Highly Parallel Actor-Based Framework for Extracting a Snapshot of GitHub Using GraphQL

By:

Kirtan Dharmesh Jhaveri
MS Computer Science, Spring 2024
UIN: 667911068

Primary Advisor:
Mark Grechanik

Secondary Reviewer:
Abolfazl Asudeh

TABLE OF CONTENTS

Abstract

Github repositories are of paramount importance in software development. Github repositories not only simplify the lifecycle and continuous development process for developers but also act as datasets for researchers. These researchers search repositories to extract patterns and trends that drive new libraries, frameworks, and improvements for existing or obsolete code bases. This project is a solution to automate GitHub repository retrieval by user-defined criteria. This application was developed using Scala, Akka, Cassandra, GraphQL, ZIO, and Caliban. The application provides a solution for efficient and scalable extraction tasks. This framework handles large datasets by combining Akka's actor modeling for operations and GraphQL's ability to streamline responses. This combination of technologies meets the needs of software researchers and developers by enabling them to efficiently retrieve data related to their custom requirements while being scalable to handle data from multiple repositories at the same time. This report justifies the technology selection, implementation details, limitations, and possible future enhancements of the framework, highlighting its impact on software engineering research.

1. Introduction

The rapid development of software engineering practices has called for tools to aggregate and analyze huge amounts of data in repositories hosted on Github. Such repositories are more than code stores; they are datasets that provide valuable information for software development, testing, and research. However, the process of selecting and curating these repositories manually would require human intervention to search Github and examine each repository's details individually to confirm if they all met a predetermined set of requirements. A few examples of the requirements would be whether the repository is empty, what is the primary programming language, if issues have been opened, and whether those issues have been resolved, which would be indicated by their connection to commits.

Our framework is a creative and dynamic way to fetch GitHub repositories automatically. While it would be simple to say, why not just use "Git clone [7]"? This framework's main objective is to help users eliminate the laborious task of matching the objective requirements they search for in a repository. It is a tool for researchers and developers to access customized datasets. Its automated approach improves the reliability, scalability, and simplicity of information retrieval, including repository information, repository issues, and issue-related commits.

The need for this tool stems from the software engineering research community increasingly needing tailored datasets to validate emerging tools and methodologies. This demand is accompanied by the requirement for a system to satisfy such requirements while abstracting away the internal workings of a platform like Github from its users.

2. Technologies

The technologies used for the framework were selected to ensure high performance and scalability while adhering to the project goals of parallel processing and integration with GitHub's API. The subsequent subsections will draw comparisons and contrasts between the other widely used technologies and provide an explanation for the choice made.

2.1. Scala

It supports functional programming paradigms in addition to concurrency and parallelism features. Scala's abstractions let developers write clean code to implement complex asynchronous operations. Furthermore, Scala's interoperability with JDK libraries grants access to a huge set of tools and resources that extend the application’s flexibility and make it less verbose than Java.

2.2. Actor Model

The actor model is a conceptual model of concurrent computation. It considers "actors" to be the universal primitives of concurring computation. In this model, an actor is any entity that:

- Receive messages.
- Determines what computation to perform based on the message.
- Sends messages or replies to other actors.
- Spawns new child actors

Actors handle messages individually, so they can work independently of other actors without introducing lock contention and race conditions that are typical in multithreaded architectures. This model naturally produces a kind of reactive programming: responsive, resilient, elastic, and message-driven programming. This structure provides some advantages, like better management of concurrent tasks and simpler parallel operations, thereby improving resource utilization and performance. Actors abstract concurrent behavior, so developers can concentrate on logic instead of thread management.

The actor system in Akka is scalable because it requires low resources and low overhead for message passing, enabling this tool to scale up across distributed environments. Akka also supports robust fault tolerance features like supervision and error handling to ensure stable operation after graceful recovery from failures/errors.

2.2.1. Actor Model Example

Imagine a customer service application that takes requests from many clients. Each client request may involve authentication, data retrieval, and response generation tasks, which must be done quickly to maintain high throughput and low latency.

Here, each task is managed by its own actor:

- Authentication Actor: Handles user authentication. This actor verifies user credentials on the arrival of a new client request and sends the result to a data retrieval agent.
- Data Retrieval Actor: Responsible for obtaining user information from databases. It receives a message from the Authentication Actor (containing user credentials verification) and retrieves the required data based on the credentials' validity.
- Response Generation Actor: Formats retrieved data into a presentable format and responds to the client.

