GraphQLConf 2026

hosted by GraphQL Foundation

# When GraphQL Gets Expensive.

## Performance & Cost Patterns in Production Serverless Architectures

**Harpreet Siddhu**
**Shravanth Venkatesh**

Apollo Server • AWS Lambda • DynamoDB • Terraform

# Meet the Librarian Writing Your Cloud Bill.

*Your default GraphQL resolvers in plain English.*

## Inefficient Librarian

*your default resolver*

*"New arrivals + author bio"*

```
→  Log → book-1 (ID A-047)
     Walk to binder → bio
→  Log → book-2 (ID A-112)
     Walk to binder → bio
→  Repeat 98 more times…
```

**100 books. 101 trips. One Newsletter**

# The Schema

*Library domain. Substitute your own*


**Swap Patron for User, Loan for Order, Book for Product. E-commerce. Swap again — healthcare. Same chain.**

*DynamoDB on-demand • API Gateway HTTP • Terraform*

# Anti-Pattern 1: N+1 Resolver Cascade

*The inefficient librarian query - 100 patrons, 3 loans each = 300 loans, 300 books fetched, 300 authors fetched*

**Client sends this query:**

```
query {
  patrons {
    loans {
      book {
        author {
          name
        }
      }
    }
  }
}
```

**What Lambda executes:**

```
    1  DDB Scan       (patrons)
×100   DDB Query      (loans)
×300   DDB GetItem    (books)
×300   DDB GetItem    (authors)
```

**701 DDB API calls**

**$0.25 per million DDB reads**

**701 calls = $0.000175 / request**

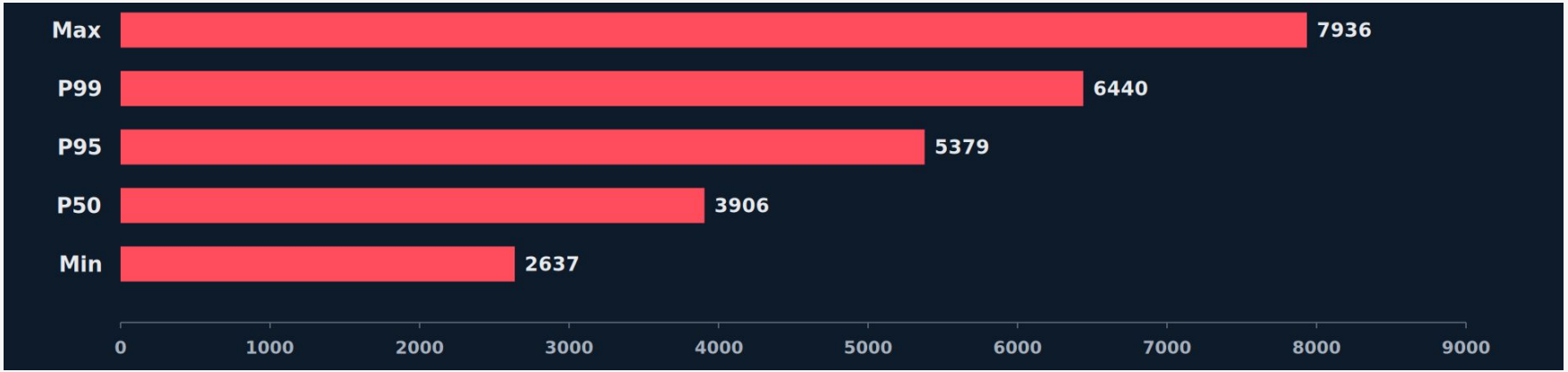
**At 1M req/day = $175/day**

# What N+1 Actually Costs

*10 concurrent users • Lambda 512MB • us-east-1*

| 6.4s | 5.4s | 701 | 43 |
|---|---|---|---|
| P99 latency | P95 latency | DDB calls / req | timeouts |

