A Principled Approach to GraphQL Query Cost Analysis

Alan Cha
IBM Research, USA
alan.cha1@ibm.com

Erik Wittern*
IBM, Germany
erik.wittern@ibm.com

Guillaume Baudart
IBM Research, USA
Guillaume.Baudart@ibm.com

James C. Davis†
Purdue University, USA
davisjam@purdue.edu

Louis Mandel
IBM Research, USA
lmandel@us.ibm.com

Jim Laredo
IBM Research, USA
laredoj@us.ibm.com

ABSTRACT

The landscape of web APIs is evolving to meet new client requirements and to facilitate how providers fulfill them. A recent web API model is GraphQL, which is both a query language and a runtime. Using GraphQL, client queries express the data they want to retrieve or mutate, and servers respond with exactly those data or changes. GraphQL's expressiveness is risky for service providers because clients can succinctly request stupendous amounts of data, and responding to overly complex queries can be costly or disrupt service availability. Recent empirical work has shown that many service providers are at risk. Using traditional API management methods is not sufficient, and practitioners lack principled means of estimating and measuring the cost of the GraphQL queries they receive.

In this work, we present a linear-time GraphQL query analysis that can measure the cost of a query without executing it. Our approach can be applied in a separate API management layer and used with arbitrary GraphQL backends. In contrast to existing static approaches, our analysis supports common GraphQL conventions that affect query cost, and our analysis is provably correct based on our formal specification of GraphQL semantics.

We demonstrate the potential of our approach using a novel GraphQL query-response corpus for two commercial GraphQL APIs. Our query analysis consistently obtains upper cost bounds, tight enough relative to the true response sizes to be actionable for service providers. In contrast, existing static GraphQL query analyses exhibit over-estimates and under-estimates because they fail to support GraphQL conventions.

CCS CONCEPTS

• **Security and privacy** → **Denial-of-service attacks**; • **Software and its engineering** → *Domain specific languages.*

KEYWORDS

GraphQL, algorithmic complexity attacks, static analysis

*Most of the work performed while at IBM Research, USA.
†Most of the work performed while at Virginia Tech.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409670

ACM Reference Format:
Alan Cha, Erik Wittern, Guillaume Baudart, James C. Davis, Louis Mandel, and Jim Laredo. 2020. A Principled Approach to GraphQL Query Cost Analysis. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409670

1 INTRODUCTION

Web APIs are the preferred approach to exchange information on the internet. Requirements to satisfy new client interactions have led to new web API models such as GraphQL [30], a data query language. A GraphQL service provides a *schema*, defining the data entities and relationships for which clients can *query*.

GraphQL has seen increasing adoption because it offers three advantages over other web API paradigms. First, GraphQL reduces network traffic and server processing because users can express their data requirements in a single query [29]. Second, it simplifies API maintenance and evolution by reducing the number of service endpoints [37]. Third, GraphQL is strongly typed, facilitating tooling including data mocking [12], query checking [8], and wrappers for existing APIs [41]. These benefits have not been lost on service providers [26], with adopters including GitHub [5] and Yelp [2].

However, GraphQL can be perilous for service providers. A worst-case GraphQL query requires the server to perform an exponential amount of work [34], with implications for execution cost, pricing models, and denial-of-service [32]. This risk is not hypothetical — a majority of GraphQL schemas expose service providers to the risk of high-cost queries [40]. As practitioners know [1, 36, 38], *the fundamental problem for GraphQL API management is the lack of a cheap, accurate way to estimate the cost of a query.*

Existing dynamic and static cost estimates fall short (§2). Dynamic approaches are accurate but impractically expensive [27, 34], relying on interaction with the backend service and assuming specialized backend functionality [27]. Current static approaches are inaccurate and do not support GraphQL conventions [20, 23, 25].

We present the first provably correct static query cost analysis for GraphQL. We begin with a novel formalization of GraphQL queries and semantics (§3). After extending the formalization with simple *configuration information* to capture common schema conventions, we define two *complexity metrics* reflecting server and client costs for a GraphQL query (§4). Then we show how to compute upper bounds for a query's cost according to these metrics (§5). Our analysis takes linear time and space in the size of the query.

Our analysis is accurate and practical (§6). We studied 10,000 query-response pairs from two commercial GraphQL APIs. Unlike

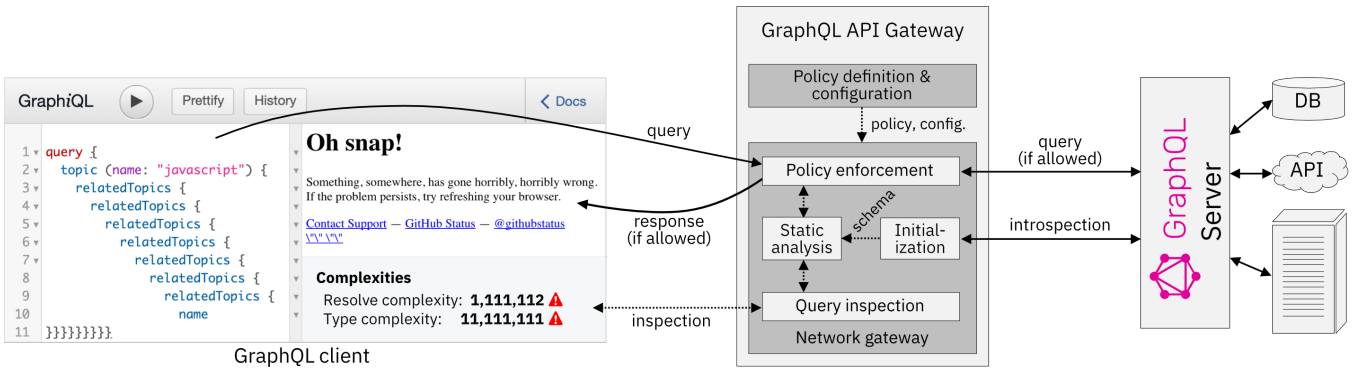

Figure 1: Proposed applications of our query analysis. The client's malicious query requests an exponentially large result from GitHub's GraphQL API. At the time of our study, GitHub permitted the shown query, but halted its execution after it exceeded a time limit. Using our techniques, client-side *query inspection* can provide feedback during composition (see *Complexities* inset). Server-side *policy enforcement* can reject queries and update rate limits based on provider-defined policies. We disclosed this denial of service vector to GitHub, and it has since been patched (§6.4.3).

existing analyses, our analysis obtains accurate cost bounds even for pathological queries. With minimal configuration, our bounds are tight enough to be actionable for service providers. Our analysis is fast enough to be used in existing request-response flows.

This paper makes the following contributions:

- We give a novel formalization of GraphQL semantics (§3).
- We propose two GraphQL query complexity measures (§4) that are used to estimate the cost of a query. We then prove a linear-time static analysis to obtain an upper bound for these measures on a given query (§5).
- We evaluate our analysis against two public GraphQL APIs and show that it is practical, accurate, and fast (§6). We also identify causes of over- and under-estimation in existing static analyses.
- We share the first GraphQL query-response corpus [31]: 10,000 unique query and response pairs from the GitHub and Yelp APIs.

We illustrate applications of our analysis by exploiting a flaw in GitHub's static analysis (Figure 1).[1] We issued an exponential-time query to GitHub. GitHub's analysis incorrectly estimated the query's cost, accepted it, and wasted resources until the evaluation timed out. Our analysis can help service providers avoid this situation. Some large queries are accidental, and our measure of *type complexity* would permit clients to understand the potential sizes of responses before issuing queries, reducing accidental service and network costs. Both type complexity and our measure of *resolve complexity* would permit service providers to understand a query's potential execution cost. Using these metrics will allow service providers to identify high-cost queries and respond appropriately.

2 BACKGROUND AND MOTIVATION

In this section, we motivate the need for GraphQL query cost analysis (§2.1) and then discuss existing query analyses (§2.2).

[1] GitHub's API also has a runtime defense, so the risk to their service was minimal.

2.1 Motivation

Our work is motivated by two aspects of software engineering practice: (1) the majority of real-world GraphQL schemas expose service providers to high-cost queries, and (2) existing strategies employed by service providers are inadequate.

High-complexity GraphQL schemas are common in practice. Hartig and Pérez showed that a GraphQL query can yield an exponential amount of data in the size of the query [34]. Such a query requests the nested retrieval of the same data, and is only possible if the schema defines self-referential relationships ("loops of lists"), and if the underlying data contains such relationships. Wittern *et al.* extended their analysis to identify schemas with polynomial-sized worst-case responses, and analyzed a corpus of GraphQL schemas for these properties [40]. In their corpus, they found that **over 80%** of the commercial or large-scale open-source schemas had exponential worst-case behavior, and that **under 40%** of *all* schemas guaranteed linear-time queries.

