DATA-DRIVEN ENGINEERING

OR: HOW I LEARNED TO STOP WORRYING AND LOVE TESTING IN PRODUCTION

Andy Cirillo - Conversant Media

WHAT IS DATA-DRIVEN ENGINEERING?

Engineers using data produced by production systems in a **disciplined** *and* **methodical** *manner to guide the development of new software.*

- Scientific Method
- Formal Experimentation
- Statistical Analysis

NOT GOING TO TALK ABOUT:

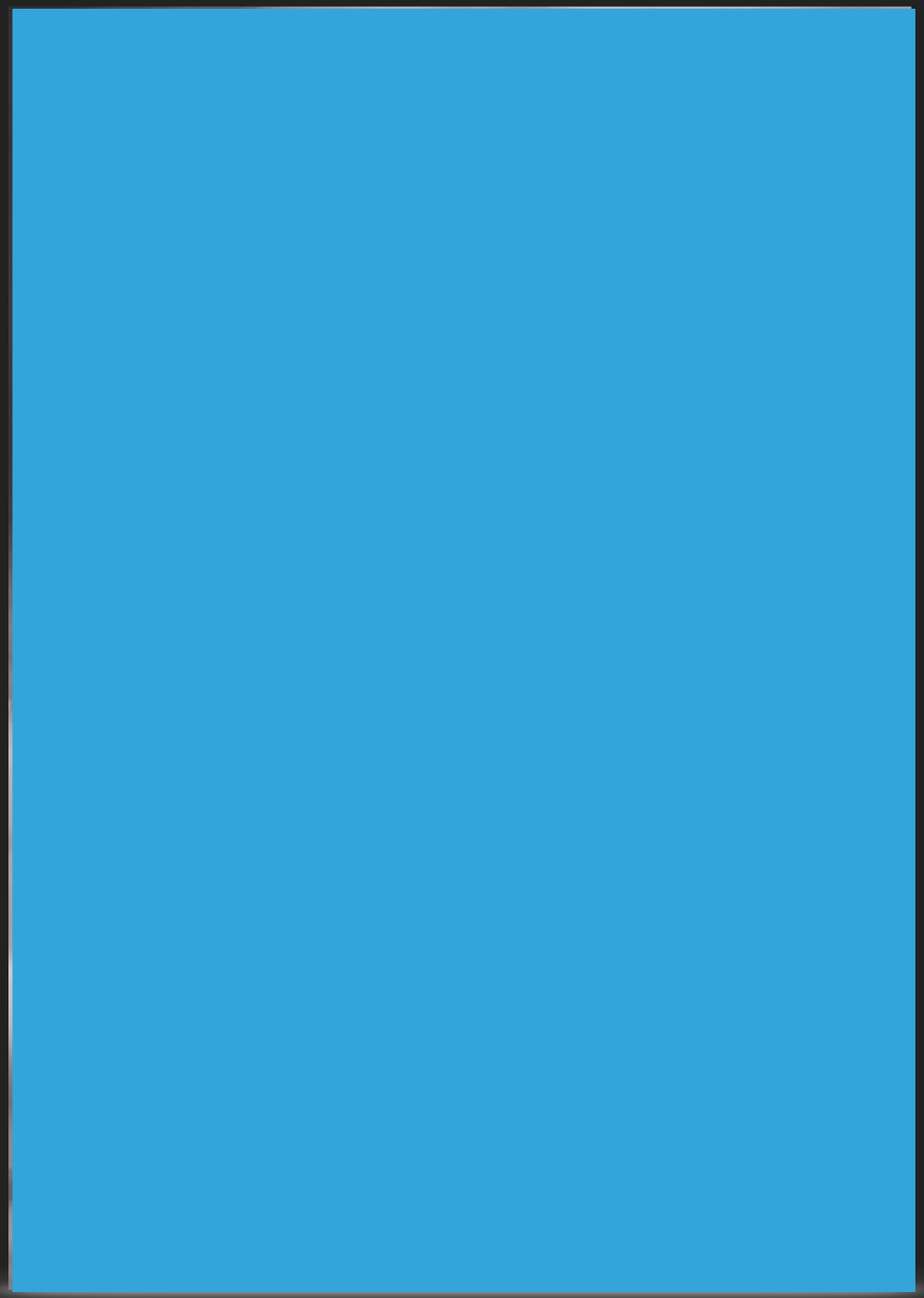
- *Data Science*
- *Data Engineering*
- *Data-Driven Product Management*

AM GOING TO TALK ABOUT:

- Software Engineers Using Data to do Their (Own) Jobs Better

WHY NOW?

- 100s of thousands, or even 100s of millions of users.
- Large-scale distributed systems with lots and lots of identical servers.
- Tools and infrastructure to collect, store and analyze massive event-oriented data sets.



