4
Experimentation Platform and Culture

If you have to kiss a lot of frogs to find a prince, find more frogs and kiss them faster and faster
– Mike Moran, Do It Wrong Quickly (2007)

As discussed in Chapter 1, running trustworthy controlled experiments is the scientific gold standard in evaluating many (but not all) ideas and making data-informed decisions. What may be less clear is that making controlled experiments easy to run also accelerates innovation by decreasing the cost of trying new ideas, as the quotation from Moran shows above, and learning from them in a virtuous feedback loop. In this chapter, we focus on what it takes to build a robust and trustworthy experiment platform. We start by introducing experimentation maturity models that show the various phases an organization generally goes through when starting to do experiments, and then we dive into the technical details of building an experimentation platform.

Important organizational considerations include leadership, process, and training, whether the work should be done in-house or outsourced, and how the results are ultimately used. The technical tools will support experiment design, deployment, scaling, and analysis to accelerate insight.

Experimentation Maturity Models

Experimentation maturity models (Fabijan, Dmitriev and Olsson et al. 2017, Fabijan, Dmitriev and McFarland et al. 2018, Optimizely 2018c, Wider Funnel 2018, Brooks Bell 2015) consist of the phases that organizations are likely to go through on the way to being data-driven and running every change through A/B experiments.

We use these four phases of maturity, following Fabijan et al. (2017):

1. **Crawl**: The goal is building the foundational prerequisites, specifically instrumentation and basic data science capabilities, to compute the summary statistics needed for hypothesis testing so that you can design, run, and analyze a few experiments. Having a few successful experiments, where success means that the results meaningfully guide forward progress, is critical to generating momentum to progress to the next stage.
2. **Walk**: The goal shifts from prerequisites and running a few experiments to a focus on defining standard metrics and getting the organization to run more experiments. In this phase, you improve trust by validating instrumentation, running A/A tests, and sample ratio mismatch (SRM) tests (see Chapter 21).
3. **Run**: The goal shifts to running experiments at scale. Metrics are comprehensive and the goal is to achieve an agreed upon set of metrics or going all the way to codifying an OEC that captures tradeoffs between multiple metrics. The organization uses experimentation to evaluate most new features and changes.
4. **Fly**: Now you are running A/B experiments as the norm for every change. Feature teams should be adept at analyzing most experiments—especially the straightforward ones—without the help of data scientists. The focus shifts to automation to support this scale, as well as establishing institutional memory, which is a record of all experiments and changes made, enabling learning from past experiments (see Chapter 17) to sharing surprising results and best practices, with a goal of improving the culture of experimentation.

As a rough rule of thumb, in the Crawl phase, an organization is running experiments approximately once a month (~10/year), and it increases by 4–5x for each phase: organizations in the Walk phase will run experiments approximately once a week (~50/year), Run is daily (~250/year), and Fly is when you reach thousand(s)/year.

As an organization progresses through these phases, the technical focus, the OEC, and even the team set-ups will shift. Before we dig into the technical aspects of building an experiment platform in the Walk, Run, and Fly phases, let's highlight several areas for organizations to focus on regardless of phase, including leadership and processes.

Leadership

Leadership buy-in is critical for establishing a strong culture around experimentation and embedding A/B testing as an integral part of the product development process. Our experience is that organizations and cultures go through stages in learning to experiment as well (Kohavi 2010). The first stage, which precedes any experimentation, is hubris, where measurement and experimentation are not necessary because of confidence in the HiPPO (Highest Paid Person's Opinion). Next is measurement and control, where an organization starts measuring key metrics and controlling for unexplained differences. As Thomas Kuhn notes, paradigm shifts happen "only through something's first going wrong with normal research" (Kuhn 1996). However, there is still a strong dependence on the HiPPO and entrenched norms, beliefs, and paradigms, as an organization may reject new knowledge that is contradictory per the Semmelweis Reflex (Wikipedia contributors, Semmelweis reflex 2019). It is only through persistent measurement, experimentation, and knowledge gathering that an organization can reach a fundamental understanding, where causes are understood, and models actually work.

To reach this last stage, in our experience, buy-in from executives and managers must happen at multiple different levels and include:

- Engaging in the process of establishing shared goals and agreeing on the high-level goal metrics and guardrail metrics (see Chapter 18) and ideally codifying tradeoffs as steps to establishing an OEC (see Chapter 7)
- Setting goals in terms of improvements to metrics instead of goals to ship features X and Y. There is a fundamental shift that happens when teams change from shipping a feature when it does not hurt key metrics, to NOT SHIPPING a feature unless it improves key metrics. Using experiments as a guardrail is a difficult cultural change, especially for large, established teams to make as they shift towards a data-informed culture.
- Empowering teams to innovate and improve key metrics within the organizational guardrails (see Chapter 21). Expecting ideas to be evaluated and for many of them to fail and showing humility when their ideas fail to move the metrics they were designed to improve. Establishing a culture of failing fast.
- Expecting proper instrumentation and high data quality.
- Reviewing experiment results, knowing how to interpret them, enforcing standards on interpretation (e.g., to minimize p-hacking (Wikipedia contributors, Data dredging 2019)), and giving transparency to how those results affect decision making.
- As discussed in Chapter 1, many of the decisions that experiments can best help inform are optimization; a long sequence of experiments can also inform overall strategy as well. For example, Bing's integration with social networks, such as Facebook and Twitter, was abandoned after experiments showed no value for two years. As another example, evaluating an idea like