Many public GraphQL APIs do not document any query analysis. We manually studied the documentation for the 30 public APIs listed by *APIs.guru*, a community-maintained listing of GraphQL APIs [15]. We used public APIs listed as of February 28th, 2020; other GraphQL APIs are unlisted or private [26]. Disturbingly, 25 APIs (83%) describe neither static nor dynamic query analysis to manage access and prevent misuse. 22 APIs (73%) make no reference to rate limiting or to preventing malicious or overly complex requests. Three APIs (10%) perform rate limiting, but only by request *frequency*, ignoring the wide range of query complexities.

A few APIs have incorporated customized query and/or response analysis into their management approach. Five APIs (17%) describe analyzing GraphQL queries to apply rate limiting based on the estimated or actual cost of a query or response. GitHub [7], Shopify [17], and Contentful [4] estimate the cost of queries before executing them. Shopify and Yelp [19] update remaining rate limits by analyzing responses, i.e., the actual data sent to clients. But these approaches have shortcomings that are discussed in §6.4.

2.2 Existing GraphQL Query Cost Analyses

A GraphQL *query cost analysis* measures the cost of a query without fully executing it. Service providers can use such an analysis to avoid denial of service attacks, as well as for management purposes.

There are two styles of GraphQL query analysis: *dynamic* [34] and *static* [20, 23, 25]. The dynamic analysis of [34] considers a query in the context of the data graph on which it will be executed. Through lightweight query resolution, it steps through a query to determine the number of objects involved. This cost measure is accurate but expensive to obtain, because it incurs additional runtime load and potentially entails engineering costs [27].[2]

Static analyses [20, 23, 25] calculate the worst-case query cost supposing a pathological data graph. Because a static analysis assumes the worst, it can efficiently provide an upper bound on a query's cost without interacting with the backend. The speed and generality of static query analysis makes this an attractive approach for commercial GraphQL API providers.

Our static approach follows a similar paradigm as existing static analyses but we differ in several ways: (1) We provide two distinct definitions of query complexity, which is used to measure query cost; (2) Our analysis can be configured to handle common schema conventions to produce better estimates; (3) We build our analysis on formal GraphQL semantics and prove the correctness of our query complexity estimates; and (4) We perform the first evaluation of such an analysis on real-world APIs. Overall, our evaluation shows the benefits of a formal and configurable approach, identifying shortcomings in existing static analyses.

3 A NOVEL GRAPHQL FORMALIZATION

In this section we introduce GraphQL schemas and queries. Then, we give a novel formalization of the semantics of query execution based on the GraphQL specification [3] and reference implementation [13]. Compared to Hartig and Pérez [34], our semantics is more compact, closer to the concrete GraphQL syntax, and includes the context object for GraphQL-convention-aware static analysis.

3.1 GraphQL Schemas and Queries

For a visual introduction to GraphQL queries, see Figure 2. On the left is an excerpt of GitHub's API's schema. In the center is a sample query requesting a `topic` named `"graphql"`, the `name`s of two `relatedTopics`, the `totalCount` of stargazers, and the `name`s of two stargazers. The right side of the figure shows the server's response.

A GraphQL *schema* defines the data *types* that clients can query, as well as possible *operations* on that data. Types can be scalars (*e.g.,* `Int`, `String`), enumerations, object types defined by the provider (*e.g.,* `Topic`), or lists (*e.g.,* `[Topic]`). In addition, there are also input types, used to define object types for arguments. Each field of an object type is characterized by a name (*e.g.,* `relatedTopics`) and arguments used to constrain the matching data (*e.g.,* `first`). All schemas define a `Query` operation type, which contains fields that form the top-level entry points for queries [11]. Schemas may also define a `Mutation` operation type, which contains fields that allow queries to create, update, or delete data, or a `Subscription` operation type, which provides event-based functionality.

The syntax of a GraphQL *query* is as follows:[3]

$$
\begin{array}{rcll}
q & ::= & \text{label} : \text{field}\ (\text{args}) & \text{(basic query)} \\
 & | & \ldots \text{on type}\ \{q\} & \text{(filter)} \\
 & | & q\, q & \text{(concatenation)} \\
 & | & \text{label} : \text{field}\ (\text{args})\{q\} & \text{(nesting)}
\end{array}
$$

A **basic query**, $\text{label} : \text{field}\ (\text{args})$, requests a specific *field* of an object with a list of named arguments $args = a_1 : v_1, \ldots, a_n : v_n$. For example `topic(name: "graphql")` in Figure 2 queries the `topic` field with the argument `name: "graphql"`. The *label* renames the result of the query with an arbitrary name. In the GraphQL syntax, *label* can be omitted if it matches the *field*, and the list of arguments can be omitted if empty. A simple *field* is thus a valid query.

Inline fragments $\ldots \text{on type}\ \{q\}$ **filter** a query q on a type condition, only executing query q for objects of the correct *type*, e.g., `...on Starrable` in Figure 2. A query can also **concatenate** fields $q_1\, q_2$, or request a sub-field of an object via **nesting** with the form $\text{label} : \text{field}\ (\text{args})\{q\}$. Before execution, GraphQL servers validate incoming queries against their schema.

3.2 Query Execution

To support a schema, a GraphQL server must implement *resolver functions*. Each field of each type in the schema corresponds to a resolver in the GraphQL backend. The GraphQL runtime invokes the resolvers for each field in the query, and returns a data object that mirrors the shape of the query.

The evaluation of a query can be thought of as applying successive filters to a *virtual data object* that initially corresponds to the complete data graph. These filters follow the structure of the query and return only the relevant fields. For example, when executing the query in Figure 2, given the `Topic` whose name is `"graphql"`, the resolver for field `relatedTopics` returns a list of `Topic`s, and for each of these `Topic`s the resolver for `name` returns a `String`.

The indirection of resolver functions makes the semantics of GraphQL agnostic to the storage of the data. The data object is an access point, populated *e.g.,* from a database or external service(s) that a resolver contacts. A resolver must return a value of the appropriate type, but the origin of that value is up to the implementation.

Semantics. Formally, Figure 3 defines the semantics of the kernel language as an inductive function over the query structure. The formula $[\![q]\!](o, \text{ctx}) = o'$ means that the evaluation of query q on data object o with context *ctx* returns an object o'. The context tracks information for use deeper in a nested query. In our simplified semantics we track the type, field, and arguments of the parent.

Querying a single field $[\![\text{label} : \text{field}\ (\text{args})]\!](o, \text{ctx})$ calls a resolver function $\texttt{resolve}(\text{o}, \text{field}, \text{args}, \text{ctx})$ which returns the corresponding *field* in object o. The response object contains a single field *label* populated with this value. The interpetation the arguments *args* in the resolver is not part of the semantics; it is left to the service developers.

[2] In particular, their analysis repeatedly interacts with the GraphQL backend, and assumes that the backend supports cheap queries for response size. This is plausible if the backend is a traditional database, but GraphQL is backend agnostic (§3).

[3] We omit some "syntactic sugar" of GraphQL constructs. They can be expressed as combinations of our kernel elements.

```
schema { query: Query }
type Query { topic(name: String): Topic }
type Topic {
  relatedTopics(first: Int): [Topic]
  name: String
  stargazers(after: String, last: Int):
    StargazerConnection }
type StargazerConnection {
  totalCount: Int
  edges: [StargazerEdge]
  nodes: [User] }
type StargazerEdge {
  node: User
  cursor: String }
type User { name: String }
```

```
query {
  topic(name: "graphql") {
    relatedTopics(first: 2) {
      name
    }
    ...on Starrable {
      stargazers(last: 2, after: "Y3...") {
        totalCount
        edges { # Connections pattern
          node { name }
          cursor
        }
} } } }
```

```
{ "data": {
    "topic": {
      "relatedTopics": [
        {"name": "api"},
        {"name": "rest"}
      ],
      "stargazers": {
        "totalCount": 1252,
        "edges": [
          {"node": {"name": "XXX"},
           "cursor": "Y3V..."},
          {"node": {"name": "XXX"},
           "cursor": "Y3V..."}
        ]
} } } }
```

Figure 2: A GraphQL schema (left) with a sample query that uses the connections pattern (center) and response (right).

