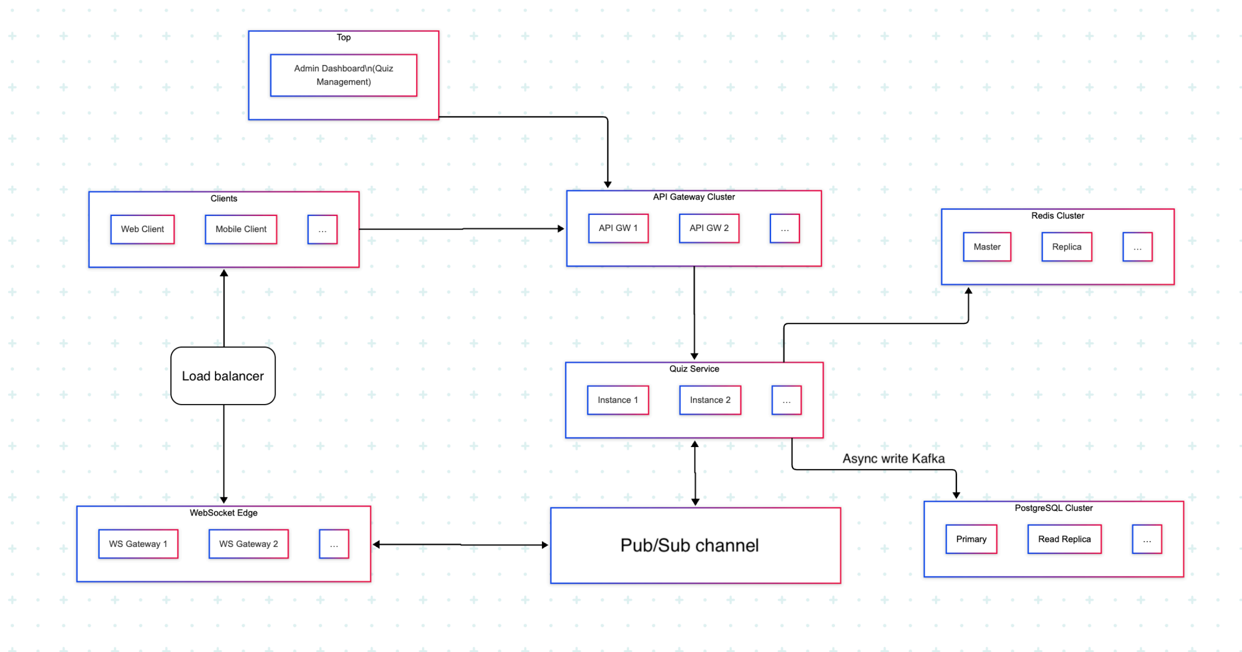
Real time quiz feature engineering proposal

Context

Create a technical solution for a real-time quiz feature for an English learning application. This feature will allow users to answer questions in real-time, compete with others, and see their scores updated live on a leaderboard.

