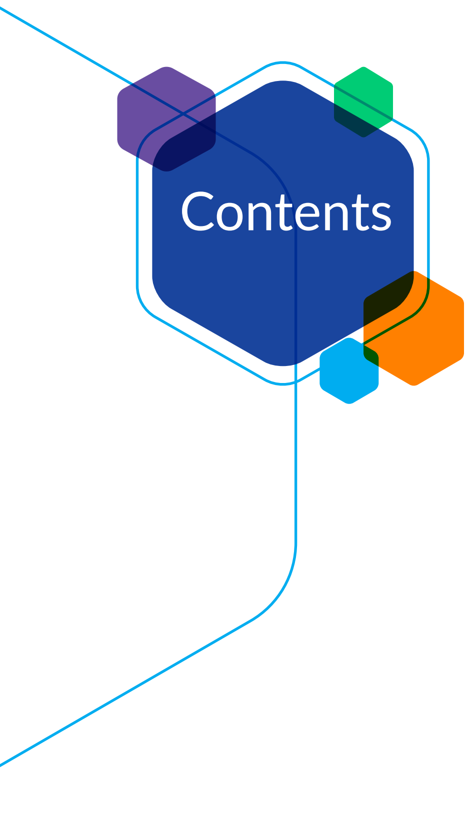
How CloudBees Uses Feature Flags to Gain a Competitive Edge



3 Introduction

Using CloudBees Feature Management to Increase Our Own Developers' Productivity

4 How we used CloudBees Feature Flags for feature control

8 Rapid Deployment of New Subscription Billing with Flag-Driven Development

15 Using Feature Flags to Validate the Public Rest API

Improving Customer Experience Using CloudBees Feature Management

16 Launching the new CloudBees.com wasn't a risky proposition using feature flags

19 CloudBees Feature Management for hiding billing details from certain customers

22 Conclusion

Introduction

Ok, are you ready to hear all of the secrets of why CloudBees has grown to be a leader in the DevOps market for the last 10 years? For the sake of not making our bosses really mad, we'll have to keep some things to ourselves, but we're happy to share one area that has heavily contributed to our success: our usage of feature flags internally. Our adoption of a "feature flag-driven development culture" has permanently changed the way we develop, experiment with and release new features in our products.

While it's no secret that we are a big fan of flagging features given the clear benefits of reduced deployment risk, shortened feature feedback loops and faster releases, it's quite another thing for us to show you this in action versus handing you another brochure with a long list of features and benefits that sound too good to be true.

So, here we go! Learn how some of the best and brightest engineers at CloudBees "eat their own dogfood," "drink their own champagne" or, quite simply, flag their own features. This whitepaper contains use cases centered around improving developer productivity and creating better experiences for customers through the use of feature flags, written by the developers and architects who use the flags everyday themselves. We hope these use cases inspire you to imagine new possibilities for feature flags and inspire you to begin creating your own culture of feature flag-driven development!

Using CloudBees Feature Management to Increase Our Own Developers' Productivity

How we used CloudBees Feature Management for feature control

"CloudBees Feature Management has played key a part in helping us experiment with skunkworks projects"

Pete Muir,
Lead Solution Architect

CloudBees Feature Management has played a critical role in allowing our research and development teams to innovate freely with little risk. Some of these "skunkwork" projects end up being future projects and others lend various lessons or new features to our current product base.

In one case, we used feature flags to help us determine the viability of a cloud native CI/CD solution. Our team was heavily inspired by the State of Devops Report and the Accelerate book, both of which advocate short-lived branches as a characteristic of high performing teams (something that is definitely borne out by experience). This practice means that we used different branches for different versions of the CI/CD platform that we sought to develop, one for a potential open source version and one proprietary enterprise version. Feature flags are a natural solution here as they allow you to continuously merge new features into master, whilst keeping them hidden behind a feature flag. You can then enable the feature(s) for certain users when you are ready.

We were fortunate to have access to CloudBees Feature Management and it worked out really well for us. In our React web UI, we take advantage of the CloudBees Feature Management Javascript library which allows us to load the feature flags, with default values specified in the code.

As we are using functional components in React, we can store the feature flags in a react context:

```
export function createFeatureFlags(): FeatureFlags {
   return {


       devPod: new Rox.Flag(),
       environmentActions: new Rox.Flag(),
       loggedInUserFeature: new Rox.Flag()
   };
}


export const FeatureFlags = createContext(createFeatureFlags());
```

and access them as needed. Using the feature flag is as simple as an if statement:

```
const { environmentActions } = useContext(FeatureFlags);


if (environmentActions.isEnabled()) {
   // Dosomething
}
```

Now we can control whether the environment actions are enabled or not from the CloudBees Feature Management UI.

CloudBees Feature Management's main control grouping feature is the "environment" and you use a different key to address each environment. In order to allow us to change the key without rebuilding the UI containers, we added a REST endpoint to the UI backend that exposes the key. We load that key from an environment variable that is set on the container by Kubernetes.

