9 Ways to Secure Your Graph

You’ve built, tested & deployed your graph

🎉

You’ve built, tested & deployed your graph

🎉

What can go wrong?

Malicious actors?

Malicious actors?

Slow or failing queries?

Malicious actors?

Slow or failing queries?

Manage public schema access?

Malicious actors?

Slow or failing queries?

Manage public schema access?

Handling deprecations safely?

Malicious actors?

Slow or failing queries?

Manage public schema access?

Handling deprecations safely?

Well-known GraphQL exploits?

Let’s learn

a baseline for graph security

Auth

1. Authentication
2. Authorization

Reducing the attack surface area

3. Mitigating malicious queries
4. Limiting API discoverability
5. Batched requests

Operations & Governance

6. Stability
7. Managing graph access
8. Observability
9. Monitoring

Auth

Authentication

You are who you say you are

Sessions + Identity

Authorization

What are you allowed to see and do?

Permissions + Capabilities

Auth

#1. Authentication

- Maintain session for a particular user through the use of context
- Different ways to handle this
  - JWT
  - 3rd party (ex: Auth0)

```
const { ApolloServer } = require('apollo-server');

const server = new ApolloServer({
 typeDefs,
 resolvers,
 context: ({ req }) => {

   // Get the user token from the headers.
   const token = req.headers.authorization || '';

   // Try to retrieve a user with the token
   const user = getUser(token);

   // Add the user to the context
   return { user };
 },
});

server.listen().then(({ url }) => {
 console.log(`🚀 Server ready at ${url}`)
});
```

Auth

#1. Authentication

- Authenticating within GraphQL, you can then use the context object to pass session information to lower layers.

```
const resolvers = {
  ...
  me: (parent, args, context) => {
    if (!context.user) {
     return null;
    }

    return context.models.User
      .getById(context.user.id);
   }
}
```

Auth

#1. Authentication

You can also:

- Handle auth in data models
- Use custom directives
  - Ex: type Reviews @isAuthenticated
- Perform auth work outside of GraphQL (pass to REST endpoint)
  - Ex: Request → GraphQL → RESTful API (auth)
  - Makes sense for RESTful APIs that already have auth logic built in

Key resource

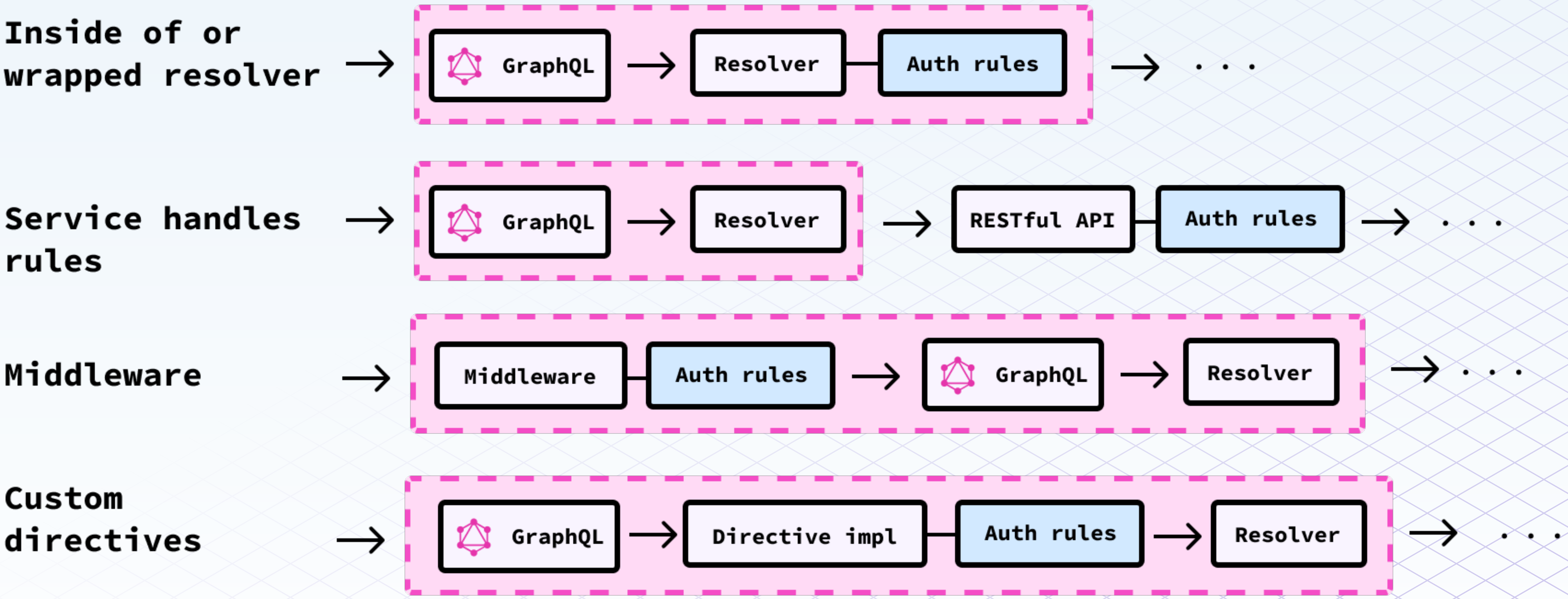
Apollo Docs "Authentication and authorization"

apollographql.com/docs/apollo-server/security/authentication/

Auth

#2. Authorization

- Do you have permission to do this?
    - Example roles:
        - `Admin`, `Editor`, `Contributor`, `Subscriber`
    - Roles have permissions/capabilities:
        - Admin → `EditPage`, `EditOthersPages`, `ReadPrivatePosts`
        - Editor → `EditPage`



Inside of or wrapped resolver → GraphQL → Resolver Auth rules → . . .

Service handles rules → GraphQL → Resolver → RESTful API Auth rules → . . .

Middleware → Middleware Auth rules → GraphQL → Resolver → . . .

Custom directives → GraphQL → Directive impl Auth rules → Resolver → . . .

Auth

#2. Authorization

- There is no single correct way to set up authorization
- Custom directives (e.g @auth (requires: ADMIN))
- Wrap resolver functions
- Put auth rules into middleware layer (e.g. graphql-shield)
- Delegate to use case/application layer

Key resources

"How to Auth: Secure a GraphQL API with Confidence" by Mandi Wise

From GraphQL Summit Worldwide 2020

"Setting Up Authentication and Authorization with Apollo Federation" by Mandi Wise

via the Apollo Blog

Rules and Capabilities in WordPress

https://wordpress.org/support/article/roles-and-capabilities/

Reducing the attack surface area

Reducing the attack surface area

#3. Mitigating malicious queries

→ *Limit query depth*

- GraphQL gives clients the ability to ask for data in a variety of different ways. Because of the various entry-points available to request data, it's possible to write exceptionally large nested queries.
- Queries like this are dangerous
  - They're expensive to compute.
  - They could crash our API and take up all available resources.

```
query {
 author(id: 42) {
   posts {
     author {
       posts {
         author {
           posts {
             author {
               # and so on...
             }
           }
         }
       }
     }
   }
 }
}
```

Reducing the attack surface area

#3. Mitigating malicious queries

→ *Limit query depth*

- graphql-depth-limit
  - https://github.com/stems/graphql-depth-limit
- easily limit the maximum depth of incoming queries

```
app.use('/api', graphqlServer({
  validationRules: [depthLimit(10)]
}));
```

Reducing the attack surface area

#3. Mitigating malicious queries

→ *Paginate list fields*

- Query depth isn't the only thing to worry about. We should also be conscious of how query amount could affect the performance of our API.
- Example: If there were 100 authors, each with 100 posts, this query would attempt to return 100,000 nodes 💀 .
- Can slow (or DoS) your server.

```
query {
  authors(first: 1000) {
    name
    posts(last: 100) {
      title
      content
    }
  }
}
```

Reducing the attack surface area

#3. Mitigating malicious queries

→ *Paginate list fields*

- graphql-input-number
  - https://github.com/joonhocho/graphql-input-number
- Example: We can restrict the maximum value to 100
- We can also perform these checks in the resolver imperatively.

```
const PaginationAmount = GraphQLInputInt({
  name: 'PaginationAmount',
  min: 1,
  max: 100,
});

...


type Thread {
  messages(first: PaginationAmount,
   after: String): [Message]
}
```

Reducing the attack surface area

#3. Mitigating malicious queries

→ Improve validation & sanitization

- Standard web application security practices.
- When you accept data from a user,
  one should always expect that user-provided data could be malicious.
- Two especially malicious techniques in this area:
  - Data exfiltration: tricks the database into returning more data than originally intended
  - Data destruction: destroy production data

```
query User {
  user (id: "User*") {
    email
    id
  }
}
```

Reducing the attack surface area

#3. Mitigating malicious queries

→ *Improve validation & sanitization*

- Follow the usual rules for web application sanitization in addition to the **OSWAP GraphQL-specific recommendations** like:
  - Reject invalid input without giving away too many details

Key resource
OSWAP "GraphQL Cheat Sheet"
https://cheatsheetseries.owasp.org/cheatsheets/GraphQL_Cheat_Sheet.html#general-practices

Reducing the attack surface area

#3. Mitigating malicious queries

→ *Improve validation & sanitization*

- Follow the usual rules for web application sanitization in addition to the **OSWAP GraphQL-specific recommendations** like:
  - Reject invalid input without giving away too many details
  - Leverage the GraphQL schema to support validation

Key resource

OSWAP “GraphQL Cheat Sheet”

https://cheatsheetseries.owasp.org/cheatsheets/GraphQL_Cheat_Sheet.html#general-practices

Reducing the attack surface area

#3. Mitigating malicious queries

→ *Improve validation & sanitization*

- Follow the usual rules for web application sanitization in addition to the **OSWAP GraphQL-specific recommendations** like:
  - Reject invalid input without giving away too many details
  - Leverage the GraphQL schema to support validation
  - Beware of using JSON scalars (prone to malicious queries if not properly sanitized)

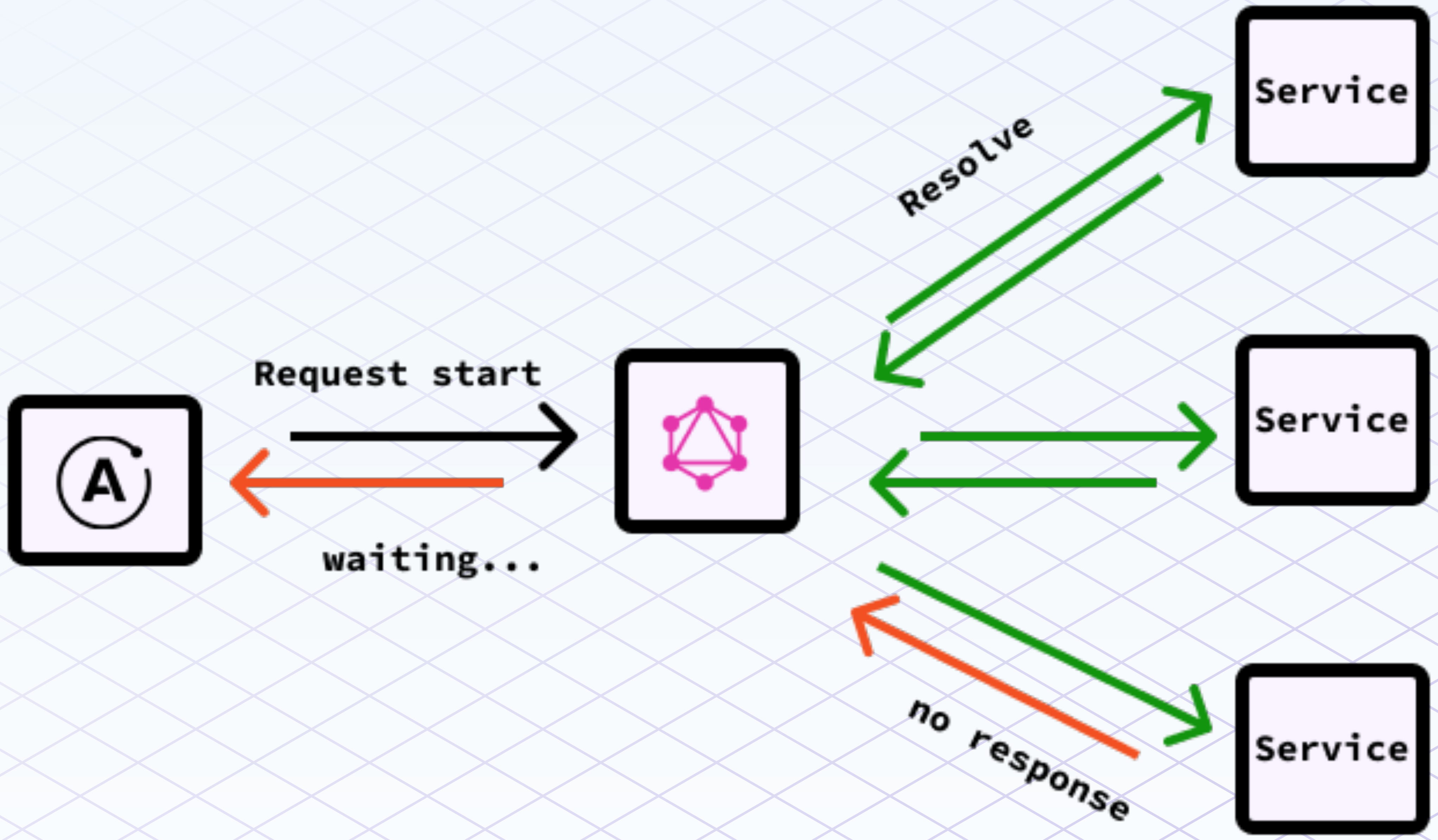
Key resource
OSWAP “GraphQL Cheat Sheet”
https://cheatsheetseries.owasp.org/cheatsheets/GraphQL_Cheat_Sheet.html#general-practices

Reducing the attack surface area

#3. Mitigating malicious queries

→ *Use timeouts*

- When resolving data, there are various reasons why it may take a long time to respond.
  - Services could be down
  - Queries may be expensive
  - or something else might be going on.
- We don't want our GraphQL API to hang for too long, waiting for a response.



