An Empirical Study of GraphQL Schemas

Erik Wittern[1], Alan Cha[1], James C. Davis[2],
Guillaume Baudart[1], and Louis Mandel[1]

[1] IBM Research, USA
{witternj,lmandel}@us.ibm.com, {alan.cha1,guillaume.baudart}@ibm.com
[2] Virginia Tech, USA davisjam@vt.edu

Abstract. GraphQL is a query language for APIs and a runtime to execute queries. Using GraphQL queries, clients define precisely what data they wish to retrieve or mutate on a server, leading to fewer round trips and reduced response sizes. Although interest in GraphQL is on the rise, with increasing adoption at major organizations, little is known about what GraphQL interfaces look like in practice. This lack of knowledge makes it hard for providers to understand what practices promote idiomatic, easy-to-use APIs, and what pitfalls to avoid.
To address this gap, we study the design of GraphQL interfaces in practice by analyzing their schemas – the descriptions of their exposed data types and the possible operations on the underlying data. We base our study on two novel corpuses of GraphQL schemas, one of 16 commercial GraphQL schemas and the other of 8,399 GraphQL schemas mined from GitHub projects. We make available to other researchers those schemas mined from GitHub whose licenses permit redistribution. We also make available the scripts to mine the whole corpus. Using the two corpuses, we characterize the size of schemas and their use of GraphQL features and assess the use of both prescribed and organic naming conventions. We also report that a majority of APIs are susceptible to denial of service through complex queries, posing real security risks previously discussed only in theory. We also assess ways in which GraphQL APIs attempt to address these concerns.

Keywords: GraphQL · Web APIs · Practices.

1 Introduction

GraphQL is a *query language* for web APIs, and a corresponding *runtime* for executing queries. To offer a GraphQL API, providers define a *schema* containing the available data *types*, their *relations*, and the possible *operations* on that data. Clients send *queries* that precisely define the data they wish to retrieve or mutate. The server implementing the GraphQL API executes the query, and returns exactly the requested data. Figure 1 shows, on the left, an example query for GitHub's GraphQL API [12]. It aims to retrieve the `description` of the `graphql-js` repository owned by `graphql`. The query, in case that this owner is an `Organization`, further requests the `totalCount` of all members of that organization,

and the names of the first two of them. The right hand side of Figure 1 shows the response produced by GitHub's GraphQL API after executing that query,[3] which contains exactly the requested data.

```
query {
  repository(name: "graphql-js", owner: "graphql") {
    description
    owner {
      ... on Organization {
        membersWithRole(first: 2) {
          totalCount
          nodes {
            name
          }
} } } } }
```

```
{ "data": {
  "repository": {
    "description": "A reference imple...",
    "owner": {
      "membersWithRole": {
        "totalCount": 4,
        "nodes": [
          {"name": "Member 1"},
          {"name": "Member 2"}
        ]
} } } } }
```

Fig. 1. Example of a GraphQL query (left), and corresponding JSON response (right).

GraphQL is seeing adoption at major organizations thanks in part to its advantages for performance and usability. In some use-cases, allowing users to precisely state data requirements using GraphQL queries can lead to fewer request-response roundtrips and smaller response sizes as compared to other API paradigms, e.g., REST-like APIs [16]. GraphQL prescribes a statically *typed* interface, which drives developer tooling like GraphiQL, an online IDE helping developers explore schemas and write and validate queries [2], or type-based data mocking for testing services [3]. Major organizations have begun to embrace it, including GitHub [12], Yelp [10], The New York Times [11], or Shopify [13].

As any new technology is deployed, users begin to follow useful patterns and identify best practices and anti-patterns. Our aim is to shed light on emerging GraphQL uses and practices, in the spirit of similar studies for REST(-like) APIs [18,22,23]. By studying technological practices in the GraphQL context, we benefit the entire GraphQL community: Our study will help GraphQL providers build idiomatic, easy-to-use GraphQL APIs, and avoid pitfalls others have experienced before. Our findings also inform tool developers about the practices that are more (and less) important to support. Obviously GraphQL consumers will benefit from the resulting well-designed GraphQL APIs and effective tool support. And finally, our contributions may influence the evolution of GraphQL itself, as we highlight challenges that the specification may eventually address.

Specifically, the contributions of this work are:

– We present two novel GraphQL schema corpuses, derived respectively from commercial GraphQL deployments and open-source projects (§3). We make parts of the open-source corpus – as permitted by schema licenses – publicly available for other researchers [30], and also share the scripts to reproduce the whole open-source corpus [29].

[3] We anonymized the returned names.

– We analyze our corpuses for common schema characteristics, naming conventions, and worst-case response sizes, and describe practices that address large responses (§4).

In brief, we find that: (1) There are significant differences between commercial and open-source schemas; (2) Schemas commonly follow naming conventions, both documented and not; (3) A majority of schemas have large worst-case response sizes, which schema developers and endpoint providers should consider; and (4) Mechanisms to avoid these large response sizes are applied inconsistently.

