MRVA for CodeQL

Michael Hohn

Technical Report 20250224

Contents

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 $\mathscr{Q}$ targeting a repository set $\mathscr{R}$.
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:

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

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

A more rigorous description of this is in section **??**.

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.

Type Name	Field	Type

2.3 Messages and their Types

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

Type Name	Field	Type
ServerState	NextID	() → int
	GetResult	JobSpec → IO (Either Error AnalyzeResult)
	GetJobSpecByRepoId	(int, int) → IO (Either Error JobSpec)
	SetResult	(JobSpec, AnalyzeResult) → IO ()
	GetJobList	int → IO (Either Error **[AnalyzeJob]**)
	GetJobInfo	JobSpec → IO (Either Error JobInfo)
	SetJobInfo	(JobSpec, JobInfo) → IO ()
	GetStatus	JobSpec → IO (Either Error Status)
	SetStatus	(JobSpec, Status) → IO ()
	AddJob	AnalyzeJob → IO ()
JobSpec	sessionID	int
	nameWithOwner	string
AnalyzeResult	spec	JobSpec
	status	Status
	resultCount	int
	resultLocation	ArtifactLocation
	sourceLocationPrefix	string
	databaseSHA	string
ArtifactLocation	Key	string
	Bucket	string
AnalyzeJob	Spec	JobSpec
	QueryPackLocation	ArtifactLocation
	QueryLanguage	QueryLanguage
QueryLanguage		string
JobInfo	QueryLanguage	string
	CreatedAt	string
	UpdatedAt	string
	SkippedRepositories	SkippedRepositories
SkippedRepositories	AccessMismatchRepos	AccessMismatchRepos
	NotFoundRepos	NotFoundRepos
	NoCodeqlDBRepos	NoCodeqlDBRepos
	OverLimitRepos	OverLimitRepos
AccessMismatchRepos	RepositoryCount	int
	Repositories	**[Repository]**
NotFoundRepos	RepositoryCount	int
	RepositoryFullNames	**[string]**
Repository	ID	int

Type Name	Field	Type
	Name	string
	FullName	string
	Private	bool
	StargazersCount	int
	UpdatedAt	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	α	Independently polls, executes jobs, and accumulates results
Agent Set	A	The set of all available agents
Query Suite	$\mathscr{Q}$	Collection of queries submitted by the client
Repository List	$\mathscr{R}$	Collection of repositories
i-th Repository	$\mathscr{R}_i$	Specific repository indexed by i
j-th Query	$\mathscr{Q}_j$	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	$\mathscr{R}_i^{\mathscr{Q}_j}$	Set of all results for query j on repository i
Accumulated Results	$\mathscr{R}_i^{\mathscr{Q}}$	All results from executing all queries on $\mathscr{R}_i$

4 Full Round-Trip Representation

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

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

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

5 Result Representation

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

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

where:

- N is the total number of repositories.

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

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

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

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

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

```
# Distributed Query Execution with Agent Polling and Accumulated Results

# Initialization
$\mathscr{R}$ = set() # Repository list
$Q$ = [] # Job queue
$A$ = set() # Set of agents
$\mathscr{R}_i^{\mathscr{Q}}$ = {} # Result storage for each repository

# Initialize result sets for each repository
for $R_i$ in $\mathscr{R}$:
    $\mathscr{R}_i^{\mathscr{Q}} = \{\}$ # Initialize empty result set

# Enqueue the entire query suite for all repositories
for $R_i$ in $\mathscr{R}$:
    $Q$.append(($\mathscr{Q}$, $R_i$)) # Enqueue $(\mathscr{Q}, \mathscr{R}_i)$ pair

# Processing loop while there are jobs in the queue
while $Q \neq \emptyset$:
    # Agents autonomously poll the queue
    for $\alpha$ in $A$:
        if $\alpha$.is_available():
            $(\mathscr{Q}, \mathscr{R}_i)$ = $Q$.pop(0) # Agent polls a job

            # Agent execution begins
            $\mathscr{R}_i^{\mathscr{Q}} = \{\}$ # Initialize results for repository $R_i$

            for $\mathscr{Q}_j$ in $\mathscr{Q}$:
                # Execute query $\mathscr{Q}_j$ on repository $\mathscr{R}_i$
                $r_{i,j,1}, \ldots, r_{i,j,k_{i,j}}$ = $\alpha$.execute($\mathscr{Q}_j$, $R_i$)

                # Store results for query $j$
                $\mathscr{R}_i^{\mathscr{Q}_j} = \{r_{i,j,1}, \ldots, r_{i,j,k_{i,j}}\}$

                # Accumulate results
                $\mathscr{R}_i^{\mathscr{Q}} = \mathscr{R}_i^{\mathscr{Q}} \cup \mathscr{R}_i^{\mathscr{Q}_j}$

            # Send all accumulated results back to the server
            $\alpha$.send_results($S$, ($\mathscr{Q}$, $R_i$, $\mathscr{R}_i^{\mathscr{Q}}$))

            # Server sends results for $(\mathscr{Q}, \mathscr{R}_i)$ back to the client
            $S$.send_results_to_client($C$, ($\mathscr{Q}$, $R_i$, $\mathscr{R}_i^{\mathscr{Q}}$))
```

