Online Code Judge System Design Documentation

1. Overview, Requirements, and Assumptions

A. Functional Requirements:

- Problem Management: Create and manage coding challenges with test cases.
- Code Submission and Evaluation: Users submit code, which is compiled and run in isolated environments.
- Contest and Practice Modes: Support for both competitive contests and practice sessions.
- Plagiarism Detection: Identify duplicate or similar submissions.
- Analytics and Reporting: Provide detailed performance statistics and feedback.

B. Nonfunctional Requirements:

- Low Latency: Code evaluations must complete within seconds.
- High Throughput: Handle thousands of concurrent submissions.
- Scalability: Support millions of users and submissions, especially during contests.
- Fault Tolerance: Provide high availability and recovery in case of node failures.
- Security: Secure code execution in a sandbox and protect user data via encryption.

C. Assumptions:

- Users span the globe; only a subset edit or submit concurrently.
- Supports multiple programming languages with modular compiler integrations.
- The system is built on a microservices architecture with containerized judge nodes.

2. High-Level Architecture and Component Responsibilities

A. Client Tier:

- User Interfaces: Web-based code editor and submission portal using HTTPS and secure WebSocket for real-time updates.

B. Global Access and Routing:

- DNS and regional load balancers direct users to the nearest data center.

C. API Gateway and Authentication:

- API Gateway authenticates requests, enforces rate limits, and routes submissions.
- Authentication Service validates credentials and issues secure tokens (JWT).

D. Submission and Job Scheduling:

- Submission Service validates and enqueues code submissions into a distributed message broker (e.g., Kafka).

E. Code Evaluation Engine:

- Judge Nodes, running as isolated containers, compile and execute code using language-specific compilers/interpreters.
- The system supports sandboxing to isolate execution and enforce resource limits.

F. Data Persistence:

- Submission Results Store captures evaluation outcomes, compiler messages, and execution logs in a distributed NoSQL database.
- Operation Logs are retained for analysis, auditing, and troubleshooting.

G. Analytics and Real-Time Feedback:

- Real-time notifications and dashboards provide immediate feedback and performance metrics.
- Leaderboards and contest rankings are updated based on submissions.

3. Detailed Workflow

A. Submission Flow:

1. A user submits code via the web editor; the code, language, and metadata are sent over HTTPS to the API Gateway.
2. The API Gateway authenticates and forwards the submission to the Submission Service.
3. The Submission Service validates the request and enqueues the submission in a distributed message broker.

B. Code Evaluation Flow:

1. Judge Nodes dequeue submissions from the broker.
2. Each Judge Node creates an isolated sandbox (container) tailored to the submission's language.
3. Code is compiled (if required) and executed against pre-defined test cases with strict timeout and resource limits.
4. Execution results (output, resource usage, error logs) are collected and returned to the Submission Service.

C. Post-Evaluation Processing:

1. Results are stored in the Submission Results Store and operation logs are updated.
2. The user is notified of the outcome in real time via WebSocket, and detailed feedback is available in their account.

D. Analytics and Leaderboards:

1. Aggregated submission data feeds into analytics services for performance tracking and leaderboard updates.

2. Detailed dashboards and reports are provided for both users and administrators.

4. Scalability, Fault Tolerance, and Global Distribution

A. Horizontal Scalability:

- API Gateways, Submission Services, and Judge Nodes are stateless and can scale horizontally behind load balancers.
- The distributed message broker partitions submissions across multiple nodes.
- Data stores are sharded and replicated for high throughput and durability.

B. Fault Tolerance:

- Judge Nodes execute code in isolated containers, limiting the impact of individual failures.
- Redundant services and global failover mechanisms ensure continuity during outages.

C. Global Distribution:

- Regional data centers and global DNS routing minimize latency by directing users to the nearest service instance.
- Autoscaling dynamically adjusts capacity during peak contest periods.

5. Protocols and Security

A. Communication Protocols:

- External communications between clients and servers use HTTPS.
- Real-time updates are delivered over secure WebSocket (WSS).
- Interservice communication employs gRPC or REST over secured TCP channels with mutual TLS.

B. Security and Isolation:

- Code is executed in tightly controlled, sandboxed environments with strict resource limits.
- All sensitive data is encrypted in transit (TLS) and at rest.
- Authentication and RBAC ensure that only authorized users can submit and access problem data.
- Regular security audits and container image scans enhance overall system security.

6. Final Thoughts

This design for an online code judge system (similar to HackerRank) provides a comprehensive framework that meets the challenges of:

- Real-time, secure code evaluation with low latency and robust sandboxing.

- Handling massive concurrent submissions and scaling during peak contest events.
- Supporting multiple programming languages through modular compiler integrations.
- Providing rich feedback, analytics, and leaderboard functionality to engage users.

While actual implementations employ numerous proprietary optimizations and custom hardware, this conceptual design outlines the key architectural principles necessary to build a modern, scalable online code judge platform.