split

Feature Flags: Choosing to Build or Buy

Kicking off an internally-developed feature flagging system is standard practice for many companies today as feature flags help engineering teams release faster at lower risk. The initial use case might be to support a move to continuous integration to provide engineers rapid feedback on how well new code integrates with the master.

Once the benefit of feature flags are proven, additional dev-driven use cases quickly emerge. The scope of the in-house application grows and in turn, so does the list of challenges when choosing to build in-house. Product teams recognize the immediate benefits and look to quickly leverage feature flagging and introduce their list of use cases. The scope of the in-house application grows and in turn, so does the list of challenges of in-house development. With each additional feature request or use case, the burden of maintaining an in-house solution continues to grow.

This guide walks through the typical progression of an in-house solution that we have seen through many customer conversations. From dev-driven to business-driven use cases, the challenges grow almost as fast as, if not faster than, the list of requirements. Teams face multiple decision points to continue investing in an in-house solution or look at external options. Not all these use cases will apply to your feature flagging practice, but they do reflect a very common progression.

Contents

Dev-driven use cases for feature flagging

Continuous Integration

Engineering teams have embraced continuous integration (CI) to accelerate their release cadence. By frequently integrating new code into the main branch, then kicking off automated build and unit tests, engineers shift left to test early and often, efficiently evaluating code quality at each commit.

Feature flags are a fundamental requirement for a CI branching strategy. With feature flags, code can be incrementally merged into the main branch but kept dark, reducing merge conflicts when multiple engineers are also trying to integrate new code at the same time. As a result, everyone on the team can push smaller amounts of code to master more frequently and identify issues faster. Once successfully merged, the feature branch can be quickly cleaned up to keep branches short-lived.

Engineers could implement a feature flag by reading from a config file, for example, to control which code path gets exposed to a subset of internal users. The simple on/off switch probably works fine with a small number of developers and proves out the concept of feature flagging without much difficulty.

Because feature flagging helps engineers release faster with lower risk, ideas for additional dev-driven use cases grow quickly. Dev teams want to exploit all that feature flags can provide. The in-house tool that was initially built to support CI grows in importance and scope.

Controlled rollouts

For starters, existing processes for rolling out new functionality are typically slow and have limited flexibility. Using feature flags to implement controlled rollouts streamlines the release process and reduces the risk of production failure because the new functionality is exposed to the user base gradually as stability is assessed. Controlled rollouts help engineers balance the business demand for both increased release velocity and improved product quality.

Kill Switch

With this cautious approach to rollouts comes the need to quickly rollback when issues are uncovered. Having a kill switch gives engineering teams confidence that with just one click of a button they can revert to previously successful application behavior. Feature flags create a low-risk framework where problematic code can be made instantly invisible to users without needing to rollback a deployment or issue a patch.

Build or buy decision point #1

And that's typically in just the first six months of using feature flags! The upside is the team has experienced the benefits of feature flagging, and there's no desire to turn back to a monolithic release concept. But the delivery teams are moving fast, and release velocity is increasing from DevOps efforts, and momentum is building. There never seems to be time to stop and evaluate a 3rd party feature flagging solution let alone expand the scope of an in-house solution. What's next? Now that phased rollouts and kill switches are standard, engineering teams envision more uses for feature flags.

Additional dev-driven use cases for feature flagging

Test in production

Testing in production– conducting functional and performance tests with real-life traffic, systems, and data– provides several significant benefits:

- Some things are difficult to test in a pre-production environment, such as network latency or the response time of a specific endpoint.
- QA teams can eliminate the need for a 'production-like' staging environment behind the firewall, going directly to test in the production environment for functional, performance, and UAT testing.
- Mobile applications benefit from the ability to test in production. With feature flags, engineers can upload a build to the application store that contains new code and bug fixes, then expose them only to employees for testing before customer release.

"Digital products aren't just designed for one device anymore. They have to work across an array of devices and retain context as users "multi-screen" between desktop, tablet, phone, and back again. Data has become an enormous differentiator in digital products."

Product Cloud Alliance
@ www.productcloud.io

Full-stack control

In the digital world, rarely will the customer experience exist solely through a website. On the front-end, the proliferation of end-user devices means a range of digital interfaces which add complexity to feature releases. Maintaining a consistent user experience across all devices and touchpoints requires advanced planning and careful execution. By keeping new functionality behind feature flags, teams can align release schedules to establish feature parity across languages and devices.