**Response time distribution — naive resolver under load:**




*All numbers from production-equivalent Artillery load tests.*

# Anti-Pattern 2: Unbounded Query Depth

*A circular query your schema allows — by default*

```
query DepthBomb {
  patrons {
    loans {
      book {
        loans {        # depth 4
          patron {
            loans {    # depth 6
              book {
                loans { # depth 8
                  patron {
                    name   # depth 10
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

**No depth limit**

- ✗ **503 Service Unavailable**
- ✗ **Lambda runs for 30 seconds**
- ✗ **Billed for 30s of compute**
- ✗ **Zero data returned**

**The same query — done right**

- ✓ **Rejected in < 50ms seconds**
- ✓ **Zero resolvers fired**
- ✓ **Zero DDB calls. Free.**
- ✓ Stopped before any damage

***1,800× more expensive than it has to be***

# Anti-Pattern 3: Over-fetching

*Carrying the whole shelf when you needed one book*

**Over-fetch query**

```
query OverFetch {
  books {
    id
    title                ← used
    isbn                 ✗
    publishedYear        ✗
    description          ✗  500 chr
    availableCopies
    totalCopies          ✗
    author {
      name               ← used
      nationality        ✗
      birthYear          ✗
      biography          ✗  300 chr
    }
  }
}
```

**Minimal query**

```
query Minimal {
  books {
    id
    title
    availableCopies
    author {
      name
    }
  }
}
```

**DynamoDB: same RCU cost — both read full items. Over-fetching costs: 11× network • Lambda RAM • serialization time.**

# Anti-Pattern 4: Resolver Fan-out at Scale

*"I'll use Promise.all — that fixes N+1, right?"*

```
// The "smart fix"
return Promise.all(
  loans.map(loan =>
    fetchBook(loan.bookId)
  )
);


// 1 user fires 300 calls. All at once.
// 10 users = 3,000 calls. At the same moment.
// Burst limit → throttling → retries → cold starts
```

**Promise.all**

**300 GetItem calls**

*Burst, throttle, retry*

*vs*

**300 calls or 3? Same query. Same data.**

# Solution 1: DataLoader

*Batch and cache all resolver calls within a single request*

| | |
|---|---|
| 1 | Resolver calls loader.load(bookId) → returns a Promise (no DB call yet — just registers the ID) |
| 2 | GraphQL runs ALL resolvers concurrently in one event loop tick |
| 3 | DataLoader collects all pending IDs: ["book-1", "book-2", ... "book-300"] |
| 4 | ONE BatchGetItem per 100 IDs → 3 total round trips (instead of 300) |
| 5 | All 300 Promises resolve simultaneously |

**Key insight:** DataLoader exploits the event loop's natural batching window. **300 load() calls = 3 BatchGetItem calls. Not 300.**

# DataLoader in Code

*Two files. One changed line per resolver.*

### loaders.ts — the batch function

```
const bookLoader = new
  DataLoader(
    async (ids) => {
      const res = await ddb
        .send(new BatchGetCommand({
          RequestItems: {
            "library-books": {
              Keys: ids.map(
                id => ({ id }))
            }
          }
        }));
      const items = res
        .Responses["library-books"];
      return ids.map(id =>
        items.find(i => i.id === id));
    },
    { cache: true }
  );
```

### Resolver — one line changes

```
// BEFORE (N+1):
Loan: {
  book: async (loan) =>
    ddb.send(
      new GetCommand({
        Key: { id: bookId }
      }))                    ← 1 call
}

// AFTER (DataLoader):
Loan: {
  book: async (loan, _, ctx) => {
    return ctx.loaders
      .bookLoader
      .load(bookId);      ← batched
  }
}
```

⚠️ **Always create a fresh DataLoader per request — never module-level. Module-scope caches leak user A's data into user B's response.**

# Same Query • Same Data • One Pattern Change

*10 concurrent users • before and after DataLoader*

| | Naive N+1 | DataLoader |
|---|---|---|
| DDB calls / request | 701 | 4 |
| P50 latency | 3,906 ms | 699 ms |
| P95 latency | 5,379 ms | 1,043 ms |
| P99 latency | 6,440 ms | 1,495 ms |
| Timeouts / 1,215 req | 43 | 0 |
| DDB cost / 1M req | ~$175 | ~$1 |

**175× fewer DDB calls • 4.3× faster P99 • $174 saved per million requests**

# Solution 2: Schema Guardrails

*Reject expensive queries before a single resolver fires*

**Depth limit — 2 lines**

