials (`gs://halconfig`)



VALUE	NAME
● 6e-4	api/request_count:gcs_bucket(ListObjects, us, halconfig, spinnaker-marketplace, OK)
● 0.03	api/request_count:gcs_bucket(ReadObject, us, halconfig, spinnaker-marketplace, OK)
● 8e-3	api/request_count:gcs_bucket(WriteObject, us, halconfig, spinnaker-marketplace, OK)

Profiles

```
class Profile {
     String contents;
     final String outputFile;
     final String name;
}
```

Profile Factories

1. Each `ProfileFactory` can create a certain kind of `Profile`
2. Supply your `Halconfig` and `SpinnakerRuntimeSettings` to build a `Profile`
3. Most of `Halconfig` looks very similar to Spinnaker’s config, so these factories are generally quite short
4. Any field annotated with `@LocalFile` is rewritten to point at that file copied into the staging directory with its fully-qualified directory name hashed

spinnaker.yml

1. Built from `SpinnakerRuntimeSettings`
2. Contains endpoint information for all possible services
3. Distributed with every Spinnaker service

```
services:

  clouddriver:

    enabled: true

    baseUrl: http://spin-clouddriver.spinnaker:7002

  clouddriverBootstrap:

    enabled: true

    baseUrl: http://spin-clouddriver-bootstrap.spinnaker:7002

  deck:

        ...
```

spinnaker.yml

1. Services reference `spinnaker.yml` via SPEL
2. Toggling profiles controls which `Services` are in communication
3. `orca.yml` vs `orca-bootstrap.yml`

```
services:
  clouddriver:
    enabled: true
    baseUrl: http://spin-clouddriver.spinnaker:7002
  clouddriverBootstrap:
    enabled: true
    baseUrl: http://spin-clouddriver-bootstrap.spinnaker:7002
  deck:
       ...
```

4. Deployments

Config Mounting

1. `Profile`s need to get into the environment Spinnaker is running on
2. Each `Profile` is translated into a `ConfigMount`

```
class ConfigMount {

    String id;

    String mountPath;

}
```

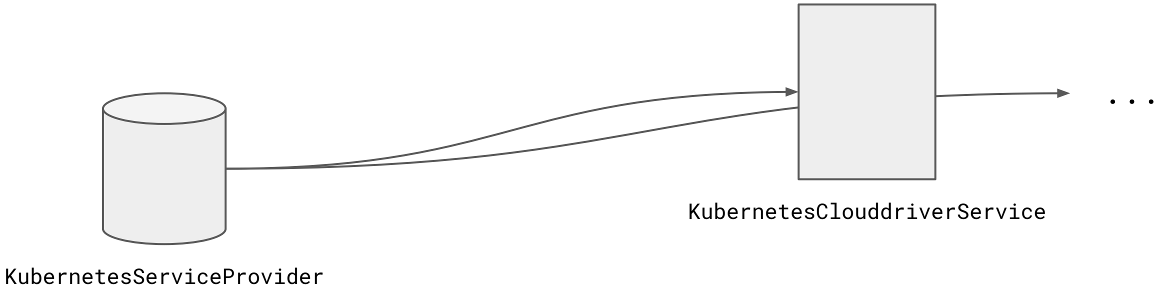
Config Mounting

1. Each cloud provider is responsible for uploading their `Profile`s, and giving each an `id`
2. `Profile`s specify their own `mountPath`, where they are read from

```
class ConfigMount {
    String id;
    String mountPath;
}
```