Pseudocode example:

```
// Actor definitions
class AuthenticationActor extends Actor {
  def receive = {
    case LoginRequest(username, password) =>
      if (validateCredentials(username, password)) {
        dataRetrievalActor ! FetchData(username)
      } else {
        sender() ! LoginFailed("Invalid credentials")
      }
  }
  def validateCredentials(username: String, password: String):
Boolean = {
    // Check the credentials against a credentials database or
service
    return database.verifyUser(username, password)
  }
}
class DataRetrievalActor extends Actor {
  def receive = {
    case FetchData(username) =>
      val userData = database.fetchUserData(username)
      responseGenerationActor ! GenerateResponse(userData)
  }
}
class ResponseGenerationActor extends Actor {
  def receive = {
    case GenerateResponse(userData) =>
      val response = formatResponse(userData)
      sender() ! response
  }
  def formatResponse(userData: UserData): String = {
    // Convert user data into a presentable format
    return s"Welcome back, ${userData.name}! Your last login was at
${userData.lastLogin}."
  }
}
// Example of sending a message to the AuthenticationActor
val authActor = system.actorOf(Props[AuthenticationActor],
"authActor")
authActor ! LoginRequest("user123", "password")
```

These actors act independently and send asynchronous messages. That means while one actor waits for a database response, it releases its thread to the pool, which another actor can use for another task. This nonblocking behaviour maximizes resource utilization and system scalability.

2.2.2. What Akka has to offer and why Akka over Erlang OTP?

Isolation: Each actor in Akka deals with errors locally, while isolating failure prevents system stability. If the Data Retrieval actor fails because of an issue with the database, it will not impact the Authentication actor or the Response Generation actor.

Concurrency: Akka manages thousands of actors in one JVM instance. This model scales well with system expansion, handling more client requests without a linear increase in resource use.

Resource Efficiency: Actors are very light in memory and computation and can have millions of concurrent actors performing tasks.

These features are offered by Erlang OTP, which was one of the first languages that made the actor model popular. However, Akka integration with the JVM aids us in leveraging the already-available Java libraries. Moreover, it also has the message delivery rule “at least once and exactly once" [4], which could significantly reduce performance penalties in terms of extra messages being sent and increase reliability for this data-intensive framework.

2.3. GraphQL

A few reasons pushed us to use the GraphQL API [3] endpoint instead of traditional REST endpoints from GitHub. GraphQL, a data query language developed by Facebook in 2012 and released publicly in 2015, is more flexible and efficient.[8] Unlike REST, where data is typically requested from many endpoints to build a full dataset, GraphQl lets clients specify what data they want in a single request. This

level of detail in data fetching greatly minimizes network overhead and latency, enabling faster and more responsive interactions with GitHub. Since GraphQL fetches only what is requested in a query by the user, less data is transmitted over the network, which is an advantage for mobile applications where bandwidth or performance are critical. GraphQL also supports dynamic query construction and schema evolution without affecting existing queries because it is introspective. This adaptability makes it a good choice for applications with an evolving data model that enables seamless API transitions and updates without breaking existing functionality.

2.3.1. GraphQL features include

- Highly Typed Schema: The GraphQL API is defined by its schema. This strong typing allows the API to validate queries against the schema, so queries ask for only what's possible and give clear and helpful errors.
- Single Endpoint: GraphQL uses one endpoint to handle API versioning and does not need to manage multiple endpoints for various data views.
- Real-Time Data with Subscriptions: GraphQL allows real-time data updates via subscriptions; the server pushes updates to clients in real time.

2.3.2. Understanding GraphQL with an Example

Consider a web application that lets users see their profile, posts, comments on posts, and friends' activities. With REST, fetching this data would typically require calls to multiple endpoints: one for the user profile, one for posts, and one for friends. The fact that each endpoint returns extra data that the client does not need could cause inefficiencies.

With GraphQL, a single query could be written to fetch all this data at once, tailored to the application's needs. [2]

Below is an example of a GraphQl query:

```
Query GetUserProfile($userId: ID!) {
  user(id: $userId) {
    name
    posts {
      title
      comments {
        content
        author {
          name
        }
      }
    }
    friends(last: 5) {
      name
      activity {
        date
        type
      }
    }
  }
}
```

2.4. Caliban: Caliban is a functional library used to create GraphQL servers and clients in Scala. In our framework, Caliban is used to create GraphQL queries using a SelectionBuilder. “A SelectionBuilder[Origin, A] is a selection from a parent type “Origin” that returns a result of type "A"." [1]

3. Implementation

The implementation section specifies the main components, their interactions, and how the messages that flow influence the application behavior.

3.1. Main: The main object represents the entry point of the application. It initializes the Acka actor system, “spawns” the RootActor, and sends a start message. However, before diving directly into the actor implementation, we need to

understand how the call to the Github API is made using two different GraphQL queries.

