eppo

Building A Modern Experimentation Stack

Every ambitious AB testing system has the same architecture, with seven core components. For an experimentation program to generate ROI, they all have to work together. Let's walk through each of these components, and the opportunities and failure modes they present us with.

In a previous era, data teams that wanted to run experiments had to rely on an open-source distributed compute platform like Spark or Hive. This forced teams to build bespoke solutions, since these systems didn't have a canonical method of implementation.

Nowadays, organizations can build upon a Modern Data Stack, which includes a data warehouse like Snowflake, Databricks, Redshift, or BigQuery, and a transformation layer like dbt or Airflow.

These new tools enable companies to run experiments early and often.

What is an experiment?

An experiment is a tool for understanding cause and effect. By controlling inputs to a set of experiment subjects and measuring outputs, we gain knowledge about the input/output relationship. We quantify that knowledge using statistics.

Inputs

The inputs to experiments can include things like:

- A new algorithm for recommendations, search, or matching
- A new product feature (or the choice to remove a feature)
- Marketing copy, design alternatives
- Anything you want to test!

Subjects

The subjects of experiments are generally users who have a User ID and/or a Session ID. Some of these users see version A, and others see version B, randomly, and we compare the two groups.

Outputs

The outputs of experiments will include things like:

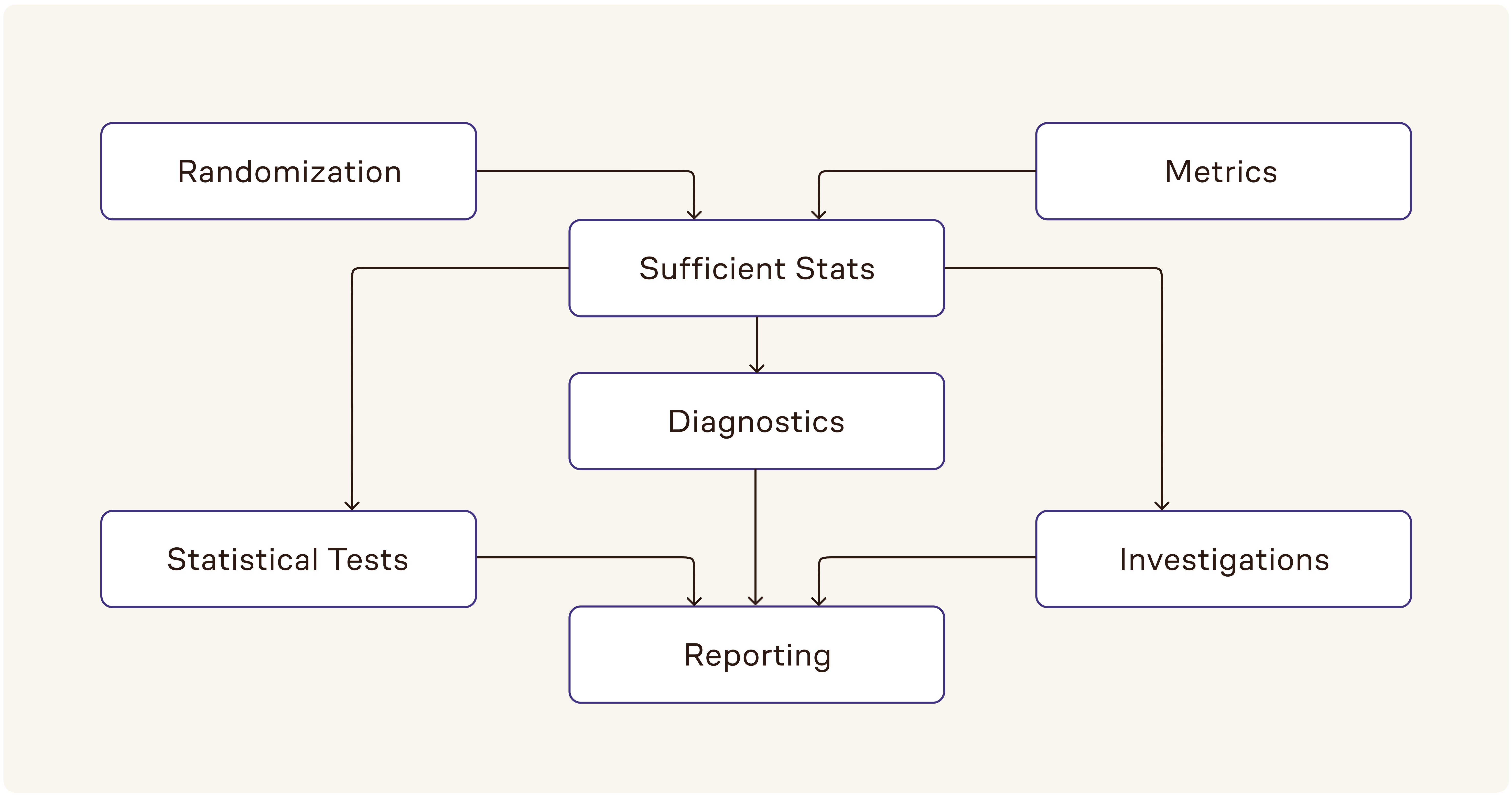
- Conversions (binary outcome)
- Revenue (continuous outcome)
- Any measurable subject behavior

There's extensive evidence that we are all bad at predicting whether new features improve or degrade performance; in fact only about 1/3rd of features lead to positive outcomes. Experimentation is the best tool to efficiently figure out what works, and what does not.

Companies that have realized this, like Airbnb, Microsoft, Meta, and Netflix, built big in-house experimentation platforms, using their large-scale engineering resources. And every one of these systems has the same architecture.

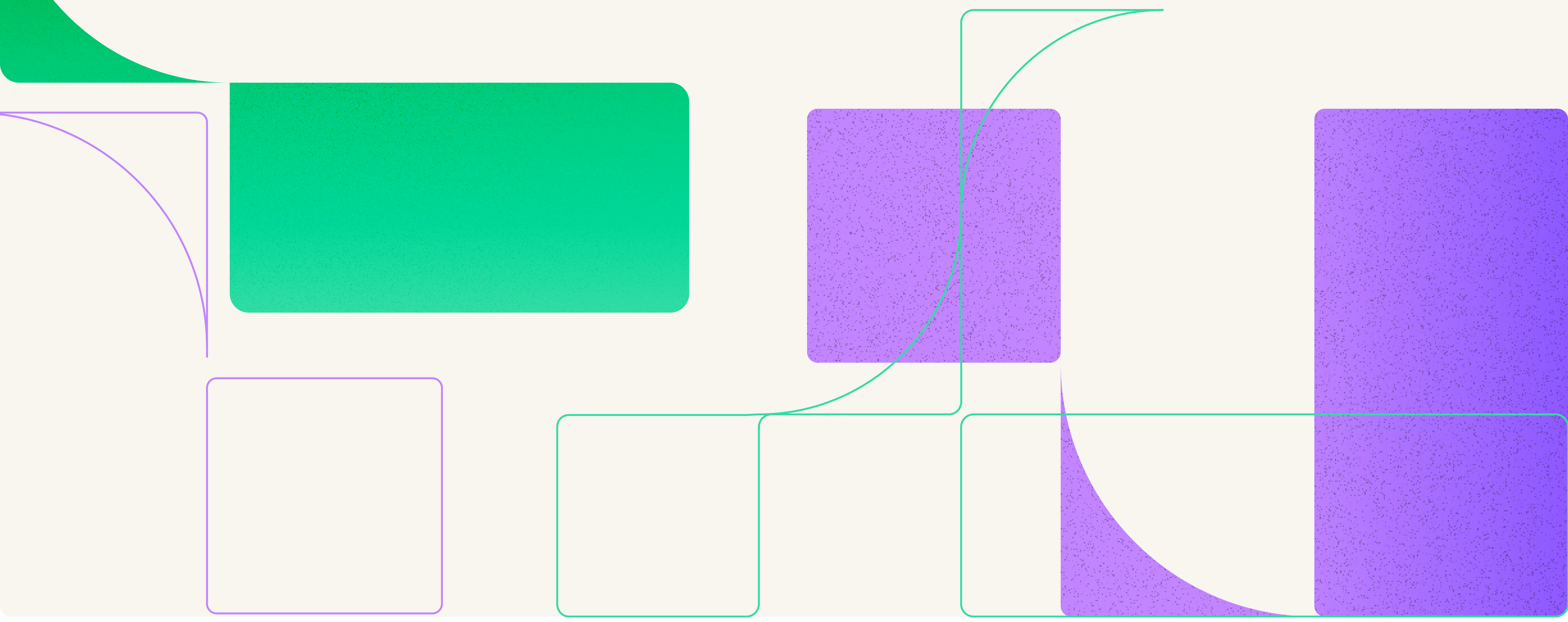
Each of these components has depth and complexity, but for experimentation to generate meaningful ROI, they all have to work together. When even a single component isn't working, it can really set your experimentation practice back.