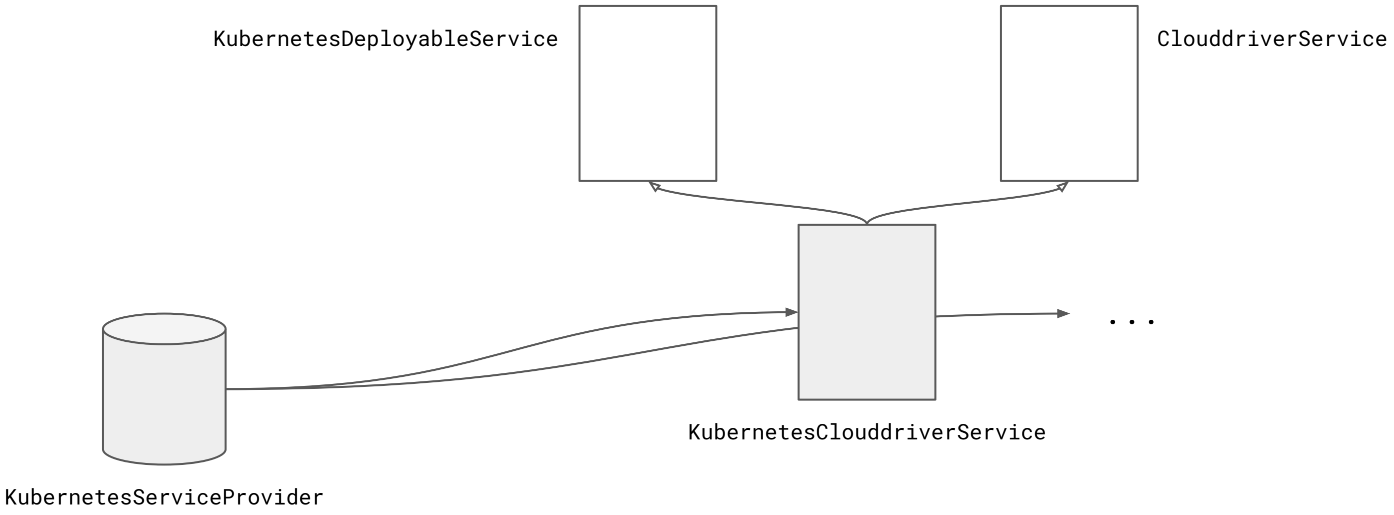
Service Provider

1. Every type of Spinnaker deployment has a `ServiceProvider`



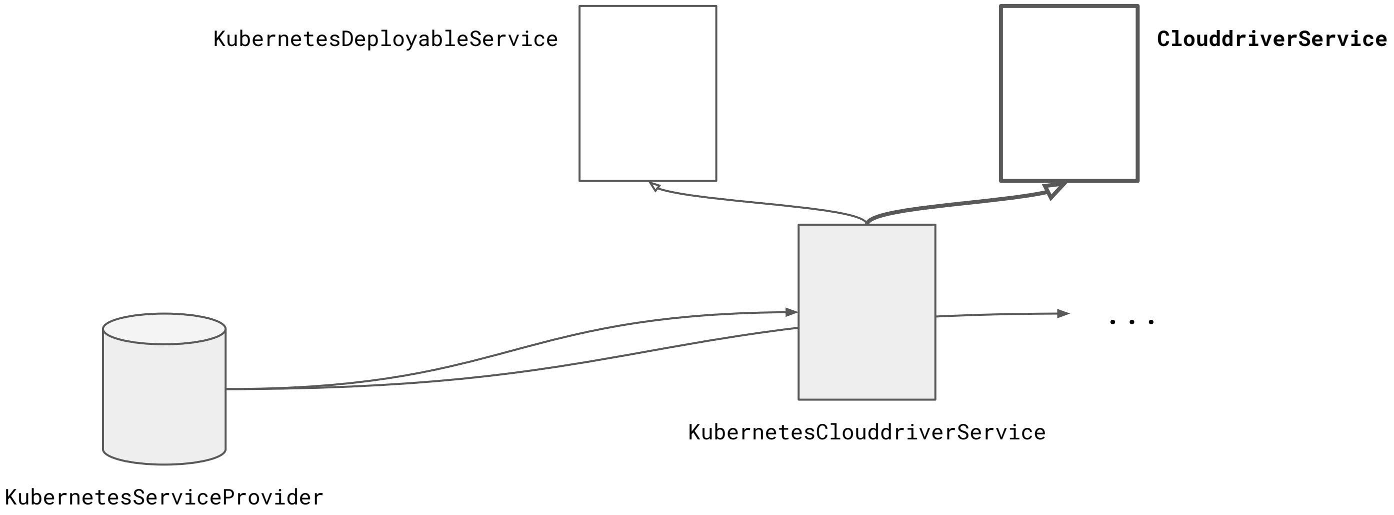
Service Provider

1. Every type of Spinnaker deployment has a `ServiceProvider`
2. Services implement interfaces that make them “deployable” or “installable”


KubernetesDeployableService
CloudriverService
KubernetesServiceProvider
KubernetesClouddriverService
...