$$[\![\text{label} : \text{field}\ (\text{args})]\!](o, \text{ctx}) = \{\text{label} : \texttt{resolve}(\text{o}, \text{field}, \text{args}, \text{ctx})\}$$

$$[\![\ldots \text{on type}\ \{q\}]\!](o, \text{ctx}) = \begin{cases} [\![q]\!](o, \text{ctx}) & \text{if } \texttt{typeof}(o) = \textit{type} \\ \{\} & \text{otherwise} \end{cases}$$

$$[\![q_1\ q_2]\!](o, \text{ctx}) = \texttt{merge}([\![q_1]\!](o, \text{ctx}), [\![q_2]\!](o, \text{ctx}))$$

$$[\![\text{label} : \text{field}\ (\text{args})\{q\}]\!](o, \text{ctx}) = \begin{cases} \{\text{label} : [[\![q]\!](o_1, \text{ctx}'), \ldots, [\![q]\!](o_n, \text{ctx}')]\} & \text{if } o' = [o_1, \ldots, o_n] \\ \{\text{label} : [\![q]\!](o', \text{ctx}')\} & \text{otherwise} \end{cases}$$

where $o' = \texttt{resolve}(\text{o}, \text{field}, \text{args}, \text{ctx})$
and $\textit{ctx}' = \{\texttt{type} : \texttt{typeof}(o), \texttt{field} : \textit{field}, \texttt{args} : \textit{args}\}$

Figure 3: Semantics of GraphQL.

A fragment $[\![\ldots \text{on type}\ \{q\}]\!](o, \text{ctx})$ only evaluates the sub-query q on objects of the correct *type* ($\texttt{typeof}(o)$ returns the type of its operand). In the example of Figure 2, the field `stargazers` is only present in the response if the topic is a `Starrable` type.

Querying multiple fields $[\![q_1\ q_2]\!](o, \text{ctx})$ merges the returned objects, collapsing repeated fields in the response into one.[4]

A nested query $[\![\text{label} : \text{field}\ (\text{args})\{q\}]\!](o, \text{ctx})$ is evaluated in two steps. First, $\texttt{resolve}(\text{o}, \text{field}, \text{args}, \text{ctx})$ returns an object o'. The second step depends on the type of o'. If o' is a list $[o_1, \ldots, o_n]$, the returned object contains a field *label* whose value is the list obtained by applying the sub-query q to the all the elements $o_1, \ldots, o_n$ with a new context $\textit{ctx}'$ containing the type, field, and arguments list of the parent. Otherwise, the returned object contains a field *label* whose value is the object returned by applying the sub-query q on o' in the new context $\textit{ctx}'$. By convention the top-level field of the response, which corresponds to the `query` resolver, has an implicit label `"data"` (see *e.g.,* the response in Figure 2).

4 QUERY COMPLEXITY

A GraphQL query describes the structure of the response data, and also dictates the resolver functions that must be invoked to satisfy it (which resolvers, in what order, and how many times). We propose two complexity metrics intended to measure costs from the perspectives of a GraphQL service provider and a client:

Resolve complexity reflects the server's query execution cost.
Type complexity reflects the size of the data retrieved by a query.

GraphQL service providers will benefit from either measure, *e.g.,* leveraging them to inform load balancing, threat-prevention, resolver resource allocation, or request pricing based on the execution cost or response size. GraphQL clients will benefit from understanding the type complexity of a query, which may affect their contracts with GraphQL services and network providers, or their caching policies.

Complexity metrics can be computed on either a query or its response. For a **query**, in §5.2 we propose static analyses to estimate resolve and type complexities *before* its execution given minimal assumptions on the GraphQL server. For a **response**, resolve and type complexity are determined similarly but in terms of the fields and data in the response object.

The intuition behind our analysis is straightforward. A GraphQL query describes the size and shape of the response. With an appropriate formalization of GraphQL semantics, an upper bound on resolve complexity and type complexity can be calculated using weighted recursive sums. But unless it accounts for common GraphQL design practices, the resulting bound may mis-estimate complexities. In §6.4 we show this problem in existing approaches.

In the remainder of this section, we describe two commonly-used GraphQL pagination mechanisms. If a GraphQL schema and query uses these mechanisms, either explicitly (§4.1) or implicitly (§4.2), we can obtain a tighter and thus more useful complexity bound. Research reported that both of these conventions are widely used in real-world GraphQL schemas [40], so supporting them is also important for practical purposes.

4.1 GraphQL Pagination Conventions

At the scale of commercial GraphQL APIs, queries for fields that return lists of objects may have high complexity — *e.g.,* consider the (very large) cross product of all GitHub repositories and users. The official GraphQL documentation recommends that schema developers bound response sizes through pagination, using *slicing* or the *connections pattern* [10]. GraphQL does not specify semantics for such arguments, so we describe the common convention followed by commercial [2, 5, 18] and open-source [40] GraphQL APIs.

[4] $\texttt{merge}(o_1, o_2)$ recursively merges the fields of o_1 and o_2.

Resolvers can return lists of objects which can result in arbitrarily large responses – bounded only by the size of the underlying data. **Slicing** is a solution that uses *limit arguments* to bound the size of the returned lists (*e.g.,* `relatedTopics(first: 2)` in Figure 2).

The **connections pattern** introduces a layer of indirection for more flexible pagination, using virtual `Edge` and `Connection` types [10, 16]. For example, in Figure 2-left the field `stargazers` returns a single `StargazerConnection`, allowing access to the `totalCount` of stargazers and the `edges` field, returning a list of `StargazerEdge`s. This pattern requires limit arguments to target children of a returned object (*e.g.,* `stargazers(last: 2)` in Figure 2-middle applies to the field `edges`).

The size of a list returned by a resolver can thus depend on the current arguments and the arguments of the parent stored in the context. Ensuring that limit arguments actually bound the size of the returned list is the responsibility of the server developers:

Assumption 1. *If the arguments list (*args*) or the context (*ctx*) contains a limit argument (*$\mathrm{arg} : \mathrm{val}$*) the list returned by the resolver cannot be longer than the value of the argument (*val*), that is:*

$$\texttt{length(resolve(o,field,args,ctx))} \leq \mathrm{val}.$$

If this assumption fails, it likely implies a backend error.

Pagination, not panacea. While slicing and the connections pattern help to constrain the response size of a query and the number of resolver functions that its execution invokes, these patterns cannot prevent clients from formulating complex queries that may exceed user rate limits or overload backend systems. Our pagination-aware complexity analyses can statically identify such queries.

4.2 Configuration for Pagination Conventions

As we discuss in §6, ignoring slicing arguments or mis-handling the connections pattern can lead to under- or over-estimation of a query's cost. Understanding pagination semantics is thus essential for accurate static analysis of query complexity. Since GraphQL pagination is a convention rather than a specification, we therefore propose to complement GraphQL schemas with a configuration that captures common pagination semantics. To unify this configuration with our definitions of resolve and type complexity, we also include weights representing resolver and type costs. Here is a sample configuration for the schema from Figure 2:

```
resolvers:
  "Topic.relatedTopics":
    limitArguments: [first]
    defaultLimit: 10
    resolverWeight: 1
  "Topic.stargazers":
    limitArguments: [first, last]
    limitedFields: [edges, nodes]
    defaultLimit: 10
    resolverWeight: 1
```

```
types:
  Topic:
    typeWeight: 1
  Stargazer:
    typeWeight: 1
```

This configuration specifies pagination behavior for slicing and the connections pattern. In this configuration, resolvers are identified by a string "type.field" (*e.g.,* `"Topic.relatedTopics"`). Their limit arguments are defined with the field `limitArguments`. For slicing, the limit argument applies directly to the returned list (see `"Topic.relatedTopics"`). For the connections pattern, the limit argument(s) apply to children of the returned object (see `limitedFields` for `"Topic.stargazers"`). The `defaultLimit` field indicates the size of the returned list if the resolver is called without limit arguments. We must make a second assumption (using JavaScript dot and bracket notation to access the fields of an object):

Assumption 2. *If a resolver is called without limit arguments, the returned list is no longer than the configuration c's default limit.*

$$\texttt{length(resolve(o,field,args,ctx))} \leq \texttt{c.resolvers["type.field"].defaultLimit}$$

In the following, $\texttt{limit(c,type,field,args,ctx)}$ returns the maximum value of the limit arguments for the resolver "type.field" if such arguments are present in the arguments list *args* or the context *ctx*, and the default limit otherwise. If a resolver returns unbounded lists, the default limit can be set to ∞, but we urge service providers to always bound lists for their own protection.

