shopify

Tackling Slow Queries

A Practical Approach to Prevention and Correction

Kurni Famili
Senior SRE, Platform Resiliency
Singapore

Brad Feehan
Senior SRE, Platform Resiliency
Melbourne, Australia

Thanks for coming to our talk on "**Tackling Slow Queries**"!

- Kurni: originally from Indonesia, now living in Singapore.
- Brad: based in Melbourne, Australia

We're part of the globally distributed "Platform SRE" team at Shopify

- we work alongside other engineering teams
- to improve our infrastructure + keep it resilient
- Will talk more about what our team does in a minute.

Takeaways

01 **Shopify 💚 SRE**

02 **Resilient databases**
at planetary scale

03 **Slow Query Correction**

04 **Slow Query Prevention**

05 **Actionables**

06 **Q+A**

We’re gonna start off by

- giving a **quick glimpse** into Shopify and how it relates to SRE
- We’ll **recap the basics** of database resiliency at planetary scale
- **You’ll hear about** the opportunities we discovered, and the solutions we built
    - to correct + even prevent, slow queries
    - inside a large company, with a high development velocity
- We’ll close with a list of actionables you can take back to your org / team, followed by 10 minutes dedicated for Q+A.

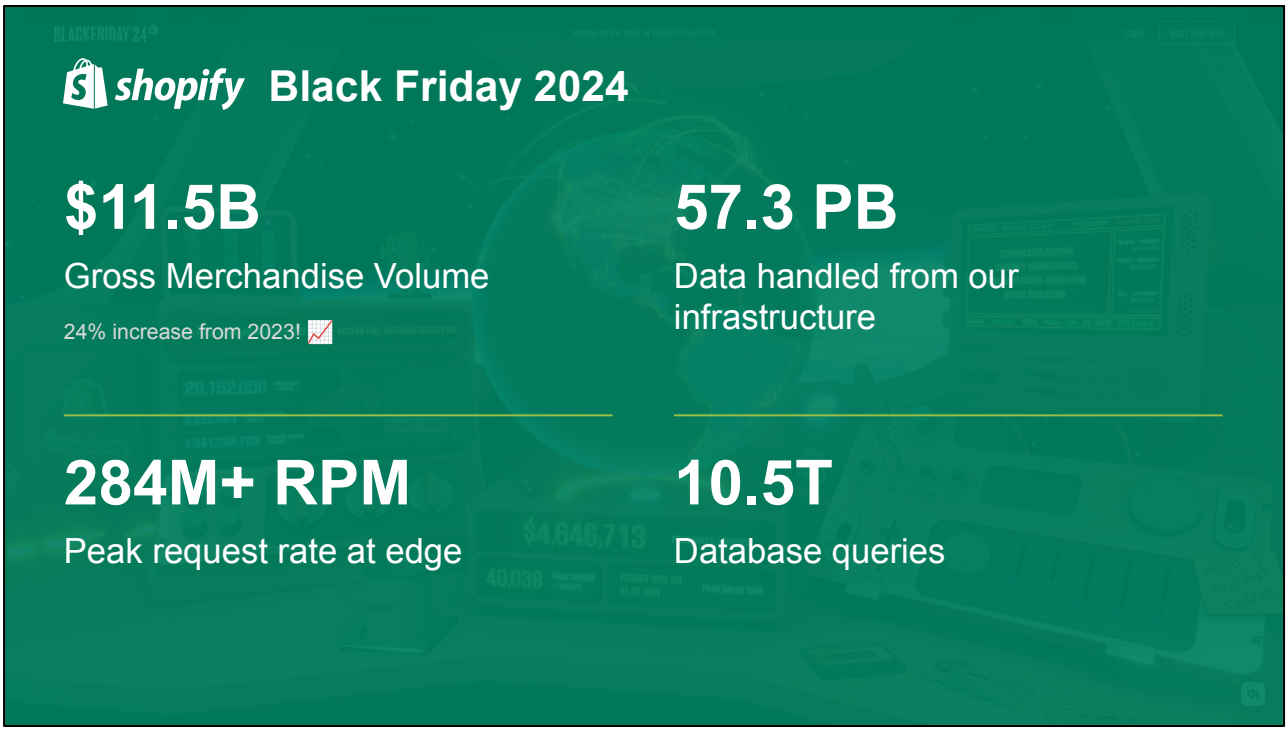
shopify

Since 2006 we've grown to ~8,000 employees, and **millions of merchants** in 175 countries.

- Shopify's mission is **make commerce better for everyone**.
- Millions of entrepreneurs, all around the world, trust us to run their business.

For those entrepreneurs, the biggest weekend of the year is Black Friday/Cyber Monday weekend

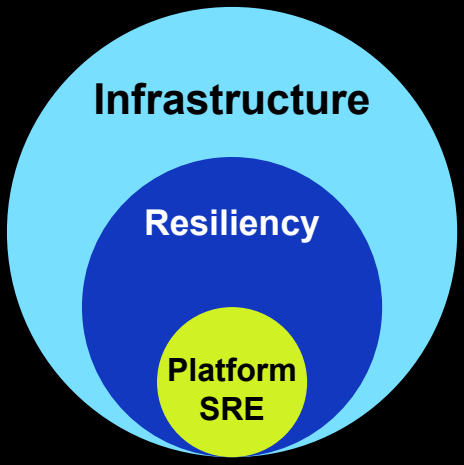
- All that activity was visualised in this cool globe built in-house, rendered in real-time for the public to see
- Our merchants' total sales peaked at $4.6M per min



Final stats for that four-day weekend:

- Our merchants' sales totaled 11½ billion dollars, up 24% from the year before
- We handled 57 peta-bytes of data across our infrastructure
- Request rate peaked at 280 million RPM at the edge
- and our databases served a total of 10.5 trillion queries, with a peak of 45M QPS == 2.7B QPM

shopify
Engineering



Within Infrastructure group @ Shopify:

- The resiliency organization
  - **Primarily** focused on **checkouts**, **storefronts**, and **admin**.
  - We want to **make resiliency easy** for developers
  - and a critical part of our culture.

This is the home of SRE at Shopify.

- We build tools and processes to improve reliability
- But most importantly, We hold a pager 24/7 whe

We hold the pager 24/7

- Follow-the-sun
- We command incidents
- We operate mitigation tools at any layer of the stack
  - from **the edge** and **networking**, all the way through to **the app** layer, **compute or persistence**, you name it.

SREs @ Shopify?

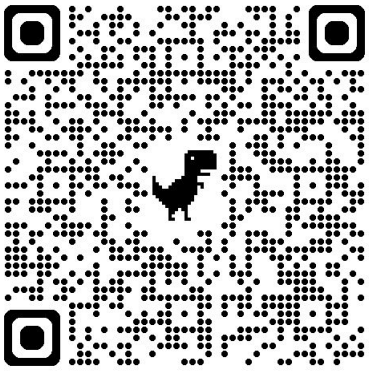
Which is all about

- remaining calm and collected during fires (incidents) as we are the first in-line to be notified when something goes wrong on the platform
- we make sure we **bring in everybody** we need, to mitigate the incident
- guide the conversation, gathering relevant context
- Then follow-up, address the root cause, and conduct a proper post-mortem.

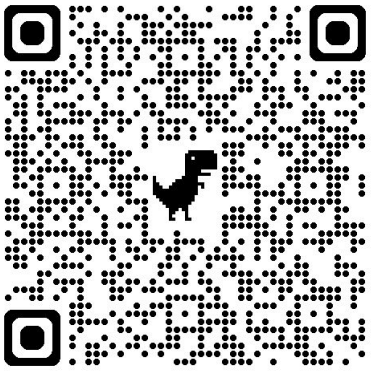
The **reality is** we have **so much support** and recognition,
and we're **definitely set up for success**.

So it actually is fine.

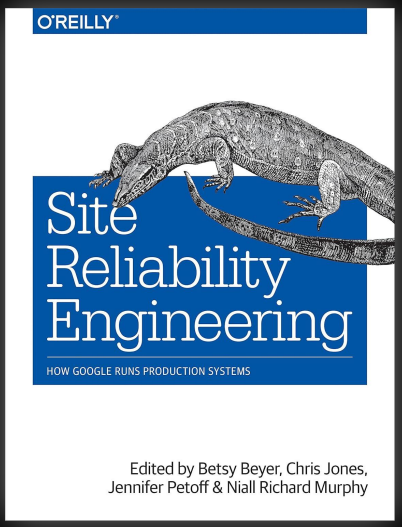
Site Reliability



Infrastructure



If that sounds like an interesting position to be in, come talk to us! We're regularly hiring SREs.



One important mantra at Shopify reminds us to

- “Do things, tell people”
- We believe in sharing our experiences, our work, and our ideas with each other.
- and documenting outcomes (**both** positive and negative)

The original SRE book

- strongly emphasizes storytelling and learning

That’s why we wanted to talk about, this project we are working on.

- For both of us, it’s our first time speaking at a conference
- we’re extremely grateful
- and humbled to be among the ranks of so many great speakers and industry professionals like you at SREcon.

Project inception

We follow up incidents

Empowering teams to make the platform resilient

Our database is resilient

But slow queries still cause severe incidents

It came about because:

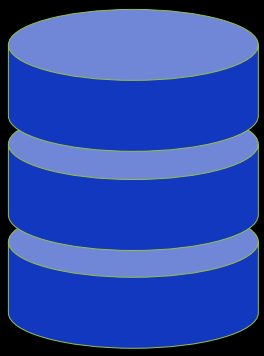
- When we follow up incidents, we empower teams to make the platform more resilient.
- This has **resulted in a very resilient database setup**, as Brad will cover a bit later
- And yet we observed, **as many as 1-in-5 merchant-impacting incidents** escalated to SRE, were **attributed to slow queries**

Hand over to Brad to go over what makes our database resilient

```
        SELECT too, much, data FROM big_table ...

INSERT INTO growing_table VALUES ...

   UPDATE everything_at_once SET ...
```



Speaking of flashy visualisations, here's what happens when you send too many slow queries to your database…

```
        SELECT too, much, data FROM big_table ...

INSERT INTO growing_table VALUES ...

   UPDATE everything_at_once SET ...
```

…and it falls over… and it gets sad… and it catches fire.
You know how it is.

Slow queries exhaust database capacity.

“Can’t we just pay for more?  ”

– Any shrewd, enterprising businessperson

If slow queries use up all the capacity of your database, the first thought is to buy more capacity.

If we’re losing more money than it would cost to pay somebody else to fix the problem, it’s a logical conclusion

Welcome to the Database Capacity Shop

Vertical scaling
(larger instances)

- More CPU cores
- More RAM
- Faster storage (NVMe)

Horizontal scaling
(more instances)

- Add read replicas

Utilization
(tune configuration)

- InnoDB buffer pool: based on system RAM
- Multi-thread replication
- Custom optimizer cost constants (MySQL 5.7)

Two main ways to **buy more database capacity** short term.

1. **Vertical**: Bigger, upgraded instances
2. **Horizontal**: More instances (you are using read replicas, right?)
3. You will only see improvement
    - if it's **configured to optimally use** the available resources.

Database Resiliency Investments

- That covers **short** term scaling
- beyond that: you need to make longer-term investments
- **to keep scaling** beyond a certain point.

At Shopify:

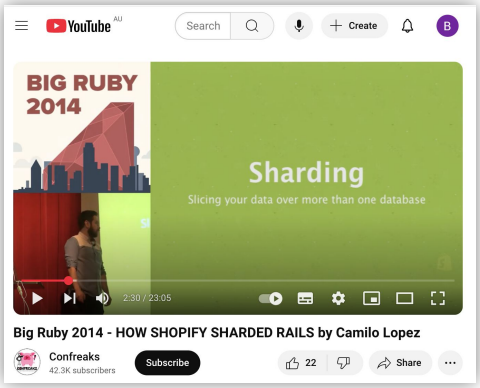
- I'll speed-run through this section to give some background context
  - I won't be able to go into detail or teach much about these
  - But, I'll **point you towards other resources** that cover them in more detail
  - This talk is focused more on what makes it through the gaps in the setup.
  - If it's too fast, don't worry too much, the gist of it is
    - Pretty much **everything you can do**
      - to **deploy MySQL in a more resilient** way
      - it's **already been implemented** at Shopify.
  - And more-or-less, everyone involved has done a great job but things still get through.