2 Background

As sketched above, a schema describes the types of data offered by a GraphQL API, the relations between those types, and possible operations on them. In this section, we outline selected concepts related to GraphQL schemas. GraphQL providers can define schemas either programmatically using libraries like graphql-js [6], or they can define them declaratively using the *Schema Definition Language* (SDL). Figure 2 shows an example schema defined in the SDL.

```
schema {
  query: Query
  mutation: Mutation
}
type Mutation {
  createOffice(input: OfficeInput!): Office
}
type Query {
  company(id: ID!): Company
}
type Company {
  id: ID!
  name: String
  address: String
  age: Int @deprecated(reason: "No longer relevant.")
  offices(limit: Int!, after: ID): OfficeConnection
}
```

```
type OfficeConnection {
  totalCount: Int
  nodes: [Office]
  edges: [OfficeEdge]
}
type OfficeEdge {
  node: Office
  cursor: ID
}
type Office {
  id: ID!
  name: String
}
input OfficeInput {
  name: String!
}
```

Fig. 2. Example of a GraphQL schema in the Schema Definition Language (SDL).

The `schema` defines *fields* `query` and `mutation`, one of which forms the entry for any valid query. Every GraphQL schema must contain a `Query` operation type, which in this case is the `Query` object type. According to this schema, queries can retrieve a `company` field that returns a `Company` identified by an `id` *argument* of type `ID` (the character "`!`" indicates that the argument is required). The returned `Company` again allows queries to retrieve its `id`, `name`, `address`, `age`, and/or `offices`. The latter requires the user to `limit` the number of offices returned. Offices, implementing the *connections pattern* for pagination [7], are related to a company via an `OfficeConnection`, that contains information about the `totalCount` of offices of that company, and grants access to them directly via the `nodes` field or indirectly via the `edges` field. Querying for an `OfficeEdge` allows users to obtain

a `cursor` that they can use (in subsequent queries) to precisely slice which `offices` to retrieve from a `Company` via the `after` argument.

```
query { company(id: "n3...") { offices(limit: 10, after: "mY...") { edges: {
     cursor
     node { name }
} } } }
```

The schema further allows to mutate data via the `createOffice` field. The data about the office to create is defined in a dedicated input object type called `OfficeInput` and passed as an argument. A corresponding query may look like:

```
mutation { createOffice(input: { name: "A new office" }) {
   id
} }
```

In GraphQL, basic types like `String`, `Int`, or `Boolean` are called *scalars*, sets of predefined strings are called *enums*, and complex types that contain fields are called *object types* (e.g., `Company` in Figure 2). GraphQL further allows developers to define *interfaces* that can be implemented by object types or extended by other interfaces, and *unions* which state that data can be of one of multiple object types. For example, in line 5 of Figure 1, `... on Organization` is a *type condition* that queries fields on the interface `RepositoryOwner` returned by field `owner` only if the owner happens to be an `Organization`. Beyond queries that retrieve data, GraphQL schemas may also define a root `Mutation` operation type, whose fields define possible mutations, e.g., to create, edit, or delete data. Input for mutations is defined using arguments, which are (lists of) scalars, enums, or *input* object types. Finally, GraphQL schemas may contain *directives* that define metadata or behavioral changes associated with field, type, argument or even the whole schema definitions. For example, in Figure 2 the field `age` in type `Company` is marked with a directive as deprecated. This information could, for example, be displayed by documentation tooling. Tools or clients can send an *introspection* query to retrieve the latest schema from a GraphQL API.

After defining their GraphQL schema, to offer a GraphQL API a provider must build a mapping between the data types defined in the schema and their representation in the back-end storage system(s). The provider does this by implementing a *resolver function* for each field defined in the schema, which can retrieve or mutate the corresponding data. Resolver functions can, for example, interact with databases, other APIs, or dynamically compute a result — GraphQL is agnostic to their implementation. To execute a query, a GraphQL runtime validates it against the schema, and then in sequence calls all resolver functions required to fulfill the query.

Although a schema definition does not tell us everything about a GraphQL API (e.g., how its resolver functions are implemented), GraphQL schemas can still tell us about GraphQL practices. For example, from a GraphQL schema we can learn the characteristics of the corresponding GraphQL API, the nature of possible queries to its API, and the conventions followed in designing it. Schema definitions thus comprise useful research artifacts. In the next section we discuss the schema definitions we sampled to understand these and other topics.

3 Data: Two Novel GraphQL Schema Corpuses

We created two corpuses of GraphQL schemas: one from introspecting publicly accessible commercial GraphQL APIs (§3.1), and the other from mining GitHub for GraphQL schema definitions (§3.2). Figure 3 illustrates the GraphQL schema populations we sampled to create these corpuses.

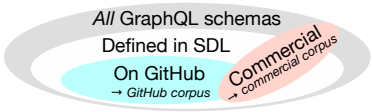

Fig. 3. Schema corpuses used in this work. A subset of all GraphQL schemas is defined using the SDL rather than programmatically. We mine the subset hosted on GitHub. Schemas in our commercial corpus can be defined either way, and may be hosted (privately) on GitHub.