Reducing the attack surface area

#3. Mitigating malicious queries

→ *Use timeouts*

- Explore using timeouts in the following contexts:
  - On resolver functions (and using REST data sources)
  - [Federation] On requests to the gateway's Node HTTP server
  - [Federation] On requests to the subgraphs services

```
// Federation gateway – subgraph timeout
// example (credit Mandi Wise)
const gateway = new ApolloGateway({
  // ...
  buildService({ name, url }) {
    // Sets a 3 second timeout on requests
    // to subgraph
    const fetcher = (input, init) => {
      if (init) {
        init.timeout = 3000;
      } else {
        init = { timeout: 3000 };
      }
      return fetch(input, init);
    };
    return new RemoteGraphQLDataSource({
      url, fetcher
    });
  }
});
```

Reducing the attack surface area

#3. Mitigating malicious queries

→ *Rate limit APIs*

- Dictates how many requests a client can make per some time.
- Often used to prevent brute-forcing login details, scraping data, or denial of service attacks.

Key resources

GitHub's approach: "Resource limitations" based on maximum node limit + num requests in query
https://docs.github.com/en/graphql/overview/resource-limitations

Shopify's approach: "Query cost points" and the leaky bucket algorithm
https://shopify.dev/api/usage/rate-limits

graphql-rate-limit
npmjs.com/package/graphql-rate-limit

Reducing the attack surface area

#3. Mitigating malicious queries

→ *Query cost analysis*

- Despite our best efforts using query depth + amount limiting techniques, it's still possible to overload the server with semantically expensive queries.
- Sometimes we can't just look at the depth or potential amount of nodes.

```
query evilQuery {
  thread(id: "54887141-57a9-4386-807c") {
    messageConnection(first: 100) { ... }
    participants(first: 100) {
      threadConnection(first: 100) { ... }
      communityConnection { ... }
      channelConnection { ... }
      everything(first: 100) { ... }
    }
  }
}
```

```
import costAnalysis from
   ‘graphql-cost-analysis'

const costAnalyzer = costAnalysis({
  maximumCost: 1000,
})

...
```

```
type Query {
  # will have the default cost value
  defaultCost: Int

  # will have a cost of 2 because this field does not depend
  # on its parent fields
  customCost: Int @cost(useMultipliers: false, complexity: 2)

  # complexity should be between 1 and 10
  badComplexityArgument: Int @cost(complexity: 12)

  # the cost will depend on the `limit` parameter passed to the field
  # then the multiplier will be added to the `parent multipliers` array
  customCostWithResolver(limit: Int): Int
    @cost(multipliers: ["limit"], complexity: 4)

  # for recursive cost
  first(limit: Int): First
    @cost(multipliers: ["limit"], useMultipliers: true, complexity: 2)

  # you can override the cost setting defined directly on a type
  overrideTypeCost: TypeCost @cost(complexity: 2)
  getCostByType: TypeCost

  # You can specify several field parameters in the `multipliers` array
  # then the values of the corresponding parameters will be added together.
  # here, the cost will be `parent multipliers` *
  # (`first` + `last`) * `complexity
  severalMultipliers(first: Int, last: Int): Int
    @cost(multipliers: ["first", "last"])
}
```

Reducing the attack surface area

#3. Mitigating malicious queries

→ *Safelist operations*

- During development, front-end engineers can explore all the data available and fetch what they need for the components they're building.

- But in production, this amount of flexibility can be unnecessary and undesirable

- If we know what operations we’re going to perform, can’t we make it so that we can only perform those?

Reducing the attack surface area

#3. Mitigating malicious queries

→ *Safelist operations*

- Catchall approach: maintain a list of approved operations allowed to execute against your graph
    - **Operation safe listing**
- Setup
    - 1. Register your schema
    - 2. Register the operations from your client bundle
    - 3. Add the operation registry plugin to Apollo Server

```
const server = new ApolloServer({
  // Existing configuration
  typeDefs,
  resolvers,
  subscriptions: false,
  // ...
  // New configuration
  plugins: [
    require('apollo-server-plugin-operation-registry')({
      forbidUnregisteredOperations: true,
    }),
  ],
});
```

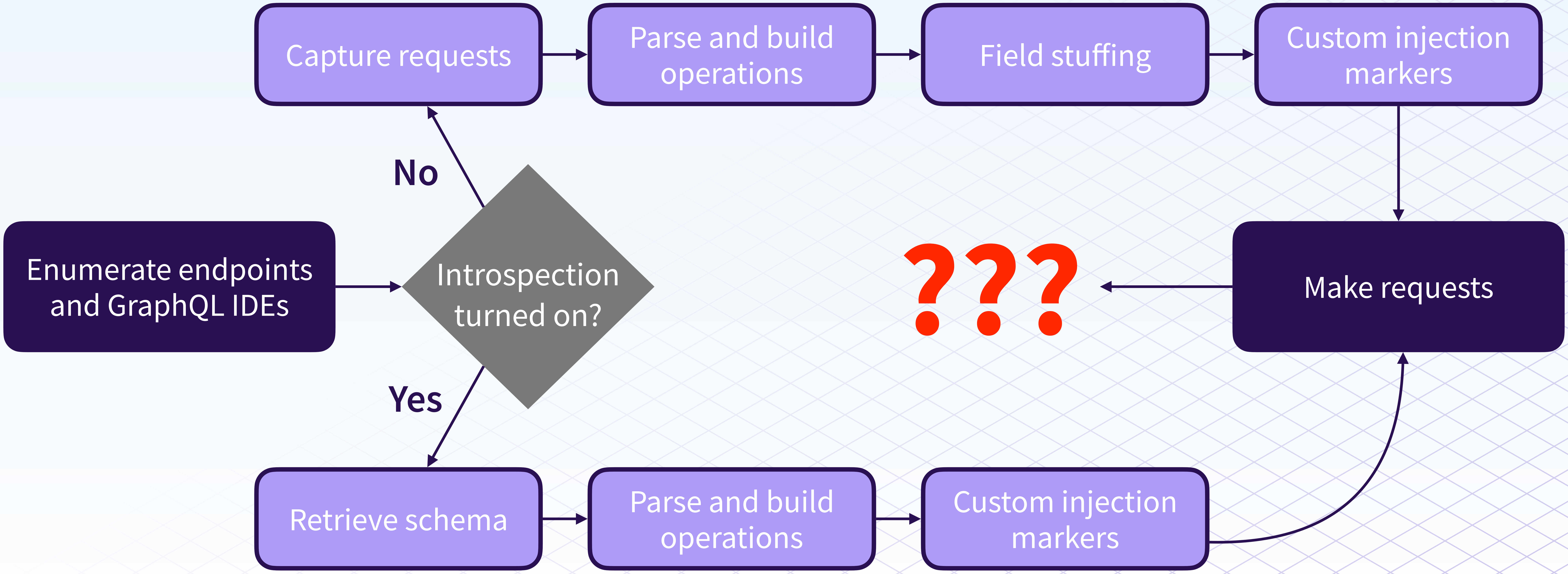
Reducing the attack surface area

#4. Limit API discoverability

→ *Turn off introspection in production*

- Introspection is for development and tooling purposes.
- Behind the scenes, GraphQL IDEs are powered by introspection queries
- With Apollo Server, introspection is on by default unless the NODE_ENV environment variable is set to production

```
const server = new ApolloServer({
  typeDefs,
  resolvers,
  introspection: process.env.NODE_ENV
    !== 'production'
});
```



Adapted from: https://youtu.be/NPDp7GHmMa0

Reducing the attack surface area

#4. Limit API discoverability

→ *Turn off introspection in production*

- With introspection disabled, how do we:
  - Enable new developers to explore the current schema and its capabilities?
  - Utilize tooling during development?
  - Query production data?

Use a schema registry

> There should be a **single source of truth** for registering and tracking the graph
>
> - via principledgraphql.com

- Similarly to how your track your source code with Git, a schema registry exists to keep track of your graph and how it changes over time

- Here are two ways to register your schema to Apollo Studio

  - 1. **Through schema reporting**
    - In Apollo Server set `APOLLO_SCHEMA_REPORTING=true`

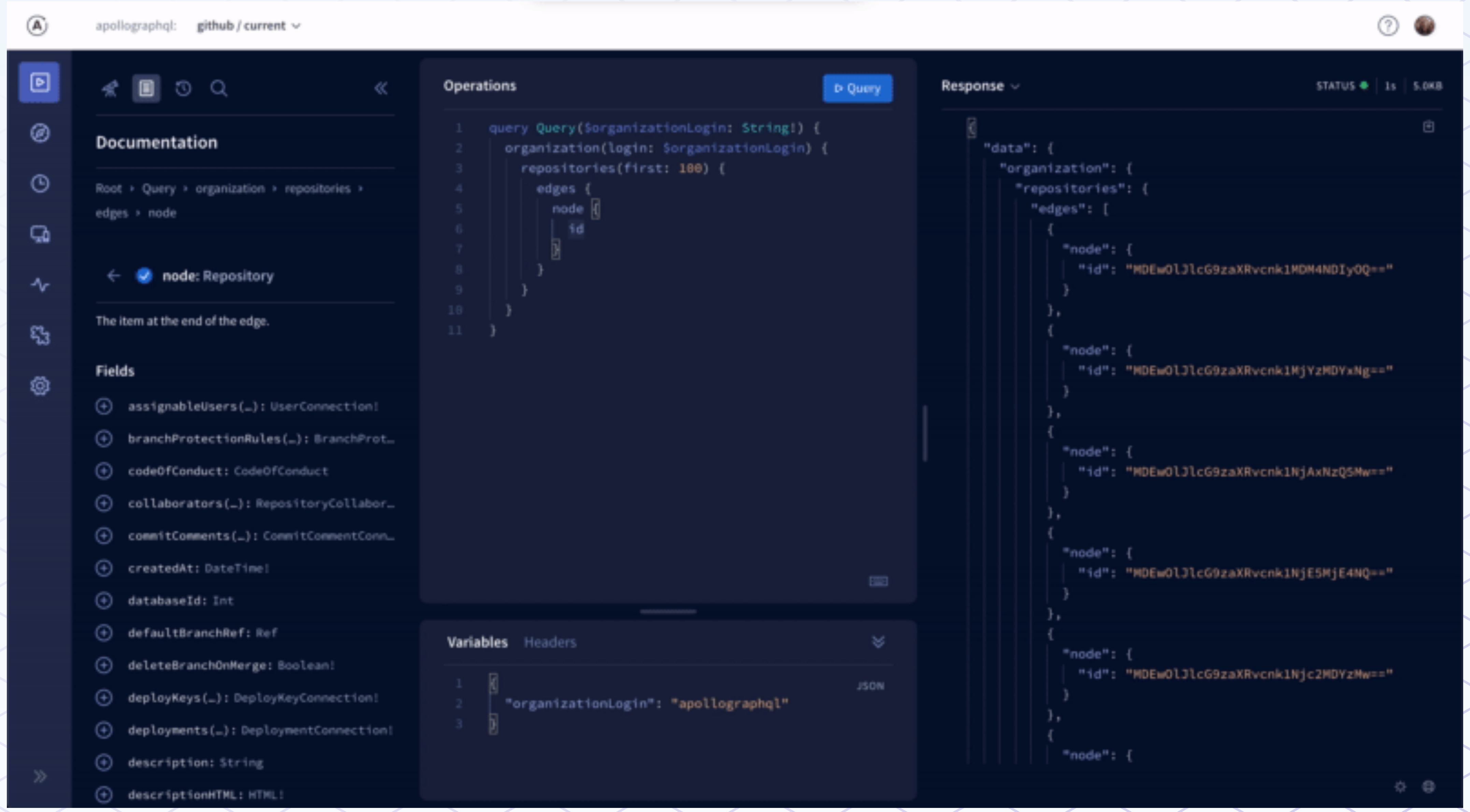
  - 2. **Through the Rover CLI**
    - `rover graph publish`

Explore the schema’s shape and data

Explorer — *build queries and explore data*

Schema reference — *out of the box documentation*

Graph README — *to onboard developers to the graph*



Root › Query › organization › repositories › edges › node

← ✓ node: Repository

The item at the end of the edge.

