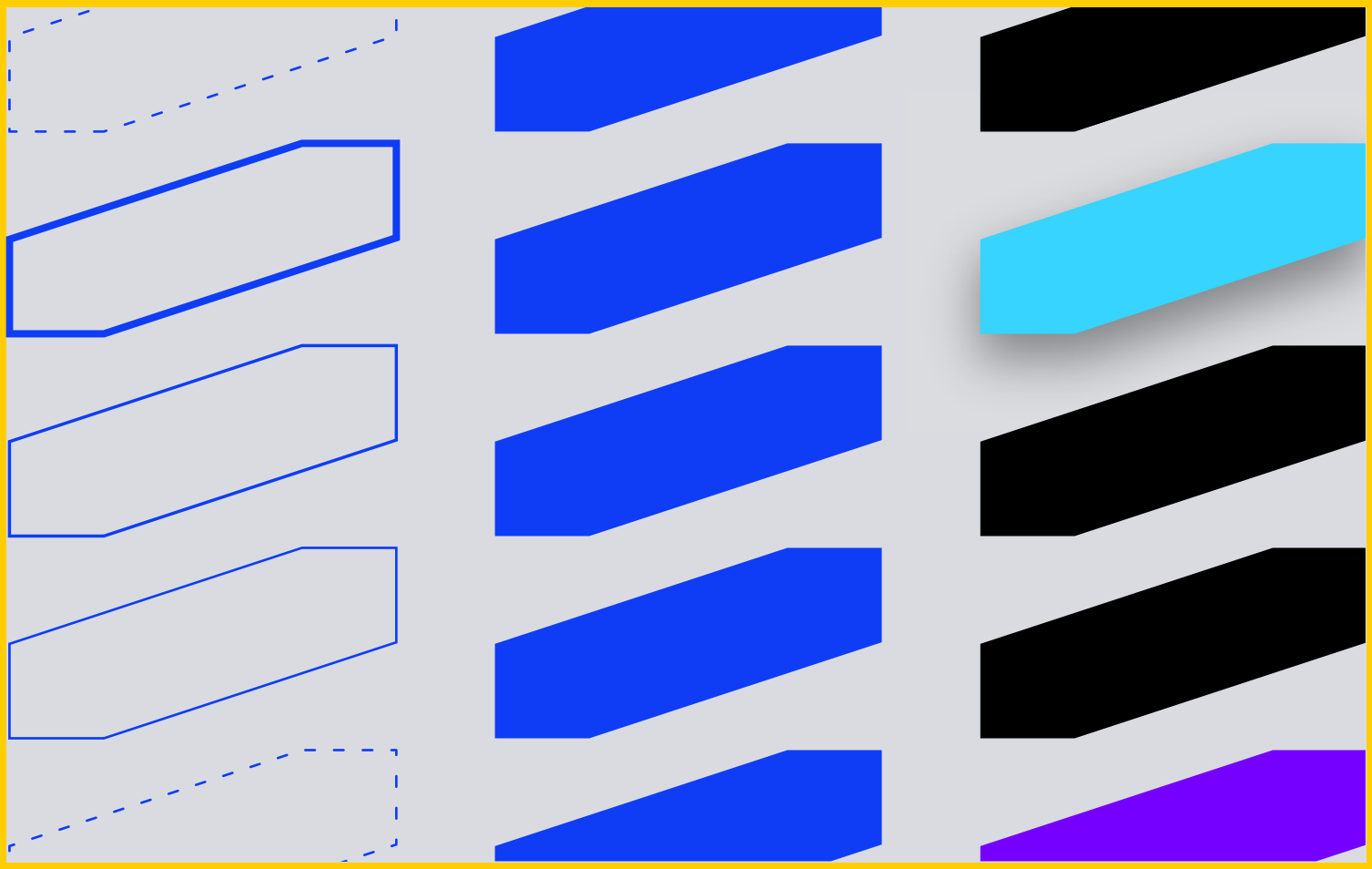


EBOOK

Build, Buy, or Complement

How to choose the right progressive delivery
and experimentation solution for your organization

Optimizely

Progressive delivery and experimentation helps deliver better software products that drive growth, faster.

Product teams not only want to move fast, they want to deliver high quality experiences that delight users in ways that drive business impact—they want to move fast and get it right. Many teams are investing in progressive delivery (feature flagging, phased rollouts, 1-click rollbacks) and experimentation solutions that release software more quickly and safely, and enable testing to learn what works in production with real users. In general, there are three ways to establish a progressive delivery and experimentation program: invest in an in-house solution; buy a commercial, industry-tested platform; or complement an in-house solution with a commercial product.

Evaluating these options comes down to four key areas of consideration that we've identified based on hundreds of conversations with Optimizely customers and prospects looking to add feature flagging to their software delivery process and experimentation-driven product development. This guide will help you understand each consideration and their trade-offs. At a high level, they are: **Capabilities**, **Cost**, **Statistical rigor**, and **Adoption**.

Industry trendsetters use progressive delivery and experimentation to build products users love.

What do the product development teams at Amazon, Uber, Netflix, Google, and Facebook have in common? They all build and deliver great products that have dominated their markets, that customers love. And they've done it with a laser focus on building strong product development teams that use progressive delivery and experimentation as critical business processes.

To perform both progressive delivery and experimentation at scale, Netflix has dedicated hundreds of engineering and data science resources over many years to develop in-house solutions that support building the most optimal product. They've created toggle switches for feature code to improve the process of delivering software and they've run experiments on every new feature in a controlled environment before rolling them out widely. Experimentation has helped them determine the functionality that users will love, while progressively delivering software code safely to minimize risk.

Part of the success of large tech organizations is the setup of the product development teams, composed of tight-knit groups representing cross-functional disciplines from engineering, product, design/UX, and data science. Their progressive delivery and experimentation platforms have specific functionality that allows each discipline to operate independently from any other team. It's also the expectation that these teams work with

"Our success at Amazon is a function of how many experiments we do per year, per month, per week, per day."