So far we have not needed to add any custom properties as simply being able to enable or disable the feature for an environment has been sufficient.

How we used CloudBees Feature Management to validate new configurations

We quickly ran into a second challenge – how can we easily enable developers to test out new configurations in our playground, without impacting the entire team if the new configuration is not correct? We needed a way to change the configuration for a specific user - or group of users - whilst keeping the existing, tested, configuration for the rest of the team.

Our potential cloud native offering was is built around GitOps, which allows us to version and audit all of our configuration. The classic way to test out a configuration change in our "playground" is to switch from using the upstream master branch of the git repo used for GitOps to your personal fork. CloudBees Feature Management was very helpful here as it allowed us to override the default GitOps repo with our personal forks without having to build a custom admin UI.

Our backend systems are entirely implemented in Go, and luckily CloudBees Feature Management also provides a Go library. Similar to the frontend, we load the CloudBees Feature Management Environment Key from an environment variable anywhere we want to use CloudBees Feature Management and then use the Go client to access the value of the feature flag. This time we pass in a custom property "email" so we can allow each developer to specify a different value when they log in:

```
roxContext := context.NewContext(map[string]interface{}{
  "email": email,
})



URL := ff.GitURL.GetValue(roxContext)
if URL != "" {
  log.Logger().Infof(" git url is %s", URL)
  url = url + "&URL=" + URL
}
```

We then define an experiment with multiple conditions and allow people to specify their own URL for the git repo to pull the configuration with.

CloudBees Feature Management played a key part in helping us innovate and test this new potential cloud-based CI/CD offering, and we plan to improve the integration between the two to make continuous delivery more powerful and easier than ever.

Rapid Deployment of New Subscription Billing with Flag-Driven Development

> *"Using Flag-driven development (FDD) as a mental model, we can take a complicated feature and break it down into smaller, more functional components"*
>
> Rafi Yagudin,
> Senior Engineer

As software developers, we are tasked with the challenge of telling the future. There should be little to no surprises when we release a new feature, although many times this is not the case. We try to minimize surprises through various practices – discovery, grooming, code review, test coverage, etc. Unfortunately, these take a lot of energy, effort and most importantly, time.

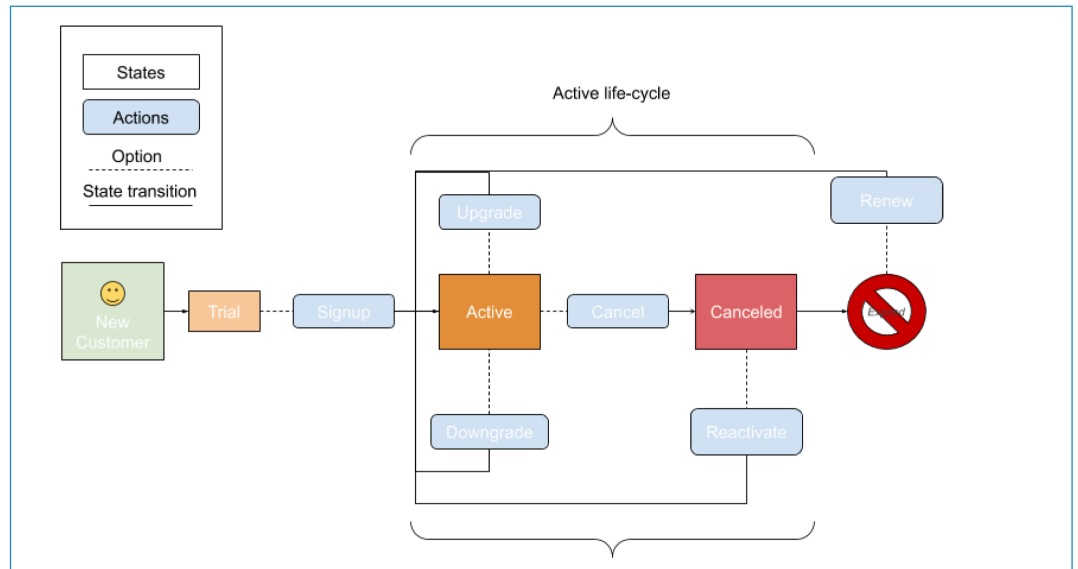
Releasing a new feature can take weeks and has unfortunate side-effects like gigantic feature branches that conflict with your master branch. A worse side-effect: losing a customer because you didn't have a feature ready in time.