Menu

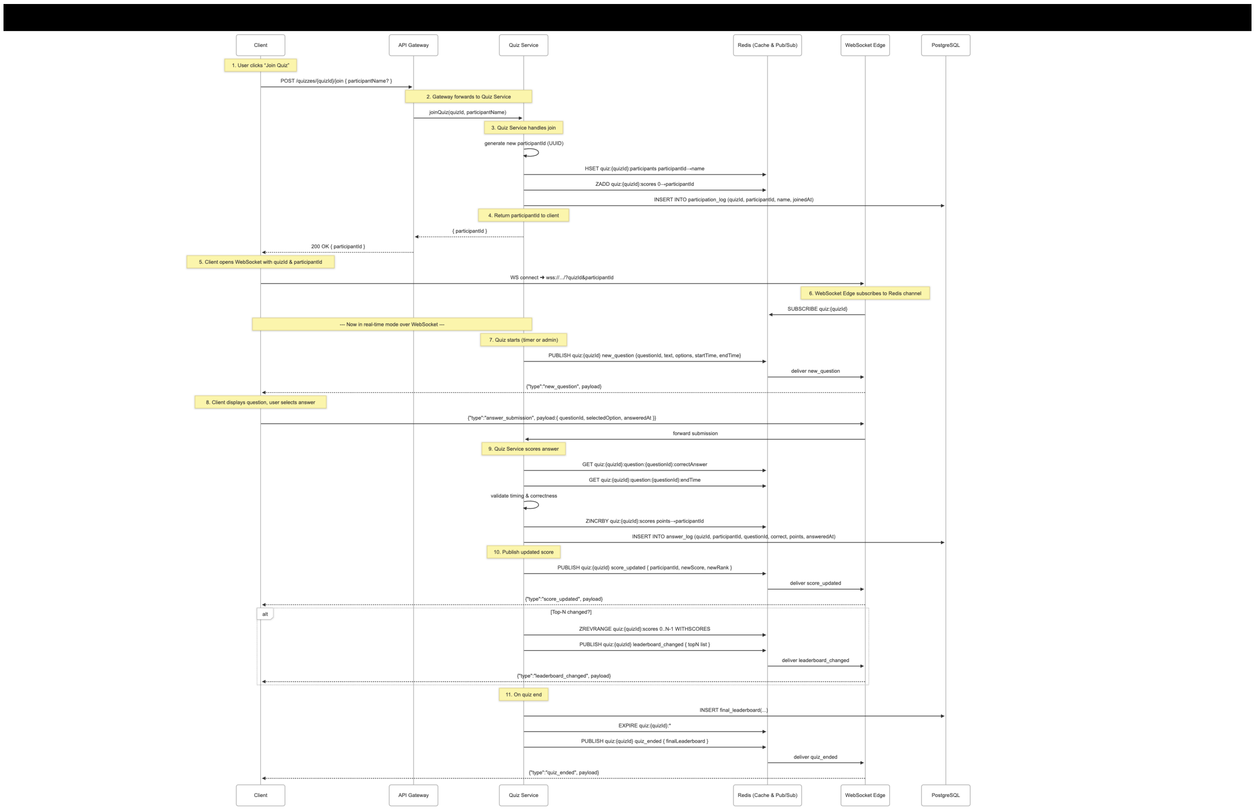
Diagram



Component Descriptions

Component	Role
Admin Dashboard	UI for creating quizzes, defining questions, and monitoring active sessions.
API Gateway Cluster	Exposes REST endpoints (join, metadata), handles auth, rate-limits.
Quiz Service	Core logic: validates joins, sequences questions, scores answers, and publishes events. Stateless & horizontally scalable.
Redis Cluster	In-memory store for active session state (participants, scores, current question) and Pub/Sub broker for real-time events.
WebSocket Edge Cluster	Maintains client socket connections, subscribes to Redis channels, and pushes real-time messages (questions, scores, leaderboard) to participants.
Clients	Web or mobile apps that (a) REST-join the quiz, (b) open one WebSocket, (c) render questions & leaderboards, (d) send answer messages.
PostgreSQL Cluster	Durable storage of quiz definitions, historical logs (joins, answers, final scores) for audit and analytics.

Data flow



- **User Joins the Quiz**
    - The participant enters a quiz ID and clicks "Join."
    - A HTTP request lands at API gateway, which creates a unique `participantId` for them, stores it in our fast in-memory store (Redis), and records the join in our database.
- **WebSocket Connection Established**
    - Armed with their new `participantId`, the client immediately opens a single persistent WebSocket connection, identifying itself by quiz and participant IDs.
    - Behind the scenes, that socket is tied to a Redis Pub/Sub channel for this quiz.

- **Question Broadcast**
  - When the quiz is ready to start when participants > 2 the quiz will start in 5 seconds (my product design), the core Quiz Service publishes a “new question” event into Redis.
  - Our WebSocket layer picks that up and instantly pushes the question (text, options, start/end times) to every connected participant.
- **Answer Submission Over Socket**
  - Each user selects their answer and sends an “answer submission” message over the same WebSocket.
  - That message carries the `questionId`, chosen option, and a timestamp.
- **Scoring & Immediate Feedback**
  - The Quiz Service validates the answer (correctness and timing), then updates the user’s score atomically in Redis.
  - It also logs the detail in PostgreSQL for auditing.
- **Real-Time Score Update**
  - As soon as the score changes, the WebSocket layer relays this in under 100 ms to **all** participants, so everyone sees the updated score and rank.
  - If the top standings shift, the service pulls the top-N list from Redis and publishes a “leaderboard changed” event.
  - Again, WebSocket instantly rebroadcasts that to every client.
- **Quiz Completion & Cleanup**
  - At the end, final scores are written to our durable store, Redis session data expires, and a final “quiz ended” event delivers the conclusive leaderboard to all participants.