Jeff Bezos / Amazon

an experimental, psychologically safe mindset, focused on the same goals. While they're all working toward delivering functionality users will love, product owners, data scientists, engineers, and marketers all have different responsibilities and needs around how to effectively deliver software and run experiments. Let's take a look at the needs for each role:

Product owners want more control over their feature releases, using feature flags and rollouts to gradually control the blast radius of their features, to run a beta program for example, to see what works with their users and what doesn't. While in-house solutions support this need, they are often code-based with no UI, so managing and remotely controlling the feature flag and rollout still requires an engineer. Running experiments poses a similar challenge, as changes must be made to the code and there is no UI to enable product owners to set up, configure, and execute an experiment, then see how it's performing.

Data scientists often take the lead in implementing experimentation solutions internally and certifying test results. This approach works best when companies can afford a large, centralized data science team as well as embedded analysts in product teams. But smaller organizations may struggle to ensure rigorous analysis and bring in third-party platforms or libraries to automate common analysis tasks.

Engineers want to code and deploy with greater speed and confidence and have traditionally built feature flagging and/or A/B/n testing tools in house to enable experimentation directly within the codebase. However, these systems can sometimes introduce added latency, more complex deploys, or challenges with scaling to new devices, channels, and services.

Marketing teams often look for a platform like Optimizely's flagship Web Experimentation, that enables anyone to make quick changes to the frontend of a website with JavaScript and run A/B/n tests without the help of a developer.

Ensuring your product development team can run a successful progressive delivery and experimentation program comes down to the type of platform you choose to adopt, which varies widely across in-house, open source, commercial, or hybrid solutions. When evaluating whether you should build, buy, or complement an internal solution with commercial capabilities, keep these four considerations in mind:

TOP CONSIDERATIONS

1. **Breadth of capabilities and compatibility:** Which advanced features, functions, and integrations will you need for everyone to easily manage features, run more experiments, and collaborate across teams?
2. **Total cost of ownership:** What is the total cost of ownership for developing and maintaining the platform over time?
3. **Statistical rigor:** Do you have confidence that your experimentation platform is producing reliable results and that teams are able to interpret those results without help from data scientists?
4. **Adoption and scale:** What support and processes do you need for your platform to scale and be adopted across your organization?

WHY IT MATTERS

1. Advanced features and compatibilities in a unified platform can enable more software and experiment throughput.
2. The initial expense of the platform is just one factor of the total cost of ownership. Other costs include ongoing maintenance, product development, education, evangelism, security, and adoption.
3. When making business decisions based on experiment results, you must know that the results are backed by a high level of statistical rigor and confidence.
4. Experimentation is more than a technology platform. Successful teams invest in people and process to ensure that cultural change takes root, and that data-driven decision making becomes the norm.

CONTENTS

01

Breadth of Capabilities and Compatibility

Feature flagging and experimentation on their own can be simple. But a platform that has both progressive delivery controls (feature flags, phased rollouts, and 1-click rollbacks) and experimentation, is quite complex because it has built-in capabilities that consistently enable collaboration across teams. The product managers, engineers, data scientists, and marketers who adopt a platform are typically at uneven maturity levels and require unique features to meet their diverse needs. Some advanced features that product development teams often rely on include:

Business critical features:

- Confidence in results
- Advanced targeting
- Remote configuration
- Easy-to-understand UI
- Mutually exclusive experiment groups
- Consistent experiment bucketing
- Ease of QA and troubleshooting
- Documentation
- Compliance (PCI compliance, GDPR, ISO standards)
- Permissions
- Change history/audit log
- Workarounds for CDN caching

These features are essential to creating a basic experimentation or feature management platform. Building a strong statistical model and consistent bucketing helps ensure you get results you can trust that drive your decision making. Ease of QA, permissions, and change history are crucial for distributed teams. Take the time to plan out the capacity of your product and development teams to build and test each of these features.

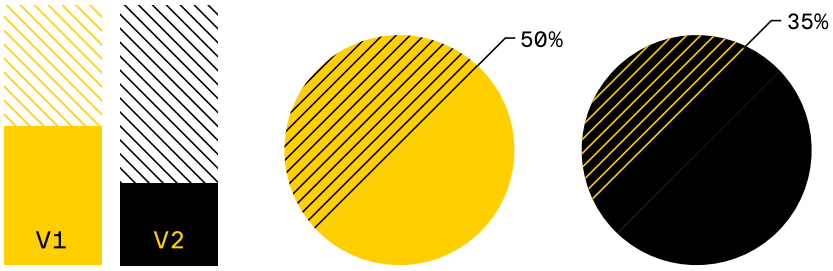
Advanced features:

- Program management of ideas, workflows, searchable repository, program reporting
- Personalization
- Multivariate testing (MVT)
- Results report building and charting
- Ideas and results sharing
- Collaboration functions and chat
- Integrations with JIRA, Slack, and other business-critical solutions
- Real-time experiment results

Building these features on your own can be time- and resource-intensive. For example, adding in the capability to keep experiments mutually exclusive to avoid contaminating results. When multiple teams are testing on the same group of users, it can become difficult to keep sample groups from colliding and maintain clean samples.

