Automated Application Rollback Insurance for Release Teams

07/2020

Written by: Eric Calabretta

Summary

System failures are an unfortunate fact of life in the new digital first world. Automated rollback and rollfoward enable application teams to quickly recover from a failure and minimize any resulting business impacts. This paper provides an overview of rollback vs. rollforward and how to implement within Chef Habitat.

Contents

Perspective

Sometimes in IT certain terms take on a life of their own. They push past their original meaning and become something different, rollback is one of these terms.

In the language of enterprise IT, "rollback" means getting the system back into a working state. This ensures a system can be immediately restored if a system failure occurs and that there are no disruptions to business. I've heard "rollback" from CIOs, Directors of Operations, and VPs of Development. When discussed we weren't comparing the technical implications of rollback vs. other techniques. What they were really asking is "what type of insurance policy does Chef provide to get the system back into a working state after an issue has been detected?"

With the criticality of IT systems and the pressure for speed, companies need an insurance policy for when things go wrong. They need to be able to quickly restore service after a failed change or a failed release.

Rollforward vs. Rollback

Rollforward is required for application types where returning to a previous version may be destructive making Rollback impossible. The process would look something like this:

Start running version A – > upgrade to B -> detect failure -> correct failure in new version C -> upgrade to C

Your application may have a breaking change, or commonly databases may have a schema change making rollback destructive. As an example with SQL upgrades should be additive, it is not safe to return to a previous version of the application. To restore service after a failed deployment the failure must be diagnosed and corrected in a new version.

Rollback is a good choice if you can return to a previous working version of your application. The process would look something like this:

Start running version A – > upgrade to B -> detect failure -> return to version A

This can be advantageous for applications where reverting to a previous version is sufficient to restore service. As an example with a Java/Tomcat application this could be as simple as removing your failed WAR file(Web ARchive) and redeploying the previous working WAR file.

The application architecture will dictate if Rollforward is required or if you can situationally choose to Rollback or Rollforward.

3 Step Plan for Implementing Rollback

Step 1: Plan for Failure

The time to figure out how to respond to a failed release isn't after the release has failed. We need to plan for failure and practice responding to failure. In order to minimize risks and ensure application availability every application delivery plan needs to include a recovery failure methodology and testing for that methodology ahead of time.

"Plans are useless, but planning is indispensable" – Dwight D. Eisenhower

Step 2: Decouple the Application from Supporting Components

Today's modern applications are built upon an interconnected web of components. By decoupling the application from the various supporting components and providing a clean contract between the application and the components we enable a more manageable rollback scenario. Can you rollback an individual component or do you need to rollback a dozen components in a specific order? Are the functions & expectations between each component well established? Do you understand what a deployment or rollback to component A does to component B? By answering these questions ahead of time and decoupling the application from dependencies enables us to avoid "big bang" deployments, keep releases as small as possible, along with keeping rollbacks as small as possible.

Step 3: Select the right technology

The capabilities of your application delivery solution matter when it comes time to roll-back. For rollback to be a real insurance policy you need to be confident it'll work when you need it.

It's not enough for your solution to simply revert to the previous version of your application, what counts is that you can quickly restore service. This may actually require you to rollforward depending on your application architecture.

Capabilities needed for rapid service restoration:

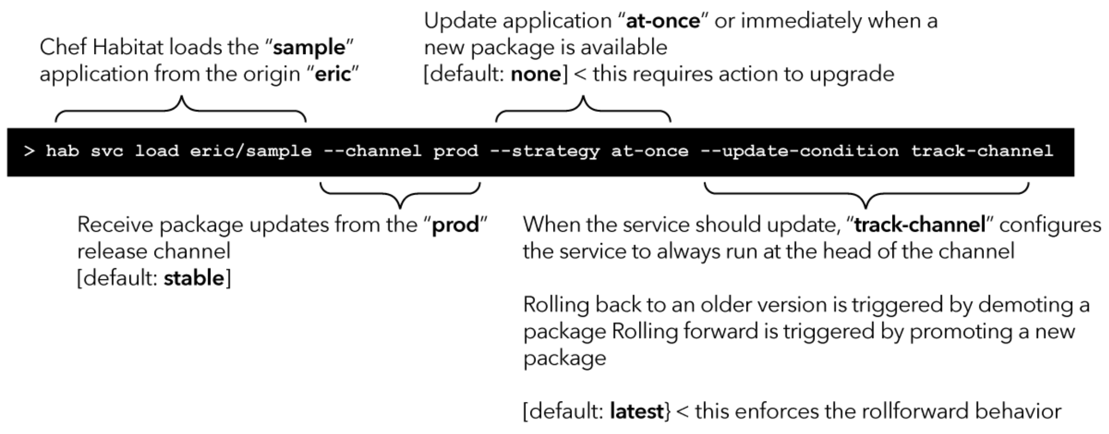
- Intelligent agent
- Pull model with fast deploys
- Native health validation
- Immutable Artifacts
- Isolated application & dependencies
- Simplified CI/CD

Introduction to Chef Habitat

Chef Habitat is an open-source automation solution that enables you to define, package and deliver your applications to any environment without rewriting or refactoring. The software creates platform-independent build artifacts that are deployed and managed in the same way.