whether including videos in promotional e-mails results in higher conversion rates would require testing multiple implementations.
- Ensuring a portfolio of high-risk/high-rewards projects relative to more incremental gain projects, understanding that some will work, and many —even most—will fail. Learning from the failures is important for continued innovation.
- Supporting long-term learning from experiments, like running experiments just to collect data or establish return-on-investment (ROI). Experimentation is not just useful for making ship/no-ship decisions on individual changes, but also holds an important role in measuring impact and assessing ROI for various initiatives. For example, see Chapter 5 and long-term experiments (Hohnhold, O'Brien and Tang 2015).
- Improving agility with short release cycles to create a healthy, quick feedback loop for experimentation, requiring establishing sensitive surrogate metrics (see Chapter 7).

Leaders cannot just provide the organization with an experimentation platform and tools. They must provide the right incentives, processes, and empowerment for the organization to make data-driven decisions. Leadership engaging in these activities is especially crucial in the Crawl and Walk maturity phases to align the organization on goals.

Process

As an organization moves through the phases of experimentation maturity, establishing educational processes and cultural norms is necessary to ensure trustworthy results. Education ensures that everyone has the basic understanding to do a good job at designing and executing trustworthy experiments and interpreting the results correctly. The cultural norms help set an expectation of innovation, celebrating surprising failures and always wanting to learn. Note that this is an ongoing challenge, as at a summit in 2019 with 13 online companies on experiments, establishing cultures and process that encourage experimentation and innovation continues to be a challenge (Gupta, Kohavi et al. 2019).

For education, establishing just-in-time processes during experiment design and experiment analysis can really up-level an organization. Let's consider this example from Google: When experimenters working on search wanted to run an experiment, they had to complete a checklist that was reviewed by experts. The checklist included basic questions like, "What is your hypothesis?" and "How big of a change do you care about?" and went all the way through to power analysis questions. Because trying to teach everyone to do a proper power analysis was unrealistic, the checklist also helped ensure that experiments were sufficiently powered by linking to a power calculator tool. Once the organization was sufficiently up-leveled, the search organization no longer needed such an explicit checklist process.

Generally, experimenters only require hand-holding the first few times they run an experiment. They get faster and more independent with every subsequent experiment. The more experienced the experimenter, the better they explain concepts to teammates and, over time, serve as expert reviewers. That said, even experienced experimenters typically still need help for experiments requiring unique designs or new metrics.

Both LinkedIn and Microsoft (Google too, although not regularly) hold classes to keep employees aware of experimental concepts (Kohavi, Crook and Longbotham 2009). The classes have grown in popularity as the culture grows more accepting of experiments over time.

Analogous to the checklist at experiment design time, regular experiment review meetings for analysis results provide similar just-in-time education benefits. In these meetings, experts examine the results, first for trustworthiness – oftentimes finding instrumentation issues, especially for first-time experimenters – before diving into useful discussions resulting in launch/no-launch recommendations that experimenters could take to their leaders. These discussions broadened the understanding of goal, guardrail, quality, and debug metrics (see Chapter 6) and developers were more likely to anticipate those issues during the development lifecycle. These discussions also established metric tradeoffs that can be codified and captured in an OEC (see Chapter 7). These experiment reviews are also where failed experiments are discussed and learned from: many high-risk/high-reward ideas do not succeed on the first iteration, and learning from failures is critical for the refinement needed to nurture these ideas to success, as well as to decide when to move on (see Chapter 1).

Over time, the experts see patterns in changes, such as seeing how the impact of an experiment relates to similar prior experiments, and how this can be further examined in a meta-analysis (see Chapter 8) that can lead to user experience improvements and updates to key metric definitions. The other unintended, but positive, outcome we noticed about this experiment analysis review forum is that it brought together different teams in a single meeting so that they could learn from each other. Note that we have observed that the teams do need to be working on the same product and share the same metrics and OEC so that there is enough shared context for learning. If the teams are too diverse or if there is insufficient maturity in the tooling, then this meeting can be unproductive. We suspect that this type of review starts being effective in the late Walk or in the Run phases of maturity.

Through the platform or processes, we can share learnings from experiments broadly, be it the meta-learning from experts observing many experiments or the learning gained from a single experiment. This can happen via regular newsletters, Twitter-feed, a curated homepage, a "social network" attached to the experiment platform to encourage discussion (as is done at Booking.com) or other channels. Institutional memory (see Chapter 8) becomes increasingly useful in the Fly phase.