3.2. GraphQl queries using Caliban

3.2.1. RepoQuery

```
object RepoQuery extends ZIOAppDefault {
  def run: ZIO[Any, Throwable, Any] =
    //Combining the fields to be sent in the query
    val repository: SelectionBuilder[Repository, RepoInfoList] =
Repository.name ~ Repository.ownerInterface(owner =
RepositoryOwner.login) ~ Repository.diskUsage ~
Repository.hasIssuesEnabled ~ Repository.isArchived ~
Repository.isDisabled ~ Repository.isEmpty ~
Repository.primaryLanguage(Language.name) ~ Repository.url

    val Rquery: SelectionBuilder[RootQuery,
Option[List[Option[Option[Option[RepoInfoList]]]]]] =
      Query.search(first = Some(searchFirst), query =
s"$searchLanguage", `type` = REPOSITORY)(
          SearchResultItemConnection.
            edges(SearchResultItemEdge.
              nodeOption(onRepository = Some(repository))
))

    val call1 =
sendRequest(Rquery.toRequest(githubGraphqlEndpoint, useVariables =
true))

    val result:=
      call1
        .provideLayer(HttpClientZioBackend.layer())
        .tap(response => printLine(s"Response: $response"))

    result
}
```

```
def sendRequest[T](req: Request[Either[CalibanClientError, T],
Any]): RIO[SttpBackend[Task, ZioStreams with WebSockets], T] =
    ZIO
      .serviceWithZIO[SttpBackend[Task, ZioStreams with
WebSockets]] { backend =>
        req.headers(Header("Authorization", s"Bearer
$githubOauthToken")).send(backend)
      }
      .mapError { error =>
        logger.error(s"Error during request: $error")
        error
      }
      .map(_.body)
      .absolve
```

The code excerpts above show how the application uses ZIO and Caliban to create GraphQL queries to fetch information about repositories. “ZIO is a purely functional, type-safe, composable library for asynchronous, concurrent programming in Scala." [6][9] ZIO uses dependency injection to provide the query with an HTTP backend layer using the sendRequest method, which is required in order to carry out a network request. The query fetches repository name, owner, disk usage, issue status, archive status, disabled status, empty status, primary language, and URL.

Search parameters for this query, which could include a programming language or any search string, are provided by the user in a configuration file and submitted to the application. This information is used to filter out what repositories we might be interested in for further use, and based on a preset condition, we filter repositories, and for each repository, the IssueQuery is executed with the help of actors.

3.2.2. IssueQuery

Similar to RepoQuery, this class constructs a graphQL query to get issues related to a GitHub repository. It contains fields like issue ID, title, body, and associated commits, which include the commit oid and message.

Parameters like repository name and owner are provided dynamically from the response from the RepoQuery.

3.3. RepoActor

The RepoActor object is triggered when repository information is needed, as demonstrated by the code snippet below, which highlights a key aspect of the design. To fetch repository information, the RepoActor executes the RepoQuery, which is basically a ZIO effect. We execute this effect as a "future" so that it runs asynchronously and the actor remains non-blocking while the repository information is fetched.

```
object RepoActor {
  // Define messages
  sealed trait Message
  case class Start(message: String, replyTo:
ActorRef[RootActor.Message]) extends Message

  def apply(): Behavior[Message] = Behaviors.receive { (context,
message) =>
    implicit val ec: ExecutionContext = context.executionContext
    message match {
      case Start(msg, replyTo) =>
        val runtime = Runtime.default
        Unsafe.unsafe { implicit unsafe =>
          val queryEffect = RepoQuery.run
          val future = runtime.unsafe.runToFuture(queryEffect)
          future.onComplete {
            case scala.util.Success(value) =>
              // Handle the successful result
              replyTo ! RootActor.RepoActorReply(value)
              case scala.util.Failure(exception) =>
              // Handle the failure (e.g., log error)
          }
        }
        Behaviors.same
    }
  }
}
```

3.3.1. RepoActor Behavior

It reacts to the start message indicating the start of repository queries and executes RepoQuery. As soon as the query results are received, it replies to the RootActor with repository information.

3.4. IssueActor:

Similar to the RepoActor , the IssueActor is spawned when the response from the RepoActor has been received and is processed. It executes an instance of the IssueQuery based on a future so that the ZIO effect is executed asynchronously in a non-blocking manner. However, the only difference is that since multiple IssueActors run in parallel, they all execute an instance of the IssueQuery class, whereas the RepoActor was executing a singleton object RepoQuery. The reason to create an instance for each IssueActor is to eliminate them sharing an object (which would be analogous to shared memory). This helps us achieve the best performance by enabling the framework to be completely parallel and asynchronous.