```
import depthLimit from
  'graphql-depth-limit';

const server = new
  ApolloServer({
    typeDefs,
    resolvers,
    validationRules: [
      depthLimit(7)  ← all you need
    ],
  });

// depth 7: valid  ✓
// depth 9: rejected  ✗
```

**Complexity scoring**

```
async didResolveOperation
  ({ document, schema }) {
    const complexity =
      getComplexity({
        schema,
        query: document,
        estimators: [
          simpleEstimator({
            defaultComplexity: 1
          })
        ]
      });
    if (complexity > 200) {
      throw error;
    }
}
```

```
No limit  →  503 after 30.2s  ·  full timeout  ·  $0.00025
Depth: 7  →  400 in < 50ms  ·  0 resolvers  ·  $0.00000058
1,800× more expensive without the limit.
```

# Solution 3: Response Caching

*Cache hit = zero Lambda compute = zero DDB cost*

```
ApolloServerPluginResponseCache({
  ttl: 60,  // seconds
})

// Request 1 (cache miss):
//   4 DDB calls → 699ms

// Request 2 (cache hit):
//   0 DDB calls → 12ms   ← FREE

// 1,000 users, same query within 60s
// = cost of 1 request
```

**TTL strategy:**

```
authors   →  1 hour     (stable)
books     →  60s        (changes often)
loans     →  0          (real-time)
```

**Caching layers**

**DataLoader cache**

Per-request dedup • TTL: one request

**Response cache**

Across requests • TTL: 60s

**API GW cache**

HTTP-level • TTL: minutes

**Persisted queries**

Pre-approved IDs • blocks arbitrary queries

# The Real Cost: Per Million Requests

*AWS us-east-1 • Lambda 512MB • same query throughout*

| Pattern | P99 | DDB | Lambda $ | DDB $ | Total |
|---|---|---|---|---|---|
| 🔴 **Naive N+1** | 6,440 ms | 701 | $14.40 | $175 | **$189.40** |
| 🟢 **+ DataLoader** | 1,495 ms | 4 | $7.20 | $1 | **$8.20** |
| 🟢 **+ Guardrails** | 1,495 ms | 4 | $7.20 | $1 | **$8.20** |
| 🟢 **+ Cache 60s** | 12 ms | ~0 | $0.36 | $0.03 | **$0.39** |

*Lambda: $0.0000166/GB-sec • DynamoDB: $0.25/million reads*

## 99.8% cost reduction

**$189.40 per million → $0.39 per million**

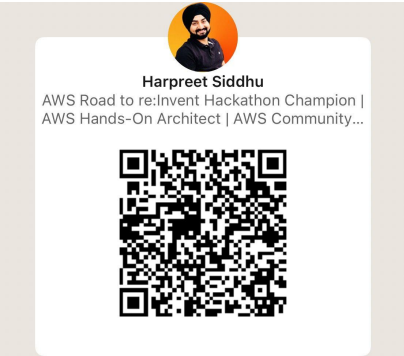
# 5 Rules — Ship These on Day One

1. **DataLoader for every list resolver**
   If a resolver can be called N times, it goes through a loader. Always. No exceptions.
2. **depthLimit(7) before production**
   Two lines. If a query needs more, have the conversation. Raise it intentionally.
3. **Score query complexity, set a budget**
   One afternoon to tune. Permanent protection. Let the error message teach clients.
4. **Cache by entity freshness, not by default**
   Authors = 1hr. Books = 60s. Loans = 0. Match TTL to how often data changes.
5. **Instrument with X-Ray before you scale**
   AWS_XRAY_SDK_ENABLED=true. CloudWatch alarms. Cost is invisible until you measure it.

# Questions?

## When GraphQL Gets Expensive

**Harpreet Siddhu • Shravanth Venkatesh**





**Runnable code — all four patterns + Terraform:**

`https://github.com/hsiddhu2/graphql-cost-performance-demo`

GraphQLConf 2026
hosted by GraphQL Foundation