SHARED EXTERNALLY

Spinnaker CLI Design

Author: jacobkiefer@
Last Updated: 2018-07-18

Motivation

Spinnaker provides an API for any authenticated user to call via Gate. Currently consumers can hand-roll scripts calling Spinnaker's API; however, writing such scripts requires an in-depth understanding of Spinnaker's supported authentication mechanisms in order to call endpoints from scripts. Along with that burden, our API surface grew organically and contains quirks we can fix. We need to make programmatic automation against Spinnaker's API more accessible to consumers and reduce the friction of automating workflows that interact with the API.

Reducing the friction of automating workflows enables implementations that are much easier to maintain for advanced consumers already utilizing such workflows and enables more consumers to implement automated flows where they would have failed previously due to the difficulty threshold. We describe a few specific, motivating use cases in the use case section.

Goal

- Reduce friction of automating workflows that programmatically interact with Spinnaker's API.

Non-goals

- Provide any new authentication methods.
- Lock users in to one templating engine baked into Spinnaker.

Overview

Roles

Quick definition of roles involved in the discussion below:

- Consumer - anybody using or maintaining a Spinnaker deployment. Both "operators" and "end users" fall into this category.
- Operator - person administering and managing a Spinnaker instance, with root access (e.g. SRE).
- End user - line engineer using the Spinnaker instance to deploy software.

Relevant Prior Tooling

Generally we want to address the difficulty of the programmatic automation gap for consumers of Spinnaker. A few CLI tools already exist for consumers to modify Spinnaker application configuration (roer, foremast, halyard).

Halyard deals with configuring and managing the Spinnaker services themselves, which is a separate concern than the motivation of this design doc, so we will elide it from the overview discussion in this section. We will, however, discuss considerations to address the motivating concerns within Halyard in the design section.

Roer and Foremast were designed to provide programmatic interfaces to the Spinnaker, and hence are in scope for this overview.

Roer is housed under the spinnaker org in github, and deals strictly with manipulating pipeline templates and configurations as defined in the spec. Roer is effectively in a “dead alpha” state, which means development has stopped on it. It includes an opinionated templating solution based on jinja with server-side hooks for template hydration based on execution history and other contextual information. Roer supports x509 as its only auth mechanism.

Foremast is a tool open-sourced by GogoAir and hosted in github. It supports managing pipelines with some templating capabilities, as well as managing relatively static infrastructure declaratively by directly interacting with the AWS APIs. Foremast supports x509 and gitlab tokens (example configurations, only mention of auth in their docs).

All other programmatic solutions are generally one-offs for specific internal use cases. These largely are not open sourced and not useful for the OSS community (for instance, internal custom scripts that initialize a new application for one specific provider).

We propose creating a new CLI surface ‘spin’ to address our main goal of reducing the friction of automation against Spinnaker’s API. This doc contains a discussion of design decisions and tradeoffs.

Use Cases

Our primary focus is reducing the difficulty of programmatic access to Spinnaker’s API. There are a few concrete, major user flows that would benefit from this work immediately.

Flows

There are three major user flows a spin would address.

1. Programmatically manage and share pipeline across teams

Several companies (e.g. Spotify, Datadog, Nest, Waze, Cognite) are building out CI pipelines to share “best practice” Spinnaker application config and pipelines with the development teams they are serving. These pipelines interact with a production-grade authenticated Spinnaker.

Example use case - manage/update a set of pipelines shared across several teams or applications as an operator:

- Store in a private git repo a set of jsonnet files that generate a set of pipelines.
- Configure an automated TravisCI job to trigger on a commit to the git repo.
- When the job triggers, clone the repo, use jsonnet to generate pipeline json.
- Pipe each generated pipeline json to the authenticated spinnaker CLI to update or save the newly generated pipelines in Spinnaker. The generated pipeline json should contain an application and id to locate which pipeline to update. For a given application/id pair, we can fetch the pipeline config and diff against the payload to 1. fingerprint for optimistic locking and 2. prevent needless writes.

Example use case - onboard a new dev team and create a new application

- Store a jsonnet file that generates an application given a set of parameters in a Jenkins job.
- When approached by a new team, manually trigger the Jenkins job, parameterized with the team’s Spinnaker application name and and config values to generate the application and a few sample pipelines to start the new team off.