For experimentation to succeed and scale, there must also be a culture around intellectual integrity—the learning matters most, not the results or whether we ship the change. From that perspective, full transparency on the experiment impact is critical. Here are some ways we found to achieve this:

- Compute many metrics, ensure that the important metrics, such as the OEC, guardrail, and other related metrics, are highly visible on the experiment dashboard, so that teams cannot cherry-pick when sharing results.
- Send out newsletters or e-mails about surprising results (failures and successes), meta-analyses over many prior experiments to build intuition, how teams incorporate experiments, and more (see Chapter 8). The goal is to emphasize the learning and the needed cultural support.
- Make it hard for experimenters to launch a Treatment if it impacts important metrics negatively. This can go from a warning to the experimenter, a notification to people who care about those metrics, all the way to even potentially blocking a launch (this last extreme can be counterproductive as it is better to have a culture where metrics are looked at and controversial decisions can be openly discussed).
- Embrace learning from failed ideas. Most ideas will fail, so the key is to learn from that failure to improve on subsequent experiments.

Build vs. Buy

Figure 4.1 shows how Google, LinkedIn, and Microsoft scaled experimentation over the years, with year-1 being a year where experimentation scaled to over an experiment per day (over 365/year). The graph shows an order of magnitude growth over the next four years for Bing, Google, and LinkedIn. In the early years, growth was slowed by the experimentation platform capabilities itself. In the case of Microsoft Office, which just started to use controlled experiments as a safe deployment mechanism for feature rollouts at scale in 2017, the platform was not a limiting factor because of its prior use in Bing, and experiments grew

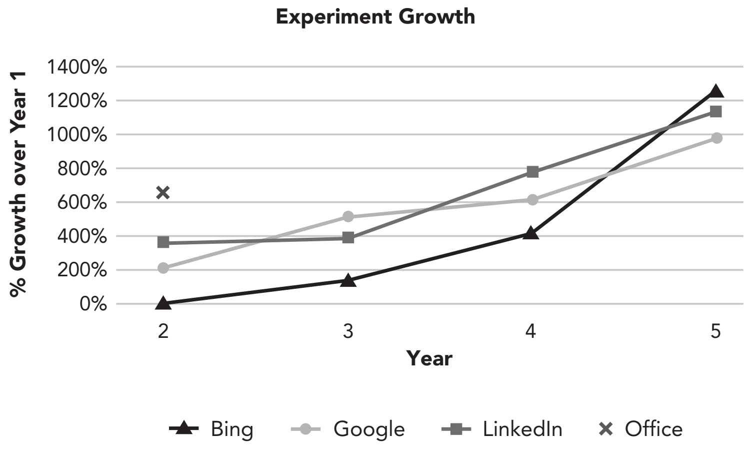

Figure 4.1 Experimentation Growth over the years for Bing, Google, LinkedIn, and Office. Today, Google, LinkedIn, and Microsoft are at a run rate of over 20,000 controlled experiments/year, although counting methodologies differ (e.g., ramping up the exposure from 1% of users to 5% to 10% can be counted as one or three experiments; an experiment consisting of a Control plus two Treatments can count as either one or two experiments)

by over 600% in 2018. Growth slows when the organization reaches a culture of "test everything" and the limiting factor becomes its ability to convert ideas into code that can be deployed in controlled experiments.

While we all have been heavily involved in building in-house experimentation platforms at our respective companies, we are not necessarily recommending that every company should build their own. Especially in the Walk phase, building or buying is an ROI decision (statistics on building versus buying available in Fabijian et al. (2018)). Here are several questions to consider when making that decision.

Can an External Platform Provide the Functionality You Need?

- Consider the types of experiments you want to run, such as frontend vs. backend, server vs. client, or mobile vs. web. Many third-party solutions are not versatile enough to cover all types. For example, solutions based on JavaScript would not work for backend experiments or scale well for many concurrent experiments. Some vendors are strong in one channel, but not others (e.g., great softwar development kit (SDK) for mobile, but weak ability to handle web; or great WYSIWYG editor for web, but a mobile SDK that crashes too often).

- Consider website speed. Several external solutions require additional JavaScript, which is known to slow down page loads (Optimizely 2018, Kingston 2015, Neumann 2017, Kesar 2018, Abrahamse 2016). As shown in Chapter 5, increased latency impacts user engagement.
- Consider the dimensions and metrics you may want to use. For example, some external platforms are limited on what metrics you can compute on experiments. Complex metrics that require sessionization are not possible in external solutions. Even metrics like percentiles, which are commonly used to measure latency where the tail rather than average tends to be more sensitive, are not often supported. Since broad business reporting might have to be built separately, it could also be harder to establish a common language of dimensions and metrics, so ensuring consistency may be more difficult if you buy.
- Consider what randomization unit you want to use and what data sharing is acceptable (e.g., to ensure that user privacy is respected). There are usually restrictions on what information (especially about users, see Chapter 9) can be passed on to external parties, which may be limiting or induce additional costs.
- Is data logged to the external party easily accessible? Do clients need to log to two places (dual-logging)? What happens when summary statistics diverge? Are there tools to reconcile? These are often under-estimated complexities that have reduced trust and raised valid questions about the reliability of different systems.
- Can you integrate additional sources of data? Do you want to integrate purchase data, returns, demographics? Some external systems do not allow you to join such external data.
- Do you need near real-time (NRT) results? These are often useful for quickly detecting and stopping bad experiments.
- Are you running enough experiments that you want to establish your own institutional memory? Many third-party experiment systems do not have institutional memory features.
- Can you implement your feature in its final version? Many WYSIWYG systems require you to re-implement your feature for real post-experiment. At scale, this can be limiting, with a queue of features that need re-implementation.