On the back-end, developers may be working on machine-learning models or other server-side changes. Putting all new functionality behind a feature flag to control rollouts and monitor code stability gives teams the confidence to meet aggressive release dates.

Build or buy decision point #2

This is the point where many companies start to seriously weigh the opportunity cost of engineering resources working on an in-house solution that has been continuously growing in scope. The engineers developing the in-house solution (the same engineers who are also working on core business projects) will need to lay the foundation for a more robust feature flagging system to support the growing use cases and related requirements. To support the dev-driven use cases described above the project scope now includes:

- Percentage allocations to create random distributions for phased rollouts
- Targeted rollouts for whitelisted accounts or internal teams
- Kill switch for an immediate rollback
- Centralized functionality for feature flag archiving and removal
- SDKs to service the entire application stack
- Integrations to project management, application performance, and log management tools to streamline workflows

Achieving continuous delivery requires consistent collaboration with product management. And just as dev-driven use cases grow when engineering teams leverage the benefits of feature flagging, so do product-driven use cases. Product teams realize feature flags give them extensive control over the user experience, increasing the requirements list for the in-house solution.

Product-driven use cases for feature flagging

Beta programs and paywalls

Feature flags are an easy way to establish a beta program– essentially a phased rollout– exposing new functionality to a specific set of customers. Feature flags can also define entitlement paywalls by serving a select group of features to different user groups depending on their license level. In either case, feature flags provide product managers granular control of user segments to effectively evaluate application functionality.

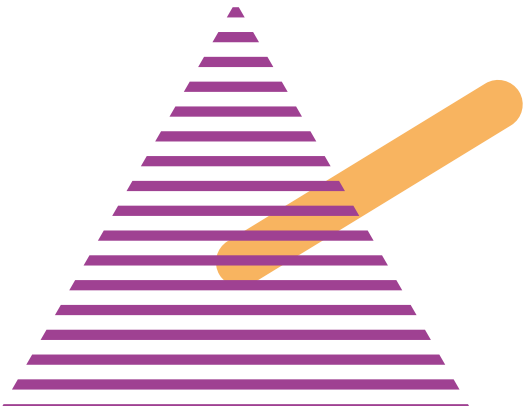
A/B/n testing

With A/B/n testing, product managers make better product decisions. A/B or multivariate testing provides a critical feedback loop for feature iteration and continuous improvement. Product managers can define any number of feature variations and randomly assign users to each treatment at the same time, reducing their reliance on both engineering and business intelligence resources.

Access to metrics that measure the overall customer experience is critical here, such as the number of actions or time spent in the application. Feature flags provide product managers a way to measure the impact of their features without risking exposure to the entire user base. Product managers can't manage the customer experience if they don't measure that experience.

Roadmap planning

Product managers bring together the design, engineering, and customer success teams to ensure products are both functional and delightful for end users. They need tools that help them leverage engineering and design resources in a more efficient way. Access to both qualitative and quantitative data gives product managers more insights to make better production decisions. A growing eco-system of SaaS tools purpose-built for product teams can augment analysis garnered using feature flagging tests, bringing greater insight and agility to roadmap planning.



Build or buy decision point #3

There's no doubt that feature flags grow in importance once engineering and product teams begin to recognize the benefits of feature flags. By separating code deployment from feature release, feature flags transform from a niche project to business critical functionality. As the value of the feature flagging system increases so does the need for continual investment.

With the growing list of use cases and need for an operating environment, the scope of the in-house project continuous to creep:

- UI
- Metrics dashboard
- Granular control of user segments
- Multivariate testing
- Randomization engine
- Integrations to the product tool suite
- Permissions for phased access and edit rights
- Audit trail of all changes to flags and target segments
- SAML sign-on

There's also testing and maintenance to work out as well. And do engineers really want to document an in-house application? Who is going to take responsibility for training and support? This is not our area of expertise! Exit, stage left! Can the in-house solution scale to meet the needs of the business and the increased number of users? Without huge commitment and investment, the challenges of developing an in-house system mount.

Top Challenges When Building a Feature Flagging Solution from the Ground Up

Challenge #1: Manual config changes

In-house feature flag solutions typically leverage a file or database for turning features on or off, so an engineer must make a config change for every feature release. When the flagging implementation is database-backed, and a feature is ready for rollout, a column is added to the database with a Boolean on/off indication. Any code change is inherently risky and comes with the potential for error.