In both corpuses, we included only schemas that are parsable (e.g., written in valid SDL syntax) and complete (e.g., contains a *query operation* and definitions of all referenced types). We checked these constraints using the *parsing* and *validation* capabilities offered by the graphql-js reference implementation [6], thus ensuring that schemas can be processed and analyzed without risking runtime errors. We make available the schemas in the open-source corpus – considering the constraints for redistributing them defined in their licenses [30]. We also make available the scripts to collect the whole open-source corpus [29]. These scripts contain the schema reconstruction logic described in §3.2.

3.1 Commercial Corpus (Schemas Deployed in Practice)

Our commercial corpus (16 schemas) represents GraphQL schemas written and maintained by professional software developers for business-critical purposes. This corpus allows us to reason about GraphQL practices in industry.

To identify commercial GraphQL APIs, we started with the community-maintained list provided by *APIs.guru* [1].[4] We manually assessed the documentation for all 33 of the "Official APIs" listed on May 1^{st} 2019 to remove demo interfaces and non-commercial APIs. We then used introspection to collect these commercial GraphQL schemas. After discarding invalid schemas (validity is defined in §3.2), we obtained our final corpus of 16 valid, unique GraphQL schemas maintained by commercial websites. The corpus includes, among others, schemas of prominent GraphQL APIs like GitHub, Shopify, Yelp, and BrainTree.

[4] We submitted a pull request adding several public GraphQL APIs that were missing from the *APIs.guru* list, but that we found using web searches. The *APIs.guru* maintainers accepted the pull request and we included those schemas in this analysis.

3.2 Open-Source Corpus (Schemas in GitHub Projects)

Our open-source corpus (8,399 schemas) provides another perspective on GraphQL practices, depicting (ongoing) development efforts and privately-deployed APIs. For this corpus, we aimed to collect schema definitions written in the SDL (cf. §2) and stored in a GitHub project. Figure 4 summarizes the stages of our data-collection methodology.

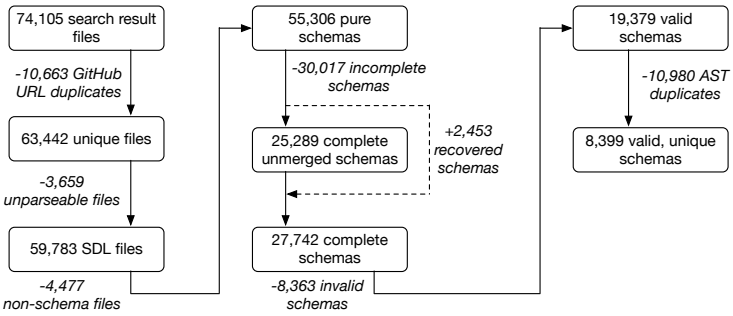

Fig. 4. Filters to construct the open-source schema corpus.

We used GitHub's code search API to obtain **search result files** that likely contain schemas on May 21st 2019 using this query:

```
type extension:graphql extension:gql
  size:<min>..<max> fork:false
```

The pieces of this query have the following meaning. The *search term* `type` is used in any non-trivial schema in the GraphQL SDL. The *extensions* `.graphql` *and* `.gql` are common file suffixes for the GraphQL SDL. The *file sizes* `<min>` *and* `<max>` partitioned the search space by file size in order to work around GitHub's limit on code search query results. We omitted project forks to avoid duplicates.

We removed duplicates by URL to obtain **unique files**. We filtered unparsable files per the graphql-js [6] reference implementation to obtain **SDL files**. The GraphQL SDL can describe not only schemas, but also executables like queries, mutations, or subscriptions (e.g., the query in Figure 1) or a mixture of both. Because we are only interested in schemas, we obtained **pure schemas** by removing any files that contain executables, a.k.a. *executable definitions* [14].

These steps left us with parsable SDL files, but not all are **complete**.[5] We observed that some schemas contain reference errors, e.g., because they are divided across multiple files for encapsulation. Supposing that a repository's complete schema(s) can be produced through some combination of its GraphQL files, we used heuristics to try to reconstruct these *partitioned schemas*, thus

[5] A *complete* schema (1) contains a *query operation* (a *SchemaDefinition* node [15] or a *Query* object type [8]), and (2) defines all referenced types and directives.

adding **recovered schemas** back to our data. For every schema that contains a *query operation* but also reference errors, we searched for the missing definitions in the repository's other GraphQL files. When we found a missing type in another file, we appended that file's contents to the current schema.[6] We repeated this process until we obtained either a complete schema or an unresolvable reference. Of the 30,017 **incomplete schemas**, there are 5,603 that contain an *query operation*, meaning they can form the basis of a merged schema, and from these schemas, we were able to recover 2,453 schemas (43.8% success rate). This success rate suggests that *distributing GraphQL schema definitions across multiple files is a relatively common practice.*

We obtained **valid schemas** by removing ones that could not be validated by the graphql-js reference implementation, and finally **valid, unique schemas** by removing duplicates by testing for abstract syntax tree (AST) equivalence. We discarded about half of the remaining schemas during deduplication (Figure 4). Inspection suggests many of these schemas were duplicated from examples.

Our final open-source schema corpus contains 8,399 valid, unique GraphQL Schema Definition files, 1,127 of which were recovered through merging.[7] Although all of these schemas are valid, some may still be "toy" schemas. We take a systematic approach to identify and remove these in the analysis that follows.