- **Here's what I think are the six most** important historical investments we've made

01: “Horizontal” Sharding

Partition tenants into separate instances
- Unlocks scalability
- Bulkheads: isolate failures
- Tenant isolation

Downsides
- Way more complex
- Re-balancing
- More instances, more problems?



[How Shopify Sharded Rails](#)
by Camilo Lopez
Big Ruby 2014

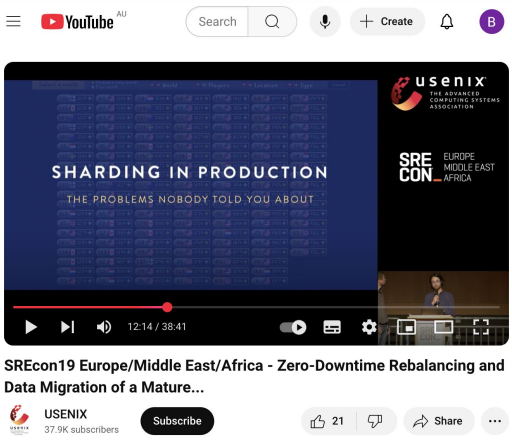
Separating tenants into partitions
- One database only has so much resources.
- So i**f you’ve reached the limits** of scaling
  - partitioning like this **unblocks further** horizontal scaling (**more instances**).

Some other bonuses: bulkheads, rebalancing, isolation for noisy neighbours.

We had to make this investment for **2013 holiday season**
so it’s been a big part of our architecture.

It increases complexity at the application layer
and operationally you have to balance the shards equally
and toil increases with instance count.

Balancing



SREcon19 Europe/Middle East/Africa - Zero-Downtime Rebalancing and Data Migration of a Mature...

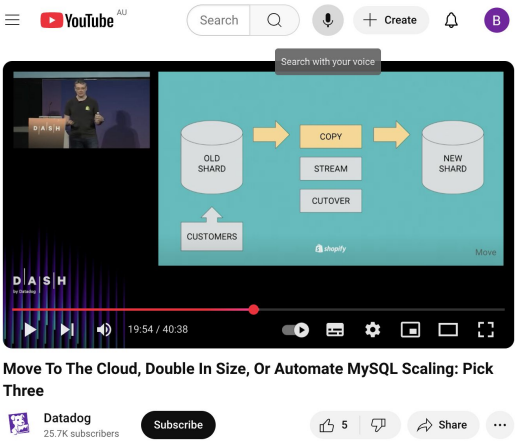
USENIX 37.9K subscribers Subscribe 21 Share

[Zero-Downtime Rebalancing and Data Migration of a Mature Multi-Shard Platform]

Justin Li and Florian Weingarten

SREcon19 EMEA

Automation



Move To The Cloud, Double In Size, Or Automate MySQL Scaling: Pick Three

Datadog 25.7K subscribers Subscribe 5 Share

[Move To The Cloud, Double In Size, Or Automate MySQL Scaling: Pick Three]

Aaron Brady

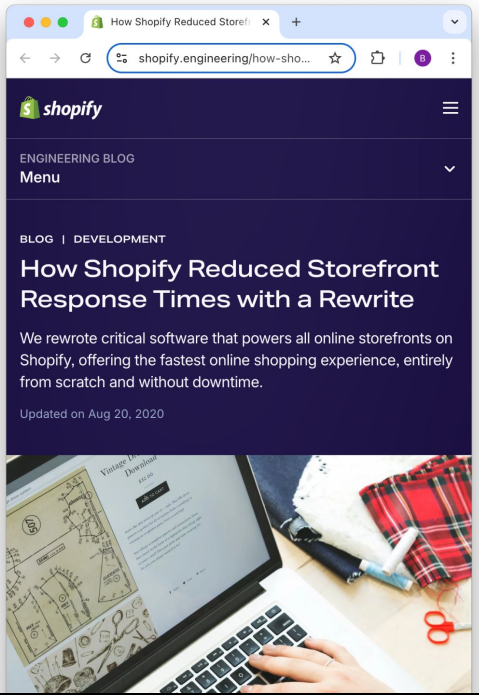
DASH by Datadog 2018

Some good talks that cover how we handle shop moves and balancing shards
And automation to relieve the burdens of operating a bigger fleet of databases.

02: “Vertical” Sharding

Functional partitioning
- Extract database tables
- or services

Examples at Shopify
- Storefront Renderer
- Identity

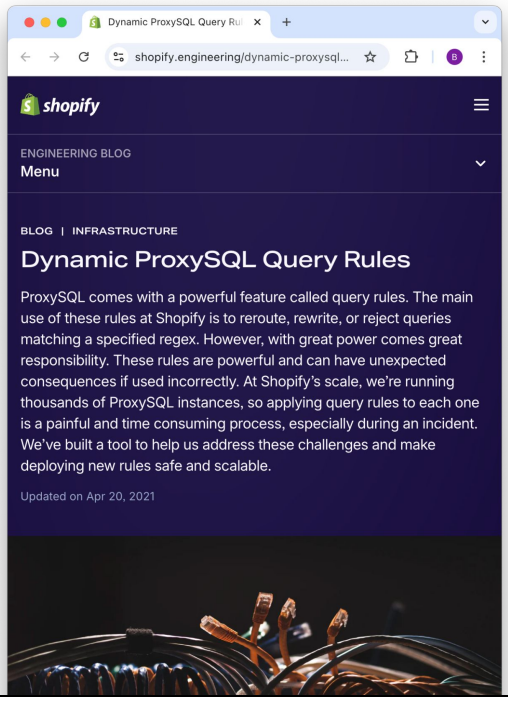


Not all workloads are equal.
Extracting a well-contained system could improve reliability for the rest of the system.
Downside: are you sure you can handle operating two services?

03: Connection Management

Fail-fast
- Timeouts
- Circuit breakers
  - Backpressure

ProxySQL
- Multiplexing
- Routing, failovers
- Aggregated query metrics



Connection management
- Timeouts
- Circuit breakers

At our scale, it's not very resilient to just connect to "the database" and hope for the best.
- ProxySQL combines many connections from app (multiplex)
  - Reduce load on DB
- Can swap out the database the app is talking to
- Gives metrics about specific query patterns

04: Schema Migrations

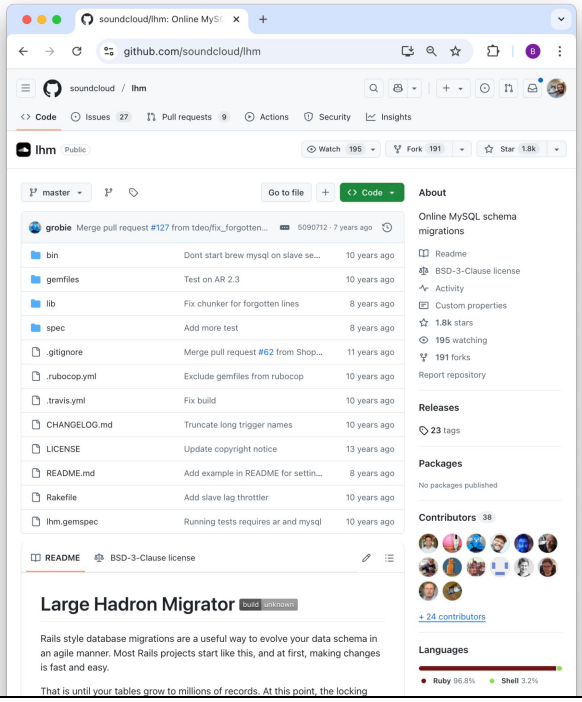
Large Hadron Migrator (LHM)

- **Zero-downtime** online schema changes
  - using copy table + triggers

Downsides

- Complex developer workflow

https://github.com/Shopify/lhm



Applying schema migrations without extreme care can block throughput.
Devs need to ensure code can handle running on both version of code before + after, to allow for phased deployments, rollbacks, etc.

05: Deployment processes

- Automatic rollbacks
- Feature flags
- Canary clusters
  - with synthetic traffic
  - or shadow traffic

Need a quick way to rollback

- Features to gate big feature releases
- Testing with synthetic traffic catches glaring mistakes

06: Developer Empowerment

Query design and optimization

- Query planning (`EXPLAIN`)
- Indexing
- Optimizer hints (a last resort)
  - e.g. `FORCE INDEX`, `IGNORE INDEX`
- Cache most common queries
  - Even fast queries add up

Dev empowerment is much more important at **larger companies** (employee count) because work is **distributed** among a lot of engineers, and needs to be a high standard

Important topics:

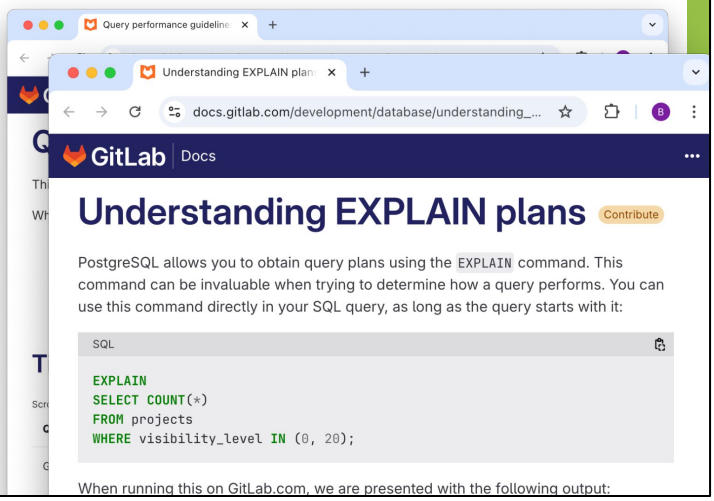
1. Query **plans** and EXPLAIN
2. Proper **indexing**
3. Levers available as last resorts; like FORCE or IGNORE INDEX
4. and **caching** – because **even fast queries** add up at scale

Caching – Warning: can mask performance issues when cache is warm

06: Developer Empowerment (cont)

Public case study – GitLab
Comprehensive docs and advice

- Query performance guidelines
- Adding Database Indexes
- Pagination performance guidelines
- Understanding EXPLAIN plans



Developer documentation:

Excellent starting point if you don’t have these internally already

06: Developer Empowerment (cont)

Public case study – GitLab (PostgreSQL)

