Ten useful ideas to scale up GraphQL

and serve millions of requests per day

Harsha Reddy

- Senior Software Engineer
- Internal API Engineering @ Wayfair

GraphQL@Wayfair

Some statistics 📣

- **~160 Million** GraphQL queries a day
- **~50%** of our Storefront traffic is GraphQL
- **10** active Graphs in various applications
- **Hundreds** of commits a day updating the graphs

1- Domain Driven Design

an exercise in saving your graph 😎

Three core principles

- Focus on the core domain concepts and their relationships
- Base complex designs on models of the domain
- Constantly collaborate with domain experts, in order to improve the application model and resolve any emerging domain-related issues

My fav mantras

- Design before your write a single line of code.
- Clear and concise use cases. Describe your data domain well.
- GraphQL schema is not a 1:1 with your DB and View layer.
  - Avoid direct use of columns from DB e.g. articleSurveyQuestionID, ugh
  - Avoid view layer concerns e.g. shouldShowHeader, ugh'ier
- Don't be afraid of multiple revisions to a design.

2 – Stop Breaking changes

If I had a dollar everytime someone broke an API contract 😭

Some hard truths

- we don't/can't version GraphQL schema
- don't do Schema Change requests due to volume of daily commits
- we cannot block developers too long, even a few minutes is bad

So how did we stop breaking changes to the schema 🤔?

Enter deprecationReason, the hero 🦸

Stop breaking changes

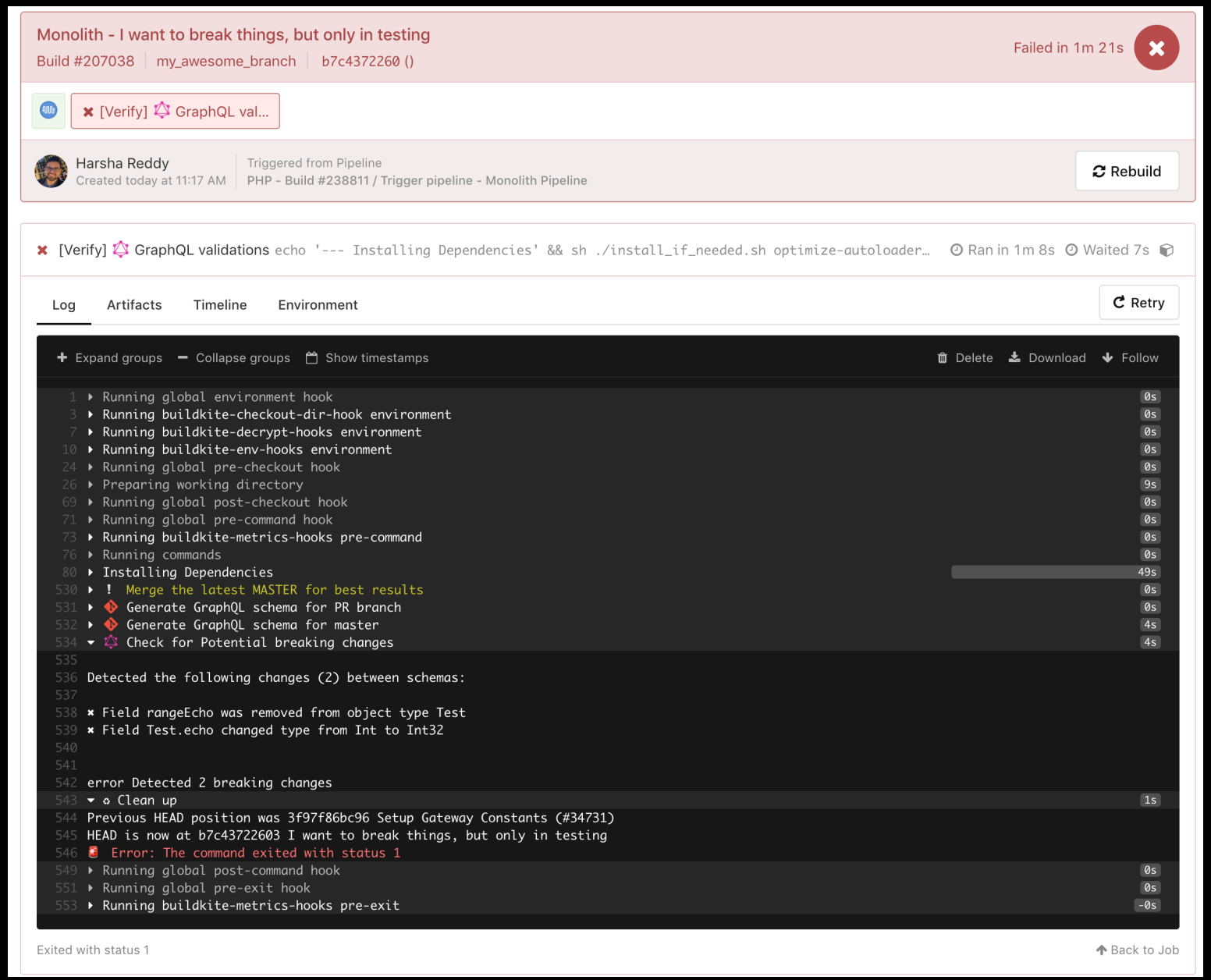
- Block any and all changes that modify and delete fields or types.
- Adding fields or types is allowed.
- Block any changes that render a schema ungenerateable

How does one delete/modify fields

- Add a deprecationReason to the field or type
- Next, Add Internal logging when these are queried by clients
  - this means queries are still using the deprecated field.
- Delete field when all queries have moved on.