Fields

- assignableUsers(…): UserConnection!
- branchProtectionRules(…): BranchProt…
- codeOfConduct: CodeOfConduct
- collaborators(…): RepositoryCollabor…
- commitComments(…): CommitCommentConn…
- createdAt: DateTime!
- databaseId: Int
- defaultBranchRef: Ref
- deleteBranchOnMerge: Boolean!
- deployKeys(…): DeployKeyConnection!
- deployments(…): DeploymentConnection!
- description: String
- descriptionHTML: HTML!

```
query Query($organizationLogin: String!) {
  organization(login: $organizationLogin) {
    repositories(first: 100) {
      edges {
        node {
          id
        }
      }
    }
  }
}
```

```
{
  "organizationLogin": "apollographql"
}
```

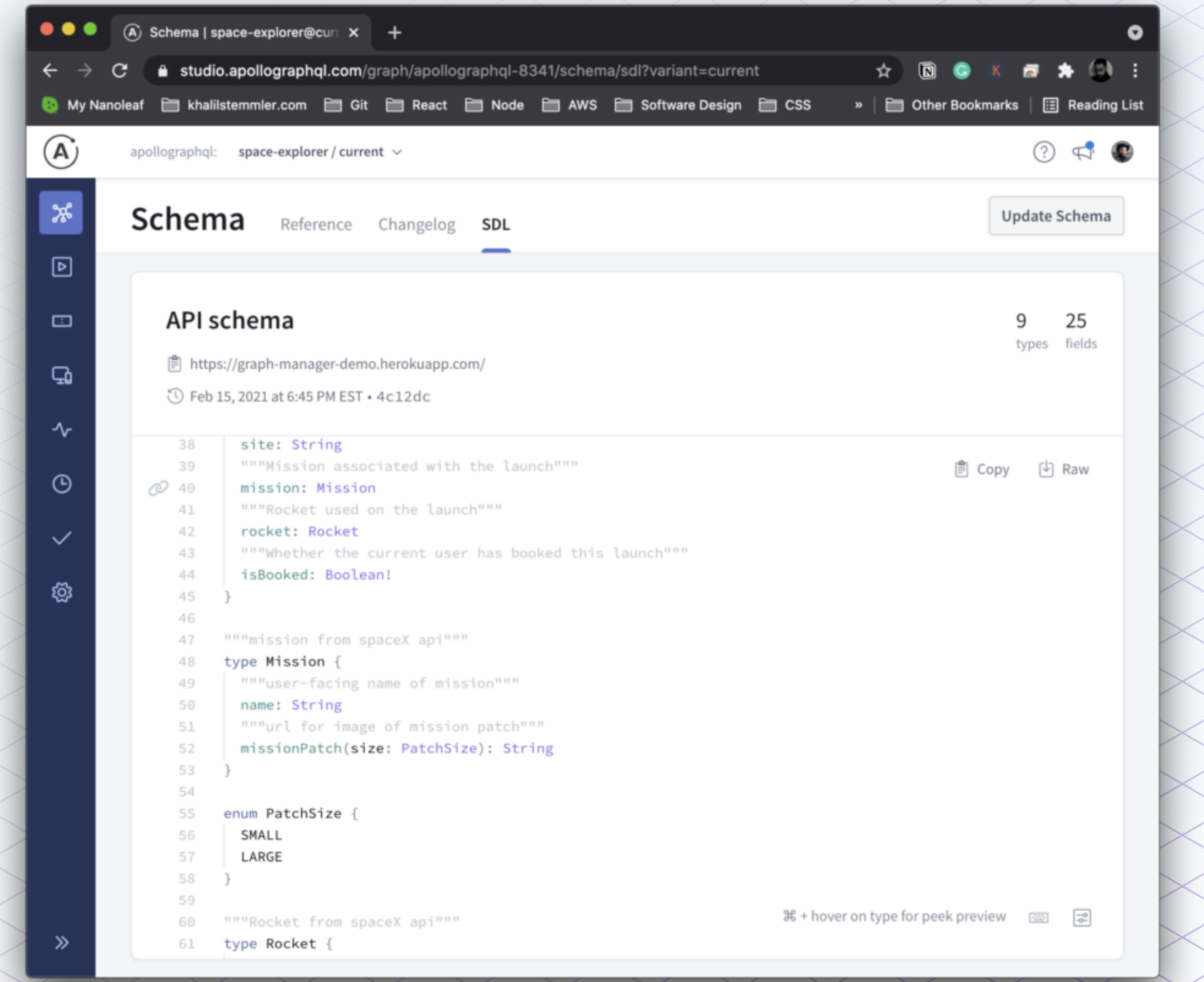
```
{
  "data": {
    "organization": {
      "repositories": {
        "edges": [
          {
            "node": {
              "id": "MDEwOlJlcG9zaXRvcnk1MDM4NDIyOQ=="
            }
          },
          {
            "node": {
              "id": "MDEwOlJlcG9zaXRvcnk1MjYzMDYxNg=="
            }
          },
          {
            "node": {
              "id": "MDEwOlJlcG9zaXRvcnk1NjAxNzQ5Mw=="
            }
          },
          {
            "node": {
              "id": "MDEwOlJlcG9zaXRvcnk1NjE5MjE4NQ=="
            }
          },
          {
            "node": {
              "id": "MDEwOlJlcG9zaXRvcnk1Njc2MDYzMw=="
            }
          },
          {
            "node": {
```

Explore the schema’s shape and data

Explorer — *build queries and explore data*

Schema reference — *out of the box documentation*

Graph README — *to onboard developers to the graph*



API schema

9 types 25 fields

https://graph-manager-demo.herokuapp.com/

Feb 15, 2021 at 6:45 PM EST • 4c12dc

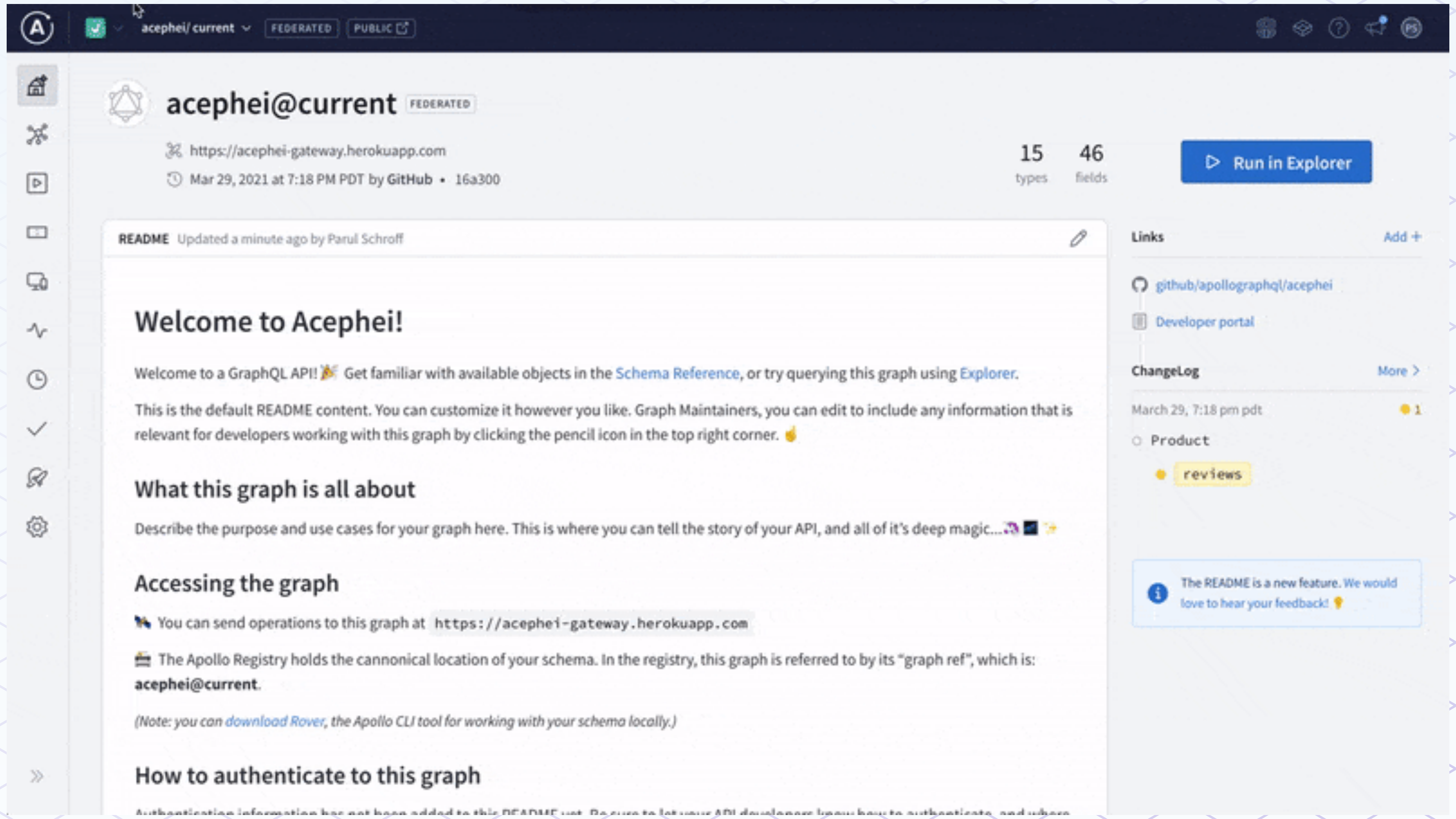
```
  site: String
  """Mission associated with the launch"""
  mission: Mission
  """Rocket used on the launch"""
  rocket: Rocket
  """Whether the current user has booked this launch"""
  isBooked: Boolean!
}

"""mission from spaceX api"""
type Mission {
  """user-facing name of mission"""
  name: String
  """url for image of mission patch"""
  missionPatch(size: PatchSize): String
}

enum PatchSize {
  SMALL
  LARGE
}

"""Rocket from spaceX api"""
type Rocket {
```

Explore the schema’s shape and data

Explorer — ***build queries and explore data***

Schema reference — ***out of the box documentation***

Graph README — ***to onboard developers to the graph***



Welcome to Acephei!

Welcome to a GraphQL API! 🎉 Get familiar with available objects in the Schema Reference, or try querying this graph using Explorer.

This is the default README content. You can customize it however you like. Graph Maintainers, you can edit to include any information that is relevant for developers working with this graph by clicking the pencil icon in the top right corner. ☝

What this graph is all about

Describe the purpose and use cases for your graph here. This is where you can tell the story of your API, and all of it's deep magic...🦄 🌌 ✨

Accessing the graph

🛰 You can send operations to this graph at `https://acephei-gateway.herokuapp.com`

🗄 The Apollo Registry holds the cannonical location of your schema. In the registry, this graph is referred to by its “graph ref”, which is: **acephei@current**.

(Note: you can download Rover, the Apollo CLI tool for working with your schema locally.)