Application definition is the process of creating a codified operational runbook. It formalizes the process of describing in code everything an application needs to be built, run, and managed. As part of the definition process you have the opportunity to specify what runtime services you want to use. When you load a runtime service with Chef Habitat you select the appropriate behavior for your application. This includes the desired upgrade behavior like if your application is compatible with "rollback" or if only rollforward behavior is acceptable.

In this example we tell the Habitat Supervisor to load the "sample" application, to receive updates from the "prod" release channel, upgrade immediately, and finally that it can "rollback" with a demote or roll-forward with a "promote". The Chef Habitat Supervisor is a light-weight agent that runs on/in a server, virtual machine, or container and manages the application according to the instructions defined in the Habitat Plan. Tasks are defined via pre-set scripts called lifecycle hooks that are included as part of the application plan.



Chef Habitat loads the "**sample**" application from the origin "**eric**"

Update application "**at-once**" or immediately when a new package is available
[default: **none**] < this requires action to upgrade

```
> hab svc load eric/sample --channel prod --strategy at-once --update-condition track-channel
```

Receive package updates from the "**prod**" release channel
[default: **stable**]

When the service should update, "**track-channel**" configures the service to always run at the head of the channel

Rolling back to an older version is triggered by demoting a package Rolling forward is triggered by promoting a new package

[default: **latest**} < this enforces the rollforward behavior

With Chef Habitat you define everything you need to deliver your application as code, then you package your application into a Habitat Artifact(.hart). This artifact includes everything your application needs to deliver & run like the application codebase, any dependencies, and instructions on how to run the application.

The created Habitat Artifact is immutable, it includes your application, locked dependencies, the definition of health status for your application along with anything else your application may need.

When you load an application the Habitat Supervisor downloads the Habitat Artifact for your application from Builder, verifies the package then unpacks to an isolated place on the file system.

For the example above my application would be stored on a path similar to this:

```
/hab/pkgs/eric/sample/0.2.0/20200309020327`
```

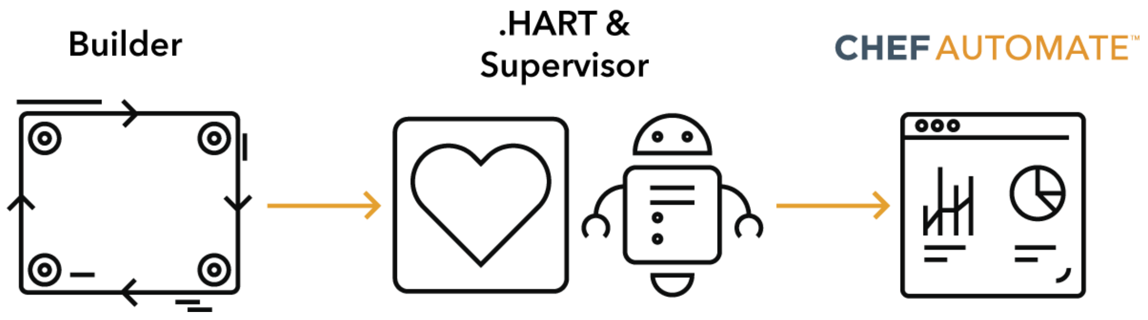
0.2.0 is the Applications version and 20200309020327 is the applications build number.

Finally after the application is unpacked the Habitat Supervisor runs your application as you defined earlier. This is typically in your run hook which is a simple bash or powershell script for how your application should be run.

The running application will be found under the svc or services path, which would be something something similar to this:

```
/hab/svc/sample
```

After the Habitat Supervisor starts your application, the supervisor will report the health of the application to the Chef Automate dashboard. Here's a visual showing the Habitat Supervisor downloading the Habitat Artifact from Builder, then reporting health status to Chef Automate.



Upgrading an Application with Chef Habitat

Now that we know how the Habitat Supervisor is used to run your application, let's review how the Habitat Supervisor upgrades your application.

By default the Supervisor will check-in with Habitat Builder every 60 seconds looking for a new Habitat Artifact. When a new Habitat Artifact is detected the Habitat Supervisor downloads, verifies and unpacks the content to:

```
 hab/pkg/applicationname/version/buildnumber
```

The Habitat Supervisor will also download and verify any application dependencies you've defined.

