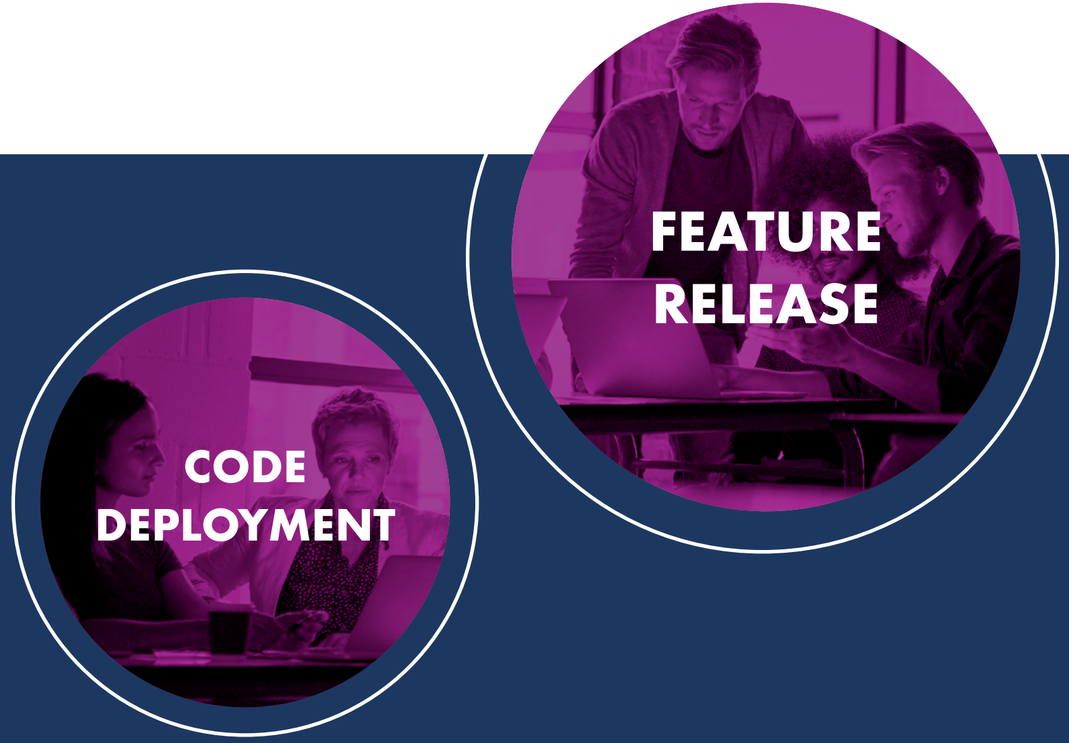
Top 10 Challenges When Building a Feature Flagging Solution From the Ground Up

split

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. A simple on/off switch probably works fine with a small number of developers and proves out the concept of feature flagging without much difficulty.



There’s no doubt that feature flags grow in importance once engineering and product teams begin to recognize the benefits. By separating code deployment from feature release, feature flagging is transformed from a niche project to business critical functionality.

Use cases for feature flags

- **Continuous Integration**
- **Phased rollouts to support Continuous Delivery**
- **Kill switch**
- **Test in Production**
- **Beta programs**
- **Paywalls**
- **A/B/n testing**

The challenges of developing an in-house feature flagging system grow almost as fast as, if not faster than, the list of requirements. Can an in-house solution really scale to meet the needs of the business?

Here are the top 10 challenges we see customers run up against when developing an in-house feature flagging system...

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.

