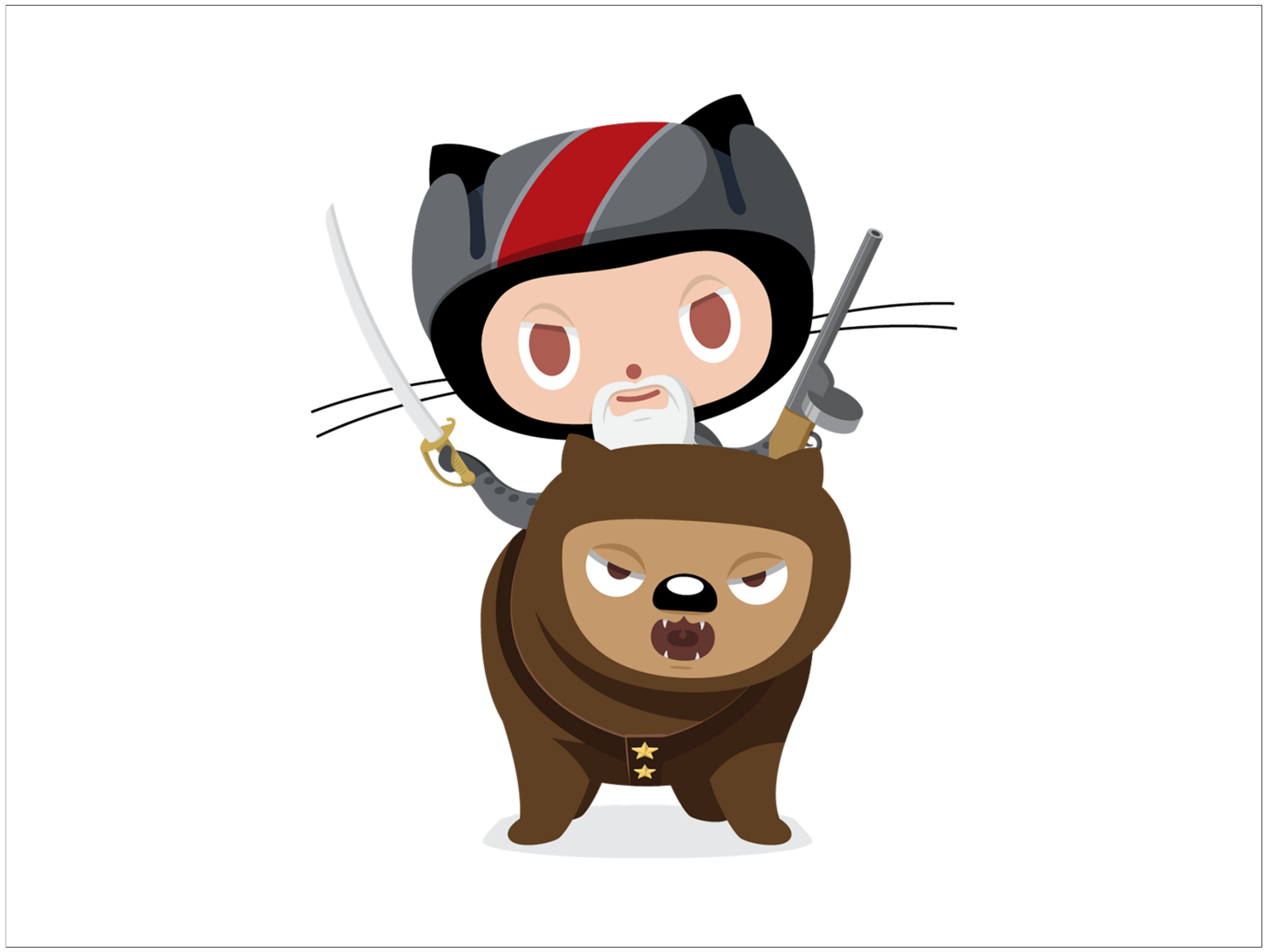
Getting down to business with GraphQL

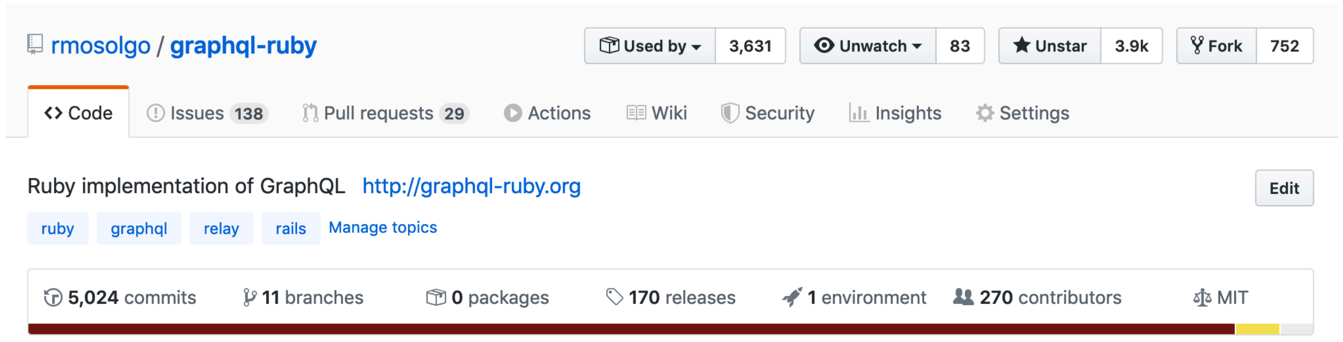
ACR 2019

@rmosolgo



I work at GitHub on the API team. The API team manages tooling, observability, and other support for GitHub's REST API and GraphQL API. I've been at GitHub for about 2.5 years.

I forgot to check our brand guidelines but I'm pretty sure the bear cavalry octocat was fair game. I hope so.


rmosolgo / graphql-ruby

Used by 3,631 · Unwatch 83 · Unstar 3.9k · Fork 752

Code · Issues 138 · Pull requests 29 · Actions · Wiki · Security · Insights · Settings

Ruby implementation of GraphQL http://graphql-ruby.org

Edit

ruby graphql relay rails Manage topics

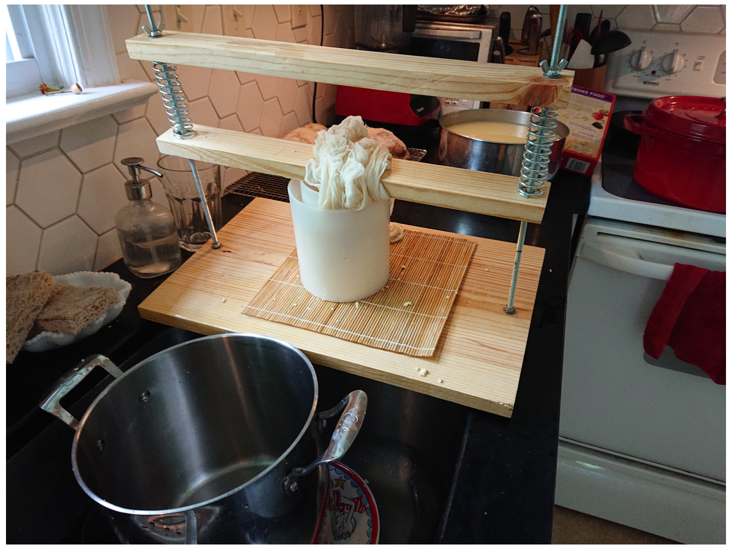
5,024 commits · 11 branches · 0 packages · 170 releases · 1 environment · 270 contributors · MIT

Besides that, I'm the maintainer of a Ruby library that implements GraphQL, more about that later. I guess you can see here that I starred my own repo. Trying to look popular I guess.

I live in Charlottesville, VA, where my wife and I are raising three daughters.

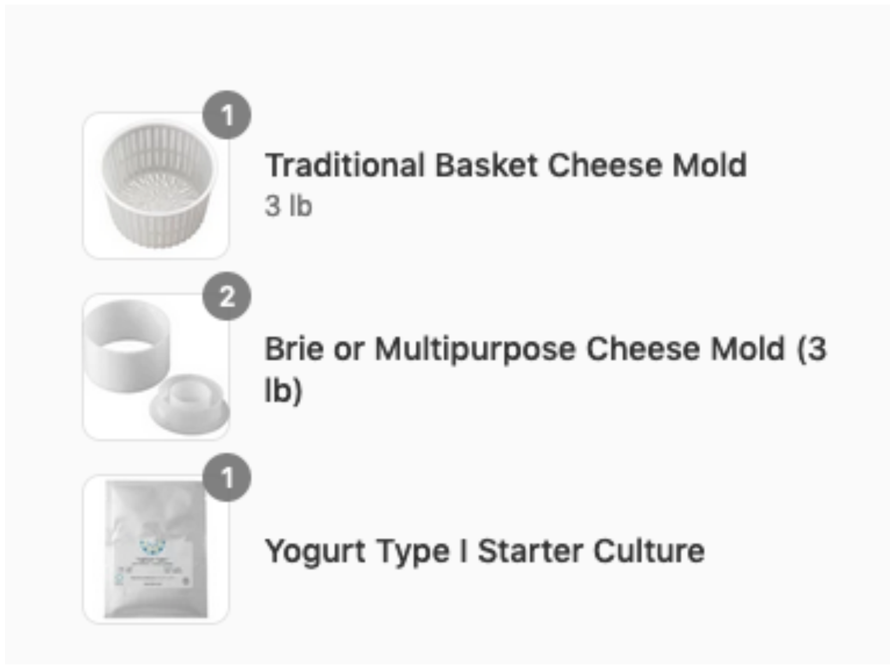
When I’m not hacking on APIs, I like making cheese at home. It’s a big mess, but I love cheese, and in general I find traditional food preservation really fascinating. Like a lot of traditional techniques, cheesemaking draws on a lot of different facets of food safety.

The first step is building acidity: that's why you can leave vinegar on the shelf without it spoiling. In cheesemaking, you add some good bacteria to eat the lactose in the milk and output lactic acid. This also prepares the proteins for the next step…



Next, you form the curd: with acidity and protein activity, you cause the proteins to undissolve from the milk, and you’re left with the storybook “curds” on the one hand: a protein matrix which has locked in the relatively large globs of milkfat; and whey on the other hand, which is mostly water with some other dissolved proteins. (They’re whey proteins — thanks to anyone who eats whey protein supplements!) Heat and pressure help. That’s my janky cheese press.

Finally, aging: the really slow fireworks show of cheesemaking! With high humidity and medium temperature, microbes continue their work very slowly. Besides that, they're dying and exploding and producing enzymes that break down the constituents of the cheese: fats are turned into fatty acids, etc. Most cheeses come into the cave very similar: similar water content, fat content, salt content, etc. But their slight differences become huge divergences because of how they continue to decompose, honestly, in the cave. Some of my favorite cheeses grow mold.



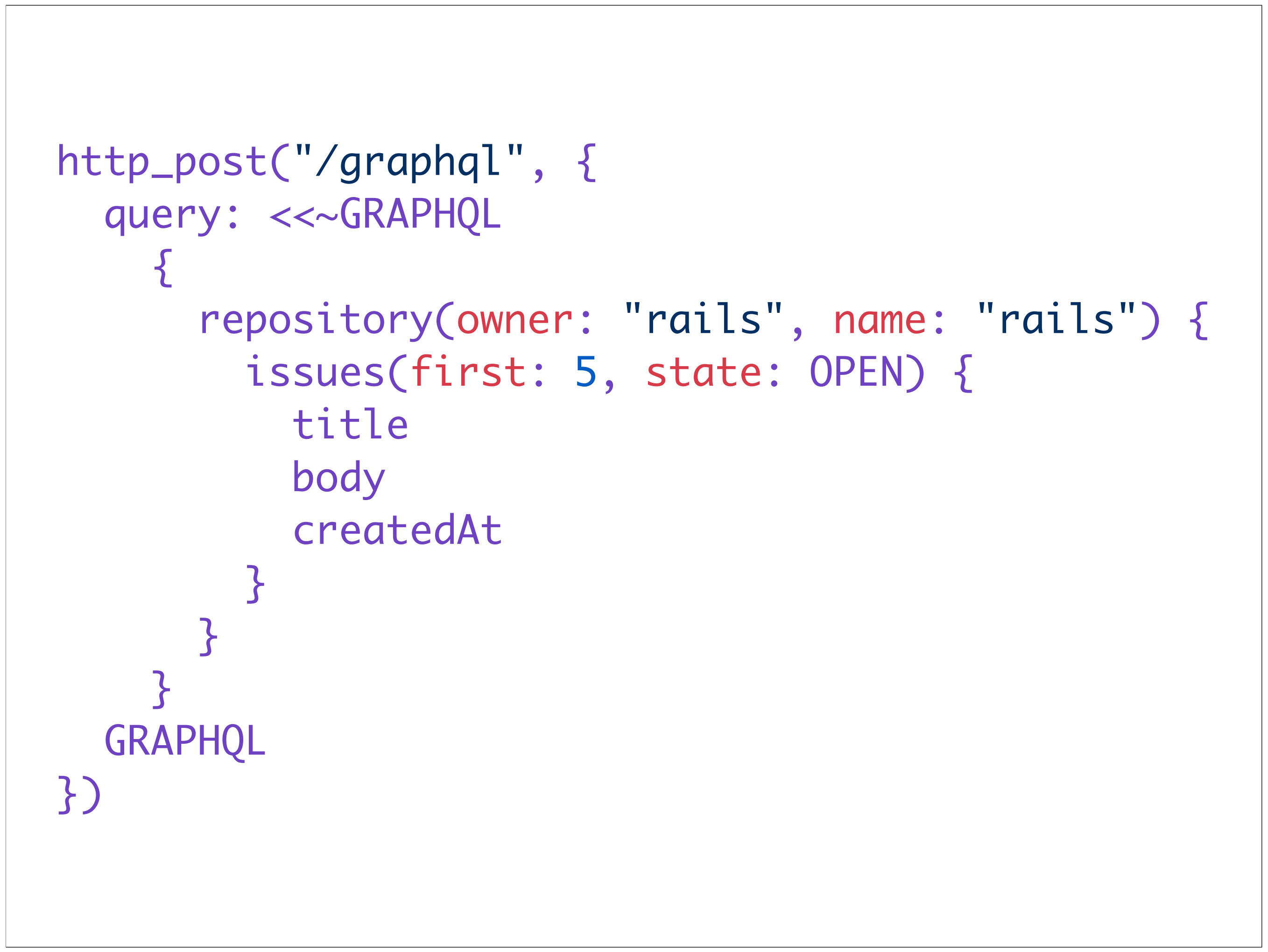
1	Traditional Basket Cheese Mold 3 lb
2	Brie or Multipurpose Cheese Mold (3 lb)
1	Yogurt Type I Starter Culture

Oh, here's my Shopify shout-out, apparently all my favorite home cheesemaking suppliers use Shopify. I always recognize the checkout flow. Rock on! I end up working with a bunch of folks from Shopify because of GraphQL stuff so I give them a nudge when I see it.

