System Design Documentation: Remote Code Execution (RCE) Platform

Date: January 23, 2026

Architecture Style: Event-Driven Microservices

1. Introduction

This document describes the high-level and detailed architecture of the Remote Code Execution Platform. The system enables users to submit code through a web interface, execute it securely in isolated containers, and receive results in real-time. It is designed to handle high concurrency, ensuring that resource-intensive code execution does not block the responsiveness of the user interface.

2. System Overview

The system follows a **microservices-based architecture** consisting of independent services responsible for authentication, job submission, execution, event notification, and real-time client updates.

High-Level Design (HLD) Breakdown

The architecture is segmented into three distinct layers:

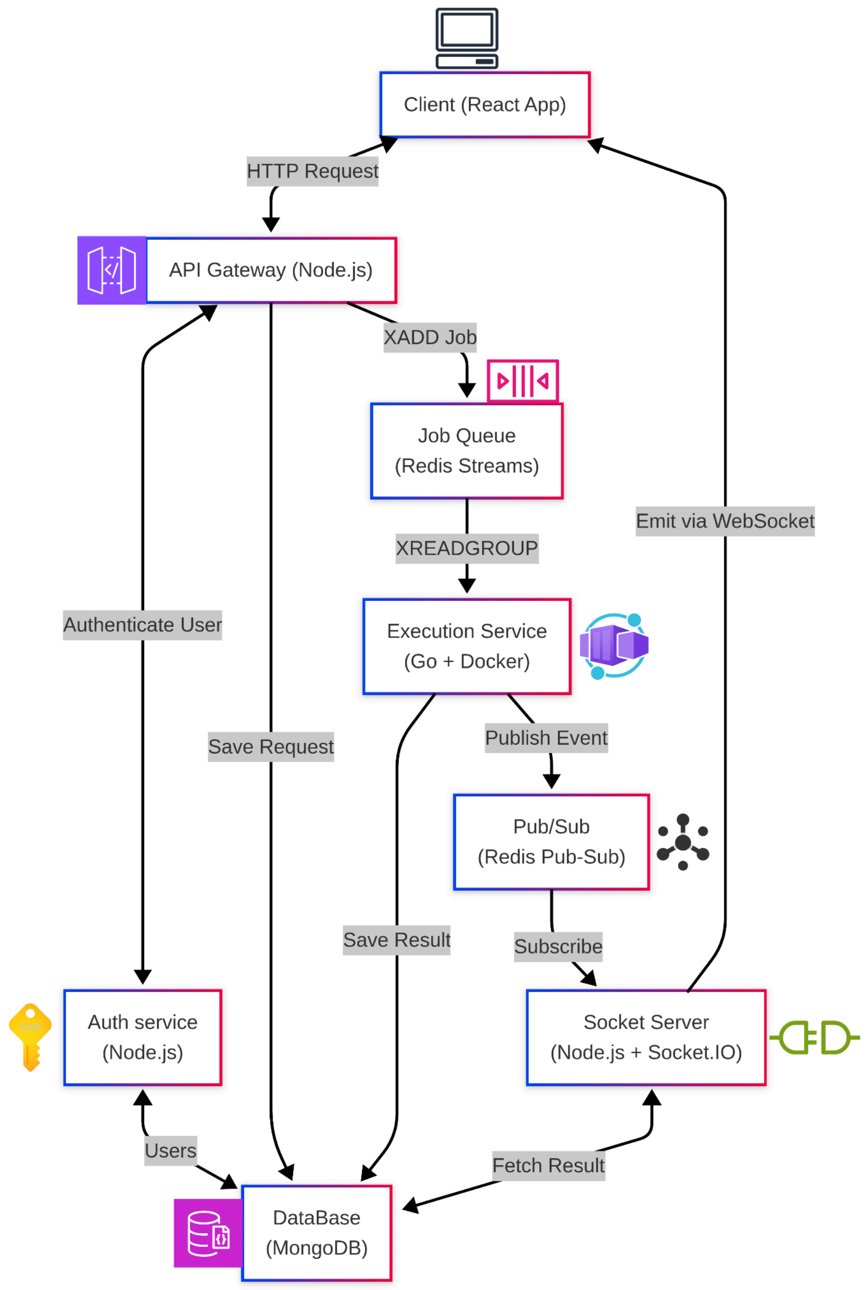
1. **Ingestion Layer:** Handles client requests, authentication, and job dispatching (Client, API Gateway, Auth Service).
2. **Processing Layer:** Manages the queueing and actual execution of code (Redis Job Queue, Execution Service).
3. **Notification Layer:** Handles the real-time broadcast of results back to the user (Redis Pub/Sub, Socket Server).

3. Architecture Components

The platform is composed of the following main services:

1. Client (React Application)
2. API Gateway (Node.js)
3. Authentication Service (Node.js)
4. Job Queue (Redis Streams)

5. Execution Service (Go + Docker)
6. Pub/Sub (Redis)
7. Socket Server (Node.js + Socket.IO)
8. Database (MongoDB)



4. Service-Level Detailed Design

4.1 Client (React Application)

- **Role:** User Interface for writing, submitting, and monitoring code execution.
- **Functionality:**
  - Provides a rich code editor (likely Monaco Editor).
  - Submits code via HTTP POST to the API Gateway.
  - Establishes a WebSocket connection to the Socket Server to receive real-time updates.

4.2 API Gateway (Node.js)

- **Role:** The central entry point for all client requests.
- **Key Responsibilities:**
  - **Traffic Management:** Validates requests and forwards authentication tasks.
  - **Job Producer:** Instead of executing code, it serializes the request and publishes it to **Redis Streams** via the XADD command.
  - **Statelessness:** It remains lightweight and highly responsive, offloading all heavy lifting to the queue.