7 Execution Loop in Pseudo-Code, declarative

Listing 2: Distributed Query Execution Algorithm

```
# Distributed Query Execution with Agent Polling and Accumulated Results

# Define initial state
ℛ: set          # Set of repositories
𝒬: set          # Set of queries
A: set           # Set of agents
Q: list          # Queue of (𝒬,ℛ_i) pairs
ℛ_results: dict = {} # Mapping of repositories to their accumulated query results

# Initialize result sets for each repository
ℛ_results = {ℛ_i: set() for ℛ_i in ℛ}

# Define job queue as an immutable mapping
Q = [(𝒬, ℛ_i) for ℛ_i in ℛ]

# Processing as a declarative iteration over the job queue
def execute_queries(agents, job_queue, repository_results):
    def available_agents():
        return {α for α in agents if α.is_available()}

    def process_job(𝒬, ℛ_i, α):
        results = {𝒬_j: α.execute(𝒬_j, ℛ_i) for 𝒬_j in 𝒬}
        return ℛ_i, results

    def accumulate_results(ℛ_results, ℛ_i, query_results):
        return {**ℛ_results, ℛ_i: ℛ_results[ℛ_i] | set().union(*query_results.values())}

    while job_queue:
        active_agents = available_agents()
        for α in active_agents:
            𝒬, ℛ_i = job_queue[0] # Peek at the first job
            _, query_results = process_job(𝒬, ℛ_i, α)
            repository_results = accumulate_results(repository_results, ℛ_i,
               query_results)

            α.send_results(S, (𝒬, ℛ_i, repository_results[ℛ_i]))
            S.send_results_to_client(C, (𝒬, ℛ_i, repository_results[ℛ_i]))

        job_queue = job_queue[1:]  # Move to the next job

    return repository_results

# Execute the distributed query process
ℛ_results = execute_queries(A, Q, ℛ_results)
```

8 Execution Loop in Pseudo-Code, algorithmic

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

procedure DistributedQueryExecution($\mathscr{Q}, \mathscr{R}, A$)
  for all $\mathscr{R}_i \in \mathscr{R}$ **do** ▷ Initialize result sets for each repository and query
    $\mathscr{R}_i^{\mathscr{Q}} \leftarrow \{\}$
  end for
  $Q \leftarrow \{\}$ ▷ Initialize empty job queue
  for all $\mathscr{R}_i \in \mathscr{R}$ **do** ▷ Enqueue the entire query suite across all repositories
    $S \xrightarrow{\text{enqueue}(\mathscr{Q},\mathscr{R}_i)} Q$
  end for
  while $Q \neq \emptyset$ **do** ▷ Agents poll the queue for available jobs
    for all $\alpha \in A$ **where** α is available **do**
      $\alpha \xleftarrow{\text{poll}(Q)}$ ▷ Agent autonomously retrieves a job
      ——————————————— ▷ Agent Execution Begins
      $\mathscr{R}_i^{\mathscr{Q}} \leftarrow \{\}$ ▷ Initialize result set for this repository
      for all $\mathscr{Q}_j \in \mathscr{Q}$ **do**
        $\mathscr{R}_i^{\mathscr{Q}_j} \leftarrow \left\{ r_{i,j,1}, r_{i,j,2}, \ldots, r_{i,j,k_{i,j}} \right\}$ ▷ Collect results for query j on repository i
        $\mathscr{R}_i^{\mathscr{Q}} \leftarrow \mathscr{R}_i^{\mathscr{Q}} \cup \mathscr{R}_i^{\mathscr{Q}_j}$ ▷ Accumulate results
      end for
      $\alpha \xrightarrow{(\mathscr{Q},\mathscr{R}_i,\mathscr{R}_i^{\mathscr{Q}})} S$ ▷ Agent sends all accumulated results back to server
      ——————————————— ▷ Agent Execution Ends
      $S \xrightarrow{(\mathscr{Q},\mathscr{R}_i,\mathscr{R}_i^{\mathscr{Q}})} C$ ▷ Server sends results for repository i back to the client
    end for
  end while
end procedure

9 Execution Loop in Pseudo-Code, hybrid

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

1. **Initialization**
   - For each repository $\mathscr{R}_i \in \mathscr{R}$:
     - Initialize result sets: $\mathscr{R}_i^{\mathscr{Q}} \leftarrow \{\}$.
   - Initialize an empty job queue: $Q \leftarrow \{\}$.
2. **Enqueue Queries**
   - For each repository $\mathscr{R}_i \in \mathscr{R}$:
     - Enqueue the entire query suite: $S \xrightarrow{\text{enqueue}(\mathscr{Q},\mathscr{R}_i)} Q$.
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: $\mathscr{R}_i^{\mathscr{Q}} \leftarrow \{\}$.
  - For each query $\mathscr{Q}_j \in \mathscr{Q}$:
  - Collect results: $\mathscr{R}_i^{\mathscr{Q}_j} \leftarrow \{r_{i,j,1}, r_{i,j,2}, \ldots, r_{i,j,k_{i,j}}\}$.
  - Accumulate results: $\mathscr{R}_i^{\mathscr{Q}} \leftarrow \mathscr{R}_i^{\mathscr{Q}} \cup \mathscr{R}_i^{\mathscr{Q}_j}$.
  - Agent sends all accumulated results back to the server: $\alpha \xrightarrow{(\mathscr{Q}, \mathscr{R}_i, \mathscr{R}_i^{\mathscr{Q}})} S$.

4. **Agent Sends Results**

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