Let's walk through each of these components, and the opportunities and failure modes they present us with.



Every experimentation system has the same architecture

Randomization

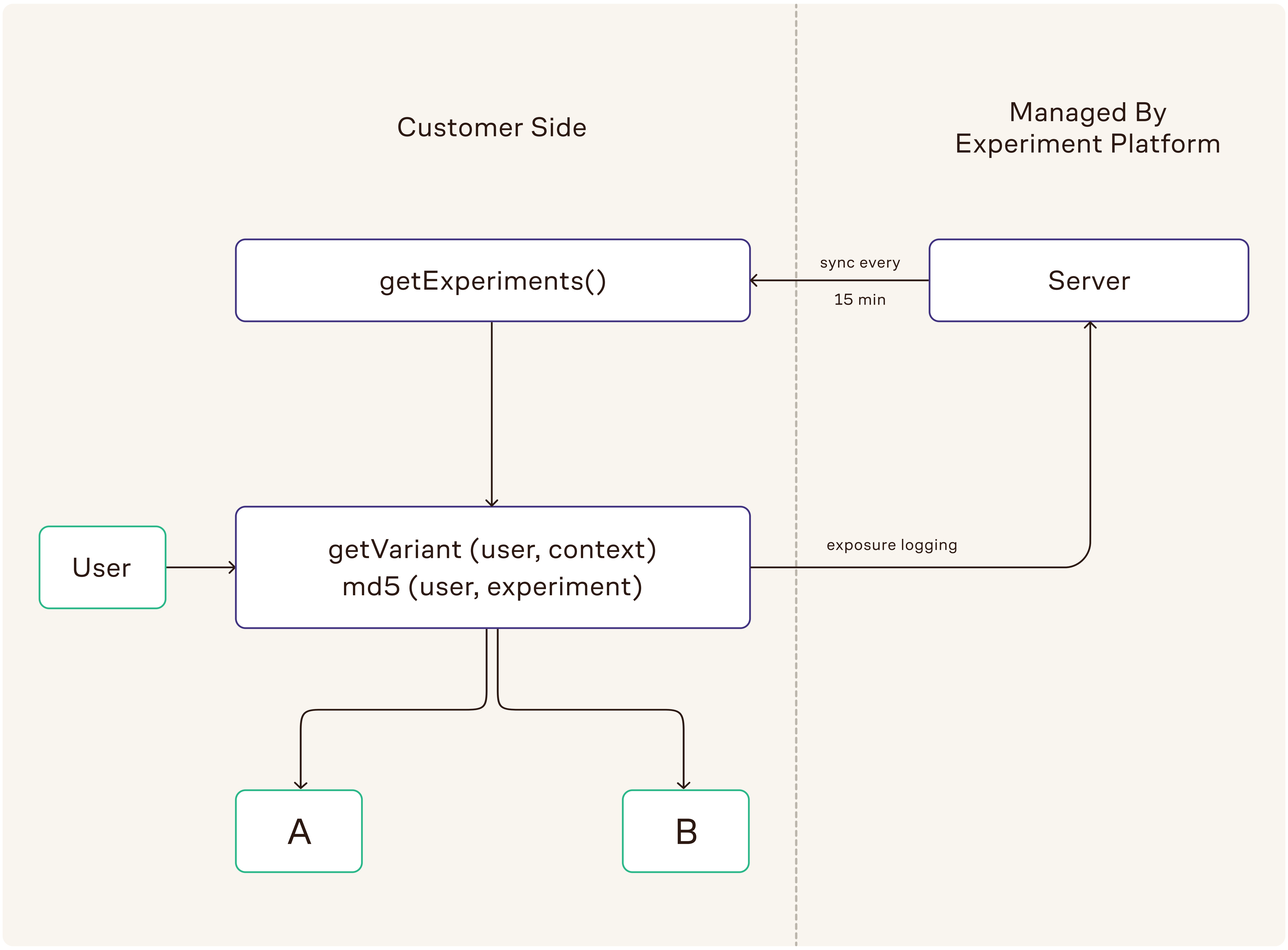


To most people, randomization is probably the most familiar component of an experimentation workflow. It's a process that selects users randomly and independently from any attributes they possess.

There are **three problems** to solve here:

1. How do you serve experiments quickly?
2. How do you randomize properly?
3. How do you instrument what you did?

Under the hood, all commercial feature-flag tooling looks essentially the same. They all have a way to convey experiment configurations that are synced onto clients every 15 minutes or so.



Pro Tip: Use md5() hashes for assignment

With experiment metadata stored locally, it becomes a quick, low-latency process to split users into different groups. Typically, they have a function called "getVariant," which sends an event back to your warehouse, identifying that one specific user was assigned to one specific experiment.

For randomization, the best practice is to use md5() hashes. Hashing is important, because whether you conduct your randomization offline or online, you can precisely replicate the process. It's idempotent — If the user hits the experiment 10 times, they'll end up in the same treatment group each time.

Metrics

In experimentation, a metric is a measurement of subject-level activity that occurs after assignment. Metrics are created by aggregating (e.g. summing, counting, etc.) events (facts).

It's fair to estimate that 80% of a data team's work is to define and serve high-quality metrics to different places, but this is something that a lot of experimentation programs get wrong.

The first thing to emphasize is that in experimentation, you want to define metrics by their underlying facts. A fact table enables you to model atomic actions that happen at a specific moment in time. For experimentation, you need a timestamp to know if a purchase or upgrade happened before or after the user was in the experiment.

Once you turn to the fact table, the best way to make an experimentation program more powerful is to use business metrics. Too many companies just use the metrics that are available via existing tooling, which tends toward shallow metrics like click-conversions. You can boost those metrics, but what you really want to show is that you boosted revenue. Ask yourself the question: If you show your experiment result to your CFO, will they care?

The biggest gap between Airbnb/Netflix/et al. and commercial tools is how easily you can use **business metrics.**

Business Metrics

- Revenue, Activation, Purchases
- What the CFO reads
- From databases, Stripe, multiple POS

vs.