In this article, I'll outline how flag-driven development (FDD) can:

- Be a useful mental model to help reduce the time from idea to execution to deployment
- Work very well with features that are defined by lifecycle states that tie to different audiences
- Manifest into rapid deployment of a feature

Subscription billing

More recently at CloudBees, I was tasked with building subscription billing for our customers. The goal was to make touchless subscription sign-ups available as quickly as possible using a Recurly integration. It got me thinking about the scope of work in terms of the life cycle of a subscription. There are different states that define the life of a subscription. Here is the state diagram of the subscription lifecycle in the CloudBees Feature Management platform:

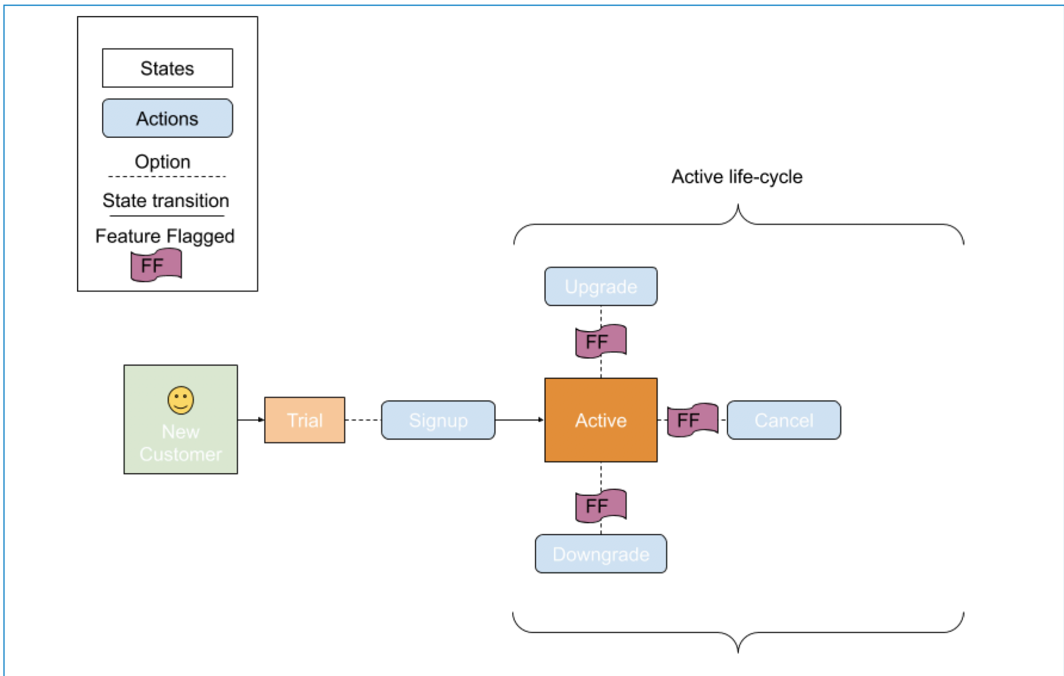


The rectangles are different states and the rounded rectangles are actions that lead to other states (buttons on the UI). The solid lines are state transitions and the dotted lines are options based on the subscription state. In a trial state, the only option you have is signup, which will transition you to an Active state.

The above state diagram represents the full feature of subscription billing. Getting to this point in the real world - through discovery, grooming, testing and perhaps large feature branches - will take time and effort. This entire feature is not conducive to rapid deployment. Breaking it down into smaller, functional components and feature flagging them – as well as avoiding large feature branches - is conducive to rapid deployment.

Using FDD to Re-Define Scope

FDD forces you to break down a feature into smaller, functional pieces that are easier to think about. Let's consider focusing only on the "active life-cycle" portion:



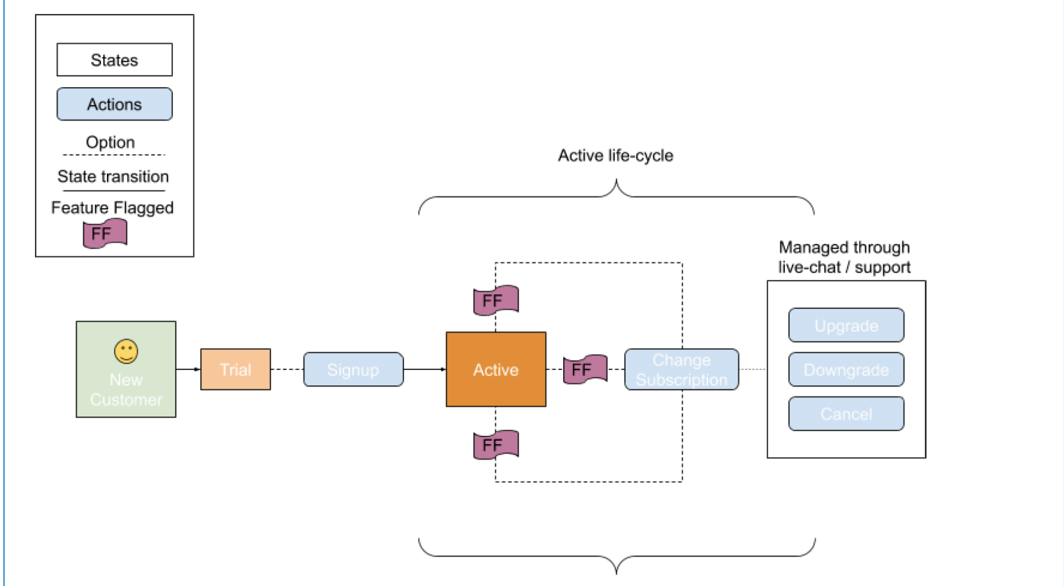
With FDD, we can narrow the scope down to just the active state of a subscription. Canceled, expired, reactivation and renewal are all states and actions that we don't have to think about for now.