Once the Habitat Supervisor has put everything in place for your upgrade, it will stop the old version of your application, and start the service for the new version of your application.

Use the available hooks to perform any actions your application needs to clean up after itself, prepare the new version and run the new version. Commonly the `init` hook is used to prepare for your application like cleaning out old versions, setting environment variables or other tasks, then the run hook is responsible for starting your application.

Finally after your new application version has been started the `health-check` hook is used to validate the health of your application. By Default the Habitat Supervisor will run the `health-check` hook every 30 seconds. The results can be sent to Chef Automate to visualize a fleet of applications.

Automated Application Rollback with Chef Habitat

To implement rollback with Chef Habitat we start with our supervisor running Version A of our application. We promote a Version B of our application to the "prod" channel. The Habitat Supervisor will see the new version. It will then immediately begin the upgrade process. Once the upgrade is complete the Habitat Supervisor will start our `health-check` hook, which will detect the deployment failure.

```
hab pkg promote eric/sample/0.2.0/20200309020327 prod
```

To Rollback we will demote version B, which removes it from the "prod" channel. The Habitat Supervisor will see this demotion, it will then immediately revert back to Version A of our application. This is the exact same process we described in the upgrading section. Once the Rollback is complete the Habitat Supervisor will start our `health-check` which will validate service has been restored.

```
hab pkg demote eric/sample/0.2.0/20200309020327 prod
```

Automated Application Rollforward with Chef Habitat

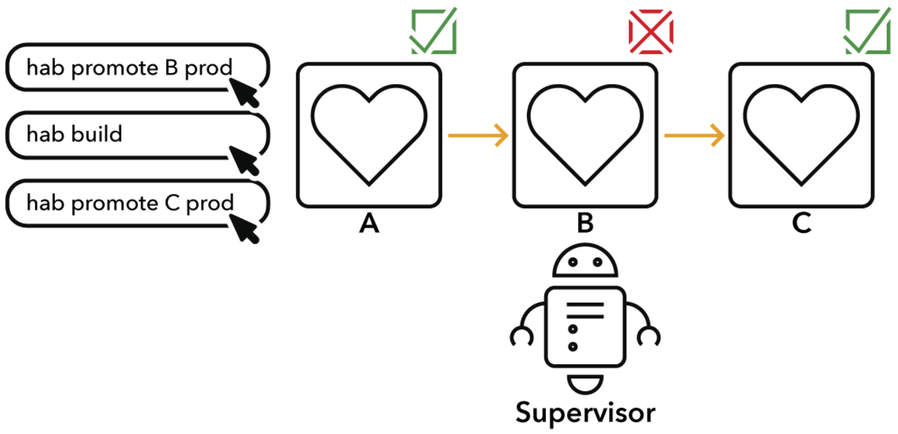
In a rollforward scenario we use Chef Habitat's upgrade capabilities slightly differently, with the same goal of quickly restoring services in the event of a failed deployment. Our application is either incompatible with rollback or we decide we want to continue forward with the new features found in our release.

We start with our supervisor running Version A of our application. We promote a Version B of our application to the "prod" channel. The Habitat Supervisor will see the new version, it will then immediately begin the upgrade process. After the upgrade the Habitat Supervisor will restart the health-check where we can see our release has failed.

We need to diagnose what caused Version B of our application to fail and correct the issue. Once we find the commit that caused the problem we can make the appropriate changes to resolve the issue. This may require us to add an additional commit or git revert the problem commit.

Whatever we decide we create a new pull request with our fixes. Our pull request should run our normal pipeline which will create a new build and run through our standard testing.

We now have a new tested version of our Application version C. We will promote version C to the "prod" channel. The Habitat Supervisor will see this promotion, it will then immediately rollforward to version C of our application. After the rollforward action is complete the Habitat Supervisor will restart our health-check which will validate service has been restored.



Conclusion

The promote or demote actions can be performed via the command line or via the habitat builder user interface bldr.habitat.sh. The cli is also a logical integration point with pipeline tools like Azure DevOps, Jenkins, Gitlab, etc.

Over time you may perform hundreds of releases with Chef Habitat. This can start to fill up your disk with unnecessary old Habitat Artifacts. You can configure the Habitat Supervisor to automatically clean up old packages with the `--keep-latest-packages` flag. The Habitat Supervisor will clean up old packages only keeping the number of packages specified.

For example we can configure the Habitat Supervisor to keep the most recent three packages with `--keep-latest-packages=3`.

Technically how you “rollback” will depend on your application architecture, your constraints, and your processes. In some scenarios Rollback will be the most desirable and in others Rollforward will be the most appropriate.

What we do know is that failures will happen and when they do Chef Habitat has you covered with an insurance policy for any architecture.

Learn more about Chef Habitat here. Or if you would like to schedule a demo contact us.