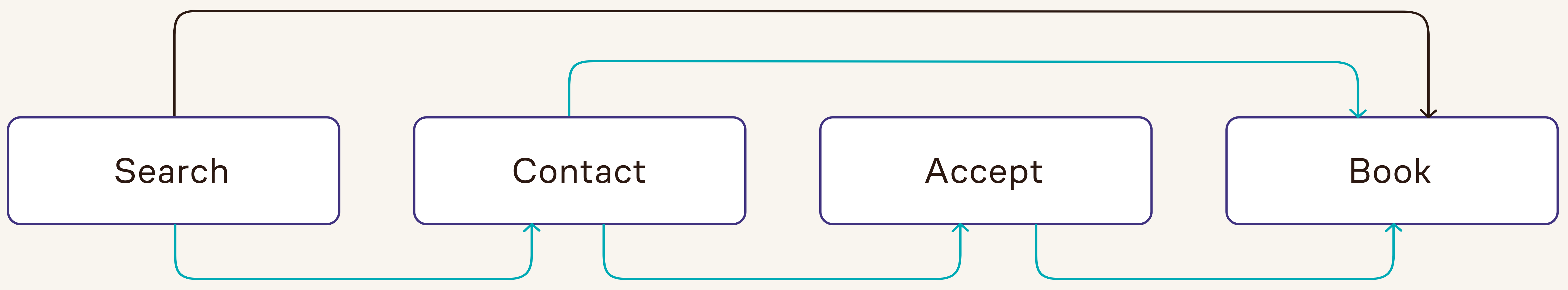
Shallow Metrics

- Signups, “conversions”
- “Directionally accurate”
- From event streams

To get real value from experimentation, you want to use financial data, such as from Stripe, and you want to be connected to the data warehouse.

Because teams are so used to shallow click metrics, they usually focus on steps higher up in the funnel, like signups or form submissions.

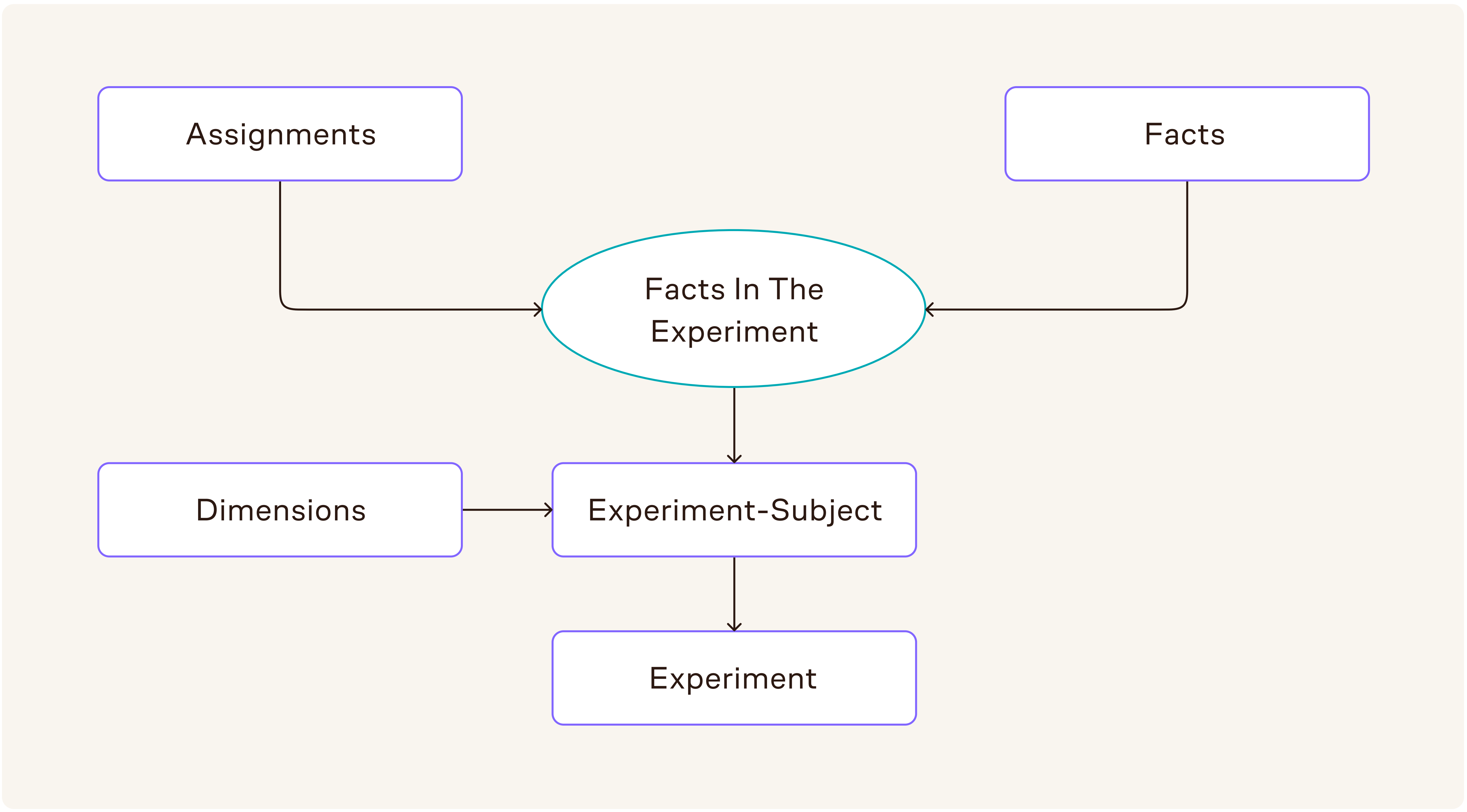
Metric	Δ	p-value
Search to Book	-0.31%	0.37
Search to Contact	-1.29%	0.04
Contact to Book	0.99%	0.06
Contact to Accept	1.58%	0.00
Accept to Book	0.58%	0.11



Most enterprise experimentation teams find that the vast majority of experiments will boost one part of the funnel, and then depress another, leaving your results completely neutral. If you use business metrics, you’ll have more confidence that you’re not just playing whack-a-mole.

Sufficient Statistics

This component is called sufficient statistics because that's what you're calculating to fuel your inference, but it's actually a significant data engineering problem.



Experiment Data Pipelines: One big JOIN and some GROUP-BYs

This is the DAG that most pipelines end up working with for experimentation. On one end, you have the assignment events from the randomization system. On the other end, you have facts, such as a purchase. And then you have to join them together, to determine: of these purchases, how many ended up in this experiment? You group it first at an experiment-subject level, and then at an experiment level.

In this DAG, all the action is primarily in that first join. It’s basically a big join and then two group-bys. The group-bys happen very quickly, and most data engineering teams don’t use much compute on them. But that experiment join is so large that it can make up a third of all your compute costs, and it's where most of the optimizations will happen.

Statistical Tests

While there are plenty of conversations around the relative benefits of Frequentist vs. Bayesian statistics, 95% of experimentation programs just run a t-test. While this isn't necessarily wrong, it puts a lot of the onus for expertise on your organization, because there's a bunch of failure modes for t-tests.