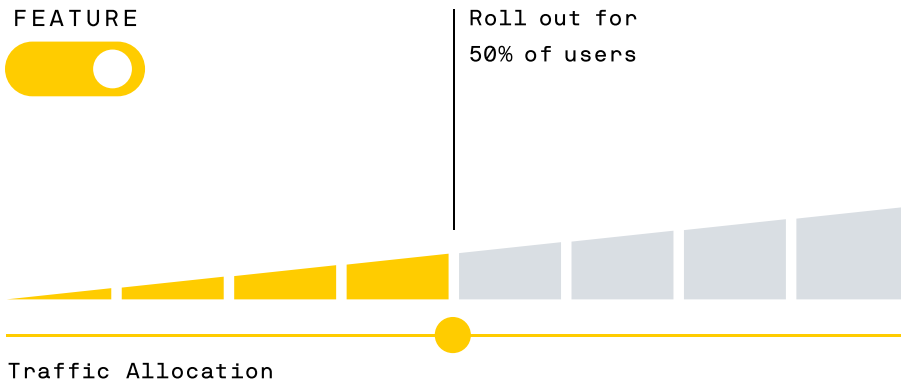
Software teams at companies like Google and Facebook, which use progressive delivery to roll out every feature as an experiment, have developed complex gatekeeping platforms for ensuring that every team can experiment without getting in each other's way. They have sophisticated

targeting platforms based on their user data that ensure test groups don't overlap. For in-house solutions, it's worth considering the time it will take to build in mutual exclusivity and gatekeeping, and to make those features accessible for product managers and engineers.



Another example is calculating statistical significance when multiple product managers want to make decisions faster than analysts can pull the data, run a significance test, and report on results. To solve this particular problem, Uber's team of data scientists have developed their own sequential testing model similar to Optimizely's Stats Engine to enable their product managers to continuously monitor rollouts and make decisions as soon as results are significant.

When best-in-class engineering teams build progressive delivery and experimentation platforms, they often add remote configuration tools to enable product managers and developers to create feature flags and control experiments without redeploying code. A sophisticated UI enables users to understand which features are on/off, for whom, for which environments, and see experiments that are running and how they are performing.

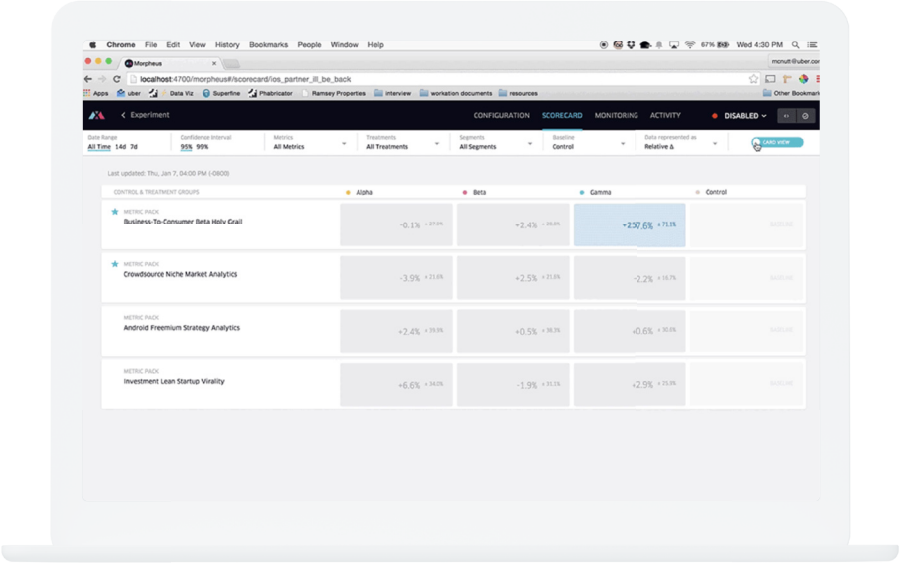


When evaluating whether your team requires remote configuration capabilities, consider how quickly you'll need to make changes to experiments and how important experimentation velocity is to the team. A commercial, enterprise-ready platform will include the ability to start and stop an experiment, change traffic allocation, or update targeting conditions in real time from a central dashboard without a code deployment. This enables your team to decouple code deployment from release and manage experiments autonomously—experiment code can be pushed to production one day, then you can start the experiment at a later time. If the feature or experiment is hurting key metrics, or a bug is discovered post-release, you can instantly stop sending traffic to that feature or experiment variation without touching code. This means validating quality and performance in your production environment, faster iteration on experiments, more control for product and business users, and less time spent releasing updates just to change the traffic allocation or experiment status.

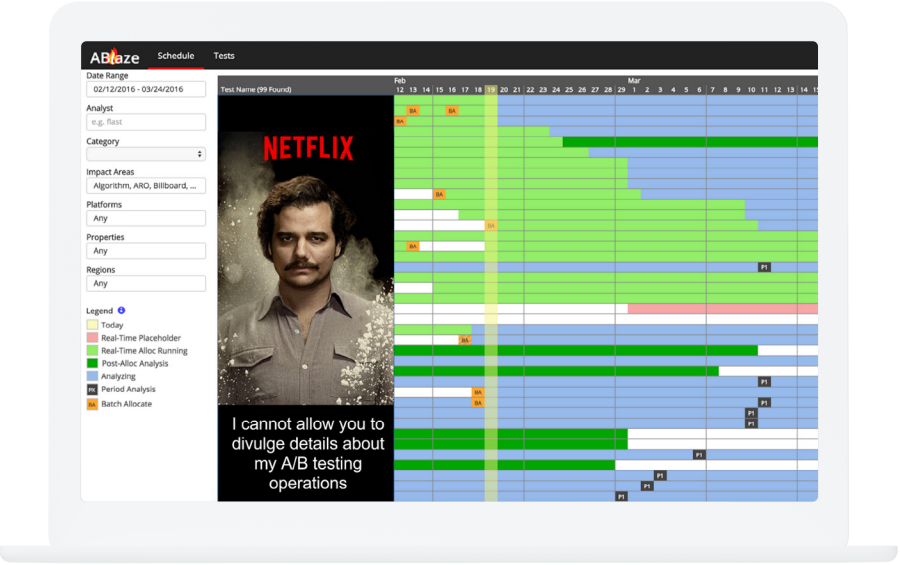
Other business-critical features of a progressive delivery and experimentation platform are easy-to-understand UI, and documentation. As more users onboard to the platform from cross-functional roles like product management, analytics, and design, coordination and up-to-date documentation becomes critical. An intuitive UI will enable quick analysis of results, provide a view into running experiments and feature flags, and display tools for avoiding conflicts. Sound documentation will support a consistent way of understanding and working with the platform, as well as education about best practices in experimentation strategy, implementation, statistical methodology, and how to interpret results, edge cases, and upleveling the program.