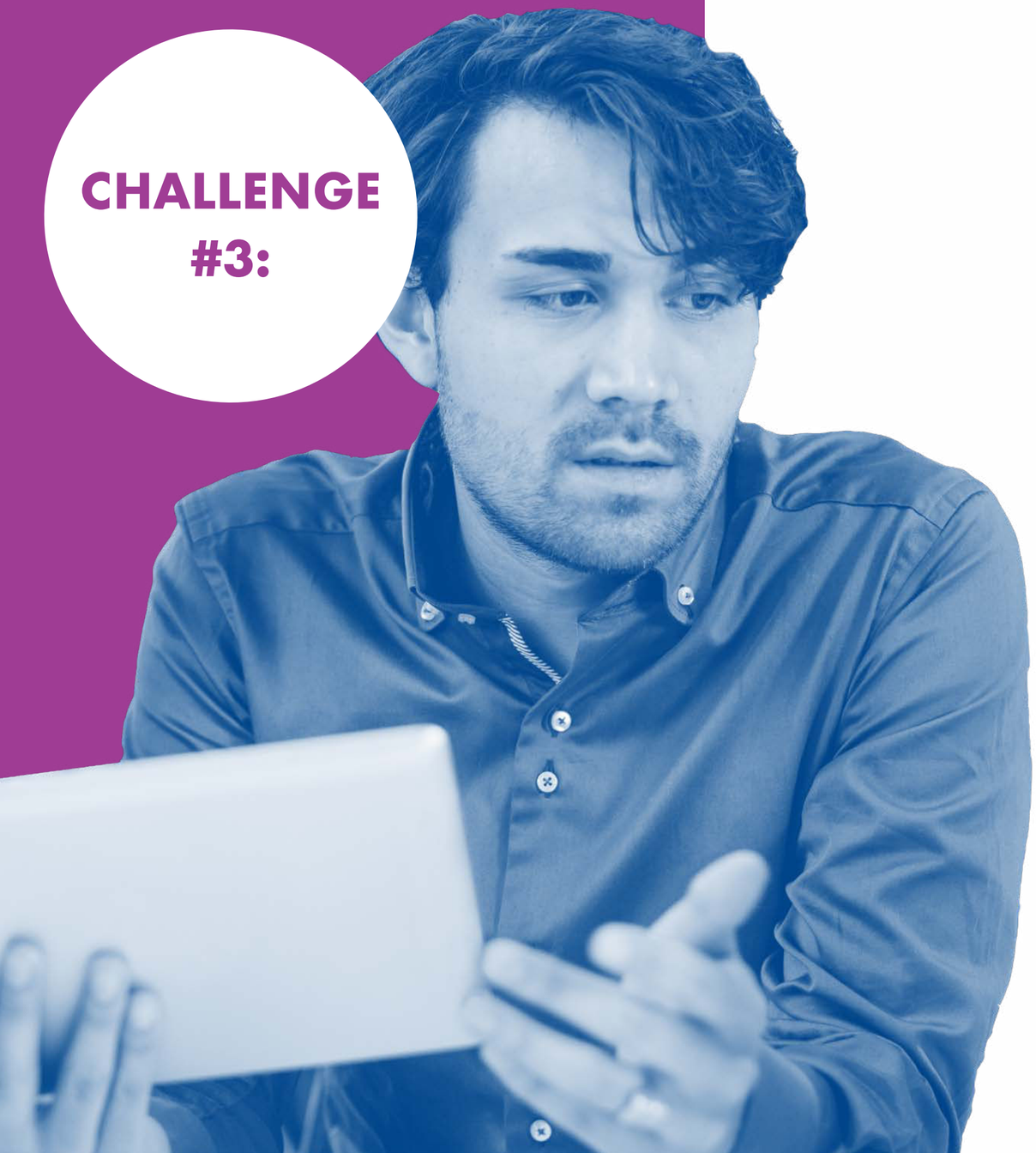
CHALLENGE #2:

Manual compilation of target segments

With this basic implementation, the feature will be either on or off for all customers. When conducting a controlled 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

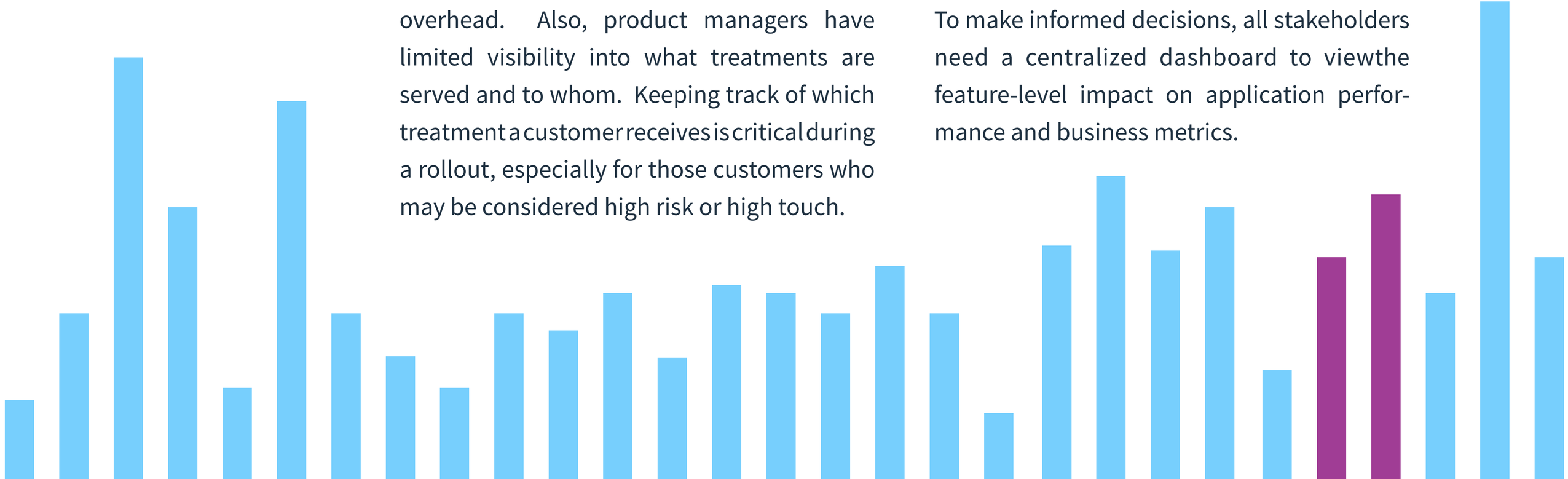
What dev teams soon discover is that open source tools are designed for just one part of the development stack, and no single library can provide all the desired capabilities. Maybe there's one for JavaScript, but it doesn't provide an audit trail. Maybe there's one for .NET, but it doesn't offer a UI 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.

CHALLENGE #7:

Lack of a UI

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 viewthe 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 access controls 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.

Exit, stage left! Enter Split.

CHALLE

Split provides a sophisticated feature flagging system, with a robust architecture and rich feature set that empowers engineers and product managers to release faster 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.

Request a tailored demo to see how Split can help you implement feature flags to release faster, lower risk, and make smarter product decisions.

Visit www.split.io

split