2. DCD/MPT user support and migration (dcd spec/roer)

A few companies (Waze, Nest, Spotify, etc) are already relying on Roer and managed pipeline templates to implement (1) in the absence of easy-to-use tooling in this space. This is problematic since companies are investing time and work into automating these pipeline templates even though Roer and the server-side DCD component in Orca are stagnant.

Example use case - migrating pipeline definitions from DCD to spin

- Start with a set of pipeline templates and configs used by roer in some process to update pipelines.
- Suppose the flow is `roer publish` > `roer save`
- Use spin (our CLI) and change the flow to `roer publish` > `roer plan | spin pipeline save` as a workaround.
- In the meantime, work on a separate templating solution to formulate the pipelines `spin pipeline save` expects via a different templating solution.
- Drop in the templating solution for `roer plan` in the process.

3. Tutorial/codelab -- programmatically get into a usable Spinnaker state quickly

Currently many of our Spinnaker codelabs have a “point and click” pipeline setup. This is a source of friction and is our main feedback when giving demos internally. Generally there is no auth enabled in Spinnaker and desired state of Spinnaker (e.g. one application, one pipeline) is quite simple; yet there is no way (outside of curl and knowing the exact Gate API surface) to easily automate such a simple state in Spinnaker.

Example use case - spin up a fresh Spinnaker for a tutorial:

- Create a fresh VM and install Halyard.
- Use Halyard to install and configure Spinnaker on the VM.
- Install spin on the VM.
- Create a new application via spin.
- Curl a sample pipeline and pipe it to spin to save the pipeline in your new Spinnaker application.
- Have a fresh Spinnaker application with basic pipelines to start from.

Design

CLI Tool

Why a CLI

The first discussion is why we should have a CLI at all. Gate exposes a sufficient API interface for consumers to call, so why do we need to do any work?

Proposed solution: codify critical user flows into a CLI
There are several critical user flows (defined in use cases) with no golden path solutions provided by Spinnaker and/or the surrounding tooling. Roer is the closest facsimile to a solution in this space and is not being developed or maintained. When (many) users ask us for a good programmatic and automatable solution, we currently have no answer that isn’t “curl” or “roer”.

A new CLI surface gives us the opportunity to center our design focus around satisfying the use cases we’re missing stories around and make it **easy** for consumers to craft solutions to the problems they’re having. We’ll do this by making auth simple and providing a useful and intuitive set of CLI operations that consumers are asking for.

Alternative: roer
Why can’t we just use roer?

Roer does not completely address the use cases, and would need expansion to satisfy what is missing. It was originally scoped strictly to interact with pipeline templates and configurations

only, with no plans to expand to a larger surface area. We'd inherit technical debt from those design decisions.

Roer also introduces an opinionated and non-standard templating engine with lots of edge cases. The templating engine is server-side in Orca and has introduced quite a bit of complexity in the code, while also locking users into one specific templating mechanism. There are many templating solutions, and parts of our user base (consumers using Kubernetes) are opinionated about the types of templating tools they want to use.

If we break away from roer's model, we can provide flexibility to consumers (they can use any templating engine they want) while reducing the complexity of our code base. We can do better.

Alternative: curl
Consumers can use curl to hit Gate's API; however, curl does not easily handle auth without introducing complexity into the consumer's CI pipelines. We could introduce documentation describing how to do this; while that may be sufficient for expert-level consumers, this would not address the difficulty level of automating interactions with Spinnaker's API. We can do better.

Which CLI Surface

We should think carefully about what the physical incarnation our CLI surface takes. It should satisfy our design motivation and also be minimally disruptive to the current community mindset.

Proposed solution: an additional CLI interface -- spin
Create a functionally and conceptually separate and distinct CLI, complete with its own repo, lifecycle, etc.

Good:
- Conceptually clean -- spin would target "end user" role only.
- No inherited technical debt from existing CLI opinions and use cases.

Bad:
- Introduces another CLI tool.

Alternative: roll functionality under Halyard
Expand Halyard's scope to include all the necessary functionality specified in this design.

Good:
- No new CLI surface.

