

# MRVA for CodeQL

Michael Hohn

Technical Report 20250224

## Contents

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>MRVA System Architecture Summary</b>           | <b>1</b> |
| <b>2</b> | <b>Distributed Query Execution in MRVA</b>        | <b>2</b> |
| 2.1      | Execution Overview . . . . .                      | 2        |
| 2.2      | System Structure Overview . . . . .               | 2        |
| 2.3      | Messages and their Types . . . . .                | 3        |
| <b>3</b> | <b>Symbols and Notation</b>                       | <b>4</b> |
| <b>4</b> | <b>Full Round-Trip Representation</b>             | <b>4</b> |
| <b>5</b> | <b>Result Representation</b>                      | <b>5</b> |
| <b>6</b> | <b>Execution Loop in Pseudo-Code</b>              | <b>6</b> |
| <b>7</b> | <b>Execution Loop in Pseudo-Code, declarative</b> | <b>7</b> |
| <b>8</b> | <b>Execution Loop in Pseudo-Code, algorithmic</b> | <b>8</b> |
| <b>9</b> | <b>Execution Loop in Pseudo-Code, hybrid</b>      | <b>8</b> |

## 1 MRVA System Architecture Summary

The MRVA system is organized as a collection of services. On the server side, the system is containerized using Docker and comprises several key components:

- **Server:** Acts as the central coordinator.
- **Agents:** One or more agents that execute tasks.
- **RabbitMQ:** Handles messaging between components.
- **MinIO:** Provides storage for both queries and results.
- **HEPC:** An HTTP endpoint that hosts and serves CodeQL databases.

On the client side, users can interact with the system in two ways:

- **VSCode-CodeQL:** A graphical interface integrated with Visual Studio Code.
- **gh-mrva CLI:** A command-line interface that connects to the server in a similar way.

This architecture enables a robust and flexible workflow for code analysis, combining a containerized back-end with both graphical and CLI front-end tools.

The full system details can be seen in the source code. This document provides an overview.

## 2 Distributed Query Execution in MRVA

### 2.1 Execution Overview

The *MRVA* system is a distributed platform for executing *CodeQL* queries across multiple repositories using a set of worker agents. The system is containerized and built around a set of core services:

- **Server:** Coordinates job distribution and result aggregation.
- **Agents:** Execute queries independently and return results.
- **RabbitMQ:** Handles messaging between system components.
- **MinIO:** Stores query inputs and execution results.
- **HEPC:** Serves CodeQL databases over HTTP.

Clients interact with MRVA via VSCode-CodeQL (a graphical interface) or `gh-mrva` CLI (a command-line tool), both of which submit queries to the server.

The execution process follows a structured workflow:

1. A client submits a set of queries  $\mathcal{Q}$  targeting a repository set  $\mathcal{R}$ .
2. The server enqueues jobs and distributes them to available agents.
3. Each agent retrieves a job, executes queries against its assigned repository, and accumulates results.
4. The agent sends results back to the server, which then forwards them to the client.

This full round-trip can be expressed as:

$$\text{Client} \xrightarrow{\mathcal{Q}} \text{Server} \xrightarrow{\text{enqueue}} \text{Queue} \xrightarrow{\text{dispatch}} \text{Agent} \xrightarrow{\mathcal{Q}(\mathcal{R}_i)} \text{Server} \xrightarrow{\mathcal{Q}(\mathcal{R}_i)} \text{Client} \quad (1)$$

where the Client submits queries to the Server, which enqueues jobs in the Queue. Agents execute the queries, returning results  $\mathcal{Q}(\mathcal{R}_i)$  to the Server and ultimately back to the Client.

A more rigorous description of this is in section 4.

### 2.2 System Structure Overview

This design allows for scalable and efficient query execution across multiple repositories, whether on a single machine or a distributed cluster. The key idea is that both setups follow the same structural approach:

- **Single machine setup:**
  - Uses *at least 5 Docker containers* to manage different components of the system.
  - The number of *agent containers* (responsible for executing queries) is constrained by the available *RAM and CPU cores*.
- **Cluster setup:**
  - Uses *at least 5 virtual machines (VMs) and / or Docker containers*.
  - The number of *agent VMs* is limited by *network bandwidth and available resources* (e.g., distributed storage and inter-node communication overhead).