Services

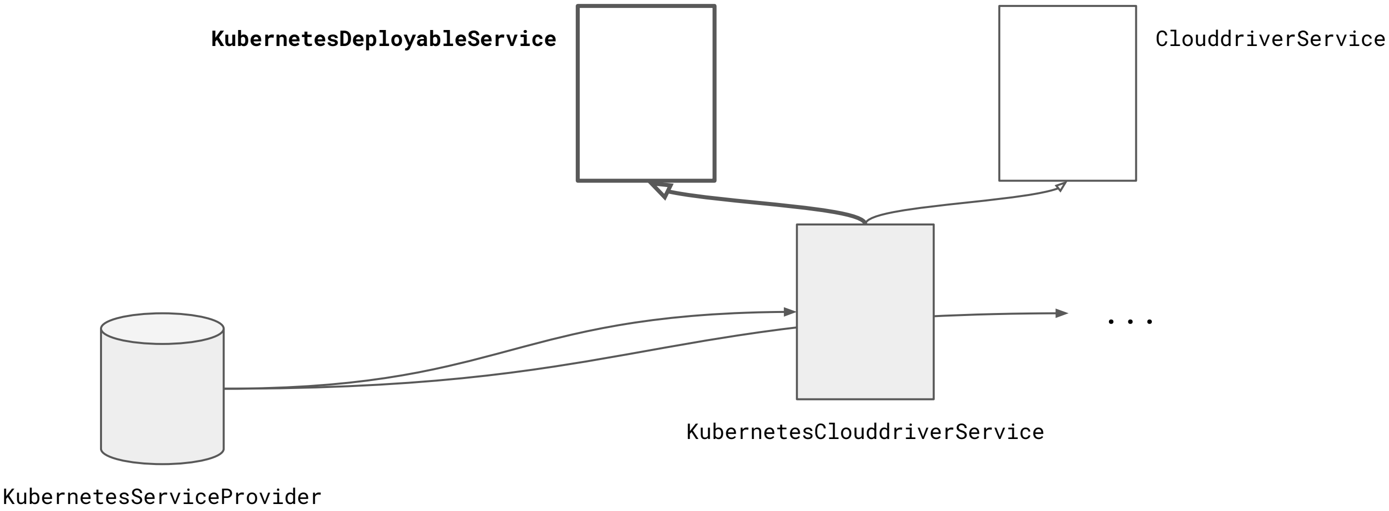
1. `Service`s (the ones from part 3.) build profiles and have abstract methods for building `RuntimeSettings`
2. Also expose Retrofit interfaces for communicating with them



KubernetesDeployableService
ClouddriverService
...
KubernetesCloudriverService
KubernetesServiceProvider

