

# 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 you 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 depreciationReason, 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 depreciationReason 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.

Monolith - I want to break things, but only in testing  
Build #207038 | my\_awesome\_branch | b7c4372260 ()

[Verify] GraphQL val... Failed in 1m 21s

Harsha Reddy Triggered from Pipeline  
Created today at 11:17 AM PHP - Build #238811 / Trigger pipeline - Monolith Pipeline

[Verify] GraphQL validations echo '--- Installing Dependencies' && sh ./install\_if\_needed.sh optimize-autoloader... Ran in 1m 8s Waited 7s

Log Artifacts Timeline Environment

+ Expand groups - Collapse groups Delete Download Follow

```
1 ► Running global environment hook
3 ► Running buildkite-checkout-dir-hook environment
7 ► Running buildkite-decrypt-hooks environment
10 ► Running buildkite-env-hooks environment
24 ► Running global pre-checkout hook
26 ► Preparing working directory
69 ► Running global post-checkout hook
71 ► Running global pre-command hook
73 ► Running buildkite-metrics-hooks pre-command
76 ► Running commands
80 ► Installing Dependencies
530 ► ! Merge the latest MASTER for best results
531 ►  Generate GraphQL schema for PR branch
532 ►  Generate GraphQL schema for master
534 ▾  Check for Potential breaking changes
535
536 Detected the following changes (2) between schemas:
537
538 * Field rangeEcho was removed from object type Test
539 * Field Test.echo changed type from Int to Int32
540
541
542 error Detected 2 breaking changes
543 ▾ 
544 Previous HEAD position was 3f97f86bc96 Setup Gateway Constants (#34731)
545 HEAD is now at b7c43722603 I want to break things, but only in testing
546  Error: The command exited with status 1
549 ► Running global post-command hook
551 ► Running global pre-exit hook
553 ► Running buildkite-metrics-hooks pre-exit
```

Exited with status 1

# **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?

The screenshot shows the Wayfair website homepage. At the top, there's a navigation bar with a back arrow, forward arrow, refresh button, and a URL bar containing "wayfair.com/?debug=1". To the right of the URL are browser extension icons and a user profile. Below the navigation is a search bar with placeholder text "Find anything home..." and a purple search icon. On the left, there's a "Shop Departments" dropdown menu and a "My Projects" section with a house icon and a question mark icon. To the right of the search bar is a shopping cart icon. The main content area features a title "Shop by Department" above a grid of six categories: Furniture, Outdoor, Bed & Bath, Decor & Pillows, Rugs, and Lighting. Each category has a small thumbnail image above its name. Below these are two more rows of categories: Renovation, Appliances, Kitchen, Baby & Kids, Storage, and two more "Sale" categories. A red circular "Sale" button is positioned next to the Storage category. At the bottom, there's a promotional banner for "HOME UPDATES" with a "UP TO 65% OFF" offer and a yellow sofa image.

wayfair.com/?debug=1

Shop Departments

Find anything home...

My Projects

Help Center

## Shop by Department

Furniture

Outdoor

Bed & Bath

Decor & Pillows

Rugs

Lighting

Renovation

Appliances

Kitchen

Baby & Kids

Storage

Sale

Sale

UP TO  
65%  
OFF

HOME  
UPDATES  
\$1500

# MoMoney

The screenshot shows a GraphQL playground interface with the following elements:

- Header:** A toolbar with tabs for "PRETTIFY" (selected), "HISTORY", and "COPY CURL". It also includes a search icon, a "+" button, and a gear icon.
- Query Editor:** A code editor containing the following GraphQL query:

```
1 ▼ {  
2   test {  
3     echo(id:123)  
4     rangeEcho(start: 10, end: 20)  
5   }  
6 }
```
- Result Panel:** A large central panel displaying the JSON response to the query. The response is:

```
▼ {  
  "data": {  
    "test": {  
      "echo": 123,  
      "rangeEcho": [  
        10,  
        11,  
        12,  
        13,  
        14,  
        15,  
        16,  
        17,  
        18,  
        19,  
        -  
      ]  
    }  
  }  
}
```
- Play Button:** A large circular play button with a play icon, positioned between the query editor and the result panel.
- Schema Panel:** A green sidebar labeled "SCHEMA" on the right side of the result panel.
- Tracing Panel:** A yellow-bordered box at the bottom labeled "TRACING" containing performance metrics:

Stage	Time
Request	177 ms
test	275 µs
test.echo	13 µs
test.rangeEcho	11 µs
Response	12 ms
- Query Variables:** A section at the bottom left labeled "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

A GraphQL query without any errors

**Response Code**

**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

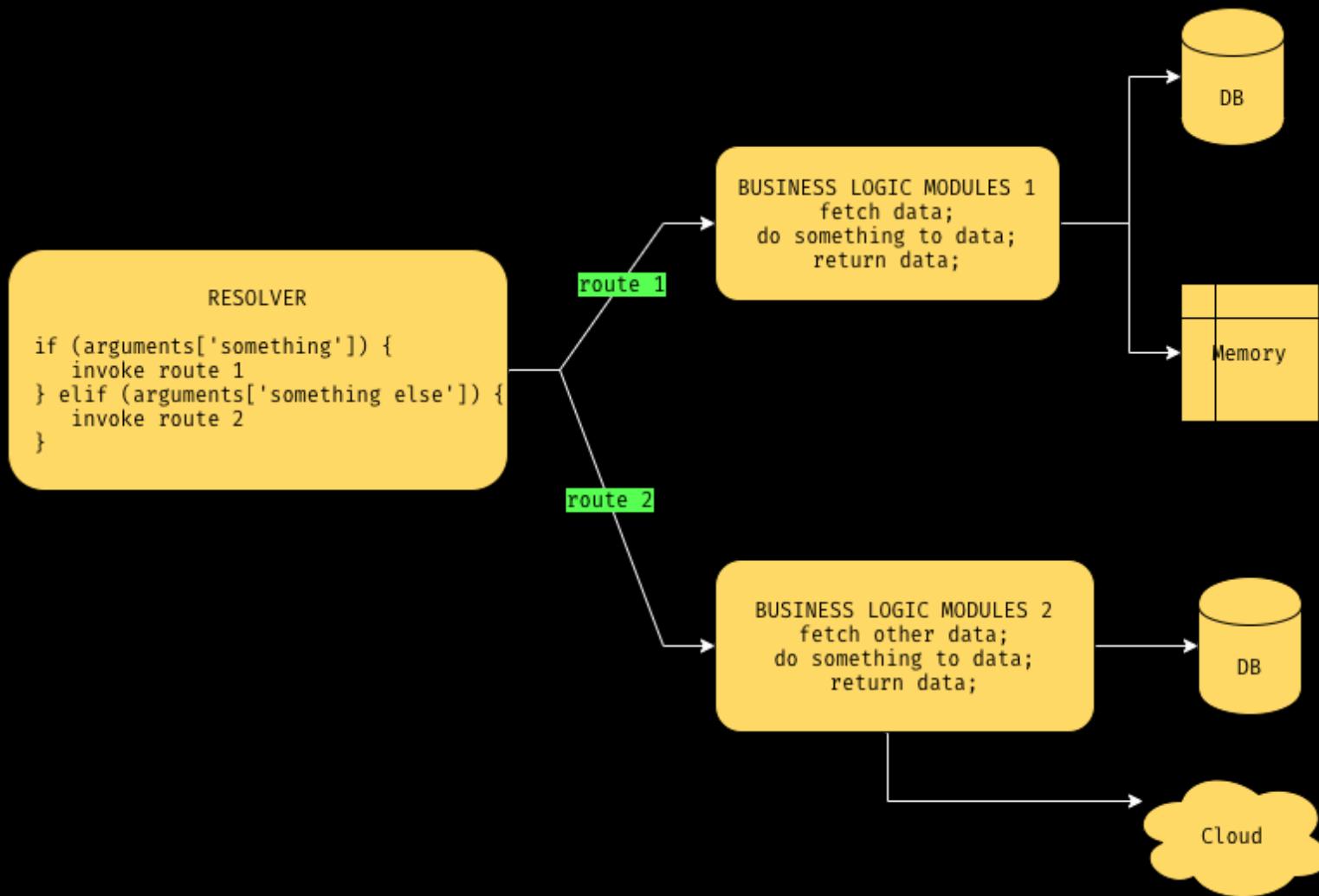