Ok, enough with my introduction by way of cheese. If you don't want to talk to me about GraphQL later, please come talk to me about dairy — and if you don't eat dairy, I'd love to know more about that too. What I want to share today is some new features of GraphQL-Ruby that we've been working on as part of our work at GitHub. But first, a quick introduction to GraphQL.

💁‍♂️ Welcome to GraphQL

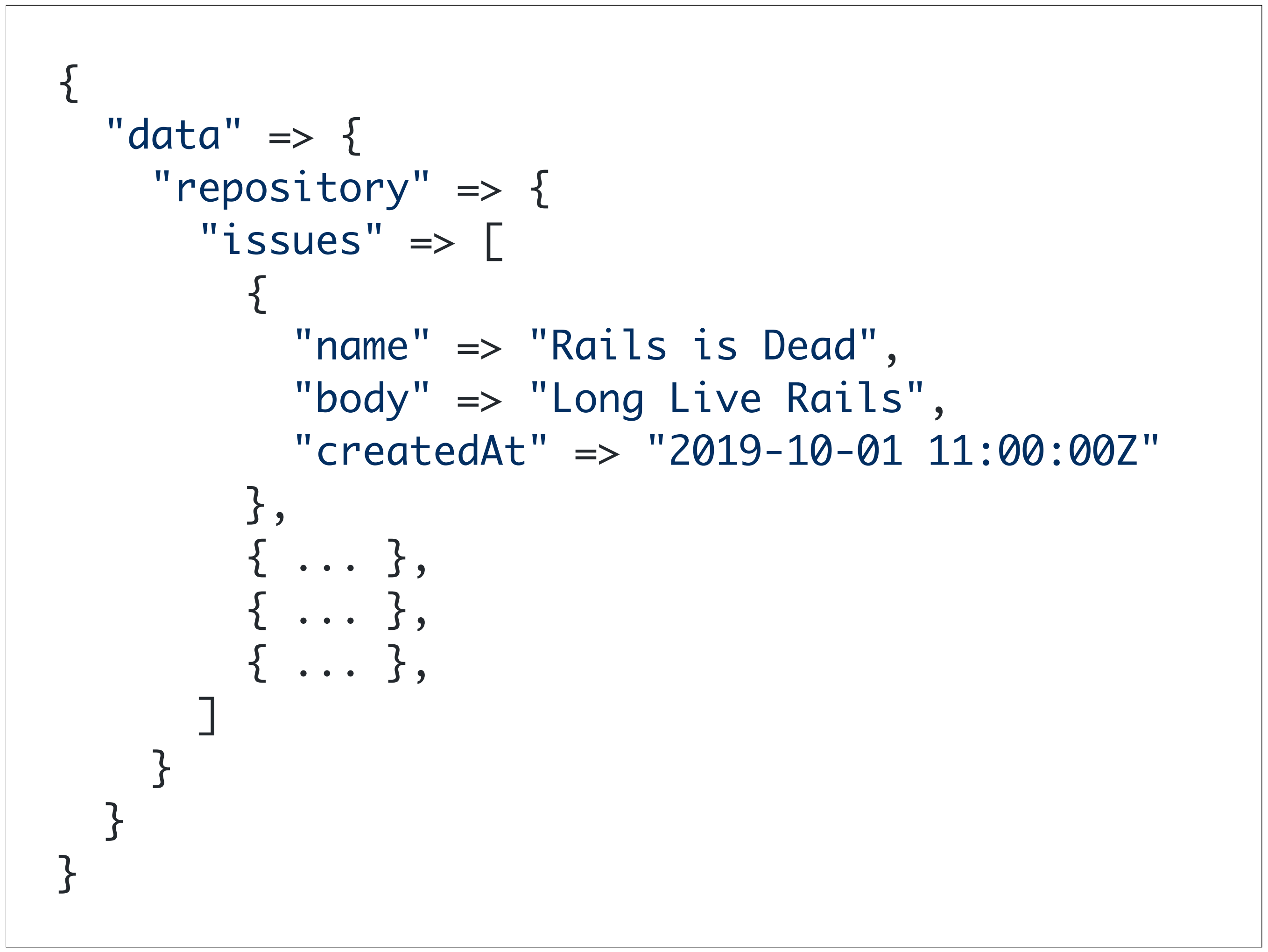
GraphQL is a “query language for your API”. Let’s break that down: it’s for APIs, so there’s gonna be an HTTP request in there. And it’s a query language, so there’s going to be a query, expressed in a language.



```
http_post("/graphql", {
  query: <<~GRAPHQL
    {
      repository(owner: "rails", name: "rails") {
        issues(first: 5, state: OPEN) {
          title
          body
          createdAt
        }
      }
    }
  GRAPHQL
})
```

Here's an imaginary call to someone's GraphQL API. Maybe it's GitHub's! This Ruby code sends an HTTP post to the `/graphql` endpoint, with "query" parameter containing a GraphQL query.

The thing to notice is that the query string is client-defined request for data — whatever the client wants — and it can bridge different objects and associations.

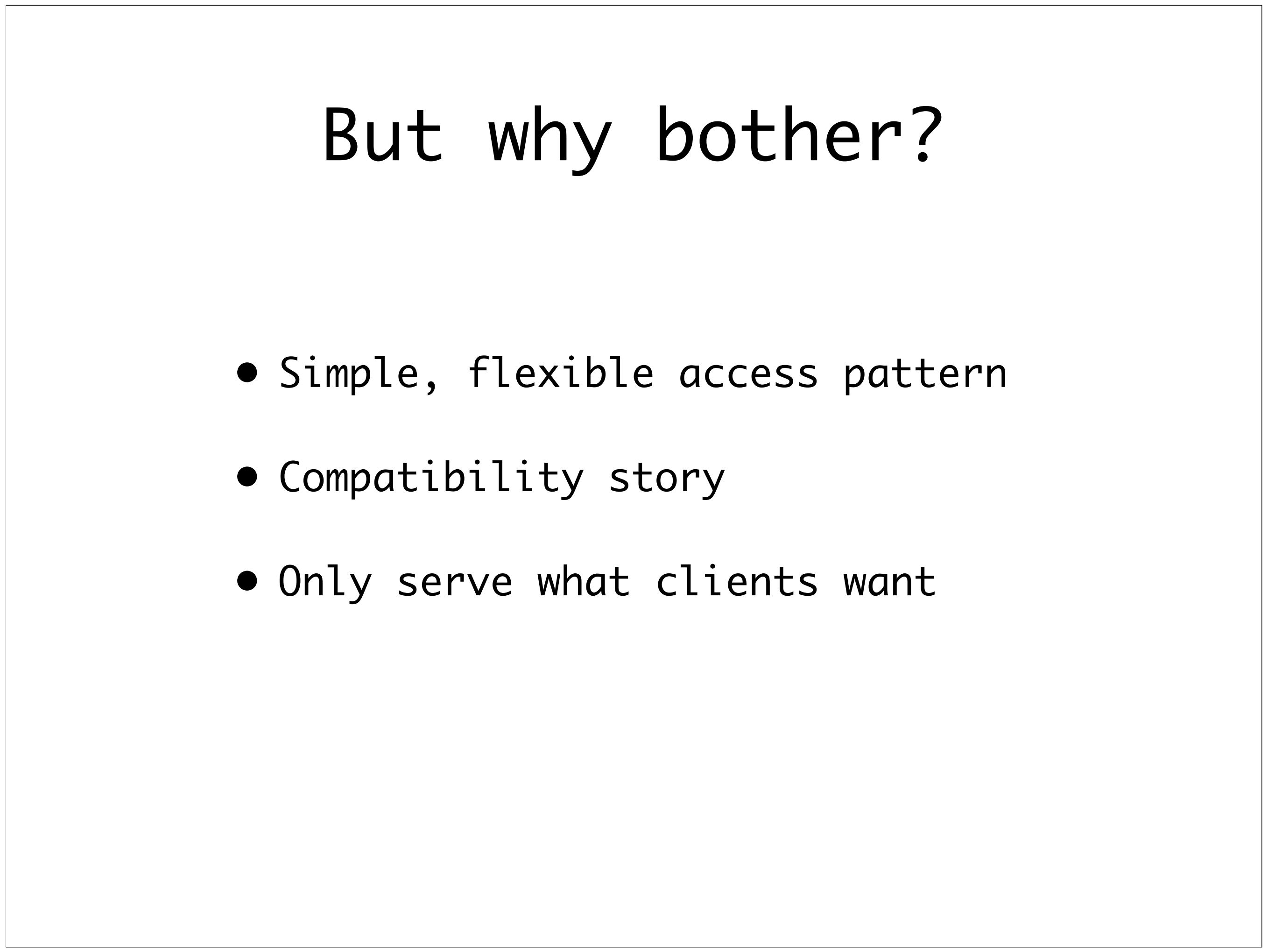


```
{
  "data" => {
    "repository" => {
      "issues" => [
        {
          "name" => "Rails is Dead",
          "body" => "Long Live Rails",
          "createdAt" => "2019-10-01 11:00:00Z"
        },
        { ... },
        { ... },
        { ... },
      ]
    }
  }
}
```

What GraphQL does is it evaluates this incoming query according to a schema you defined: a bunch of classes whose methods constitute your GraphQL API. Then, it returns the result of that evaluation to the caller. So it’s kind of like a database, except it doesn’t store data. Instead, it queries a system that *you define.*

```ruby
class Types::Repository < Types::BaseObject
  field :name, String, null: false

  field :issues, [Types::Issue], null: false do
    argument :first, Integer, required: true
    argument :state, Types::IssueState,
      required: false,
      default_value: :open
  end

  def issues(first:, state:)
    object.issues
      .where(state: state)
      .limit(first)
  end
end
```

Here's what the implementation of that part of the schema might look like: it's a Ruby class that usually maps to one of your models. It might contain glue code that translates GraphQL inputs to match your application logic.



But why bother?

- Simple, flexible access pattern
- Compatibility story
- Only serve what clients want

You’re probably thinking, very fancy, but why would you *want* that? Here’s where GraphQL shines: when you have client applications, like mobile apps or partners that integrate with your system, who need really flexible data access, so they can help themselves, and clearly-defined compatibility. Along with a predefined schema, GraphQL also has rules for how that schema can change over time. That’s where GraphQL shines at GitHub: integrators who want to request their data in specific ways, without lots of round-trips to get related objects.

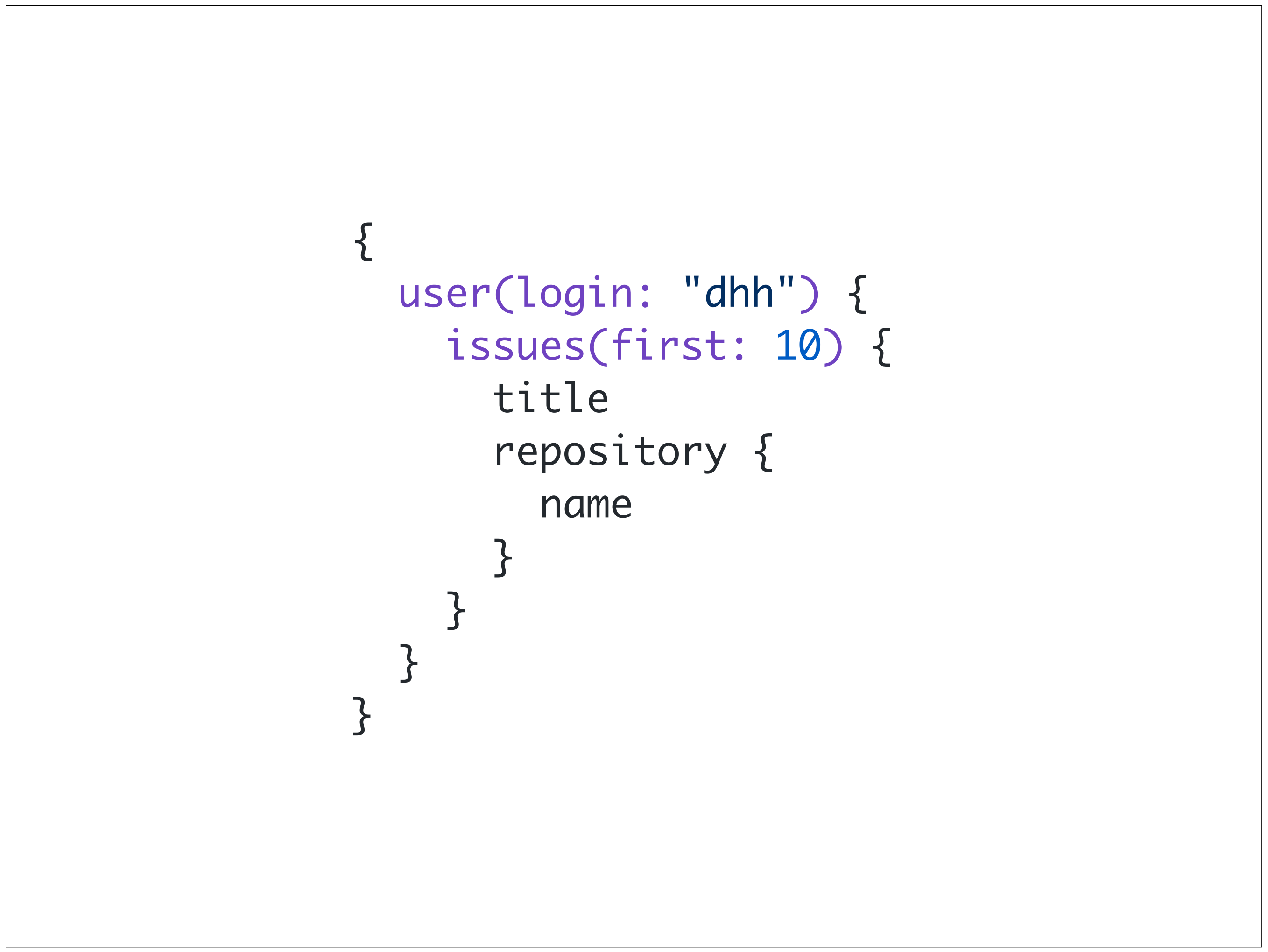
Why not?