How to authenticate to this graph

Authentication information has not been added to this README yet. Be sure to let your API developers know how to authenticate, and where

The README is a new feature. We would love to hear your feedback! 💡

Reducing the attack surface area

#4. Limit API discoverability

→ *Mask errors in production*

- When server or downstream service errors occur, it's a good idea to withhold the exact specifics of what went wrong from the client.
- Returning *complete* error details to the client exposes the current server vulnerabilities and opens the door for more concentrated attacks.

```
{
  "data": {
    "astronaut": null
  },
  "errors": [{
    "message": "Database Error: Astronaut
      does not exist",
    "extensions": {
      "code": "INTERNAL_SERVER_ERROR",
      // ...
      "exception": {
        “stacktrace": [
          "Database Error: User does not exist",
          " at __resolveReference (../services
 /vehicles/index.js:29:13)”,
          // ...
        ],
        // …
      }
    }
  }]
}
}
```

Reducing the attack surface area

#4. Limit API discoverability

→ *Mask errors in production*

- To prevent this issue, swallow errors before they get to the client.
- You can use the formatError API in Apollo Server to implement this.

```
const server = new ApolloServer({
  typeDefs,
  resolvers,
  formatError: (err) => {
    // Don't give the specific errors to
    // the client
    if (err.message.startsWith('Database Error:')) {
      return new Error(
       'Internal server error’
      );
    }
    // Otherwise return the original error
    return err;
  },
});
```

Reducing the attack surface area

#4. Limit API discoverability

→ *Mask errors in production*

- Errors vs. Exceptions
- Errors → Expected and application-specific
  - UserAlreadyExists, UserDoesntExist, InvalidPermissions
- Exceptions → Unexpected and infrastructural
  - Database, source code, or network connectivity problems

```
# Application-specific errors with
# GraphQL unions
union UpvotePost = UpvotePostSuccess
   | MemberNotFound
   | PostNotFound
   | AlreadyUpvoted
```

Reducing the attack surface area

#4. Limit API discoverability

→ *Mask errors in production*

- Errors vs. Exceptions
- Errors → Expected and application-specific
  - UserAlreadyExists, UserDoesntExist, InvalidPermissions
- Exceptions → Unexpected and infrastructural
  - Database, source code, or network connectivity problems

Key resources

Unions and interfaces

via the Apollo Docs

200 OK! Error Handling in GraphQL by Sasha Solomon

via GraphQL Summit Worldwide 2020

Reducing the attack surface area

#4. Limit API discoverability

→ Avoid schema autogeneration

- Some tools can autogenerate a GraphQL schema based on database tables, etc.
- While these tools tend to speed you up in the short run, used as your public graph, it becomes very easy to guess fields on the root operation types based on CRUD patterns.
- Prefer a demand-oriented schema

> The schema should be **built incrementally** based on actual requirements and **evolve smoothly** over time

- via principledgraphql.com

Reducing the attack surface area

#5. Batched requests

→ *Limit query breadth*

- Clients can use aliases to write batch queries like the following:
- Someone may write a query like this to purposefully disrupt performance, scrape as much data as fast as possible, or attempt to mitigate rate-limiting.

```
query MaliciousQuery {
  alias1: fieldName { subField1 subField2 ...}
  alias2: fieldName { subField1 subField2 ...}
  ...
  alias10: fieldName { subField1 subField2 ...}
  ...
  alias100: fieldName { subField1 subField2 ...
  ...
  alias1000: fieldName { subField1 subField2 ...}
  ...
}
```

```
query Mutation (
  $input1: LoginInput,
  $input2: LoginInput,
  $input3: LoginInput
  # ... And more
) {
  first: login (input: $input1) {
    token
  }

  second: login (input: $input2) {
    token
  }

  third: login (input: $input3) {
    token
  }

  # .. And so on
}
```

Brute-force attempt

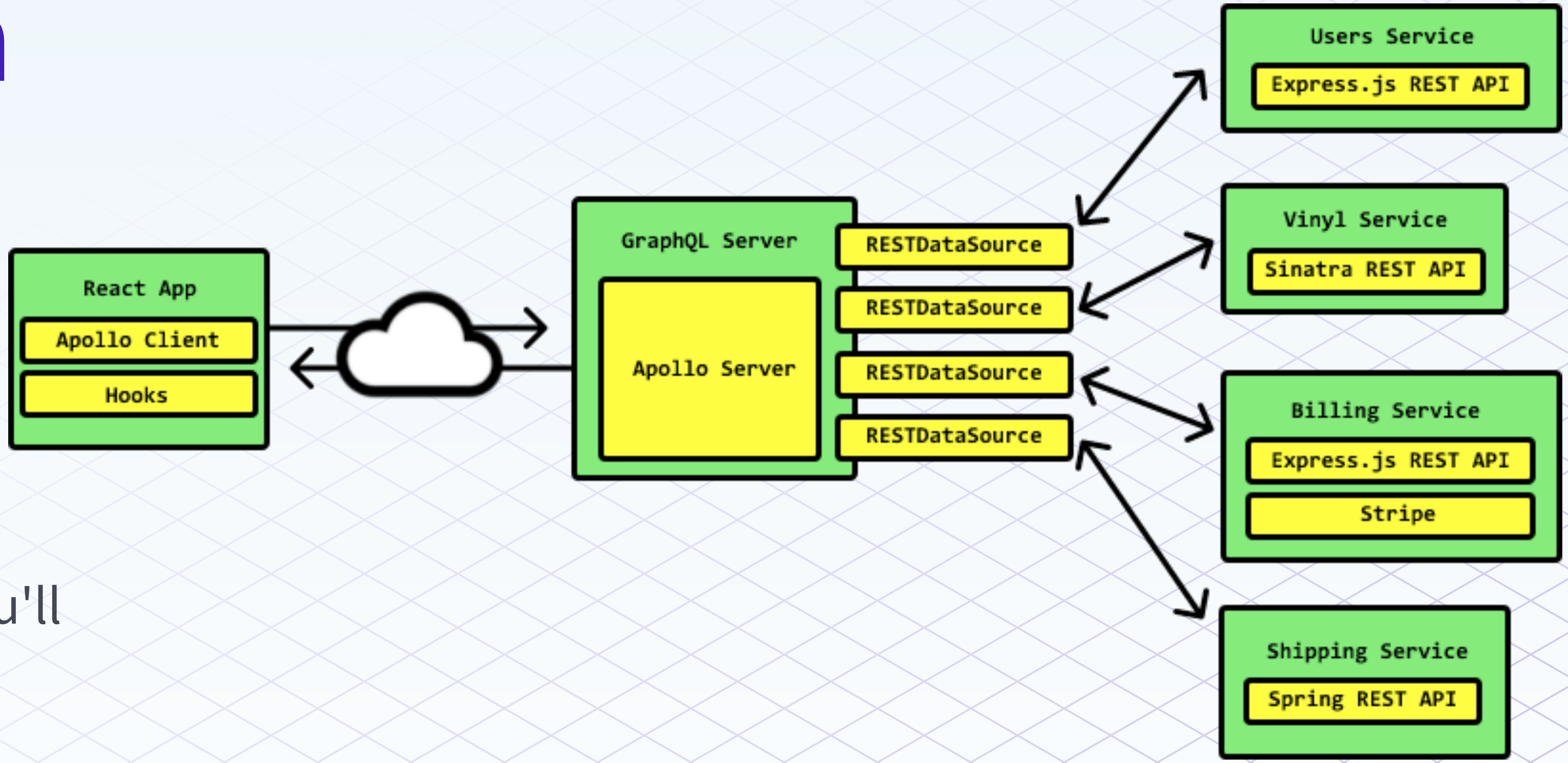
Solution: Use a combination of rate-limiting and query complexity analysis.

Reducing the attack surface area

#5. Batched requests

→ *Use data loaders to prevent DoS-ing yourself*

- If you're resolving data from backing data sources (like a REST API or a subgraph), you'll want to make efficient use of the network to prevent DoS-ing yourself.



Reducing the attack surface area

#5. Batched requests

→ *Use data loaders to prevent DoS-ing yourself*

- A great technique is to use data loaders to minimize the number of requests to backing data sources from resolvers
- Also, consider caching as an approach to mitigating the number of necessary requests. You can implement caching at various levels:
  - Client, gateway, data source, subgraph, etc

Key resources

DataLoader
https://github.com/graphql/dataloader

How Apollo REST Data Source Deduplicates and Caches API calls
https://khalilstemmler.com/blogs/graphql/how-apollo-rest-data-source-caches-api-calls/

Using Memcached/Redis as a cache storage backend
via the Apollo docs

Operations & Governance

Operations

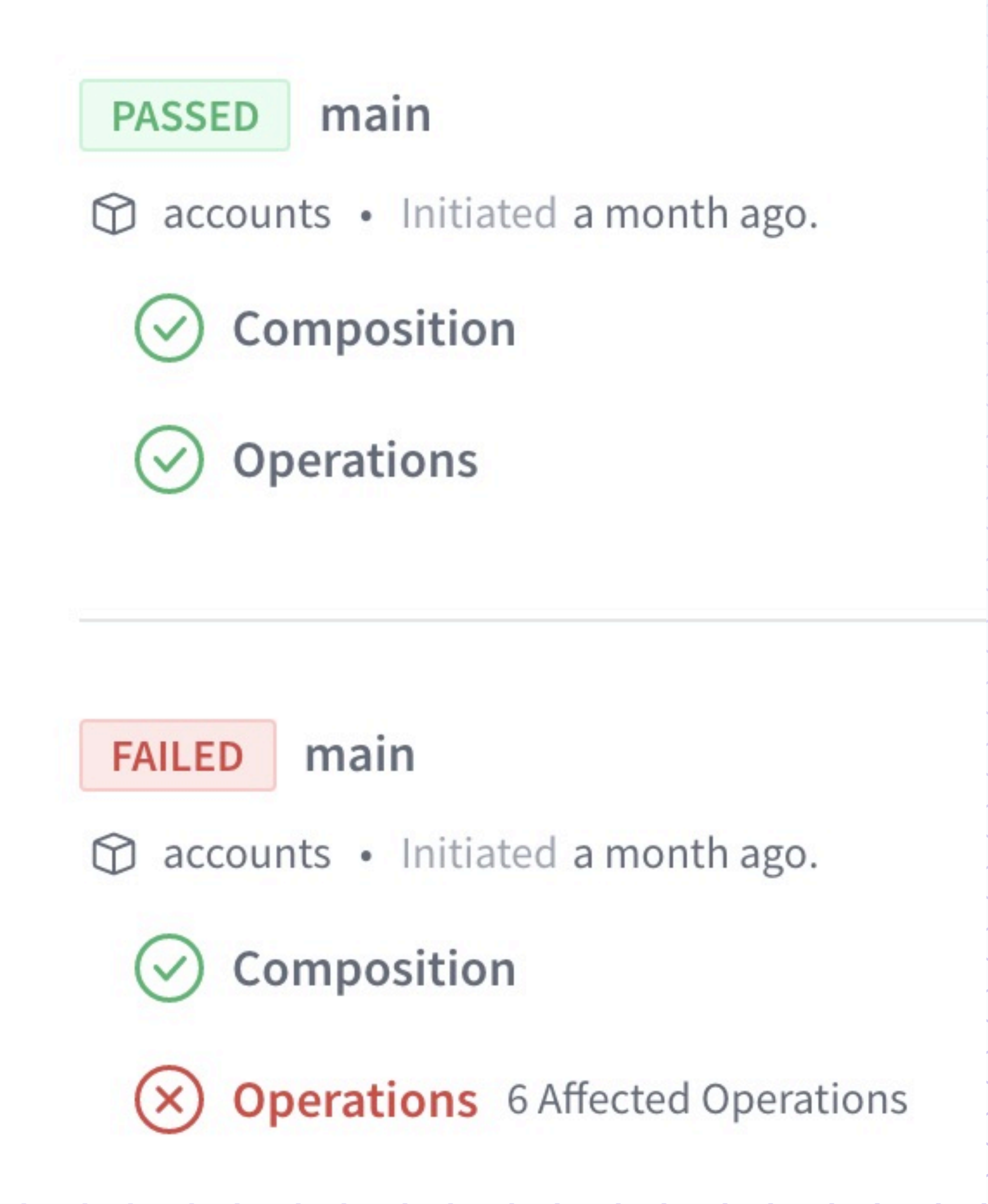
#6. Stability

- By design, GraphQL isn’t a *versioned* API.
    - In an Agile fashion, you deprecate and evolve fields (sometimes multiple times a day).
- How can we do this safely? Won’t we break clients?