Bad:
- Conflates responsibilities of operators and end users to roll functionality under Halyard.
- Halyard has no access control model and requires root access to Spinnaker - this would either require a rework of auth in Halyard or in Spinnaker to satisfy this use case.
- Adds complexity to Halyard.

Alternative: expand roer
Expand Roer's scope to include all the necessary functionality specified in this design.

Good:
- No new CLI surface.
- Small portion of reusable code.

Bad:
- Technical debt from experimental design in roer, would require a restructuring/rework of many primitives.
- Conceptually confusing to community -- roer was meant to support managed pipeline templates, which are being deprecated. In addition, roer is already being used to support that case for quite a few consumers.

Authentication

There are two "access types" we care about: interactive and non-interactive.
- interactive - the user is present and interacting directly with the CLI. We can initiate an auth flow if needed to refresh the user's session with Gate, and we can ask the user for input.
- non-interactive - the user is not present. If we need to refresh the user's session with Gate, we need to be able to do so without a user present. This is crucial for automating workflows with spin.

Authentication Config File

We should use a static auth config file to define necessary details for the support authentication methods in the CLI. We'll include the method-specific details of the configuration file in each section. We'll format the configuration file as yaml to keep in step with the rest of the Spinnaker universe.

General config file structure:

```
# ~/.spin/config

auth:
  enabled: true
  # Authentication-method specific details follow here.
  # We'll define this per method later in the doc.
  ...
```

Authentication Methods

There are four methods of authentication in Spinnaker currently:

- OAuth 2.0
- SAML
- LDAP
- X.509

We will support OAuth 2.0 and X.509 initially, as they have natural and standardized CLI flows available. We'll detail more about the CLI interactions with those two authentication methods in those sections.

We won't support SAML and LDAP authentication from the CLI; however, we can support manual session management, similar to roer as a workaround.

OAuth 2.0

Proposed solution: negotiate OAuth tokens from the CLI rather than Gate, provide Gate a Bearer token as a header.

If the CLI negotiates the OAuth tokens, we can support both the interactive and non-interactive use cases by caching the access and refresh tokens on the machine where the CLI lives. If and when the access token expires, we can immediately refresh it with the refresh token without the user being present.

Good:
- Supports automated non-interactive flows, which is our critical use case.

Bad:
- Requires CLI logic to handle flow.

Alternative: let Gate negotiate OAuth tokens and manage CLI authentication with (expirable) session cookies.

Another way to handle authentication is to delegate to Gate to perform the authentication and establish a session cookie with the CLI. Session cookies expire similar to access tokens, but provide no means to non-interactively refresh -- the user must re-authenticate via Gate to establish a new authentication session.

Good:
- Requires less CLI logic (maybe) than negotiating tokens from the CLI.

Bad:
- Requires a user to be present (interactive access) which blocks automated flows.

We should support both 3-legged (3LO) and 2-legged JWT token (2LO) OAuth 2.0 flows, as they are the two standard flows in the protocol and we can support both interactive and non-interactive access for our CLI. Our primary use case is automation (2-legged makes the

most sense here), but we want to also support interactive user interaction as well (3-legged makes sense here). Fortunately golang has an excellent OAuth 2.0 extension package to simplify the CLI logic, complete with several provider-specific flow helpers (e.g. Google service accounts).

