Live Code Execution Backend

A Scalable Architecture for Remote Code Execution

Nguyen Bao Duy

Edtronaut - SWE Intern Case Study

January 27, 2026

Outline

Problem Statement

The Challenge

Build a backend service that allows users to write and execute code in multiple programming languages remotely.

Key Requirements:

- ✓ Support multiple languages (JavaScript, Python, Java)
- ✓ Handle concurrent users safely
- ✓ Protect against malicious code (timeouts, resource limits)
- ✓ Provide session management with autosave
- ✓ Return execution results asynchronously

Core Technical Challenge

How do we execute untrusted user code safely, reliably, and at scale?

Project Overview

What Was Built:

- RESTful API for code sessions
- Queue-based async execution
- Multi-language support
- Rate limiting & spam protection
- Session state management

Technology Stack:

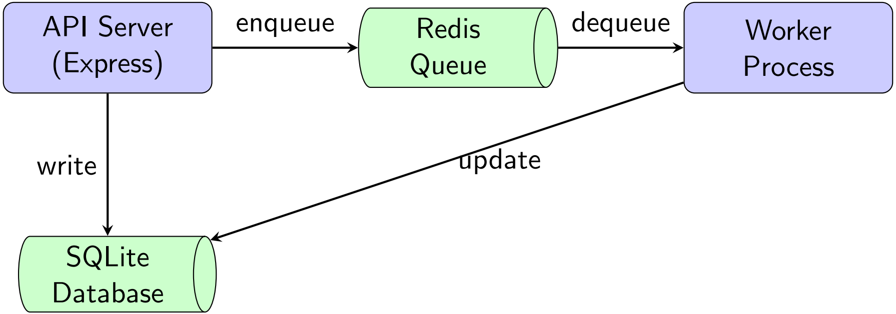
- **Runtime:** Node.js + TypeScript
- **Queue:** Redis + BullMQ
- **Database:** SQLite
- **Architecture:** API + Worker pattern

Design Philosophy

Prioritize **simplicity** and **reliability** over premature optimization.

Outline

High-Level Architecture



Three-Component Design

- **API Server**: Handles HTTP requests, manages sessions, enqueues jobs
- **Redis Queue**: Decouples API from execution, enables async processing
- **Worker Process**: Executes code in isolated environment, updates results

Outline

Design Decision: Async Execution with Queue

Problem

Running user code takes time (1-5 seconds). If API waits, it blocks other requests and degrades performance.

Solution: Queue-Based Architecture

- API adds job to Redis queue and returns **immediately**
- Separate Worker process dequeues and executes code
- Results written to database for later retrieval

Benefits:

- ✔ API stays fast and responsive
- ✔ Independent scaling of API and Workers
- ✔ Automatic retries on failure

Trade-offs:

- ✘ Increased system complexity
- ✘ Results not instant (polling required)
- ✘ Redis dependency

State Management: Execution Lifecycle

State Ownership

Clear separation: **API creates**, **Worker executes**

State	**Set by**	**Responsibility**
`QUEUED`	API	Creates the execution record
`RUNNING`	Worker	Picks up the job and starts it
`COMPLETED` / `FAILED` / `TIMEOUT`	Worker	Writes final result to database

Database as Source of Truth

SQLite stores authoritative state.

Code Isolation: Sandboxing Strategy

Problem

User code is **untrusted**. Must prevent infinite loops, excessive output, and file/network access.

Solution: Child Process Isolation

- Run code in separate child process (not main server)
- **Time limit:** Process killed if execution exceeds 5 seconds
- **Memory limit:** Language-specific flags enforce 128 MB cap
- **Output limit:** Real-time monitoring kills process at 1 MB

Trade-off

Production: Use Docker or gVisor for stronger security.

Autosave Spam Protection

Problem

Users typing rapidly generate **100+ requests/minute**, blocking SQLite.

Solution: Adaptive Throttling

Three-layer strategy:

1. **Throttling:** Max 1 write/second per session
2. **Coalescing:** Only save latest code
3. **Forced write:** If pending $>$ 5s, force write

Guarantees:

- ✔ Latest code saved
- ✔ No delay $>$ 5s

Trade-off:

- ✘ In-memory state
- ✘ Lost on crash

Execution Environment: Resource Limits

Protection Mechanisms

Multiple layers of resource enforcement:

Resource	Limit	Enforced At	Action
Execution time	5 seconds	Worker	Kill Process
Output size	1 MB	Worker	Kill Process
Memory usage	128 MB	OS/Runtime	Process Crash
Concurrent jobs	5 jobs	Worker	Wait in Queue
Executions/min	5/session	API (Redis)	Reject (429)
Cooldown	2 seconds	API (Redis)	Reject (429)

Defense in Depth

Multiple limits ensure **no single user** can overwhelm the system.

Outline

Trade-off Analysis Framework

Decision-Making Approach

Every architectural choice involves trade-offs.