However, we still want to give the impression that the user can modify their subscription. This instance is an example of where a base case/fallback method is important to consider.

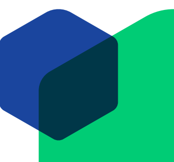
Base Case / Fallback Method

With FDD, you have to think about your various states, but it's important that all states have the option of leading to a base state that is always static. You can think of this as your "fallback method."

With subscription billing, the "base" state for our subscription billing feature was a Zendesk dialogue. The customer would click on "Change Subscription" or "Change Billing" and that would trigger a live-chat session to pop up. They would engage with customer support and support would be able to perform these changes manually through the Recurly dashboard. Let's see what this looks like in our state diagram:



Using our feature flags, we can now divert all active subscriptions to the "base case" which is our live chat session.



Feature Flag Code

Let's see what this looks like in the code using the CloudBees Feature Management SDK:

```
1    const activeSubscriptionFlags = {
2       enableSubscriptionCancellation: new Rox.Flag(false),
3       enableSubscriptionUpgrade: new Rox.Flag(false),
4       enableSubscriptionDowngrade: new Rox.Flag(false)
5    }
6    // Register the flags
7    Object.keys(activeSubscriptionFlags).forEach((key) => Rox.register('MyApp'[c], key))
8
9     // Customer properties allow us to change the behavior of the flag based on the subscription state
10   Rox.setCustomStringProperty('recurlySubscription.state', () => this[d].state.recurlySubscription.state)
```

In the above, we are:

- Defining feature flags for all of our actions associated with the active state of the subscription. We are passing in false, so the flags return false by default, ensuring they will use our fallback method
- Registering the flags with CloudBees Feature Management
- Using a customer property to let our flags know the state of our subscription

Our view code might look something like this:

```
handleCancelClick = () => {
     if (activeSubscriptionOptions.enableSubscriptionCancellation.isEnabled()) {
         // Todo: handle subscription cancellations!
     } else {
          LiveChat('Hello, I would like to cancel my plan.') // Base state
      }
}

handleUpgradeClick = () => {
     if (activeSubscriptionOptions.enableSubscriptionUpgrade.isEnabled()) {
         // Todo: handle subscription upgrades!
     } else {
          LiveChat('Hello, I would like to upgrade my plan.')
     }
}

handleDowngradeClick = () => {
     if (activeSubscriptionOptions.enableSubscriptionDowngrade.isEnabled()) {
         // Todo: handle subscription downgrades!
     } else {
          LiveChat('Hello, I would like to downgrade my plan type.')
     }
}
```

This approach allows us to rollout one feature at a time to subscriptions in the active state.

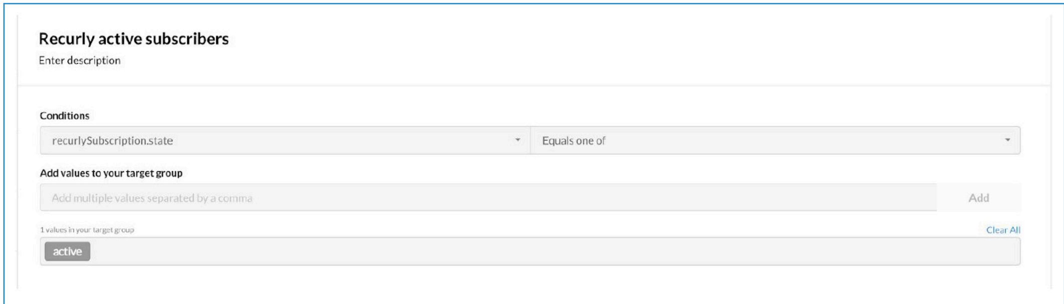
Example: Implementing Cancellation

Let's see what this looks like when we are ready to perform a cancellation:

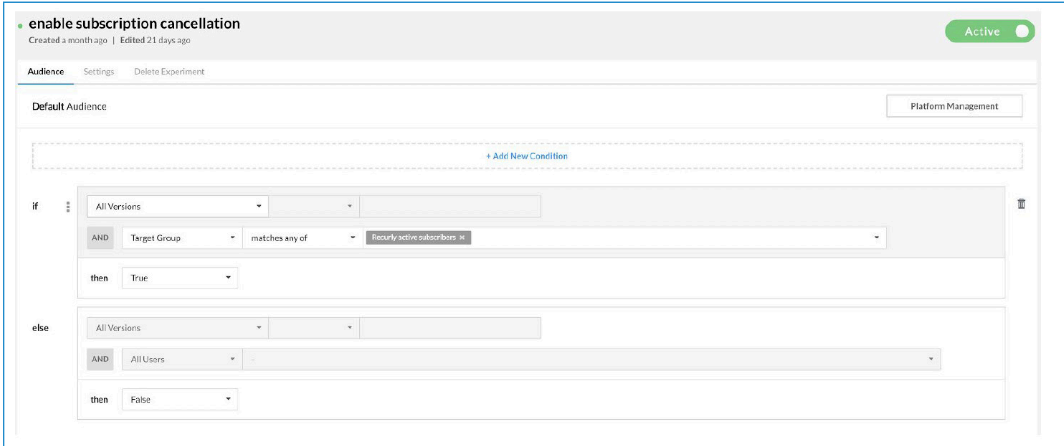
```
handleCancelClick = () => {
     if (activeSubscriptionOptions.enableSubscriptionCancellation.isEnabled()) {
         cancelSubscription() // Assume this makes a Recurly API cancellation call
     } else {
          LiveChat('Hello, I would like to cancel my plan.') // Base state
      }
}
```

Now that we have cancellation implemented, we can use the CloudBees Feature Management dashboard to configure the feature flag to return true only for active subscribers.

Under "target groups" we can define a new audience called "Recurly active subscribers" using our customStringProperty that we've set up in our code. It will match any subscription object that has a state of "active."



In our flag experiment, we can match the "enable subscription cancellation" feature flag to only return true if our target group matches "Recurly active subscribers."



Now we can rollout subscription cancellation without ever having to think about upgrades, downgrades, expiration, reactivation or any other states or state transitions!

We can then do the same for the enableSubscriptionUpgrade and enableSubscriptionDowngrade flags, allowing for rapid deployment of these features. We can even go as far as limiting the flag to be enabled for specific users in production, such as QA engineers, engineering directors or whomever.

Conclusions:

» Using Flag-driven development (FDD) as a mental model, we can take a complicated feature and break it down into smaller, more functional components

» Thinking of a "base case" or "fallback method" allows us to build in a static solution that gives users the impression that they still have access to a full feature

» Using target groups and feature flags, we can implement and rollout one component at a time and expose it to the audience of our liking with very low risk. There is no need for large feature branches.

Using Feature Flags to Validate the Public Rest API

"In two clicks within the CloudBees Feature Management dashboard, we were able to disable the input validation. This quick action pales in comparison to the additional work and stress a fallback would have brought to the team."

Guillaume Guirado,
Senior Staff Engineer

CloudBees Feature Management offers a user friendly dashboard to manage your applications, environments and flags. However, some developers also want to interact with CloudBees Feature Management directly from their CI/CD pipeline. To enable that, CloudBees offers a public REST API and has published an OpenAPI-based documentation of the API. When this was first created, no formal input validation framework was used. As a consequence, when calling the API with wrong or malformed parameters, the errors returned to the developers were not always user friendly. Sometimes they were caught too deep in the code and throwing exceptions that would simply return generic 5XX internal server errors.

In order to help the developers and also to add an additional level of security to our API, we decided to validate all the API calls. From now on, every request would be compared to the OpenAPI definition. A request that didn't match the definition would be immediately rejected with a clear error message stating which field is missing or malformed and how it should be formatted.

Before putting this change into effect, we wondered could we add this level of validation without impacting our users? Are we confident that our OpenAPI definition is perfectly describing our API? Could it be out of date? Is there a parameter somewhere that is valid, but not described in the API? Do we have places where a boolean value is expected but which, in practice, has also been accepting the strings 'true' and 'false' and the values 0 and 1? What if some of our customers are using those as part of their CI/CD?

Releasing a new version of CloudBees Feature Management with the input validation could cause issues for our users, impact their CI/CD and prevent them from deploying new versions of their software. We couldn't allow that. And we certainly didn't want a trial and error cycle of deploying a new server version, monitor logs, fallback to previous version, make changes, redeploy, monitor and fallback. In addition to impacting our users, any issues could have taken a long time to resolve and prevented us from releasing other important features. Instead, we used a feature flag.

Here is the Express.js middleware:

```
const validateInput = (req, res, next) => {
  if (rox.containers.publicApi.useInputValidation.isEnabled()) {
    swaggerValidator.validate(req, res, next)
    return
  }

  next()
}
```

This allowed us to deploy the application as usual with the flag set to false. In fact, even if the entire code behind swaggerValidator had been malfunctioning, this would have had no impact on our service. The risk was null.
We could then enable the flag on our staging environment, check our logs and monitoring for validation issues and make adjustments as needed. Once all issues caught on staging had been fixed, we moved on with production.