- Runtime overhead
- Complexity on the server-side
- Unfamiliar

With that in mind, here are a few things we've been up to GraphQL-wise at GitHub.

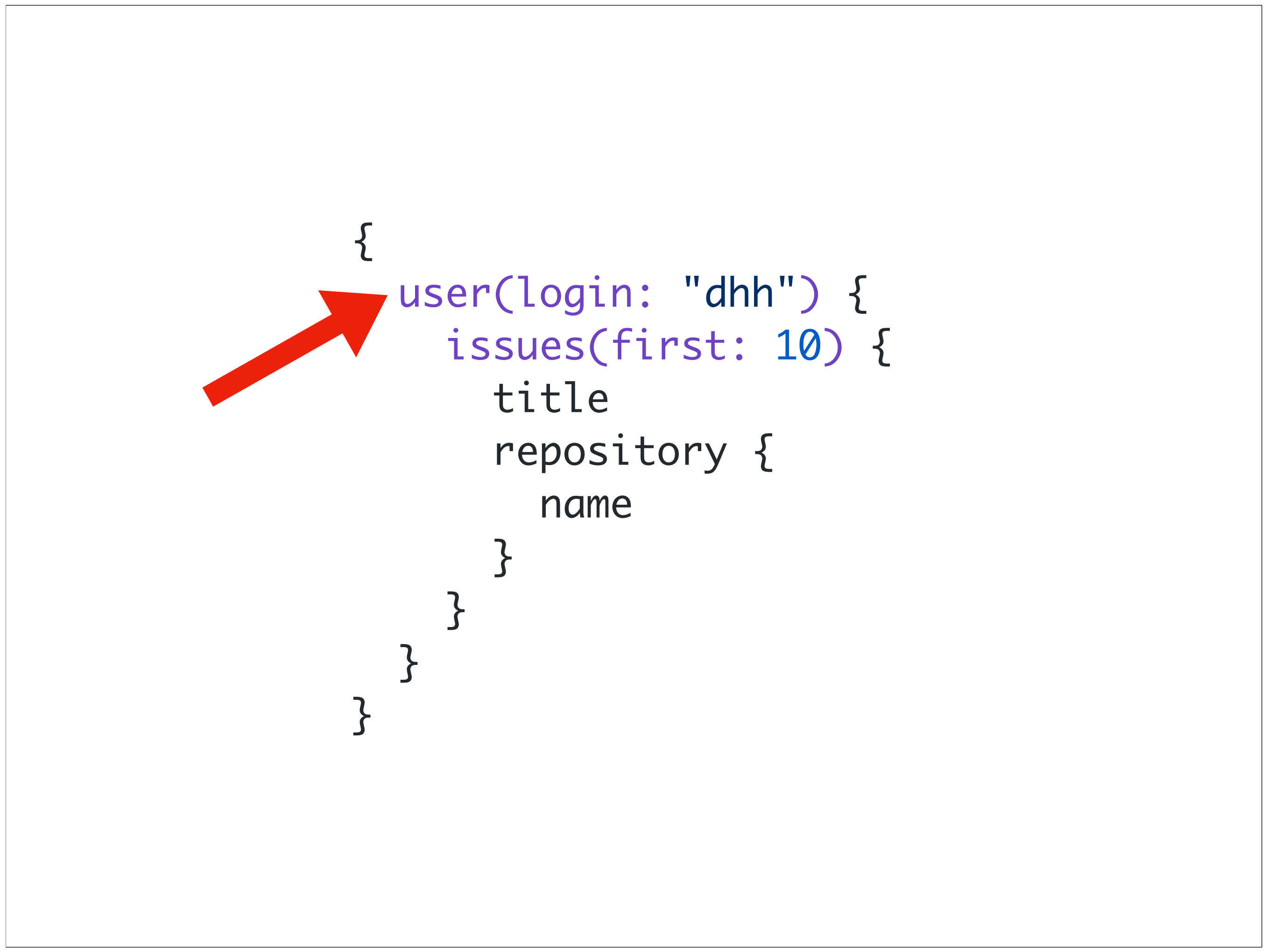
Authorization 🔒

The first one is authorization. We had something that worked pretty well in GitHub, but by building it into the GraphQL runtime, we could make it simpler and more reliable.



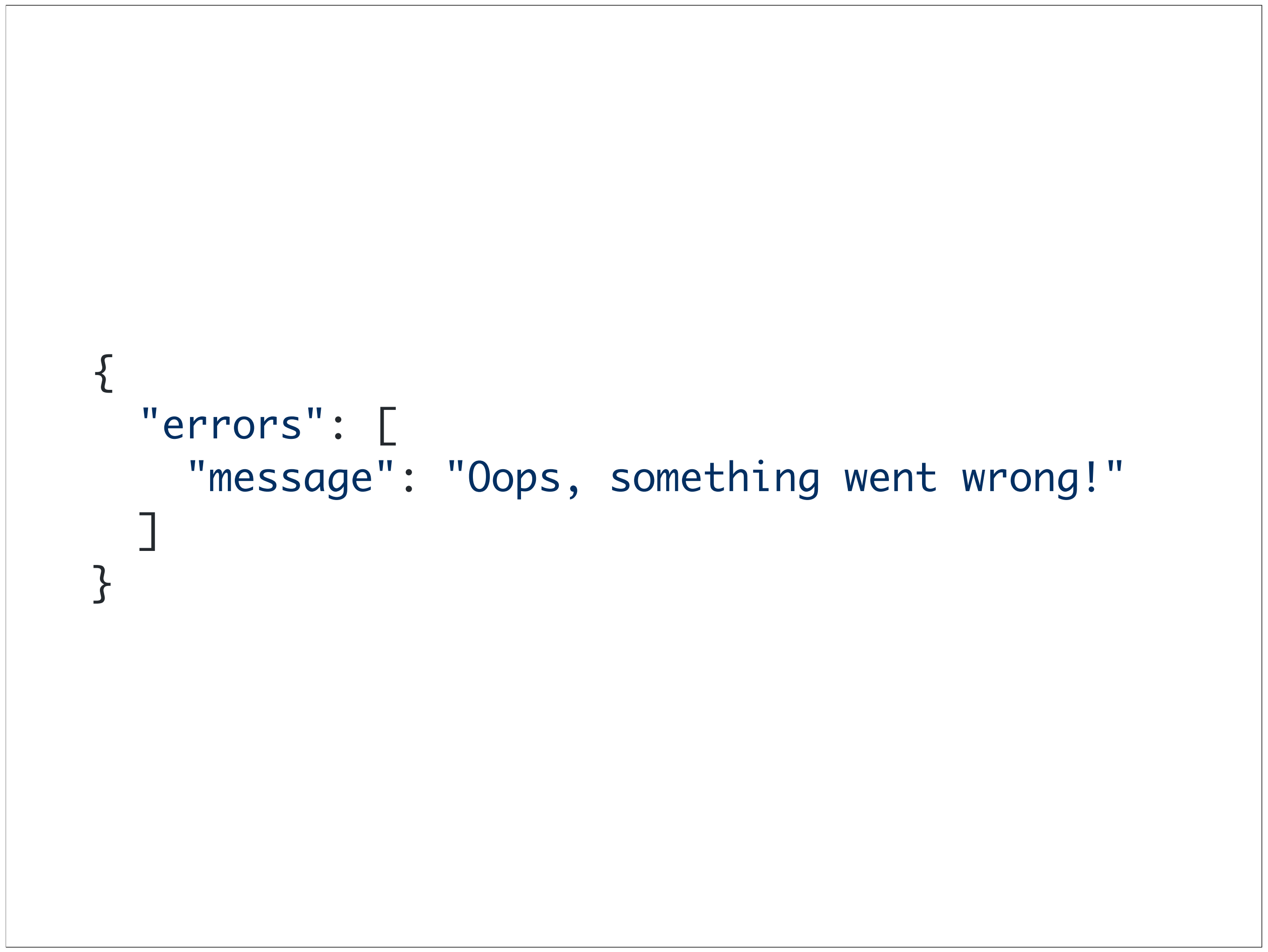
```
{
  user(login: "dhh") {
    issues(first: 10) {
      title
      repository {
        name
      }
    }
  }
}
```

So, what’s the problem here? Well, someone sends you a GraphQL query to run, you have to make sure you don’t give them anything they shouldn’t see. And this is tricky because, how do you know if a query will *touch* something that someone can’t see? You have to run it first.



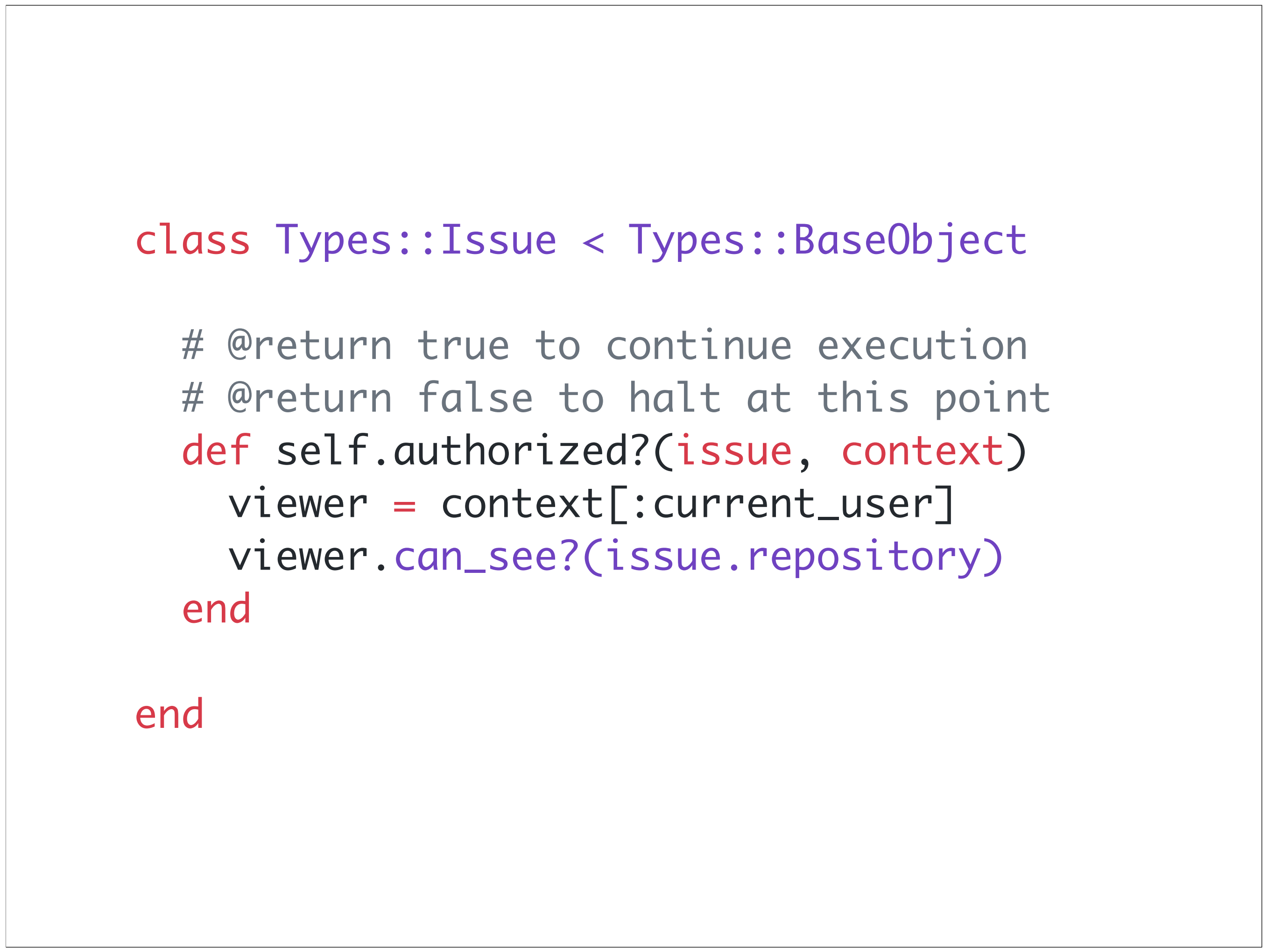
```
{
  user(login: "dhh") {
    issues(first: 10) {
      title
      repository {
        name
      }
    }
  }
}
```

So the approach we took happens at runtime. One of GraphQL's primitives is called an "object", and it's a thing with fields. So, after we *load* an object, but before we start evaluating its fields, we run an authorization hook. So we'd authorize this user, then we'd load some repositories, and authorize the first one, then this repository, then the next issue, and so on — checking *each object* before we actually return any data.



```
{
  "errors": [
    "message": "Oops, something went wrong!"
  ]
}
```

When one of these checks fail, we actually bomb the whole query. We *could* return a partial result, but we don't want to tell clients, “there's a repo here that you can't see” because that might disclose someone's super-secret project. When that happens, we get a bug in the error tracker, and whoever owns that feature looks back and the application code to see how it could be better filtered *before* reaching the authorization checks.



```ruby
class Types::Issue < Types::BaseObject

  # @return true to continue execution
  # @return false to halt at this point
  def self.authorized?(issue, context)
    viewer = context[:current_user]
    viewer.can_see?(issue.repository)
  end

end
```

So, if you want to try this at home, you can download graphql-ruby and start adding `authorized?` hooks in your object classes! If they return false, then execution doesn't proceed there. You can also customize the handling of unauthorized objects, the same way we do at GitHub. This has been in GraphQL-Ruby for about a year and a half now.

AuthZ at GitHub

- OAuth Scopes
- GitHub App permissions
- Granted user permissions
- Application checks

In practice, what do we do in these hooks? Since it's a public API, this is where we make sure that the client has the right OAuth scopes. We do some of this ahead-of-time, but in fact, we have different scopes for *public* and *private* repos, and we don't know if a repo is public or private until after we load it from the database. So we can't check that based on the query string alone. Similar for GitHub app permissions. And there really end up being two kinds of authorization checks: on the one hand, we have explicitly granted permissions, and on the other hand, we have some authorization that's based on *other* database state. For example, if you invite me to collaborate on a repository and I haven't accepted it yet, I don't have an explicitly granted permission, but I can still see it.