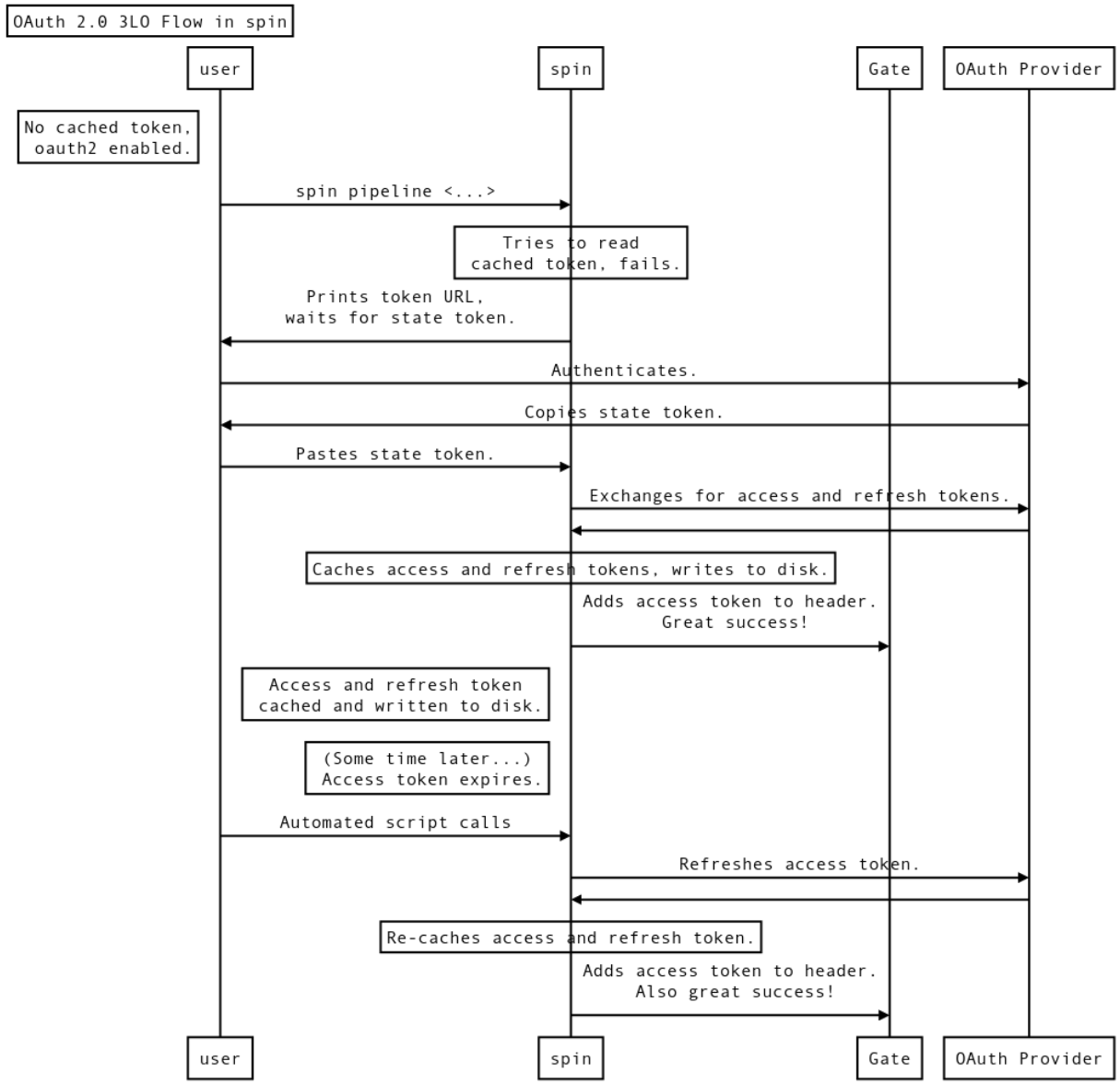
3LO

Configuration block:

```
# ~/.spin/config

auth:
  enabled: true
  oauth2:
    # generic 3LO (struct)
    tokenUrl:
    authUrl:
    clientId:
    clientSecret:
    scopes:
```

An access token (with an expiry) and a refresh token will be negotiated in the 3LO flow. In order to do the non-interactive token refresh, we'll cache the negotiated access and refresh tokens in `~/.spin/credentials`. Here is the flow diagram:



2LO

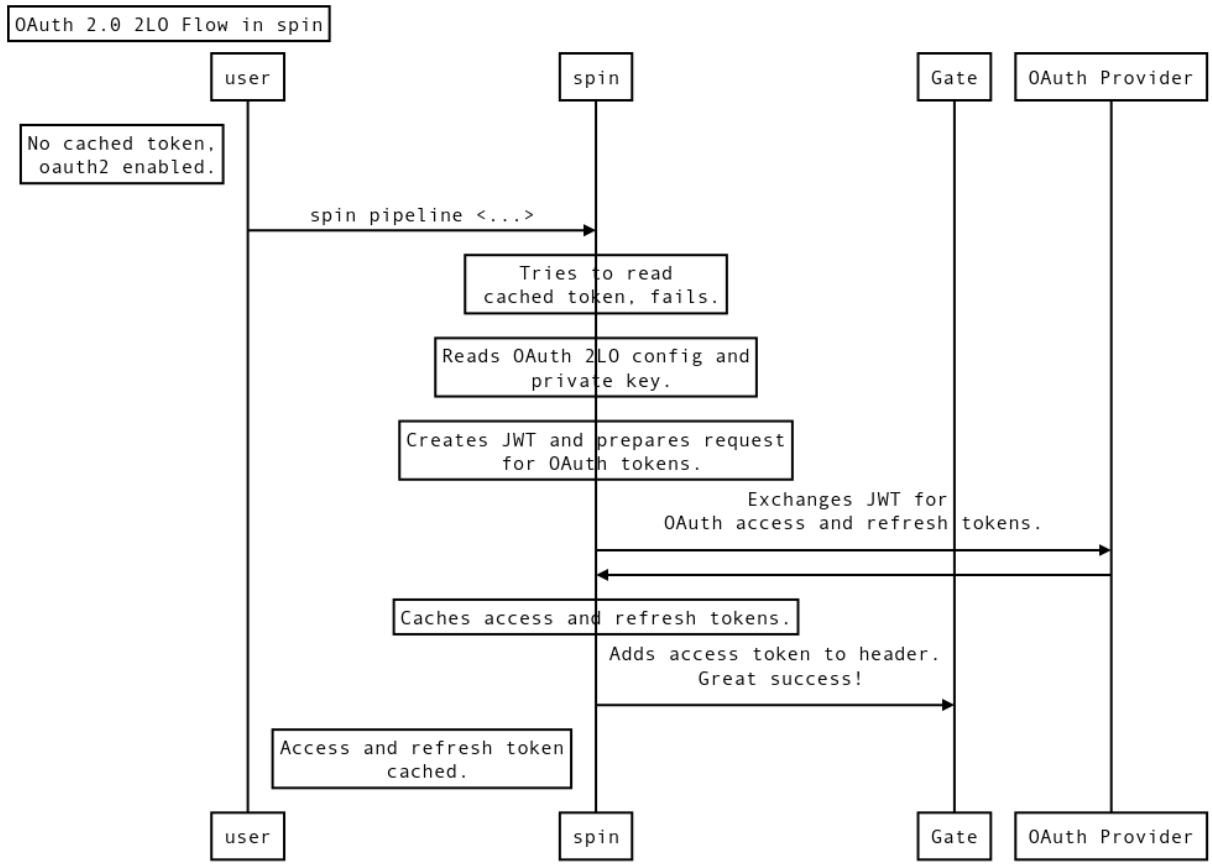
Configuration block:

```
# ~/.spin/config

auth:
  enabled: true
  oauth2:
    # generic 2LO (struct)
    email:
    privateKey:
```

```
    subject:
    tokenUrl:
```

Flow:



Google Service Accounts (Bonus)

Note that the config blocks for 3LO and 2LO above are generic. We'll start with this as a catch-all and specialize the configuration as we need to support more use cases. As an example, OAuth with a Google service account json key might look like this:

```
# ~/.spin/config

auth:
  enabled: true
  googleServiceAccount:
    jsonPath: # Path to service account key.
```

The auth flow is identical to 2LO save the configuration and exchange request content, so we will omit another flow diagram.

X.509

X.509 client certs are fairly simple to add to HTTPS calls. Similar to roer, we just need a keyfile and certfile to add to the TLS client config. Note that SAML and LDAP deployments can add X.509 to support authenticated access from our CLI. This is inherently “non-interactive access” and is easily automatable with one-shot configuration.

Configuration block:

```
# ~/.spin/config

auth:
  enabled: true
  x509:
    certPath: # Path to certfile
    keyPath: # Path to keyfile
    # OR
    cert: # base64 encoded cert
    key:  # base64 encoded key
```

API Surface

Initial CLI Surface

The following is a proposal of a basic set of API operations to support initially, as this is where we’re seeing the most demand.