Framework: Problem $\rightarrow$ Decision $\rightarrow$ Benefits vs. Costs

Optimization Priorities

1. **Simplicity:** Single database, minimal orchestration
2. **Reliability:** Automatic retries, graceful failure handling
3. **Speed:** Async queue, non-blocking API
4. **Safety:** Timeouts, output limits, rate limiting

Not Optimized For

- High availability (single SQLite = single point of failure)
- Horizontal API scaling (SQLite doesn't support concurrent writes well)

Trade-off #1: SQLite vs PostgreSQL

SQLite (Chosen)

Why:

- ✓ Zero setup, file-based
- ✓ Perfect for development
- ✓ Simple deployment

Cost:

- ✗ Single writer only
- ✗ No concurrent API instances
- ✗ Not production-ready

PostgreSQL (Alternative)

Benefits:

- ✓ Concurrent writes
- ✓ Read replicas
- ✓ Connection pooling

Cost:

- ✗ Complex setup
- ✗ Requires hosting
- ✗ Migration overhead

Trade-off #2: Child Process vs Docker

Child Process (Chosen)

Why:

- ✔ Simple, no dependencies
- ✔ Fast startup (<50ms)
- ✔ Easy local development

Cost:

- ✘ Limited isolation
- ✘ OS-level limits only
- ✘ Potential security risks

Docker (Alternative)

Benefits:

- ✔ Full sandboxing
- ✔ Network isolation
- ✔ No host access

Cost:

- ✘ Slower startup (∼500ms)
- ✘ Docker daemon required
- ✘ More complex error handling

Trade-off #3: Polling vs WebSocket/SSE

Polling (Chosen)

Why:

- ✔ Simple client code
- ✔ No connection state
- ✔ Stateless API design
- ✔ Works with load balancers

Cost:

- ✘ Higher latency (poll interval)
- ✘ Unnecessary requests
- ✘ More server load

WebSocket/SSE (Alternative)

Benefits:

- ✔ Real-time updates
- ✔ Low latency
- ✔ Better UX

Cost:

- ✘ Connection management
- ✘ Reconnection logic
- ✘ Scaling complexity
- ✘ Stateful API required

Outline

Idempotency Handling: Four-Layer Protection

Problem

Users may click "Run" multiple times quickly.
System must prevent duplicate executions without rejecting legitimate requests.

Solution: Defense in Depth

Layer 1: Active Execution Check

Before creating job, check if session has `QUEUED` or `RUNNING` execution.

Layer 2: Cooldown Period

2-second cooldown after each execution. Prevents rapid-fire requests.

Layer 3: Rate Limit

Maximum 5 executions per minute per session. Protects against abuse.

Layer 4: Database Constraint

SQLite unique constraint on execution ID as last-resort safety net.

Failure Handling: Stalled Job Detection

Problem

Worker may crash mid-execution. How does system recover?

BullMQ Automatic Recovery

1. Worker marks job as `RUNNING` and acquires lock (30s duration)
2. If worker crashes, lock expires
3. BullMQ detects stalled job via `stalledInterval` check (30s)
4. Job automatically retried (up to 3 attempts)
5. After exhausting retries → `failed` event → DB updated to `FAILED`

Guarantees:

- ✔ No jobs stuck forever
- ✔ Automatic retry on crash

Out of Scope:

- ✘ True exactly-once execution
- ✘ Dead letter queue (DLQ)

Horizontal Scaling Strategy

Stateless API Design

No per-request or per-user state in API memory.
All persistent state in shared storage: Database + Redis (queues, rate limits).

Scaling API Servers:

- Any API instance handles any request
- Load balancer distributes traffic
- **Bottleneck:** Database write contention
- **Solution:** PostgreSQL with row-level locking & connection pooling

Scaling Workers:

- Workers pull jobs from Redis queue
- Each worker: 5 concurrent jobs
- Independent processes (deploy, scale, restart separately)
- **Throughput:** Linear scaling (until CPU/Redis bottleneck)

Scalability Bottlenecks & Mitigations

Bottleneck	Current Mitigation	Production Solution
SQLite write conflicts	Autosave throttling, single DB	PostgreSQL with connection pooling
Redis memory growth	Auto-cleanup (1h, 1000 jobs)	Max queue size limit, priority queues
Worker CPU overload	Concurrency limit (5 jobs)	Kubernetes auto-scaling, resource quotas
Long execution times	5-second global timeout	Per-language timeouts, adaptive limits
Excessive output	1MB limit, kill process	Stream output, early truncation

Outline

What Was Achieved

Successfully Implemented

- ✔ **Async queue-based execution** with Redis + BullMQ
- ✔ **Multi-language support** (JavaScript, Python, Java)
- ✔ **Session management** with autosave and throttling
- ✔ **Code isolation** with timeout and output limits
- ✔ **Idempotency protection** with 4-layer defense
- ✔ **Automatic failure recovery** via BullMQ retry logic
- ✔ **Scalable worker architecture** with horizontal scaling

Production Readiness Gaps

What Would Be Improved with More Time

1. Stronger Container Isolation

Replace child processes with Docker or gVisor for full sandboxing

2. PostgreSQL Migration

Enable concurrent writes, read replicas, and horizontal API scaling

3. Observability

Structured logging, Prometheus metrics, BullMQ dashboard, alerting

4. Real-time Updates

WebSocket/SSE for pushing execution status instead of polling

5. Advanced Queue Management

Max queue size, priority queues, per-language isolation, backpressure

Live Code Execution Backend

A Case Study in Scalable System Design

GitHub: `github.com/BuhDuy256/live-code-execution-backend`

Tech Stack: Node.js, TypeScript, Redis, BullMQ, SQLite

Questions?