Operations

#6. Stability

- Schema checks
  - **Operations**: Will your proposed schema changes break any of your graph's active clients?
  - **Composition**: For federated graphs, will changes to a subgraph successfully compose with your *other* registered subgraph schemas.



PASSED main

accounts • Initiated a month ago.

✓ Composition

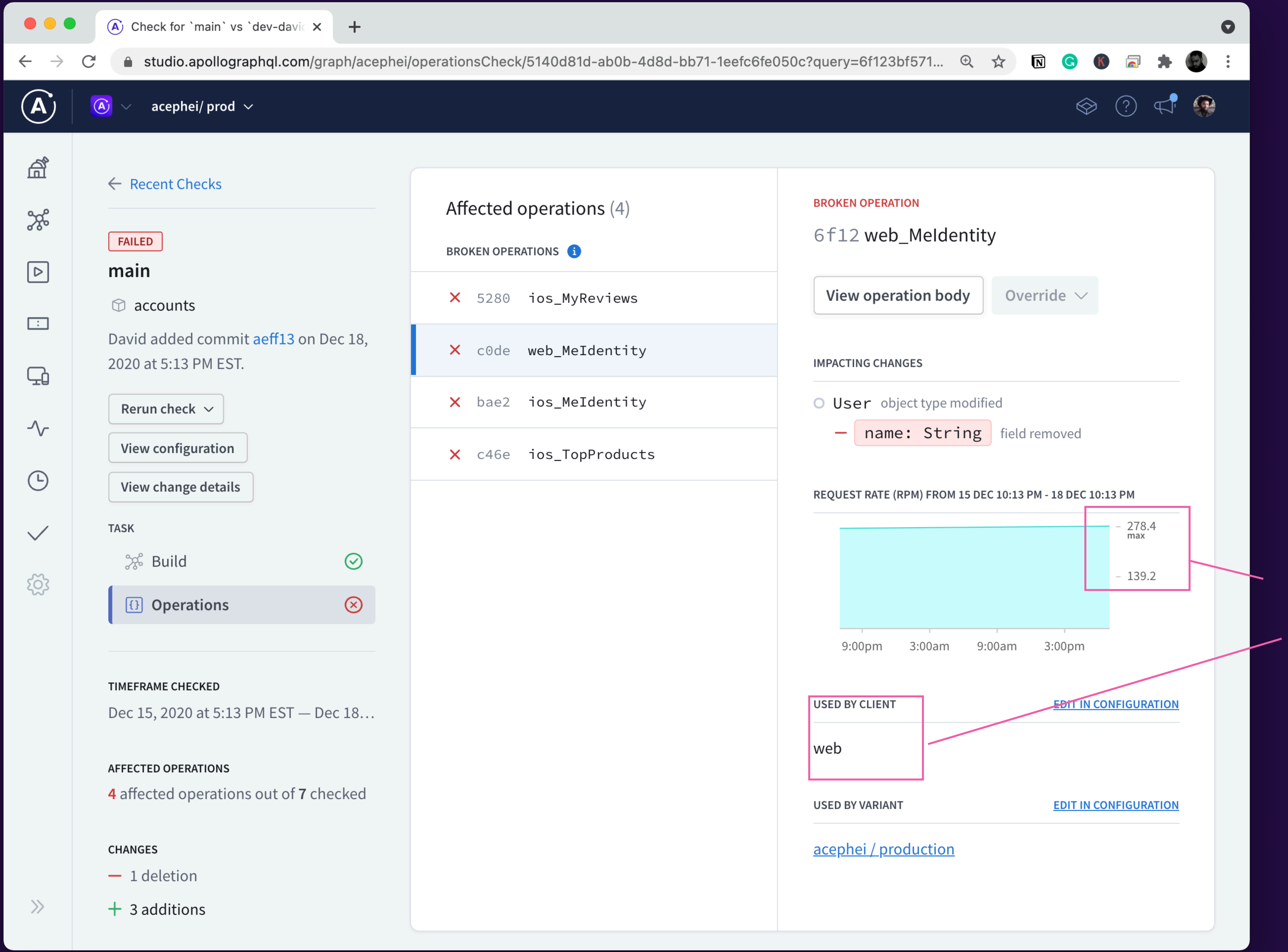
✓ Operations

FAILED main

accounts • Initiated a month ago.

✓ Composition

✗ Operations 6 Affected Operations



Check for `main` vs `dev-davi
studio.apollographql.com/graph/acephei/operationsCheck/5140d81d-ab0b-4d8d-bb71-1eefc6fe050c?query=6f123bf571...
acephei/ prod

← Recent Checks

FAILED

main

accounts

David added commit aeff13 on Dec 18, 2020 at 5:13 PM EST.

Rerun check

View configuration

View change details

TASK

Build ✓

Operations ☒

TIMEFRAME CHECKED

Dec 15, 2020 at 5:13 PM EST — Dec 18…

AFFECTED OPERATIONS

4 affected operations out of 7 checked

CHANGES

— 1 deletion

+ 3 additions

Affected operations (4)

BROKEN OPERATIONS

```
✕ 5280  ios_MyReviews
✕ c0de  web_MeIdentity
✕ bae2  ios_MeIdentity
✕ c46e  ios_TopProducts
```

BROKEN OPERATION

6f12 web_MeIdentity

View operation body

Override

IMPACTING CHANGES

○ `User` object type modified

— `name: String` field removed

REQUEST RATE (RPM) FROM 15 DEC 10:13 PM - 18 DEC 10:13 PM

USED BY CLIENT

EDIT IN CONFIGURATION

web

USED BY VARIANT

EDIT IN CONFIGURATION

acephei / production

Detecting potentially broken clients

The web client calls these operations frequently!

Operations

#6. Stability

- Recommended to use in a CI with the Rover CLI
  - Like Jenkins or CircleCI
- Define a CI job for each variant of your schema (production, staging, etc)
  - Run `rover graph check`
  - If it returns a non-zero exit code, a breaking change has been detected.

Key resources

Rover "Getting Started" docs
via Apollo docs

Schema checks
via Apollo docs

Operations

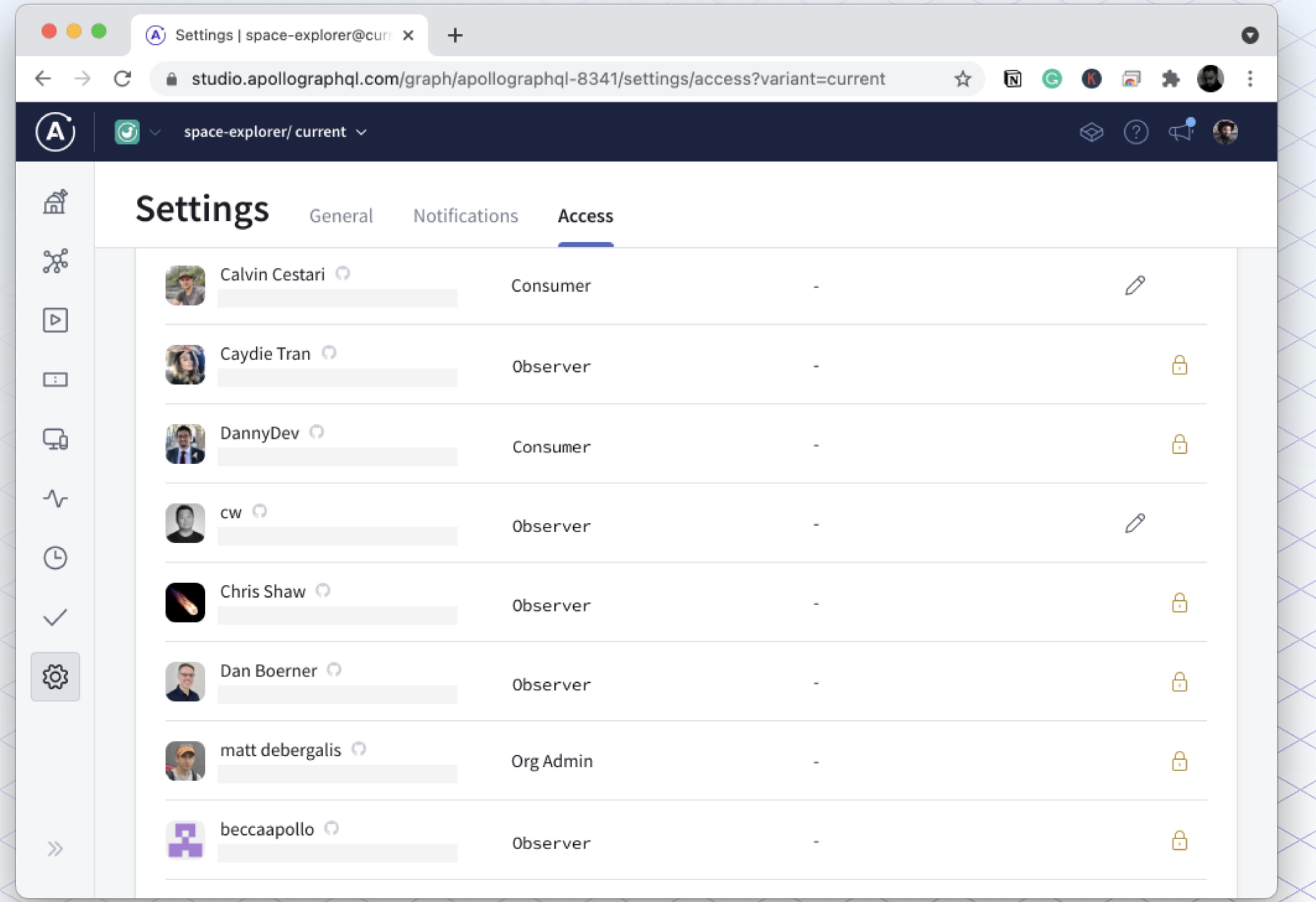
#7. Managing graph access

- As discussed earlier, we might not want our production graph to be available to everyone
  - We turn introspection off
- With introspection off, how do we safely manage graph access?
  - Teammates, non-developers, consultants, etc

Operations

#7. Managing graph access

- User roles:
  - Graph admin
  - Billing manager
  - Consumer
  - Observer
  - Contributor
  - Admin
- **Org-level roles** and **graph-level roles**