- Applications
  - save
    - `spin applications save --file <app_config>`
    - `POST /tasks <body>`
    - File payload contains name
  - list
    - `spin applications list`
    - `GET /applications`
  - get
    - `spin applications get --name`
    - `GET /applications/{name}`
  - delete
    - `spin applications delete --name`
    - `DELETE /tasks <body>`
- Pipelines
  - save
    - `spin pipelines save --file <json_file>`

        - `POST /pipelines <body>`
        - Should also accept stdin
        - File should contain pipeline spec, name, application, and id
        - Can accept the JSON currently returned by the API.
    - list
        - `spin pipelines list --application`
        - `GET /applications/{application}/pipelines`
    - get
        - `spin pipelines get --application --name`
        - `GET /applications/{application}/pipelineConfigs/{name}`
    - delete
        - `spin pipelines delete --application --name`
        - `DELETE /pipelines/{application}/{name}`
    - execute
        - `spin pipelines execute --application --name`
        - `POST /pipelines/{application}/{name}`

Auto generated client

We should consider whether to auto generate the library we use to interact with Gate. Things to keep in mind:

1. Initial CLI surface - this is quite small.
2. Overhead - auto gen requires tooling.

Proposed solution: Use swagger codegen to generate a client library to interact with Gate

The 'swagger-codegen' tool takes a swagger spec (json file), which is generated from hitting a swagger endpoint on Gate, e.g. `curl <gate host>:8084/v2/api-docs`. Given the json swagger spec, swagger-codegen can generate a client in any language it supports (many). We will generate the swagger spec in Gate's gradle build and store it Gate's git repository so anyone can consume the swagger spec. This CLI will generate a client library and use that as the Gate client.

We'll need some tooling to support this:

1. A gradle task to generate the spec.
2. A Github bot to ensure the spec is up to date with the commits in Gate. If the spec doesn't match the API, block the PR merge.
3. Release process tooling to generate the client library and test the CLI interactions.

Good:

- No ambiguity or drift from Gate's API, as future-proof as we can make the Gate API interactions.

Bad:

- Higher overhead - need tooling to automate the auto generation.

Alternative: Hand-roll a thin Gate client.

Good:

- Low overhead initially - no extra processes necessary, golang has great support for HTTP clients (language is discussed in its own section).

Bad:

- Interacting with a new controller requires a small amount of new code.
- Model changes require a refactor.

API Versioning

Regardless of whether Spinnaker is managed by Halyard or not, we should be able to determine a CLI <-> API version mismatch. Gate follows semantic versioning, which is helpful. We can state a Gate version in the CLI that it is compatible with. Gate should report a version - if the reported Gate version differs from the CLI's Gate version by a major, we request the user to upgrade.

Templating & Backwards Compatibility

There are quite a few templating solutions available that address a wide range of concerns. Forcing consumers into one particular opinion restricts how flexible the tool is for different use cases (e.g. jinja + SPEL in roer). A subset of our consumers are already templating resources in some capacity (Kubernetes) and are opinionated about templating. Hence, we are intentionally omitting templating capabilities in the CLI design. Moreover, we will provide a bridge for consumers to migrate off roer so that we can eventually remove the pipeline templating code from Orca.

Consumers will be able to migrate from roer and pipeline templates by generating hydrated pipelines with roer and piping those into spin: `roer plan config.yml | spin pipeline save`. With the previous flow as an interim solution, consumers can figure out a templating solution that fits their needs best without being locked in to using roer-flavored Jinja templating.

To provide sufficient backwards compatibility, we would take the mantle of stabilizing roer and the pipeline templating code in Orca to provide a final distribution to migrate consumers to spin.

Implementation/Distribution

Language

Proposed solution: Implemented in golang

May need to distribute spin's auth package to use in roer to bridge the gap. Turns out this is quite straightforward in golang -- just depend on the spin/auth package in roer, e.g. this usage.

Good:

- Portable to relevant platforms (linux flavors, os x).
- Great CLI and HTTP support.
- Can share auth libraries with roer.

Bad:

- ?

All alternatives (java, python) can't support roer in the manner described.

Distribution

Distributed as a Go binary as well as a containerized binary (like hal) to run on k8s. This enables containerized task runners (GCB) to run spin. spin will be dependent on Gate's API versions, so it should be included in the release BOMs. The CLI version in the BOM should match the top-level Spinnaker version (like kubectl and the kubernetes API) for easy usability.