Tools for experimenting proactively

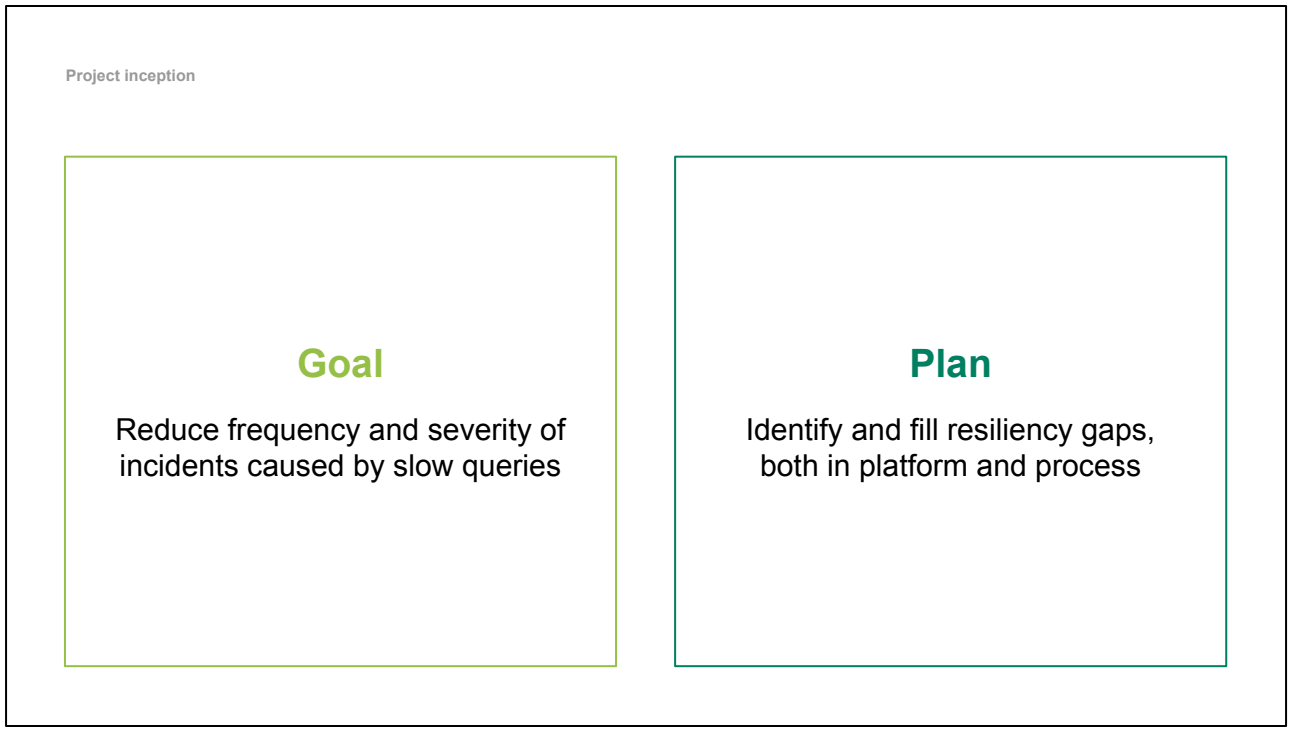
- Database Lab Engine
  - “DBLab Engine enables 🖐 database branching and ⚡ thin cloning for any Postgres database, and empowers DB testing in CI/CD”
- JoeBot
  - DBA Slack-bot, for query optimization

These tools allow performance experimentation with real datasets.
Shopify has something similar internally for MySQL to run EXPLAIN against multiple shards.

Database Resiliency Investments

01	**“HORIZONTAL” SHARDING** *(TENANT PARTITIONS)*
02	**“VERTICAL” SHARDING** *(SERVICE EXTRACTION)*
03	**CONNECTION MANAGEMENT**
04	**SCALABLE SCHEMA MIGRATIONS**
05	**DEPLOYMENT RESILIENCY**
06	**DEVELOPER EMPOWERMENT**

This was our starting point – very solid database setup
It’s not magic, but it’s table stakes for deployment at our scale.



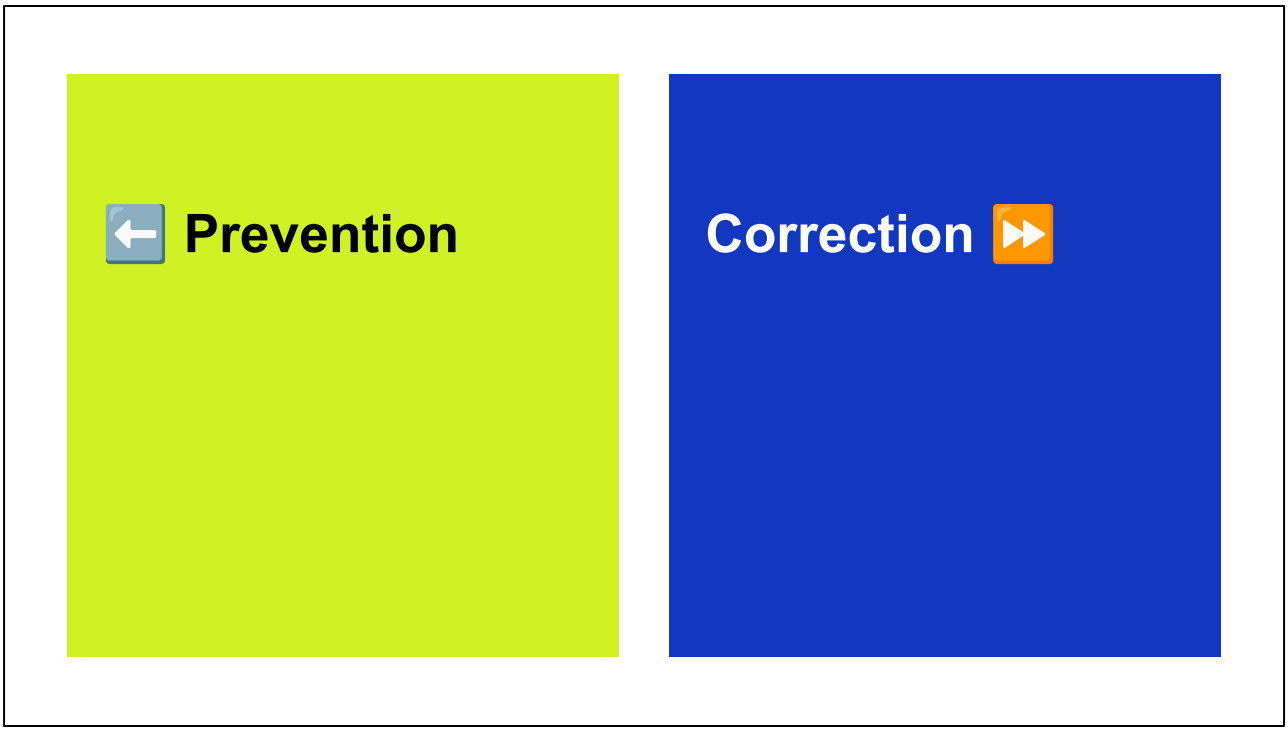
Given that setup, and that we're still seeing slow queries cause incidents:

The goal Kurni and I had at the start of the project

- Reduce frequency and severity of incidents caused by slow queries

Our plan to achieve that was

- Go through our resiliency setup
- Identify and fill gaps
- In our platform and processes



We focused our priorities in two main areas

Prevention: how can we stop slow queries from being deployed

- And from causing an issue in the first place?

Correction is about:

- timely **detection**, **mitigation**, and **remediation**
- when slow queries **are** found in production.

shopify

Correction

"So You Have A Slow Query, Now What?"

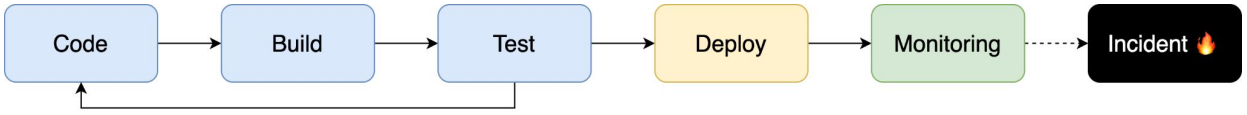
Speaker

Kurni Famili

Senior Site Reliability Engineer

Thanks, Brad. So you have a slow query. Now what? What can we do to address it?

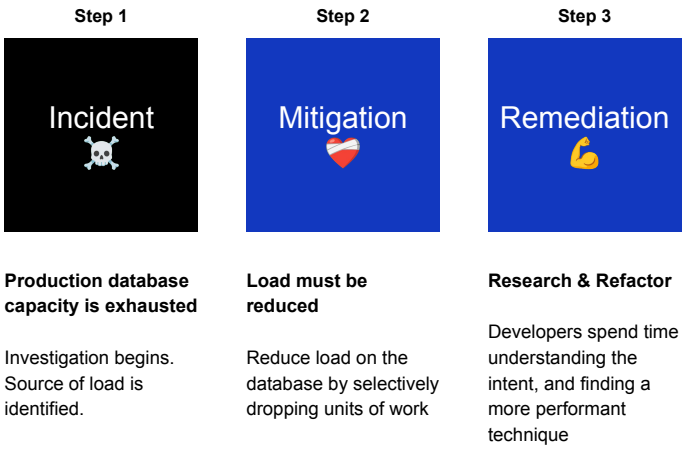
Development lifecycle



Before we jump to business, this is a snippet of a typical dev lifecycle that we're familiar with.

- **Code, build, test**
- **Deploy, then we sort of go into the eternal phase of monitoring**
- **In the worst case, a degraded state was observed and we go to incident phase.**

Incident Lifecycle



Looking deeper into the incident phase lifecycle, it would look something like this for a slow query incident.

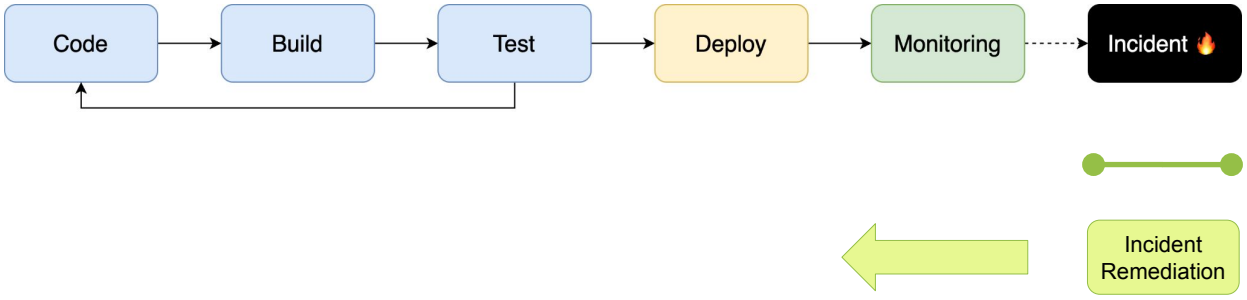
1. DB capacity is exhausted, investigation begins to identify the source of load.
2. Then we move to Mitigation, where load usually must be reduced. Units of work is dropped by applying firewall rules, API throttling, and so on.
3. For Remediation, we work alongside devs to understand the intent of the DB query, and find a more performant technique.

This is our “catch-all” process, or how we follow up incidents that fall through the cracks of ALL our defenses. Everybody here probably has their own version of this.

This whole cycle takes a LONG time:

- It’s an intentionally rigorous process
  - as the whole point is to learn as much as possible from unique failures
  - and ensure we never fail twice for the same reason
- But it’d be less useful when it’s the same pattern repeatedly

Development lifecycle (2)



Going back to this diagram, the incident remediation process falls under this part. Our goal going into the project is to find opportunities to shorten the feedback loop, essentially [CLICK] shifting left on this diagram.