**400 Bad Request**

# **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 than 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 😎.

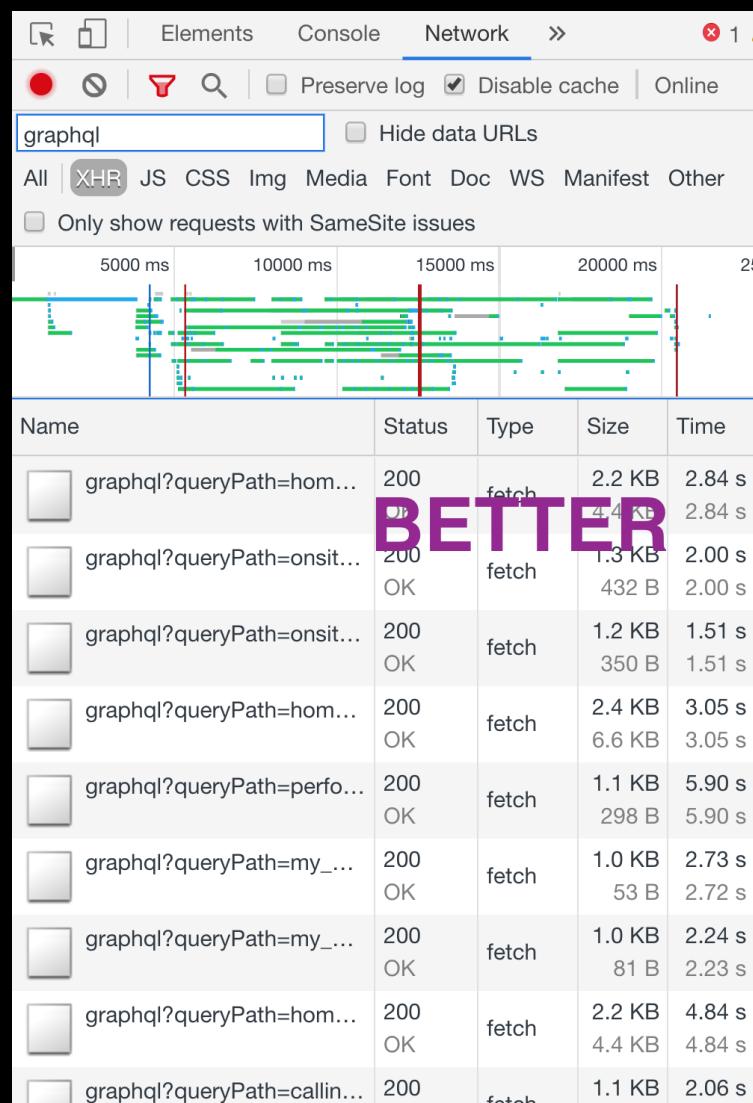
# **6- Persisted Queries**

They are freaking awesome

- Persisted 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

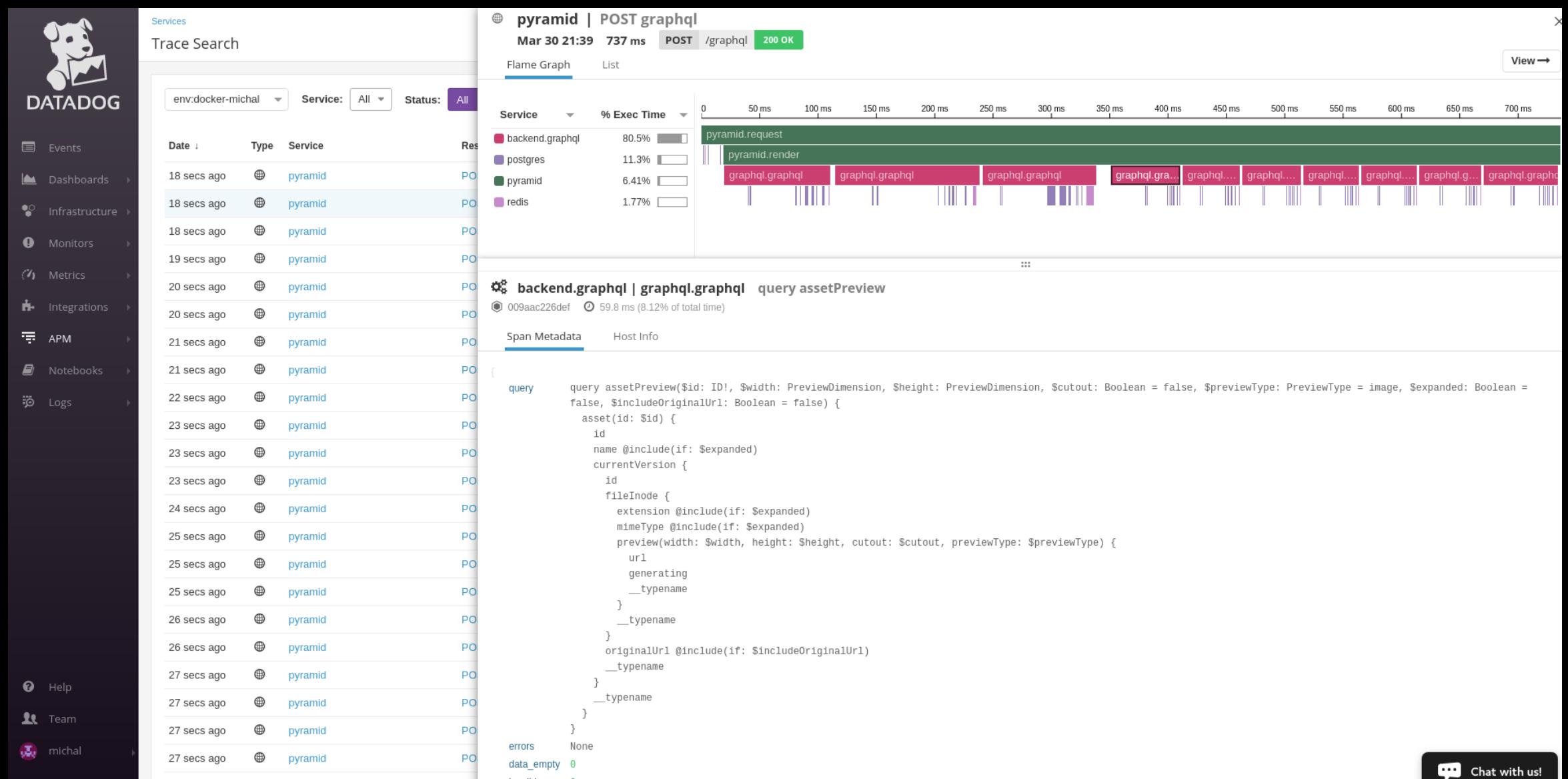
- [Distributed \(Open\) Tracing](#), also called distributed request tracing, is a method used to profile and monitor applications, especially those built using a microservices architecture.
- Distributed tracing helps pinpoint where failures occur and what causes poor performance.
- Services like [Datadog](#)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