BASIC INGREDIENTS

- Structured Log Format
  - CSV, Apache Avro
  - ~~log4j, slf4j~~
- Infrastructure for Collecting, Storing and Querying Distributed Log Data
  - Kafka, Flume
  - RDBMS, Data Warehouse, Hadoop
- Infrastructure for Partial Production Deployments

```
event_log : {

  // basic stuff
  timestamp : long,
  event_type : short,
  user_id : long,
  session_id : long,
  transaction_id : long,
  response_code : short,
  server_id : int,
  response_time : long,

  // optional extras
  skip_mask : int,
  diagnostic_blob : text

  // application-specific
  …

}
```

PART I:

TESTING IN PRODUCTION

EXAMPLE: A TYPICAL TEST SCENARIO

Scenario 1: Refunded items should be returned to stock

Given that a customer previously bought a black sweater from me
And I have three black sweaters in stock.
When they return the black sweater for a refund
Then I should have four black sweaters in stock.

Source: Wikipedia

Scenario 1: Refunded items should be returned to stock

For all customers, items, and t_n, t_m, such that $n < m$,
Given that the customer bought the item from me at time t_n
And I have x of those items in stock at time t_m.
When they return the item for a refund at time t_m
Then I should have x + 1 of those items in stock at time t_{m+1}.

- In general, most tests are a mix of **driver** code and **assertion** code.

- In practice, complete assertion code is often relatively **achievable**, complete driver code, however, is somewhere between **hard** and **mathematically impossible**.

STRATEGY:

1. Logically split tests into driver code and assertion code.
2. Write test assertions against log output instead of directly accessing state. *(Cor. Everything that needs to be tested needs to be logged.)*
3. In QA, run drivers and assertions, in production run just the assertions.

- Add sku, store_count to the event log.
- Log the current value of the store with every event.
- Split tests into “drivers” and “assertions.”
- Queries take the place of assertions.
- You can run the same queries in production!

```
select ts, store_count,
  case
    when event_type = 'purchase' then -1
    when event_type = 'return' then 1
    else 0
  end expected
from event_log
where ts >= current_date
and sku = ‘black sweater’
order by ts;
```

ts	store_count	expected
1556307290	**100**	**-1**
1556307378	**99**	**-1**
1556307386	**98**	**-1**
1556307450	**97**	**-1**
1556307457	**96**	**-1**
1556307465	**97**	**1**
1556307472	**96**	**-1**
1556307479	**95**	**-1**

WELL I, FOR ONE, DON'T TEST MY CODE IN PRODUCTION BECAUSE I HAVE UNIT TESTS!

thinks somebody in the audience . . .

TOP SIX REASONS WHY YOU SHOULD TEST IN PRODUCTION:

1. You don’t know all the scenarios.
2. Your imagination sucks.
3. Your product manager’s imagination sucks.
4. Things can change.
5. You still don’t know all the scenarios.
6. Production is different.

- This is a distributed system though…
- Store count is only *eventually* consistent…
- Have to relax the requirement a bit…
- But not too much!

```
select ts, store_count,
  case
    when event_type = 'purchase' then -1
    when event_type = 'return' then 1
    else 0
  end expected
from event_log
where ts >= current_date
and sku = ‘black sweater’
order by ts;
```

ts	store_count	expected
1556307290	**100**	**-1**
1556307378	**99**	**-1**
1556307386	**98**	**-1**
1556307450	**97**	**-1**
1556307457	**96**	**-1**
1556307465	**98**	**1**
1556307472	**96**	**-1**
1556307479	**95**	**-1**

```
select max(store_count) - min(store_count) = sum(expected)
from (
  select ts, store_count,
    case
      when event_type = 'purchase' then -1
      when event_type = 'return' then 1
      else 0
    end expected
  from event_log
  where ts >= current_date
  and sku = ‘black sweater’
);
```

```
select min(result) from (
  select sku, max(store_count) - min(store_count) = sum(expected) result
  from (
    select sku, ts, store_count,
       case
         when event_id = 'purchase' then -1
         when event_id = 'return' then 1
         else 0
       end expected
    from event_log
    where ts >= current_date
  )
  group by sku
);
```