How does it look?

We integrated GraphQL Inspector into Buildkite to run a validation check on Pull Requests.



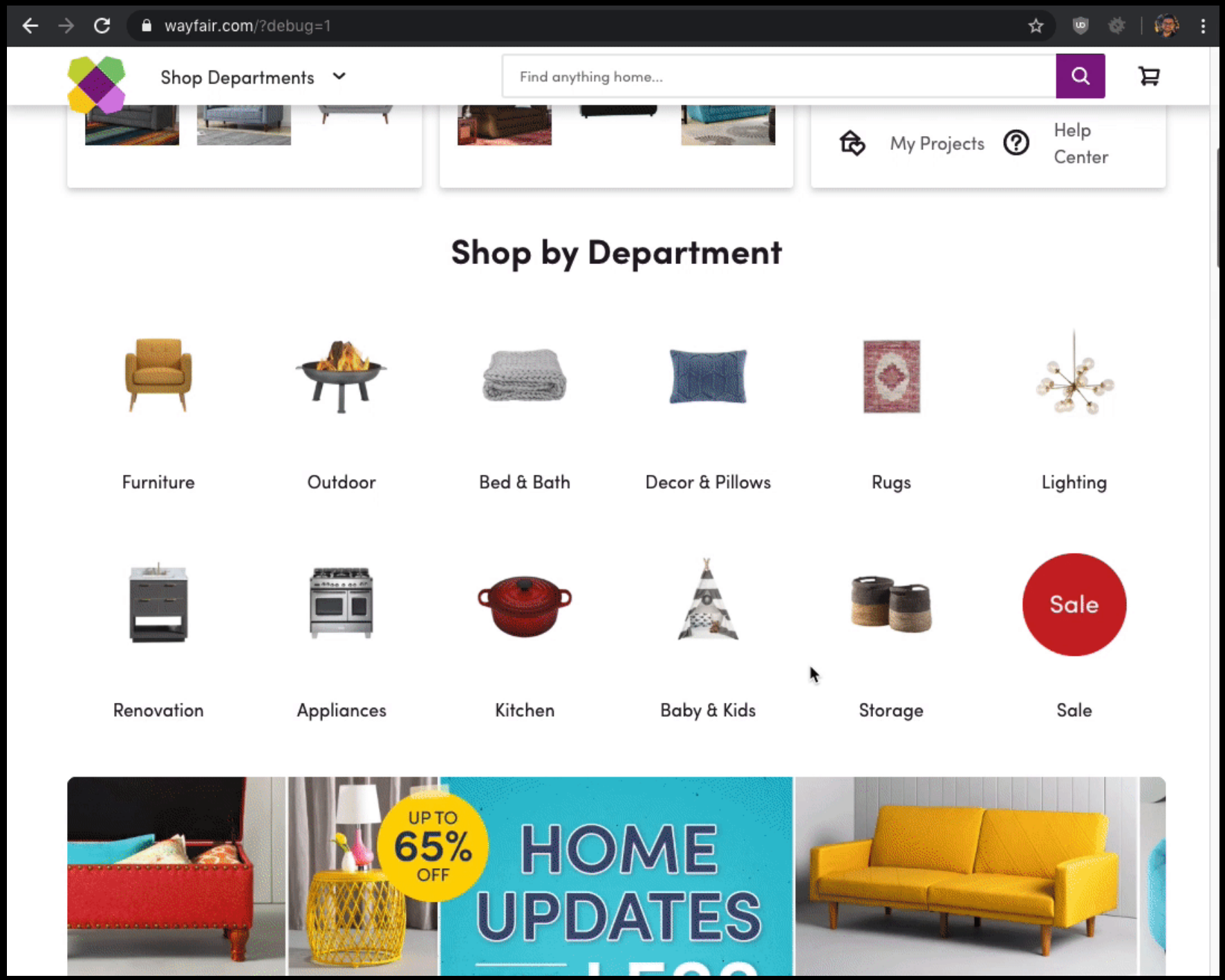
3 – Performance Monitoring

Is GraphQL slow 🐢? How dare you!!