Thus:

- The functional architecture is identical between the single-machine and cluster setups.
- The primary difference is in *scale*:
  - A single machine is limited by *local CPU and RAM*.
  - A cluster is constrained by *network and inter-node coordination overhead* but allows for higher overall compute capacity.

## 2.3 Messages and their Types

The following table enumerates the types (messages) passed from Client to Server.

| Type Name           | Field  | Type  |
|---------------------|--|---|
| ServerState         | NextID<br>GetResult<br><br>GetJobSpecByRepold<br>SetResult<br>GetJobList<br>GetJobInfo<br>SetJobInfo<br>GetStatus<br>SetStatus<br>AddJob | () → int<br>JobSpec → IO (Either Error AnalyzeResult)<br>(int, int) → IO (Either Error JobSpec)<br>(JobSpec, AnalyzeResult) → IO ()<br>int → IO (Either Error <b>[AnalyzeJob]</b> )<br>JobSpec → IO (Either Error JobInfo)<br>(JobSpec, JobInfo) → IO ()<br>JobSpec → IO (Either Error Status)<br>(JobSpec, Status) → IO ()<br>AnalyzeJob → IO () |
| JobSpec             | sessionID<br>nameWithOwner   | int<br>string   |
| AnalyzeResult       | spec<br>status<br>resultCount<br>resultLocation<br>sourceLocationPrefix<br>databaseSHA   | JobSpec<br>Status<br>int<br>ArtifactLocation<br>string<br>string  |
| ArtifactLocation    | Key<br>Bucket  | string<br>string  |
| AnalyzeJob          | Spec<br>QueryPackLocation<br>QueryLanguage   | JobSpec<br>ArtifactLocation<br>QueryLanguage  |
| QueryLanguage       |  | string  |
| JobInfo             | QueryLanguage<br>CreatedAt<br>UpdatedAt<br>SkippedRepositories   | string<br>string<br>string<br>SkippedRepositories   |
| SkippedRepositories | AccessMismatchRepos<br>NotFoundRepos<br>NoCodeqlDBRepos  | AccessMismatchRepos<br>NotFoundRepos<br>NoCodeqlDBRepos   |

| Type Name           | Field   | Type   |
|---------------------|---|--|
|                     | OverLimitRepos  | OverLimitRepos                                   |
| AccessMismatchRepos | RepositoryCount<br>Repositories                                   | int<br>[Repository]                              |
| NotFoundRepos       | RepositoryCount<br>RepositoryFullNames                            | int<br>[string]                                  |
| Repository          | ID<br>Name<br>FullName<br>Private<br>StargazersCount<br>UpdatedAt | int<br>string<br>string<br>bool<br>int<br>string |

### 3 Symbols and Notation

We define the following symbols for entities in the system:

| Concept             | Symbol                          | Description  |
|---------------------|---------------------------------|--|
| Client              | $C$                             | The source of the query submission                             |
| Server              | $S$                             | Manages job queue and communicates results back to the client  |
| Job Queue           | $Q$                             | Queue for managing submitted jobs                              |
| Agent               | $\alpha$                        | Independently polls, executes jobs, and accumulates results    |
| Agent Set           | $A$                             | The set of all available agents                                |
| Query Suite         | $\mathcal{Q}$                   | Collection of queries submitted by the client                  |
| Repository List     | $\mathcal{R}$                   | Collection of repositories                                     |
| $i$ -th Repository  | $\mathcal{R}_i$                 | Specific repository indexed by $i$                             |
| $j$ -th Query       | $\mathcal{Q}_j$                 | Specific query from the suite indexed by $j$                   |
| Query Result        | $r_{i,j,k_{i,j}}$               | $k_{i,j}$ -th result from query $j$ executed on repository $i$ |
| Query Result Set    | $\mathcal{R}_i^{\mathcal{Q}_j}$ | Set of all results for query $j$ on repository $i$             |
| Accumulated Results | $\mathcal{R}_i^{\mathcal{Q}}$   | All results from executing all queries on $\mathcal{R}_i$      |