CONCLUSIONS OF PART I

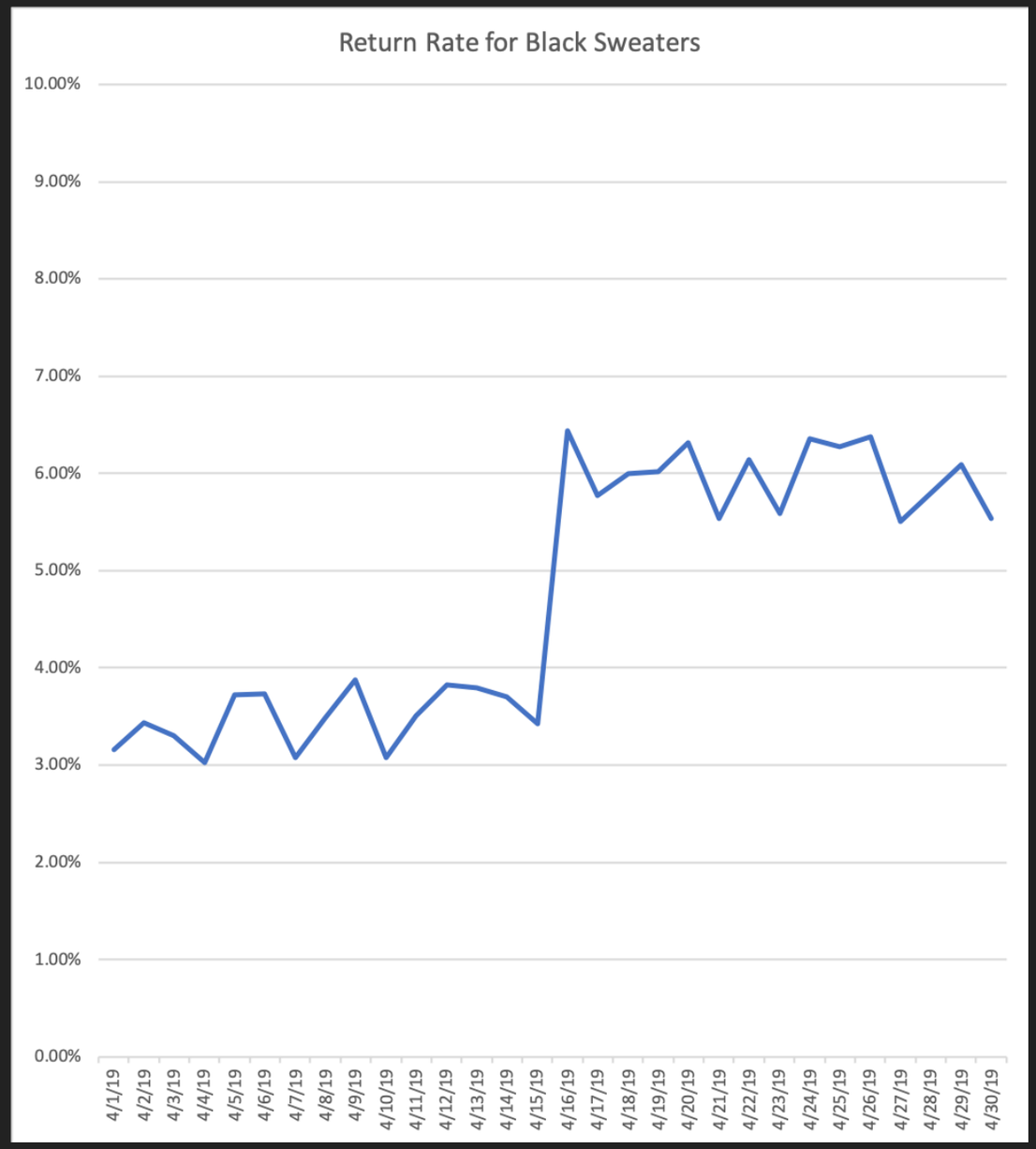
- By writing your test assertions against log output instead of internal state, you can use your QA tests in production.
- Production provides many, many, more test scenarios than you could possibly cover in QA.
- Some things can *only* be adequately tested in Production:
  - Non-Functional Requirements
  - Open-Ended Requirements
  - External Behaviors and Dependencies

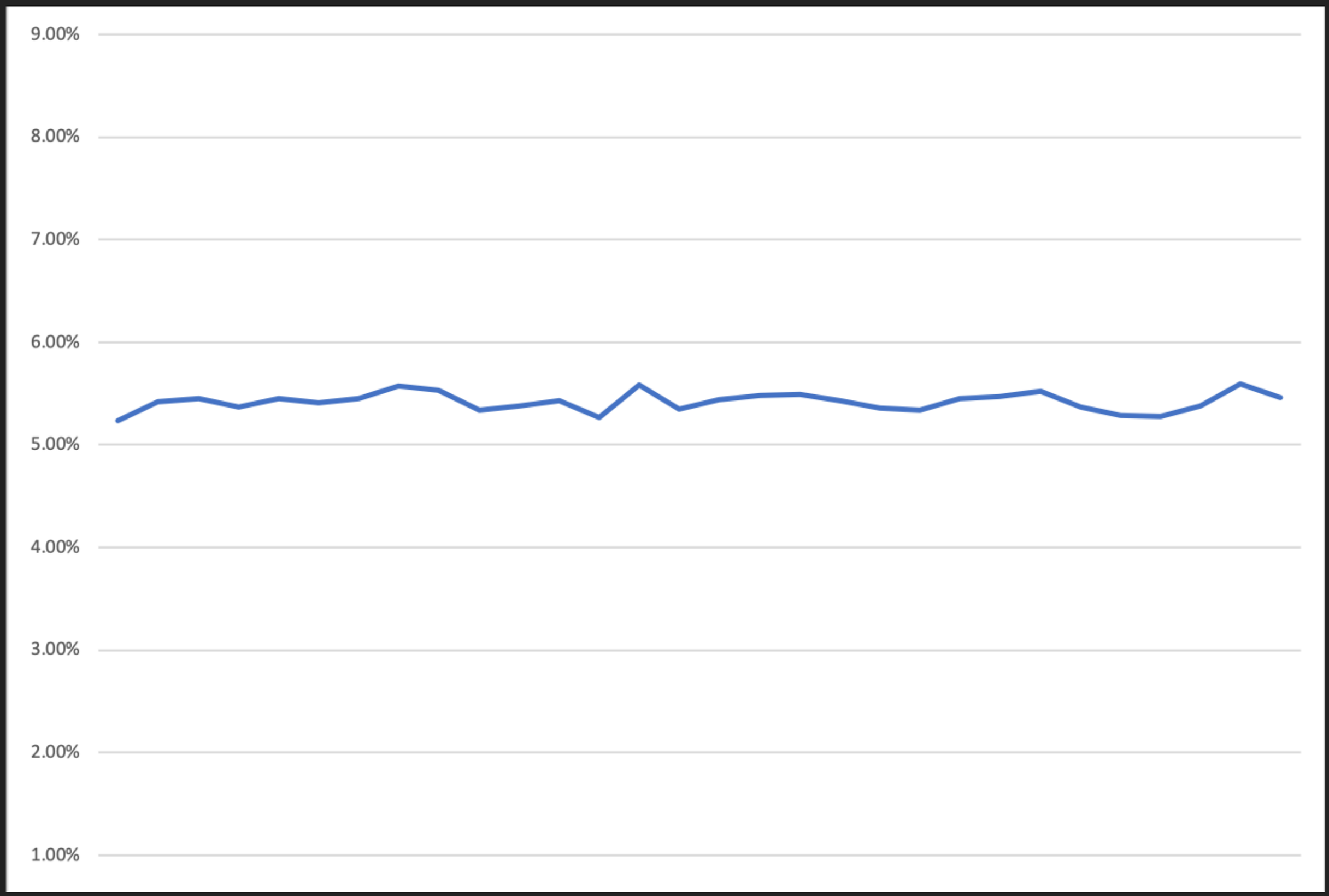
PART II:

FROM CONTINUOUS INTEGRATION TO CONTINUOUS EXPERIMENTATION

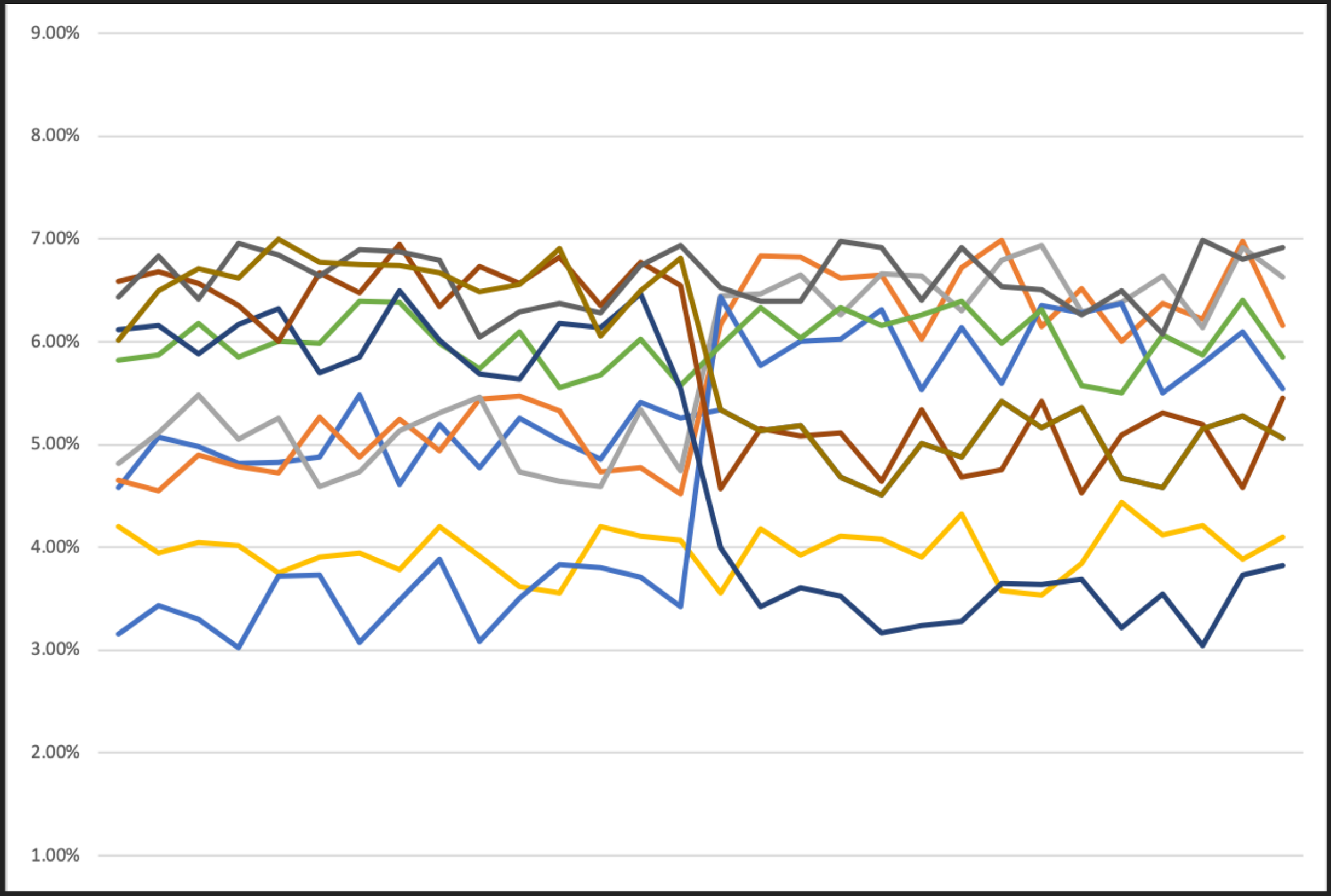
SCENARIO:

- At some point around 4/15, the return rate for black sweaters jumped to nearly twice the usual rate.
- High return rates hurt margins, leading to negative financial impacts for the company.
- New software was released on 4/15, but nothing exceptional.
- No one has complained about return rates for any products other than black sweaters.