Overview

Here's a conceptual approach to GraphQL authorization, followed by an introduction to the built-in authorization framework. Each part of the framework is described in detail in its own guide.

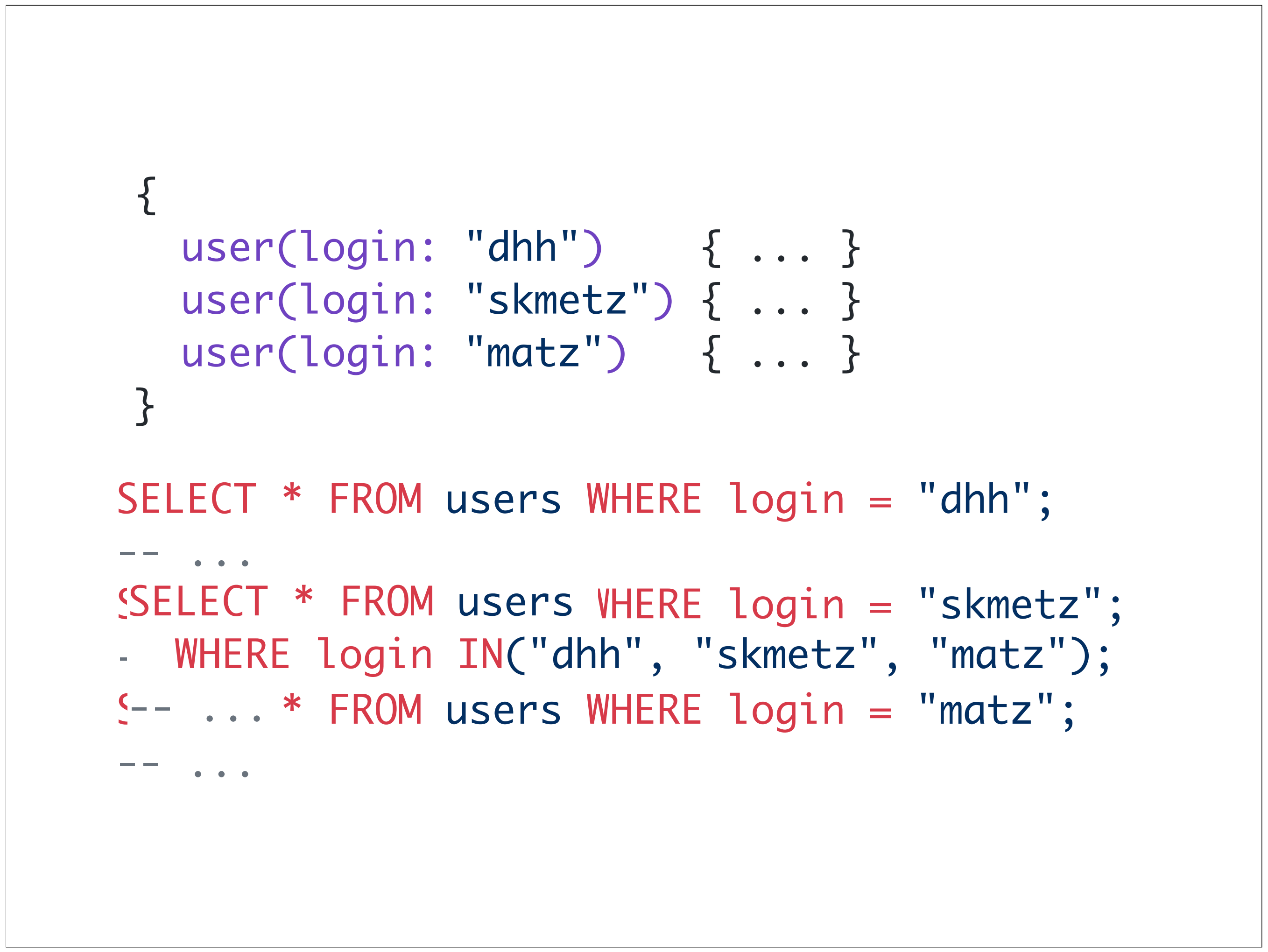
Authorization: GraphQL vs REST

In a REST API, the common authorization pattern is fairly simple. Before performing the requested action, the server asserts that the current client has the required permissions for that action. For example:

```
class PostsController < ApiController
  def create
    # First, check the client's permission level:
    if current_user.can?(:create_posts)
```

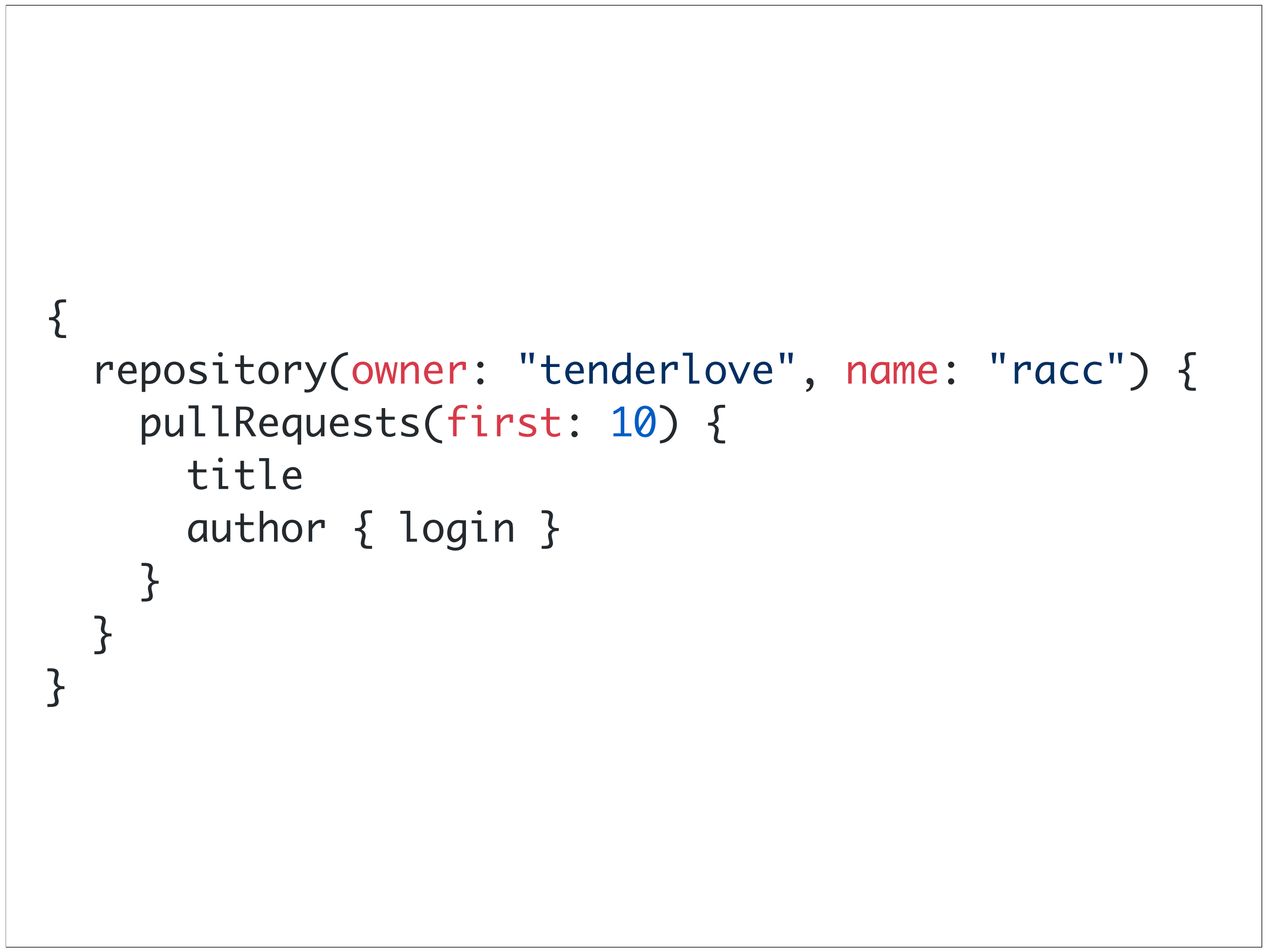
Batching 🥞

Here’s another really key area for performance and stability that we’re revisiting in GraphQL-Ruby. Basically, whether you’re rendering a Rails view or responding to a REST API request, it’s pretty straightforward to make efficient database queries to render the response. But a downside of GraphQL’s client-friendly flexibility is its unpredictability on the server side.



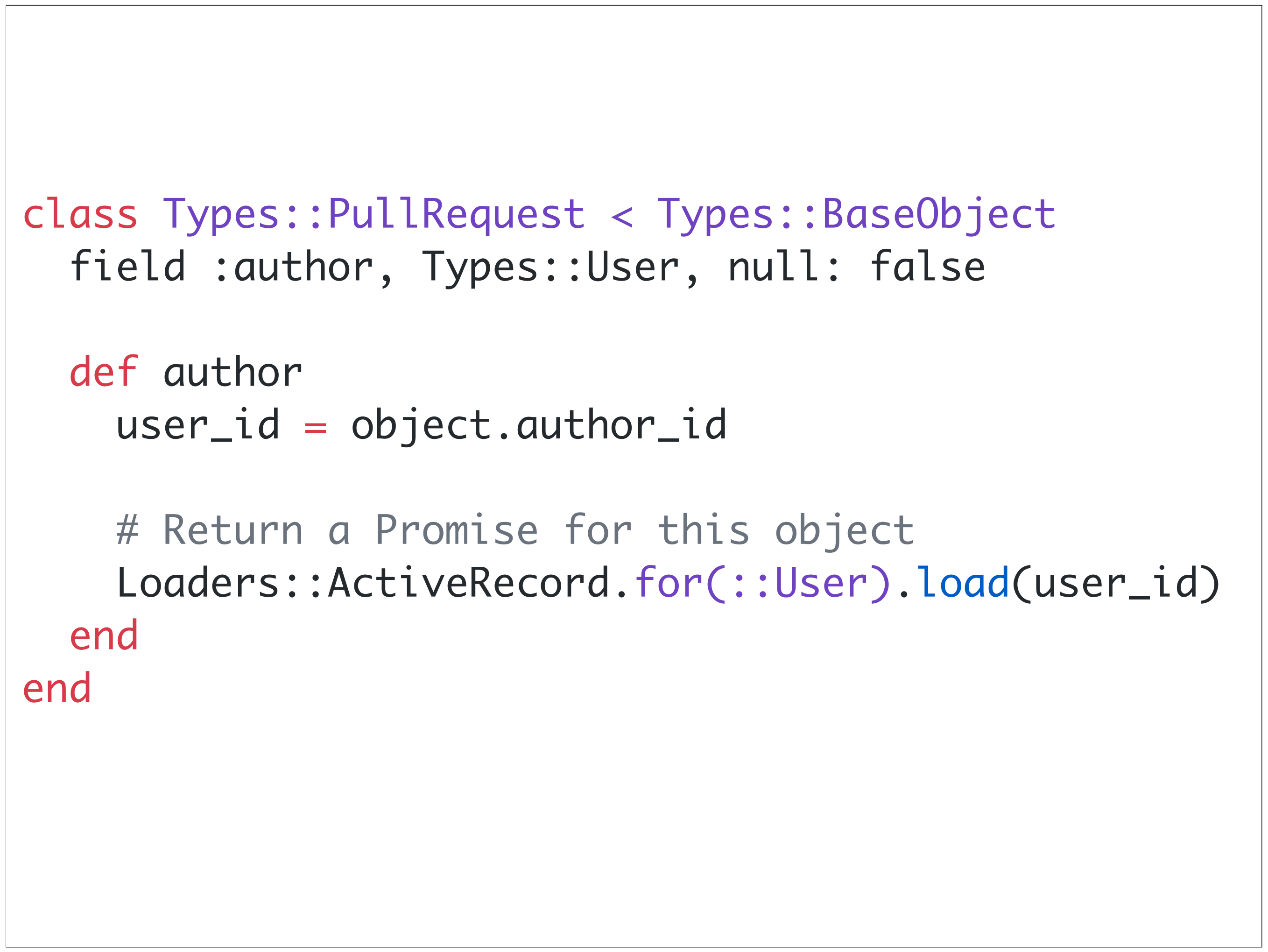
```
 {
   user(login: "dhh")    { ... }
   user(login: "skmetz") { ... }
   user(login: "matz")   { ... }
 }

SELECT * FROM users WHERE login = "dhh";
-- ...
SELECT * FROM users WHERE login = "skmetz";
-  WHERE login IN("dhh", "skmetz", "matz");
S-- ... * FROM users WHERE login = "matz";
-- ...
```

Here's an example where the simplest implementation results in an N+1 query situation. Loading each user by name will cause three round trips to the database. Can you imagine a way to make this more efficient? Well, we could look ahead, find the user logins, and prepare a *single* SQL query to load the data that we need.



