# 🏗️ Sandbox Code Execution Platform

## System Design Architecture

This document explains architecture for:

1️⃣ VM / Docker-based Architecture 2️⃣ Kubernetes-based Architecture

---

# 🚢 Problem Statement

We are building a secure multi-tenant code execution platform where:

- Each user gets an isolated runtime
- Code executes safely
- Resources are limited
- Sessions are temporary
- Execution is sandboxed

---

# 🚢 VM / Docker-Based Architecture

## Overview

In this model, the API directly manages Docker containers on a VM.

```
User
  ↓
API Server (Node.js)
  ↓
Docker Engine
  ↓
Containers (1 per user)
```

---

## Components

### 1. API Layer

- Express / Fastify
- Handles authentication
- Creates containers
- Executes code
- Tracks sessions in PostgreSQL

### 2. Docker Engine (On Same VM)

- Creates containers
- Applies resource limits
- Isolates network
- Executes commands

### 3. Database (PostgreSQL)

- Stores user sessions
- Tracks container IDs
- Stores metadata

---

## 🚚 Isolation Strategy

- Memory limit: 512MB
- CPU quota: 50%
- No network access
- PID limit
- Optional read-only root filesystem

---

## 🚡 Container Lifecycle

1. User requests sandbox
2. API checks active container map
3. Docker creates container
4. Container runs idle process
5. API writes user code
6. API exec runs code
7. API collects output
8. Container destroyed after session

---

## 🏩 Scaling Strategy (VM Model)

### Vertical Scaling

- Increase VM RAM/CPU

### Horizontal Scaling

- Multiple VMs
- Load balancer in front
- Shared PostgreSQL

Problem: - Hard to coordinate containers across VMs - Manual scaling - No auto-healing

---

## Limitations

- Single point of failure
- Docker socket exposure risk
- Manual scaling
- No scheduler
- No automatic pod rescheduling

---

# ☸ Kubernetes-Based Architecture

## Overview

In this model, the API talks to Kubernetes API instead of Docker directly.

```
User
  ↓
API Server
  ↓
Kubernetes API
  ↓
Scheduler
  ↓
Worker Nodes
  ↓
Sandbox Pods
```

---

## Components

### 1. API Server

- Uses @kubernetes/client-node
- Creates Pods dynamically
- Executes into Pods
- Deletes Pods after use

### 2. Kubernetes Control Plane

- API Server
- Scheduler
- Controller Manager

### 3. Worker Nodes

- Run sandbox Pods
- Enforce resource limits

### 4. Container Runtime

- containerd
- CRI-O

---

## 🚚 Isolation Strategy

Each sandbox is a Pod with:

- requests/limits memory
- CPU limits
- No Service (no network exposure)
- Optional NetworkPolicy
- SecurityContext restrictions
- Non-root user

---

## 🚡 Pod Lifecycle

1. User requests sandbox
2. API creates Pod
3. Scheduler assigns node
4. Pod pulls image
5. Pod enters Running state
6. API exec writes code
7. API exec runs code
8. API deletes Pod

---

## 💒 Scaling Strategy (K8 Model)

### Horizontal Scaling

- Add worker nodes
- Cluster autoscaler

### API Scaling

- Deploy multiple API replicas
- Kubernetes Service load balances

### Auto-Healing

- If node crashes → Pod rescheduled
- If API crashes → Deployment recreates

---

### 🚜 Security Improvements Over VM Model

- No direct Docker socket exposure
- RBAC controls
- Network policies
- Pod Security Standards
- Namespaces per environment

---

# Architecture Comparison

| Feature | VM + Docker | Kubernetes |
|---|---|---|
| Orchestration | Manual | Automatic |
| Scaling | Manual | Auto-scale |
| Self-healing | No | Yes |
| Multi-node | Hard | Native |
| Isolation | Container-level | Pod-level + policies |
| Production readiness | Medium | High |

---

# 🏢 Production-Grade Architecture (Recommended)

```
            Internet
               ↓
        Load Balancer
               ↓
    API Deployment (ReplicaSet)
               ↓
     Kubernetes API Server
               ↓
       Worker Nodes
               ↓
     Sandbox Execution Pods
               ↓
       PostgreSQL (Managed)
```

---

# 🚀 Advanced Enhancements

## 1. Queue-Based Execution

User Request → Message Queue → Worker Pod → Result

Prevents API overload.

---

## 2. Pre-Warmed Pods

Instead of creating Pods per request:

- Maintain pool of warm Pods
- Assign to user
- Recycle after timeout

Reduces cold start latency.

---

## 3. Namespace Isolation Per Organization

For multi-tenant SaaS:

- One namespace per org
- ResourceQuota per namespace
- LimitRange per namespace

---

# 🏠 Final Recommendation

## For Learning / MVP

VM + Docker is simpler.

## For Production / SaaS Platform

Kubernetes is strongly recommended.

It provides:

- High availability
- Horizontal scaling
- Security boundaries
- Observability
- Cloud-native design

---

# 🚢 Conclusion

VM Model = Simple but limited.

Kubernetes Model = Scalable, secure, cloud-native, production-ready.

If your goal is building something like LeetCode, Replit, CodeSandbox, or interview platforms — Kubernetes is the correct long-term architecture.

---

You are now thinking like a systems architect 🧑‍🚀