Uber's Morpheus, Airbnb's Experimentation Reporting Framework (ERF), and Netflix's A/Blaze solution serve hundreds of internal customers. To make sure their entire teams are able to configure and control experiments, these companies have devoted significant resources to developing a UI that caters to diverse needs and can visualize results on a dashboard.

While in-house solutions serve many internal teams developing digital experiences, many times marketing teams rely on Optimizely's easy-to-use web editor to deliver amazing customer experiences .



Uber's Internal system, Morpheus, focuses on data visualizations that enable skateholders-developers and product managers alike-to make quick decisions about experiments and rollouts.



Netflix's internal A/Blaze platform has a UI component designed to show whether a test you are planning might conflict with one by another group.

Open platform that improves your ability to progressively deliver software

A platform that integrates with tools your team uses every day will enable more seamless, efficient processes around progressive delivery and experimentation.

A robust platform is built on APIs to automate certain tasks or integrate more deeply into development teams' workflows. When a platform allows you to manage projects, campaigns, experiments, audiences, pages, events, attributes, exclusion groups, and usage using code, your entire workstream is enriched. This enables tasks like creating feature flags from an automated script, building custom dashboards of experiment results, and connecting your experiments to other project management tools. Integrating with apps like JIRA and Slack delivers cross-functional collaboration for all features and experiments, allowing developers to spend less time figuring out how to run experiments, and more time working on customer-facing feature work.

Once your experiments are in play, features like callbacks for variation and experiment data can alert you immediately to key events, such as when a user partakes in an experiment, and when a user accesses a feature or variable. This level of information can enhance data you already have about your users and improve analysis.

Easy access to your events data, through an API for instance, enables your team to dig deeper into experiment metrics, beyond the performance of your hypothesis. A platform that can export this events data accommodates scenarios like troubleshooting results, creating data dashboards, and enriching outside data. A well-defined set of events to capture is critical when you want to combine them with other data sources for deeper insights into business performance. Having a flexible and open experiment data pipeline makes this possible.

As more teams across your organization incorporate client-side and server-side rollouts and testing into their development workflow, you may also need support for more languages for different parts of the application. Your web application backend might be built in Python, for example, while the mobile team requires support for JavaScript, iOS, and Android. Having support on installing, initializing, and using a variety of languages—not to mention the maintenance of languages—is a full-time job. Therefore, the ability to support microservice implementations is becoming more common and is something to consider when building and/or buying.

Many open source libraries may be written in just a few of the languages you need, and even when they have more options, you'll want to see how recently the source code was updated to ensure they're being maintained. If building yourself, determine upfront whether you are willing to support the needs of multiple teams working in multiple languages, should that become a consideration in the future.

"In order to release a feature that satisfies our users' needs, we need to be able to turn features on and off, specify audiences, test different variations, and integrate easily with current apps."

Compass

Robust depth of targeting

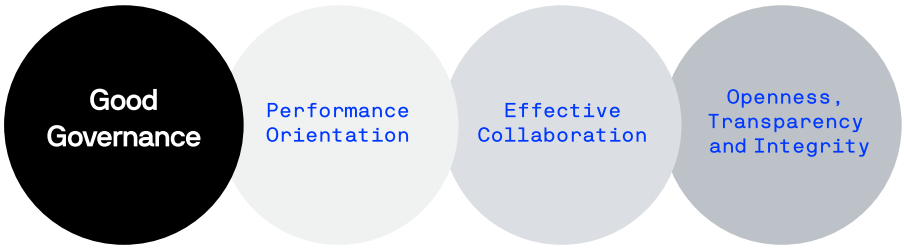
The more scenarios you can test, the more confidence you'll have around how and where to improve your product. Being able to test in as many environments as possible will lead to better builds.

As your software delivery process and experimentation program matures, you'll have to consider the delivery environments and the depth of who you want to target in your experimentation program. In-house solutions can become unwieldy for many teams when configuring feature flags and experiments in multiple environments, as well as the added layer of complexity in testing for differences across users, such as location, loyalty, and behavior.

Having a variety of ways in which to execute feature rollouts provides the most flexibility in who you want to target. The most common ways to execute feature rollouts is through targeted rollouts, where only a random percentage of users or key segments are exposed before rolling out to everyone.

Good governance features

Good governance, including aspects like change history and permissions, makes it easier for teams to ramp up quickly, collaborate faster, and build a more viable product.



Good governance features can help you tame the complexity in any platform by increasing visibility and establishing processes.

Progressive delivery and experimentation are potent tools for product development teams to drive better outcomes, whether rolling out and testing new features, optimizing customer experiences, or improving marketing funnel performance. But these techniques can easily become complex. Good governance features can help you tame the complexity in any platform by increasing visibility and establishing processes.

Features with a version control capability give teams critical visibility into what's happened in their feature flagging, rollouts, or experimentation worlds. If you're an engineer responding to an outage, being able to see changes to feature flag status, traffic allocation, and other variables that

can be changed outside of code deploys, or platform configuration, is imperative. Drilling down to view changes for specific date-time windows makes it simple to quickly evaluate if a change to an experiment or feature could be part of the problem.

As more individuals participate in the product development process, controlling access levels becomes a vital component for any organization trying to deliver software faster and move up the experimentation maturity curve toward a culture of experimentation. Using a platform with permission capabilities enables product owners or business stakeholders to release software when the business is ready, freeing up engineering time to develop software. Creating permissions for in-house solutions can become complex quickly. Especially if your solution needs to integrate with workflow tools like JIRA, you'll have to set levels of access at multiple touchpoints. Consider whether you'll need different permission structures for who can create, view, or publish features and experiments.