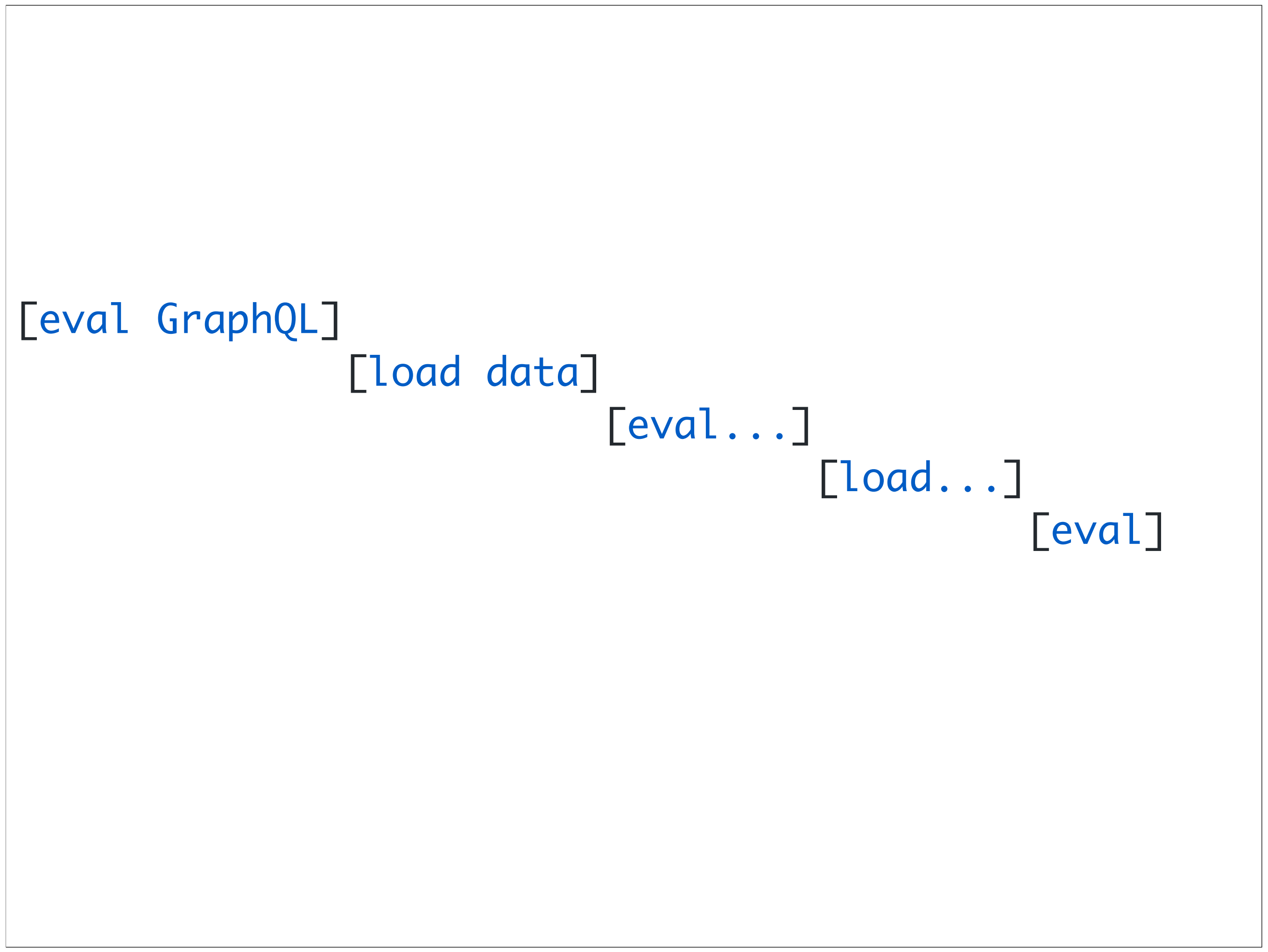
```
{
  repository(owner: "tenderlove", name: "racc") {
    pullRequests(first: 10) {
      title
      author { login }
    }
  }
}
```

But here’s a tougher example. We want to load a *lot* of different users, and it’s not clear ahead of time which users we’re going to need. We have to start *running* the query to figure out what to load from the database.



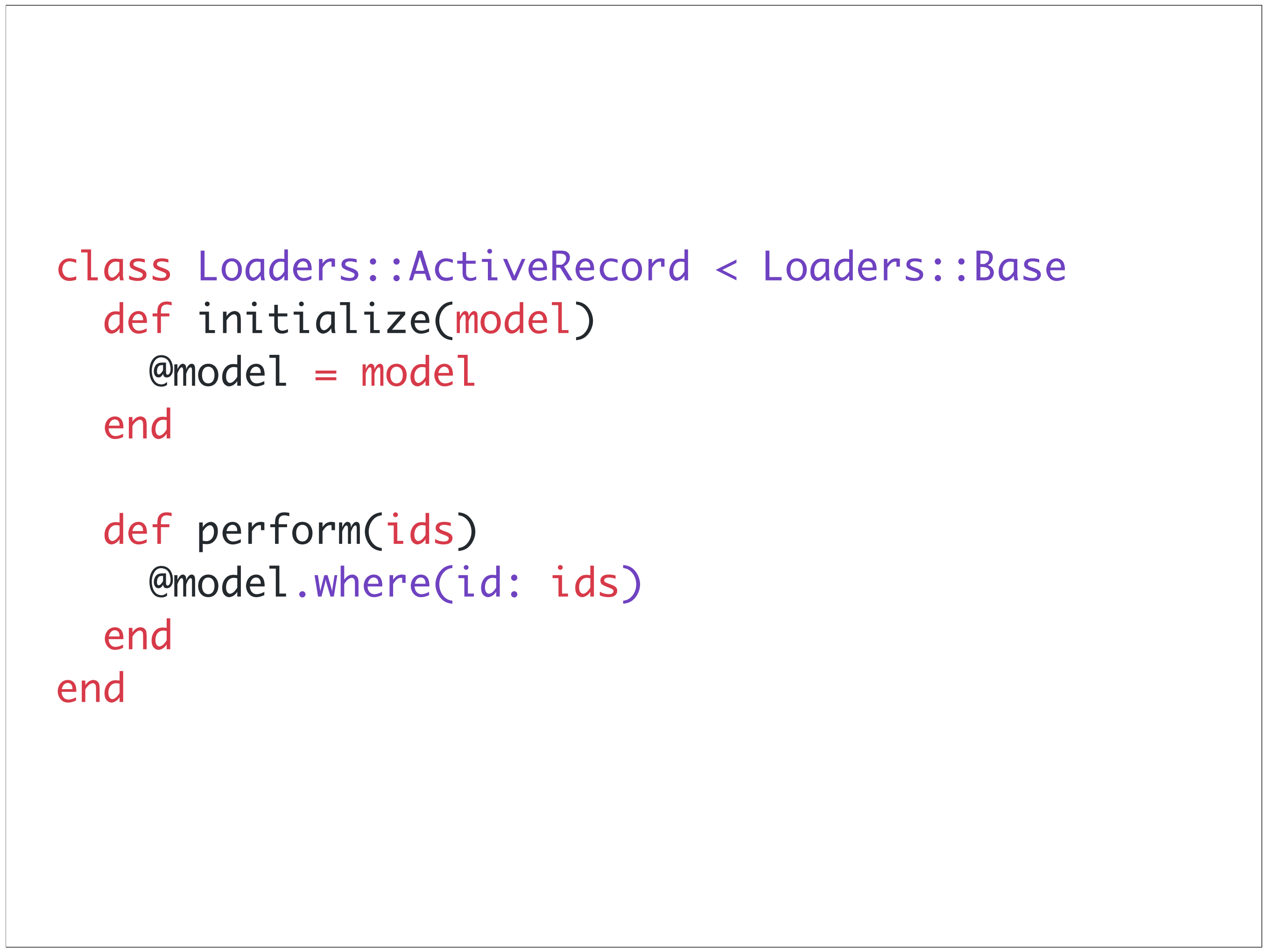
```ruby
class Types::PullRequest < Types::BaseObject
  field :author, Types::User, null: false

  def author
    user_id = object.author_id

    # Return a Promise for this object
    Loaders::ActiveRecord.for(::User).load(user_id)
  end
end
```

Here's what a solution looks like under the hood: the `author` field *doesn't* return a User. Instead, it registers a request for a user object and tells GraphQL to wait until that object is ready.



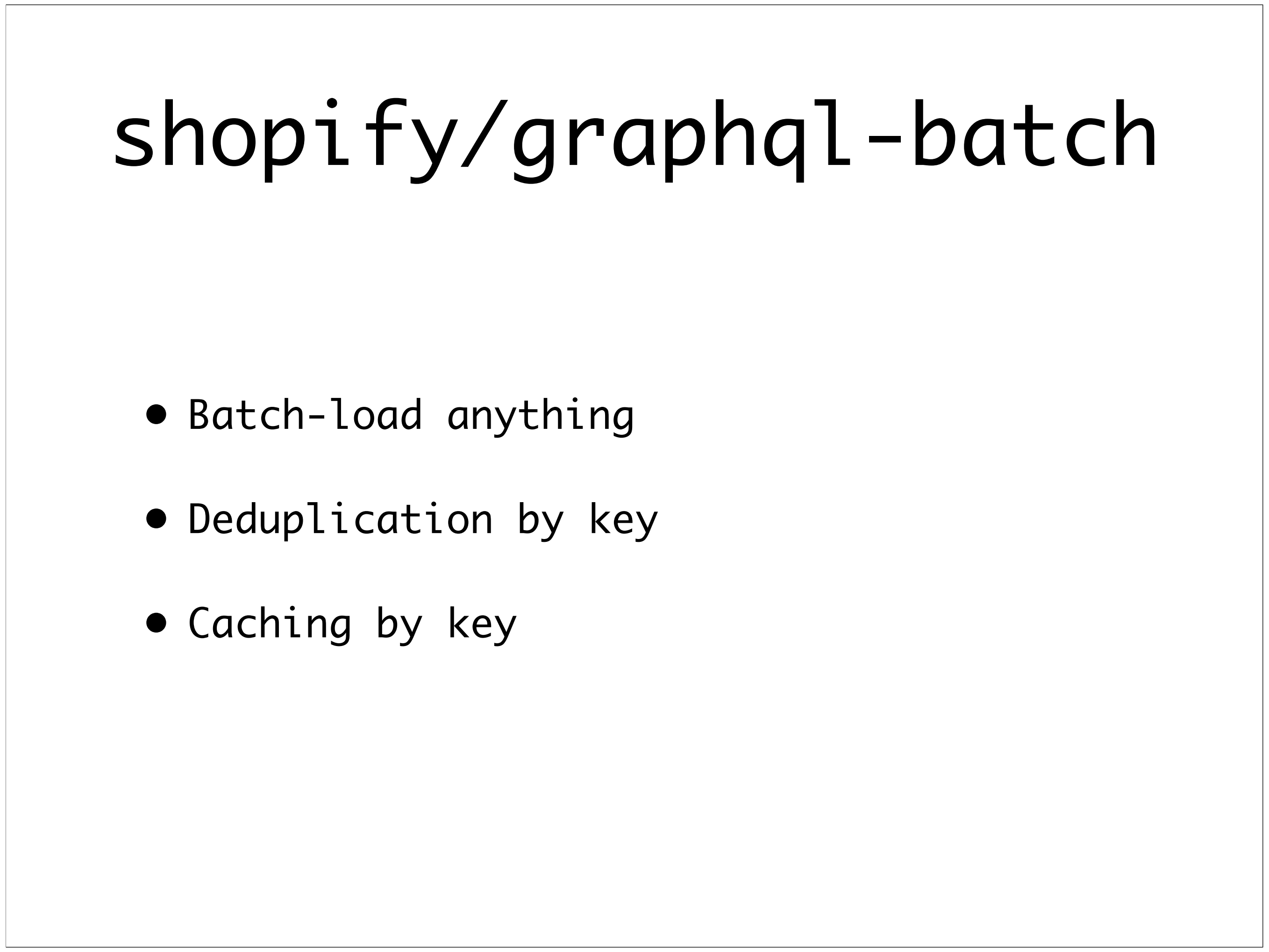
```
[eval GraphQL]
              [load data]
                         [eval...]
                                  [load...]
                                           [eval]
```

The resulting runtime flow is sequential waves of evaluation: GraphQL runs for a while, registering these requests for data, then when it can't evaluate anything else, it dispatches those registered requests.



```ruby
class Loaders::ActiveRecord < Loaders::Base
  def initialize(model)
    @model = model
  end

  def perform(ids)
    @model.where(id: ids)
  end
end
```

Under the hood, what does that look like?

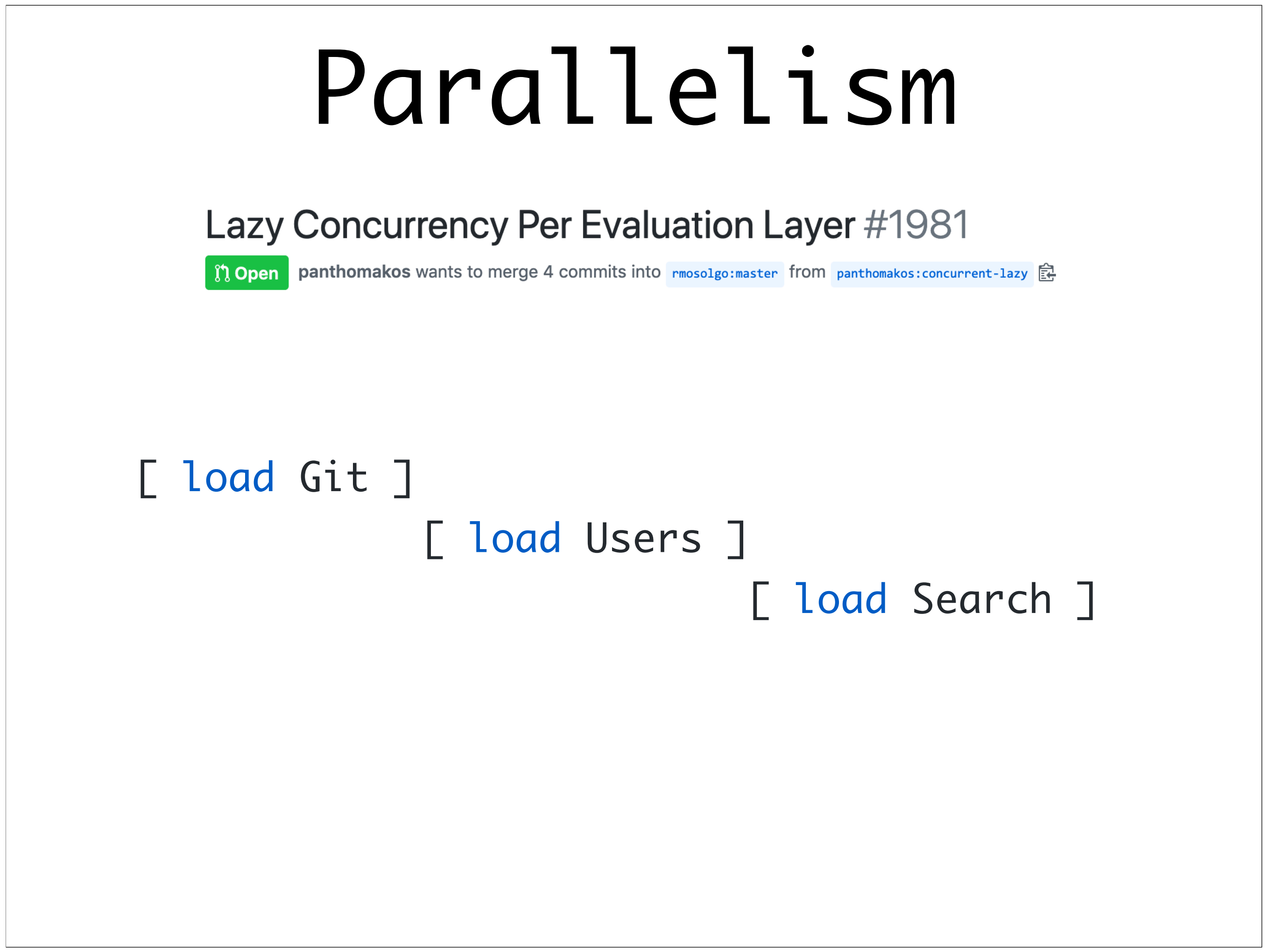


shopify/graphql-batch

- Batch-load anything
- Deduplication by key
- Caching by key

We use this extensively at GitHub: besides ActiveRecord, our GitRPC calls are batched, and complicated SQL lookups like permission checks, and external service calls. We use a library from Shopify called GraphQL-Batch, at besides batching, it has some awesome features:

- Deduplication (don't load the same object twice, even if it's requested twice)
- Caching: already-loaded objects are returned right away



Parallelism

Lazy Concurrency Per Evaluation Layer #1981

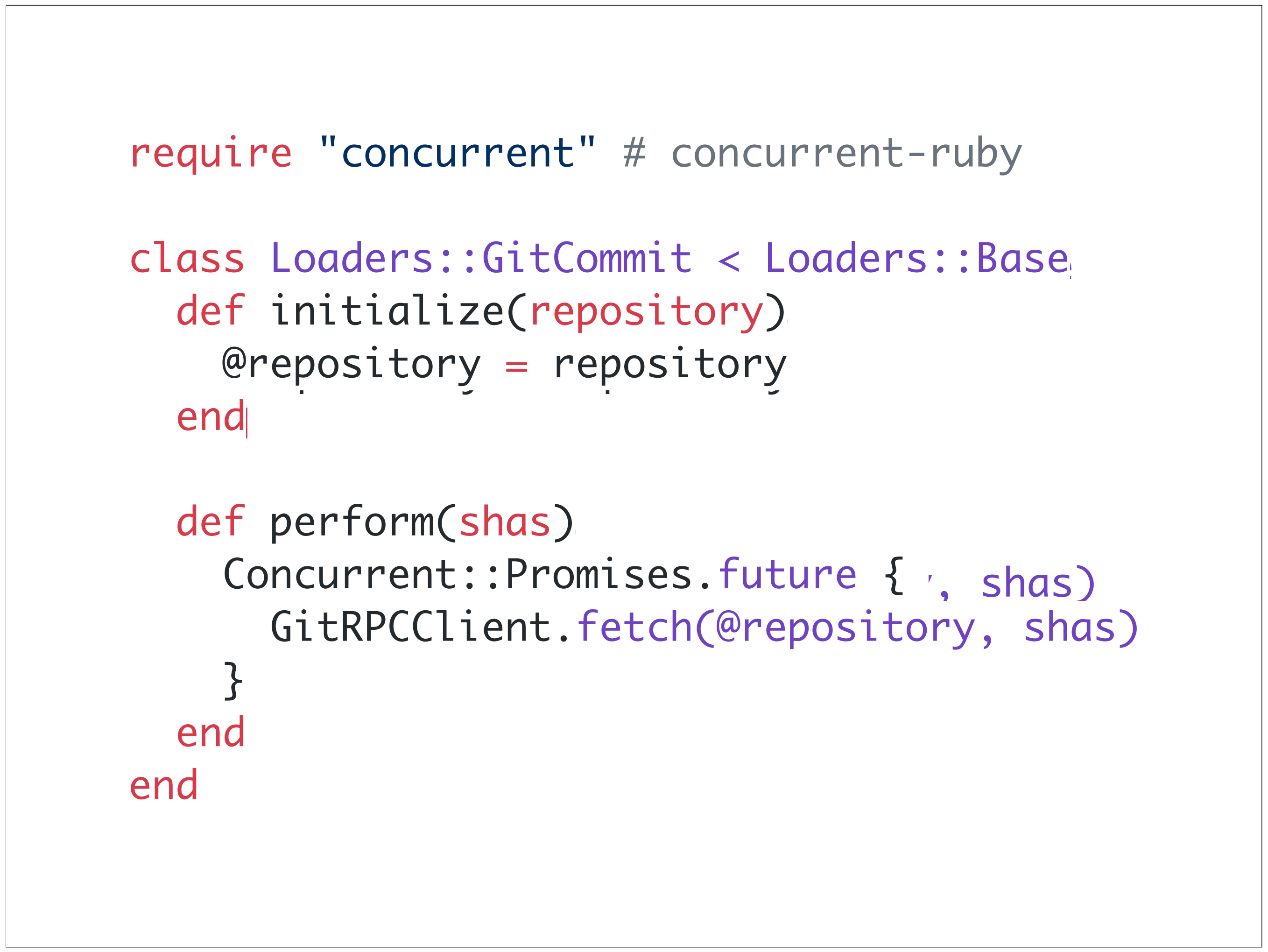
Open panthomakos wants to merge 4 commits into `rmosolgo:master` from `panthomakos:concurrent-lazy`

```
[ load Git ]
             [ load Users ]
                            [ load Search ]
```

There's another really awesome possibility here that I'm excited to integrate soon. There's a POC by a guy who works at Strava and the idea is, when you have several batched loads to evaluate, the current flow is to run them in sequence. But, assuming that these loads are IO-heavy, we'd benefit by running them in parallel. As a reminder, Ruby's interpreter lock means you can only run Ruby code one-thread-at -a-time, *but* when Ruby is waiting for external services (like MySQL or ElasticSearch, or a rat's nest of microservice calls), it can have several threads wait at once.

The goal there is that GraphQL spends less time waiting for external service calls, which means API clients spend less time waiting for responses. Which means I spend less time responding to unhappy clients.

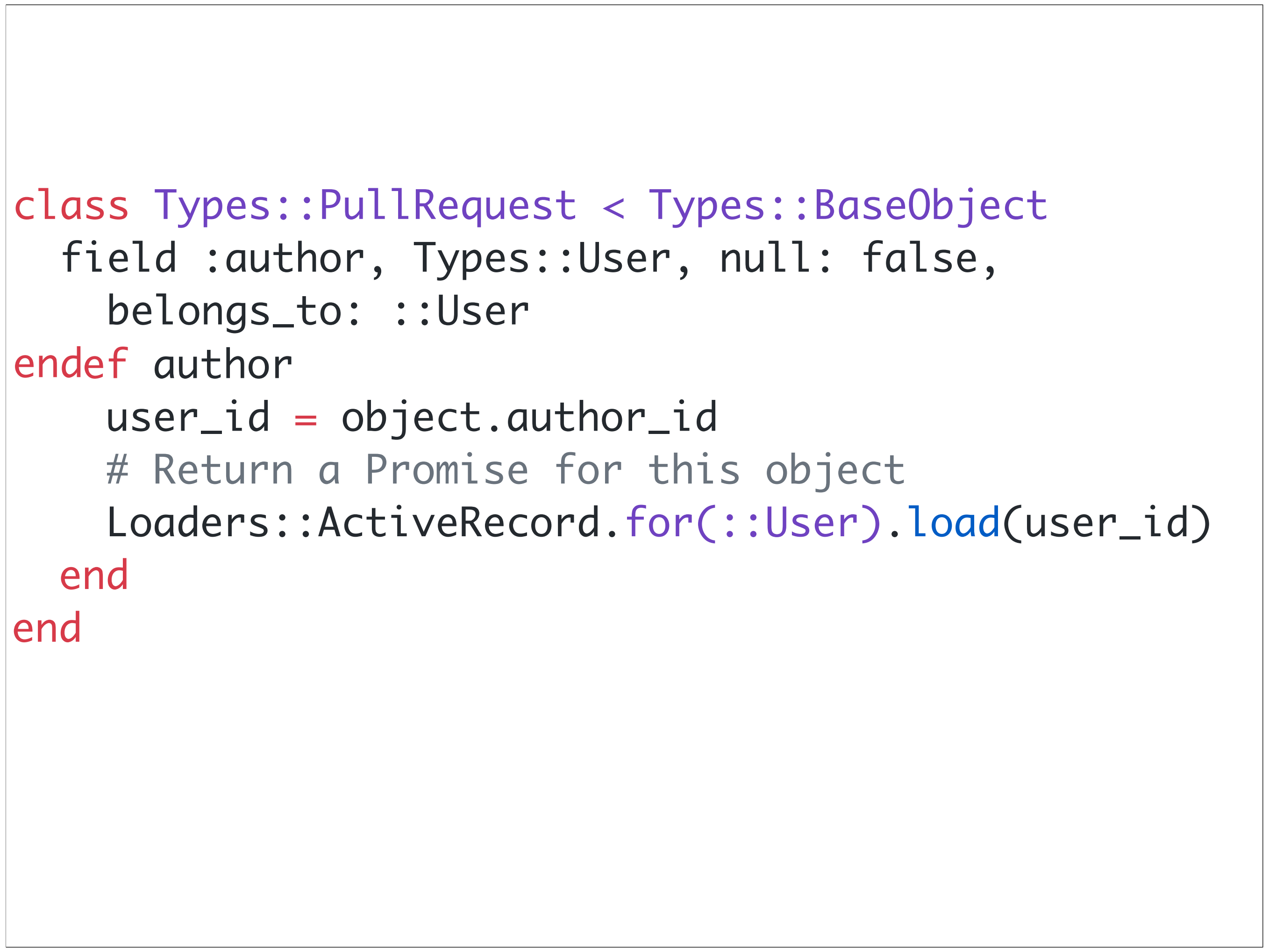
There's a catch though: it requires that your application is *thread-safe*, because different calls will be happening on different threads.



```
require "concurrent" # concurrent-ruby

class Loaders::GitCommit < Loaders::Base
  def initialize(repository)
    @repository = repository
  end

  def perform(shas)
    Concurrent::Promises.future { ', shas)
      GitRPCClient.fetch(@repository, shas)
    }
  end
end
```

That sounds great but, how would it *work*? Well, a normal loader looks *something* like this. It has a scope for batching — in this case, a repository. And then it collects keys and finally dispatches the loader with the keys it has collected — in this case, references to commits.

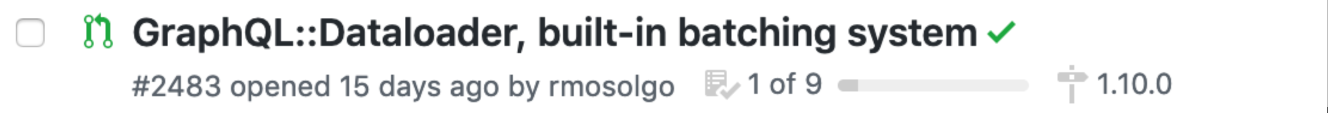
What I'd like to do is leverage concurrent-ruby, which is rock solid, and I think it ships with Rails now. It has a feature called "futures" where it will take a block, find a thread from its own thread pool, and start running the block on that other thread. What it returns is an object that keeps tabs on the task running on another thread — so you can check its status, and eventually get its return value or the error it encountered.



```
class Types::PullRequest < Types::BaseObject
  field :author, Types::User, null: false,
    belongs_to: ::User
endef author
    user_id = object.author_id
    # Return a Promise for this object
    Loaders::ActiveRecord.for(::User).load(user_id)
  end
end
```

Another area we could really improve is by building in some good defaults. Take ActiveRecord for example. We could reduce a lot of boilerplate by including some better built-in support. In this case, loading a belongs-to relation is a solved problem, but if you download graphql-ruby today, you're gonna have to solve it again, sorry. Try again in a couple of months! I think we can include some better defaults.

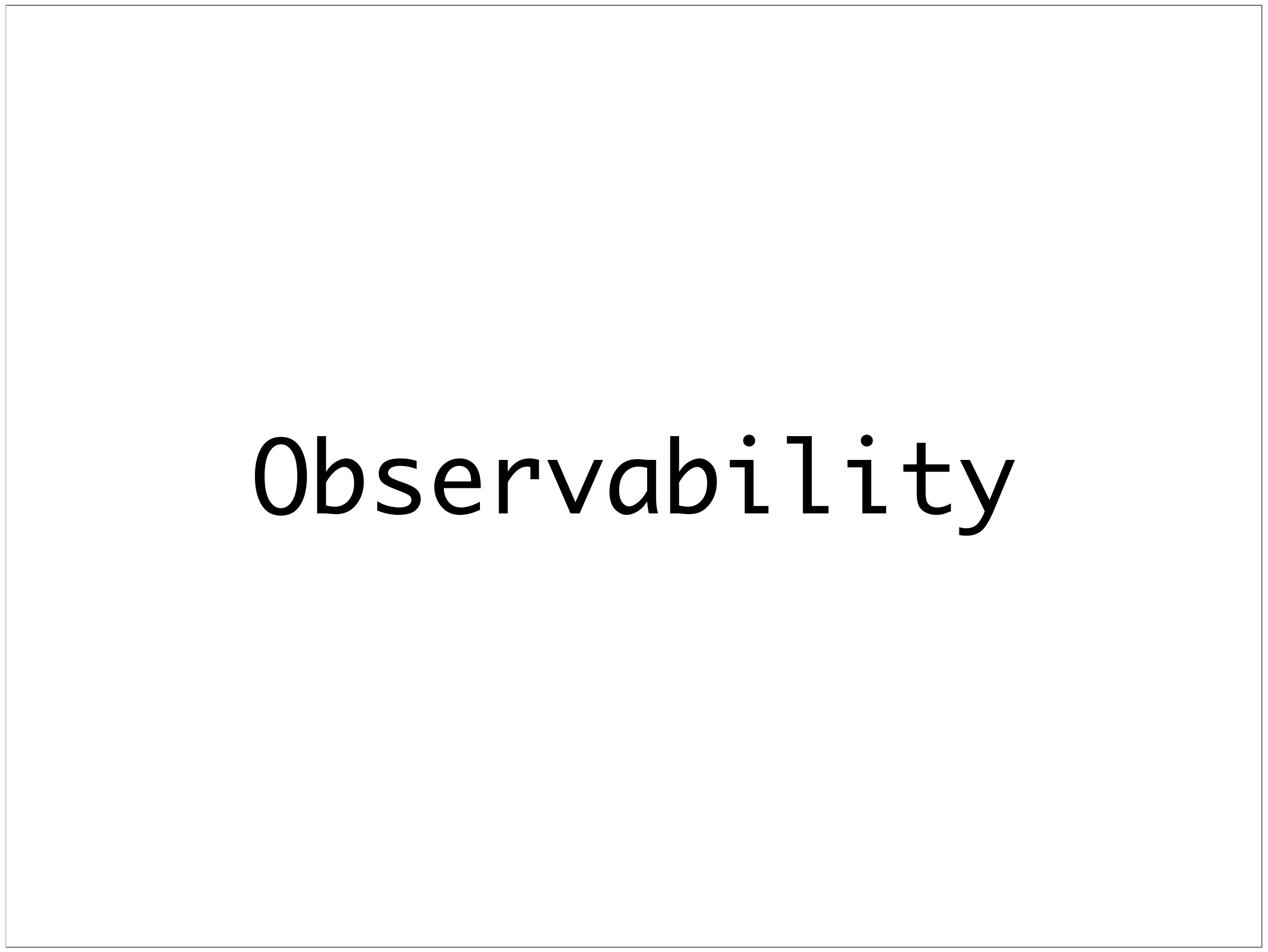
GraphQL::Dataloader



GraphQL::Dataloader, built-in batching system ✓

#2483 opened 15 days ago by rmosolgo 1 of 9 1.10.0

So, I'm trying to be inspired by Rails's approach, that if someone building a system will need it, then include it in the framework. You can keep your eyes on that issue for progress! Or if the design I discussed sounds like a really bad idea, that's a great place to let me know.