Identified gaps

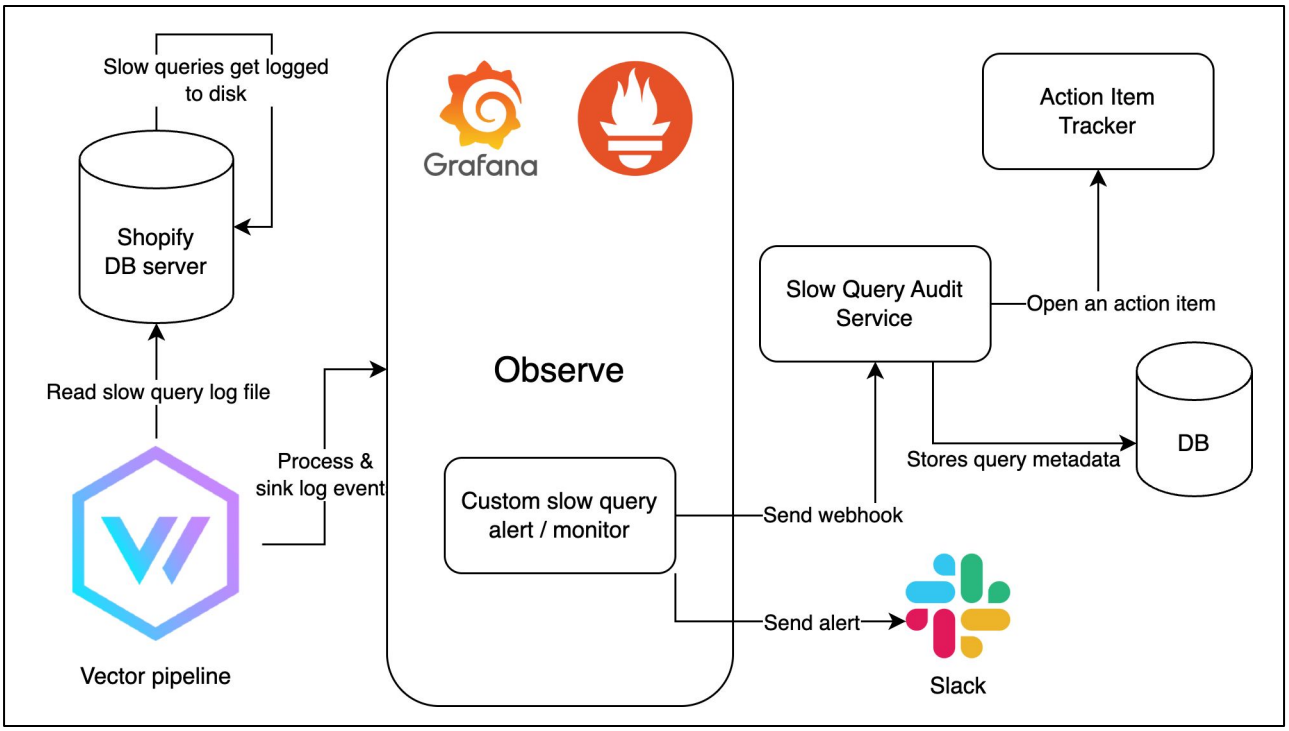
Or things that suck

Dependency on incidents

Lack of ownership and accountability

Fragmented developer resources for debugging

To summarise, the gaps we saw on our previous system was these.
We will explain more about them and how our solution addressed each in a bit.



First, we'd like to give a high-level overview of what we built.

1. Starting from the DB layer, our DB servers emit slow query logs when queries exceed a certain threshold in execution time. For MySQL, what we're using, this is built-in and configurable.
2. [CLICK] These logs are then read by our Vector pipeline. Vector is an open-source, lightweight observability pipeline solution. In this case, we use it to read the logs, preprocess, then [CLICK] sink the logs to our in-house observability service, called Observe.
3. Observe is our unified observability system built in-house, based on Grafana and Prometheus. We actually have a public talk going deep into what Observe offers on our YouTube channel (ShopifyDevs), if you're interested to know more.
4. But anyway, Observe contains our logs, and we also configured an alert rule / monitor for slow query logs, that aggregates the parsed log data on fields such as P99 execution time, the size of the fetched data (heaviness), and if a query exceeds a certain threshold, say 1s, then the monitor will be triggered.
5. [CLICK] In which case, a Slack alert will be sent to our alerts channel.
6. [CLICK] Then more importantly, a webhook will be sent to our slow query audit service, [CLICK] which would automatically create an action item with our action item tracker, and auto-assigns them to the relevant team.
7. [CLICK] It will also store some query metadata, for example, to prevent multiple open action items for the same slow query.

Dependency on Incidents

Ideally, **we** detect bad queries (before end-users)

Response

- Built automated alert
  - Auto-create action items
  - Catch bad queries early
- Prioritize worst-performing queries

Now, let's see how the system we built addressed the gaps.

The first gap is dependency on incidents.

In our incident remediation model, we only discover bad queries after they have had an impact. At best, it will be a case of slow response time. At worst, it might bring the whole DB down. Not to say that this model is wrong, as we can't catch every bad query, but for certain patterns we've seen before, we can catch them early.

—

The system we built:

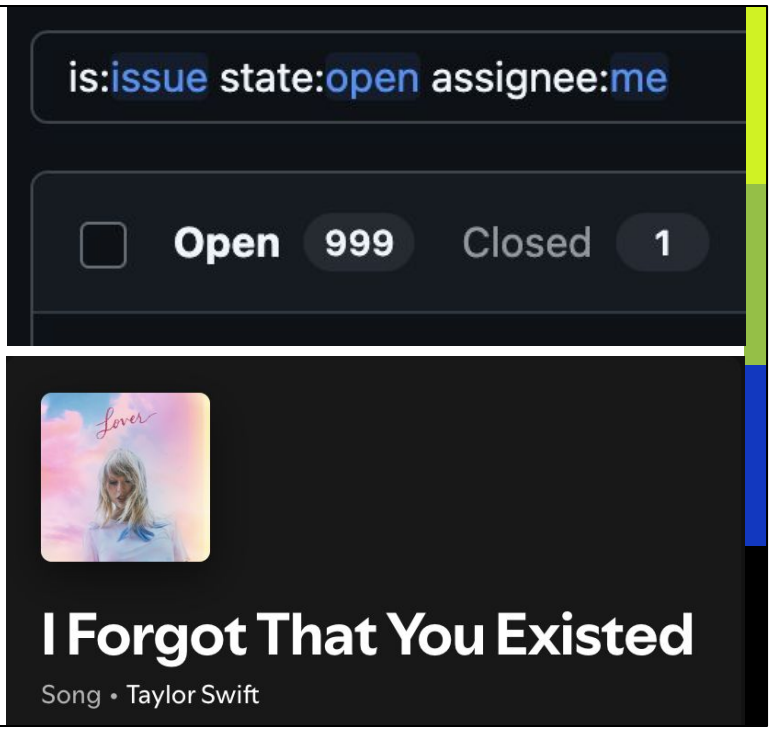
- Included an automated alert that will auto-create action items (AIs)
  - Removing manual effort
  - Catching slow queries before they manifest into full-blown incidents
- As the query data is parsed and aggregated for each query, we could prioritise the fix for the worst offenders, the ones hogging the CPU the most.

Lack of ownership and accountability

An issue can be open indefinitely, with or without an assignee

Response

- Automating assignment
- Ownership of tables/queries
- Leadership buy-in
- Merge-blocking



The second gap is lack of ownership and accountability.
We see in some cases, an issue can be open indefinitely, with or without an assignee.

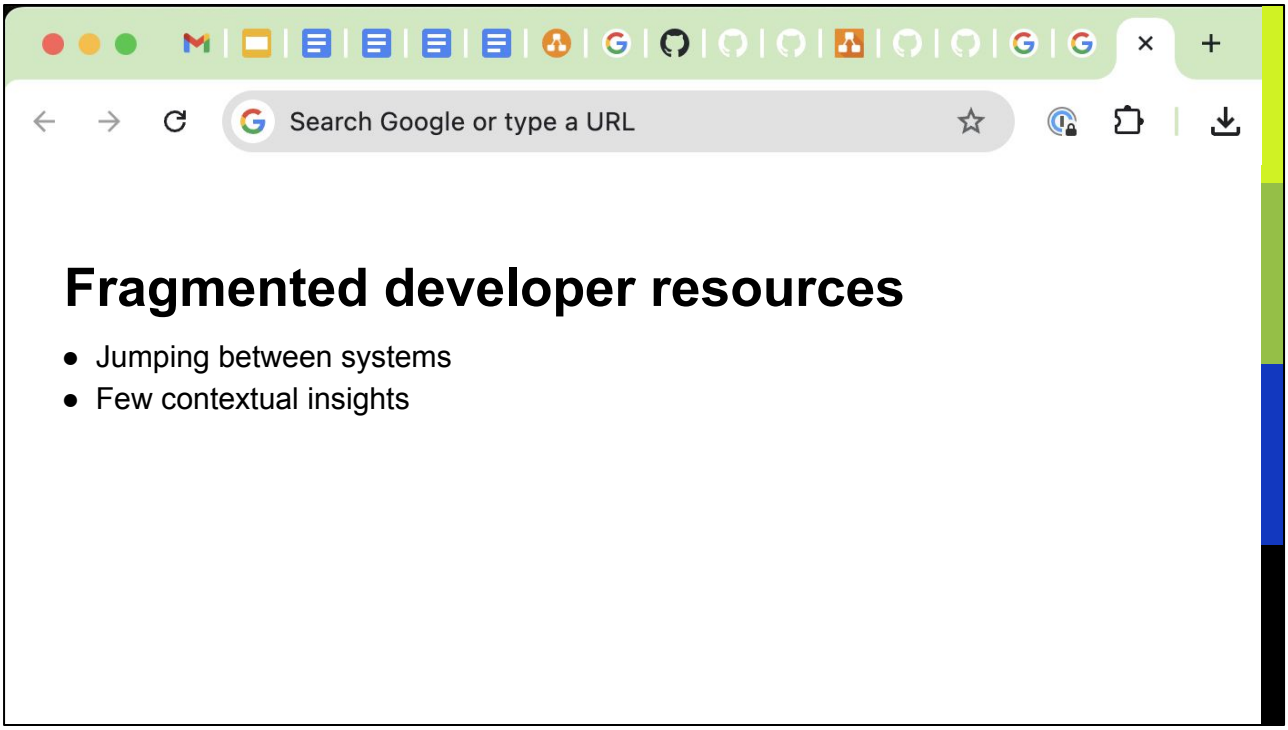
It might be because:

- the dev has 999 other things to worry about
- or they simply forgot it even existed

The system we built automates the assignment process, which is possible because ownership of tables and queries have been properly defined and easily retrievable. This leaves no room for issues being mistakenly assigned to the wrong owner.

We also have buy-ins from our engineering leadership. It has been a company focus to prioritise slow queries, as they are a major source of problems. The leaders have regular check-ins to assess open slow query items for their teams. Devs would then have an easier time deciding which tasks to prioritise.

After sending reminders to devs, if the action items stay unaddressed until it's overdue, as a last resort mechanism, PRs of the team / component would start to get blocked until they are addressed. This is a way to encourage accountability from these issues being overdue.

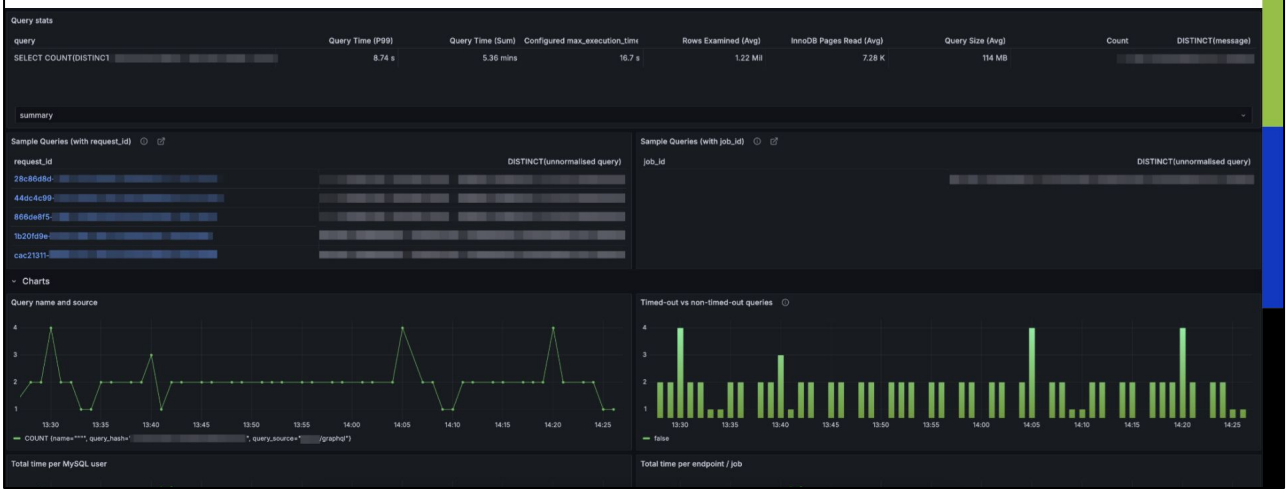


For the last gap, developer resources are fragmented across systems, leading to devs needing to jump between them during debugging, like we have different systems for logs, traces, metrics, and so on. The systems have different UI and UX, possibly with their own query language, making them hard to work with in tandem.