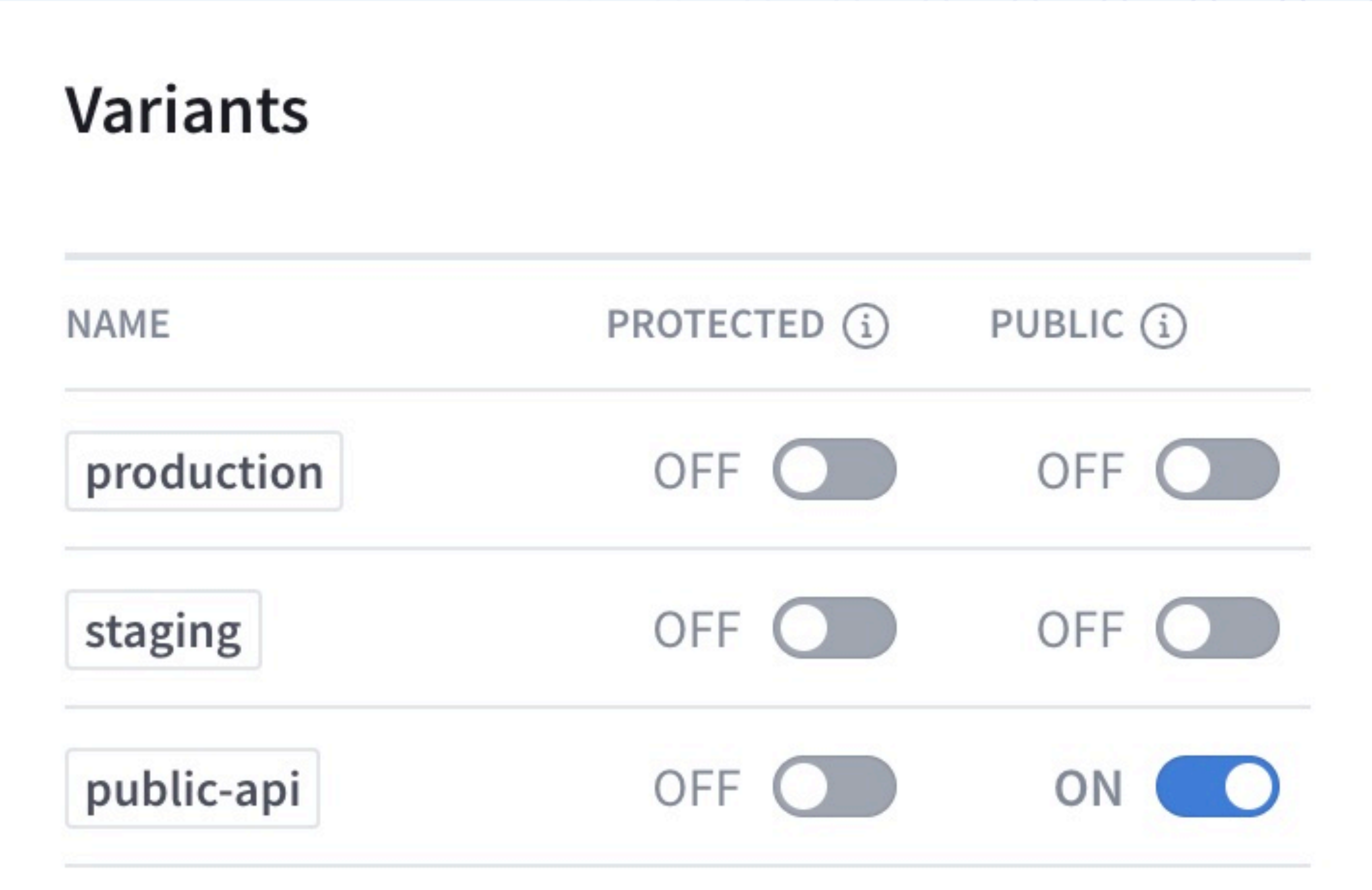

Settings | space-explorer@curr
studio.apollographql.com/graph/apollographql-8341/settings/access?variant=current
space-explorer/ current
Settings
General
Notifications
Access

Calvin Cestari	Consumer	-	✎
Caydie Tran	Observer	-	🔒
DannyDev	Consumer	-	🔒
cw	Observer	-	✎
Chris Shaw	Observer	-	🔒
Dan Boerner	Observer	-	🔒
matt debergalis	Org Admin	-	🔒
beccaapollo	Observer	-	🔒

Operations

#7. Managing graph access

- User roles
- Graph variants
  (public, private, protected)



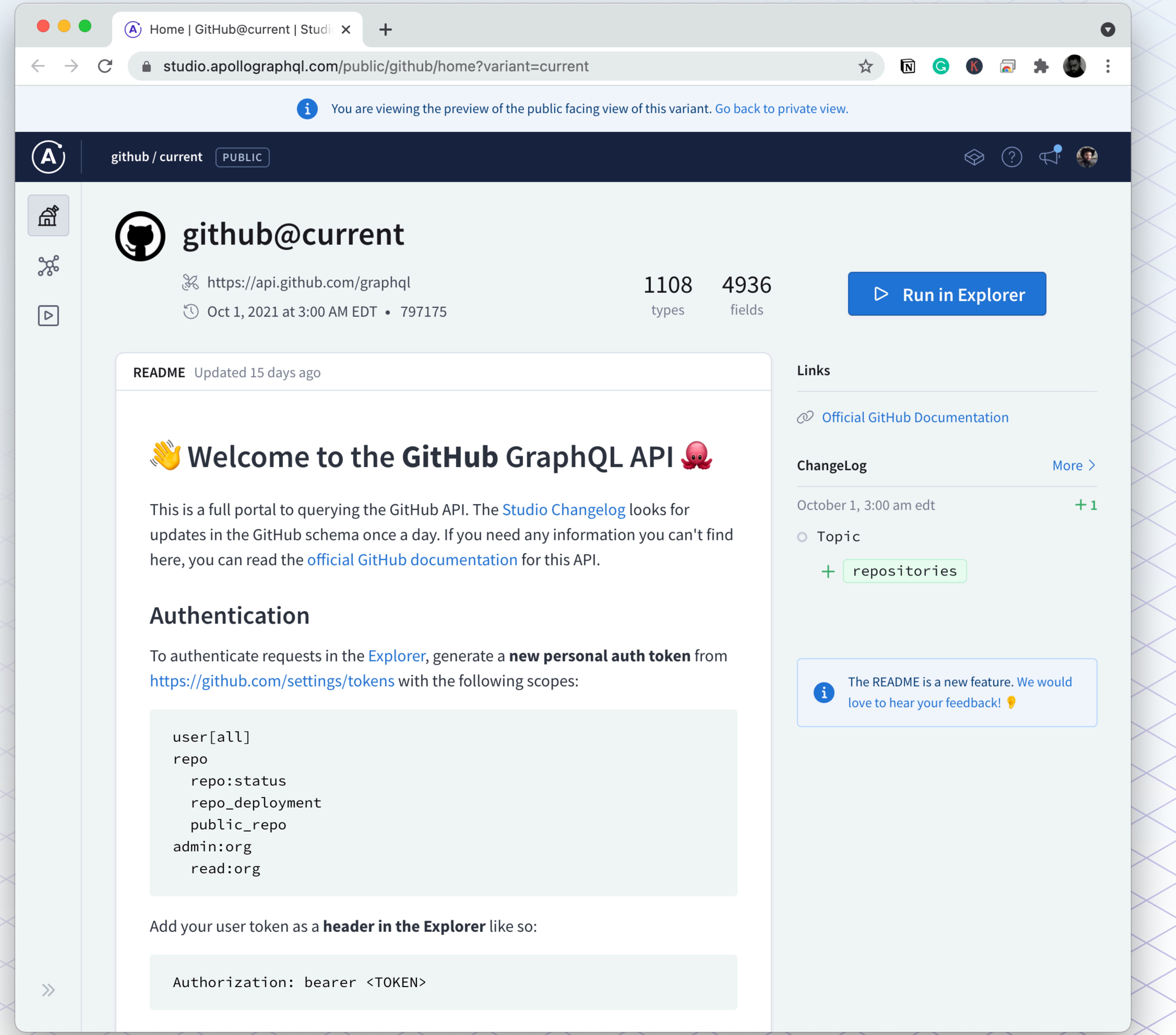
Variants

NAME	PROTECTED ⓘ	PUBLIC ⓘ
production	OFF	OFF
staging	OFF	OFF
public-api	OFF	ON

Operations

#7. Managing graph access

- User roles
- Graph variants (public, private, protected)
- Public readme page


Home | GitHub@current | Studi
studio.apollographql.com/public/github/home?variant=current
github / current
PUBLIC
Run in Explorer

You are viewing the preview of the public facing view of this variant. Go back to private view.

github@current

https://api.github.com/graphql

Oct 1, 2021 at 3:00 AM EDT • 797175

1108 types

4936 fields

README Updated 15 days ago

👋 Welcome to the GitHub GraphQL API 🐙

This is a full portal to querying the GitHub API. The Studio Changelog looks for updates in the GitHub schema once a day. If you need any information you can't find here, you can read the official GitHub documentation for this API.

Authentication

To authenticate requests in the Explorer, generate a **new personal auth token** from https://github.com/settings/tokens with the following scopes:

```
user[all]
repo
  repo:status
  repo_deployment
  public_repo
admin:org
  read:org
```

Add your user token as a **header in the Explorer** like so:

```
Authorization: bearer <TOKEN>
```

Links

Official GitHub Documentation

ChangeLog More >

October 1, 3:00 am edt +1

○ Topic

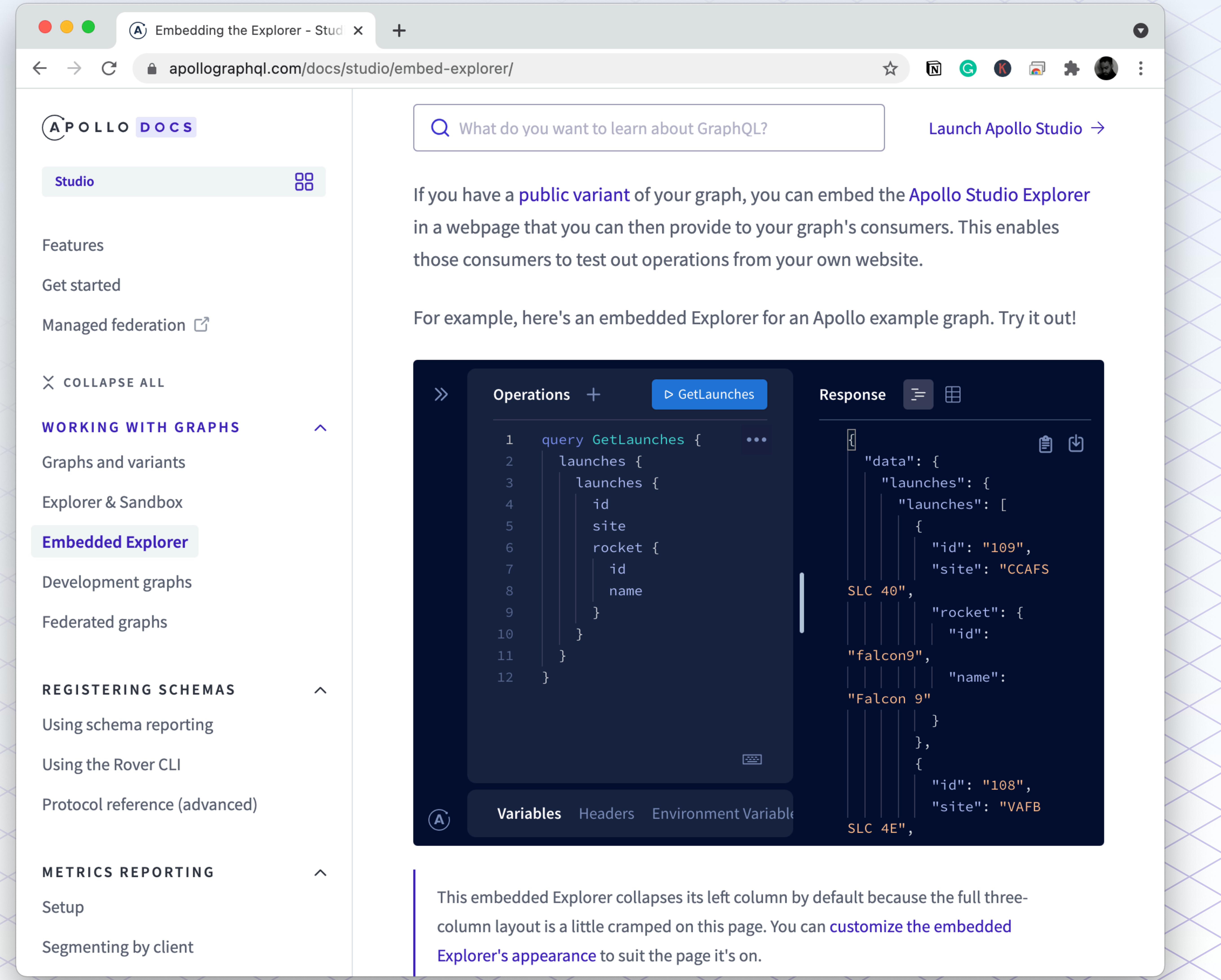
+ repositories

The README is a new feature. We would love to hear your feedback! 👂

Operations

#7. Managing graph access

- User roles
- Graph variants (public, private, protected)
- Public readme page
  - Embeddable explorer: public variants can also be embedded into your docs as an iframe



If you have a public variant of your graph, you can embed the Apollo Studio Explorer in a webpage that you can then provide to your graph's consumers. This enables those consumers to test out operations from your own website.

For example, here's an embedded Explorer for an Apollo example graph. Try it out!

```
query GetLaunches {
  launches {
    launches {
      id
      site
      rocket {
        id
        name
      }
    }
  }
}
```

```
{
  "data": {
    "launches": {
      "launches": [
        {
          "id": "109",
          "site": "CCAFS
SLC 40",
          "rocket": {
            "id":
"falcon9",
            "name":
"Falcon 9"
          }
        },
        {
          "id": "108",
          "site": "VAFB
SLC 4E",
```

> This embedded Explorer collapses its left column by default because the full three-column layout is a little cramped on this page. You can customize the embedded Explorer's appearance to suit the page it's on.