Also, config changes often go through the same deploy process as standard code which may take hours or even days to push out code. That means any config change to flags will take just as long. Finally, manual config changes mean product managers are unable to make their own changes and are therefore dependent on engineers for every single change.

Challenge #2: Manual compilation of target segments

With this basic implementation, the feature will be either on or off for all customers. When conducting a phased rollout, compiling the list of customer IDs for each feature exposure will be a manual process, which requires an engineer to go to an admin page and turn the feature on for each user.

Expanding the customer segment for a phased rollout will require another manual compilation of customer IDs, and so on until 100% of the customer base is reached. Keeping track of the treatment served to each user can quickly get out of hand. Also, a customer base is not going to be static, and names will be continually added and removed over time. Tracking who is entering and who is leaving the customer base will also require manual tracking. Somehow.

Challenge #3: Problematic customer support

Access to which customer has which feature turned on or off becomes a real issue. At best the data is stored in a plain text file with UUIDs, but these aren't readable by product management or the customer support team. Worse, the data may be hard-coded as environment variables that are impossible to access. Not knowing what experience a customer has received makes customer support more troublesome. Anything that causes a problem for customers or increases support calls should be immediately turned off, but this will take precious time without rapid access to the user's treatment.

Challenge #4: Technical debt

Many of the companies we've worked with often come to us with disjointed solutions across their dev teams with no centralized source of truth of "what is flagged". Imagine each microservices dev team creating their own unique feature flagging system. As a result, monitoring and clean up of old feature flags becomes increasingly difficult as teams generate more and more flags. Old flags can reduce code readability or worse, result in accidental misconfiguration. Even with a sourced feature flagging system in place, technical debt can be an issue (we recommend several best practices for managing feature flag debt.

Challenge #5: Lack of documentation

Identifying the owner to a specific feature flag and documenting information such as what the flag does and why it was created can be difficult if an in-house solution doesn't track this data. Employee turnover or simply time passing can result in teams having to re-establish the original purpose of the flag to determine if it's still needed or risk keeping it in the code and forgetting it altogether.

Challenge #6: Incomplete open source options

What dev teams soon discover is that open source tools are designed for just one part of the development stack, and no single library can provide all the desired capabilities. Maybe there's one for JavaScript, but it doesn't provide an audit trail. Maybe there's one for .NET, but it doesn't offer a UI for viewing metrics. To support the full application stack, access to complete functionality and breadth of languages will require multiple tools. Toolset fragmentation is not a desirable path. Now what? (Hint: Split free trial)

Challenge #7: Lack of a UI or metrics dashboard

The lack of a UI ties the product manager to an engineer for every unique feature request. Product managers will need to work directly with a developer on every feature rollout, which causes significant coordination overhead. Also, product managers have limited visibility into what treatments are served and to whom. Keeping track of which treatment a customer receives is critical during a rollout, especially for those customers who may be considered high risk or high touch.

Everyone on the delivery team needs access to metrics when measuring the impact of the release: "Should we keep releasing or stop because there is a problem?" "Are users responding well to the new functionality?" To make informed decisions, all stakeholders need a centralized dashboard to view the feature-level impact on application performance and business metrics.

Challenge #8: Projects get orphaned

An in-house feature flagging system is not the expertise of the company, and the project can be easily reprioritized to jobs that more clearly reflect business critical requirements. Or that one engineer who spent time on the project leaves or is re-assigned. Typically with little to no documentation (see Challenge #5), there isn't the institutional knowledge needed to keep the project going.

Challenge #9: Application performance takes a hit

Without careful design of the framework, an in-house feature flagging system may cause a reduction in application performance. If there is a remote API call for every flag computation or decision this could have a significant impact on application performance as the number of feature flags and treatments grows.

Challenge #10: No controlled access or audit history

What was initially a simple on/off switch has grown into a significant application. This can become a burden on the organization's internal resources, which now has to monitor and maintain the app over time. For organizations to manage the feature flagging solution, there should be change and access control so dev teams don't have to worry about someone making a change they shouldn't. You'll also need audit logging and SAML sign-on.