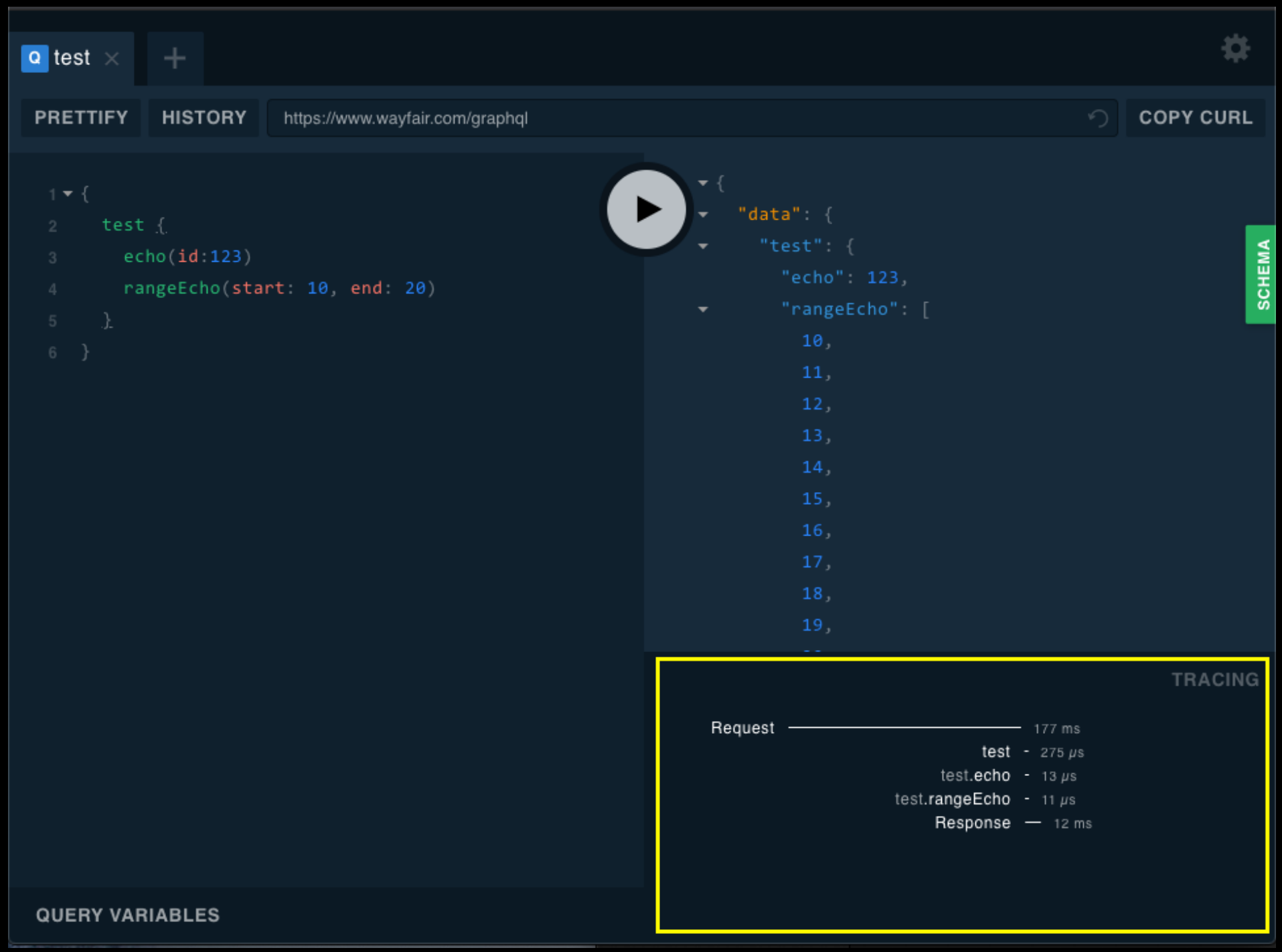
Give them the power of speed

- Utilize Apollo Tracing to record query Performance
- We reverse engineered the above into all our applications as a GraphQL middleware.
- Show this info to engineers in an easy and digestable way.

Show me the money?



Mo Money


test
PRETTIFY
HISTORY
https://www.wayfair.com/graphql
COPY CURL
{
test {
echo(id:123)
rangeEcho(start: 10, end: 20)
}
}
{
"data": {
"test": {
"echo": 123,
"rangeEcho": [
10,
11,
12,
13,
14,
15,
16,
17,
18,
19,
SCHEMA
TRACING
Request 177 ms
test - 275 µs
test.echo - 13 µs
test.rangeEcho - 11 µs
Response 12 ms
QUERY VARIABLES

4 – Error Handling

Catch the sneaky ones

Our rules

- **Observability** is a primary focus
  - logs MUST identify the failing query
  - logs MUST capture the resolver that caught/logged the exception
  - logs MUST record a stack trace that has reference to business logic code run along with GraphQL code
  - logs MUST tune severity correct, not everything needs to be a red alert Error.
- Logs are **THROTTLED** when needed to avoid blowing up infrastructure.
- Differentiate between **User Generated** and **System errors**.