Simple statistical tests stress the organization

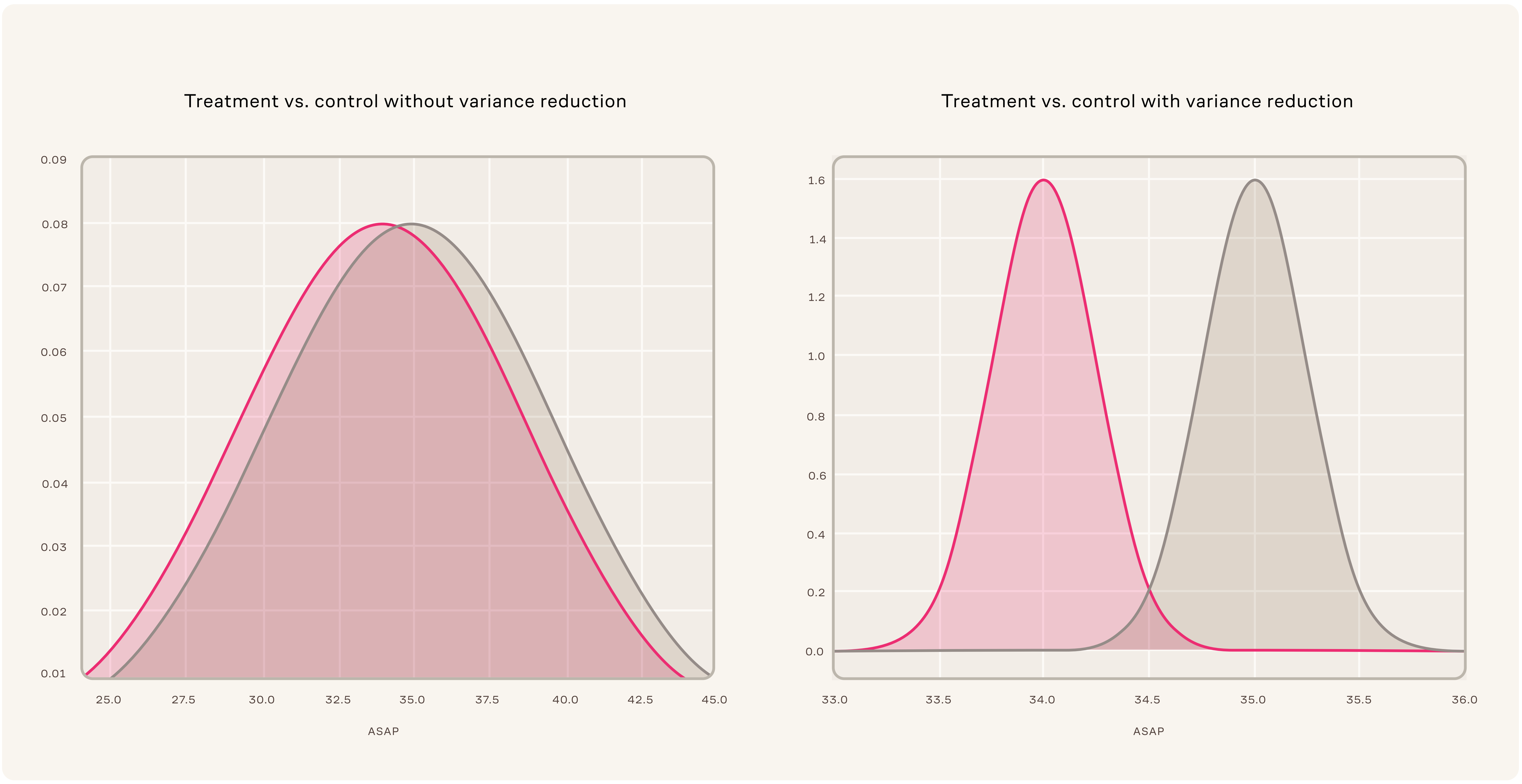
When using t-tests, you need to:

- Not look at the results until it's done
- Not test multiple variants without a statistical correction
- Not have outliers/power laws

Specifically, you're not supposed to look at the results until they're done. You make a promise saying that you're going to run an experiment, get 1000 samples in both groups, and then look at the results and make a decision. Anyone who runs an experimentation program knows that this never works in practice — peeking is the norm.

There are a bunch of other failure modes. For example, ratio metrics require adjusting for the correlation between the numerator and denominator via the delta method to obtain valid statistical results.

Because most teams don't realize these points of failure, they end up incurring a bunch of false positives. So simple statistical tests end up stressing the organization.



Advanced statistics can speed things up

But there's a few things that can help avoid statistical problems. To get around the peeking problem, you can use sequential methodology; these methods assume that results of an experiment are monitored during the entire runtime of an experiment. They adjust results in a way that false positives are kept low.

Another way to help your organization is through variance reduction, and a technique like CUPED. By leveraging historical data, you can make your experiments run faster. You can save weeks of product time and increase your learning rate, and there's really no downside.

CUPED controls for variance that we can predict

Goal:

Improve # of happy meals purchased

Insight:

We can predict whether some will purchase a happy meal

- Are they a family?
- Have they purchased a happy meal recently?

Suppose you're McDonald's, and you're running an experiment to boost the number of Happy Meals sold. The easy way to do it is to run some experiment, then see if the purchase number goes up. But suppose every time someone walks in the door, you make a guess as to whether they'll buy a Happy Meal, and then you compare the actual result against your guesses. In that paradigm, if someone walked in with 3 children, you would guess that they'd get a Happy Meal. Suddenly, you are accounting for some natural variation, removing it from the equation, and getting a lot more signal.

CUPED has become a mainstream method at companies like Airbnb, Microsoft, and Meta, but because it involves a little more complexity from a data engineering and statistics standpoint, you don't yet see it across the commercial landscape.

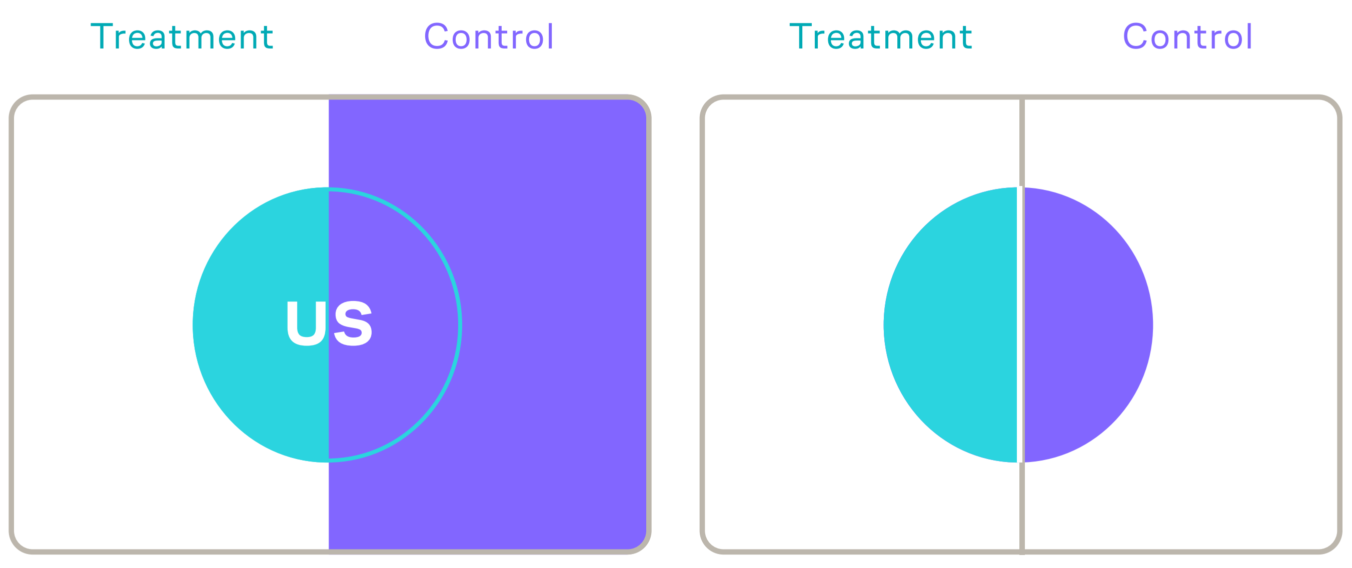
Diagnostics

The underlying principle of experimentation is trust. The whole process is a leap of faith, where you let a system separate the winners from the losers, so trust is paramount.