### 4 Full Round-Trip Representation

The full round-trip execution, from query submission to result delivery, can be summarized as:

$$C \xrightarrow{\mathcal{Q}} S \xrightarrow{\text{enqueue}} Q \xrightarrow{\text{poll}} \alpha \xrightarrow{\mathcal{Q}(\mathcal{R}_i)} S \xrightarrow{\mathcal{R}_i^{\mathcal{Q}}} C$$

- $C \rightarrow S$ : Client submits a query suite  $\mathcal{Q}$  to the server.
- $S \rightarrow Q$ : Server enqueues the query suite  $(\mathcal{Q}, \mathcal{R}_i)$  for each repository.
- $Q \rightarrow \alpha$ : Agent  $\alpha$  polls the queue and retrieves a job.
- $\alpha \rightarrow S$ : Agent executes the queries and returns the accumulated results  $\mathcal{R}_i^{\mathcal{Q}}$  to the server.
- $S \rightarrow C$ : Server sends the complete result set  $\mathcal{R}_i^{\mathcal{Q}}$  for each repository back to the client.

## 5 Result Representation

For the complete collection of results across all repositories and queries:

$$\mathcal{R}^{\mathcal{Q}} = \bigcup_{i=1}^N \bigcup_{j=1}^M \left\{ r_{i,j,1}, r_{i,j,2}, \dots, r_{i,j,k_{i,j}} \right\}$$

where:

- $N$  is the total number of repositories.
- $M$  is the total number of queries in  $\mathcal{Q}$ .
- $k_{i,j}$  is the number of results from executing query  $\mathcal{Q}_j$  on repository  $\mathcal{R}_i$ .

An individual result from the  $i$ -th repository,  $j$ -th query, and  $k$ -th result is:

$$r_{i,j,k}$$

$$C \xrightarrow{\mathcal{Q}} S \xrightarrow{\text{enqueue}} Q \xrightarrow{\text{dispatch}} \alpha \xrightarrow{\mathcal{Q}(\mathcal{R}_i)} S \xrightarrow{r_{i,j}} C$$

Each result can be further indexed to track multiple repositories and result sets.

## 6 Execution Loop in Pseudo-Code

Listing 1: Distributed Query Execution Algorithm

```
1  # Distributed Query Execution with Agent Polling and Accumulated Results
2
3  # Initialization
4   $\mathcal{R}$  = set() # Repository list
5   $Q$  = [] # Job queue
6   $A$  = set() # Set of agents
7   $\mathcal{R}_i^{\mathcal{Q}}$  = {} # Result storage for each repository
8
9  # Initialize result sets for each repository
10 for  $R_i$  in  $\mathcal{R}$ :
11      $\mathcal{R}_i^{\mathcal{Q}}$  = {} # Initialize empty result set
12
13 # Enqueue the entire query suite for all repositories
14 for  $R_i$  in  $\mathcal{R}$ :
15      $Q.append((\mathcal{Q}, R_i))$  # Enqueue  $(\mathcal{Q}, R_i)$  pair
16
17 # Processing loop while there are jobs in the queue
18 while  $Q \neq \emptyset$ :
19     # Agents autonomously poll the queue
20     for  $\alpha$  in  $A$ :
21         if  $\alpha.is\_available()$ :
22              $(\mathcal{Q}, R_i) = Q.pop(0)$  # Agent polls a job
23
24             # Agent execution begins
25              $\mathcal{R}_i^{\mathcal{Q}}$  = {} # Initialize results for repository  $R_i$ 
26
27             for  $\mathcal{Q}_j$  in  $\mathcal{Q}$ :
28                 # Execute query  $\mathcal{Q}_j$  on repository  $R_i$ 
29                  $r_{i,j,1}, \dots, r_{i,j,k_{i,j}}$  =  $\alpha.execute(\mathcal{Q}_j, R_i)$ 
30
31                 # Store results for query  $j$ 
32                  $\mathcal{R}_i^{\mathcal{Q}_j} = \{r_{i,j,1}, \dots, r_{i,j,k_{i,j}}\}$ 
33
34                 # Accumulate results
35                  $\mathcal{R}_i^{\mathcal{Q}} = \mathcal{R}_i^{\mathcal{Q}} \cup \mathcal{R}_i^{\mathcal{Q}_j}$ 
36
37             # Send all accumulated results back to the server
38              $\alpha.send\_results(S, (\mathcal{Q}, R_i, \mathcal{R}_i^{\mathcal{Q}}))$ 
39
40             # Server sends results for  $(\mathcal{Q}, R_i)$  back to the client
41              $S.send\_results\_to\_client(C, (\mathcal{Q}, R_i, \mathcal{R}_i^{\mathcal{Q}}))$ 
```