The dev also had few contextual insights. They needed to analyze the observability data on their own

Fragmented resources

Response 1: Unified Grafana dashboard

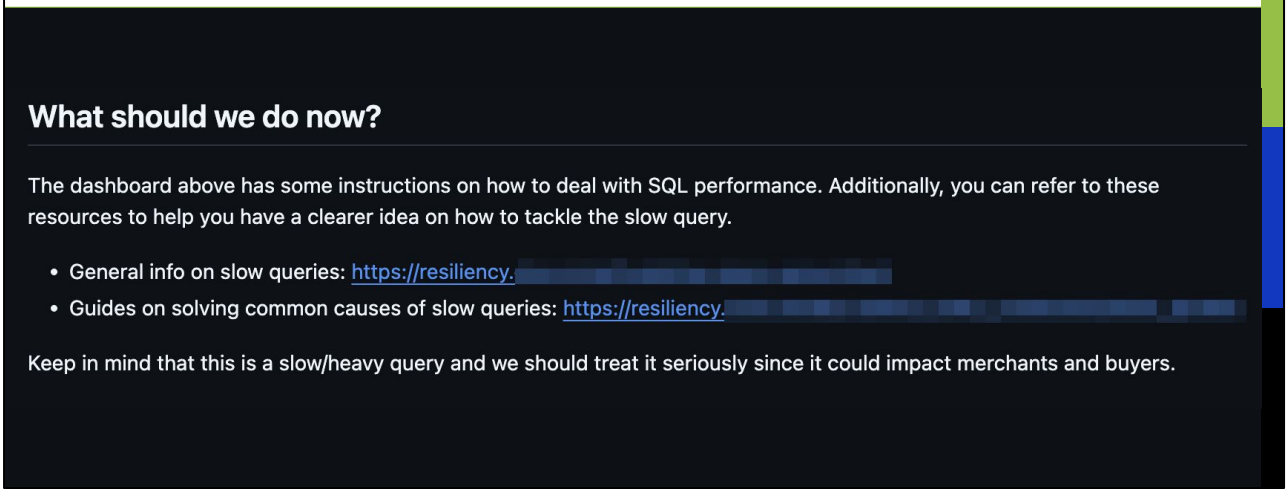


To address this, we built a

- Unified Grafana dashboard with important info, such as stats, source, links to logs, traces on top of Observe. This is only possible because all observability data source is in Observe and it allows us to build a custom dashboard using multiple data sources.
- As you can see in this view, they're able to first of all see the bad query
- The stats of the query itself. In terms of the P 99, the average amount of data that the query fetched, and so on,
- Links to samples of query runs are also available. So in each sample, they're able to see what exactly went wrong during that query run, how much time was spent in the database, in the application layer, and other things, so that they're able to make a better judgment on how to resolve that.
- Lastly, we also provide relevant charts that provide data over time, such as how many queries timed out, the query source, like it came from a particular job or a controller.
- All of this contextual data and helpful links are in one place.

Fragmented resources

Response 2: Action items have contextual insights, advice, and links



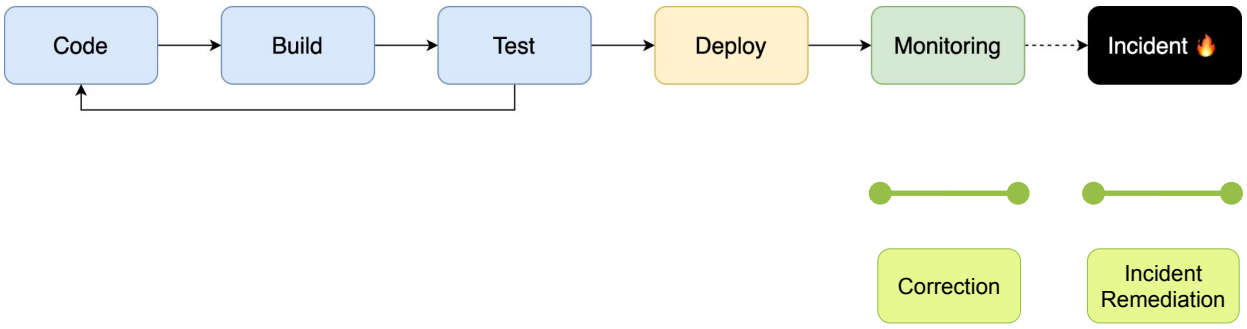
- We also give contextual insights outright in the action item description, like the query source, controller name.
- The dev has a better idea going through the action item before even opening the dashboard on what data they really need to look at to address the issue.
- We also have advice on common bad patterns that we see across the org and ways to address them. All around, we aim to make the developer experience better.

Closing the gaps - TL;DR

Dependency on incidents	**Proactively detect slow queries and scale the correction process via automation**
Lack of ownership and accountability	**Open action items automatically and block shipping if overdue**
Fragmented developer resources	**Unified system to show important contextual information**

To summarise the gaps we just discussed, and how we address them.

Development lifecycle (3)



On our dev lifecycle chart, the corrective measures would be in the “Monitoring” phase. We have shifted left successfully, but can we shift-left further and do something earlier in the dev lifecycle? For that, I pass off the time back to Brad to talk more about the prevention side.

shopify

Prevention

Avoiding problems caused by slow queries

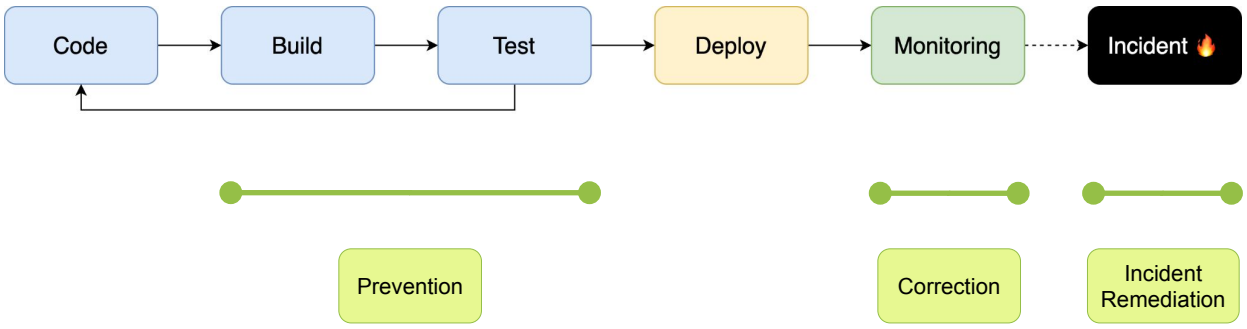
Speaker

Brad Feehan

Senior Site Reliability Engineer

Thanks! Let's talk about **preventing the typical problems associated** with slow queries.

Development lifecycle (3)



The **core of our work on prevention** is about developer workflows,

- before slow queries are deployed in the first place.

We wanted to highlight issues earlier.

- Because the further we can **shift left**
- the further we **reduce feedback** loops,
- and **distribute work** back to developers
- **preserving context**.