As part of your evaluation, ask:

- Which features are important to your team, and how long will it take to build them? Are they available with the open source library you've chosen?
- Will teams be able to control experiments remotely without additional code deploys?
- What kind of data visualizations will you need to help stakeholders understand results?
- Which APIs do you need to integrate with?
- How will you support multiple software languages or multiple platforms?
- What types or quantities of experiments are expected within your organization across stakeholders?
- Will you need a solution with governance capabilities built in?

The bottom line:

- A robust platform delivers feature flagging, phased rollouts, remote controls, 1-click rollbacks, and experimentation, serves an entire organization, and has built-in capabilities to enable collaboration across teams.
- Open platforms easily integrate with everyday tools to make workflows possible.
- Having a variety of ways in which to execute feature rollouts provides the most flexibility in who you want to target.
- Built-in governance features will enable visibility and help establish processes to make sure your platform can perform at scale.

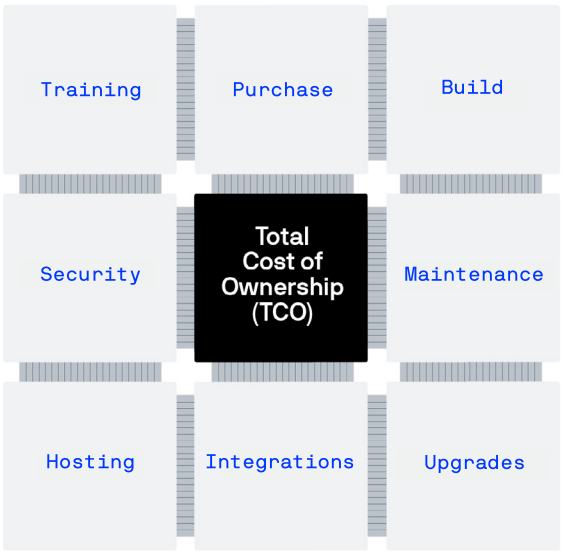
02

Total Cost of Ownership

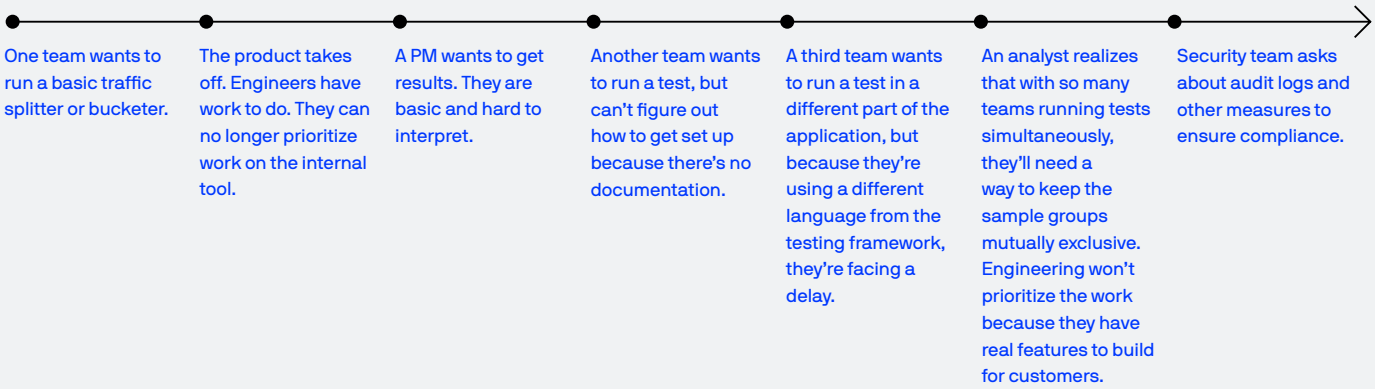
Building in-house solutions or adopting an open source framework typically comes with a relatively small upfront investment. Over time, however, additional features and customizations become necessary as more client-side and server-side teams use the platform, and maintenance burdens begin to distract engineers from core-product focus. The same is true for data scientists who are bogged down by performing experiment analysis, pulling them away from building pipelines, modeling data, and providing higher order analysis.

Committing to build an in-house solution yourself is a commitment to continuing to innovate on experimentation and develop new functionality to support your teams.

One product manager we spoke with said it took them four years—through fits and starts throughout the entire building cycle—to expand their in-house solution from code-based server-side testing to include client-side testing. At first, their solution could only split traffic, but couldn't be managed in a UI, and it took a few months to build. Next, they spent 18 months building a UI that could manage server-side tests, but it still had minimal targeting capabilities and simple permissions. After another six months, they added client side testing, live QA and additional targeting capabilities.



Lifespan of an internally Built A/B Testing Framework



Manageable maintenance cost

Unlimited flexibility to integrate with your application codebase, existing platforms, and processes can be costly in the long run.

Building an in-house progressive delivery and experimentation solution is a major business decision. The upfront cost may appear manageable at first, as a drop in the bucket of your total budget, however the long-term maintenance and care is where you'll spend most of your resources. The total cost of ownership calculation often ends up in favor of buying a commercial platform—even when factoring in additional support, headcount, or services costs—because keeping an in-house solution in working order can eat up unallocated time and money.

While world-class teams have the bandwidth to constantly monitor performance, fix bugs, and make improvements in their in-house solutions, many teams find that building a robust feature flagging tool with flexible targeting and/or an enterprise-scale experimentation platform is not their core competency, and that unlimited flexibility means more opportunities for something to break. Companies that successfully scale an in-house solution have engineers on staff who are dedicated to the ongoing maintenance it requires.

On many smaller teams, the person who built the feature flagging and/or experimentation tool holds much of the knowledge of how it works and how to troubleshoot. Typically, documentation and support become secondary priorities in favor of working on customer-facing features. For in-house solutions, you'll need to identify who will own the code, fix it when it breaks, and who will be responsible for minimizing technical debt. We've heard from

engineering teams that it's difficult to prioritize maintenance because it's not their core competency, so product development on in-house solutions is deprioritized as other customer-facing projects come up.