What Would the Cost Be to Build Your Own?

Building a scalable system is both hard and expensive, as you will see in our discussion on the technical platform issues later in this chapter.

What's the Trajectory of Your Experimentation Needs?

This type of infrastructure investment is about anticipation, that is, how many experiments your organization will run if it truly embraces experimentation, not how many are currently running. If the momentum and demand is there, and the volume may grow beyond what an external solution can accommodate, build. It takes longer to build an internal solution, but integrating an external solution takes effort too, especially if you need to switch to a different solution as the company scales.

Do You Need to Integrate into Your System's Configuration and Deployment Methods?

Experimentation can be an integral part of a continuous deployment process. There is a lot of synergy between experimentation and how the engineering system handles configuration and deployment (see Chapter 15). If the integration is necessary, such as for more complicated debugging situations, it may be harder with a third-party solution.

Your organization may not be ready for the investment and commitment of building your own platform, so it may make sense to leverage an external solution to demonstrate the impact from more experimentation before determining if and when to make a case for building your own experiment platform.

Infrastructure and Tools

In Chapter 3, we showed that there are many ways an experiment can go wrong. Creating an experiment platform is not just about accelerating innovation with experimentation, it is also critical to ensuring the trustworthiness of the results for decision making. Scaling experimentation at a company not only involves building the infrastructure for the experiment platform but also the tools and processes to embed experimentation deeply into the company's culture, development, and decision-making processes. The goal of an experiment platform is to make experimentation self-service and minimize the incremental costs of running a trustworthy experiment.

An experimentation platform must encompass every step of the process, from designing and deploying experiments to analyzing them (Gupta et al. 2018). If you look at the components of an experiment platform from Bing

(Kohavi, Longbotham et al. 2009), LinkedIn (Xu et al. 2015), or Google (Tang et al. 2010), there are four high-level components:

- Experiment definition, setup, and management via a user interface (UI) or application programming interface (API) and stored in the experiment system configuration
- Experiment deployment, both server- and client-side, that covers variant assignment and parameterization
- Experiment instrumentation
- Experiment analysis, which includes definition and computation of metrics and statistical tests like p-values.

You can see how these components fit together in Figure 4.2. In this section, we dive into each of these components.

Experiment Definition, Set-up, and Management

To run many experiments, experimenters need a way to easily define, setup, and manage the experiment lifecycle. To define, or specify, an experiment, we need an owner, a name, a description, start and end dates, and several other fields (see Chapter 12). The platform also needs to allow experiments to have multiple *iterations* for the following reasons:

- To evolve the feature based on experiment results, which may also involve fixing bugs discovered during the experiment.
- To progressively roll out the experiment to a broader audience. This could either be via pre-defined rings (e.g., developers on the team, all employees within the company) or larger percentages of the outside population (Xia et al. 2019).

All iterations should be managed under the same experiment. In general, one iteration per experiment should be active at any time, although different platforms may need different iterations.

The platform needs some interface and/or tools to easily manage many experiments and their multiple iterations. Functionalities should include:

- Writing, editing, and saving draft experiment specifications.
- Comparing the draft iteration of an experiment with the current (running) iteration.
- Viewing the history or timeline of an experiment (even if it is no longer running).

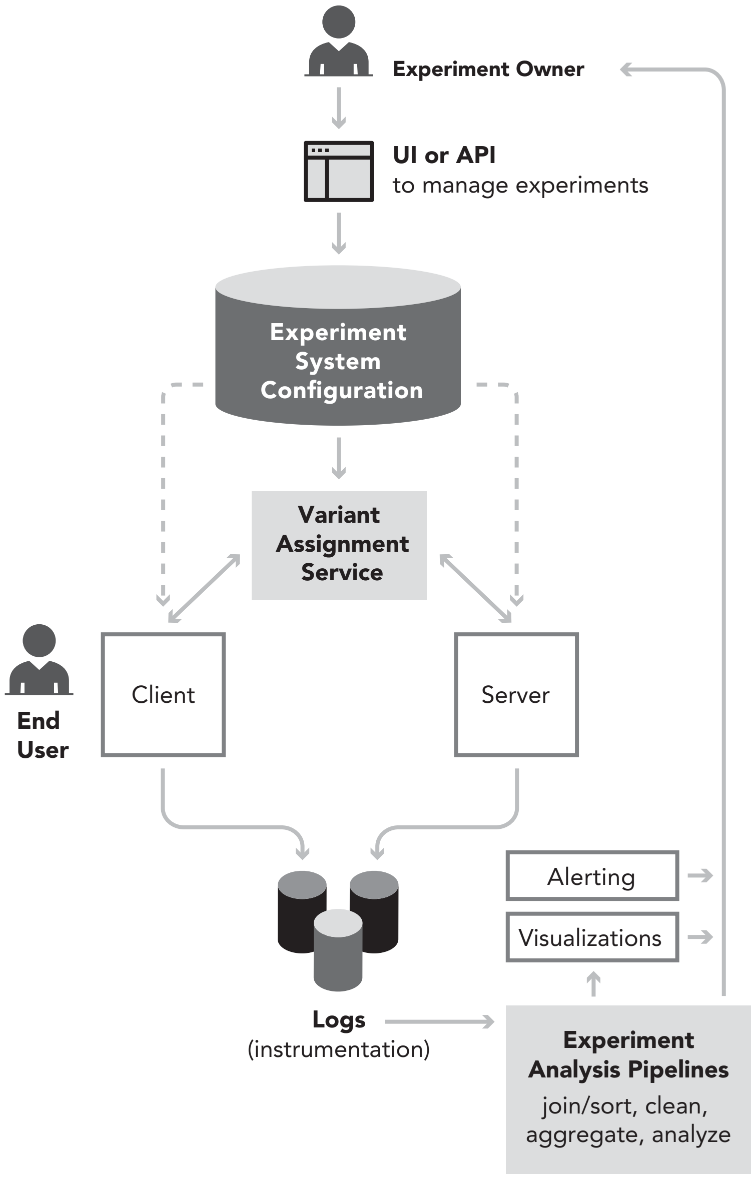
Experiment Owner
UI or API
to manage experiments
Experiment System Configuration
Variant Assignment Service
End User
Client
Server
Logs
(instrumentation)
Alerting
Visualizations
Experiment Analysis Pipelines
join/sort, clean, aggregate, analyze