From Assumptions 1 and 2, we have:

Property 1. *Given a configuration c and a data object o of type t, if the context ctx contains the information on the parent of o, we have:*

$$\texttt{length(resolve(o,field,args,ctx))} \leq \texttt{limit(c,t,field,args,ctx)}.$$

Concise configuration. Researchers have reported that many GraphQL schemas follow consistent naming conventions [40], so we believe that regular expressions and wildcards are a natural way to make a configuration more concise. For example, the expression `"/.*Edge$/.nodes"` can be used for configuring resolvers associated with `nodes` fields within all types whose names end in `Edge`. Or, the expression `"User.*"` can be used for configuring resolvers for all fields within type `User`.

5 GRAPHQL QUERY COST ANALYSIS

In this section we formalize two query analyses to estimate the type and resolve complexities of a query. The analyses are defined as inductive functions over the structure of the query, mirroring the formalization of the semantics presented in §3. We highlight applications of these complexities in §6.4.

Like other static query cost analyses (§2.2), our analysis returns upper bounds for the actual response costs. For example, if a query asks for 10 `Topic`s, our analysis will return 10 as a tight upper bound on the response size. If there are only 3 `Topic`s in the data graph, our analysis will over-estimate the actual query cost.

5.1 Resolve and Type Complexity Analyses

As mentioned in §4, we propose two complexity metrics: resolve complexity and type complexity. In this section, we formalize the resolve complexity analysis and then explain how to adapt the approach to compute the type complexity. We will work several complexity examples, relying on the $W_{1,0}$ *configuration*: a resolver weight of 1 for fields returning object and list of object and 0 for all other fields and the top-level operation resolvers (i.e. `query`, `mutation`, `subscription`), and a type weight of 1 for all objects (including those returned in lists) and 0 for all other types and the top-level operation types (i.e. `Query`, `Mutation`, `Subscription`).

Resolve Complexity. Resolve complexity is a measure of the execution costs of a query. Given a configuration c, the resolve complexity of a response r: $\texttt{rcx}(r, c)$ is the sum of the weights of the resolvers that are called to compute the response. Each resolver call populates

$$\texttt{qrcx}(\text{label}:\text{field}\ (\text{args}), \text{c}, t, \text{ctx}) = \texttt{c.resolvers["t.field"].resolverWeight}$$

$$\texttt{qrcx}(\ldots\text{on type}\ \{q\}, \text{c}, t, \text{ctx}) = \begin{cases} \texttt{qrcx}(q, \text{c}, t, \text{ctx}) & \text{if } t = \textit{type} \\ 0 & \text{otherwise} \end{cases}$$

$$\texttt{qrcx}(q_1\ q_2, \text{c}, t, \text{ctx}) = \texttt{qrcx}(q_1, \text{c}, t, \text{ctx}) + \texttt{qrcx}(q_2, \text{c}, t, \text{ctx})$$

$$\texttt{qrcx}(\text{label}:\text{field}\ (\text{args})\{q\}, \text{c}, t, \text{ctx}) = \begin{cases} w + l \times \texttt{qrcx}(q, \text{c}, t', \text{ctx}') & \text{if t}[\textit{field}] = [t'] \\ w + \texttt{qrcx}(q, \text{c}, \text{t}.\textit{field}, \text{ctx}') & \text{otherwise} \end{cases}$$

where w = `c.resolvers["t.field"].resolverWeight`
and l = `limit(c,t,field,args,ctx)`
and *ctx'* = {`type` : t , `field` : *field*, `args` : *args*}

Figure 4: Resolve complexity analysis. The analysis operates on the types defined in the schema starting from `Query`.

a field in the response. The resolve complexity of a response is thus the sum of the weights of each field. With the $W_{1,0}$ configuration, the resolve complexity of the response in Figure 2-right is 6: 1 `topic`, 1 `relatedTopics`, 1 `stargazers`, 1 `edges`, 2 `nodes`.

The query analysis presented in Figure 4 computes an estimate `qrcx` of the resolve complexity of the response. Each call to a resolver is abstracted by the corresponding weight, and the sizes of lists are bounded using the `limit` function. `limit` uses the limit argument if present; otherwise, it will use the default limit in the configuration.

The analysis is defined by induction over the structure of the query and mirrors the semantics presented in Figure 3 with one major difference: the complexity analysis operates solely on the types defined in the schema starting from the top-level `Query` type. The formula $\texttt{qrcx}(q, \text{c}, t, \text{ctx}) = x$ means that given a configuration c, the estimated complexity of a query q on a type t with a context *ctx* is $x \in \mathbb{N} \cup \{\infty\}$. For example, using the $W_{1,0}$ configuration, the resolve complexity *of the query* in Figure 2-middle is also 6.

Theorem 1. *Given a configuration c, the analysis of Figure 4 always returns an upper-bound of the resolve complexity of the response. For any query q over a data object o, and using* {} *to denote the empty initial context:*

$$\texttt{qrcx}(q, \text{c}, \texttt{Query}, \{\}) \geq \texttt{rcx}(\llbracket \text{q} \rrbracket(\text{o}, \{\}), c)$$

Proof. The theorem is proved by induction on the structure of the query. The over-approximation has two causes. First, `limit` returns an upper bound on the size of the returned list (Property 1). If the underlying data is sparse, the list will not be full; the maximum complexity will not be reached. Second, we express the complexity of $\texttt{merge}(o_1, o_2)$ as the sum of the complexity of the two objects. This is accurate if o_1 and o_2 share no properties, but if properties overlap then the merge will remove redundant fields (cf. §3.2). □

Type Complexity. Type complexity is a measure of the size of the response object. Given a configuration c, the type complexity of a response object r: $\texttt{tcx}(\text{r}, c)$ is the sum of the weights of the types of all objects in the response. Using the $W_{1,0}$ configuration, the type complexity of the response in Figure 2-right is 8: 1 `Topic`, 2 (related) `Topic`s, 1 `StargazerConnection`, 2 `StargazerEdge`s, 2 `User`s.

Similar to our resolve complexity analysis `qrcx` (Figure 4), our type complexity analysis bounds the response's type complexity without performing query execution. We call the estimated query type complexity `qtcx`. To compute `qtcx`, we tweak the first and final rules from the `qrcx` analysis:

(1) The call to a resolver is abstracted by the weight of the returned type $t' = \text{t}[\textit{field}]$.

$$\texttt{qtcx}(\text{label}:\text{field}\ (\text{args}), \text{c}, t, \text{ctx}) = \texttt{c.types[t'].typeWeight}.$$

(2) When a nested query returns a list ($\text{t}[\textit{field}] = [t']$), the type complexity must reflect the cost of instantiating every element. Every element thus adds the weight of the returned type t' to the complexity.

$$\texttt{qtcx}(\text{label}:\text{field}\ (\text{args})\{q\}, \text{c}, t, \text{ctx}) = l \times (w + \texttt{qtcx}(q, \text{c}, t', \text{ctx}'))$$

where w = `c.types[t'].typeWeight`.

With the $W_{1,0}$ configuration, the type complexity *of the query* in Figure 2 is also 8.

Theorem 2. *Given a configuration c, the type complexity analysis always returns an upper-bound of the type complexity of the response. For any query q over a data object o, and using* {} *to denote the empty initial context:*

$$\texttt{qtcx}(q, \text{c}, \texttt{Query}, \{\}) \geq \texttt{tcx}(\llbracket \text{q} \rrbracket(\text{o}, \{\}), c)$$

The proof is similar to the proof of Theorem 1.

5.2 Time/Space Complexity of the Analyses

The type and resolve complexity analyses are computed in one traversal of the query. The time complexity of both analyses is thus $O(n)$, where n is the size of the query as measured by the number of derivations required to generate it from the grammar of GraphQL. Both analyses need to track only the parent of each sub-query during the traversal. This implies that the space required to execute the analyses depends on the maximum nesting of the query which is at worst n. The space complexity is thus in $O(n)$.

We emphasize that these are static analyses – they do not need to communicate with backends.

5.3 Mutations and Subscriptions

So far we have considered only GraphQL queries. GraphQL also supports *mutations* to modify the exposed data and *subscriptions* for event-based functionality [3]. Our resolve complexity approach also applies to mutations, reflecting the execution cost of the mutation. API providers can use resolve weights to reflect costly mutations in resolve complexity calculations. However, our type complexity approach only estimates the size of the returned object, ignoring the amount of data modified along the way. Assuming the user provides the new data, computing the type complexity of arguments passed to mutation resolvers may give a reasonable approximation. We leave this for future work. Our analysis can also produce resolve and type complexities for subscription queries. Policies around subscriptions may differ, though. For rate-limiting, for example, the API provider could reduce the remaining rates based on complexities when a subscription is started, and replenish them once the client unsubscribes.