The most common failure mode for experimentation is imbalanced treatment groups. What you think is a 50/50 split may actually be closer to 52/48, and that difference almost always signals an underlying issue. The types of users throwing off this 50/50 split might have low latency, or might be on a broken client. The way to check for this is via the sample ratio mismatch test, which can diagnose the underlying problem. All mainstream experimentation tools build this check into their day-to-day practice.

Problem:

Imbalanced treatment groups



Biased implementation

Correct, unbiased implementation

Causes:

These issues are usually due to

- Latency of experiment delivery
- Bad implementation

Solution:

Sample ratio mismatch test (SRM)

$$x^2 = \sum \frac{(\text{observed} - \text{expected})^2}{\text{expected}}$$

Make sure you have balanced groups

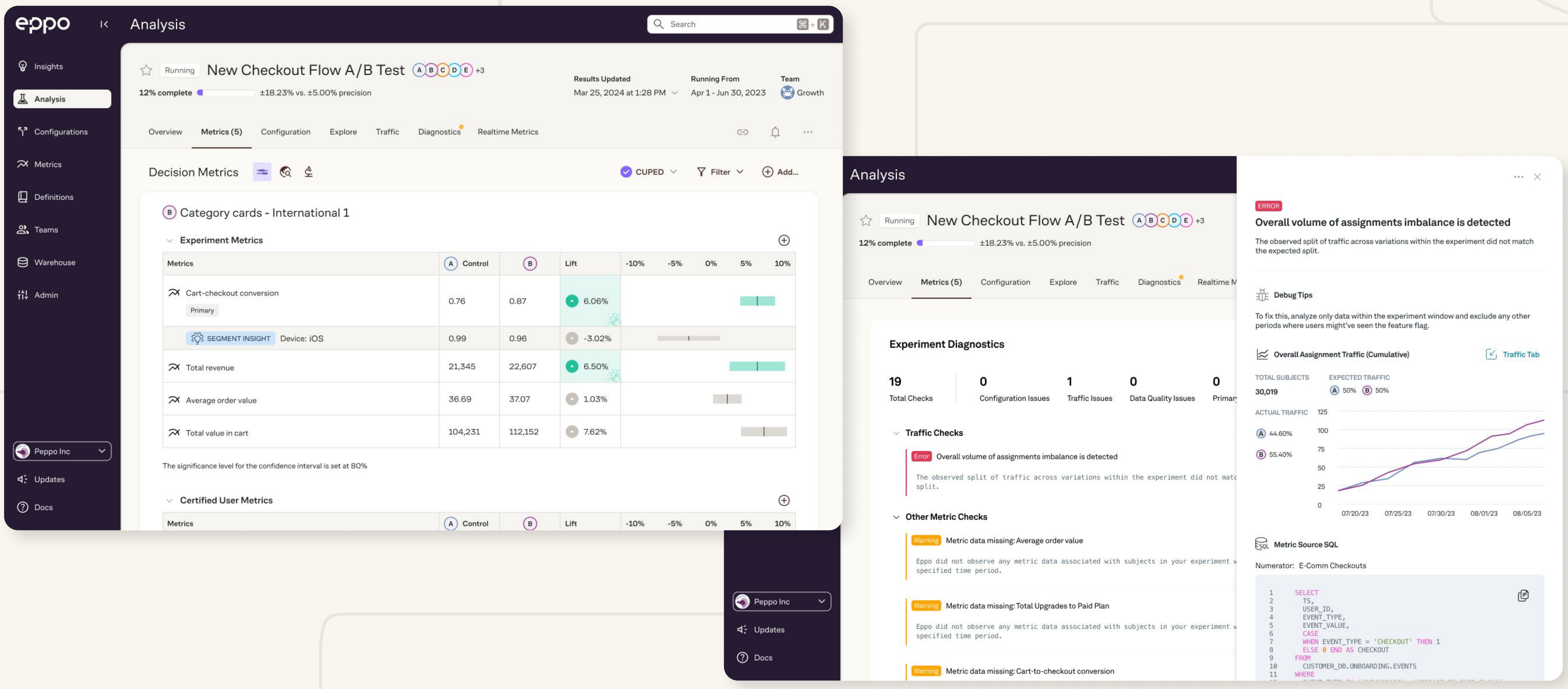
The next diagnostic is outliers. As 'The Black Swan' author Nassim Nicholas Taleb emphasizes, social phenomena metrics follow power laws, and not the finite moment distributions that all of statistics is built upon. There will always be outliers that skew your results, and a small number of observations can wreck the analysis.

To handle outliers, one can use non-parametric tests that make no assumptions about distributions. But in commercial practice, it is more common to see **two things**:

1. Winsorizations, which take the 99th percentile as the maximum for the metric.
2. And CUPED, which accounts for prior history.

Another challenge with experimentation is that it makes your underlying data quality issues invisible.

In product analytics, it's typically clear that something is broken. If the revenue chart drops to zero, someone will ring alarm bells. However, this is often harder to spot in experiment analyses, as the control and treatment groups are often equally broken, but that does not show up in the difference. So you need a way to monitor data quality before running the experiment.



Running New Checkout Flow A/B Test A B C D E +3

12% complete ±18.23% vs. ±5.00% precision

Results Updated: Mar 25, 2024 at 1:28 PM
Running From: Apr 1 - Jun 30, 2023
Team: Growth

B Category cards - International 1

Experiment Metrics

Metrics	A Control	B	Lift
Cart-checkout conversion Primary	0.76	0.87	6.06%
SEGMENT INSIGHT Device: iOS	0.99	0.96	-3.02%
Total revenue	21,345	22,607	6.50%
Average order value	36.69	37.07	1.03%
Total value in cart	104,231	112,152	7.62%

The significance level for the confidence interval is set at 80%

Certified User Metrics

Experiment Diagnostics

19	0	1	0	0
Total Checks	Configuration Issues	Traffic Issues	Data Quality Issues	Primar

Traffic Checks

Error Overall volume of assignments imbalance is detected

The observed split of traffic across variations within the experiment did not matc split.

Other Metric Checks

Warning Metric data missing: Average order value

Eppo did not observe any metric data associated with subjects in your experiment w specified time period.

Warning Metric data missing: Total Upgrades to Paid Plan

Eppo did not observe any metric data associated with subjects in your experiment w specified time period.

Warning Metric data missing: Cart-to-checkout conversion

ERROR

Overall volume of assignments imbalance is detected

The observed split of traffic across variations within the experiment did not match the expected split.

Debug Tips

To fix this, analyze only data within the experiment window and exclude any other periods where users might've seen the feature flag.