4 Schema Analysis: Characteristics and Comparisons

In this section, we analyze our GraphQL schema corpuses. We discuss schema metrics, characterize the corpuses, and compare and contrast them. Specifically, we analyze some general characteristics (§4.1), identify naming conventions (§4.2), estimate worst-case query response sizes (§4.3), and measure the use of pagination, a common defense against pathological queries (§4.4).

Because our purpose is to understand GraphQL practices in open-source and in industry, we extracted a subset of the GitHub corpus called the "GitHub-large" (GH-large) corpus that is comparable in complexity to the commercial corpus. This distinction is useful for measurements that are dependent on the "quality" of the schema, e.g., worst-case response sizes and defenses, though for studies like trends in naming conventions we think it is appropriate to also consider the full GitHub corpus. In future analyses, other measures of quality could be considered to segment the data, for example the number of stargazers of the associated GitHub repository.

We identified the GitHub-large corpus using a simple measure of schema complexity, namely its number of distinct definitions (for types, directives, operations etc.). As shown in Figure 5, the smallest commercial schema contains 8 definitions, while half of the GitHub corpus contains 9 or less definitions. To

[6] If multiple possible definitions were found, we broke ties under the assumption that developers will use the directory hierarchy to place related files close to each other.

[7] We collected data in November 2018 using the same methodology, and found $5,345$ unique schemas, 701 of which resulted from merging. This reflects a growth of 57% in half a year.

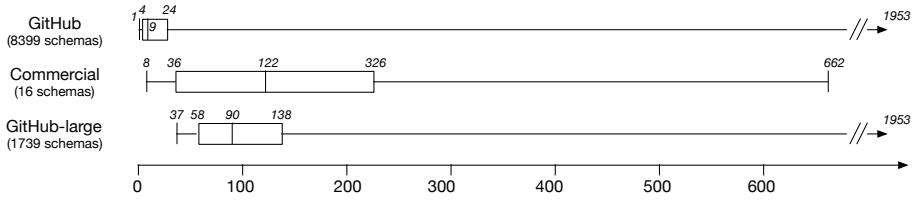

Fig. 5. Distributions of schema complexity (number of definitions) in the GitHub, commercial, and GitHub-large schema corpuses. Whiskers show min and max values and the boxes show the quartiles.

avoid a bias toward these toy schemas and to accommodate the small sample size of the commercial corpus, we conservatively define a GitHub schema as *large* if it is at least as complex as the first quartile of the commercial corpus (i.e., has more than 36 definitions). We include separate measurements on this GitHub-large corpus (1,739 schemas, 20.7% of the GitHub corpus, 10 of which were recovered through merging). The complexity distribution of the GitHub-large corpus is not perfectly aligned with the commercial corpus, but it is a better approximation than the GitHub corpus and allows more meaningful comparisons of open-source and industry GraphQL practices.

4.1 Schema Characteristics

First, we provide a reference for what GraphQL schemas look like in practice. This snapshot can inform the design of GraphQL backends (e.g., appropriately sizing caches) as well as the development of the GraphQL specification (e.g., more and less popular features). We parsed each schema using the graphql-js reference implementation [6] and analyzed the resulting AST.

Table 1. Characteristics & Features Used in Schema Corpuses.

	Commercial (16)	**GitHub** (8,399)	**GH-large** (1,739)
Median object types (OTs)	60	6	35
Median input OTs	44	6	43
Median fields in OTs	3	3	3
Median fields in Input OTs	2	3	3
Have interface types	11 (68.8%)	2,377 (28.3%)	1,395 (80.2%)
Have union types	8 (50.0%)	506 (6.0%)	330 (19.0%)
Have custom directives	2 (12.5%)	160 (1.9%)	26 (1.5%)
Subscription support	0 (0.0%)	2,096 (25.0%)	1,113 (64.0%)
Mutation support	11 (68.8%)	5,699 (67.9%)	1,672 (96.1%)

Table 1 shows clear differences among all three corpuses. Not surprisingly, commercial and GitHub-large schemas are larger, containing more object and

input object types. The sizes of individual object and input object types, however, look similar in all corpuses. In terms of feature use, commercial schemas apply interface types, union types, and custom directives most frequently, followed by GitHub-large schemas and then GitHub schemas. Conversely, GitHub-large schemas have mutation and subscription[8] support most frequently, followed by GitHub schemas and then commercial schemas.

Analyzing multiple corpuses provides a fuller picture of GraphQL practices. For example, suppose you were to propose changes to the GraphQL specification based solely on one of these corpuses, e.g. to identify little-used features as deprecation candidates. Considering only commercial schemas, subscription support appears to be unpopular (none of the commercial schemas offer subscriptions), so subscriptions might be a deprecation candidate. But the GitHub-large corpus tells a different story: subscriptions are offered in 64% of the GitHub-large schemas. Considering only the GitHub-large corpus instead, you might conclude that custom directives are a deprecation candidate (only 1.5% of GitHub-large schemas use them), even though 12.5% of the commercial corpuses use them. In both cases a single-corpus analysis is misleading, showing the value of our multi-corpus analysis.