I'll avoid going into too much detail on this. It's a session on its own.

Show me something unique Wayfair does

Our response codes might be interesting

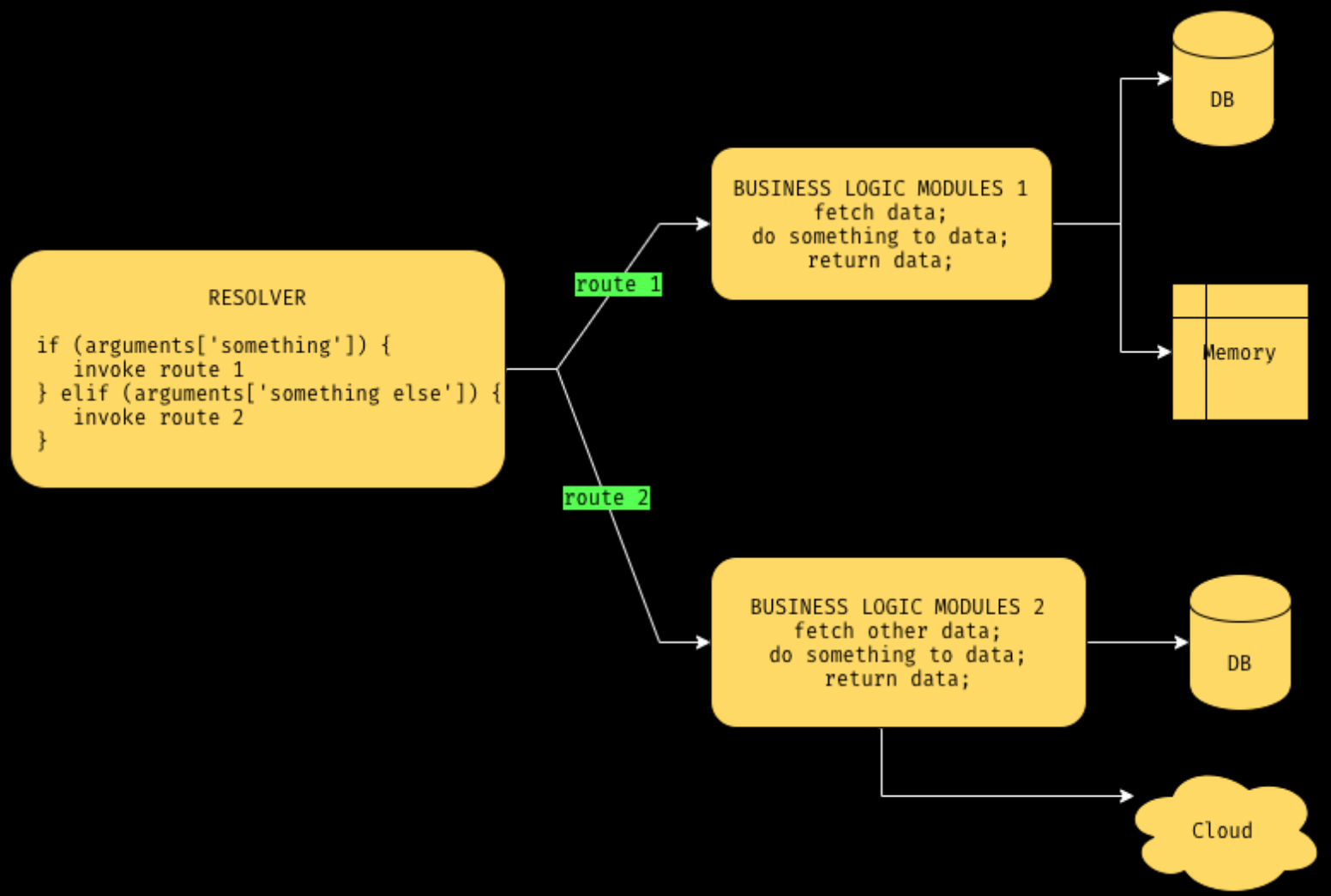
	Response Code
A GraphQL query without any errors	**200 OK**
Partial query success with any errors	**207 MULTI-STATUS**
A complete query failure	**500 Internal Server Error**
A code change the resulted in schema/framework failure	**503 Service Unavailable**
A query request from client that we don't recognize	**400 Bad Request**

5 – Simple Resolvers

How I stopped worrying and learned to love dumb resolvers

Avoid business logic in resolvers

- Resolvers should act as routes to the business logic
- Bad arguments to resolvers result in a fast fail
- Valid arguments help in directing to the right business logic



Reasons

- Sharing business logic modules between REST, GraphQL. Yes, that is more common that you think.
- Avoid fracturing business logic between resolvers and other places in code like models.
- Avoiding Fractured unit tests. It is not fun to write unit tests twice.
- Helps keep the schema relatively clean, devoid of any major issues.
- Easy to change business logic by just changing the route.

and more 😬.