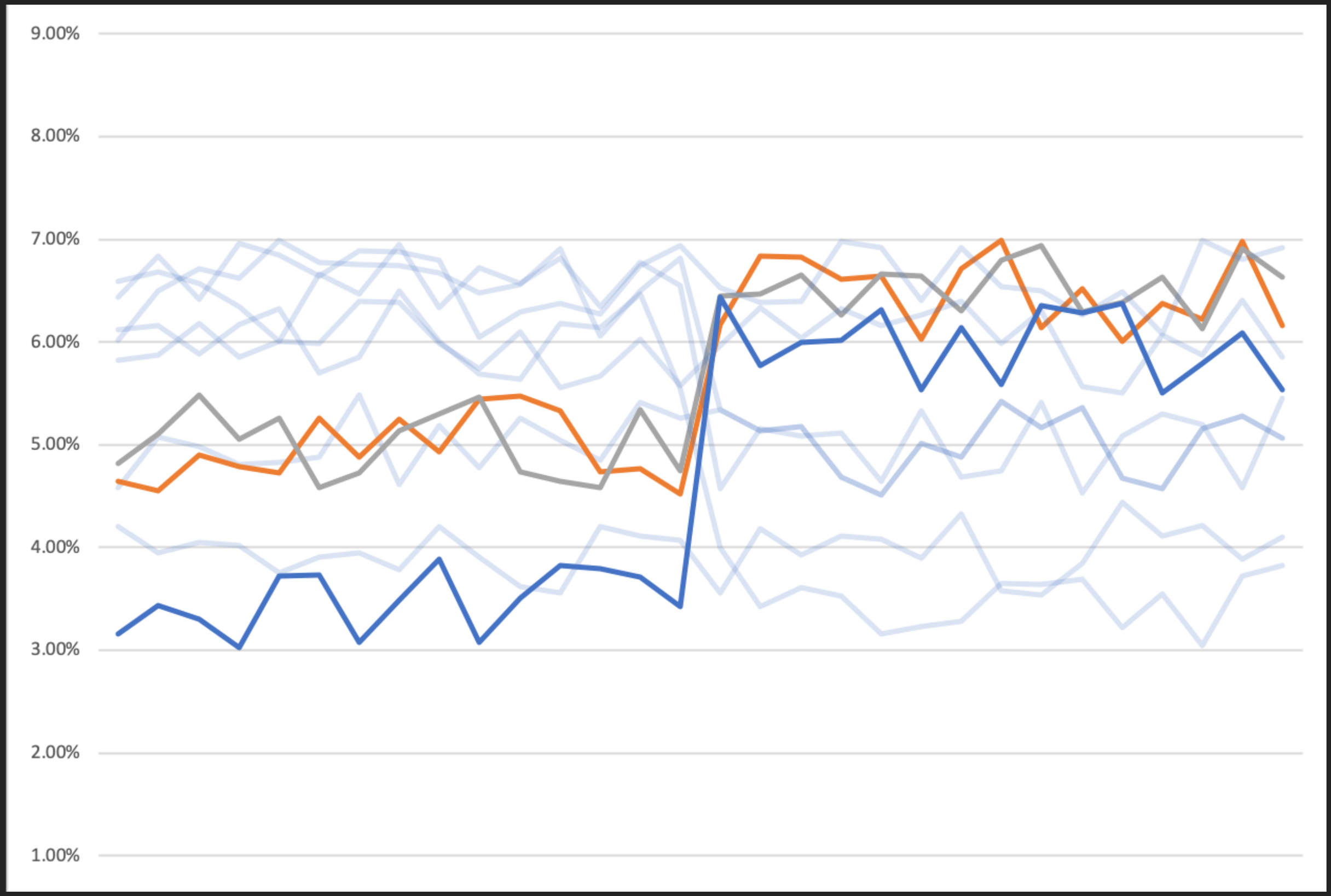




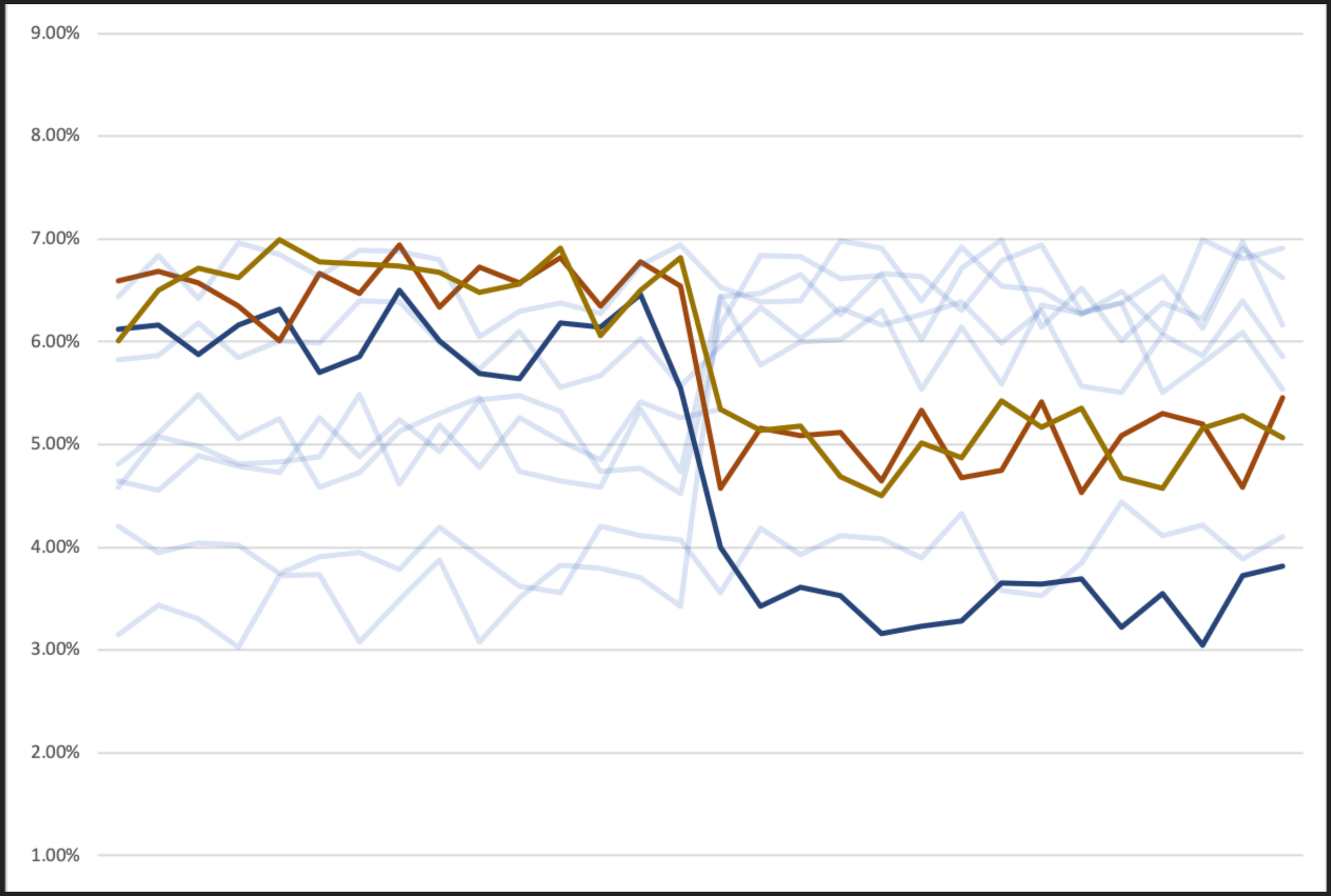
Overall Return Rate



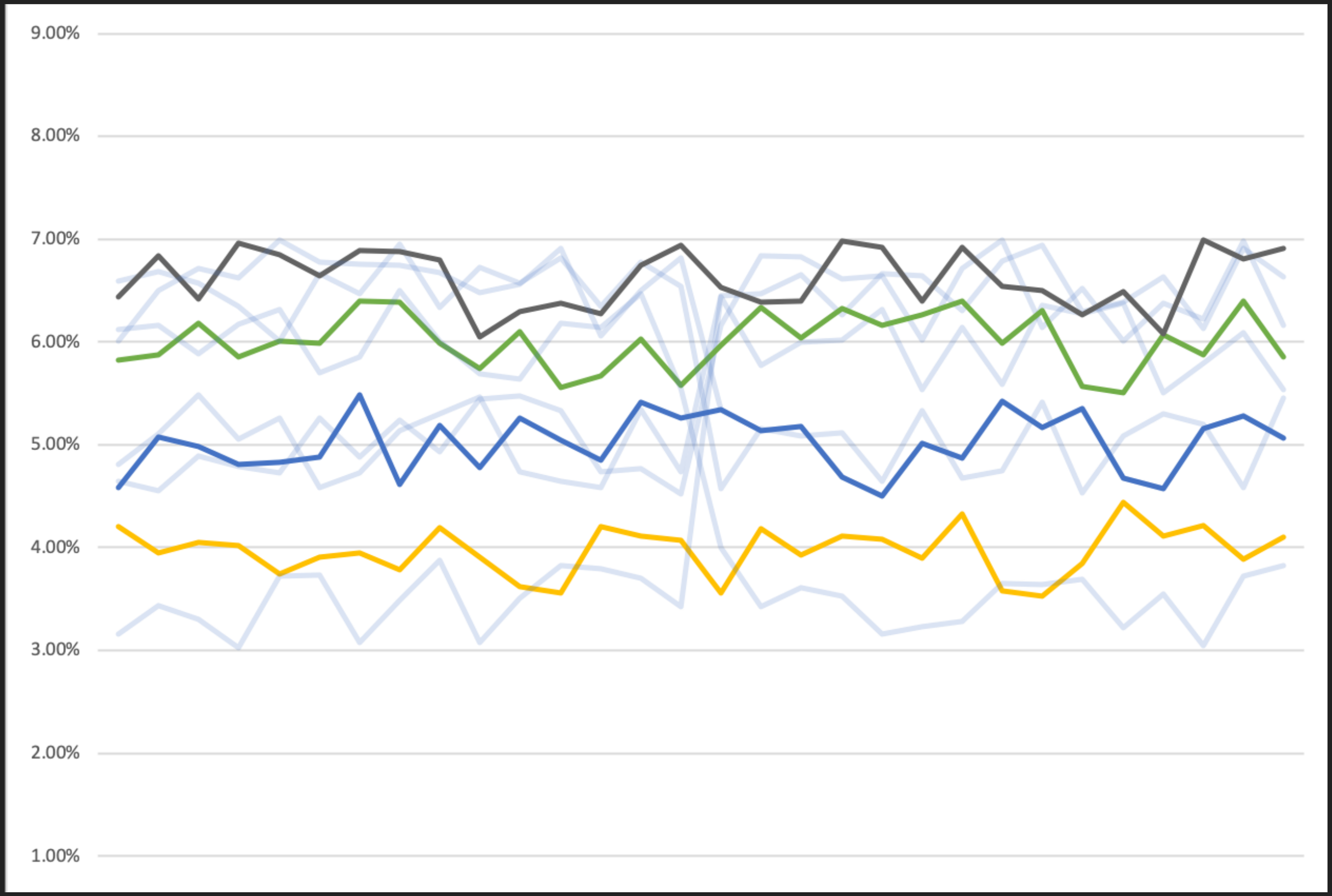
Return Rate by SKU



Return Rate by SKU



Return Rate by SKU



Return Rate by SKU

Code Before 4/15

```
select product_img
from sku_product_carousel
where sku = $1;
```

Code After 4/15

```
select product_img
from sku_product_carousel
where sku = $1
order by update_date;
```

Could the change in carousel order be causing higher return rates for some products?

How can we test this hypothesis?

SCIENTIFIC PROCESS

1. Formulate a Question
2. Define Hypothesis
3. Use Hypothesis to Make Predictions
4. Test Predictions Empirically