Overall Assignment Traffic (Cumulative)

TOTAL SUBJECTS 30,019
EXPECTED TRAFFIC A 50% B 50%
ACTUAL TRAFFIC A 44.60% B 55.40%

Metric Source SQL

Numerator: E-Comm Checkouts

```
SELECT
  TS,
  USER_ID,
  EVENT_TYPE,
  EVENT_VALUE,
  CASE
  WHEN EVENT_TYPE = 'CHECKOUT' THEN 1
  ELSE 0 END AS CHECKOUT
FROM
  CUSTOMER_DB.ONBOARDING.EVENTS
WHERE
```

Monitor your data quality

Investigations

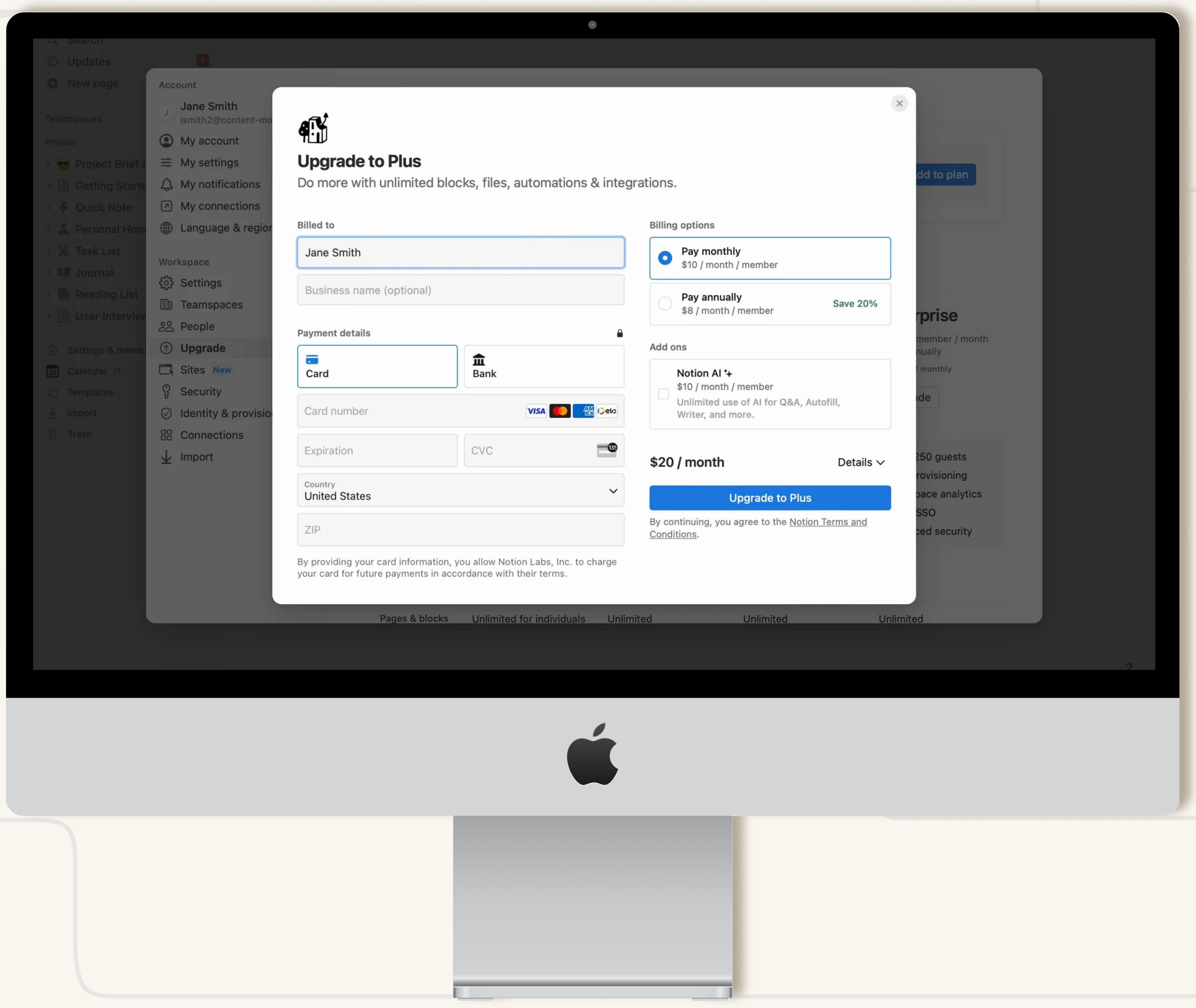
Every experimentation practice discovers that delivering wins is a learning process.

Investigations help you learn

**“First you must learn to test.
Then you learn to learn.
Then you learn to win.”**

Elena Verna
Head of Growth & Data @ Dropbox

First you have to be able to run an experiment. And then you have to learn to diagnose problems quickly. And then, you can get to a point where you reliably deliver wins.



Upgrade to Plus

Do more with unlimited blocks, files, automations & integrations.

Billed to: Jane Smith

Business name (optional)

Payment details: [✓ Card] Bank

Card number

Expiration | CVC

Country: United States

ZIP

By providing your card information, you allow Notion Labs, Inc. to charge your card for future payments in accordance with their terms.

Billing options:
[✓ Pay monthly — $10 / month / member]
Pay annually — $8 / month / member — Save 20%

Add ons:
Notion AI — $10 / month / member — Unlimited use of AI for Q&A, Autofill, Writer, and more.

$20 / month — Details

By continuing, you agree to the Notion Terms and Conditions.

Browser	Δ	p-value
All	-0.27%	0.29
Chrome	2.07%	0.01
Firefox	2.81%	0.00
IE	-3.66%	0.00
Safari	0.86%	0.26
Rest	-0.74%	0.33

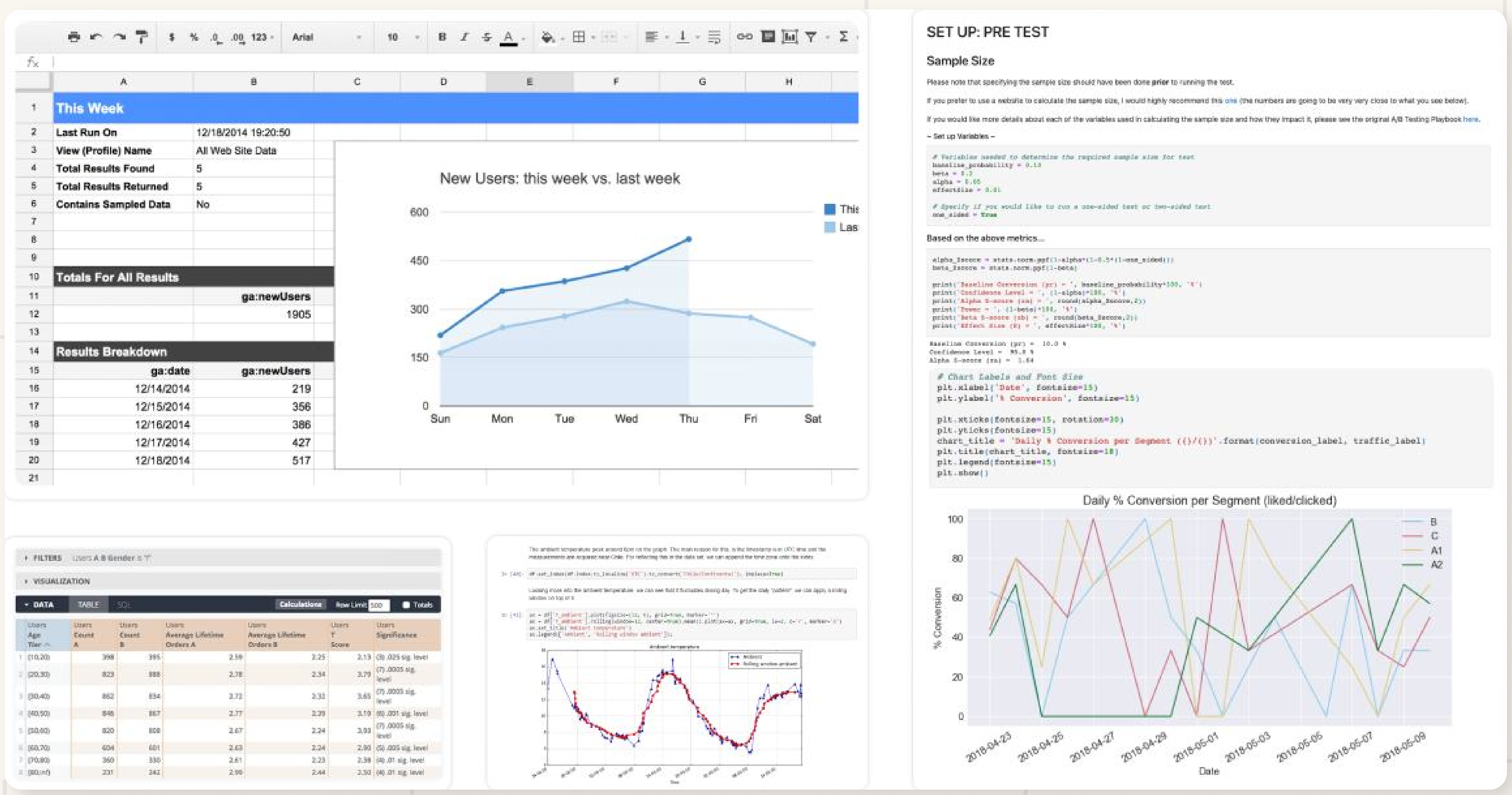
A specific segment of your product's users might particularly dislike your experiment