6 – Persisted Queries

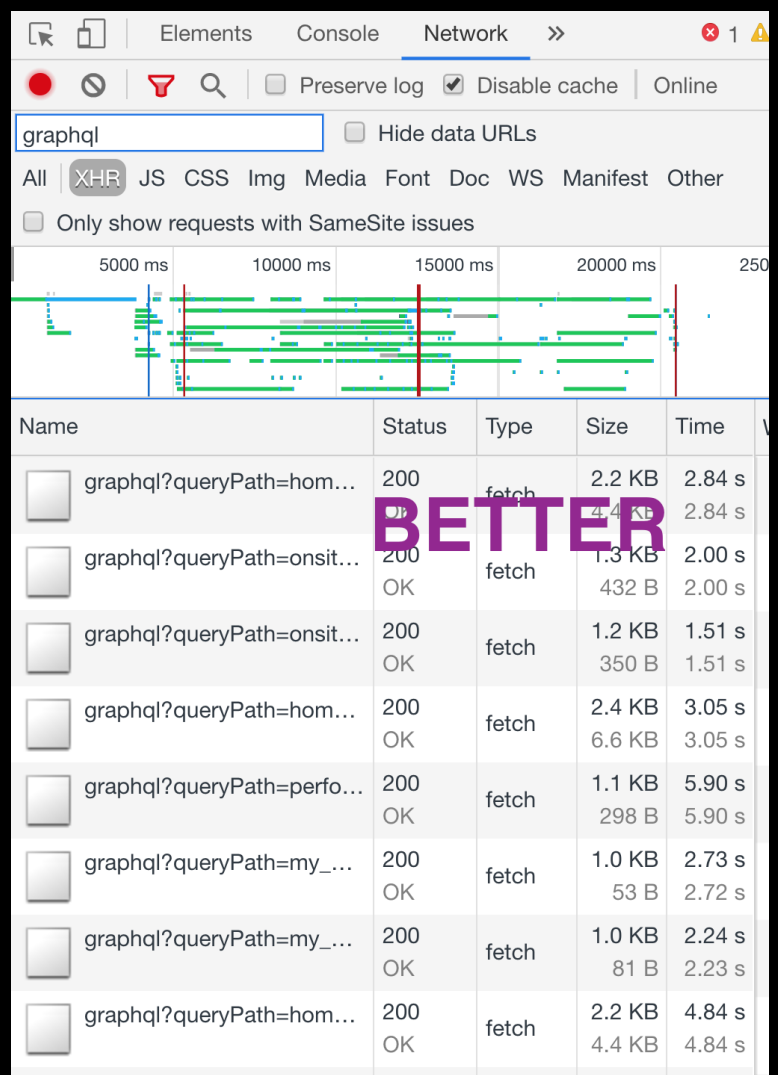
They are freaking awesome

Info

- Perisisted query is a fancy term for clients sending a unique query identifier instead of the full GraphQL query over the wire.
- If you are in the Apollo ecosystem, I highly recommend you opt into Persisted queries.
- Even outside that ecosystem, it's easy enough to generate and persist queries. Here's a simple way to go about it
  - A persisted query ID can be the md5 of the query contents in an application
  - Persist that to storage like a file or DB with a caching layer in front.
  - Load this data with all queries into memory. (Refer to Read Through Cache pattern for this)
  - Any GraphQL request will use that to lookup the query given an ID.
  - When code is deployed to production, re-populate the persisted queries.
- At Wayfair, we use the above and add a few techniques on top to make it secure and robust.

How to enhance this?

- At Wayfair Persisted query ids permeate all of our ecosystem from logs, performance monitoring, time series etc.
- All GraphQL requests hit a single endpoint
  - We append the persisted query id to the url to help with observability.



7 - Custom Validation rules

Rules that run before query gets executed

Example

- Custom Rules can run on field, type, directive etc.
- Check out the default rules here Validation Rules GQL JS
- A good example is a custom deprecation rule.
  - Log any usage of deprecated fields
  - Keep a dashboard of the usages
  - Teams actively use this to update their queries
  - Once teams are happy a deprecated field is no longer used, it is deleted.