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

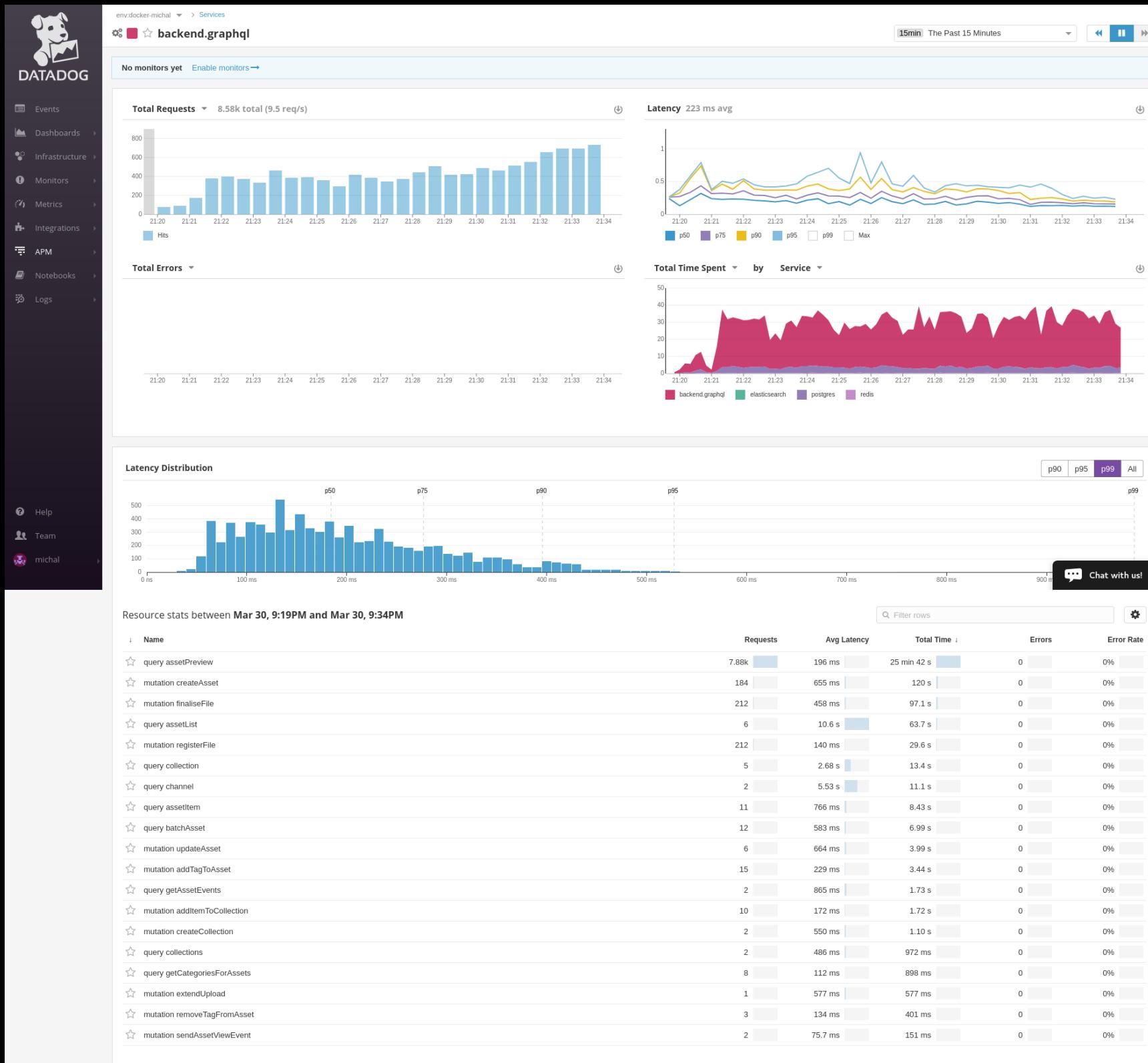
# Examples

## DDTrace GraphQL in Python

## Datadog APM docs

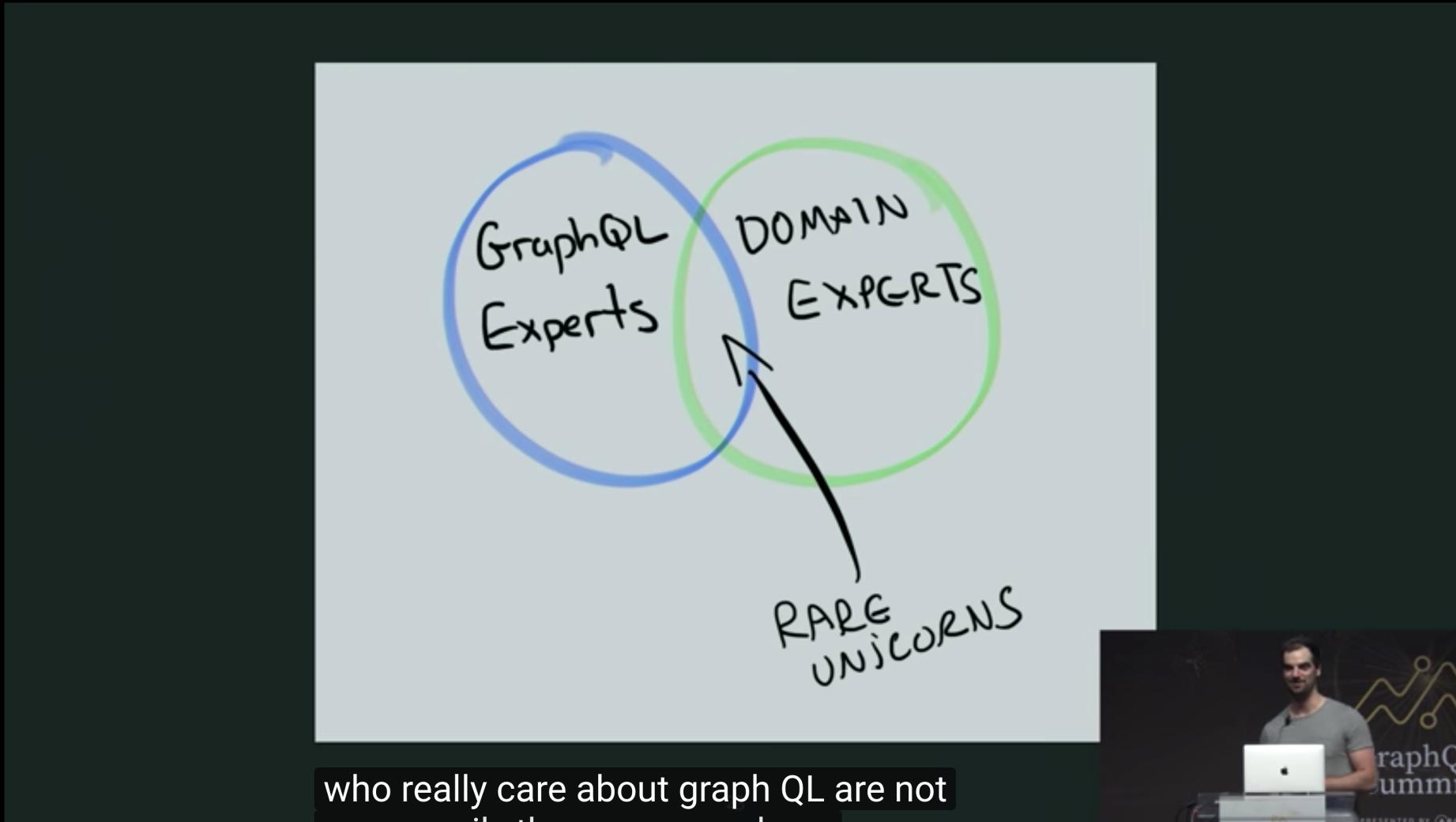


# Examples (cont.)



# 9- Inclusive community

- Empower teams to be experts
- They know their domain much better
- [GraphQL Schema Design](#)



# 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](https://twitter.com/stymied_sloth)

LinkedIn [www.linkedin.com/in/harshadeepreddy/](https://www.linkedin.com/in/harshadeepreddy/)

Github [github.com/StymiedSloth](https://github.com/StymiedSloth)

**Fin.**

Thank you.