6 EVALUATION

We have presented our query analysis and proved its accuracy. In our evaluation we consider five questions:

Table 1: Characteristics of the evaluated APIs.

Schema	GitHub	Yelp
Number of object types	245	25
Total fields on all object types	1,569	121
Lines of Code (LoC)	22,071	760
Pagination w. slicing arguments	Yes	Yes
Pagination w. connections pattern	Yes	Yes
Configuration	**GitHub**	**Yelp**
Number of default limits	21	13
LoC (% of schema LoC)	50 (0.2%)	39 (5.1%)

RQ1: Can the analysis be *applied to real-world GraphQL APIs*, especially considering the required configuration?
RQ2: Does the analysis produce *cost upper-bounds* for queries to such APIs?
RQ3: Are these bounds *useful*, i.e. close enough to the actual costs of the queries for providers to respond based on the estimates?
RQ4: Is our analysis *cheap enough* for use in API management?
RQ5: How does our approach *compare* to other solutions?

Our analysis depends on a novel dataset (§6.1). We created the first GraphQL query-response corpus for two reputable, publicly accessible GraphQL APIs: GitHub [5] and Yelp [2]. Table 1 summarizes these APIs using the metrics from [40].

To answer RQ1, we discuss configuring our analysis for these APIs (§6.2). To answer RQ2-RQ4, we analyzed the predicted and actual complexities in our query-response corpus (§6.3). For RQ5, we compare our findings experimentally to open-source static analyses, and qualitatively to closed-source commercial approaches (§6.4).

6.1 A GraphQL Query-Response Corpus

Answering our research questions requires a multi-API query-response corpus. None existed so we created one. We automatically generated queries for GitHub and Yelp, and collected $5,000$ unique query-response pairs each from April 2019 to March 2020.

General approach. We developed and open-sourced a GraphQL query generator [31]. Its input is a GraphQL schema and user-supplied parameters. Following standard grammar-based input generation techniques [33], it generates queries recursively and randomly by building an abstract syntax tree and rendering it as a GraphQL query string. The *depth* probability dictates the likelihood of adding nested fields. The *breadth* probability controls the likelihood of adding additional adjacent fields into each query level.

Providing Argument Values. The AST generated by this approach includes *argument variables* that must be concretized. We populate these values using a *provider map* that maps a GraphQL argument to functions that generate valid values.

Queries for GitHub and Yelp. We tuned the query generation to our experimental design and context. We used breadth and depth probabilities of 0.5. For *numeric arguments*, we used normally distributed integer values with mean and variance of $\langle 5, 1.0\rangle$ (GitHub) and $\langle 7, 2.0\rangle$ (Yelp).[5] For *enum arguments*, we randomly chose a valid enum value. For *named arguments* (*e.g.,* GitHub projects, Yelp restaurants), we randomly chose a valid entity, *e.g.,* the 100 most-starred GitHub repositories. For *context-dependent arguments*, we used commonly-valid values, *e.g.,* `README.md` as a GitHub filepath.

Ethically, because we are issuing queries against public APIs, we omitted overly-complex queries and queries with side effects. We issued queries with: (1) estimated type or resolve complexity $\leq 1,000$; (2) nesting depth ≤ 10; and (3) `query`, not `mutation` or `subscription` operations.

Query realism. Our query generation algorithm yielded diverse queries for each API. To assess their realism, we compared them to example queries provided by each API. Yelp did not provide any examples, but GitHub's documentation includes 24 example queries [6]. We analyzed them using the $W_{1,0}$ configuration (cf. §5.1) to determine the size of a typical query. Only 14 could be analyzed using the same GitHub schema version. Of these, all but one had a type complexity of 302 or less. Therefore, we used a type complexity of ≤ 300 to categorize *typical queries*.

6.2 RQ1: Configuration

Our aim is a backend-agnostic query cost analysis. As prior work [27, 34] cannot be applied to estimate query cost for arbitrary GraphQL servers, we first assess the feasibility of our approach.

Per §4.2, our analysis requires GraphQL API providers to configure limit arguments, weights, and default limits. We found it straightforward to configure both APIs. Our configurations are far smaller than the corresponding schemas (Table 1).

Limit arguments. GitHub and Yelp uses consistent limit arguments names (`first` and `last` for GitHub, `limit` for Yelp), which we configured with regular expressions. GitHub has a few exceptional limit arguments, such as `limit` on the `Gist.files` field, which we configured with both strings and regular expressions. Following the conventions of the connections pattern [16], we set all `Connection` types to have the limited fields `edges` and `nodes` for GitHub. GitHub also has a few fields that follow the slicing pattern, so we did not set any limited fields for these. The Yelp API strictly follows the slicing pattern, so no additional settings were required.

Weights. For simplicity, we used a $W_{1,0}$ configuration (cf. §5.1). This decision permitted us to compare against the open-source static analysis libraries, but may not reflect the actual costs for GraphQL clients or service providers. For example, we set the type weights for scalars and enums to 0, supposing that these fields are paid for by the resolvers for the containing objects.

Default limits. We identified default limits for the 21 fields (GitHub) and 13 fields (Yelp). These fields are unpaginated lists or lists that do not require limit arguments. We determined these numbers using API documentation and experimental queries.

6.3 RQ2-RQ4: Complexity Measurements

Using this configuration, we calculated type and resolve complexities for each query-response pair in the corpus. Figure 5 summarizes the results for Yelp and GitHub using heat maps. Each heat map shows the density of predictions for request/response complexity cells. Cells above the diagonal are queries whose actual complexity we over-estimated, cells below the diagonal are under-estimates.

[5] Yelp has a maximum nesting depth. Larger values yielded complex, shallower queries.

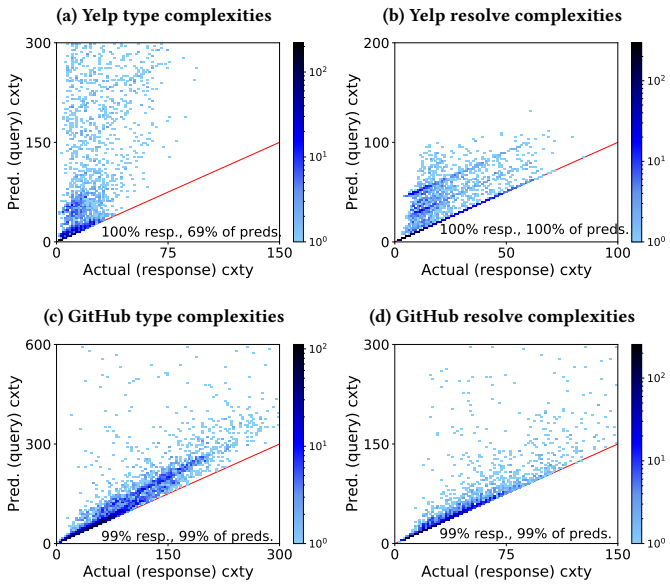

Figure 5: Actual (response) complexities and predicted (query) complexities, using our analysis on the corpus. Each figure has text indicating the percentage of responses that are shown (the remainder exceed the x-axis), and the percentage of the corresponding predictions that are shown (the remainder exceed the y-axis).

Table 2: Over-estimation of our type and resolve complexities. The first table summarizes our approach on all queries. The second shows the results for "typical" queries, *i.e.,* those with type complexities of up to 300 (cf. §6.1).

	Yelp (5,000)		GitHub (5,000)	
All queries	Resolve	Type	Resolve	Type
Underestimation	*None*	*None*	*None*	*None*
No overestimation	60.1%	21.8%	54.7%	8.4%
Overestimation <25%	63.0%	26.1%	84.9%	60.3%
Overestimation <50%	66.7%	30.1%	92.2%	84.0%

	Yelp (3,416)		GitHub (4,703)	
"Typical" queries	Resolve	Type	Resolve	Type
Underestimation	*None*	*None*	*None*	*None*
No overestimation	83.4%	31.9%	57.9%	9.0%
Overestimation <25%	85.3%	38.2%	88.3%	63.8%
Overestimation <50%	88.1%	44.1%	95.2%	87.3%

6.3.1 RQ2: Upper Bounds. In Figure 5, all points lie on or above the diagonal. In terms of our analysis, every query's predicted complexity is an upper bound on its actual complexity. This observation is experimental evidence for the results theorized in §5.1.