8 – APM integration

Application Performance Monitoring

Distributed Tracing and GraphQL

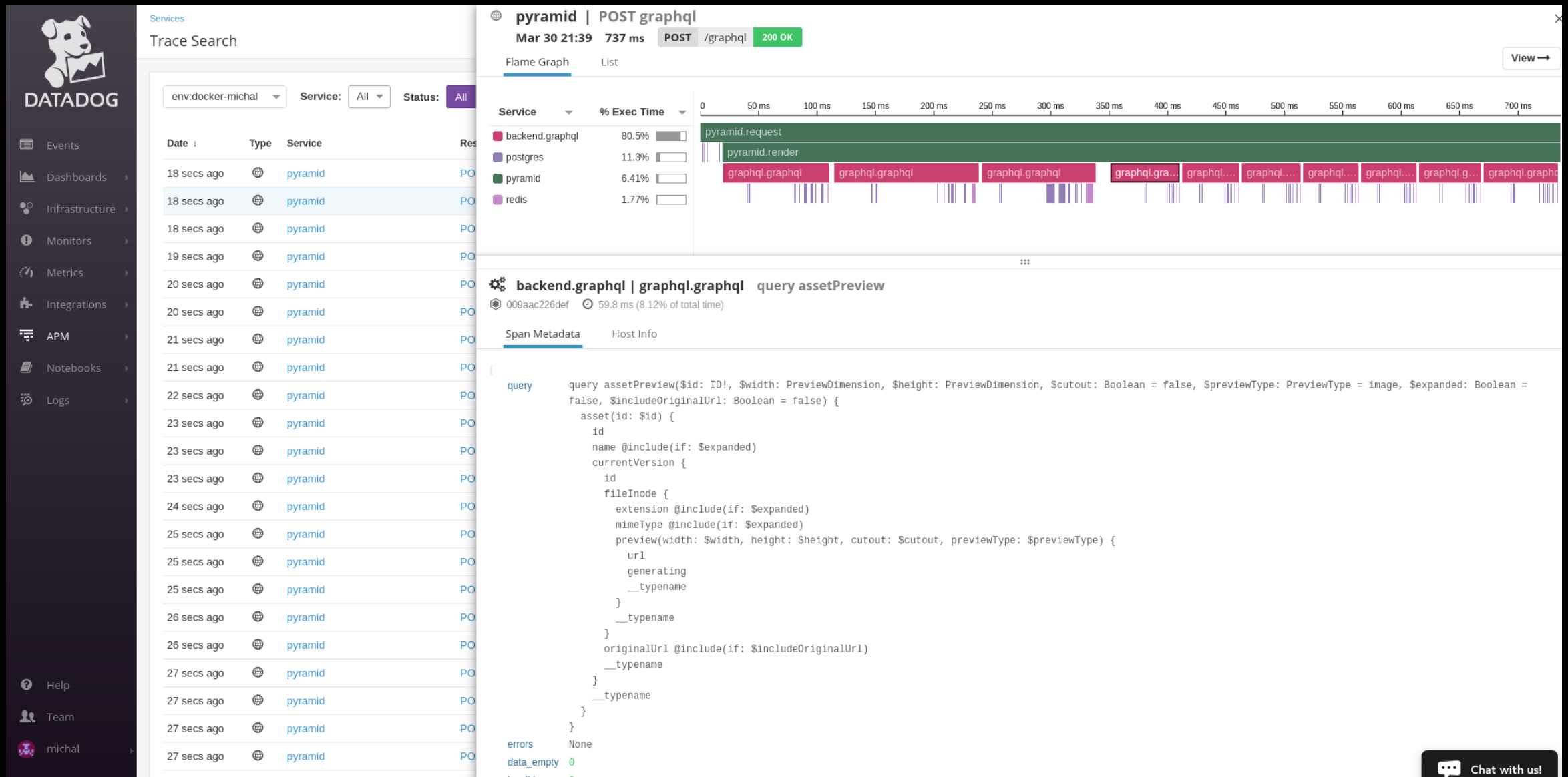
- Distributed (Open) Tracing, also called distributed request tracing, is a method used to profile and monitor applications, especially those built using a microservices architecture.
- Distributed tracing helps pinpoint where failures occur and what causes poor performance.
- Services like Datadog can help monitor traces effectively by utlizing distributed tracing.

Using this at scale to help monitor GraphQL query performance and perform deep dives is magical.