Figure 4.2 Possible experiment platform architecture. The client and/or the server can call the Variant Assignment Service. The Variant Assignment Service may be a separate server, or a library embedded in the client and/or server (in which case the configurations would be pushed directly to the client and/or server). See the discussion later on in this chapter for a discussion of the different architecture options

- Automatically assigning generated experiment IDs, variants, and iterations and adding them to the experiment specification. These IDs are needed in the experiment instrumentation (discussed later in this chapter).
- Validating that there are no obvious errors in the specifications, such as configuration conflicts, invalid targeting audience, and so on.
- Checking the status of an experiment as well as starting/stopping an experiment. To guard against human error, usually only experiment owners or individuals with special permission can start an experiment. However, due to the asymmetry of harming users, anyone can stop an experiment, although alerts are generated to ensure that experiment owners are informed.

Also, since the experiment is impacting real users, additional tools or workflows are needed to check the experiment variants before they go live. Options range from test code that must be run before deployment or a permission control system where experiments must get approval from trusted experts.

Beyond these basic checks, especially in the Fly phase when experiments are being run at scale, the platform also needs to support:

- Automation of how experiments are released and ramped up (see Chapter 15 for more detail)
- Near-real-time monitoring and alerting, to catch bad experiments early
- Automated detection and shutdown of bad experiments.

These increase the safety of the experiments.

Experiment Deployment

After creating an experiment specification, the specification needs to be deployed to affect a user's experience. Deployment usually involves two components:

1. An experimentation infrastructure that provides experiment definitions, variant assignments, and other information
2. Production code changes that implement variant behavior according to the experiment assignment.

The experimentation infrastructure must provide:

- **Variant assignment**: Given a user request and its attributes (e.g., country, language, OS, platform), which experiment and variant combinations is that request assigned to? This assignment is based on the experiment specification and a pseudo-random hash of an ID, that is, f(ID). In most cases, to ensure the assignment is consistent for a user, a user ID is used. Variant assignment must also be independent, in that knowing the variant assignment of one user should not tell us anything about variant assignment for a different user. We discuss this in more depth in Chapter 14. In this chapter, we assume *user* is the randomization unit.

- **Production code, system parameters and values**: Now that you have variant assignment and definitions, how do you ensure that the user receives the appropriate experience: how do you manage different production code and which system parameters should change to what values?

This interface (or interfaces) is represented as the Variant Assignment Service in Figure 4.2, and can return either just the variant assignment or a full configuration with the parameter values for performance reasons. In either case, the variant assignment service does not need to be a distinct server. Instead, it can be incorporated directly into the client or server via a shared library. Regardless of interface, a single implementation is critical to prevent inadvertent divergence and bugs.

There are important subtleties to consider when implementing the infrastructure, especially when operating at scale. For example, is *atomicity* required, and if so, at what granularity? Atomicity means whether all servers simultaneously switch over to the next iteration of an experiment. One example of where atomicity is important is in a web service, where a single request can call hundreds of servers, and inconsistent assignment leads to an inconsistent user experience (e.g., imagine a search query that requires multiple servers, each handling a disjoint part of the search index; if the ranking algorithm has changed, the same algorithm must be used by all servers). To fix this example, the parent service can perform variant assignment and pass it down to the child services. There are also differences in experiment deployment between client-based and server-based experiments, discussed further in Chapter 12.

Another consideration is where in the flow variant assignment happens (i.e., when the variant assignment interface is called). As discussed in Kohavi, Longbottom et al. (2009), variant assignment can happen in several places: outside of production code entirely using traffic splitting (e.g., traffic front door), client side (e.g., mobile app), or server side. To be better informed while making this decision, consider these key questions:

- **At what point in the flow do you have all the required information to do variant assignment?** For example, if you only have a user request, you may have information, such as user ID, language, and device. To use additional information, such as the age of the account, the time of their last visit, or frequency of visits, you may need to do a look-up before you can use that criteria for variant assignment. This could push variant assignment to later in the flow.