Operations

#8. Observability

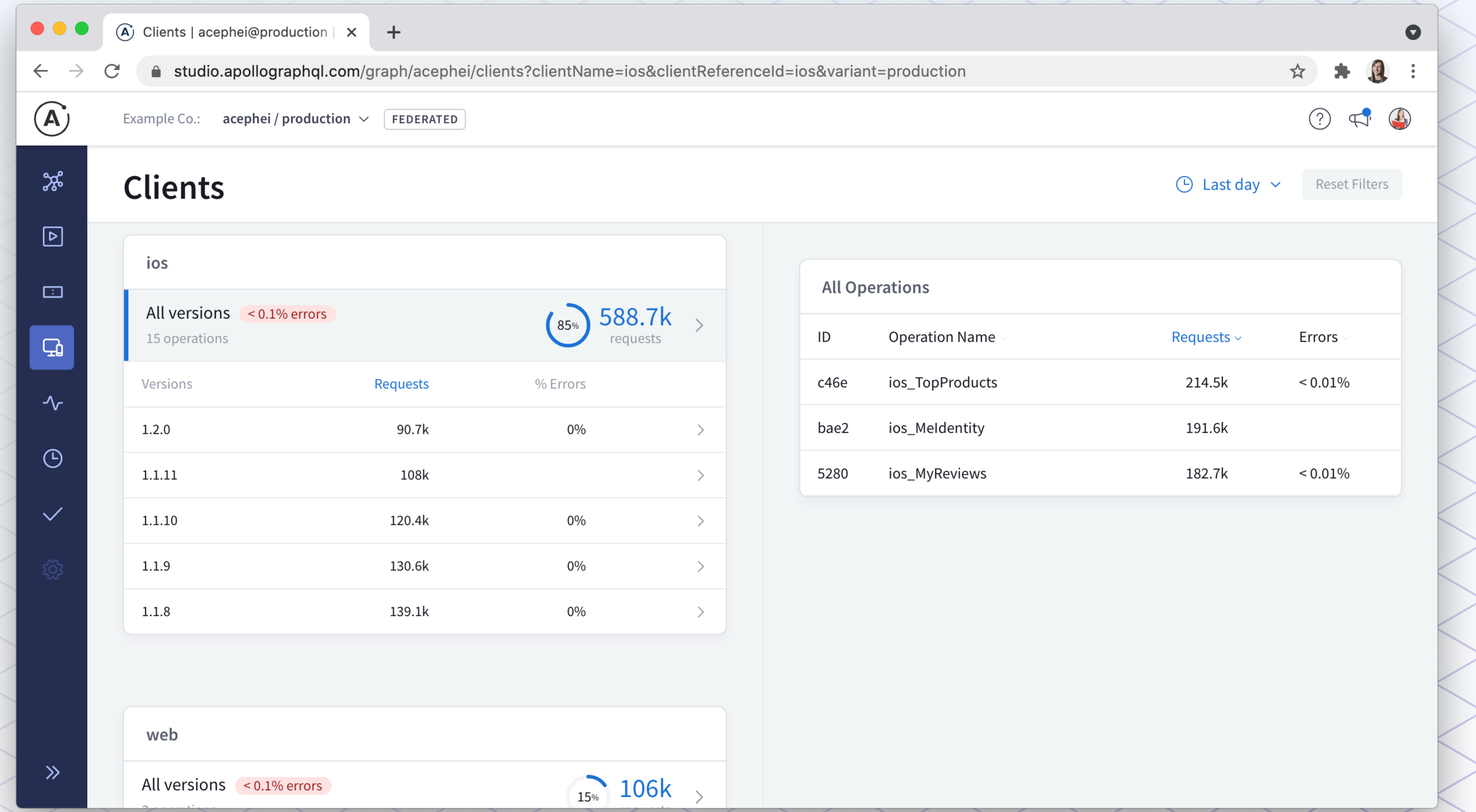
- There’s a lot going on in any production graph.
- We need a way to keep track of what’s going on.
- We can view our graph’s usage by the **org, client**, **field**, and **operation** level.

Operations

#8. Observability

→ *Client awareness*

- Know who is using your graph
  - See precisely which clients are querying your graph and what operations they're sending.
- Require clients to identify themselves and consistently name operations to enhance API usage understandability



Clients | acephei@production
studio.apollographql.com/graph/acephei/clients?clientName=ios&clientReferenceId=ios&variant=production
Example Co.:
acephei / production
FEDERATED

Clients

ios

All versions < 0.1% errors
15 operations
85% 588.7k requests

Versions	Requests	% Errors
1.2.0	90.7k	0%
1.1.11	108k	
1.1.10	120.4k	0%
1.1.9	130.6k	0%
1.1.8	139.1k	0%

web

All versions < 0.1% errors
15% 106k

All Operations

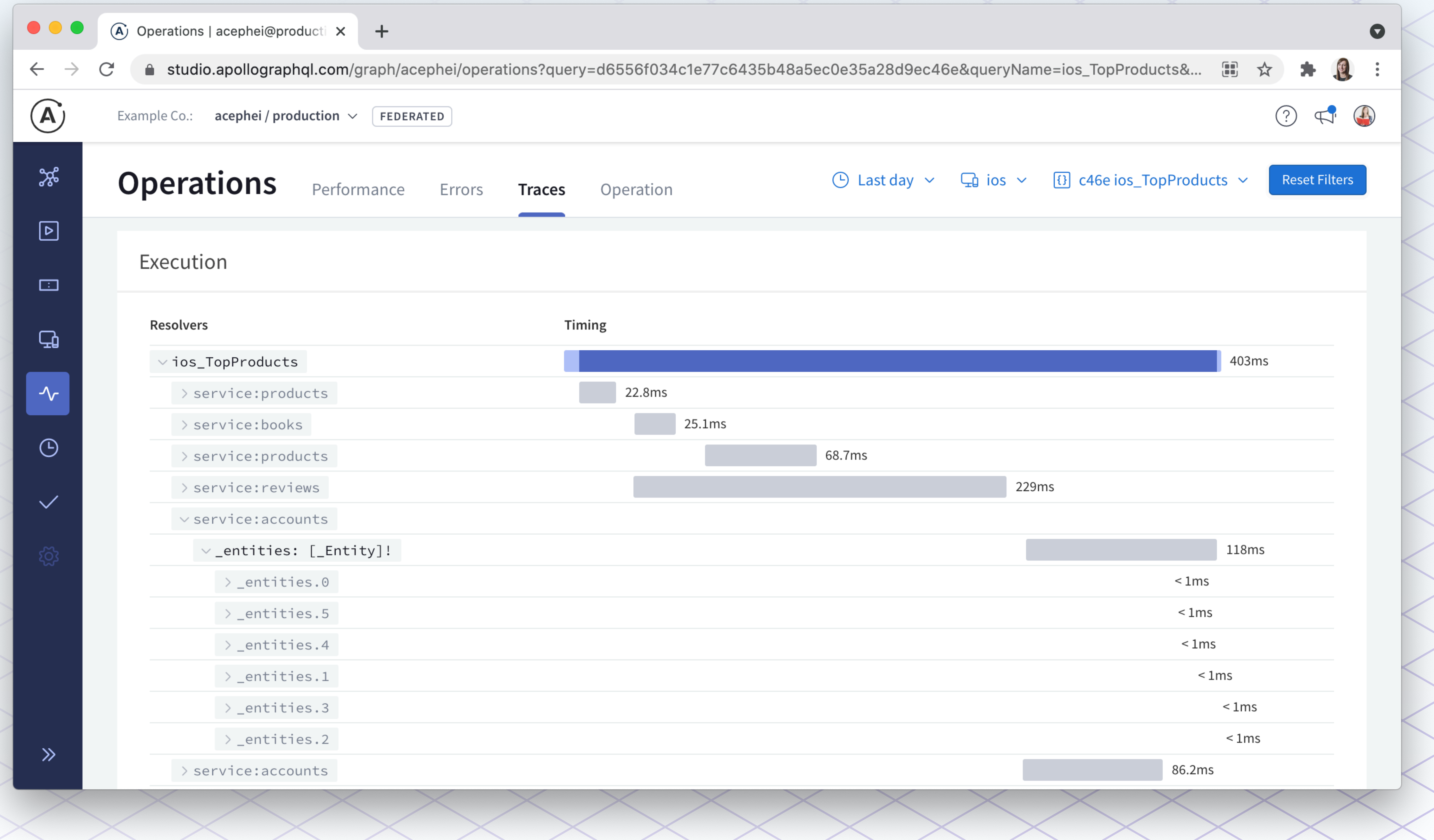
ID	Operation Name	Requests	Errors
c46e	ios_TopProducts	214.5k	< 0.01%
bae2	ios_MeIdentity	191.6k	
5280	ios_MyReviews	182.7k	< 0.01%

Operations

#8. Observability

→ *Field-level tracing*

- You can set up tracing as well for a detailed breakdown of the performance of your resolvers


Operations | acephei@producti
studio.apollographql.com/graph/acephei/operations?query=d6556f034c1e77c6435b48a5ec0e35a28d9ec46e&queryName=ios_TopProducts&...
Example Co.:
acephei / production
FEDERATED
Operations
Performance
Errors
Traces
Operation
Last day
ios
c46e ios_TopProducts
Reset Filters

Execution

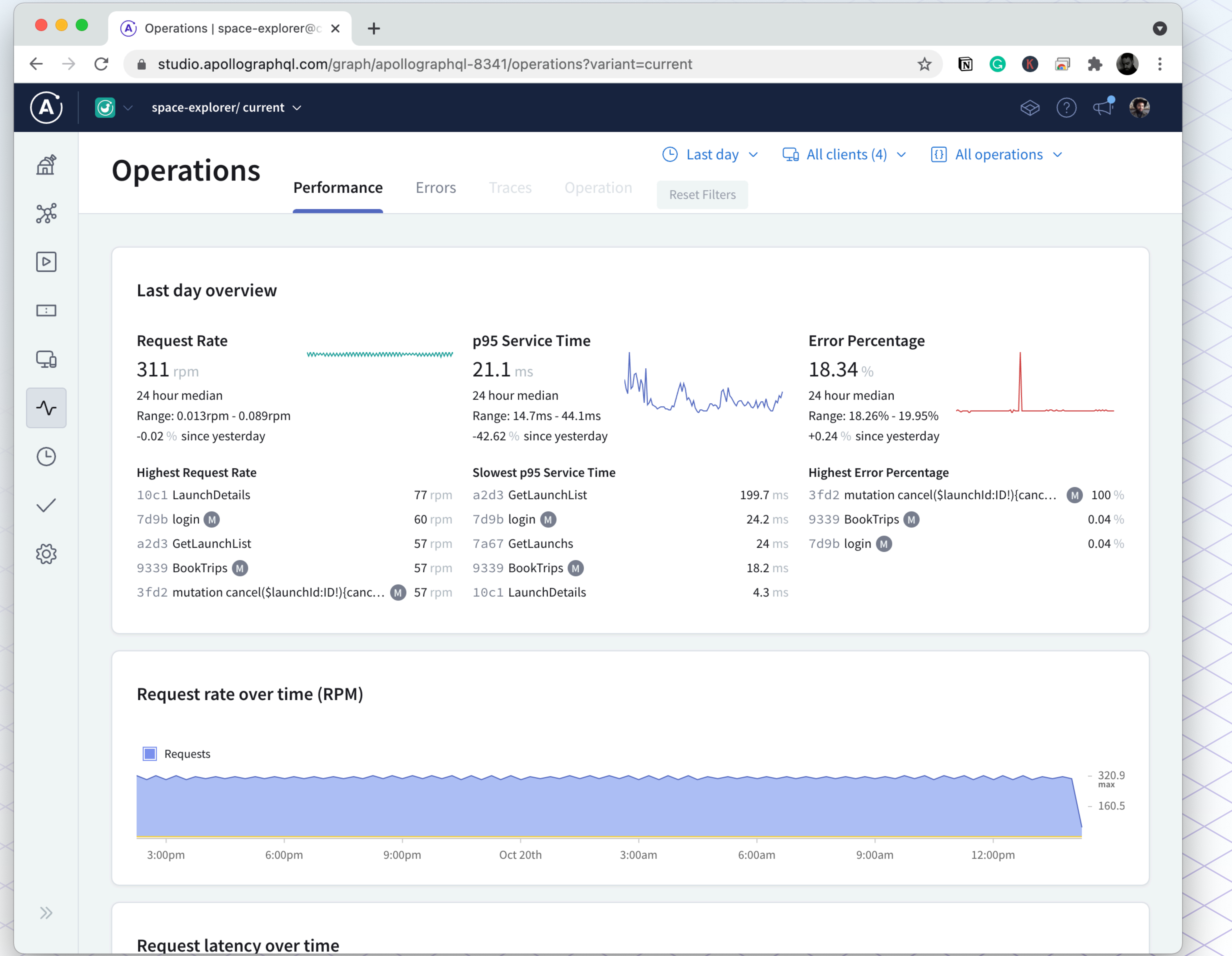
Resolvers	Timing
ios_TopProducts	403ms
service:products	22.8ms
service:books	25.1ms
service:products	68.7ms
service:reviews	229ms
service:accounts	
_entities: [_Entity]!	118ms
_entities.0	< 1ms
_entities.5	< 1ms
_entities.4	< 1ms
_entities.1	< 1ms
_entities.3	< 1ms
_entities.2	< 1ms
service:accounts	86.2ms