As an additional note, the striations that can be seen in Figure 5b are the result of fields with large default limits. Queries that utilize these fields will have similar estimated complexities.

6.3.2 RQ3: Tight Upper Bounds. Answering RQ2, we found our analysis computes upper bounds on the type and resolve complexities of queries. Other researchers have suggested that these cost bounds may be too far from the actual costs to be actionable [34].

Our data show that the bounds are tight enough for practical purposes, and are as tight as possible with a static, data-agnostic approach. Figure 5 indicates that our upper bound is close or equal to the actual type and resolve complexities of many queries — this can be seen in the high density of queries near the diagonals.

Our bounds are looser for more complex queries. This follows intuition about the underlying graph: larger, more nested queries may not be satisfiable by an API's real data. Data sparsity leads responses to be less complex than their worst-case potential. Table 2 quantifies this observation. It shows the share of queries for which our predictions over-estimate the response complexity by <25% and <50%. Over-estimation is less common for the subset of "typical" queries whose estimated type complexity is ≤ 300.

However, per the proofs in §5.2, our upper bounds are as tight as possible without dynamic response size information. The over-estimates for larger queries are due to data sparsity, not inaccuracy. For example, consider this pathological query to GitHub's API:

```
query {
  organization (login: "nodejs") {
    repository (name: "node") { issues (first: 100) { nodes {
      repository { issues (first: 100) { nodes {
        ... }}}}}}}}
```

This query cyclically requests the same `repository` and `issues`. With two levels of nesting, the query complexities are 10,203 (resolve) and 20,202 (type). If the API data includes at least 100 `issues`, the response complexities will match the query complexities.

6.3.3 RQ4: Performance. Beyond the functional correctness of our analysis, we assessed its runtime cost to see if it can be incorporated into existing request-response flows, *e.g.,* in a GraphQL client or an API gateway. We measured runtime cost on a 2017 MacBook Pro (8-core Intel i7 processor, 16 GB of memory).

As predicted in §5.2, our analysis runs in linear time as a function of the query and response size.[6] The median processing time was 3.0 ms for queries, and 1.1 ms for responses. Even the most complex inputs were fairly cheap; 95% of the queries could be processed in < 7.3 ms, and 95% of the responses in < 4 ms. The open-source analyses we consider in §6.4 also appear to run in linear time.

6.4 RQ5: Comparison to Other Static Analyses

In this section we compare our approach to state-of-the-art static GraphQL analyses (§2.2). To permit a fair comparison across different notions of GraphQL query cost, we tried to answer two practical *bound questions* for the GitHub API with each approach.

BQ1: How large might the response be?
BQ2: How many resolver functions might be invoked?

BQ1 is of interest to clients and service providers, who both pay the cost of handling the response. Various interpretations of "large" are possible, so we operationalized this as the number of distinct objects (non-scalars) in the response. BQ2 is of interest to service providers, who pay this cost when generating the response.

[6] We define *query size* as the number of derivations required to generate the query from the grammar presented in §3.1. It can be understood as the number of lines in the query if fields, inline fragments, and closing brackets each claim their own line.

We configured and compared our analysis against three open-source analyses experimentally, with results shown in Figure 6. The static GraphQL analyses performed by corporations are not publicly available, so we discuss them qualitatively instead.

6.4.1 Configuring our Analysis to Answer BQ1 and BQ2. Our measure of a query's type complexity can answer BQ1. Using the $W_{1,0}$ configuration (cf. §5.1), the type complexity measures the maximum number of objects that can appear in a response, *i.e.,* response size.

Our measure of a query's resolve complexity is suitable for answering BQ2. We assume that the cost of a resolver for a scalar or an enum field is paid for by some higher-level resolver. Thus, we again configured our analysis for GitHub using the $W_{1,0}$ configuration from §5.1. The resolve complexity resulting from this configuration will count each possible resolver function execution once.

6.4.2 Comparison to Open-Source Static Analyses. We selected open-source libraries for comparison. We describe their approaches in terms of our complexity measures, configure them to answer the questions as best as possible, and discuss their shortcomings.

Library selection. We considered analyses that met three criteria: (1) *Existence*: They were hosted on GitHub, discoverable via searches on `graphql {static|cost|query} {cost|analysis|complexity}`; (2) *Relevance*: They statically compute a query cost measure; and (3) *Quality*: They had ≥ 10 stars, a rough proxy for quality [28].

Three libraries met this criteria: libA [20], libB [25], and libC [23].

Understanding their complexity measures. At a high level, each of these libraries defines a cost measure in terms of the data returned by each resolver function invoked to generate a response. Users must specify the *cost* of the entity type that it returns, and a *multiplier* corresponding to the number of entities. These libraries then compute a weighted recursive sum, much as we do. Problematically, these libraries do not always permit users to specify the sizes of lists, leading to over- and under-estimates. In terms of the analysis from §5.1, the list field size l cannot always be obtained (see `qtcx`, last rule above Theorem 2). We discuss the details below.

Configuring the libraries to answer BQ1. We summarize our approach here. Our artifact includes the configuration details [31].

libA: We set the cost of all objects to 1 and all other types to 0. The library allows the maximum size of a list to be set. Therefore, we set the maximum sizes of unpaginated lists using values identified in §6.2. Unfortunately, the library cannot use arguments and consequently, cannot take advantage of pagination. To configure paginated lists, we instead set their maximum sizes to GitHub's maximum list size, 100 [7].

libB: As with libA, we set the cost of objects to 1 and other types to 0. We also leveraged libB's partial pagination support. We configured it to use the limit arguments of paginated lists. However, libB treats unpaginated lists are treated as a single (non-list) entity. It also does not support the connections pattern.

libC: libB and libC appear to be related. Their support for BQ1 is equivalent, and we configured them similarly.

Comparing outcomes on BQ1. Figure 6 illustrates the effectiveness of each approach when answering BQ1. The same query-response corpus was used in each case, and the response sizes were calculated using the method discussed in §5. The variation is on the y-axis, the predicted response size estimated from the query. As implied by our proofs of correctness, our approach consistently provides an upper bound on the actual response size (no under-estimation).

As mentioned, libA cannot use arguments as limits. Configuring all paginated lists to have a maximum list size equal to GitHub's maximum list size resulted in significant over-estimation. Additionally, this configuration caused in the striations in Figure 6b, which lie at intervals of 100 (the maximum list size); queries fall into bands based on the number of paginated lists they contain. In contrast to the striations found in Figure 5b, these fields nest within each other, allowing for repeated and regularly spaced stripes. To illustrate the overestimation, about 81.2% of libA's predictions were more than double the actual response size. In contrast, 84% of our analysis's predictions over-estimate less than 50% (cf. Table 2). Furthemore, the median over-estimation of our analysis (i.e. median error) is 19%, whereas that of libA is 634%. The 90% percentile over-estimation of our analysis is 66.0% and that of libA is 14,568.2%. As a result, our analysis conclusively performs better than libA.

libB and libC both over-estimate *and* under-estimate. Because they do not support default limits and treat unpaginated lists as a single (non-list) entity, they are prone to under-estimation. The under-estimation can compound geometrically when multiple unpaginated lists are nested. About 64% of their predictions were under-estimations. Additionally, because they do not support the connections pattern style of pagination and do not properly utilize limit arguments in these cases, they are prone to over-estimation. In any case, because libB and libC can under-estimate, they do not reliably produce upper bounds, which is a problem in security critical contexts. In contrast, our analysis consistently produces upper bounds and notably, tight upper bounds. Because our analysis is not at risk of this security problem, our analysis also performs better than libB and libC.

Configuring the libraries to answer BQ2. We were *unable* to configure these libraries to answer BQ2. Fundamentally, libA, libB, and libC measure costs in terms of the entities returned by the query, not in terms of the resolvers used to obtain the results. Trying to apply the multipliers to count resolvers will instead inflate them. The trouble is illuminated by our resolve complexity analysis (Figure 4): in the first clause of the final rule, the resolver multiplier should be used to account for the l uses of each child resolver, but the parent should be counted just once. In contrast, when answering BQ1, the type multiplier should be applied to both a field and its children.

This finding highlights the novelty of our notion of resolve complexity. We believe its ability to answer BQ2 also shows its utility.

6.4.3 Comparison to Closed-Source Analyses. GitHub and Yelp describe their analyses in enough detail for comparison. GitHub's analysis can approximate BQ1 and BQ2, while Yelp's cannot.