- **Do you allow experiment assignment to happen only at one point in the flow or at multiple points?** If you are in the early stages of building your experiment platform (Walk or early Run phases), we recommend having only one point where experiment assignment happens to keep it simple. If you have multiple assignment points, you will need orthogonality guarantees (e.g., overlapping experiments, as discussed in *Concurrent Experiments* later in this chapter) to ensure that experiment assignment that happens earlier does not bias experiment assignment that happens later in the flow.

Now that you have assigned variants, it is time to ensure that the system provides the appropriate Treatment to the user. There are three main choices for architecture.

- The first architecture creates a code fork based on variant assignment:

```
variant = getVariant(userId)
If (variant == Treatment) then
  buttonColor = red
Else
  buttonColor = blue
```

- The second architecture moves to a parameterized system, where any possible change that you want to test in an experiment must be controlled by an experiment parameter. You can either choose to continue to use code forks:

```
variant = getVariant(userId)
If (variant == Treatment) then
  buttonColor = variant.getParam("buttonColor")
Else
  buttonColor = blue
```

  Or move to:

```
variant = getVariant(userId)
...
buttonColor = variant.getParam("buttonColor")
```

- The third architecture removes even the `getVariant()` call. Instead, early in the flow, variant assignment is done, and a configuration with the variant and all parameter values for that variant and for that user are passed down through the remaining flow.

```
buttonColor = config.getParam("buttonColor")
```

  Each system parameter has a default setting (e.g., the default `button-Color` is blue), and for Treatment, you only need to specify which system parameters change and their values. The `config` that is passed contains all parameters and the appropriate values.

There are advantages and disadvantages to each architecture. The main advantage of the first architecture is that variant assignment happens close to the actual code change, so handling triggering is easier. The Control and Treatment populations in the first architecture both contain only the affected users (see Chapter 20); however, the disadvantage can be escalating technical debt, as managing forked code paths can become quite challenging. The second architecture, especially with the second option, reduces the code debt while maintaining the advantage of handling triggering more easily. The third architecture moves variant assignment early, so handling triggering is more challenging. However, it can also be more performant: as a system grows to have hundreds to thousands of parameters, even if an experiment likely affects only a few parameters, then optimizing parameter handling, perhaps with caches, becomes critical from a performance perspective.

Google shifted from the first architecture to the third based on a combination of performance reasons as well as the technical debt and the challenges of reconciling code paths when it came time to merge back into a single path to make future changes easier. Bing also uses the third architecture. Microsoft Office uses the first option in the second architecture but implemented a system where a bug ID is passed as an experiment parameter, triggering an alert after three months to remind engineers to remove experimental code paths.

Regardless of which architecture you choose, you must measure the cost and impact of running experiments. An experiment platform can also have performance implications, so running some traffic outside of the experimentation platform is itself an experiment to measure the impact of the platform, be that in site speed latency, CPU utilization and machine cost, or any other factor.

Experiment Instrumentation

We assume you already log basic instrumentation, such as user actions and system performance (see Chapter 13 for what to instrument). Especially when testing new features, you must update your basic instrumentation to reflect these new features, as these updates to your instrumentation allow you to perform proper analysis. The focus during the Crawl phase is on this level of instrumentation, and leadership must ensure that instrumentation is constantly being reviewed and improved.

For experiments, you should also instrument every user request and interaction with which variant and iteration is run. The iteration is important especially when an experiment starts or ramps up, because not all servers or clients will simultaneously change the user Treatment (Chapter 12).

In many cases, especially as you get to the Run and Fly phases, we want to log the counterfactual, or what would have happened. For example, for a Treatment user, we may want to log what search results would have returned if they were the Control variant. In the system parameterized architecture described above, where variant assignment happens early, you may find counterfactual logging quite challenging but necessary (see Chapter 20). Counterfactual logging can be expensive from a performance perspective, in which case you may need to establish guidelines about when it is needed. If your product has a place for users to enter feedback, that feedback and the variant IDs must be logged. This is helpful when feedback is specific to the variant.

Scaling Experimentation: Digging into Variant Assignment

As companies move from the Walk to Run phase, to provide enough statistical power to experiments, a sufficient percentage of users must be assigned to each variant. Where maximal power is desired, an experiment will run at 50%/50% and include all users. To scale the number of experiments, users must be in multiple experiments. How does that work?

Single-Layer Method

Variant assignment is the process by which users are consistently assigned to an experiment variant. In the Walk phase, the number of experiments is usually small and it is common to divide all traffic with each experiment variant receiving a specified fraction of the total traffic. You might have one experiment with one Control and two Treatment variants taking up 60% of traffic, and another experiment with just one Control and one Treatment taking up the other 40% of traffic (Figure 4.3). This assignment is typically done using a

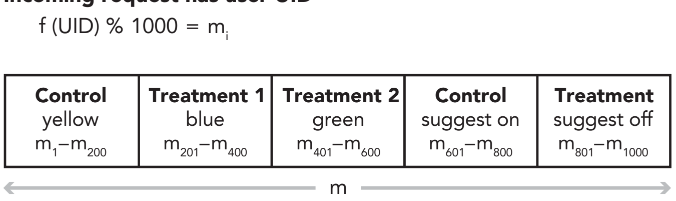
Incoming request has user UID
$f(UID) \% 1000 = m_i$
Control yellow $m_1 - m_{200}$
Treatment 1 blue $m_{201} - m_{400}$
Treatment 2 green $m_{401} - m_{600}$
Control suggest on $m_{601} - m_{800}$
Treatment suggest off $m_{801} - m_{1000}$
m

Figure 4.3 Example Control-Treatment assignment in the Single-Layer Method

hash function to consistently assign the users to buckets. In this example, we use 1,000 disjoint buckets and specify which variant gets which buckets. In this example, a variant with 200 buckets has a 20% traffic allocation.