API validation would now be used for a select, small portion of the production traffic. Monitoring quickly revealed that validation was suspiciously failing on a given endpoint. With two clicks within the CloudBees Feature Management dashboard, we were able to disable the input validation. This quick action pales in comparison to the additional work and stress a fallback would have brought to the team. Once the feature was disabled, the developer in charge of it could spend as much time as necessary to understand where the issue was and fix it. And then the process will start again, enabling the feature for a small percentage of traffic, monitoring, increasing the percentage, fixing the code (or OpenAPI definition if necessary) and so on, until we reached 100 percent of traffic using input validation and the confidence that everything was fine.

Improving Customer Experience Using CloudBees Feature Management

Launching the new CloudBees.com wasn't a risky proposition using feature flags

"The new CloudBees.com website went live seamlessly, quietly, uneventfully and without the explosions that sometimes accompany 'Big Bang' deployments. Just exactly like we planned."

David Pitkin,
Director of Engineering

Last year, we decided to migrate from our old content management system (CMS) - which was better suited for a smaller company - to a more robust, modern system. The new CMS - Contentful - gives us more enterprise capabilities, such as finely-tuned access controls that allow designated CloudBees employees to edit the web content they own directly.

Bee-ing a team at CloudBees meant we also wanted to do this in a continuous fashion with a modern pipeline. Website launches are almost always a "big bang" release, with a lot of effort focused on a perfect reveal. But that's not how CI/CD best practices are written and certainly not in keeping with progressive delivery. Understandably, our marketing team was a little worried, They had reservations about SEO impact and the impression an unfinished product might have on a customer or potential customer. We knew these were real concerns, but still wanted to find a way to quickly deliver new updates and features without holding out for perfection.

How could we do both? Then, in a divinely inspired act of "eating our own granola," we decided to use feature flags. With the new website available behind a feature flag, our authentication system could tell us if a logged-in user was an employee. Using that segmentation criteria we used CloudBees Feature Management to enable what would be the production website. This way internal stakeholders could keep track of what was happening and the team was able to deploy to production.

So how do we deploy to production at CloudBees?

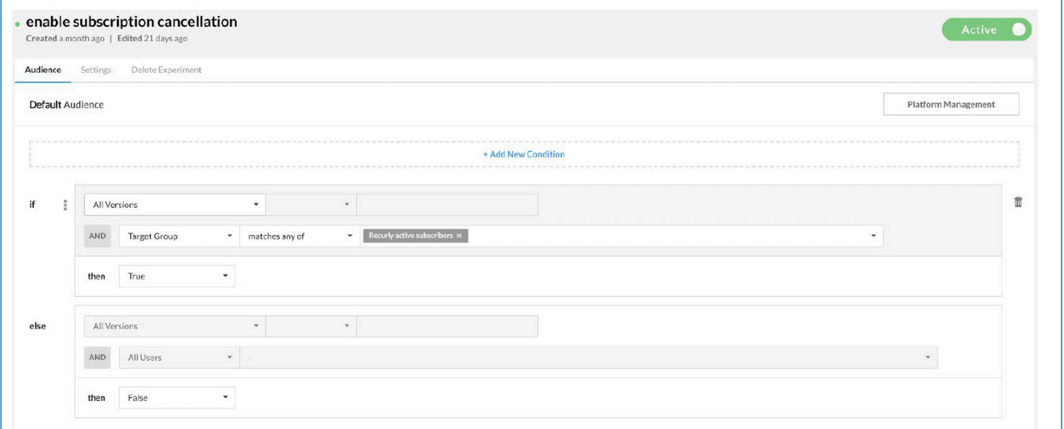
Our CI/CD is based on CloudBees CI, where our Jenkins pipeline runs at every commit. A pull request only gets merged if approved by at least one other engineer and if all steps in Jenkins were successful. Some of our most important steps are:

1. TypeScript code compilation
2. Linter rules check
3. Unit tests
4. Creation of the docker image
5. Apply migrations to our Contentful model
6. Tests against the docker image and the Contentful environment

At every merge on our master branch, a new build is run and code is automatically deployed to our staging environment. Because of the numerous in-depth changes made daily to the website codebase, we decided to have an engineer checking the state of staging daily and, when OK, triggering the production deployment (all done within CloudBees CI of course!)

We could have done this without any human interaction, but it would have required a heavy investment in automated testing. We were going through iterations so quickly between marketing, design and the engineering team, that this approach didn't make sense for us.

While Jenkins was running dozens of builds every day, the general public was only able to view a few pages -made of a handful of UI components - and could barely notice any change. However, every logged-in CloudBees employee was able to witness our progress and give feedback in real-time via Slack. No VPN necessary, no magic URL. All done with a simple feature flag via CloudBees Feature Management:



Things we learned

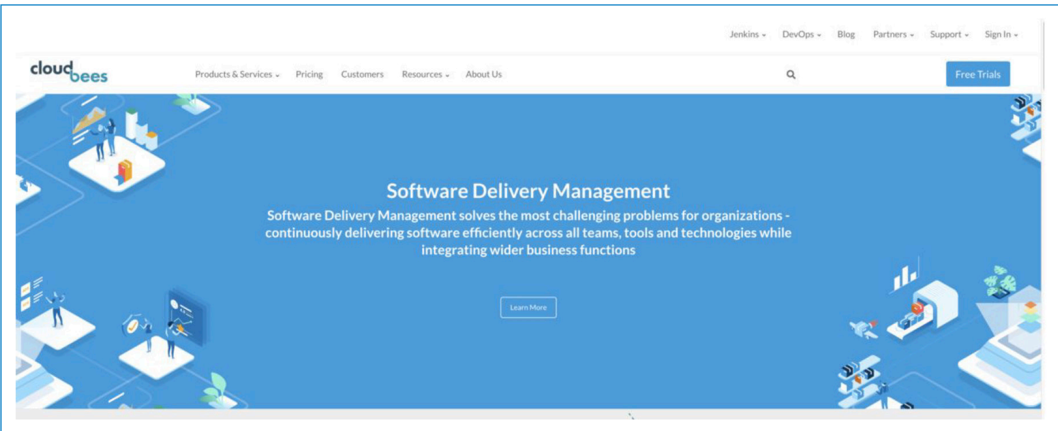
It's always important when eating your own granola to keep track of the things that were harder than expected. During this project, we learned a few things that are actively becoming part of our product roadmap.

1. It was hard to keep track of work that was and was not behind the feature flag.
2. For the website, it was hard to determine the flow of code/features and the flow of content. We were copying production to staging for content but code went from staging to production.
3. With only local development, a staging environment and production environment, some members of the team could not really view design changes.

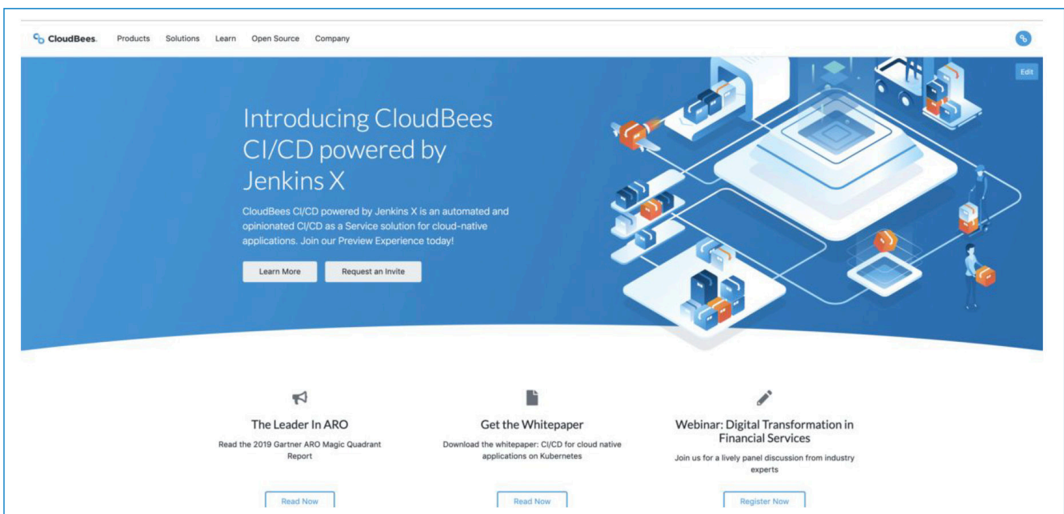
The most boring go-live (in the best way)

On November 25, 2019, the new CloudBees.com website went live seamlessly, quietly, uneventfully and without the explosions that sometimes accompany 'Big Bang' deployments. Just exactly like we planned.

Before:



After:



How we used CloudBees Feature Management for hiding billing details from certain customers

"We needed to find a way to hide that new billing tab to specific users. This was a great case for feature flagging."

Rafi Yagudin,
Senior Engineer

In 2020, we migrated to a new billing system for CloudBees Feature Management. Our customer account dashboards were updated to reflect the new system. However, as a significant portion of our customers were still on the legacy system and waiting to be moved over, we needed to find a way to hide that new billing tab to specific users. This was a great case for feature flagging, so we decided to put the billing tab behind a **billingTabVisible** feature flag.