Catch slow queries while testing in CI

without impacting performance

Detect new queries

- Register ActiveRecord subscriber
- Save queries encountered in test

Criteria to identify problematic queries

Auto-EXPLAIN to analyze performance

- Detect full table scan

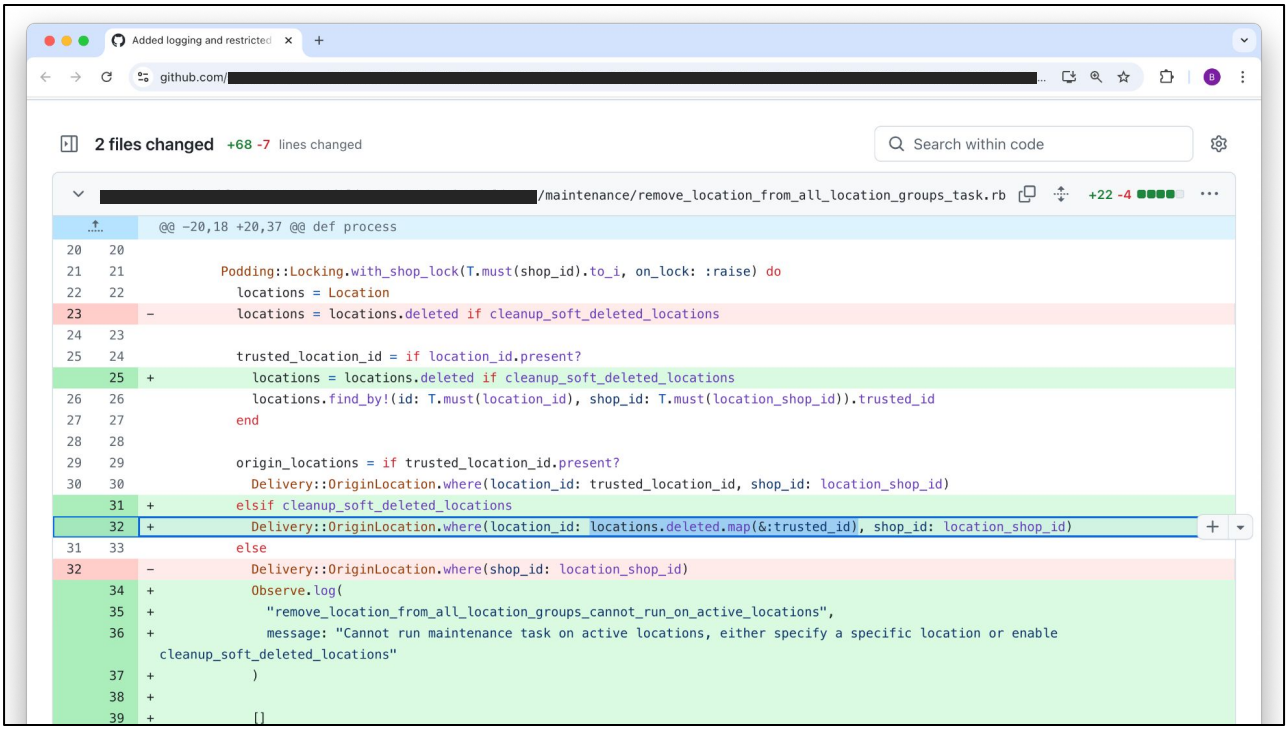
Actions taken when detected

- Warning comment with evidence
- Contextual advice and links to tools

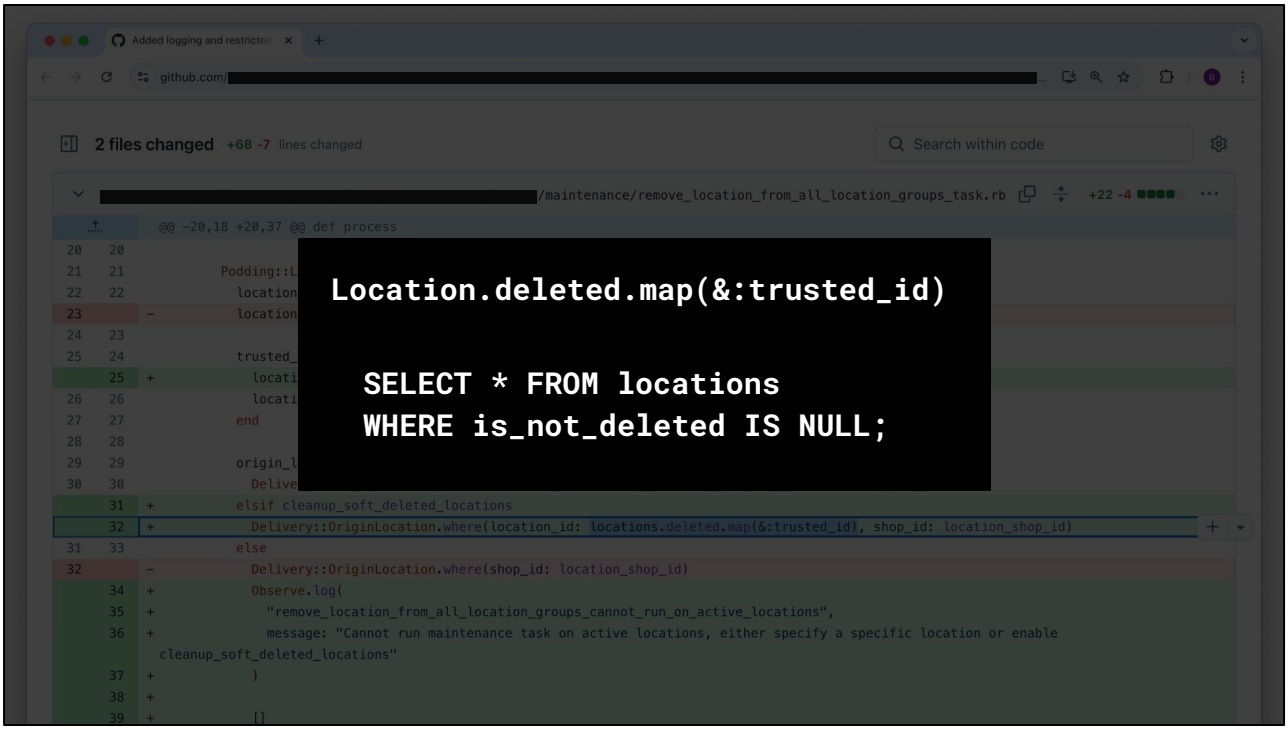
So we built a system to **catch** slow queries in CI, **before they get deployed** at all.

- It **detects queries encountered** during testing
- Runs EXPLAIN, and **currently detects** full table scans.
- When a new slow query is detected, it triggers a warning comment with useful advice.

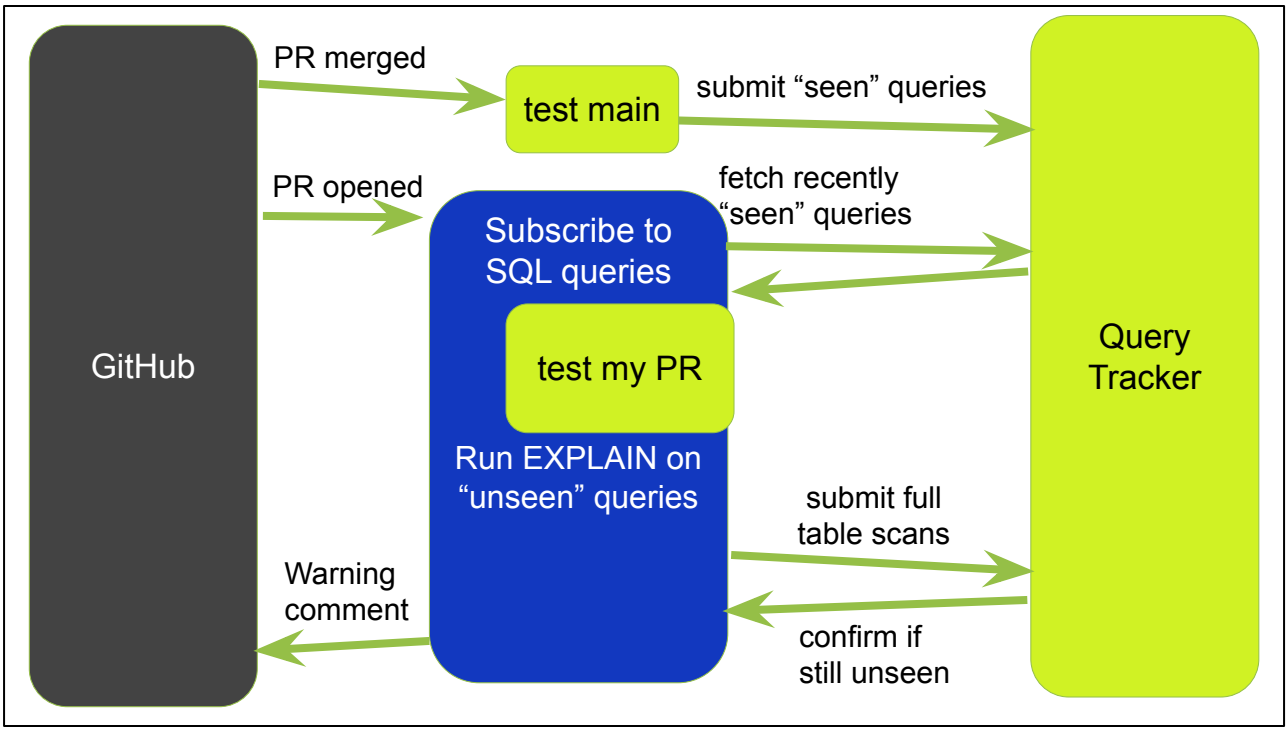
I'll show an example.



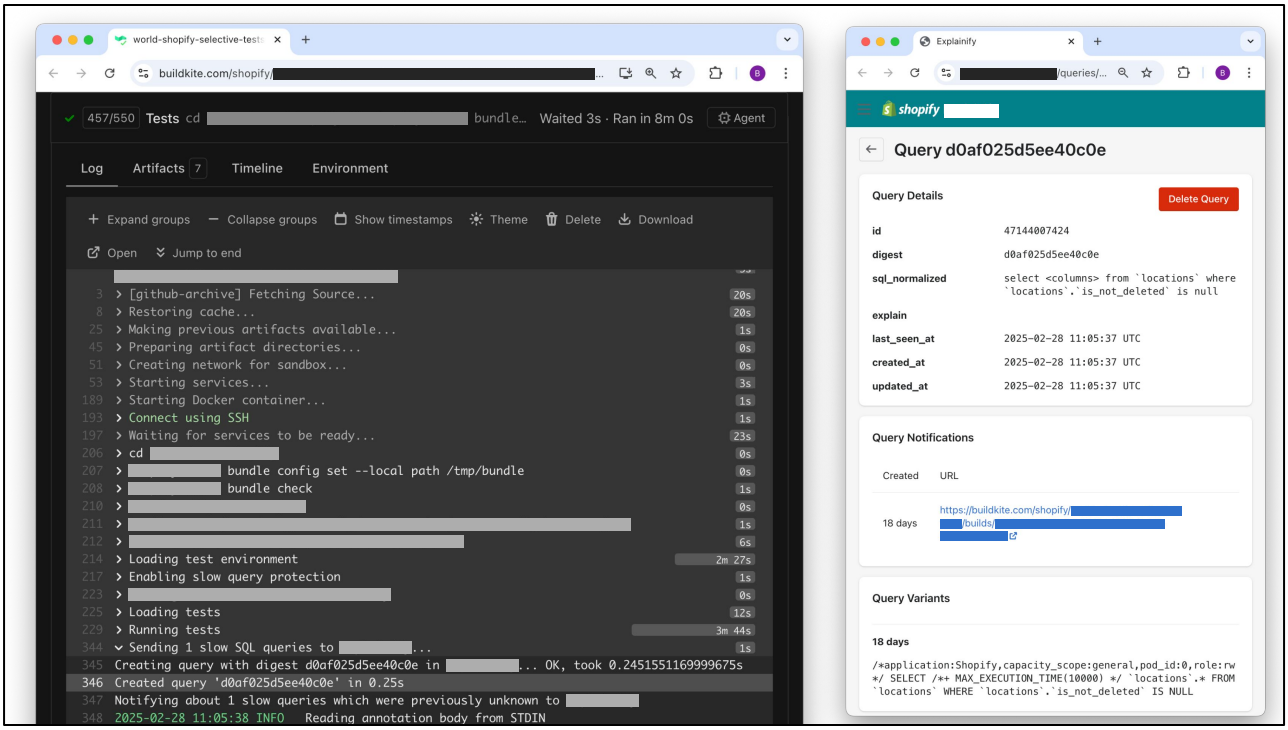
A few weeks ago, a developer was working on a new maintenance task.



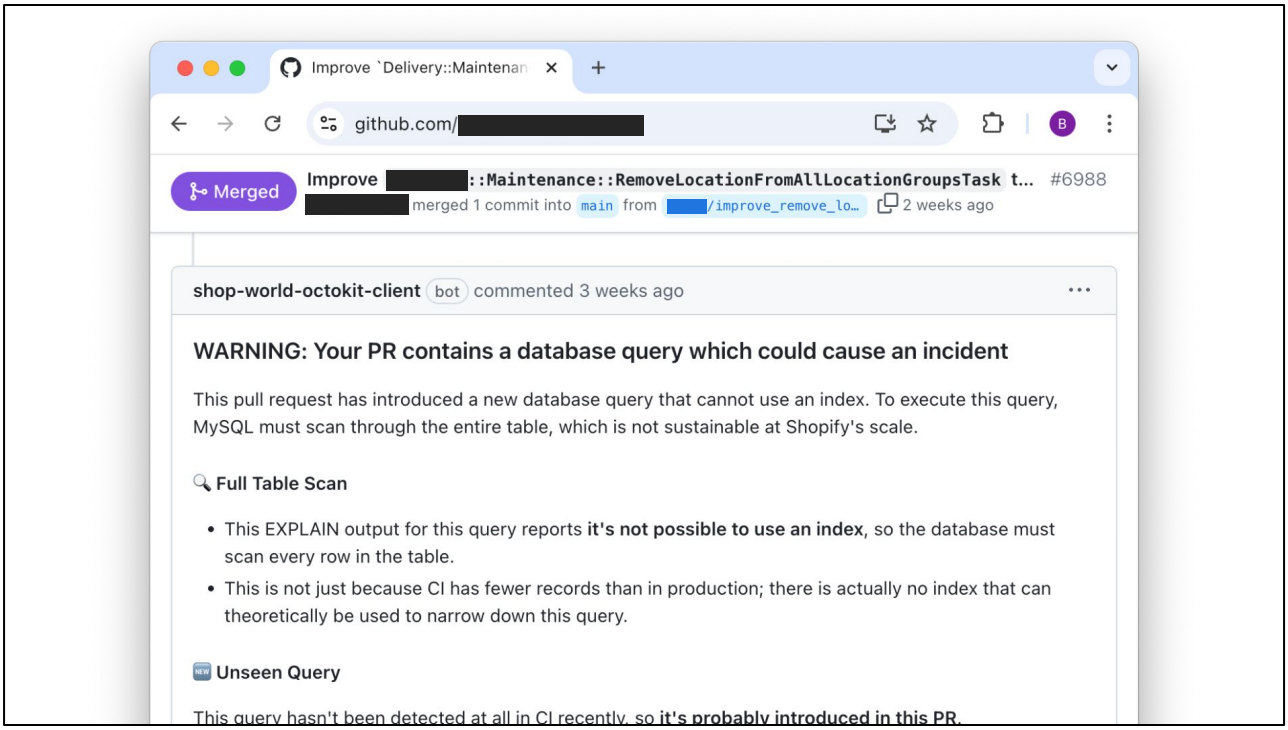
I wanna draw your attention to the highlighted line.