The most important way to learn is by segmenting your results. In one common scenario, a team sees a result that looks negative, but once they segment it by browser, they discover that it's just a browser-specific bug, and the overall results are positive.

You need to be able to split your results by user segment, and make the process democratized. That's the only way to reliably understand your experiment results.

Reporting

Reporting is the most underrated part of data science. You do all this work with such complexity, but in the end, your goal is to drive a decision in your organization. The last mile of data science is communication.



	A	B
1	This Week	
2	Last Run On	12/18/2014 19:20:50
3	View (Profile) Name	All Web Site Data
4	Total Results Found	5
5	Total Results Returned	5
6	Contains Sampled Data	No
10	Totals For All Results	
11		ga:newUsers
12		1905
14	Results Breakdown	
15	ga:date	ga:newUsers
16	12/14/2014	219
17	12/15/2014	356
18	12/16/2014	386
19	12/17/2014	427
20	12/18/2014	517

FILTERS Users A B Gender is 'F'

VISUALIZATION

DATA TABLE SQL Calculations Row Limit 500 Totals

	Users Age Tier	Users Count A	Users Count B	Users Average Lifetime Orders A	Users Average Lifetime Orders B	Users T Score	Users Significance
1	[10,20)	398	395	2.59	2.25	3.13	(3) .025 sig. level
2	[20,30)	823	888	2.78	2.34	3.79	(7) .0005 sig. level
3	[30,40)	862	834	2.72	2.32	3.65	(7) .0005 sig. level
4	[40,50)	846	867	2.77	2.39	3.19	(6) .001 sig. level
5	[50,60)	820	808	2.67	2.24	3.93	(7) .0005 sig. level
6	[60,70)	604	601	2.63	2.24	2.90	(5) .005 sig. level
7	[70,80)	360	330	2.61	2.23	2.38	(4) .01 sig. level
8	[80,inf)	231	242	2.99	2.44	2.50	(4) .01 sig. level

SET UP: PRE TEST

Sample Size

Please note that specifying the sample size should have been done **prior** to running the test.

If you prefer to use a website to calculate the sample size, I would highly recommend this one (the numbers are going to be very very close to what you see below).

If you would like more details about each of the variables used in calculating the sample size and how they impact it, please see the original A/B Testing Playbook here.

– Set up Variables –

```
# Variables needed to determine the required sample size for test
baseline_probability = 0.10
beta = 0.2
alpha = 0.05
effectSize = 0.01

# Specify if you would like to run a one-sided test or two-sided test
one_sided = True
```

Based on the above metrics...

```
alpha_Zscore = stats.norm.ppf(1-alpha*(1-0.5*(1-one_sided)))
beta_Zscore = stats.norm.ppf(1-beta)

print('Baseline Conversion (pr) = ', baseline_probability*100, '%')
print('Confidence Level = ', (1-alpha)*100, '%')
print('Alpha Z-score (za) = ', round(alpha_Zscore,2))
print('Power = ', (1-beta)*100, '%')
print('Beta Z-score (zb) = ', round(beta_Zscore,2))
print('Effect Size (E) = ', effectSize*100, '%')
```

```
Baseline Conversion (pr) =  10.0 %
Confidence Level =  95.0 %
Alpha Z-score (za) =  1.64
```

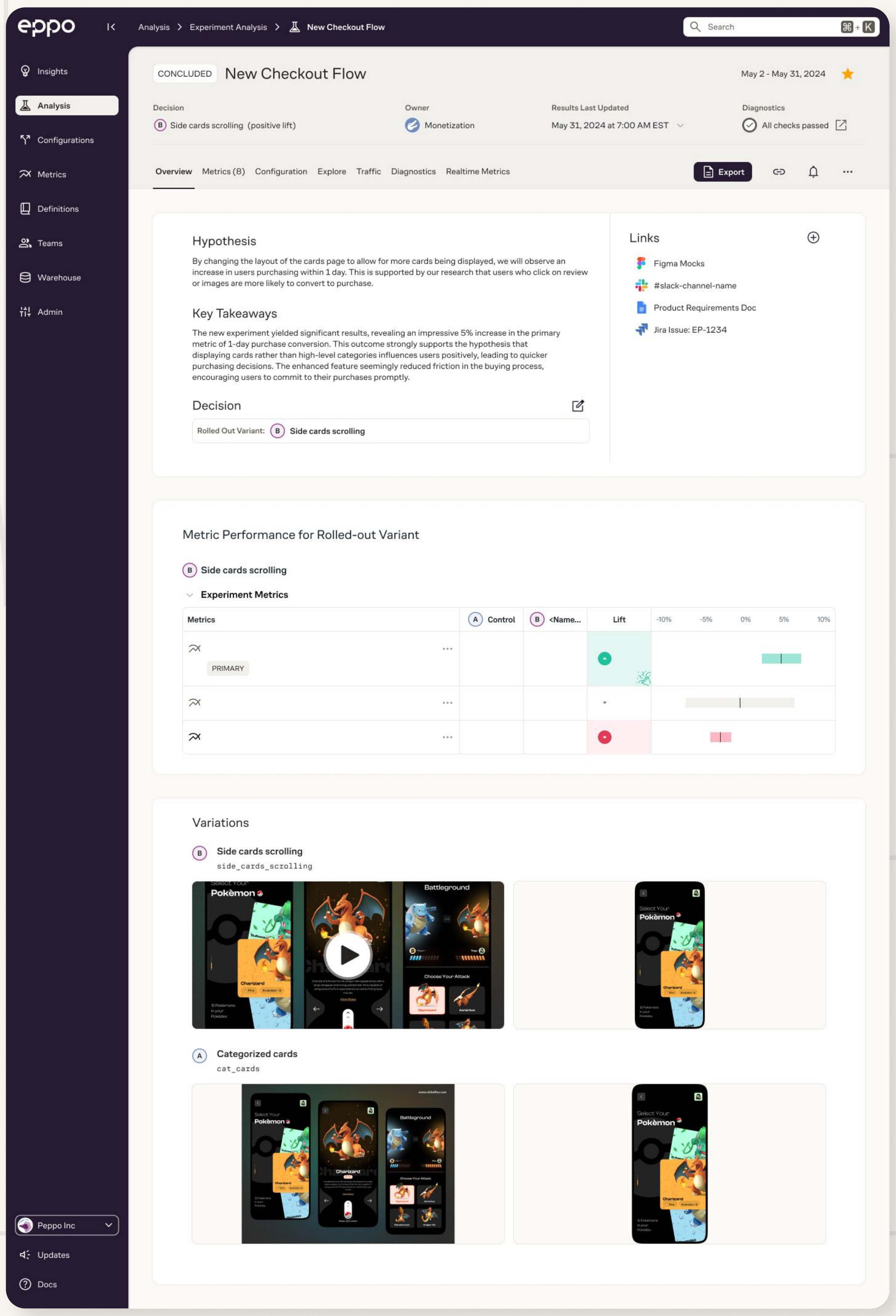
```
# Chart Labels and Font Size
plt.xlabel('Date', fontsize=15)
plt.ylabel('% Conversion', fontsize=15)

plt.xticks(fontsize=15, rotation=30)
plt.yticks(fontsize=15)
chart_title = 'Daily % Conversion per Segment ({}/{})'.format(conversion_label, traffic_label)
plt.title(chart_title, fontsize=18)
plt.legend(fontsize=15)
plt.show()
```

Bad reporting undermines all your hard work on the math and engineering behind getting results

The current state of reporting is suboptimal, and it creates a norm where data scientists on your team have to explain/translate the results in order to make them intelligible.