Operations

#9. Monitoring

→ *Performance alerts*

- We can also track performance degradations, and error spikes
- You can also set up alerts to be notified when something goes wrong:
  - Increase in requests per min
  - Change in p50, p95, p99 response time
  - Errors in operations


Operations | space-explorer@c
studio.apollographql.com/graph/apollographql-8341/operations?variant=current
space-explorer/ current
Last day
All clients (4)
All operations
Performance
Errors
Traces
Operation
Reset Filters

Operations

Last day overview

Request Rate
311 rpm
24 hour median
Range: 0.013rpm - 0.089rpm
-0.02 % since yesterday

p95 Service Time
21.1 ms
24 hour median
Range: 14.7ms - 44.1ms
-42.62 % since yesterday

Error Percentage
18.34 %
24 hour median
Range: 18.26% - 19.95%
+0.24 % since yesterday

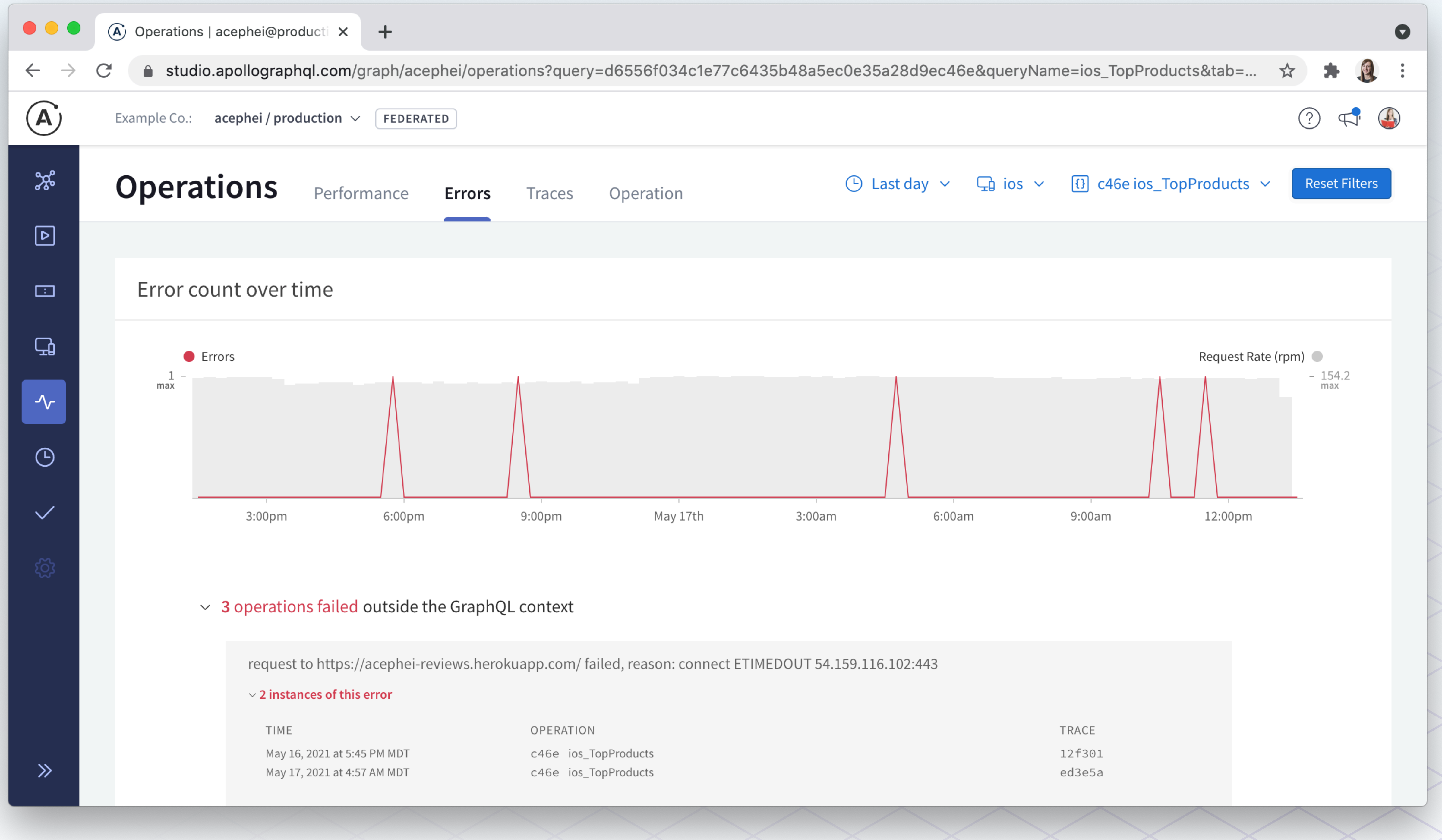
Highest Request Rate	
10c1 LaunchDetails	77 rpm
7d9b login M	60 rpm
a2d3 GetLaunchList	57 rpm
9339 BookTrips M	57 rpm
3fd2 mutation cancel($launchId:ID!){canc... M	57 rpm

Slowest p95 Service Time	
a2d3 GetLaunchList	199.7 ms
7d9b login M	24.2 ms
7a67 GetLaunchs	24 ms
9339 BookTrips M	18.2 ms
10c1 LaunchDetails	4.3 ms

Highest Error Percentage	
3fd2 mutation cancel($launchId:ID!){canc... M	100 %
9339 BookTrips M	0.04 %
7d9b login M	0.04 %

Request rate over time (RPM)

Request latency over time



3 operations failed outside the GraphQL context

request to https://acephei-reviews.herokuapp.com/ failed, reason: connect ETIMEDOUT 54.159.116.102:443

2 instances of this error

TIME	OPERATION	TRACE
May 16, 2021 at 5:45 PM MDT	c46e ios_TopProducts	12f301
May 17, 2021 at 4:57 AM MDT	c46e ios_TopProducts	ed3e5a

Key resources

Performance Alerts from Apollo Studio
via Apollo docs

Sending metrics to Apollo Studio
via Apollo docs

Segmenting metrics by client
via Apollo docs

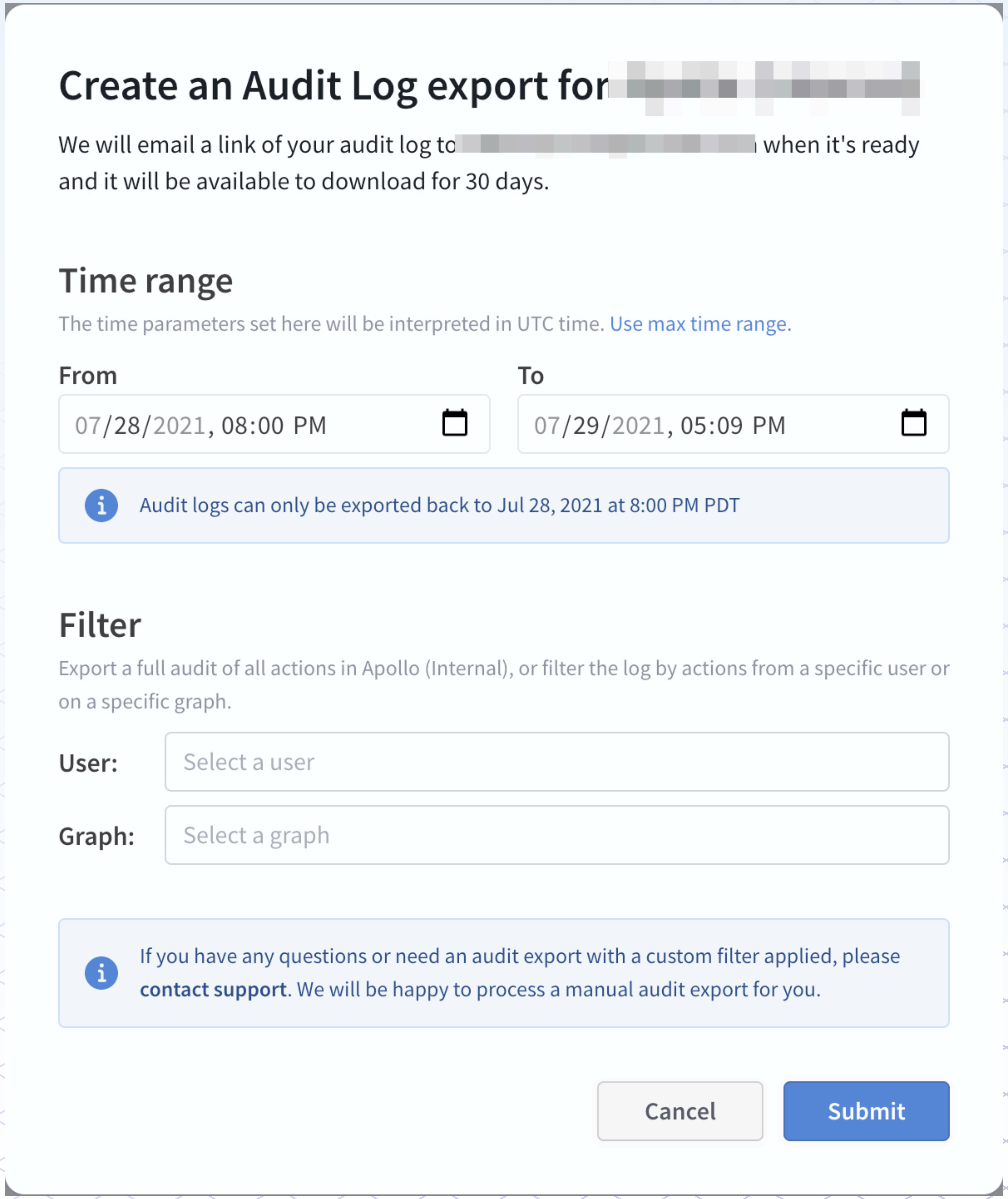
GraphQL Observability by Ashley Narcisse
GraphQL Galaxy Conference 2020

Operations

#8. Observability

→ *Audit logs*

- Export a data file with key actions taken within your organization
- Investigate an incident and see what actions lead up to that incident by exporting a log for a time period and graph
- See what actions an individual has taken within a time period
- Investigate your automated systems that are changing the graph



Create an Audit Log export for [redacted]

We will email a link of your audit log to [redacted] when it's ready and it will be available to download for 30 days.

Time range

The time parameters set here will be interpreted in UTC time. Use max time range.

From: 07/28/2021, 08:00 PM

To: 07/29/2021, 05:09 PM

Audit logs can only be exported back to Jul 28, 2021 at 8:00 PM PDT

Filter

Export a full audit of all actions in Apollo (Internal), or filter the log by actions from a specific user or on a specific graph.

User: Select a user

Graph: Select a graph

If you have any questions or need an audit export with a custom filter applied, please **contact support**. We will be happy to process a manual audit export for you.

In conclusion

In conclusion

We covered nine ways to secure your graph

In conclusion

We covered nine ways to secure your graph

Auth

1. Authentication
2. Authorization

In conclusion

We covered nine ways to secure your graph

Auth

1. Authentication
2. Authorization

Reducing the attack surface area

3. Mitigating malicious queries
4. Limiting API discoverability
5. Batched requests

In conclusion

We covered nine ways to secure your graph

Auth

1. Authentication
2. Authorization

Reducing the attack surface area

3. Mitigating malicious queries
4. Limiting API discoverability
5. Batched requests

Operations & Governance

6. Stability
7. Managing graph access
8. Observability
9. Monitoring

Thanks!

Chat w/ me @stemmlerjs