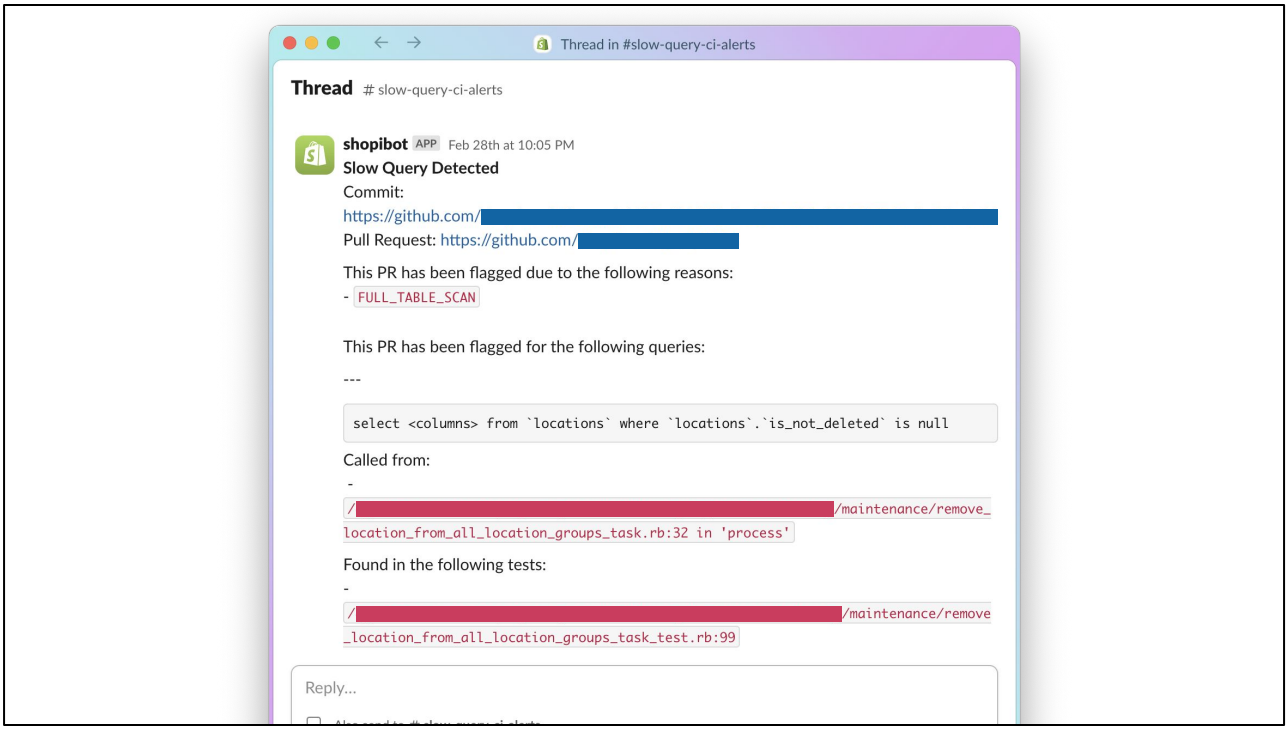
1. We use GitHub to host our code.
2. When devs ship a feature to main, we run our full test suite.
3. We catch all queries that occur, and send them to an internal service
    - This will track the state of queries in the `main` branch.
4. When the next PR is opened, we test the changes in CI.
5. Before it starts running tests, it downloads “seen” queries from the Query Tracker.
    - These will be ignored while scanning the PR, as they’re already in `main`.
6. Then it sets up a subscriber, listening to all database queries
7. After the test suite is complete, it runs EXPLAIN against all the newly found queries.
8. Any full table scans found are sent to the Query Tracker.
9. If still unseen in the Query Tracker, the CI runner posts a comment to the PR on GitHub.



This is the result of CI, a slow query is found and logged to the Query Tracker.

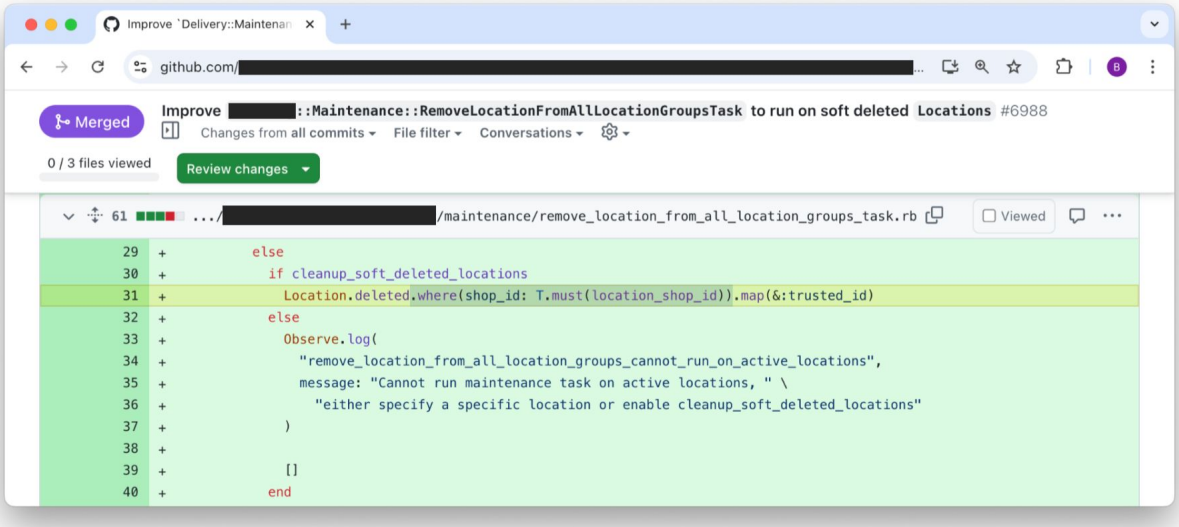


Here’s the comment added to the GitHub PR.
It explains the situation and gives some contextual advice, including the example stack trace.



Our team also gets a notification in a dedicated Slack channel.
So we can keep an eye on the performance of the system.

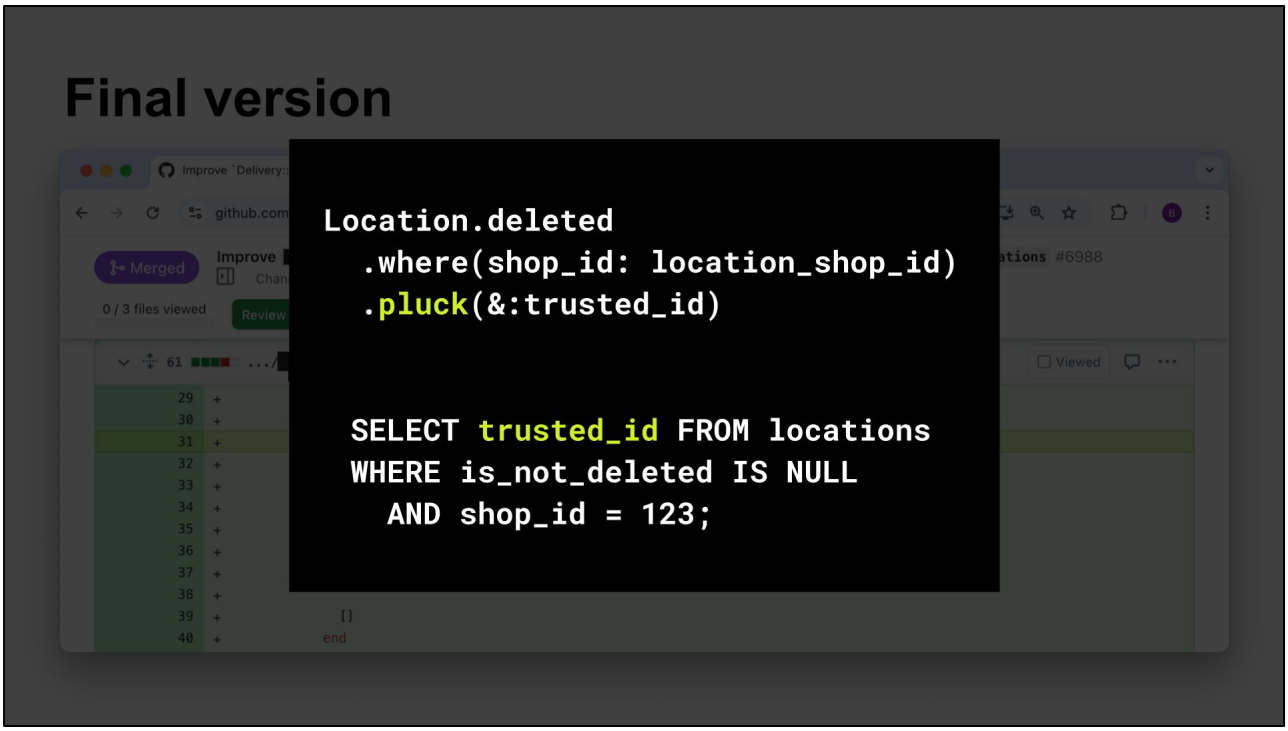
Final version



So what was the developer's response?

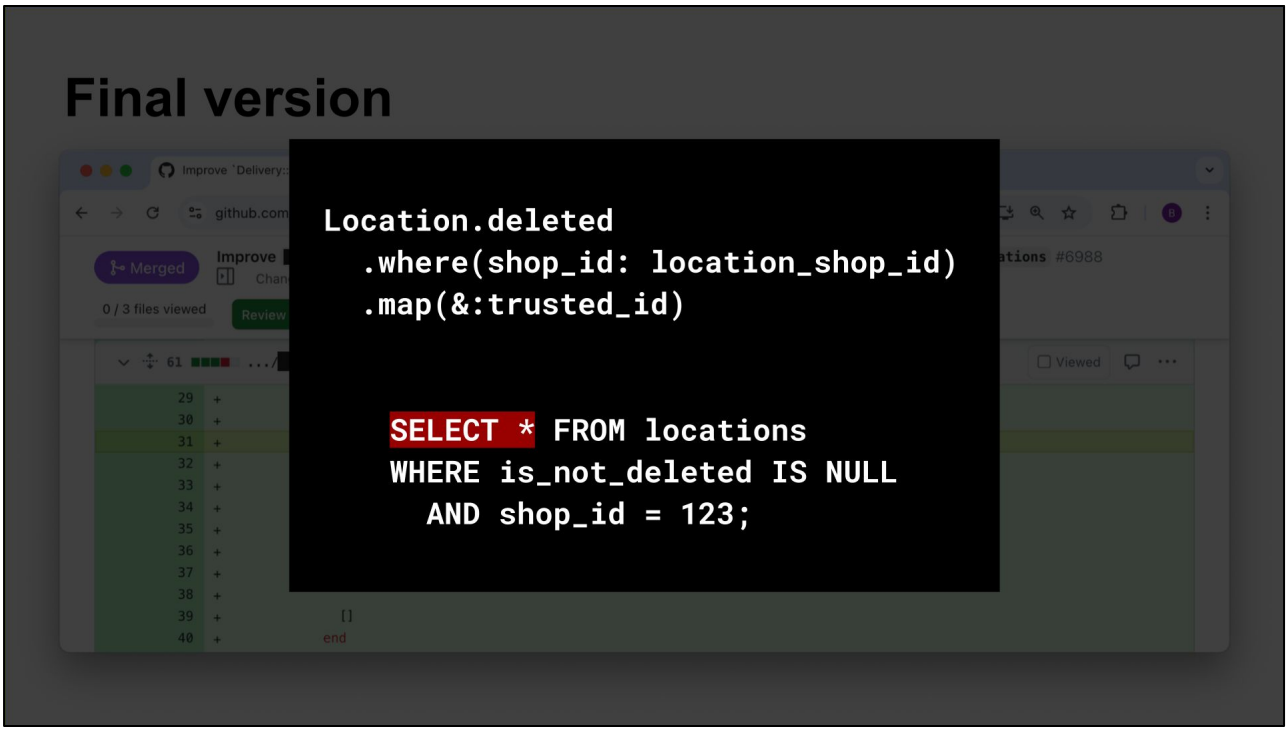
Here's the final diff of the PR.