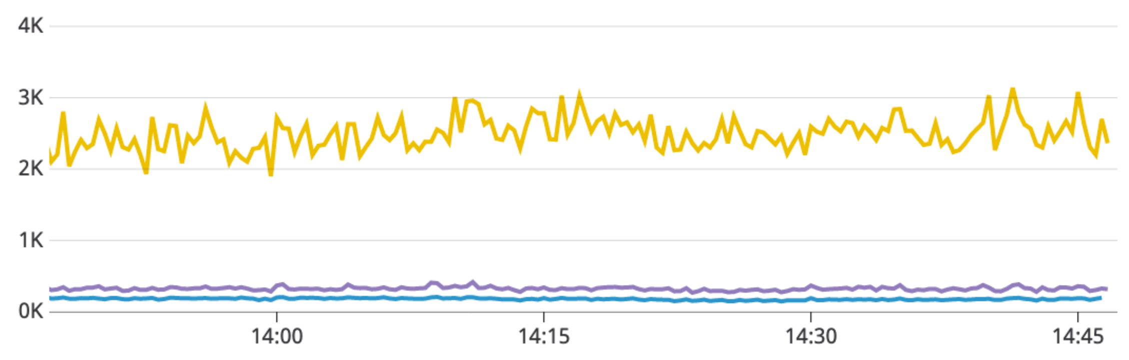


Observability

The last thing I wanted to share is not really part of the gem, but I want to share some work we've been doing, in the hope that it will get your gears turning and we can work together on solutions.

There are really two parts to this: who's using what, and does it seem like we're doing a good job with it?

POST /graphql



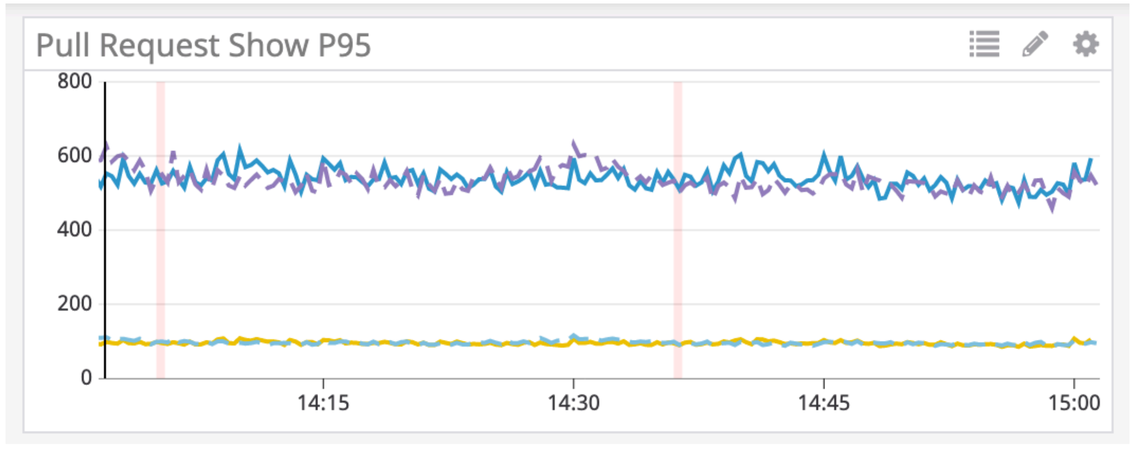
The fact is, with a REST API, this is really easy: you track who hits what endpoints, and look at response time for those endpoints. We think that GraphQL can give a better experience to our integrators, but it means we've had to rethink solutions here.

So, here's the response time breakdown for POST /graphql. There's p50 at the bottom, then p75, then ten times higher is p99. If you saw this for a normal endpoint, you'd be worried — but for GraphQL, this is normal. Why?

It's because the endpoint deals with a wide variety of requests. Some are simple requests for data from MySQL. We can return that all day long. Others are really complex searches with additional git diffs, and code highlighting, etc — so it makes *sense* that they would take a long time.

The problem is, if I ship code, how do I know if I *broke* something?

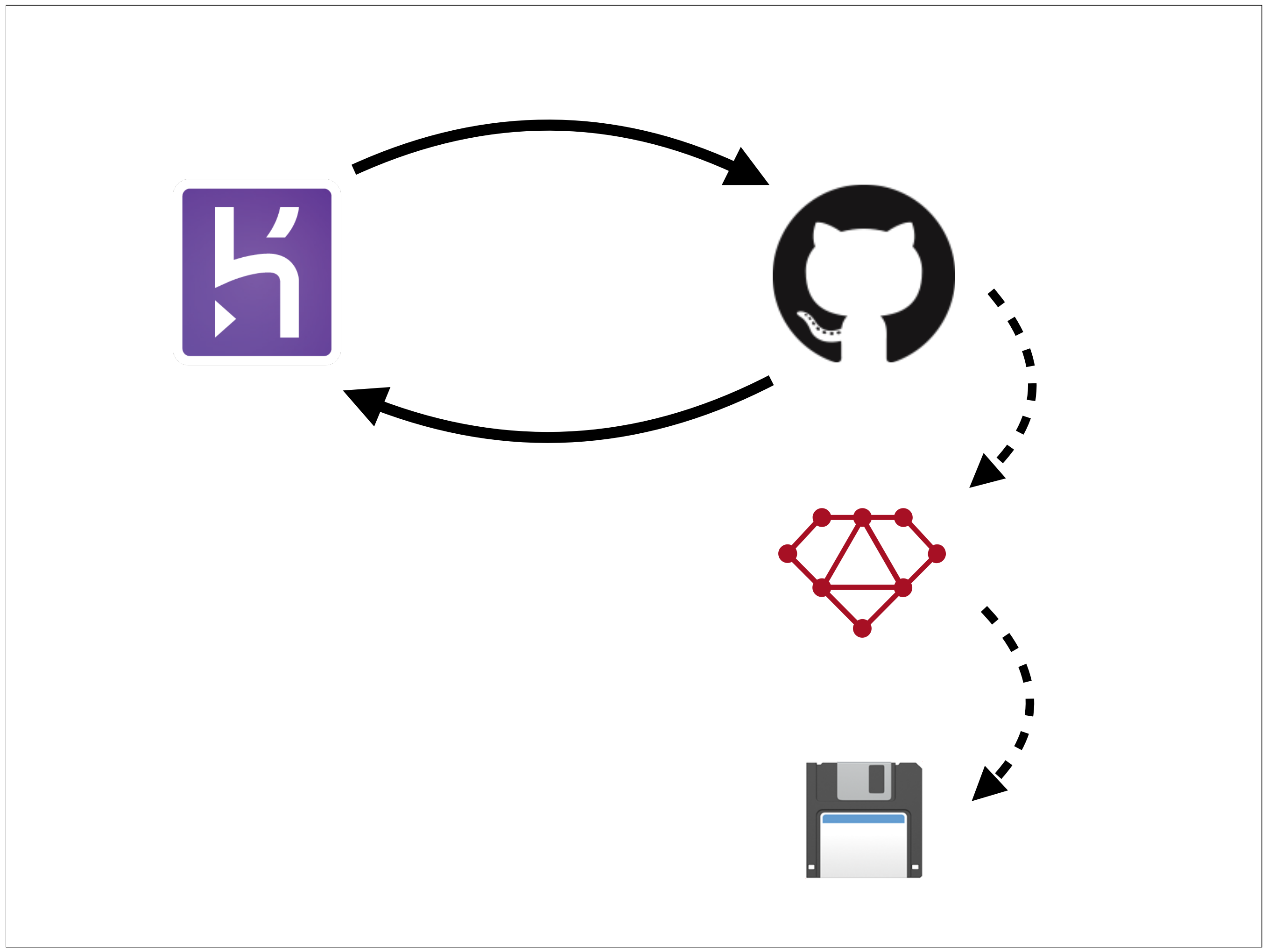
```
GET /:owner/:repo/pull/:number
```



So, here's how I used to do it, and this became a clue for me later on. This is the response time for viewing a pull request on GitHub.com. Interestingly, we actually run a GraphQL query to load that page (as well as many others), so when I want to make sure I haven't suddenly made things slower for everyone, I use that page as a canary in the coal mine. (We do have canary deploys too … but that's different!)

But how could we do this for the API? We don't know what pages our clients are loading with our data!

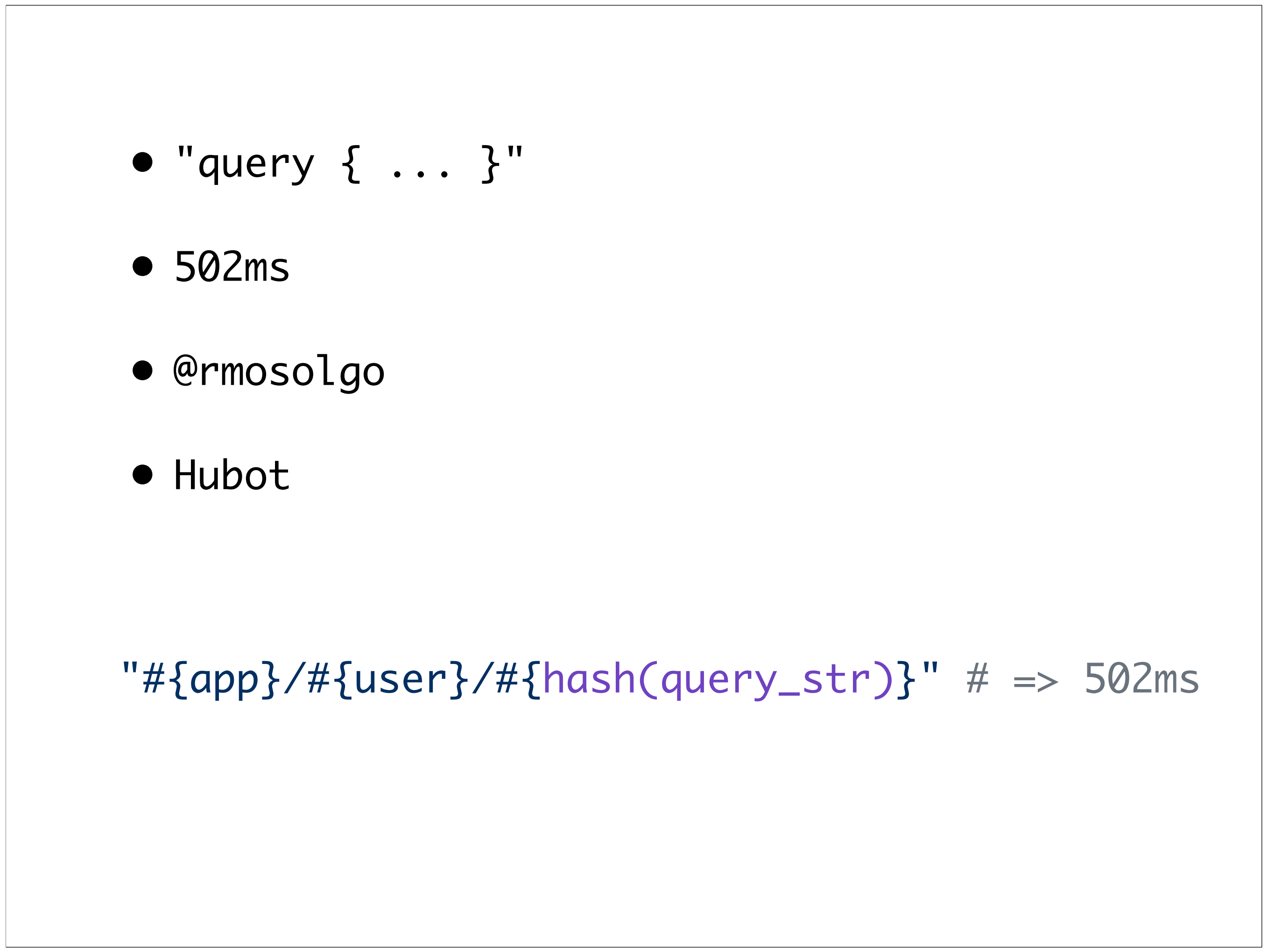
This gave me an idea



That's when it hit me. We actually *do* track incoming GraphQL queries, basically so that when we make changes to the API, we can assess current usage and evaluate the expected impact.

Basically, when the Ruby app gets a GraphQL request, it serves a response, but it also enqueues a job to *process* that request, parsing the query and storing some granular usage info in the data warehouse.

This is where we can do something interesting.