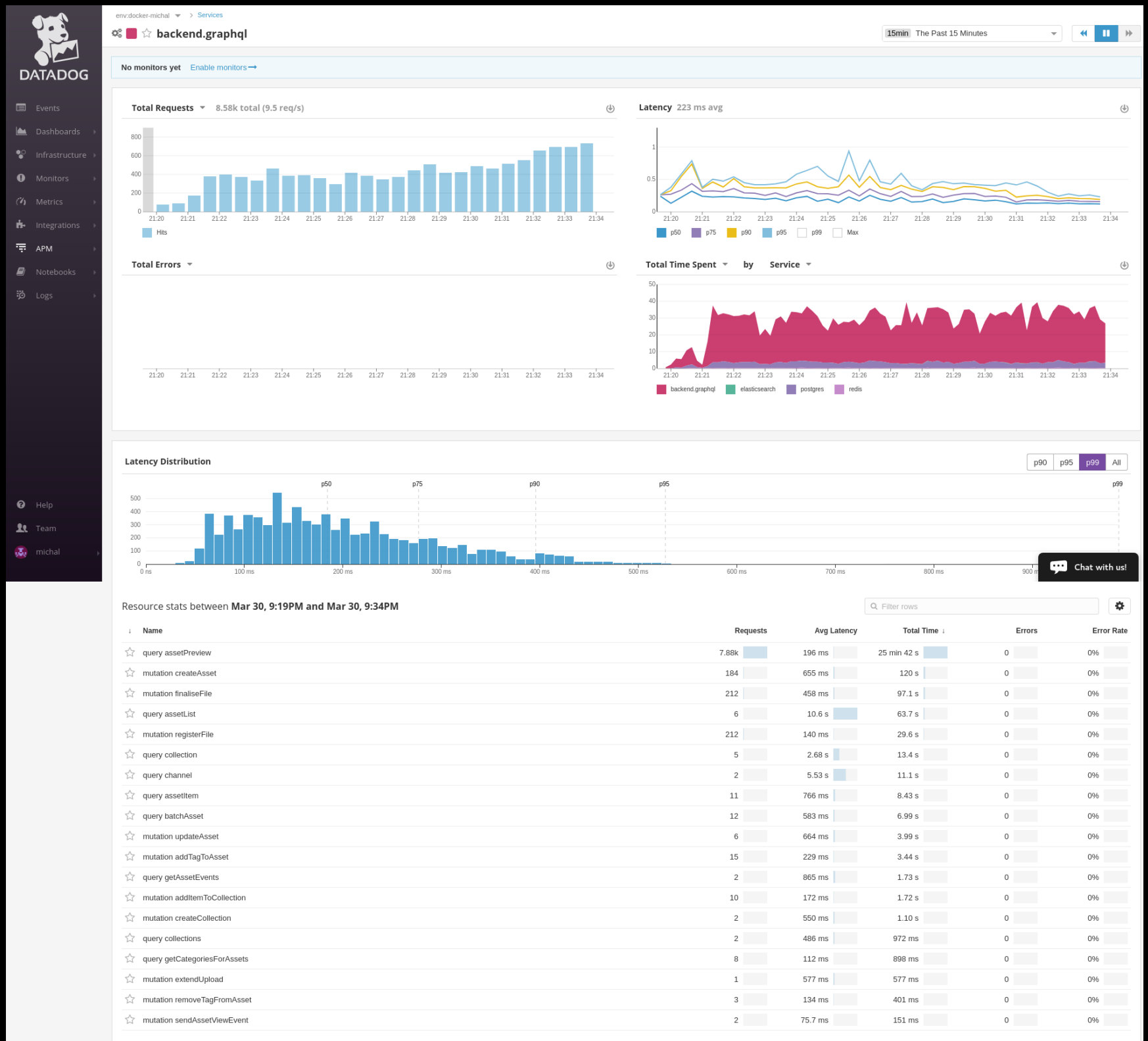
Examples

[DDTrace GraphQL in Python]

[Datadog APM docs]



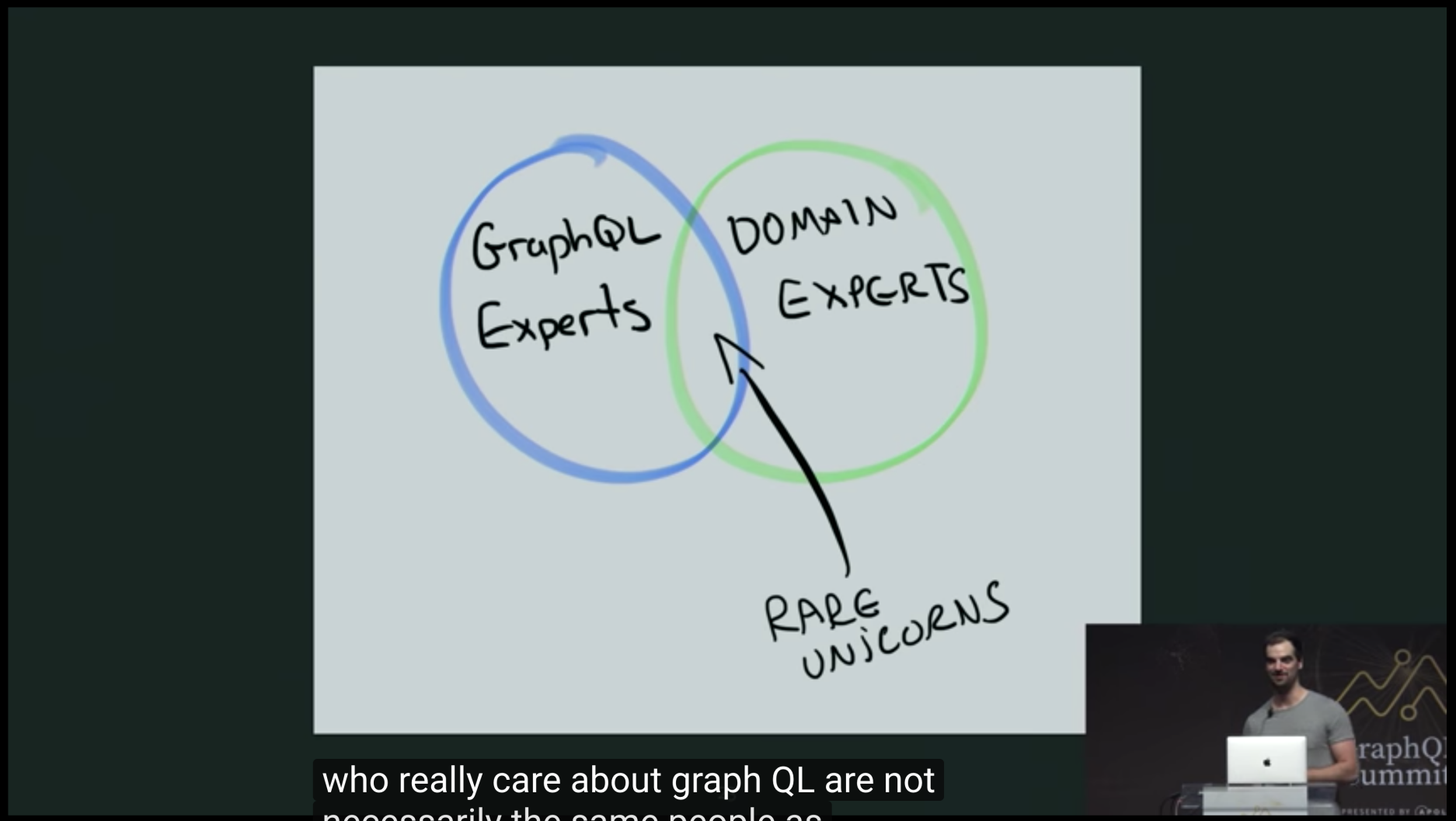
Examples (cont.)