Finding 1: Commercial and GitHub-large schemas are generally larger than GitHub schemas. Reliance on different GraphQL features (e.g., unions, custom directives, subscription, mutation) varies widely by corpus.

4.2 Naming Conventions

Naming conventions help developers understand new interfaces quickly and create interfaces that are easily understandable. In this section we explore the prescribed and organic naming conventions that GraphQL schema authors follow, e.g. common ways to name types, fields, and directives. Table 2 summarizes our findings. We focus on the proportion of schemas that follow a convention *consistently*, i.e., the schemas that use them in all possible cases.

Prescribed conventions. GraphQL experts have recommended a set of naming conventions through written guidelines [4] as well as implicitly through the example schemas in the GraphQL documentation [5]. These prescribed conventions are: (1) Fields should be named in camelCase; (2) Types should be named in PascalCase; and (3) Enums should be named in PascalCase with (4) values in ALL_CAPS.

We tested the prevalence of these conventions in real GraphQL schemas[9]. As shown in Table 2, these prescribed conventions are far from universal. The only prescribed convention that is frequently used in all three corpuses is (3) PascalCase enum names, exceeding 80% of schemas in each corpuses and over

[8] Subscriptions permit clients to register for continuous updates on data.

[9] For simplicity, we tested for camelCase and PascalCase names using only the first letter. A more sophisticated dictionary-based analysis is a possible extension.

Table 2. The proportion of schemas that consistently adhere to prescribed (upper part) and organic (lower part) naming conventions. In rows marked with a † we report percentages from the subsets of schemas that use any enums, input object types, or mutations, respectively.

	Commercial (16)	**GitHub** (8,399)	**GH-large** (1,739)
camelCase field names	12.5%	53.9%	8.2%
PascalCase type names	62.5%	91.8%	82.1%
PascalCase enum names †	81.3%	96.8%	96.4%
ALL_CAPS enum values †	56.3%	35.7%	12.1%
`Input` postfix †	23.1%	71.6%	68.2%
Mutation field names †	9.1%	49.3%	62.9%
snake_case field names	0.0%	0.5%	0.1%

95% in the GitHub and GitHub-large corpuses. In contrast, (1) camelCase field names are only common in GitHub schemas, (2) PascalCase type names are common in GitHub and GitHub-large schemas and less so in commercial schemas, and (4) ALL_CAPS enum values appear in more than half of commercial schemas, but are unusual in the GitHub and GitHub-large schemas.

Organic conventions. Next we describe "organic" conventions[10] that we observed in practice but which are not explicitly recommended in grey literature like the GraphQL specification or high-profile GraphQL tutorials.

`Input` postfix for input object types. Schemas in our corpuses commonly follow the convention of ending the names of input object types with the word `Input`. This convention is also followed in the examples in the official GraphQL documentation [5], but the general GraphQL naming convention recommendations do not remark on it [4]. In GraphQL, type names are unique, so the `Input` postfix is often used to associate object types with related input object types (e.g., the object type `User` may be related to the input object type `UserInput`).

Mutation field names. Developers commonly indicate the effect of the mutation by including it as part of the field name. These names are similar to those used in other data contexts: `update`, `delete`, `create`, `upsert`, and `add`.

snake_case field names. Of the non-camelCase field names in the GitHub corpus, 90.3% follow snake_case (determined by the presence of an underscore: "_"), covering 30.6% of all field names and used in 37.3% of all schemas in the GitHub corpus. However, barely any schema across all corpuses uses this convention throughout.

In general, the observed organic conventions are much more common in GitHub and GitHub-large schemas than in commercial schemas.

[10] These conventions are "organic" in the sense that they are emerging naturally without apparent central direction. There could, however, be some hidden form of direction, e.g. many projects influenced by the same team or corporation.

Finding 2: GraphQL experts have recommended certain naming conventions. We found that PascalCase enum names are common in all three corpuses, and PascalCase type names are common in the GitHub and GitHub-large corpuses, but other recommendations appear less consistently. In addition, we observed the relatively common practice of `input` postfix and mutation field names in the GitHub and GitHub-large corpuses. We recommend that commercial API providers improve the usability of their APIs by following both recommended and "organic" conventions.

4.3 Schema Topology and Worst-Case Response Sizes

Queries resulting in huge responses may be computationally taxing, so practitioners point out the resulting challenge for providers to throttle such queries [24, 25]. The size of a response depends on three factors: the schema, the query, and the underlying data. In this section, we analyze each schema in our corpuses for the worst-case response size it enables with pathological queries and data.

A GraphQL query names all the data that it retrieves (cf. Figure 1). Provided a schema has no field that returns a list of objects, the response size thus directly corresponds to the size of the query. On the other hand, if a field can return a list of objects (e.g., `nodes` in Figure 1), nested sub-queries are applied to all the elements of the list (e.g., `name` in Figure 1). Therefore, nested object lists can lead to an explosion in response size.