Without a team to champion their in-house solutions, feature development can fade into obsolescence because no one owns the platform nor manages its processes. Additionally, if you're depending on an open source library, look at the contributions and take the time to understand how it's being maintained over time. Who's responsible, and how often are they updating it? If something breaks, how quickly will it get fixed?

Technical scaling to onboard new teams fast

Consider the time and money it will take to scale your platform and bring new teams up to speed.

It's easier to train new teams on how to use a platform, versus asking each team to build their own, especially when there are different implementations. Companies that developed in-house solutions have faced challenges around implementation in microservices environments and scale across multiple applications as use cases grow.

One company told us their engineering team would often spend 8-20 hours setting up each test with an in-house platform that was poorly documented and haphazardly maintained. Now, with Optimizely Full Stack, they spend under 30 minutes. Without documentation about how the platform worked, engineers needed to relearn the platform every time. And without a clear owner maintaining the platform, they were constantly debugging integrations with their codebase and infrastructure in the process of setting up each test.

With a commercial platform, having a partner and solution that can centralize decision making for your applications will streamline the technical scaling. Commercial options like Optimizely Agent can help companies achieve production scale faster through features like an experimentation and feature flagging microservice deployed alongside your application, and embedded SDKs that provide experimentation and feature flagging functionality. In addition, support services like account management and 24/7/365 tech support can help teams scale faster and encourage wider adoption.

Once experimentation starts to take off with one or a few teams, more people will want to run experiments and different use cases will come to life. For Booking.com, which is famous for running thousands of experiments

with their in-house solution, they encountered a scaling challenge with the first version of their platform that supported running only a handful of experiments. As more teams and users wanted to experiment, they needed to enhance their infrastructure to support mobile app testing, customer service, and backend application testing. Each team brought on new touch points and new challenges over time.

"Experimentation has become so ingrained in Booking.com culture that every change, from entire redesigns and infrastructure changes to bug fixes, is wrapped in an experiment… Such democratization is only possible if running experiments is cheap, safe and easy enough that anyone can go ahead with testing new ideas, which in turn means that the experiment infrastructure must be generic, flexible and extensible enough to support all current and future use cases."

Booking.com

Another company famous for running hundreds of experiments with their own A/B internal testing platform and Experimentation Reporting Framework (EFR) is Airbnb. When they scaled their experimentation program from a few dozen to hundreds of experiments, they were collecting a lot more data and computing thousands of distinct metrics per day. But the data analytics pipelines that were set up early on struggled to handle the massive amounts of data that came with a scaled program. Experiment analysis that used a simple script to query for results once a day, started to take more than 24 hours to run. Business-critical experiments had no access to real-time data, which meant there was very little line of sight into business metrics. Scaling can suffer and even come to a halt until resources are allocated to fix data scaling issues.

"Today we compute ~2,500 distinct metrics per day and roughly 50k distinct experiment/metric combinations. We also have introduced several advanced features (e.g. dimensional cuts, global coverage, pre-assignment bias checking) that add to our scaling challenge."

Airbnb

Low opportunity cost for the business

Every unexpected engineering task steals time from developing customer-facing features. In-house solutions can come with hidden opportunity costs that may lead to budget and schedule overruns.

Understanding the potential opportunity cost of an in-house solution can help you better predict the overall cost of ownership. In this case, opportunity costs include the time spent building an in-house solution

that could have been allocated to shipping more code faster and with confidence.

According to one study from the Harvard Business Review, IT projects that resulted in cost overruns were, on average, over budget by 27 percent. But that wasn't the surprising part. One in six of the 1,471 projects included in the study achieved a black swan status, topping 200 percent over budget, on average, while 70 percent went over schedule. These are significant costs—in both time and money—that need to be closely managed if you choose to build your own solution.

One product owner told us they spent 18 months building a UI for a server-side in-house testing solution that still didn't provide the flexibility to do all the tests they wanted. They were limited to just code deployments and releasing experiments. It took another six months to build client-side capabilities and run quick prototyping tests.

As part of your evaluation, ask:

- What functionality will your team need in the long term that may pull engineers away from focusing on building features?
- Who will own long-term maintenance and documentation?
- Who will champion your platform, develop best practices, and evangelize experimentation-driven product development?
- Does your development team have the bandwidth to keep fixing things when they break, or to help others internally to use the tool?
- How will you onboard new teams? What factors will complicate your implementation?
- Are you willing to wait for an application to be built rather than implement an off-the-shelf platform?

The bottom line:

- Understand your team's feature requirements upfront. Do discovery on the more complex features you might need down the road, and plan a roadmap for the internal solution so that you can plan for the engineering work needed beyond initial development.
- A maintenance plan that dedicates engineering time for bug fixes will help you evaluate the total cost of ownership.
- Many engineering teams have difficulty prioritizing maintenance of in-house solutions because it's not their core competency.
- Having a partner and platform that can centralize decision making for your applications will streamline the technical scaling.
- Be sure to factor in the opportunity costs of pulling engineers away from developing customer-facing features.

03

Statistical Rigor

The main goal of running experiments is to make data-driven decisions about your product, faster. When implemented correctly, progressive delivery and experimentation can help product development teams quickly identify which features and changes will drive business value, and which will fail. The best teams embrace failure and try to fail as fast as possible, so they can move on to ideas that work.

To fail faster and learn quickly, your teams need to trust that tests are being run correctly, and that the results are accurate.

Instrumentation for data and statistical analysis

Many companies underestimate the difficulty of collecting data reliably and maintaining their analytics pipelines over time. When events aren't tracked correctly or analytics integrations stop working, it results in delays and slower experimentation velocity.

One company told Optimizely that they set up an experiment with an open source framework, and didn't realize for days that it wasn't tracking events. When it came time to report results to the executive team, all they could say was "we set up the tracking wrong."