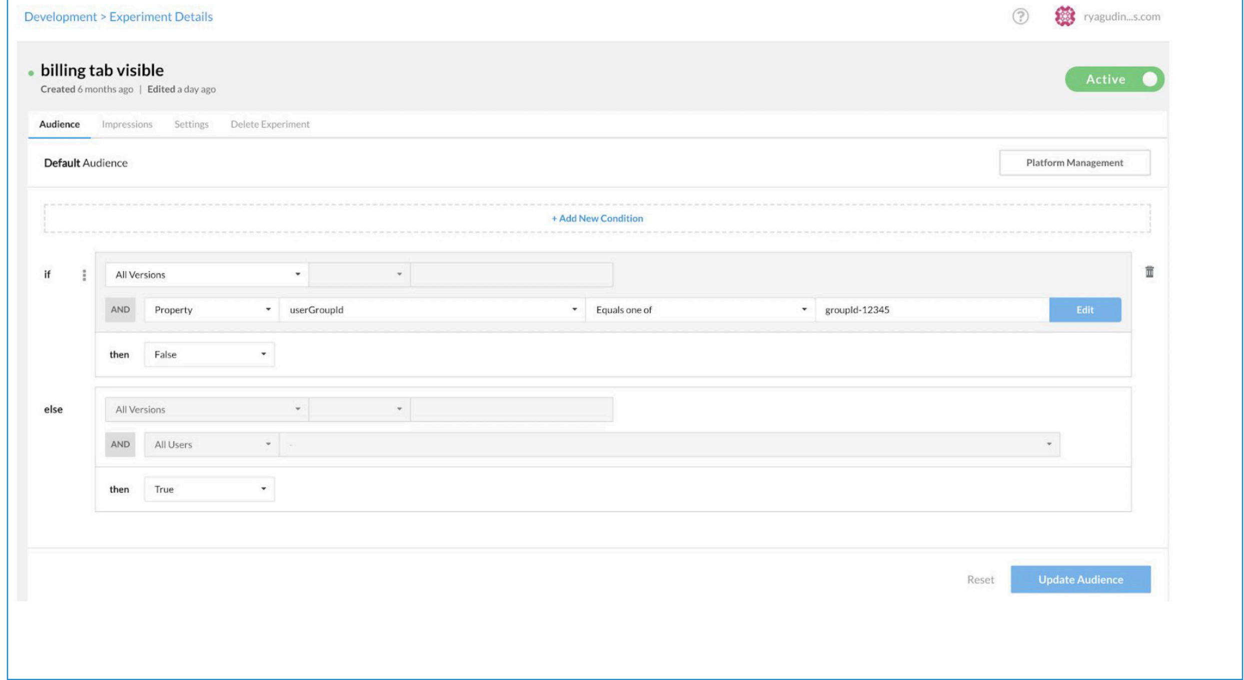
The implementation of the flag looked like this:

```
const teamManagement = {
  billingTabVisible: new Rox.Flag(false)
}

// Register the flags

Rox.register(teamManagement);

// Register a custom property for a users's group

Rox.setCustomStringProperty('useGroupId', () => (this.state.user.team ? this.state.user.team : undefined))
```

This was the UI implementation around showing the tab:

```
export default class TeamManagement extends Component {

  get tabs () {
    const tabs = {
      settings: {
        label: 'Settings',
        content: <TeamDetails />
      }
    }


    // Flag will either return false or true depending on the rules we set in the Rollout flag experiment dashboard

    // If it returns true, we append the <Billing /> component to the tab object
    if (teamManagement.billingTabVisible.isEnabled()) {
      tabs.billing = {
        label: 'Billing Plan',
        content: <Billing />
      }
    }

    return tabs
  }

  render () {
    return (
      <TabsContainer
        tabs={this.tabs}
        rightComponent={<SaveIndicator />}
        team={this.team}
        {...this.props}
      />
    )
  }
}
```

By setting up a flag experiment using the **billingTabVisible** flag - matching on a user's **userGroupId** property - we can now dictate which teams can see the new billing tab.



In the above screenshot, if a CloudBees Feature Management user belongs to a group with an id of **groupId-12345,** then the flag will return **false** and the billing tab will be hidden from this group. Of course you can add more **usereGroupId** values for the flag to match on, which would represent different customer groups in the system.

Conclusion

It should be clear from these use cases that our developers really love feature flags. It makes their day-to-day programming lives easier, and it makes changes for our end customers simpler to navigate. Some software professionals still have the perception that feature flags are primarily for experimenting with features, but we see them as a pillar of our development processes. While the idea of feature flags may be known to your team, the idea of systemized "feature flag-driven development" may not be. Apart from the specific use cases we shared, it's this idea that we hope to leave you with today: some of the best software development teams - including ours - are highly successful because they have created a development culture that embraces the frequent use of feature flags. This allows them to move fast without increasing risk or quality, and gives them increased freedom to experiment as they know they can roll any failed experiments back easily.

Of course, our engineers are fortunate in that they have an enterprise-grade feature flag management platform available to them internally at all times. If you're interested in creating your own feature flag-driven development culture, but have struggled with scaling your flag usage across your organization, feel free to give CloudBees Feature Management a try and see if an enterprise solution can help. Our developers love it, and we think yours will too!

The following resources will help your team get started with feature flags:

Get A Free Trial Today!
www.cloudbees.com/products/feature-flags

© 2022 CloudBees, Inc., CloudBees® and the Infinity logo® are registered trademarks of CloudBees, Inc. in the United States and may be registered in other countries. Other products or brand names may be trademarks or registered trademarks of CloudBees, Inc. or their respective holders.

0122v03

CloudBees, Inc.
4 North Second Street | Suite 1270
San Jose, CA 95113
United States
www.cloudbees.com
info@cloudbees.com