From the schema we can compute K, the maximum number of nested object lists that can be achieved in a query. For example, if `Query` contains a field `repos:[Repo]`, and `Repo` contains a field `members:[User]` then $K = 2$. Without access to the underlying data, we assume that the length of all the retrieved object lists is bounded by a known constant D.[11]

Polynomial response. For a query of size n, the worst-case response size is $O\big((n-K) \times D^K\big)$ — at worst polynomial in the length D of the object lists. The proof is by induction over the structure of the query. As an illustration, consider the worst-case scenario of a query with maximum number of nested lists, K. Since the query must spend K fields to name the nested lists, each object at the deepest level can have at most $(n-K)$ fields and will be of size at most $(n-K)$. Each level returns D nested objects, plus one field to name the list. The size of each level k starting from the deepest one thus follows the relation: $s_k = D \times s_{k-1} + 1$ with $s_0 = (n-K)$. The response size is given by the top level K: $s_K = (n-K) \times D^K + \frac{D^K - 1}{D-1}$, that is, $O\big((n-K) \times D^K\big)$.[12]

Exponential response. If the schema includes a cycle containing list types (e.g., a type `User` contains a field `friends:[User]`), the maximum number of nested object lists is only bounded by the size of the query, i.e., $K < n$.[13] In that case the worst-case response size becomes $O(D^{n-1})$, that is, exponential in the

[11] In practice, the size of retrieved object lists are often explicitly bounded by slicing arguments (e.g., `first: 2` in Figure 1). See also §4.4.

[12] In Table 3, we use the slightly relaxed notion $O(n \times D^K)$.

[13] In GraphQL the first field is always `query`, and cannot be a list type.

size of the query. Consider for example the following query that requests names of third degree friends (size $n = 4$ and nesting $K = 3$). If every user has at least ten friends, the size of the response is $1 + 10 \times (1 + 10 \times (1 + 10 \times 1)) = 1111$.

```
query { friends(first: 10) { friends(first: 10) { friends(first: 10) { name } } } }
```

Table 3. Worst-case response size based on type graph analysis, where n denotes the query size, and D the maximum length of the retrieved lists.

Worst-case response		**Commercial** (16)	**GitHub** (8,399)	**GH-large** (1,739)
Exponential	$O(D^{n-1})$	14 (87.5%)	3,219 (38.3%)	1,414 (81.3%)
Polynomial	$O(n \times D^6)$	0 (0.0%)	6 (0.1%)	4 (0.2%)
Polynomial	$O(n \times D^5)$	0 (0.0%)	9 (0.1%)	4 (0.2%)
Polynomial	$O(n \times D^4)$	0 (0.0%)	34 (0.4%)	7 (0.4%)
Polynomial	$O(n \times D^3)$	1 (6.3%)	186 (2.2%)	40 (2.3%)
Quadratic	$O(n \times D^2)$	1 (6.3%)	785 (9.3%)	88 (5.1%)
Linear	$O(n \times D)$	0 (0.0%)	3,112 (37.1%)	182 (10.5%)
Linear	$O(n)$	0 (0.0%)	1,048 (12.5%)	0 (0.0%)

Results. We implemented an analysis for schema topographical connectedness based on the conditions for exponential and polynomial responses sizes outlined above, and applied it to our schema corpuses. As shown in Table 3, the majority of commercial (100.0%), GitHub (50.5%), and GitHub-large (89.5%) schemas have super-linear worst-case response sizes. This finding is of course not altogether surprising, as the key to super-linear response sizes is a particular and intuitive relational schema structure, and the purpose of GraphQL is to permit schema providers to describe relationships between types. However, the implication is that GraphQL providers and middleware services should plan to gauge the cost of each query by estimated cost or response size, or otherwise limit queries.

Finding 3: The majority of commercial, GitHub, and GitHub-large schemas have super-linear worst-case response sizes, and in the commercial and GitHub-large corpuses, they are mostly exponential. Providers need to consider throttling requests to their APIs to avoid the negative consequences of expensive queries, whether malicious or inadvertent.

4.4 Delimiting Worst-Case Response Sizes through Pagination

Queries with super-linear response sizes can become security threats, overloading APIs or even leading to denial-of-service. For commercial GraphQL providers, exponential response sizes pose a potential security risk (denial of service). Even polynomial response sizes might be concerning — e.g., consider the cost of returning the (very large) cross product of all GitHub repositories and users.

The official GraphQL documentation recommends that schema developers use one of two *pagination* techniques to bound response sizes [7]: **Slicing** refers

to the use of numeric arguments to index a subset of the full response set. The **connections pattern** introduces a layer of indirection to enable more complex pagination. The addition of *Edge* and *Connection* types allows schema developers to indicate additional relationships between types, and to paginate through a concurrently updated list (cf. schema described in Section 2).

Analysis. We used heuristics relying on names of fields and types to identify the use of pagination patterns within schemas.

For slicing, we identify fields that return object lists and accept numeric *slicing arguments* of scalar type `Int`. In our corpuses these arguments are commonly named `first`, `last`, and `limit`, or `size`. We use the presence of arguments with these names as an indication that slicing is in use. We differentiate schemas that use such arguments for slicing consistently, for some fields, or not at all.

For the connections pattern, we check schemas for types whose names end in `Connection` or `Edge` as proposed in the official GraphQL docs [7]. We again check for the use of slicing arguments on fields that return connections.