To get accurate results, you'll need to know that events (e.g. page views, interactions, custom items) are being tracked reliably, and that the data pipeline is correctly filtering and aggregating those events, processing results in a consistent manner, and storing and retrieving results in a quick fashion. This often involves a data scientist or data engineer and can be time consuming to get right. In addition to data collection, processing and computing a statistical model that people agree on and is widely understood is a significant undertaking for a data scientist. For the majority of organizations that don't have armies of data scientists to create these capabilities, commercial products make sense.

It is also important that data be available and integrate easily with different applications, should you want to perform deeper analysis between experiment data and other key business data. Typically, this instrumentation

requires an on-site data scientist who's experienced in building data pipelines as well as a statistician who can develop and oversee the statistical analysis models.

Your engineering and analytics teams will need to work together to ensure that your progressive delivery and experimentation platform works well with your analytics stack, and that accurate results are accessible to business stakeholders in a timely fashion.

Results and analyses that are fast and reliable

In many organizations, the responsibility for setting a sample size and analyzing results typically falls on an analytics or data science team. But these processes can quickly become bottlenecked as additional teams begin to run more simultaneous experiments.

A data scientist at one software company told Optimizely that she currently goes through a manual process of setting an experiment duration in advance, collecting data, running it through a home-built script for analysis, and then creating a report of the results for business stakeholders. The current process takes days, and may take lower priority than the other projects she has. As a result, the business can't run as many tests, because each test takes hours.

Analysts at another company told us they tried to automate the analysis of A/B test results, in order to enable PMs and marketers to run more experiments, but ultimately couldn't do it because it was such a complex process. One company that was able to automate the process for stakeholders found the process often timed out, and even when they produced results, nobody trusted them.

One product manager told us they were only able to track a few key metrics with their internal platform because it was too hard and expensive to capture a variety of events, so they were limited to capturing funnel progression and purchase conversion. As a result, the product teams lacked additional insights on behavioral events that could have influenced other metrics.

With an out-of-the-box solution like Optimizely, everyone can access test results at any time, freeing your analysts from the burden of analyzing each test manually. Optimizely's Stats Engine also ensures that results are always valid with its combined approach of sequential testing and false discovery rate control, so team members can continuously monitor results in real time without invalidating them. The most successful teams take time upfront to define and instrument the metrics that matter most, and are then able to run a majority of their tests without an analyst's help, which supports scaling and

adoption across the organization. For experiments that entail further analysis, Optimizely provides access to the experiment data via an export and API.

As part of your evaluation, ask:

- Who will analyze test results? Is the team appropriately staffed to support the volume of tests that your organization will run?
- How quickly will data be available to stakeholders to make decisions? Are there results and statistical analyses available outside the box, and are they formatted in a way that stakeholders can understand?
- How will you instrument and QA your tests? If you choose an open source framework, what will need to be built on top of it to make everything work with your platform?
- Do you have the right skill set in place to support building data pipelines and statistical models?

The bottom line:

- Ensure your data science/analyst team is involved in any evaluation of an A/B testing solution, and is a part of your process for getting data out of the platform and analyzing it.
- If choosing to build an in-house solution, budget enough time from engineering and analytics to build, operate, and maintain the analytics pipeline and integrations.
- A platform that works with your existing analytics stack will save time and grow confidence in results.

"As our business grows, we are starting to see we need to be able to 'learn from reality' by validating our hypothesis with our real users. With the Experiments Service, all data analysis is done on-demand, we need a better way to analyze features."

Compass

04

Adoption and Enablement at Scale

For progressive delivery and experimentation to be successful, there needs to be a mindset change in creating processes and initiating evangelism for widespread adoption. These changes happen with the right people and process—it is not just a technology shift.

The practice of wrapping every feature in an experiment, then rolling out a change based on results takes a concerted effort across product, engineering, and analytics teams—plus every individual scrum team, squad, or group. When it comes to analysis, the cross-functional agreement to define the metrics and philosophy around stats that matter, can be as important, if not more so, than the technology that enables testing.

Centralized technology plus partnerships for a culture that sticks

To go from dozens of experiments to hundreds will require a platform that can centralize both feature flagging, gradual rollouts, and experimentation, plus a partner to help culture changes take root.

The top teams in the world, who have invested dozens, or even hundreds of engineers over many years to build out in-house solutions, have also invested in data scientists, product managers, and internal enablement to adopt a process of experimentation-driven product development and run it at scale. If you choose a platform that enables process management, you can minimize the complexity of running more than hundreds of experiments or rollouts simultaneously. Optimizely helps centralize both feature flagging, rollouts, feature configurations, and experimentation under one UI. All the experiment ideas and workflows necessary for experimentation are under a program management section, making it easy to know what is currently running, what will run, and what has already run, as well as the prioritization and ranking of each.

Having a partner to work alongside you as your experimentation platform scales in volume and users, can help make cultural changes take shape. Companies that use Optimizely are supported with dedicated customer success managers and professional services that set you up for strategic

"As companies try to scale up their online experimentation capacity, they often find that the obstacles are not tools and technology but shared behaviors, beliefs, and values."

Stefan Thomke

Experimentation Works: The Surprising Power of Business Experiments

success and implementation by sharing best practices from our experience—because feature flagging, rollouts, and experimentation is our core competency.

As part of your evaluation, ask:

- Who needs to be on board for your team to scale from dozens to hundreds of experiments?
- In addition to technology, what people and processes will you need to fully adopt an experimentation culture?
- When a new employee needs help setting up a flag or experiment, where will they go for help?
- If a critical experiment breaks overnight, who gets paged to fix it?

The bottom line:

- For progressive delivery and experimentation to be successful, there needs to be a mindset change in creating processes and initiating evangelism for widespread adoption.
- Having a partner to work alongside you as your experimentation platform scales in volume and users, can help make cultural changes take root.
- Centralizing functionality creates visibility and a consistent workflow for widespread adoption.

Decisions, decisions...

As with most software, the decision to build, buy, or complement a progressive delivery and experimentation solution comes down to the requirements and needs of each individual company. Every company is different, and every team has its own unique needs. The key to making a decision is to be informed, ask the right questions, and know what you're getting into ahead of time.