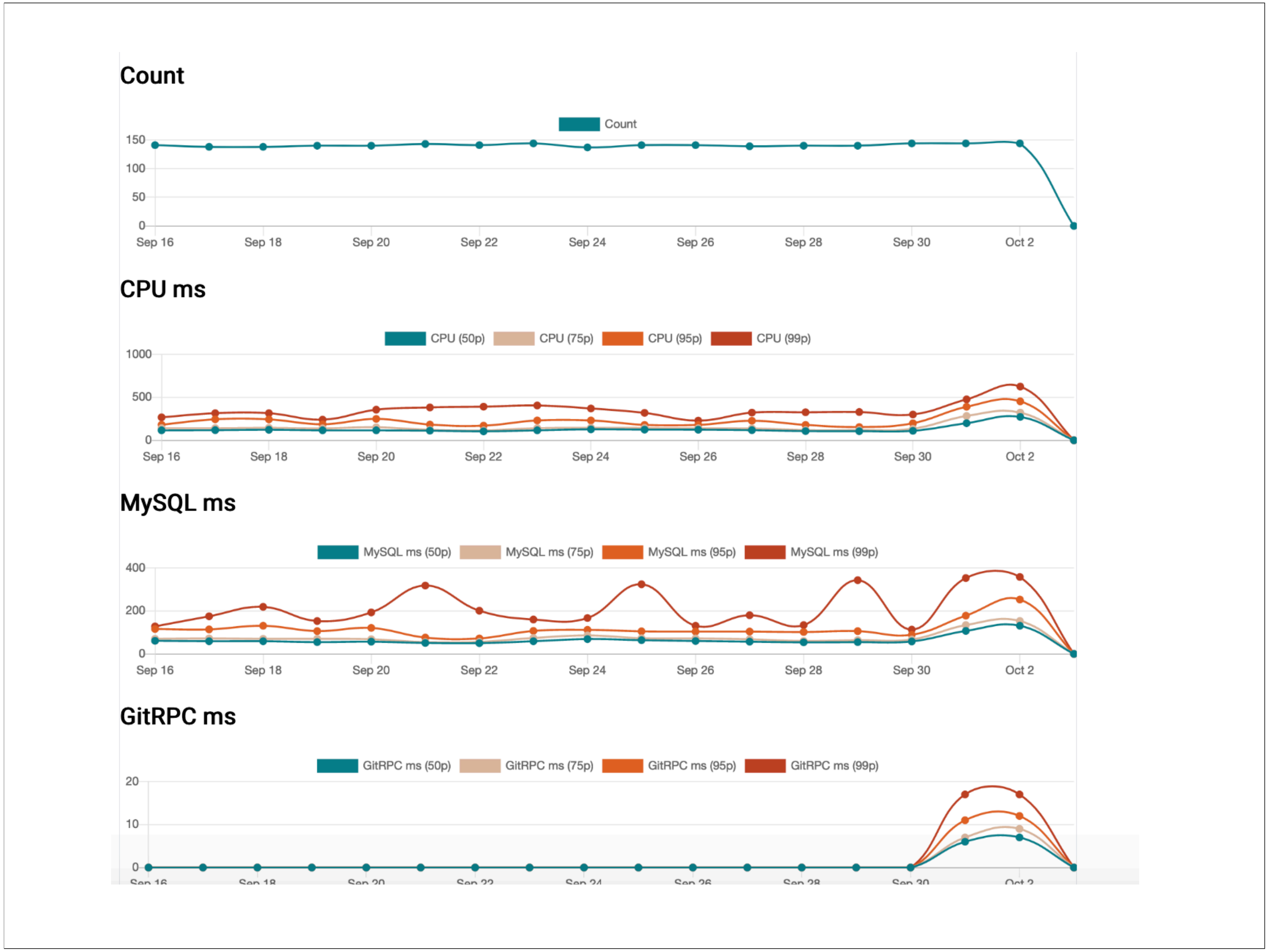
- `"query { ... }"`
- `502ms`
- `@rmosolgo`
- `Hubot`

```
"#{app}/#{user}/#{hash(query_str)}" # => 502ms
```

We have the query (values scrubbed), and we have the runtime of that query, as well as the user who ran it and the app that may have run it on that person's behalf.

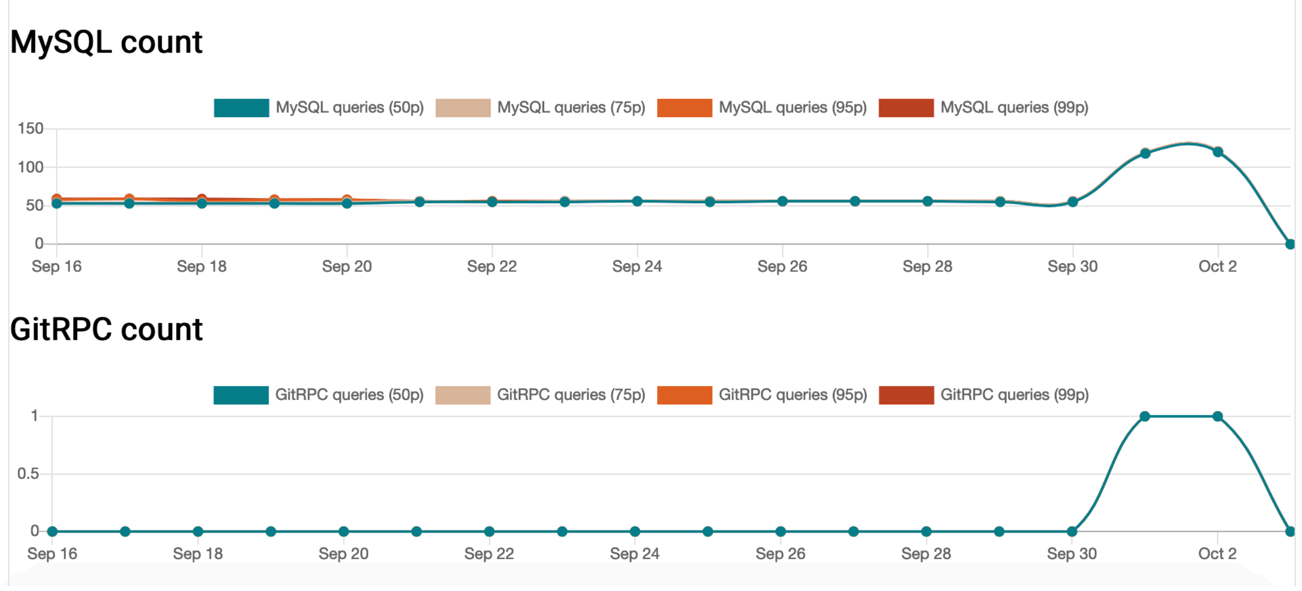
This is interestingly *like* an endpoint because it represents a kind of transaction. Especially in the case of apps that integrate with GitHub, it's a transaction that happens over and over, interacting with different resources each time.

So, we can build a hash of this information to use like a signature. And use the signature to track performance over time.

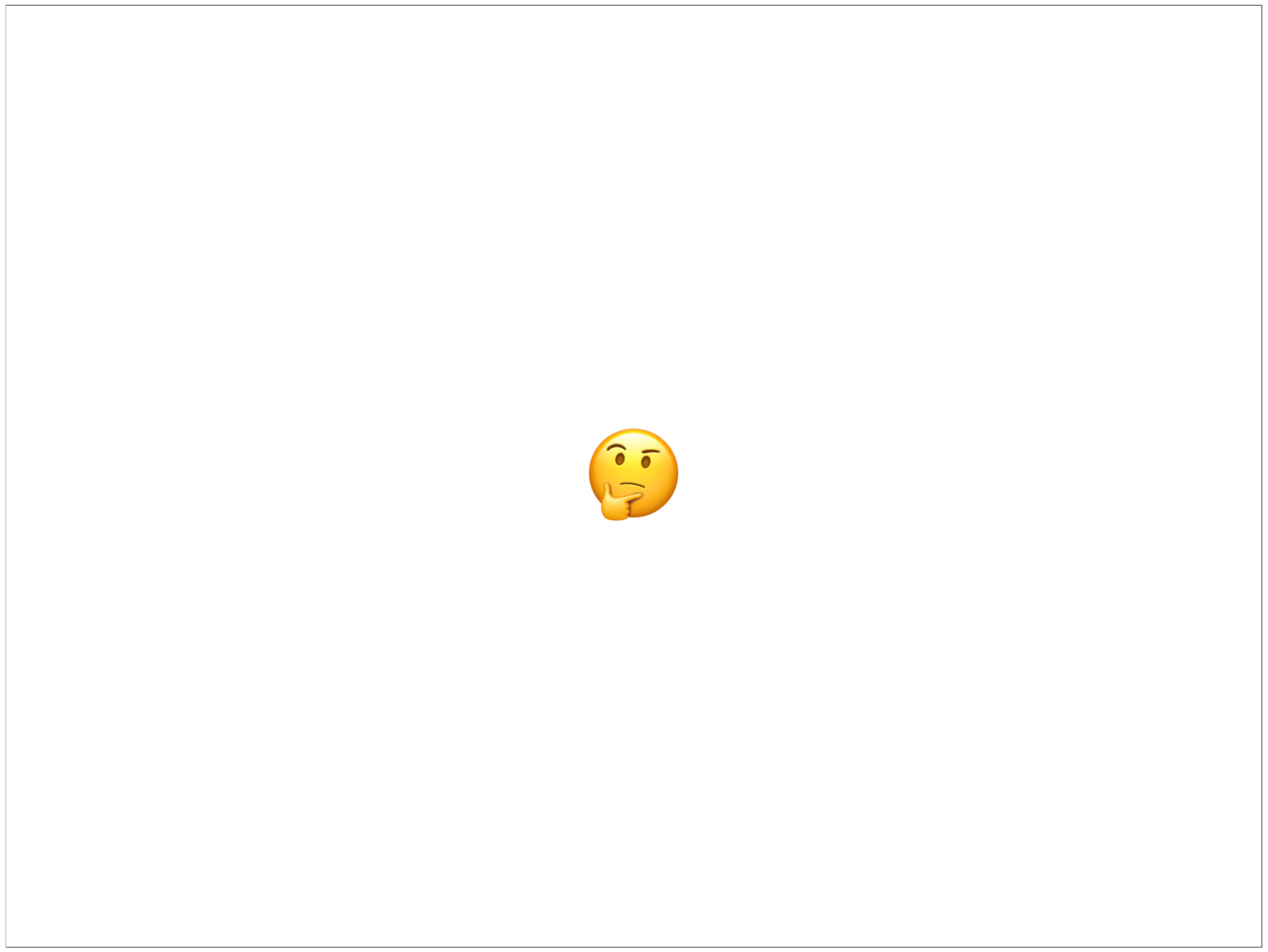


What you end up with is something like this: here's the last 2 weeks of data for a certain query, and it looks like GitRPC and MySQL time just started going up. (The CPU time did too — but it's probably a result of getting results back from a database.)

But there are still some questions here: did the time spent in MySQL grow because the underlying data changed? Something that I'm missing in this analysis is the size of the GraphQL *response* — that might tell us that it's taking longer, but not to worry, because we're also *returning more data*.

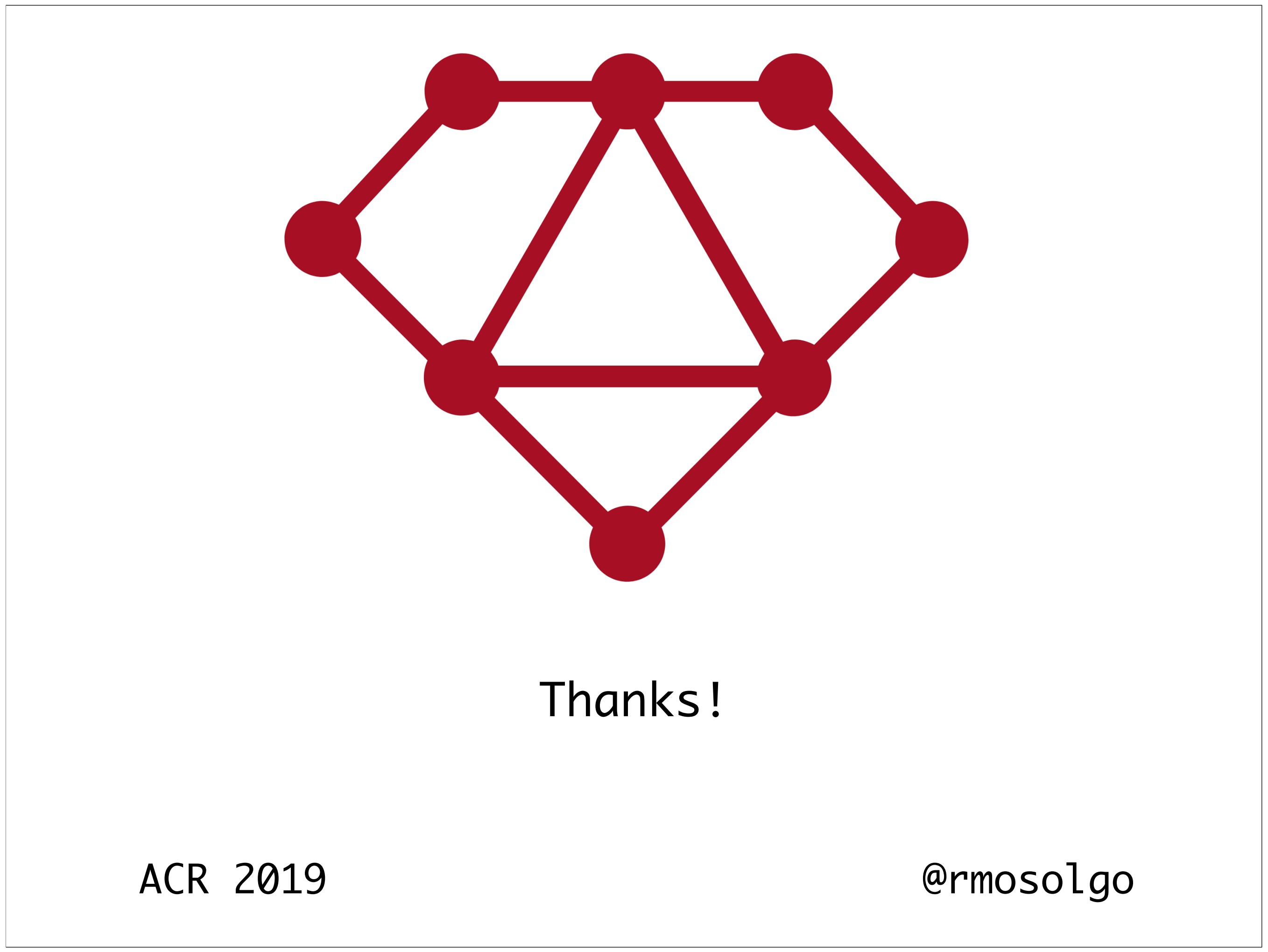


Here's another angle on it. We also track the number of external service calls needed to fill a request. If that goes up, but the response size stays the same (or doesn't get bigger), it's an indication that we might have decreased the efficiency of our implementation.



🤔

So, there's still a bit of puzzling going on for me about how to screen out the signal vs. noise. If you have a suggestion, or maybe something obvious that I forgot, please do say hi after the talk!



Thanks!

ACR 2019

@rmosolgo

That's all I have, thanks so much for your time!