The assignment of users to buckets must be random but deterministic. If you compare any two buckets running the same Treatment, they are assumed to be statistically similar (see Chapter 19):

- There should be roughly the same number of users in each bucket (see Chapter 3). If you broke it down by key dimensions, such as country, platform, or language, comparing slices across buckets will also be roughly the same.
- Your key metrics (goal, guardrail, quality), should have roughly the same values (within normal variability).

Monitoring assignments are key! Google, Microsoft, and many other companies found errors in the randomization code by monitoring bucket characteristics. Another common issue are carry-over effects (see Chapter 23), where prior experiments can taint buckets for the current experiment. Re-randomization, or shuffling, the buckets with every experiment so that they are no longer contiguous is a common solution (Kohavi, et al. 2012).

The Single-Layer (also called a numberline) method is simple and allows multiple experiments to run simultaneously (each user is only in a single experiment). It is a plausible choice in early maturity phases when few experiments run concurrently; however, the main drawback is the limitation on the number of concurrent experiments, as you must ensure that each experiment has enough traffic for adequate power. Operationally, managing experimental traffic in a Single-Layer system can be challenging, as even in this early phase, experiments are running concurrently—just not on a single user. To manage the concurrency, LinkedIn, Bing, and Google all started with manual methods (at LinkedIn, teams would negotiate traffic "ranges" using e-mails; at Bing, it was managed by a program manager, whose office was usually packed with people begging for experimental traffic; while at Google, it started with e-mail and instant messaging negotiation, before moving to a program manager). However, the manual methods do not scale, so all three companies shifted to programmatic assignment over time.

Concurrent Experiments

To scale experimentation beyond what is possible in a Single-Layer method, you need to move to some sort of concurrent (also called overlapping) experiment system, where each user can be in multiple experiments at the same time. One way to achieve this is to have multiple experiment layers where each layer

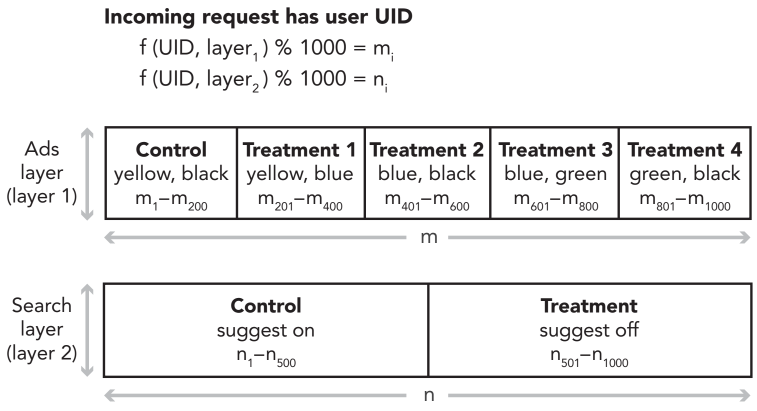
Incoming request has user UID
f (UID, layer$_1$) % 1000 = m$_i$
f (UID, layer$_2$) % 1000 = n$_i$
Ads layer (layer 1)
Control yellow, black m$_1$–m$_{200}$
Treatment 1 yellow, blue m$_{201}$–m$_{400}$
Treatment 2 blue, black m$_{401}$–m$_{600}$
Treatment 3 blue, green m$_{601}$–m$_{800}$
Treatment 4 green, black m$_{801}$–m$_{1000}$
m
Search layer (layer 2)
Control suggest on n$_1$–n$_{500}$
Treatment suggest off n$_{501}$–n$_{1000}$
n

Figure 4.4 Control-Treatment assignment example in an overlapping methodology

behaves like the Single-Layer method. To ensure orthogonality of experiments across layers, in the assignment of users to buckets, add the layer ID. This is also where you would add, as in the experiment specification discussed above, the layer ID (or some other way of specifying constraints).

When a request comes in, variant assignment is done once for each layer (see Figure 4.4 for an example with two layers). This implies that both production code and instrumentation must handle a vector of variant IDs. The main question with a concurrent experiment system is how to determine the layers, and there are several options.

One possibility is to extend a full factorial experiment design into a full factorial platform design. In a full factorial experiment design, every possible combination of factors is tested as a variant. If we extend that to a platform, then a user is in all experiments simultaneously: the user is assigned to a variant (Control or any of the Treatments) for every experiment running. Each experiment is associated with a unique layer ID, so all experiments are orthogonal to each other. Iterations of the same experiment usually share the same hash ID to ensure a consistent experience for a user. This simple parallel experimentation structure allows you to scale the number of experiments easily in a decentralized manner.