If you're ready to invest in a progressive delivery and experimentation solution, Optimizely is a complete platform that enables every team in your organization to deliver software faster, with less risk and with confidence. We have over ten years of experience in continuous experimentation and personalization across websites, mobile apps, and connected devices.

Visit https://www.optimizely.com/platform/ today to get started.

Cheklist

Does Your Platform Meet Your Requirements?

Technology leaders consider a number of factors when investing in enterprise software, including stability and performance, configurability, scalability, security, and costs associated with implementation and maintenance. These are often addressed as part of a requirements gathering exercise, a key step in evaluating any software platform. Here's a quick guide for evaluating whether a progressive delivery and experimentation platform is enterprise-ready and meets your requirements.

Build your requirements checklist:
Agreeing on requirements is a key step in evaluating whether to build, buy, or complement a progressive delivery and experimentation platform. Gather stakeholders from across the business who will use the platform and decide together which requirements matter most for your business. In addition to features and functionality that your team requires, make sure you consider table stakes factors such as performance, scalability, security, and adoption.

Stability and performance

- Performance impact: Application and page load speed is critical to providing a good user experience. How will the solution impact performance?
- Performance monitoring: Will you be able to continuously monitor performance?
- Platform uptime and availability: What is the expected stability? How will uptime be guaranteed?

Scalability

- Technology and backend growth: Is the solution designed to collect, process, and report on massive data sets? Does the technology support multiple languages and implementation options like a microservices environment?
- Usage growth: Is the solution designed for advanced experimentation programs and large, multi-team organizations?
- Multi-channel enablement: Is full functionality available to support experimentation on multiple digital channels? How important is this to your teams?

Security and compliance

- Compliance standards: Does the solution meet internal security and privacy requirements or external standards such as PCI and SOC 2?
- Privacy: Does the solution handle PII or other sensitive information? If so, is it properly handled?
- Audit logs: Can you audit every change to features and experiments?
- Access controls and permissioning: Can you control who has access to the solution through enterprise security measures such as single sign-on and two-factor authentication?

Features and functionality

1. Breadth of capabilities and compatibility

- Confidence in results
- Audience/advanced targeting
- Remote configuration
- Simple UI to test prototypes or MVPs
- Mutually exclusive experiment groups
- Consistent experiment bucketing
- QA tool
- Documentation
- Compliance (PCI compliance, GDPR, ISO standards)
- Permissions
- Change history/audit log
- Workarounds for CDN caching
- Built-in governance
- Single sign-on
- Program management
- Multivariate treatment
- Results report building and charting
- Ideas and results sharing

- Collaboration functions and chat
- Integrations with JIRA and Slack
- Real-time experiment results
- Global holdout
- Traffic sampling
- State persistence
- Automation
- Whitelisting
- Logging
- Feature flags
- Feature rollouts
- Kill switches
- Scheduling
- Staging environments
- Audience definitions
- Webhooks
- REST API
- Event dispatching
- Event storage
- Sessionization
- Statistical methods
- Metric definitions
- User aliasing
- Segmentation
- Data warehouse integration
- Visualizations

2. Total cost of ownership

- New product development costs
- Engineering time dedicated to ongoing maintenance
- Data scientist time spent analyzing tests instead of performing higher order analysis
- Technical scaling capabilities including simplified implementation across multiple applications and microservices
- A partner to support training and continuous support in adoption for speed to value
- Data collection, storage, processing

3. Statistical rigor

- Compatibility with your analytics stack
- Data collection, storage, processing
- Computing results with a reliable statistical analysis model
- Visualizations
- Ability to involve your data science/analyst team
- Ability to build, operate, and maintain analytics pipeline and integrations

4. Adoption and scale

- Willingness for mindset and culture changes
- Evangelism
- Process scaling
- Program management
- Partner to enable scale and adoption

Calculating Total Cost of Ownership

Here are some key costs to help you begin calculating the total cost of ownership of building or continuing to build an enterprise-ready progressive delivery and experimentation solution.

This TCO framework considers seven key elements of costs:

1. Software licensing costs: the cost of software licensed, typically in the form of annual subscriptions, to enable an in-house solution.
2. Build costs: the engineering hours it takes for the solution to achieve parity. This can be a one-time cost incurred over the first one or two years.
3. Hosting costs: the cost of hosting the application in-house or through a third-party vendor (e.g. AWS etc.). This is typically a recurring annual cost.
4. Support costs: the costs to enable and train internal users. This is typically a recurring annual cost.
5. Maintenance costs: the effort to maintain the technology, including data backup, archive, single sign-on or security, bugs fixes, and application administration. This is typically a recurring annual cost.
6. Integration costs: the cost to develop and maintain key integrations such as analytics, CRM, and JIRA. This is generally a one-time cost and includes the effort required to develop and deploy the integration.
7. Upgrade costs: the cost to perform a major version upgrade of the application along with the underlying software platform. This is typically incurred every three years to account for any major upgrade of the application, such as the addition of a major capability accompanied with the upgrade of the underlying software development platform that may also require a review of integrations.

As a best practice, a TCO analysis should evaluate costs over the course of three to five years. This way you account for a major upgrade and avoid surprise costs.

	Build	Buy	Complement
SOFTWARE LICENSING COSTS			
BUILD COSTS			
HOSTING COSTS			
SUPPORT COSTS			
MAINTENANCE COSTS			
INTEGRATION COSTS			
UPGRADE COSTS			
KEY TAEKAWAY	Initial investment may seem low, but the opex can be potentially high over a 3-5 year period. Consider availability of expertise to build advanced capabilities.	Initial investment may seem high, but the TCO over a 3-5 period is generally low compared to the build option, and there is the added benefit of availability of continuous innovation.	The costs may fall in between the build and buy options. Analyze the complexity of maintaining multiple platforms, user experience, and adoption of multiple platforms.