5. Refine Hypothesis (Step 2)
6. Draw Conclusions

OBSERVATIONAL STUDY

- If the change in the order of carousel images is what caused the change in return rates, then it would stand to reason that any products for which the order did not change, would not have seen a change in return rates.
- We can now look for products with only one image, or for which database order coincides with update_date.

CONTROLLED EXPERIMENT

- Make a code change that uses a different order.
    1. Revert to the original order.
    2. Sort by a different key.
    3. Use a random order.
- Deploy the changed code such that:
    - Some users get the original code (Control Group)
    - Some users get the changed code (Test Group)

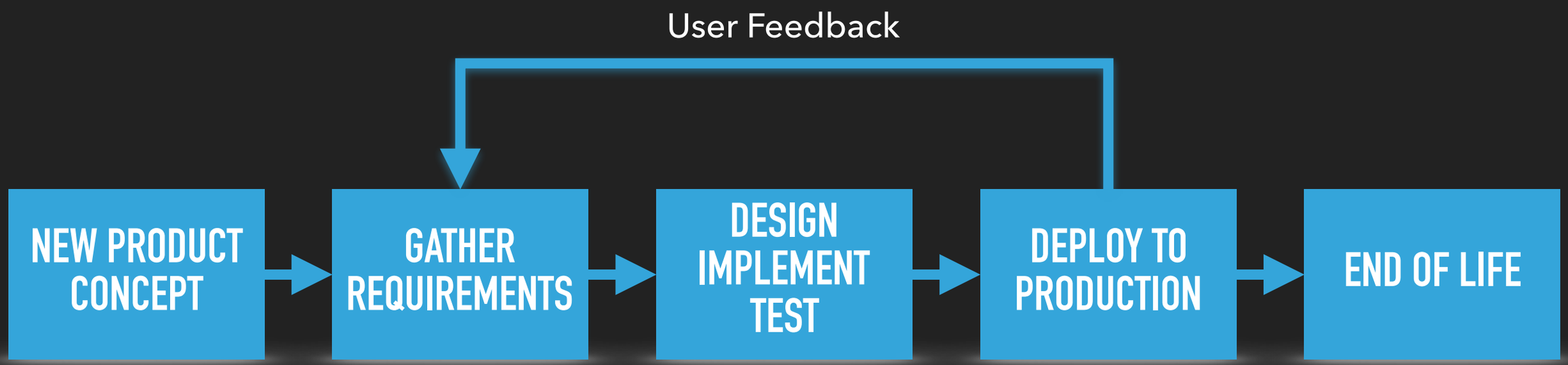
DEFINING TEST AND CONTROL GROUPS

- Deploy changed code to one server, use server_id to break out the data sets (canary).
- Use transaction_id to tie returns back to the original purchase.
- Branch logic in code, split on user_id.
- Use diagnostic_blob.
- What about Before and After???

```
event_log : {

  // basic stuff
  timestamp : long,
  event_type : short,
  user_id : long,
  session_id : long,
  transaction_id : long,
  response_code : short,
  server_id : int,
  response_time : long,

  // optional extras
  skip_mask : int,
  diagnostic_blob : text

  // application-specific
  sku : string,
  store_count : int,

}
```