Good reporting practices:

- Don't try to teach p-values or stat tests
- Don't list 100 numbers without guidance
- Be opinionated, consistent with choice of numbers



CONCLUDED New Checkout Flow

May 2 - May 31, 2024

Decision: B Side cards scrolling (positive lift)

Owner: Monetization

Results Last Updated: May 31, 2024 at 7:00 AM EST

Diagnostics: All checks passed

Hypothesis

By changing the layout of the cards page to allow for more cards being displayed, we will observe an increase in users purchasing within 1 day. This is supported by our research that users who click on review or images are more likely to convert to purchase.

Key Takeaways

The new experiment yielded significant results, revealing an impressive 5% increase in the primary metric of 1-day purchase conversion. This outcome strongly supports the hypothesis that displaying cards rather than high-level categories influences users positively, leading to quicker purchasing decisions. The enhanced feature seemingly reduced friction in the buying process, encouraging users to commit to their purchases promptly.

Decision

Rolled Out Variant: B Side cards scrolling

Links

- Figma Mocks
- #slack-channel-name
- Product Requirements Doc
- Jira Issue: EP-1234

Metric Performance for Rolled-out Variant

B Side cards scrolling

Experiment Metrics

Metrics	A Control	B <Name...	Lift	-10% -5% 0% 5% 10%
PRIMARY			●	
			●	

Variations

B Side cards scrolling
side_cards_scrolling

A Categorized cards
cat_cards

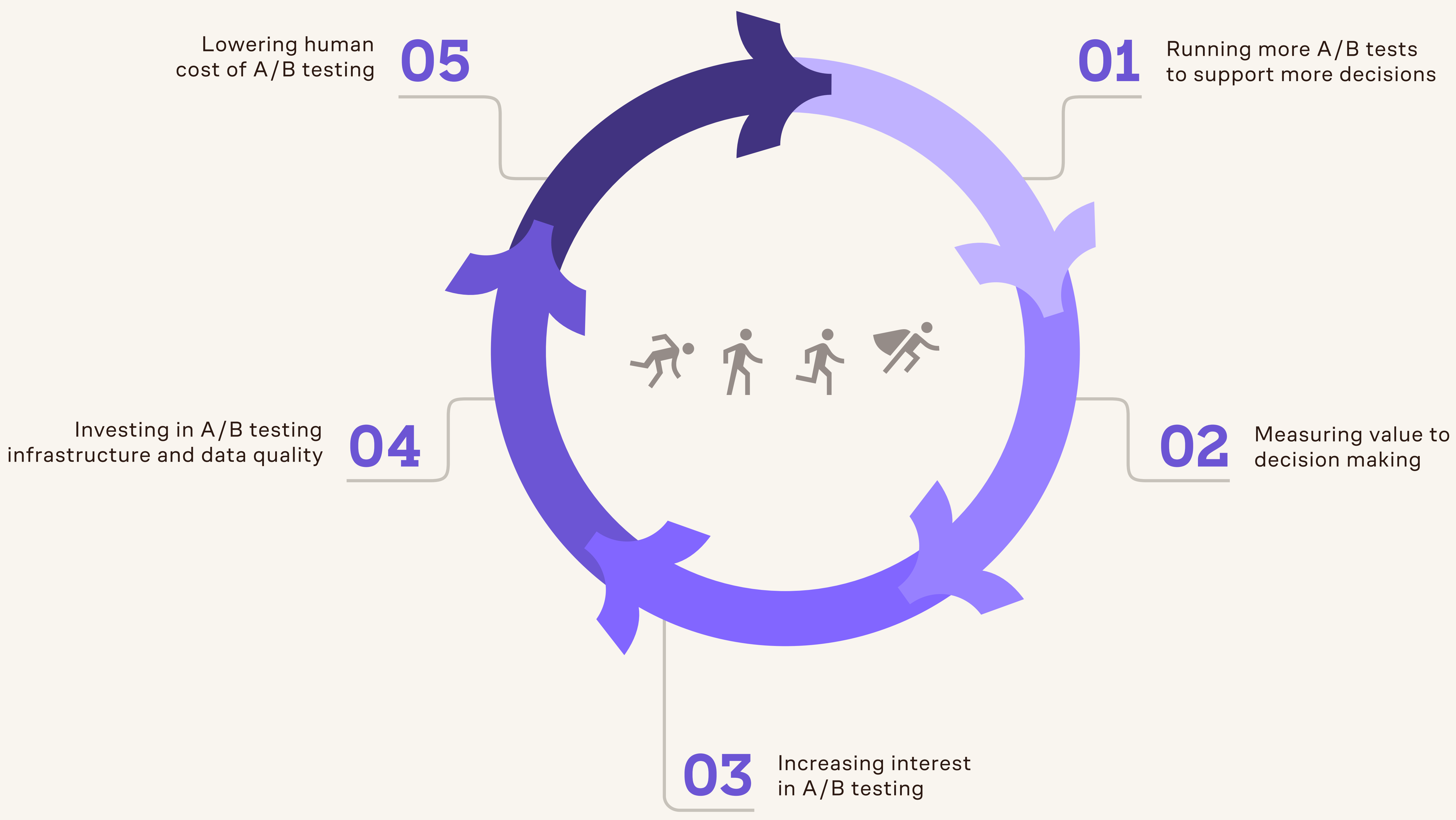
Good reporting assumes no statistics, infrastructure knowledge

The best forms of reporting don't assume context and knowledge. They don't assume you know what a p-value is, or what warehouse tables are. They simplify things, so that a junior Product Manager straight out of college can make a decision based on the results. This is a hugely underrated component of getting experimentation to scale.

Conclusion

Ultimately, building an experimentation tool is a complicated endeavor – there are a bunch of different components to get into sync. But even if you start out with something manual and low-fidelity, like Jupyter notebooks, you can begin to establish a flywheel that leads to experimentation success.

The A/B Testing Flywheel


01
Running more A/B tests to support more decisions
02
Measuring value to decision making
03
Increasing interest in A/B testing
04
Investing in A/B testing infrastructure and data quality
05
Lowering human cost of A/B testing

Crawl, Walk, Run, Fly Progression

When you run your first successful A/B tests, and start to influence decision-making on a small scale, you will start to increase your organization's interest in experimentation. Once you demonstrate that A/B testing leads to better decisions, your team will begin to invest in experimentation infrastructure, which ultimately lowers the labor costs of conducting further tests.

At Eppo, we think this flywheel is still a lot more painful than it needs to be. We built a tool that includes all the necessary components to quickly develop the experimentation culture that the most successful companies have used to win their markets.