🐳 AI Code Execution Architecture Using Docker (Loveable-Style Platform)

🙈 Overview

This document explains how AI-generated code is executed securely using Docker containers in a sandboxed environment. The architecture is designed for platforms similar to AI app builders where users provide prompts and receive working applications or script execution results.

The system consists of three major layers:

1. AI Code Generation Layer
2. Container Execution Layer (DockerManager)
3. Preview / Hosting Layer

---

1. High-Level Execution Flow

User Prompt ↓ AI generates code ↓ Backend receives generated code ↓ DockerManager.createContainer(userId) ↓ DockerManager.executeCode(userId, code) ↓ Code runs inside isolated Docker container ↓ Output streamed back to frontend

🌀 Important Principle

Your DockerManager executes code.

The AI does NOT execute code.

The AI only:

- Generates code
- Modifies code
- Explains code
- Fixes errors

Execution is always done by your sandbox.

🌗 Architecture for AI-Powered Code Platform

If you're building something like Loveable:

```
User
  ↓
Frontend (Editor UI)
```

```
  ↓
Backend API
  ↓
LLM Service (Generate Code)
  ↓
DockerManager.executeCode()
  ↓
Container
  ↓

Return Output
```

Step-by-Step Flow (Real Scenario)

🐓 User says:

> "Create a simple express server"

🐕 Backend calls LLM

```
const response = await openai.chat.completions.create({
  model: "gpt-4",
  messages: [{ role: "user", content: prompt }]
});

const generatedCode = response.choices[0].message.content;
```

Now you have code as string.

🐗 Send Code to DockerManager

```
const result = await DockerManager.executeCode(userId, generatedCode);
```

This runs inside:

- 512MB container
- No network
- CPU limited
- Timeout 10 seconds

🐚 Return Output

```
{
  "output": "...",
```

```
  "error": ""
}
```

🔥 So What Actually Happens Internally?

In your system:

AI Layer

- Just text generator
- Lives separately
- Could be OpenAI API or your own model

Execution Layer

- Your DockerManager
- Runs untrusted code

Orchestrator Layer

- Connects AI → Execution

🌒 2. Core Components in DockerManager

🌧️ Imported Modules

- dockerode → Communicates with Docker daemon
- uuid → Generates unique container names
- pgPool → Tracks container sessions in PostgreSQL
- tar-fs → Builds Docker image from folder
- fs & path → File system utilities

🐳 3. Image Management

Base Image

node:18-alpine

- Lightweight
- Secure minimal OS
- Fast startup

Custom Sandbox Image

IMAGE_NAME = "custom-node-sandbox\:latest"

The system:

1. Pulls the base image
2. Builds a sandbox image using a local Dockerfile
3. Tags it as custom-node-sandbox\:latest

This ensures:

- Controlled runtime
- Pre-installed dependencies (if needed)
- Isolated environment

🦏 4. Ensuring Image Exists

Before creating containers:

- List existing images
- Check if sandbox image exists
- If not:
- Pull base image
- Build sandbox image

This prevents rebuilding on every request.

🌳 5. Creating a Container

Each user gets a dedicated container.

Container configuration:

- Memory: 512MB
- MemorySwap: 512MB
- CpuQuota: Limited
- NetworkMode: none (no internet access)
- PidsLimit: 64
- WorkingDir: /workspace

Container runs:

"tail -f /dev/null"

This keeps container alive as a persistent sandbox.

The container ID is:

- Stored in memory (activeContainers map)
- Persisted in PostgreSQL (user_sessions table)

▶ 6. Executing AI-Generated Code

Step 1: Write Code Safely

Code is converted to Base64:

- Prevents shell injection
- Prevents command breakouts

Then written inside container:

/workspace/index.js

Step 2: Execute Code

Command executed inside container:

node /workspace/index.js

Execution results:

- stdout captured
- stderr captured
- Timeout enforced (e.g., 10 seconds)

Returned as:

{ output: string, error: string }

---

7. Timeout Protection

runExec() implements:

- Stream monitoring
- Manual timeout
- Safe stream destruction

Prevents:

- Infinite loops
- Blocking execution
- Resource exhaustion

---

8. Stopping and Cleanup

When user session ends:

- Stop container
- Remove container
- Delete DB session
- Remove from activeContainers map

Also supports:

- cleanupAll() for graceful shutdown
- SIGINT and SIGTERM handlers

🌻 9. Security Mechanisms

1. Memory limits
2. CPU throttling
3. No network access
4. PID limits
5. Controlled working directory
6. Base64 write to avoid injection

This creates a secure sandbox runtime.