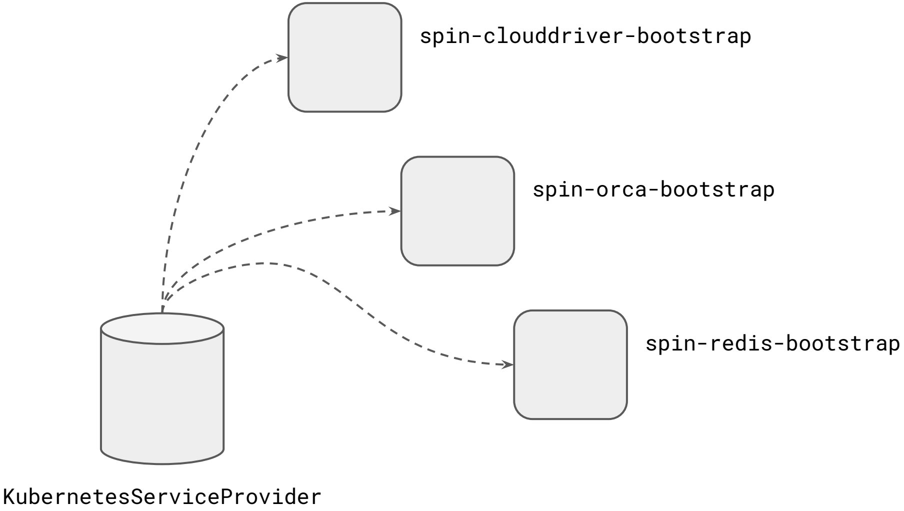
Deployable Services

1. `DeployableService`s build pipelines that can be sent to Orca to deploy that exact service (server group + load balancer)


KubernetesDeployableService
CloudddriverService
...
KubernetesClouddriverService
KubernetesServiceProvider

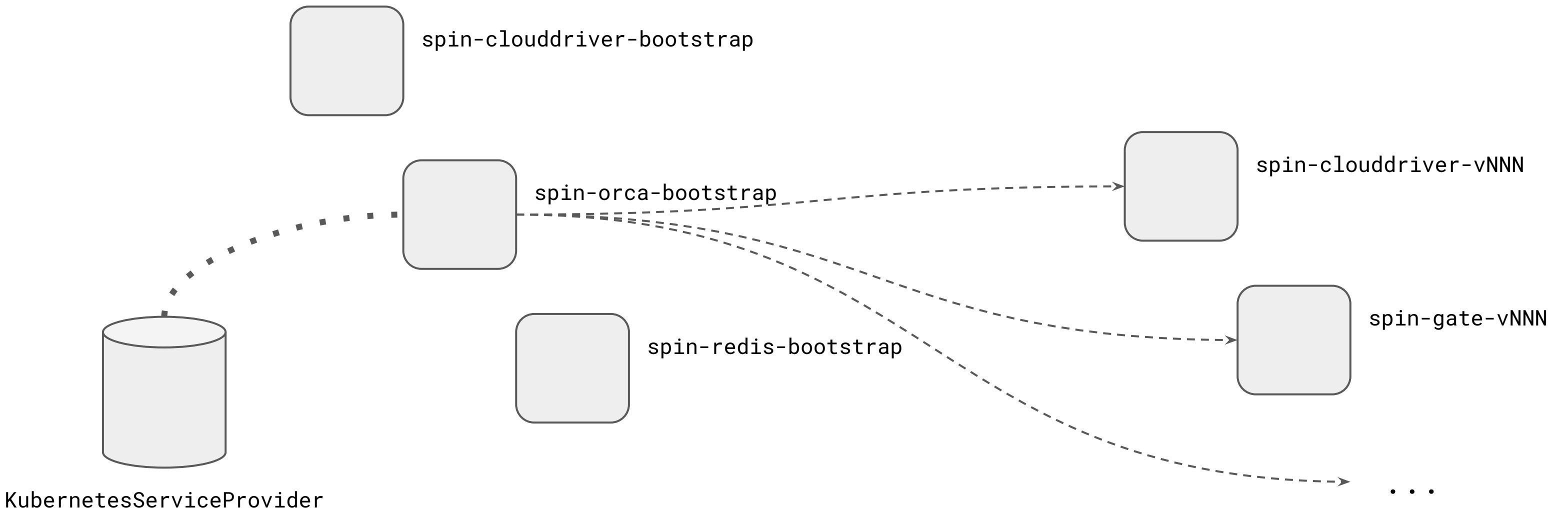
Deployment Procedure

1. `ServiceProvider` lists all services marked as required for “bootstrapping”
2. These services are deployed directly using the cloudprovider’s API