GitHub. GitHub's GraphQL API relies on two static analyses for rate limiting and blocking overly complex queries prior to executing them [7]. Both analyses support pagination via the connections pattern as described in §4.1. For BQ1, their *node limit* analysis disregards types associated with the connections pattern.[7] We can replicate this behavior with our analysis by setting the weights of types associated with the connections pattern to 0 and 1 otherwise. For BQ2, their *call's score* analysis only counts resolvers

[7] A possible explanation: virtual constructs like `Edge` and `Connection` types may not significantly increase the amount of effort to fulfill a query.

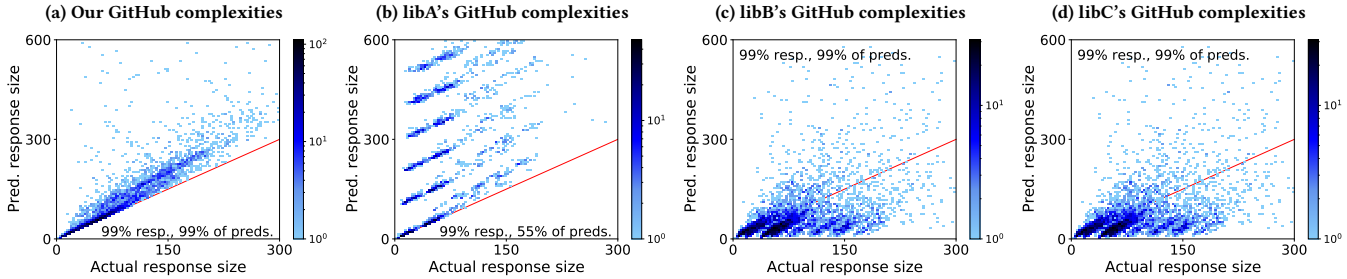

Figure 6: BQ1: Actual and predicted response sizes based on type complexity, from our analysis and the libraries on the GitHub data. libB and libC produce identical values under our configuration. All static approaches over-estimate due to data sparsity. The libraries have sources of over-estimation beyond our own. libB and libC also under-estimate (cells below the diagonal).

that return `Connection` types. We can also replicate this behavior by setting a weight of 1 to these resolvers and 0 otherwise. In any case, because GitHub's metrics cannot be weighted, they cannot distinguish between more or less costly types or resolvers.

GitHub's focus on the connections pattern may have caused them issues in the past. When our study began, GitHub shared a shortcoming with libB and libC: it did not properly handle unpaginated lists (which would not employ the connections pattern). We demonstrated the failure of their approach in Figure 1. We reported this possible denial of service vector to GitHub. They confirmed the issue and have since patched their analysis.

Yelp. Yelp's GraphQL API analysis has both static and dynamic components. Statically, Yelp's GraphQL API rejects queries with more than four levels of nesting. This strategy bounds the complexity of valid queries, but expensive queries can still be constructed with this restriction using large nested lists. Dynamically, Yelp then applies rate limits by executing queries and retroactively replenishing the client's remaining rate limits according to the complexity of the response. It is therefore possible to significantly exceed Yelp's rate limits by submitting a complex query when a client has a small quota remaining. Using our type complexity analysis, Yelp could address this problem by rejecting queries whose estimated complexities exceeded a client's remaining rates.

7 DISCUSSION AND RELATED WORK

Configuration and applicability. Our experiments show that our analysis is configurable to work with two real-world GraphQL APIs. Applying our analysis was possible because it is static, *i.e.,* it does *not* depend on any interaction with the GraphQL APIs or other backend systems. This contrasts with dynamic analyses, which depend on probing backends for list sizes [34]. Our analysis is more broadly applicable, and can be deployed separately from the GraphQL backend if desired, *e.g.,* in API gateways (cf. §8). The static approach carries greater risk of over-estimation, however, and API providers may consider a hybrid approach similar to GitHub's: a static filter, then dynamic monitoring.

We have identified three strategies for managing over-estimation. First, an unpaginated list field may produce responses with a wide range of sizes, leading our approach to overestimate. Schema designers may respond by paginating the list, which will bound the degree of overestimation. Second, in our tests we used the $W_{1,0}$ configuration, which assigned all types and resolvers the same weights. In contexts where different data and resolvers carry different costs, schema designers can tune the configuration appropriately. Lastly, service providers may resort to a hybrid static/dynamic system to leverage the graph data at runtime. The design of such a system is a topic for further research.

The value of formalization. Our formal analysis gives us provably correct bounds, provided that list sizes can be obtained from an analysis configuration. This contrasts with the more ad hoc approaches favored in the current generation of GraphQL analyses used by practitioners. A formal approach ensured that we did not miss "corner cases", as in the unpaginated list entities missed by libB, libC, and GitHub's internal analysis. Although our formalisms are not particularly complex, they guarantee the soundness and correctness missing from the state of the art.

Data-driven software engineering. Our approach benefited from an understanding of GraphQL as it is used in practice, specifically the use of pagination and naming conventions. Although pagination is not part of the GraphQL specification [30], we found that the GraphQL grey literature emphasized the importance of pagination. A recent empirical study of GraphQL schemas confirmed that various pagination strategies are widely used by practitioners [40]. We therefore incorporated pagination into our formalization (viz. that list sizes can be derived from the context object) and supported both of the widely used pagination patterns in our configuration. This decision differentiates our analysis from the state of the art, enabling us to avoid common sources of cost under- and over-estimation. In addition, the prevalence of naming conventions in GraphQL schemas inspired our support for regular expressions, which allowed our configuration answer BQ1 and BQ2 remarkably concisely. In contrast, the libraries we used required us to manually specify costs and multipliers for each of the (hundreds of) GitHub schema elements — they did not scale well to real-world schemas.

Bug finding. One surprising application of our analysis was as a bug finding tool. When we configured Yelp's API, we assumed that limit arguments would be honored (Assumption 1, §4). In early experiments we found that Yelp's resolver functions for the `Query.reviews` and `Business.reviews` fields ignore the limit argument. Yelp's engineering team confirmed this to be a bug. This interaction emphasized the validity of our assumptions.

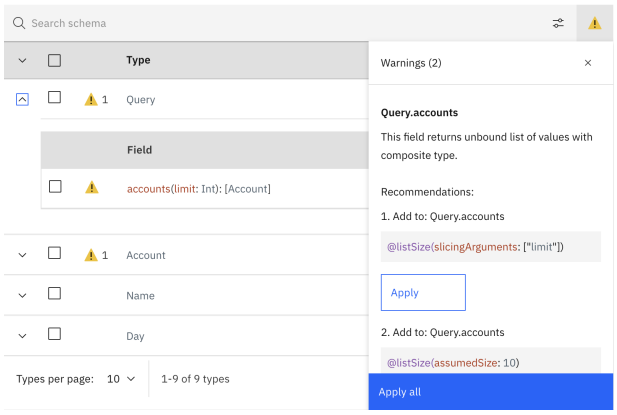

Figure 7: Screenshot of the configuration GUI in IBM's DataPower API gateway. The warnings indicate incomplete configuration and make recommendations.

Database Query Analyses. There has been significant work on query cost estimation for database query languages to optimize execution. Our analysis is related to the estimation of a database query's cardinality and cost [14]. However, typical SQL servers routinely optimize queries by reordering table accesses, which makes static cost evaluation challenging [35]. In comparison, our analysis takes advantage of the limited expressivity of GraphQL, and the information provided in the schema (e.g., via pagination mechanism) to guarantee robust and precise upper-bounds before execution.

8 APPLICATION EXAMPLE: API GATEWAY

We designed our GraphQL query cost analysis as a building block for a GraphQL *API gateway* that offers *API management* for GraphQL backends (Figure 1). We worked with IBM's product division to implement a GraphQL API gateway based on our ideas. Following patterns for API gateways for REST-like APIs [21, 22, 24], this gateway is backend agnostic, made possible by our data- and backend-independent query cost analysis. This gateway was incorporated into v10.0.0 of IBM's API Connect and DataPower products [39].

A weakness of our approach is the need for configuration. During productization we explored two ways to support this task: a graphical user interface (GUI) and automatic recommendations. The gateway automatically ingests the backend's schema using introspection [9]. Users can then configure using the GUI depicted in Figure 7. They can manually configure fields with weights, limit arguments, and/or default limits, either one at a time or bulk-apply to all types/fields matching a search. To mitigate the security risks of schemas with nested structures, the GUI automatically identifies some problematic fields and proposes an appropriate configuration based on schema conventions. For example, it flags fields that return lists and infers possible configurations based on type information.