CONTINUOUS EXPERIMENTATION

SOFTWARE ENGINEERING PROCESS

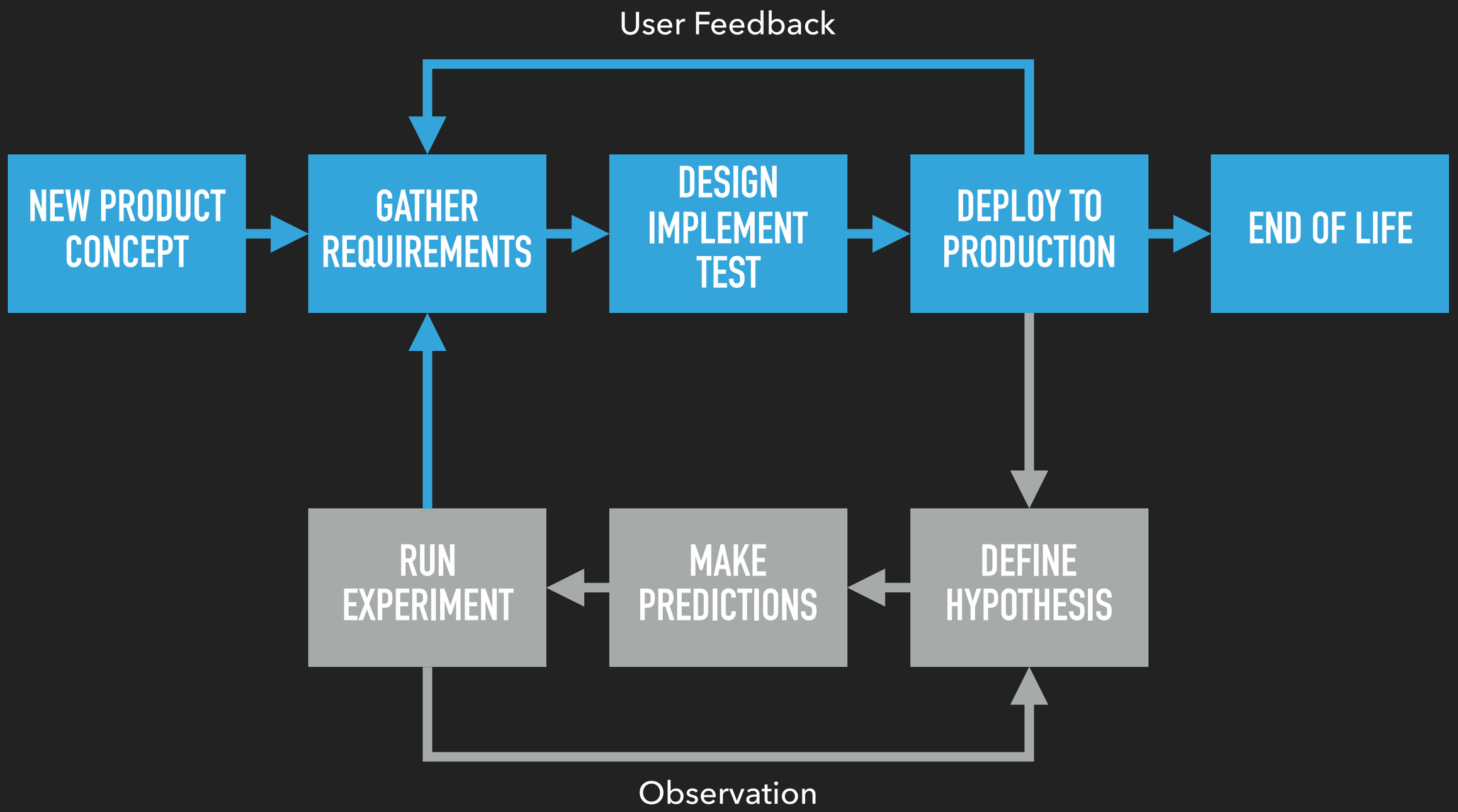


SCIENTIFIC PROCESS

The goal of the software engineering process is to solve a problem.

The goal of the scientific process is to answer a question.

DATA-DRIVEN ENGINEERING



CONTINUOUS EXPERIMENTATION – DEVOPS

- Must already have very robust CI and CD infrastructure in place.
- Must be able to deploy multiple, arbitrary, versions of code to production concurrently.
- Feature branching helps.
- Automated rollbacks.

CONTINUOUS EXPERIMENTATION – CULTURE

- Need to be comfortable with writing lots of throwaway code.
- Need to build a top-to-bottom culture of experimentation.
  - Engineers
  - Product Managers
  - Stakeholders
  - Partners/Clients
- Need to know what's off limits.