# Resource limitations

**In this article**

The GitHub GraphQL API has limitations in place to protect against excessive or abusive calls to GitHub's servers.

## Node limit

To pass schema validation, all GraphQL API calls must meet these standards:

- Clients must supply a `first` or `last` argument on any connection.
- Values of `first` and `last` must be within 1-100.
- Individual calls cannot request more than 500,000 total nodes.

### Calculating nodes in a call

These two examples show how to calculate the total nodes in a call.

1. Simple query:

```
query {
  viewer {
    repositories(first: 50) {
      edges {
        repository:node {
          name

          issues(first: 10) {
            totalCount
            edges {
              node {
                title
                bodyHTML
              }
            }
          }
        }
      }
    }
  }
}
```

Calculation:

```
50         = 50 repositories
 +
50 x 10   = 500 repository issues

           = 550 total nodes
```

2 Complex query:

```
query {
  viewer {
    repositories(first: 50) {
      edges {
        repository:node {
          name

          pullRequests(first: 20) {
            edges {
              pullRequest:node {
                title

                comments(first: 10) {
                  edges {
                    comment:node {
                      bodyHTML
                    }
                  }
                }
              }
            }
          }

          issues(first: 20) {
            totalCount
            edges {
              issue:node {
                title
                bodyHTML

                comments(first: 10) {
                  edges {
                    comment:node {
                      bodyHTML
                    }
                  }
                }
              }
            }
          }
        }
      }
    }

    followers(first: 10) {
      edges {
        follower:node {
          login
        }
      }
    }
  }
}
```

Calculation:

```
50              = 50 repositories
 +
50 x 20         = 1,000 pullRequests
 +
50 x 20 x 10    = 10,000 pullRequest comments
 +
50 x 20         = 1,000 issues
 +
50 x 20 x 10    = 10,000 issue comments
 +
10                = 10 followers

                  = 22,060 total nodes
```

## Rate limit

The GraphQL API limit is different from the REST API's rate limits.

Why are the API rate limits different? With GraphQL, one GraphQL call can replace multiple REST calls. A single complex GraphQL call could be the equivalent of thousands of REST requests. While a single GraphQL call would fall well below the REST API rate limit, the query might be just as expensive for GitHub's servers to compute.

To accurately represent the server cost of a query, the GraphQL API calculates a call's **rate limit score** based on a normalized scale of points. A query's score factors in first and last arguments on a parent connection and its children.

- The formula uses the `first` and `last` arguments on a parent connection and its children to pre-calculate the potential load on GitHub's systems, such as MySQL, Elasticsearch, and Git.
- Each new connection has its own point value. Points are combined with other points from the call into an overall rate limit score.

Rate limiting is **disabled** by default for your GitHub Enterprise Server instance. When rate limiting is disabled, there is no limit to how many GraphQL points you can use.

However, a site administrator can enable rate limits for your GitHub Enterprise Server instance. If enabled, the rate limit is configurable, with a default of 200 points per hour. For more information, see "Configuring rate limits."

### Returning a call's rate limit status

With the REST API, you can check the rate limit status by inspecting the returned HTTP headers.

With the GraphQL API, you can check the rate limit status by querying fields on the `rateLimit` object:

```
query {
  viewer {
    login
  }
  rateLimit {
    limit
    cost
    remaining
    resetAt
  }
}
```

- The `limit` field returns the maximum number of points the client is permitted to consume in a 60-minute window.
- The `cost` field returns the point cost for the current call that counts against the rate limit.
- The `remaining` field returns the number of points remaining in the current rate limit window.
- The `resetAt` field returns the time at which the current rate limit window resets in

ISO 8601 format.

## Calculating a rate limit score before running the call

Querying the `rateLimit` object returns a call's score, but running the call counts against the limit. To avoid this dilemma, you can calculate the score of a call before you run it. The following calculation works out to roughly the same cost that `rateLimit { cost }` returns.

1. Add up the number of requests needed to fulfill each unique connection in the call. Assume every request will reach the `first` or `last` argument limits.

2. Divide the number by **100** and round the result to get the final aggregate cost. This step normalizes large numbers.

**Note**: The minimum cost of a call to the GraphQL API is **1**, representing a single request.

Here's an example query and score calculation:

```
query {
  viewer {
    login
    repositories(first: 100) {
      edges {
        node {
          id

          issues(first: 50) {
            edges {
              node {
                id

                labels(first: 60) {
                  edges {
                    node {
                      id
                      name
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

This query requires 5,101 requests to fulfill:

- Although we're returning 100 repositories, the API has to connect to the viewer's account **once** to get the list of repositories. So, requests for repositories = **1**
- Although we're returning 50 issues, the API has to connect to each of the **100** repositories to get the list of issues. So, requests for issues = **100**
- Although we're returning 60 labels, the API has to connect to each of the **5,000** potential total issues to get the list of labels. So, requests for labels = **5,000**
- Total = **5,101**

Dividing by 100 and rounding gives us the final score of the query: **51**