3.4.1. IssueActor Behavior

It receives messages containing repository details. Following the message, it executes an instance of the IssueQuery to get issues and related commits for a given repository. After receiving the results, it formats the issues, replies to the RootActor, and inserts the information into the database, including the issues' details and commits that are linked to the issues that were opened in the repository.

3.5. RootActor

The root actor is at the core of the application workflow. It handles messages regarding repository information and issue fetching. It serves as the parent to the RepoActor and IssueActor.

3.5.1. RootActor Behavior

It gets messages such as Start (to start a process) and Repo ActorReply (for handling responses from repository actors). It spawns a RepoActor when it gets the Start message and sends a message to the RepoActor.

When RepoActor replies, the RootActor processes repository information, writes data to the database based on a condition (which can be replaced by a lambda function), and spawns an IssueActor instance for fetching issues for each repository that passes the condition.It tracks active issue fetches from the IssueActor and ends when all fetches are finished.

3.6. Cassandra

The information retrieved via the framework is stored in Cassandra. Cassandra runs as a Docker container for ease of setup and management.

- Containerization with Docker: “Docker is a lightweight, portable way to deploy applications"[10] such as databases like Cassandra. By containerizing Casandra, the framework maintains consistency across environments and simplifies setup.
- Persistence: Data fetched from GitHub, like repository information, issues,and commits, are persistently stored in Cassandra. This allows the application to keep querying this data after a restart or failure.
- Statements from CQL: For database operations, we use CQL (Cassandra Query Language) to add repository ,issue and commit data to the database.
- Async: Asynchronous execution is used when interacting with Cassandra to avoid blocking behavior and to optimize resource usage. This enables the application to perform a number of database operations simultaneously without blocking the main thread.

4. Limitations

4.1. Dependency on search engine accuracy:

The application uses a search query to pull repository data from user input. Although the application lets users define search criteria in detail, accuracy and relevance depend on how repository data is' indexed and 'presented by GitHub'. For example, if somebody searches the Fibonacci sequence, the tool might show a repository whose descriptions and documentation contain this particular string, even if the repository content isn’t about the Fibonacci algorithm itself. This problem is compounded when the repository creator misleads and rambles information about their projects that could retrieve poor or irrelevant quality repositories. So, while our system allows efficient data querying,the quality and relevance of search results are influenced by external factors that are outside the system's control.

4.2. GitHub API Rate Limiting

Another limitation of the system is the fact that GitHub caps the number of repositories accessed per query at 100. Though the total allowable quantity in API calls within a certain time limit is higher, this per-query limit restricts the amount that may be pulled instantly. Even though GitHub provides mechanisms like cursors for pagination, permitting sequential data retrieval across several queries, the framework currently doesn't implement this particular feature. This restricts the tool to fetching huge datasets in one call.

5. Future Scope

Cursor-based page pagination might be added around GitHub's rate cap. This would enable the application to obtain larger datasets via a series of sequential API calls. Integration with other version control platforms would be much more usable and appealing if the application supported platforms like GitLab and Bit Bucket. An improved user interface with intuitive controls and customizable dashboards might make the application usable by users of different technical skills and interests.

6. Conclusion

This framework has demonstrated the process of automating the task of retrieving and examining GitHub repositories. Built on Scala, Akka, Cassandra, GraphQL, and Caliban Client, it satisfies the essential needs of software development research: efficiently accessing and managing Github repository data. It meets most operational demands, but additional improvements might allow its further expansion and adaptation. Hopefully, this system will evolve over time to better serve the research environment.

References:

1. GhostDogPR. "Caliban Client Documentation." https://ghostdogpr.github.io/caliban/docs/client.html.
2. GraphQL. "GraphQL Queries." https://graphql.org/learn/queries/.
3. GitHub. "GitHub GraphQL API Documentation." https://docs.github.com/en/graphql.
4. Akka. "Akka Documentation." https://akka.io/docs/.
5. Scala Language. "Scala Documentation." https://docs.scala-lang.org/.
6. ZIO Contributors. "ZIO Overview - Getting Started." https://zio.dev/overview/getting-started.
7. Git. "Git Clone Documentation." https://git-scm.com/docs/git-clone.
8. Facebook Engineering. "GraphQL: A Data Query Language." https://engineering.fb.com/2015/09/14/core-infra/graphql-a-data-query-language/.
9. Scalac.io. "ZIO: Functional and Concurrent Programming in Scala." https://scalac.io/zio/#:~:text=ZIO%20is%20a%20purely%20functional,lot%20of%20concurrency%20and%20speed.
10. Docker. "Docker Overview." https://docs.docker.com/get-started/overview/.