Table 4. Use of Slicing Arguments and Connections Pattern.

	Comm. (16)	**GitHub** (8,399)	**GH-large** (1,739)
Have fields returning object lists	16 (100.0%)	7,351 (87.5%)	1,739 (100.0%)
...with no slicing arguments	10 (62.5%)	5,335 (63.5%)	385 (22.1%)
...with slicing args. sometimes	6 (37.5%)	1,771 (21.1%)	1,265 (72.7%)
...with slicing args. throughout	0 (0.0%)	245 (2.9%)	89 (5.1%)
Have types with names matching `/Edge$/` and `/Connection$/`	9 (56.3%)	2,073 (24.7%)	1,365 (78.5%)
...with no slicing arguments	1 (6.3%)	1,397 (16.6%)	1,073 (61.7%)
...with slicing args. sometimes	2 (12.5%)	48 (0.6%)	31 (1.8%)
...with slicing args. throughout	6 (37.5%)	628 (7.5%)	261 (15.0%)

Results. Using our heuristics, Table 4 summarizes the use of the pagination patterns in our corpuses. In no corpus are these pagination patterns used consistently, strengthening the threat of the worst-case responses discussed in §4.3. For the schemas that do use pagination patterns, the commercial and GitHub-large schemas tend to use the more complex yet flexible connections pattern, while slicing alone is used inconsistently across all schemas.

Finding 4: No corpus consistently uses pagination patterns, raising the specter of worst-case response sizes. When pagination patterns are used, commercial and GitHub-large schemas tend to use the connections pattern, while slicing is not used consistently. Our worst-case findings from §4.3 urge the wider adoption of pagination.

5 Related Work

Our work is most closely related to that of Kim et al., who also collected and analyzed GraphQL schemas [21]. They analyzed 2,081 unique schemas mined from open-source repositories on GitHub. Our works are complementary. We use different mining techniques and conduct different analyses. For **mining**, to identify GraphQL schemas on GitHub, both works queried the GitHub API for filenames with GraphQL-themed substrings. We additionally proposed a novel schema stitching technique to repair incomplete schemas, which permitted us to recover thousands of schemas that their methodology would discard (§3.2). In **analysis**, we compared multiple corpuses, while they focused solely on schemas obtained from GitHub and did not distinguish between the larger and smaller schemas therein. Where our analyses overlap, our findings agree: in our GitHub schema corpus we report similar proportions of schemas using mutations (we: 67.9%, they: 70%) and subscriptions (we: 25.0%, they: 20%). Similarly, in our GitHub corpus we found a similar proportion of schemas with type cycles (we: 38.3%, they: 39.7%). Our analyses of naming conventions, worst-case response sizes, and pagination are novel.

Our worst-case response size analysis (§4.3) benefits from the work of Hartig and Pérez. They complemented the GraphQL specification [17] with a formal description for key parts of GraphQL [19, 20]. They also proved the existence of GraphQL schema-data (graph) combinations on which a query will have exponential-sized results (cf. [20, Propositions 5.2 and 5.3]) and gave an upper bound for the response size (cf. [20, Theorem 5.4]). In comparison, our analysis in §4.3 explicitly identifies object lists as the cause of the response size explosion, and we use this observation to provide a tighter upper bound.

The remaining academic literature on GraphQL focuses on the challenges of creating a GraphQL API. Several research teams have described their experiences exposing a GraphQL API or migrating existing APIs to GraphQL [16, 27, 28]. Others have described automatic techniques for migration [31] and testing [26].

Our work is similar in spirit to studies of REST(-like) APIs, which have focused on API design best practices [22, 23] or assessed API business models [18]. Because of the paradigmatic differences between GraphQL and REST (single endpoint, typed schema, queries formed by clients, etc.), this work complements existing ones.

6 Threats to Validity

Construct validity. In §4.3 we assume that response size is the primary measure of query cost. We leave to future work a more fine-grained analysis dependent on backend implementation details (e.g. resolver function costs).

Internal validity. Our name-based analyses depend on heuristics which could be inaccurate, although they are grounded in the grey literature where possible.

External validity. Our corpuses may not be representative of the true state of GraphQL schemas in practice, affecting the generalizability of our results.

The commercial corpus contains the 16 public commercial GraphQL APIs we could identify, well short of the 100+ companies that use GraphQL (presumably internally) [9]. We restricted the open-source corpus to statically defined schemas stored in GitHub. By analyzing the "GitHub large" schemas separately, we provide a better understanding of both (1) methodologically, the risks of treating all GitHub schemas alike, and (2) scientifically, the properties of larger schemas.

7 Conclusions

GraphQL is an increasingly important technology. We provide an empirical assessment of the current state of GraphQL through our rich corpuses, novel schema reconstruction methodology, and novel analyses. Our characterization of naming conventions can help developers adopt community standards to improve API usability. We have confirmed the fears of practitioners and warnings of researchers about the risk of denial of service against GraphQL APIs: most commercial and large open-source GraphQL APIs may be susceptible to queries with exponential-sized responses. We report that many schemas do not follow best practices and thus incompletely defend against such queries.