Enter Split

Split provides a sophisticated feature flagging system that helps teams meet the business need to release faster with lower risk. The robust architecture and rich feature set empower engineers and product managers to release more quickly at lower risk. Best of all Split is available now and is in use at companies from a range of industries including healthcare, financial services, retail, travel, and many more.

Advantages of Split

- Robust and flexible targeting for any rollout plan design Management console provides intuitive reporting on experimentation results
- Leverage multivariate rollouts to experiment with multiple versions of a feature before fully releasing the most performant, or permanently maintain multiple versions of the same feature
- Same platform for feature flagging and product experimentation
- Create whitelists and blacklists directly from the UI or via Split's API
- Release features to dynamically created subsets of your customer base based on any customer dimension you track
- Data to correlate every code change on performance, product, customer experience, and business metrics
- Statistical engine to determine the statistical significance of all metric impacts
- Customizable UI serves all stakeholders involved in the release process - engineering, product, QA, data scientists.
- Changelogs to audit and troubleshoot issues
- Track the status and progress of any rollout
- REST API to ingest user events from any location
- SDKs available in multiple languages to service your entire stack
- Indicators assist management of feature splits and avoid technical debt
- Integrations with other collaboration, performance monitoring, project management, log management, data lake, exception tracking, and product management tools
- Dedicated team to assist with internal onboarding, training, and support

Envisioning the Future: Product Experimentation

With a feature flagging system in place, use cases often evolve to increase business value through phased experiments, where teams measure the impact of new functionality against key business metrics such as product conversions, engagement, growth, subscription rates, or revenue. Delivering what customers want at the speed they demand requires a cultural shift to hypothesis-driven development and data-driven decisions, where dev and product management work hand in hand with definition and analysis to make outcome-based product decisions. The final step in the evolution is when the entire company views every feature release as an experiment, facilitating a culture of continuous improvement.

"We don't like to spend time doing things that aren't our core competency. Why maintain a feature flagging system, build and maintain its UI, or monitor its performance? Split does what we need it to do, and any engineer in the company can use it."

Chris Conrad, VP Engineering, WePay

Split provides the deep statistical capability required for feature experimentation so teams can easily augment their feature flagging practice to include analytics. There is no need to implement another toolset or re-enter user segment lists.

Hypothesis-driven development

Engineering and product teams use Split as a tool to iterate and improve features based on a robust statistical analysis. If the minimum viable feature doesn't test well with users, rather than kill the feature right away, they will work to make it better, then execute another user test. The rapid feedback loop helps engineers refine functionality and speed up the iteration of ideas.

Data-driven product decisions

Product managers rely on Split to drive data-driven product decisions by measuring feature impact on business outcomes. Real-time analytics track how customers interact and respond to new features, providing actionable insights that increase feature adoption and accelerate revenue growth. Experimentation is the product manager's Jedi trick for beating the competition. Experiment early and often, understand what works, and discard any features that provide little benefit or hinder the customer experience.