The main drawback of this platform design is that it does not avoid potential collisions, where certain Treatments from two different experiments give users a poor experience if they coexist. For example, we could be testing blue text in Experiment One and blue background in Experiment Two. It would have been a horrible experience for any users who happen to fall into both Treatments.

In statistical terms, these two experiments "interact" with each other. It is not only a poor user experience, the results measured for each experiment independently may also be incorrect without considering any interactions between the two experiments. Note that not all interactions are antagonistic—sometimes being in both Treatments helps more than the sum.

That said, a factorial platform design might be preferred if the reduction on statistical power when splitting up traffic outweighs the potential concern of interaction. Moreover, if we set up these experiments independently, we can analyze to see which experiments interact what their effects would be without interaction. Of course, if there is no significant interaction, each experiment can be analyzed separately, and each gets to enjoy the full amount of traffic available for maximum power. Microsoft's experimentation platform has a robust system that automates the detection of interactions (Kohavi et al. 2013).

To prevent poor user experiences, we can either use a *nested* platform design (Tang et al. 2010) or a *constraints-based* platform design (Kohavi et al. 2013). For scalability, Google, LinkedIn, Microsoft, and Facebook use some variation of these designs (Xu 2015, Bakshy, Eckles and Bernstein 2014).

In a nested design, system parameters are partitioned into layers so that experiments that in combination may produce a poor user experience must be in the same layer and be prevented by design from running for the same user. For example, there might be one layer for the common UI elements (e.g., the header of the page and all information in the header), another layer for the body, a third layer for back-end systems, a fourth layer for ranking parameters, and so on.

A constraints-based design has experimenters specify the constraints and the system uses a graph-coloring algorithm to ensure that no two experiments that share a concern are exposed to the user. Automated systems for detecting interactions (Kohavi et al. 2013) can be a useful extension.

Experimentation Analytics

To move to the later phases of experimentation maturity, we also need automated analysis, which is crucial both for saving teams from needing to do time-consuming ad hoc analysis, and ensuring that the methodology behind the reports is solid, consistent, and scientifically founded. We assume that the work of choosing the goal, guardrail, and quality metrics is already done, as well as any codification of tradeoffs into an OEC.

Automating analysis first requires *data processing*, where the goal is to get data into a usable state to compute and visualize the experiment results. Since instrumentation about a user request may happen in multiple systems, data processing typically involves sorting and joining the different logs and cleansing, sessionizing and enriching them. This process is sometimes referred to as *cooking* the data.

When you have the processed data, the goal is to summarize and highlight the key metrics to help guide decision makers to a launch/no-launch decision. This requires *data computation* of metrics (e.g. OEC, guardrail metrics, quality metrics) by segments (e.g., country, language, device/platform), computations of p-values/confidence intervals, also trustworthiness checks, such as the SRM check. It can also include analysis to automatically find which segments are most interesting (see Chapter 3). Note that while the data computation may compute all of these in a single step, when you actually look at the experiment data, you must look at the trustworthiness checks first, before checking the OEC, doing any segmentation, and so on. However, before you look at the experiment data, all of the data cooking and computation must also be thoroughly tested and checked to ensure the trustworthiness of these processes.

Given the computation, we can finally create the *data visualization* to highlight key metrics and interesting metrics and segments in an easy-to-understand way. This visualization can be as simple as an Excel-like spreadsheet. Metrics are presented as a relative change, with clear indication if results are statistically significant, often using color-coding to make significant changes stand out. To foster a culture of intellectual integrity, ensure that results use common definitions that are tracked and accessible, as well as frequently reviewed, agreed upon and updated.

As an organization moves into the Run and Fly phase, there can be many metrics –even thousands! This is when you group metrics by tier (company-wide, product-specific, feature-specific (Xu et al. 2015, Dmitriev and Wu 2016)) or by function (OEC, goal, guardrail, quality, debug; see Chapter 7). Multiple testing becomes more important as the number of metrics grow, and we found that one common question arose from experimenters: Why did this metric move significantly when it seems irrelevant?

While education can help, options in the tool to use p-value thresholds smaller than the standard 0.05 value are effective. Lower thresholds allow experimenters to quickly filter to the most significant metrics (Xu et al. 2015).

Use visualization tools to generate per-metric views of all experiment results, which allows stakeholders to closely monitor the global health of key metrics and see which experiments are most impactful. This transparency encourages conversations between experiment owners and metric owners, which in turn increases the overall knowledge of experimentation in your company.

Visualization tools are a great gateway for accessing *institutional memory* to capture what was experimented, why the decision was made, and successes and failures that lead to knowledge discovery and learning. For example, through mining historical experiments, you can run a meta-analysis on which kind of experiments tend to move certain metrics, and which metrics tend to move together (beyond their natural correlation). We discuss this more in Chapter 8. When new employees join the company, visuals help them quickly form intuition, get a feel for corporate goals, and learn your hypothesis process. As your ecosystem evolves, having historical results and refined parameters allows you to rerun experiments that failed.