4.3 Authentication Service

- **Role:** Handles user registration, login, token generation (JWT), and validation.
- **Persistence:** Communicates with MongoDB to store user credentials and profile data.

4.4 Job Queue (Redis Streams)

- **Role:** A reliable, persistent job queue for distributing code execution tasks.
- **Why Redis Streams:**
  - Supports **Consumer Groups**, allowing multiple worker instances to read from the same queue without duplicating work.
  - Ensures **Fault Tolerance** via message acknowledgment (XACK). If a worker crashes, the job remains in the "Pending Entry List" (PEL) and is reassigned to a healthy worker.

4.5 Execution Service (The Worker Engine)

- **Technology:** Go (Golang) + Docker
- **Role:** Consumes jobs from Redis Streams and executes user code in isolated containers.
- **Why Go (Golang)?**
  - **Concurrency Model:** We utilize Go's **Goroutines** to handle high concurrency with minimal memory overhead. Unlike Node.js (single-threaded) or Python (heavy threads), a single Go worker can efficiently manage multiple blocking Docker operations simultaneously.
  - **Performance:** Go offers near-native execution speed, minimizing the latency overhead of the orchestration layer.

    - **System Control:** Go's context package allows for precise timeout management, ensuring user code cannot hang the system indefinitely.
- **Worker Configuration:**
    - **Internal Concurrency:** Each worker instance implements a **Worker Pool** (e.g., 5-10 internal Goroutines), allowing it to process multiple jobs in parallel.
    - **Scaling:** The system starts with a baseline of **3 Worker Replicas**, providing a total capacity of ~15-30 simultaneous executions.
- **Execution Workflow:**
    1. **Fetch:** Listens to Redis Stream via XREADGROUP.
    2. **Isolate:** Spawns a **fresh Docker container** for every job.
    3. **Execute:** Runs the code, capturing stdout and stderr.
    4. **Teardown:** Destroys the container immediately to ensure a clean slate.
    5. **Publish:** Pushes the result to Redis Pub/Sub.

4.6 Database (MongoDB)

- **Role:** The primary source of truth.
- **Data Storage:** Stores user data, execution request logs, and final execution results.
- **Choice Rationale:** MongoDB's flexible schema is ideal for storing varied output formats and metadata associated with different programming languages.

4.7 Real-Time Notification Layer

- **Redis Pub/Sub:** Broadcasts execution completion events from the Worker to the Socket Server.
- **Socket Server (Node.js + Socket.IO):**
    - Subscribes to Redis Pub/Sub channels.
    - Maintains active WebSocket connections with clients.
    - Emits the final result to the specific user who submitted the job.

5. Communication Flow

1. **Submission:** Client submits code to API Gateway via HTTP.
2. **Auth:** API Gateway authenticates the user.
3. **Enqueue:** Job is published to Redis Streams (XADD).
4. **Processing:** Execution Service (Go) consumes the job, creates a Docker container, and executes it.
5. **Persistence:** Result is stored in MongoDB.
6. **Broadcast:** Event is published to Redis Pub/Sub.
7. **Delivery:** Socket server receives the event and sends the update to the client via WebSocket.

6. Scalability Strategy

- **Horizontal Scaling:**
    - **Frontend/API:** Stateless Node.js services can be scaled behind a load balancer.

    - **Workers:** The Consumer Group pattern allows us to add infinite Go Worker instances. New workers automatically pick up the load from the Redis Stream without configuration changes.
- **Autoscaling:**
    - Based on **Queue Lag**. If the number of pending messages in Redis > 100, the infrastructure (e.g., Kubernetes) spins up additional Go Worker pods.
- **Database Scaling:** MongoDB clusters and Redis clusters support backend scaling.

7. Security & Isolation

- **Container Isolation:** Every job runs in a dedicated Docker container with no network access (--network none) to prevent external attacks.
- **Resource Limits:** strict Docker flags are enforced to prevent Denial of Service (DoS):
    - --memory: Limits RAM usage (e.g., 256MB).
    - --cpus: Limits CPU usage (e.g., 0.5 cores).
    - --pids-limit: Prevents fork bombs.
- **Timeouts:** The Go worker enforces a hard timeout (e.g., 5 seconds) using Context cancellation to kill infinite loops.
- **Input Sanitization:** The API Gateway validates payloads before they even reach the processing queue.

8. Reliability and Fault Tolerance

- **Job Durability:** Redis Streams persists jobs to disk.
- **At-Least-Once Delivery:** If a worker crashes, the job remains in the "Pending" state and is reclaimed by another worker.
- **Stateless Services:** API and Socket servers are stateless, enabling fast recovery and easy replacement.

9. Technology Choices Summary

- **Node.js:** Chosen for I/O-heavy services (API Gateway, Socket Server).
- **Go (Golang):** Chosen for the high-performance Execution Service due to its concurrency model and speed.
- **Docker:** Provides the critical sandboxing and isolation layer.
- **Redis:** Enables fast messaging (Pub/Sub) and reliable queuing (Streams).
- **MongoDB:** Ensures schema flexibility for diverse data storage needs.

10. Future Enhancements

- **Orchestration:** Migration to **Kubernetes** for better pod lifecycle management.
- **Advanced Sandboxing:** Implementation of **gVisor** or **Firecracker** for kernel-level isolation.
- **Observability:** Integration of **Prometheus** and **Grafana** for centralized logging and metrics (e.g., average execution time, error rates).

- **Distributed Tracing:** To track a request's lifecycle across all microservices.

11. Conclusion

The platform provides a scalable, secure, and reliable environment for remote code execution, suitable for educational and competitive programming platforms. By leveraging Go for compute-heavy tasks and Node.js for I/O operations, the system achieves an optimal balance of performance and maintainability.