9 THREATS TO VALIDITY

Construct validity. Our study does not face significant threats to construct validity. We believe our definitions of type complexity and resolve complexity are useful. We do not rely on proxy measures, but rather measure these complexities directly from real queries.

Internal validity. These threats come from configuration and query realism. In our evaluation, we created configurations for the GitHub and Yelp APIs. Errors would affect the accuracy of our bounds. Our evaluation showed that we did not make errors leading to under-estimation, but we may have done so for over-estimation.

Although RQ2 showed that our analysis produces upper bounds, in RQ3 our conclusions about the practicality of our bounds rely on the realism of our query-response corpus. Our evaluation is based on randomly generated queries, parameterized as described in §6.1. Some of our queries used popular projects, which are more likely to have associated data (decreasing over-estimates from data sparsity). Other queries lacked contextual knowledge and may result in "unnatural" queries unlikely to be filled with data. To avoid harming the public API providers, we bounded the complexity of the queries we issued, and this may have skewed our queries to be smaller than realistic queries. We plan to pursue a more realistic set of queries, *e.g.*, obtained through collaboration with a GraphQL API provider or by mining queries from open-source software.

External validity. Our work makes assumptions about the properties of GraphQL schemas and backend implementations that may not hold for all GraphQL API providers. For example, the complexity calculations depend on the presence of slicing arguments in queries, on resolver function implementations to enforce these limits, and on a proper configuration. By relying on default limits (§4.2), we enable our analysis to function even if slicing arguments are not enforced in (parts of) a schema. We demonstrated that proper configuration is possible even when treating the backend as a grey box, as we did when evaluating on the GitHub and Yelp APIs (§6.2).

10 CONCLUSION

GraphQL is an emerging web API model. Its flexibility can benefit clients, servers, and network operators. But its flexibility is also a threat: GraphQL queries can be exponentially complex, with implications for service providers including rate limiting and denial of service. The fundamental requirement for service providers is a cheap, accurate way to estimate the cost of a query.

We showed in our evaluation that existing *ad hoc* approaches are liable to both over-estimates and under-estimates. We proposed instead a principled approach to address this challenge. Grounded in a formalization of GraphQL semantics, in this work we presented the first provably-correct static query cost analyses. With proper configuration, our analysis offers tight upper bounds, low runtime overhead, and independence from backend implementation details. We accompany our work with the first GraphQL query-response corpus to support future research.

REPRODUCIBILITY

An artifact containing the GraphQL query generator, the query-response corpuses, library configurations, and corpus measurements can be found here: https://zenodo.org/record/3906094. Institutional policy precludes sharing our analysis prototype.

ACKNOWLEDGMENTS

The authors are grateful to the reviewers and to A. Kazerouni for their helpful feedback. We thank the IBM API Connect and DataPower teams for working with us on the GraphQL API Gateway.

REFERENCES

[1] 2016. How do you prevent nested attack on GraphQL/Apollo server? (2016). https://stackoverflow.com/questions/37337466/how-do-you-prevent-nested-attack-on-graphql-apollo-server/37338465

[2] 2017. Yelp – Introducing Yelp's Local Graph. (2017). https://engineeringblog.yelp.com/2017/05/introducing-yelps-local-graph.html

[3] 2018. GraphQL Specification. (June 2018). https://graphql.org/graphql-spec/

[4] 2019. Contentful – Query complexity limits. (2019). https://www.contentful.com/developers/docs/references/graphql/#/introduction/api-rate-limits

[5] 2019. GitHub – GraphQL API v4. https://developer.github.com/v4/

[6] 2019. GitHub – GraphQL Example Queries. https://github.com/github/platform-samples/tree/master/graphql/queri

[7] 2019. GitHub GraphQL API v4: GraphQL resource limitations. (2019). https://developer.github.com/v4/guides/resource-limitations/

[8] 2019. GraphiQL – An in-browser IDE for exploring GraphQL. https://github.com/graphql/graphiql

[9] 2019. GraphQL Docs: Introspection. (2019). https://graphql.org/learn/introspection/

[10] 2019. GraphQL Docs: Pagination. (2019). http://graphql.org/learn/pagination/

[11] 2019. GraphQL Docs: The Query and Mutation types. (2019). https://graphql.org/learn/schema/#the-query-and-mutation-types

[12] 2019. GraphQL Faker. https://github.com/APIs-guru/graphql-faker

[13] 2019. GraphQL.js – JavaScript reference implementation for GraphQL. https://github.com/graphql/graphql-js

[14] 2019. Oracle Database Documentation. https://docs.oracle.com/database

[15] 2019. Public GraphQL APIs. (2019). https://github.com/APIs-guru/graphql-apis

[16] 2019. Relay – Pagination Specification. (2019). https://facebook.github.io/relay/graphql/connections.htm

[17] 2019. Shopify – GraphQL Admin API rate limits. (2019). https://shopify.dev/concepts/about-apis/rate-limits#graphql-admin-api-rate-limits

[18] 2019. Shopify – Shopify Storefront API. (2019). https://shopify.dev/docs/storefront-api

[19] 2019. Yelp – GraphQL API Points-Based Daily Limit. (2019). https://www.yelp.com/developers/graphql/guides/rate$_1$imiting

[20] 2020. 4Catalyzer/graphql-validation-complexity: Query complexity validation for GraphQL.js. https://github.com/4Catalyzer/graphql-validation-complexity

[21] 2020. Google Apigee. https://cloud.google.com/apigee/

[22] 2020. IBM API Connect. https://www.ibm.com/cloud/api-connect

[23] 2020. pabru/graphql-cost-analysis: A Graphql query cost analyzer. https://github.com/pa-bru/graphql-cost-analysis

[24] 2020. RedHat 3Scale. https://www.3scale.net/

[25] 2020. slicknode/graphql-query-complexity: GraphQL query complexity analysis and validation for graphql-js. https://github.com/slicknode/graphql-query-complexity

[26] 2020. Who's using GraphQL? http://graphql.org/users

[27] Tim Andersson. 2018. *Result size calculation for Facebook's GraphQL query language.* B.S. Thesis. http://www.diva-portal.org/smash/get/diva2:1237221/FULLTEXT01.pdf

[28] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software* 146 (2018), 112–129.

[29] Gleison Brito, Thais Mombach, and Marco Tulio Valente. 2019. Migrating to GraphQL: A Practical Assessment. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 140–150. https://doi.org/10.1109/SANER.2019.8667986

[30] Lee Byron. 2015. GraphQL: A data query language. (2015). https://code.fb.com/core-data/graphql-a-data-query-language/

[31] Cha, Alan and Wittern, Erik and Baudart, Guillaume and Davis, James C., and Mandel, Louis and Laredo, Jim A. 2020. A Principled Approach to GraphQL Query Cost Analysis Research Paper Artifact. https://doi.org/10.5281/zenodo.3906094

[32] Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12* (Washington, DC) *(SSYM'03)*. USENIX Association, 29–44.

[33] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 206–215. https://doi.org/10.1145/1375581.1375607

[34] Olaf Hartig and Jorge Pérez. 2018. Semantics and Complexity of GraphQL. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 1155–1164. https://doi.org/10.1145/3178876.3186014

[35] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. 2012. 5, 11 (2012), 1555–1566.

[36] Arnaud Rinquin. 2017. Avoiding n+1 requests in GraphQL, including within subscriptions. (2017). https://medium.com/slite/avoiding-n-1-requests-in-graphql-including-within-subscriptions-f9d7867a257d

[37] Nick Shrock. 2015. GraphQL Introduction. (2015). https://reactjs.org/blog/2015/05/01/graphql-introduction.html

[38] Max Stoiber. 2018. Securing Your GraphQL API from Malicious Queries. (2018). https://blog.apollographql.com/securing-your-graphql-api-from-malicious-queries-16130a324a6b

[39] Rob Thelen. 2020. API Connect is making GraphQL safer for the enterprise. (2020). https://community.ibm.com/community/user/imwuc/blogs/rob-thelen1/2020/06/16/api-connect-is-making-graphql-safer-for-the-enterp

[40] Erik Wittern, Alan Cha, James C. Davis, Guillaume Baudart, and Louis Mandel. 2019. An Empirical Study of GraphQL Schemas. In *Proceedings of the 17th International Conference on Service-Oriented Computing (ICSOC)*, Vol. 11895.

[41] Erik Wittern, Alan Cha, and Jim A. Laredo. 2018. Generating GraphQL-Wrappers for REST (-like) APIs. In *International Conference on Web Engineering (ICWE '18)*. Springer International Publishing, 65–83. https://doi.org/10.1007/978-3-319-91662-0$_5$