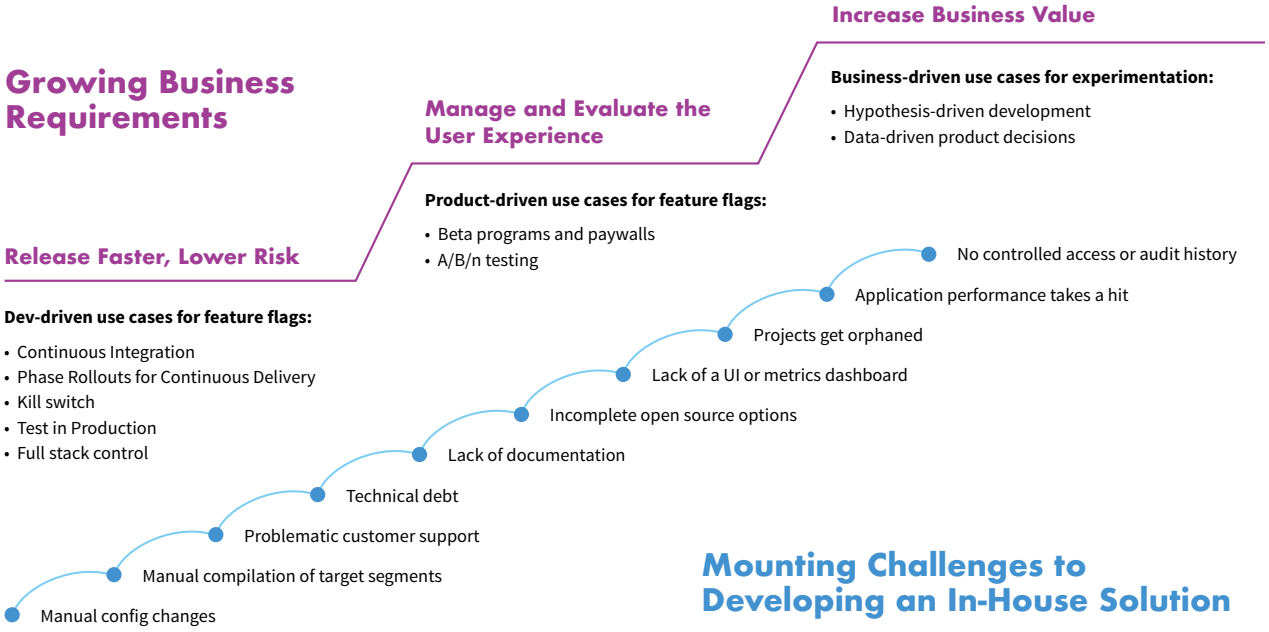
Conclusion

Creating an in-house feature flagging system seems like a good idea at first, and at the very least can help demonstrate the benefits of the technology. Dev-driven use cases such as continuous integration, phased rollouts, kill switches, testing in production, and full stack control helps engineering teams meet business needs of releasing faster with less risk to application stability. With the increased use cases comes an increase in both project scope and challenges to continued in-house development.

DevOps teams work closely with product management to continually measure the impact of features to iterate on innovation rapidly. Product management-driven use cases such as beta programs, paywalls, A/B/n testing, and roadmap planning help the business evaluate and manage the user experience. The project scope for the in-house solution grows and so will the challenges, and the need for a full-featured platform becomes evident.

Feature flagging systems grow to business-critical functionality, increasing application value by measuring feature impact on business metrics. Split's full-stack feature flag and experimentation platform empowers engineering and product teams to make smarter product decisions.

As use cases for feature flags grow, so do the challenges of building an in-house system:



Appendix: Scope/cost estimation chart

PROJECT SCOPE	(Engineering Resources + Testing Resources)		X Average Hourly Rate	= Total Line Item Cost
FEATURE FLAGGING FRAMEWORK				
Feature configuration				
Import user data for feature assignments				
Ability to create whitelists for internal teams or specific users				
Segmentation method for percentage targeting				
Kill switch				
METRICS NEEDED TO MEASURE FEATURE IMPACT				
Application load time				
User clicks				
Conversions				
UI/CONTROL PLANE FOR EXPANDED STAKEHOLDER USAGE				
Feature configuration				
User segmentation				
Metrics dashboard				
FEATURE MANAGEMENT PLATFORM				
Tracking mechanism for feature flag archiving and removal				
Permissions for controlled access and edit rights				
Tagging				
Audit log of all changes to flags and target segments				
SAML sign-on				
ADVANCED CAPABILITIES				
Assignments to different test environments				
Multivariant feature assignment for A/B/n testing				
Granular control of user segments				
Rule-based (user attributes) user targeting				
Randomization engine				
SDKS TO SERVICE THE FULL APPLICATION STACK				
Android				
GO				
iOS				
Java				
Javascript				
.NET				
Node.js				
PHP				
Python				
REST API FOR CUSTOM INTEGRATIONS				
INTEGRATIONS				
Webhooks (examples: Audit Trail, Impressions)				
Collaboration/Chat (examples: email, Slack, Hipchat)				
App Performance Management (examples: AppDynamics, New Relic, Datadog, Librato)				
Project Management (example: Jira)				
Log Management (examples: Sumologic, Papertrail)				
Data Lake/Data Hub (example: Segment)				
Exception Tracking (example: Rollbar)				
Integrations to product stack (example: Pendo)				
APPLICATION SERVICE				
Ongoing maintenance				
Documentation				
Training & support				

ABOUT SPLIT

Split is the leading platform for feature flags and experimentation, empowering businesses of all sizes to make smarter product decisions. Companies like Salesforce, Vevo, and Twilio, rely on Split to securely release new features, target them to customers, and measure the impact of features on their customer experience metrics.

Learn more at **split.io**