---

10. Extending to Full App Builder (Loveable-Style)

Current system supports:

✔️ Script execution

To support full applications:

You would add:

- package.json support
- npm install inside container
- Dev server execution
- Port exposure
- Dynamic port binding
- Preview URL generation

Example upgrades:

- Expose container port 3000
- Bind to random host port
- Return preview link to frontend

✨ 11. Scaling Considerations

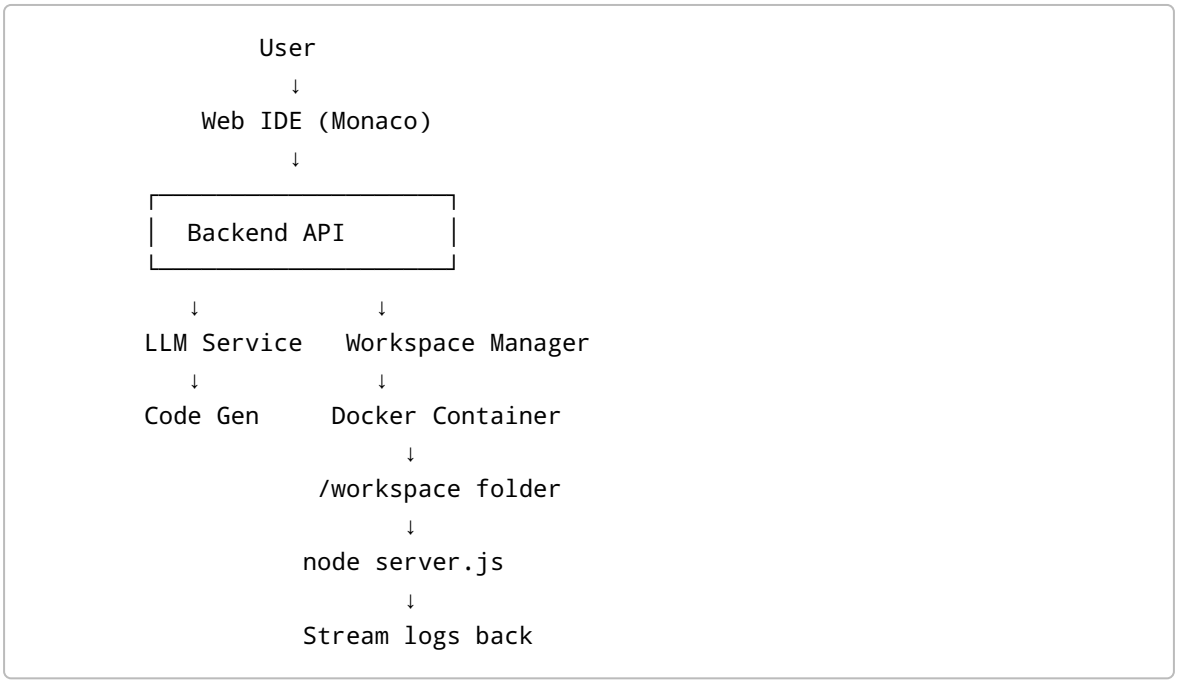
Current limitations:

- In-memory container tracking
- Single server deployment

Production improvements:

- Store state fully in PostgreSQL
- Periodic reconciliation with Docker daemon
- Container health monitoring
- Background cleanup jobs
- Eventually migrate to Kubernetes

🌓 Advanced AI Code Platform Architecture



Final Mental Model

AI generates code. Docker runs code in isolation. Database tracks sessions. Backend orchestrates execution. Frontend displays results.

This architecture forms the foundation of a cloud-based AI coding platform.