Our work motivates many avenues for future research, such as: refactoring tools to support naming conventions, coupled schema-query analyses to estimate response sizes in middleware (e.g. rate limiting), and data-driven backend design.

8 Acknowledgments

We are grateful to A. Tantawi, A. Kazerouni, and B. Pirelli for their feedback on the manuscript, and to O. Hartig for a helpful discussion.

References

1. APIs-guru/graphql-apis: A collective list of public GraphQL APIs, Available at https://github.com/APIs-guru/graphql-apis
2. GraphiQL: An in-browser IDE for exploring GraphQL, Available at https://github.com/graphql/graphiql
3. GraphQL Faker, Available at https://github.com/APIs-guru/graphql-faker
4. GraphQL Style conventions, Available at https://www.apollographql.com/docs/apollo-server/essentials/schema.html#style
5. Introduction to GraphQL, Available at https://graphql.org/learn/
6. JavaScript reference implementation for GraphQL, Available at https://github.com/graphql/graphql-js
7. Pagination, Available at http://graphql.github.io/learn/pagination/
8. Schemas and Types, Available at https://graphql.org/learn/schema
9. Whos using GraphQL?, Available at http://graphql.org/users
10. Introducing Yelp's Local Graph (2017), Available at https://engineeringblog.yelp.com/2017/05/introducing-yelps-local-graph.html

11. React, Relay and GraphQL: Under the Hood of The Times Website Redesign (2017), Available at https://open.nytimes.com/react-relay-and-graphql-under-the-hood-of-the-times-website-redesign-22fb62ea9764
12. GitHub GraphQL API v4 (2019), Available at https://developer.github.com/v4/
13. GraphQL and Shopify (2019), Available at https://help.shopify.com/en/api/custom-storefronts/storefront-api/graphql/
14. GraphQL Current Working Draft: Schema (2019), Available at https://facebook.github.io/graphql/draft/#sec-Executable-Definitions
15. GraphQL Current Working Draft: Schema (2019), Available at https://facebook.github.io/graphql/draft/#sec-Schema
16. Brito, G., Mombach, T., Valente, M.T.: Migrating to GraphQL: A Practical Assessment. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 140–150. IEEE (2019)
17. Facebook Inc.: GraphQL. Working Draft (June 2018), Available at https://facebook.github.io/graphql/
18. Gamez-Diaz, A., Fernandez, P., Ruiz-Cortes, A.: An Analysis of RESTful APIs Offerings in the Industry. In: International Conference on Service-Oriented Computing. pp. 589–604. Springer (2017)
19. Hartig, O., Pérez, J.: An initial analysis of Facebook's GraphQL language. In: CEUR Workshop Proceedings (2017)
20. Hartig, O., Pérez, J.: Semantics and Complexity of GraphQL. In: Conference on World Wide Web (WWW) (2018)
21. Kim, Y.W., Consens, M.P., Hartig, O.: An Empirical Analysis of GraphQL API Schemas in Open Code Repositories and Package Registries. In: Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW) (Jun 2019)
22. Palma, F., Gonzalez-Huerta, J., Moha, N., Guéhéneuc, Y.G., Tremblay, G.: Are RESTful APIs Well-Designed? Detection of their Linguistic (Anti) Patterns. In: Int. Conf. on Service-Oriented Computing. pp. 171–187. Springer (2015)
23. Petrillo, F., Merle, P., Moha, N., Guéhéneuc, Y.G.: Are REST APIs for cloud computing well-designed? An exploratory study. In: International Conference on Service-Oriented Computing. pp. 157–170. Springer (2016)
24. Rinquin, A.: Avoiding n+1 requests in GraphQL, including within subscriptions
25. Stoiber, M.: Securing Your GraphQL API from Malicious Queries
26. Vargas, D.M., Mayor, U., Sim, D.S., Blanco, A.F., Pablo, J., Alcocer, S., Torres, M.M., Bergel, A.: Deviation Testing: A Test Case Generation Technique for GraphQL APIs (2018)
27. Vázquez-Ingelmo, A., Cruz-Benito, J., García-Peñalvo, F.J.: Improving the OEEU's data-driven technological ecosystem's interoperability with GraphQL. In: Proceedings of the 5th International Conference on Technological Ecosystems for Enhancing Multiculturality - TEEM 2017. pp. 1–8 (2017)
28. Vogel, M., Weber, S., Zirpins, C.: Experiences on Migrating RESTful Web Services to GraphQL. In: ICSOC Workshops. Springer International Publishing (2017)
29. Wittern, E., Cha, A., Davis, J.C., Baudart, G., Mandel, L.: GraphQL Schema Collector. https://doi.org/10.5281/zenodo.3352421, accessible at https://github.com/ErikWittern/graphql-schema-collector
30. Wittern, E., Cha, A., Davis, J.C., Baudart, G., Mandel, L.: GraphQL Schemas. https://doi.org/10.5281/zenodo.3352419, accessible at https://github.com/ErikWittern/graphql-schemas
31. Wittern, E., Cha, A., Laredo, J.A.: Generating GraphQL-Wrappers for REST (-like) APIs. In: International Conference on Web Engineering. pp. 65–83. Springer (2018)