## 7 Execution Loop in Pseudo-Code, declarative

Listing 2: Distributed Query Execution Algorithm

```
1  # Distributed Query Execution with Agent Polling and Accumulated Results
2
3  # Define initial state
4   $\mathcal{R}$ : set          # Set of repositories
5   $\mathcal{Q}$ : set          # Set of queries
6  A: set           # Set of agents
7  Q: list          # Queue of ( $\mathcal{Q}, \mathcal{R}_i$ ) pairs
8   $\mathcal{R}_{\text{results}}$ : dict = {} # Mapping of repositories to their accumulated query results
9
10 # Initialize result sets for each repository
11  $\mathcal{R}_{\text{results}} = \{\mathcal{R}_i: \text{set}() \text{ for } \mathcal{R}_i \text{ in } \mathcal{R}\}$ 
12
13 # Define job queue as an immutable mapping
14 Q = [( $\mathcal{Q}, \mathcal{R}_i$ ) for  $\mathcal{R}_i$  in  $\mathcal{R}$ ]
15
16 # Processing as a declarative iteration over the job queue
17 def execute_queries(agents, job_queue, repository_results):
18     def available_agents():
19         return { $\alpha$  for  $\alpha$  in agents if  $\alpha$ .is_available()}
20
21     def process_job( $\mathcal{Q}, \mathcal{R}_i, \alpha$ ):
22         results = { $\mathcal{Q}_j: \alpha$ .execute( $\mathcal{Q}_j, \mathcal{R}_i$ ) for  $\mathcal{Q}_j$  in  $\mathcal{Q}$ }
23         return  $\mathcal{R}_i$ , results
24
25     def accumulate_results( $\mathcal{R}_{\text{results}}, \mathcal{R}_i$ , query_results):
26         return {** $\mathcal{R}_{\text{results}}$ ,  $\mathcal{R}_i$ :  $\mathcal{R}_{\text{results}}[\mathcal{R}_i] \cup \text{set}().\text{union}(*\text{query\_results.values}())$ }
27
28     while job_queue:
29         active_agents = available_agents()
30         for  $\alpha$  in active_agents:
31              $\mathcal{Q}, \mathcal{R}_i$  = job_queue[0] # Peek at the first job
32             _, query_results = process_job( $\mathcal{Q}, \mathcal{R}_i, \alpha$ )
33             repository_results = accumulate_results(repository_results,  $\mathcal{R}_i$ ,
34                                                     query_results)
35
36              $\alpha$ .send_results(S, ( $\mathcal{Q}, \mathcal{R}_i$ , repository_results[ $\mathcal{R}_i$ ]))
37             S.send_results_to_client(C, ( $\mathcal{Q}, \mathcal{R}_i$ , repository_results[ $\mathcal{R}_i$ ]))
38
39             job_queue = job_queue[1:] # Move to the next job
40
41         return repository_results
42
43 # Execute the distributed query process
44  $\mathcal{R}_{\text{results}} = \text{execute\_queries}(A, Q, \mathcal{R}_{\text{results}})$ 
```