Location.deleted.where(shop_id: location_shop_id).map(&:trusted_id)

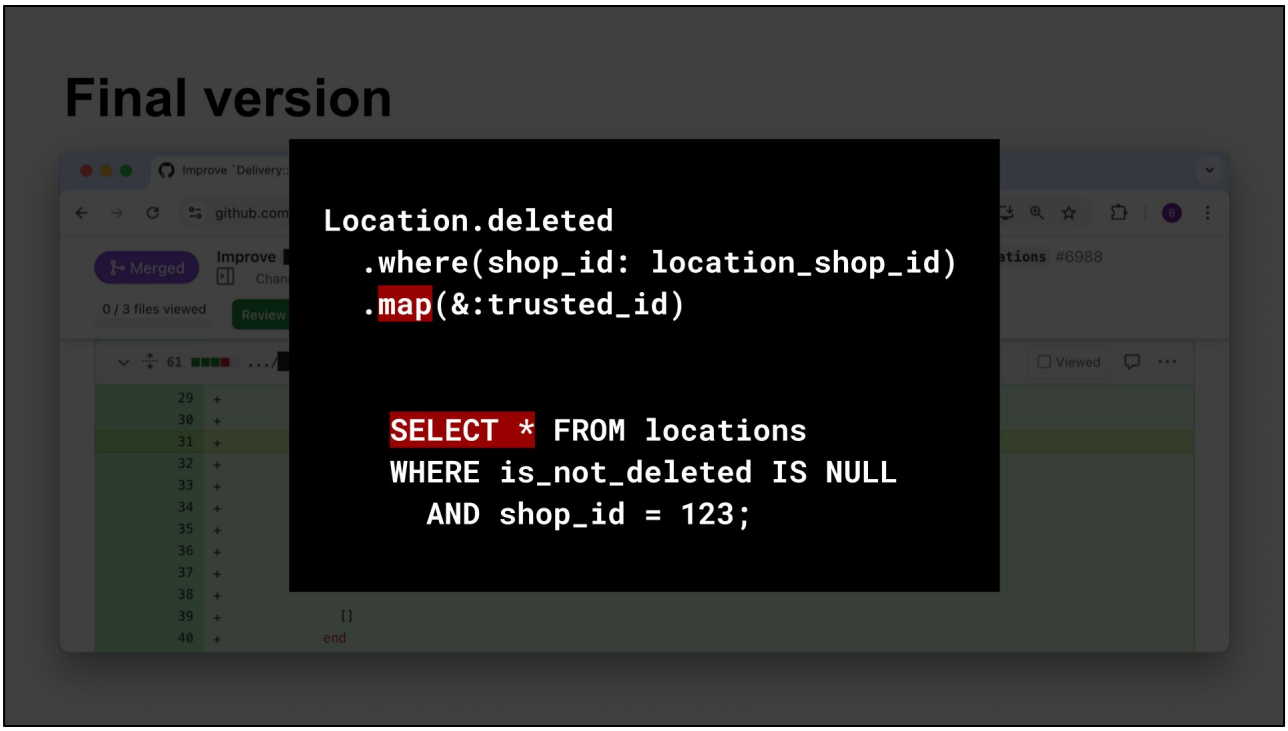


They ended up including a missing `shop_id` scope.
All sharded tables at Shopify have this, and queries should generally add a `WHERE shop_id` clause.

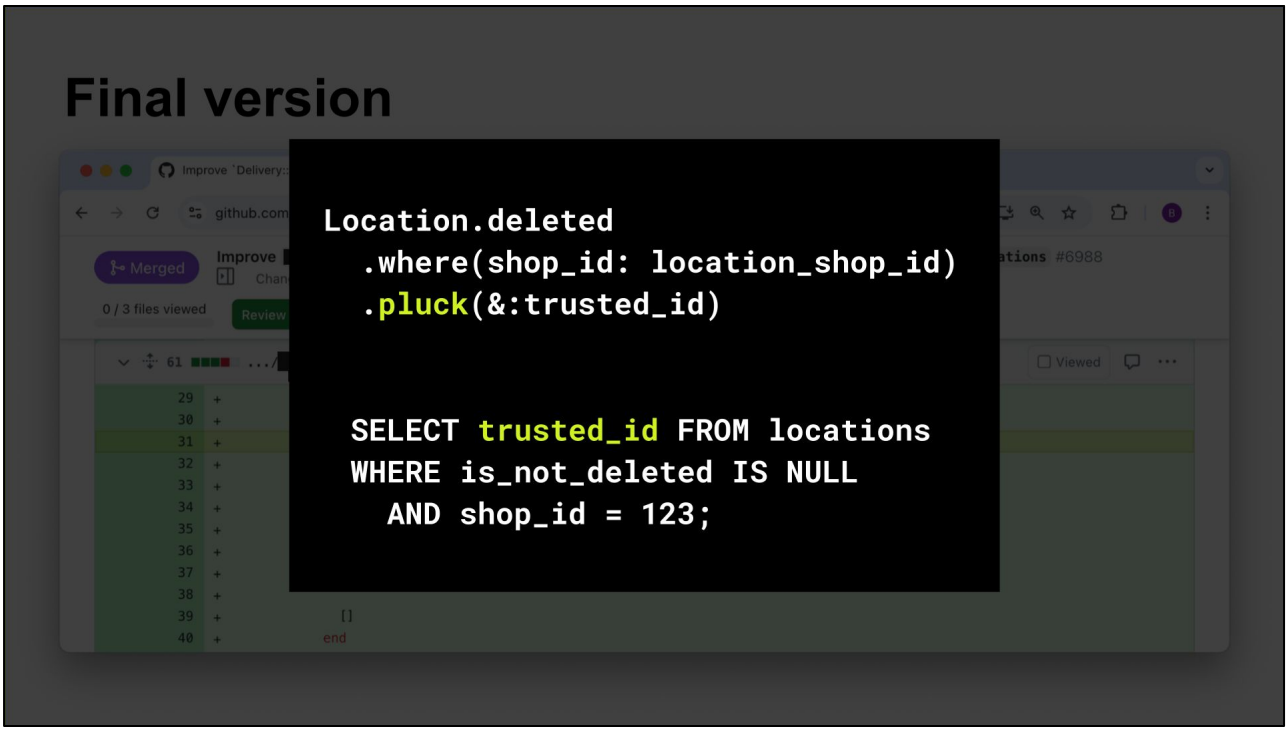
Does anybody see another issue with the query?



This retrieves all columns from the DB to the app
and then throws most of that data away, by retrieving only one column value.

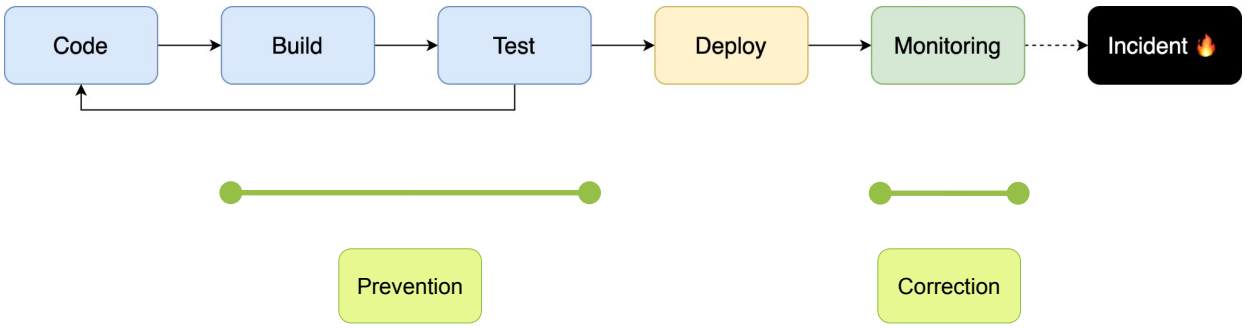


That's because `.map` is part of Ruby, it converts to an array first.



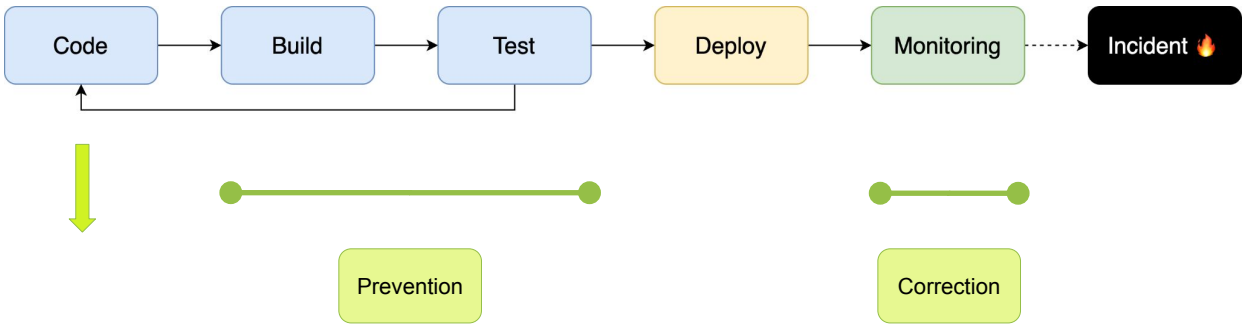
We could improve this by using `pluck` which is from Rails instead, and knows how to modify the query accordingly, to fetch only the data it needs.

Development lifecycle (3)



So that's how we integrated slow query prevention into the test phase of the development process.

Development lifecycle (3)



Having covered how we built something that will catch slow queries in Build/Test and Monitoring phases, as we “shift left” with testing of slow queries. But we were wondering, can we actually detect slow queries even sooner, like during the coding process?

We would love to continue, and surface visibility of query metrics in the development environment, either an IDE plugin or similar to inspect new queries before they’re committed.

Actionables

Here are some actions or homework you can take back to your team after the talk.

I'll admit, some of these are not so much **specific directions.**
More like **philosophical takeaways** and **things to keep in mind.**
but, nonetheless…

Actionables

- Shift left and **reduce feedback loops**
- Assess resiliency at every stage of dev lifecycle
- Get creative to reduce risks and toil
  - How to make it **easier for devs** to do the right thing
  - How to **increase visibility** of inefficient resource usage
- Remain pragmatic
  - Look for **quick wins** (e.g. sensible timeouts)
  - Start with queries using most capacity (**data-driven**)
  - Focus efforts on scalable solutions
- **Preserve precious context** and automate
- Delegate work and distribute effort

1. **Reduce feedback loops**
   - Consider how best to implement “shift-left” testing for database queries
   - Reduce wasted effort:
     - downtime itself, incident response
     - re-gathering context lost during development
     - then the fix needs implementing, testing and releasing
     - catch in dev to reduce wasted CI cycles
2. Assess **resiliency of platform and processes**, at every stage of dev lifecycle
   - Build light-weight tools and automations to surface insights and visibility/observability
   - Could be based on metrics you might even already be collecting, but not seeing
     - e.g. enable slow query logs, you may be surprised
3. **Get creative** with automations, warnings, enforcement, etc.
   - What would make life easier for devs to get it right? Human error is inevitable.
   - How to incentivize efficient resource usage? Block deploys?
   - Reduce toil!
4. Be pragmatic
   - Prioritise **queries using most capacity** (data-driven)
   - Prioritise **efforts based on RoI**, quick wins, etc. – for example, timeouts
   - Focus effort on scalable solutions (automation, delegation + distributing effort to teams/owners)

Thank you