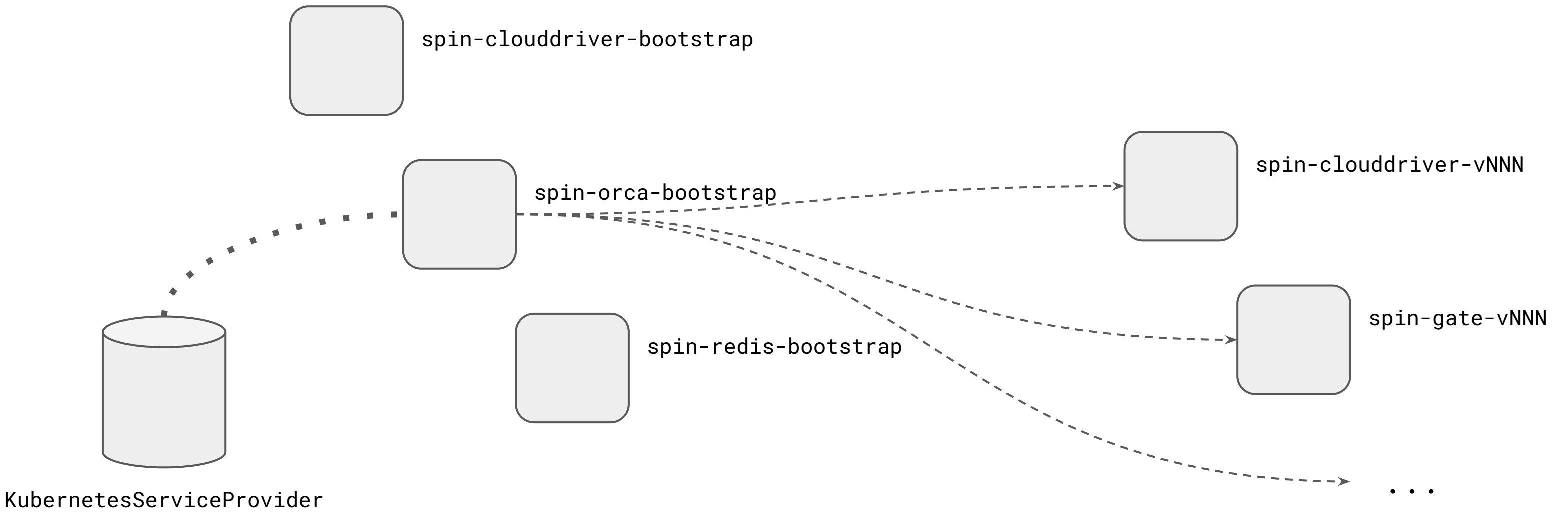
Deployment Procedure

1. An connection is opened to the instance of Orca, and it's fed each `DeployableService`'s deployment pipelines



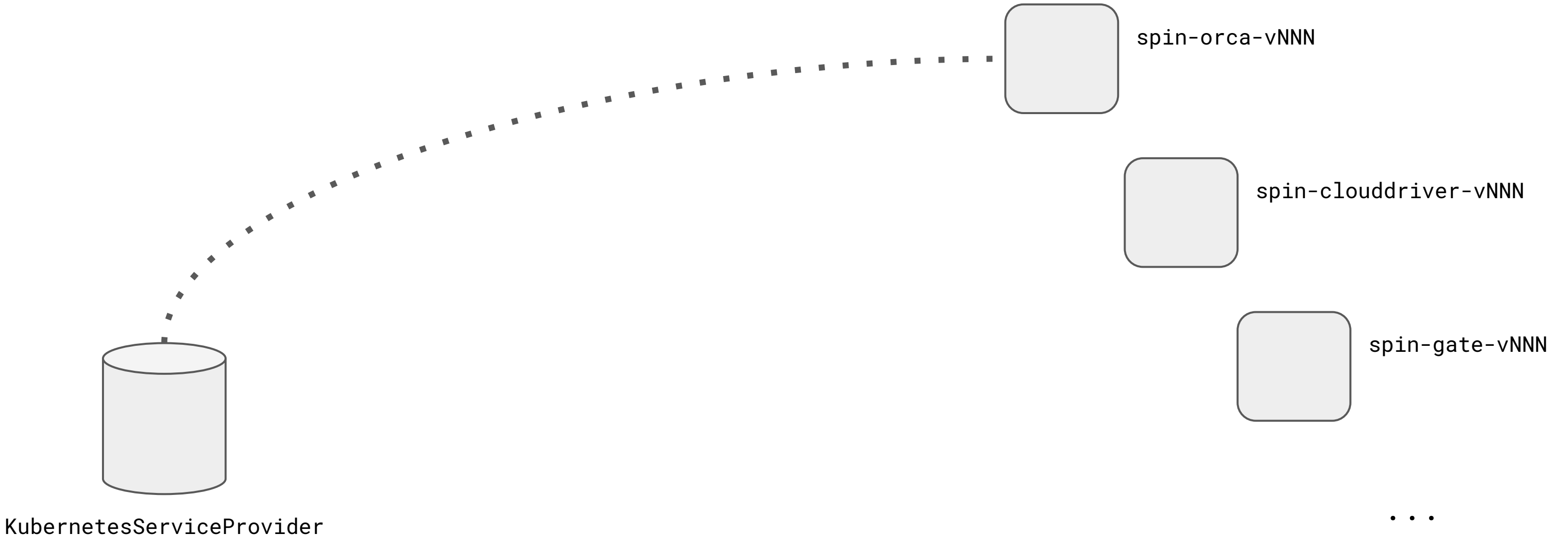
Deployment Procedure

1. Load balancers, and services marked “not safe to update” (e.g. redis) are left alone if they’re already running



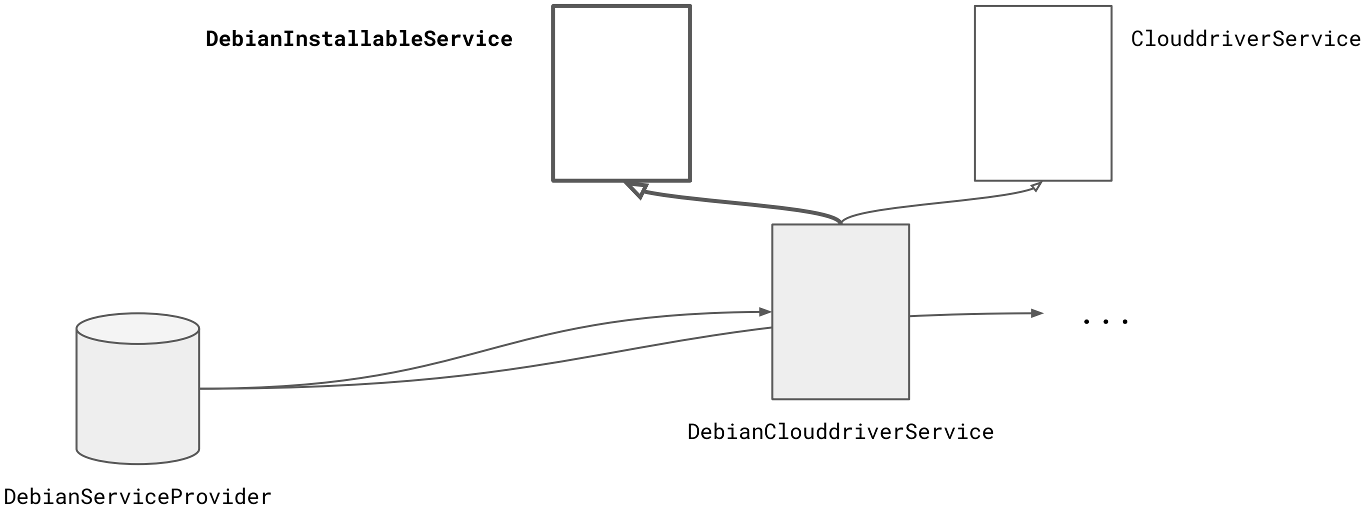
Deployment Procedure

1. Finally, the bootstrap environment is torn down
2. Orca is interrogated for instances no longer running pipelines, which are then pruned



Installable Services

1. `InstallableService`s build commands (bash) to pin & install artifacts
2. Installation involves aggregating these commands and having the client run them



Install Procedure

1. `ServiceProvider` generates a script to install/update all required services
2. The CLI is handed a `RemoteAction` to run with privilege to install packages
3. This can be done by the Daemon as well, but would require running the Daemon as `root` (seems hacky)