## 8 Execution Loop in Pseudo-Code, algorithmic

---

**Algorithm 1** Distribute a set of queries  $\mathcal{Q}$  across repositories  $\mathcal{R}$  using agents  $A$

---

```

1: procedure DISTRIBUTEDQUERYEXECUTION( $\mathcal{Q}, \mathcal{R}, A$ )
2:   for all  $\mathcal{R}_i \in \mathcal{R}$  do                                     ▷ Initialize result sets for each repository and query
3:      $\mathcal{R}_i^{\mathcal{Q}} \leftarrow \{\}$ 
4:   end for
5:    $Q \leftarrow \{\}$                                              ▷ Initialize empty job queue
6:   for all  $\mathcal{R}_i \in \mathcal{R}$  do                                     ▷ Enqueue the entire query suite across all repositories
7:      $S \xrightarrow{\text{enqueue}(\mathcal{Q}, \mathcal{R}_i)} Q$ 
8:   end for
9:   while  $Q \neq \emptyset$  do                                   ▷ Agents poll the queue for available jobs
10:    for all  $\alpha \in A$  where  $\alpha$  is available do
11:       $\alpha \xleftarrow{\text{poll}(Q)}$                                      ▷ Agent autonomously retrieves a job
12:      _____ ▷ Agent Execution Begins
13:       $\mathcal{R}_i^{\mathcal{Q}} \leftarrow \{\}$                                      ▷ Initialize result set for this repository
14:      for all  $\mathcal{Q}_j \in \mathcal{Q}$  do
15:         $\mathcal{R}_i^{\mathcal{Q}_j} \leftarrow \{r_{i,j,1}, r_{i,j,2}, \dots, r_{i,j,k_{i,j}}\}$  ▷ Collect results for query  $j$  on repository  $i$ 
16:         $\mathcal{R}_i^{\mathcal{Q}} \leftarrow \mathcal{R}_i^{\mathcal{Q}} \cup \mathcal{R}_i^{\mathcal{Q}_j}$                      ▷ Accumulate results
17:      end for
18:       $\alpha \xrightarrow{(\mathcal{Q}, \mathcal{R}_i, \mathcal{R}_i^{\mathcal{Q}})} S$                              ▷ Agent sends all accumulated results back to server
19:      _____ ▷ Agent Execution Ends
20:       $S \xrightarrow{(\mathcal{Q}, \mathcal{R}_i, \mathcal{R}_i^{\mathcal{Q}})} C$                              ▷ Server sends results for repository  $i$  back to the client
21:    end for
22:  end while
23: end procedure

```

---

## 9 Execution Loop in Pseudo-Code, hybrid

**Algorithm:** Distribute a set of queries  $\mathcal{Q}$  across repositories  $\mathcal{R}$  using agents  $A$

### 1. Initialization

- For each repository  $\mathcal{R}_i \in \mathcal{R}$ :
  - Initialize result sets:  $\mathcal{R}_i^{\mathcal{Q}} \leftarrow \{\}$ .
- Initialize an empty job queue:  $Q \leftarrow \{\}$ .

### 2. Enqueue Queries

- For each repository  $\mathcal{R}_i \in \mathcal{R}$ :
  - Enqueue the entire query suite:  $S \xrightarrow{\text{enqueue}(\mathcal{Q}, \mathcal{R}_i)} Q$ .

### 3. Execution Loop

- While  $Q \neq \emptyset$ : (agents poll the queue for available jobs)
  - For each available agent  $\alpha \in A$ :
    - \* Agent autonomously retrieves a job:  $\alpha \xleftarrow{\text{poll}(Q)}$ .

**\* Agent Execution Block**

Initialize result set for this repository:  $\mathcal{R}_i^{\mathcal{Q}} \leftarrow \{\}$ .

For each query  $\mathcal{Q}_j \in \mathcal{Q}$ :

Collect results:  $\mathcal{R}_i^{\mathcal{Q}_j} \leftarrow \{r_{i,j,1}, r_{i,j,2}, \dots, r_{i,j,k_{i,j}}\}$ .

Accumulate results:  $\mathcal{R}_i^{\mathcal{Q}} \leftarrow \mathcal{R}_i^{\mathcal{Q}} \cup \mathcal{R}_i^{\mathcal{Q}_j}$ .

Agent sends all accumulated results back to the server:  $\alpha \xrightarrow{(\mathcal{Q}, \mathcal{R}_i, \mathcal{R}_i^{\mathcal{Q}})} S$ .

**4. Agent Sends Results**

- Server sends results for repository  $i$  back to the client:  $S \xrightarrow{(\mathcal{Q}, \mathcal{R}_i, \mathcal{R}_i^{\mathcal{Q}})} C$ .