Name	Requests	Avg Latency	Total Time	Errors	Error Rate
query assetPreview	7.88k	196 ms	25 min 42 s	0	0%
mutation createAsset	184	655 ms	120 s	0	0%
mutation finaliseFile	212	458 ms	97.1 s	0	0%
query assetList	6	10.6 s	63.7 s	0	0%
mutation registerFile	212	140 ms	29.6 s	0	0%
query collection	5	2.68 s	13.4 s	0	0%
query channel	2	5.53 s	11.1 s	0	0%
query assetItem	11	766 ms	8.43 s	0	0%
query batchAsset	12	583 ms	6.99 s	0	0%
mutation updateAsset	6	664 ms	3.99 s	0	0%
mutation addTagToAsset	15	229 ms	3.44 s	0	0%
query getAssetEvents	2	865 ms	1.73 s	0	0%
mutation addItemToCollection	10	172 ms	1.72 s	0	0%
mutation createCollection	2	550 ms	1.10 s	0	0%
query collections	2	486 ms	972 ms	0	0%
query getCategoriesForAssets	8	112 ms	898 ms	0	0%
mutation extendUpload	1	577 ms	577 ms	0	0%
mutation removeTagFromAsset	3	134 ms	401 ms	0	0%
mutation sendAssetViewEvent	2	75.7 ms	151 ms	0	0%

9 – Inclusive community

- Empower teams to be experts
- They know their domain much better
- GraphQL Schema Design



who really care about graph QL are not
necessarily the same people as

Make Domain experts GraphQL Experts

- Developers have the natural enthusiasm to learn
- Provide an easy path for developers to become an expert in GraphQL.
- Incentivize and encourage them. Give them with the elusive GraphQL Expert title.
- Encourage responsibility and domain ownership amongst them.
- They up the knowledge of the entire team and expertise spreads.

(It's a Pyramid scheme .. shhhh)

Jokes aside, we have seen an incredible love for GraphQL at Wayfair. We provide a lot of docs and help to folks looking to become experts in GraphQL. It does take a while, but the heroes of the program are driving major wins across the company.

10 – GraphQL does not fit every scenario

There I said it! Watch your profanity 🤬

When Not to Use GraphQL

- **Authentication** Is usually handled by headers / hashing (stateless) or by specific service endpoints that then set up authenticated sessions or provide tokens for use with the API.☞
- **File Uploads** You could send a base64 string with a mutation, but large files result in large (and therefore unwieldy, slow to process) strings. A dedicated endpoint for uploads is more practical.
- **Dynamic Connections** This can happen with generic key/value pairs where the value amounts to a foreign key. The nature of those connections will vary from query to query. The bad smell should be that foreign key, since GraphQL is all about avoiding them in favor of connections.☞
- **RPC-Style Operations** Aren't there better solutions for this like gRPC. Prefer to avoid using mutations for such operations.

My Contact

Twitter twitter.com/stymied_sloth

LinkedIn www.linkedin.com/in/harshadeepreddy/

Github github.com/StymiedSloth

Fin.

Thank you.