Technology Justification

Concern	Chosen Tool(s)	Why This Choice?
REST API	API Gateway (e.g. AWS/GCP)	Centralized auth, throttling, and routing; familiar HTTP patterns for

		simple join operation.
Business Logic	Quiz Service (Spring Boot)	Stateless microservice; easy horizontal scaling; rich ecosystem for Redis & Postgres integration.
In-Memory State & Pub/Sub	Redis Cluster (with Pub/Sub)	Ultra-low latency for score updates; sorted sets for real-time leaderboards; built-in pub/sub.
Real-Time Delivery	WebSocket Edge (Socket.IO)	Single persistent connection per client; scales horizontally; decoupled from quiz logic via Redis.
Persistent Storage	PostgreSQL (Primary + Replicas)	ACID guarantees for quiz definitions and audit logs; read replicas support dashboard/reporting.
Deployment & Scaling	Kubernetes / Managed Containers	Automated scaling, rolling updates, multi-AZ deployments for reliability and maintainability.
Monitoring & Logging	Prometheus + Grafana; ELK / Sentry	Metrics and dashboards for health & performance; centralized logs and error tracking.

Engineering Specification

1. **POST /api/v1/quizzes/{quizId}/join**
    - **Description**: Register a participant for a quiz session.
    - **Path Parameter**:
        - `quizId` (string, required): Unique quiz identifier.
    - **Request Body** (application/json):

    ```
    {
      "participantName": "string"
    }
    ```

    - **Response** (200 OK, application/json):

```
{
  "quizId": "string",        // echoed
  "participantId": "string"   // server-generated UUID
}
```

WebSocket Contract

Clients open a single WebSocket to: `wss://<host>/ws?quizId={quizId}&participantId={participantId}`

All messages are JSON with fields:

```
{
  "type": "string",        // event type
  "payload": { ... }        // event-specific data
}
```

Client → Server Events

1. **answer_submission**

    - **When**: User submits an answer.
    - **Payload**:

    ```
    {
      "questionId": "string",
      "selectedOption": "string",
      "answeredAt": "ISO8601 timestamp"
    }
    ```

2. **get_leaderboard** (optional)

    - **When**: Client needs a full snapshot over WS.
    - **Payload**: `{}`

3. **heartbeat** (optional)

    - **When**: Keep-alive ping.

- **Payload**: `{}`

Server → Client Events

1. **new_question**
    - **When**: Quiz starts or moves to next question.
    - **Payload**:

    ```
    {
      "questionId": "string",
      "text": "string",
      "options": ["string", ...],
      "startTime": "ISO8601 timestamp",
      "endTime": "ISO8601 timestamp"
    }
    ```

2. **leaderboard_changed**
    - **When**: Top-N standings change.
    - **Payload**:

    ```
    {
      "leaderboard": [
        {"participantId":"string","score":number,"rank":integer}
      ]
    }
    ```

3. **leaderboard_data**
    - **When**: Response to `get_leaderboard`.
    - **Payload**: Same structure as `leaderboard_changed`.
4. **quiz_ended**
    - **When**: Quiz concludes.
    - **Payload**:

```
{
  "finalLeaderboard": [
    {"participantId":"string","score":number,"rank":integer}
  ]
}
```

5. **error**
   - **When**: Validation or server error.
   - **Payload**:

```
{
  "reason": "string",
  "details": { … }
}
```

JSON Schema References

- **Participant Object**:

```
{"type":"object",
"properties":{"participantId":{"type":"string"},"participantName":{"type":"string"},"score":{"type":"number"},"rank":{"type":"integer"}},"required":["participantId","score","rank"]}
```

- **Leaderboard Array**: `{"type":"array","items":{ "$ref":"#/definitions/Participant" }}`

Trade-offs

1. Redis Pub/Sub vs. Durable Messaging (e.g. Kafka)

Trade-off	Description
Pros	Redis Pub/Sub is extremely fast and simple to integrate for fanout (WS delivery). Low latency (<5ms).

Cons	No delivery guarantees. No message history or replay. If a WS node is down, it **misses the message**.
Why Acceptable	WS delivery is real-time and ephemeral—missing 1 leaderboard update is not critical. Simplicity and speed were prioritized.

2. In-memory leaderboard with Redis vs. recomputing from DB

Trade-off	Description
Pros	Redis sorted sets (`ZSET`) make score updates and leaderboard queries extremely fast (O(log N)).
Cons	Redis memory can become a bottleneck (especially on hot keys). Must manually enforce TTL or cleanup.
Why Acceptable	Redis is highly optimized for this pattern. We accept higher memory usage to keep leaderboard performance real-time at scale.

3. Async DB writes (Kafka) vs. immediate DB consistency

Trade-off	Description
Pros	Offloads write pressure from `QuizService` . Enables high-QPS submissions without overloading PostgreSQL.
Cons	DB is eventually consistent. Data isn't written immediately after submission. Cannot `SELECT` answers synchronously.
Why Acceptable	UI and real-time experience do not depend on DB writes. Consistency delay of seconds is fine for post-game analysis.

4. Single leaderboard (top-N) vs. per-participant ranking requests

Trade-off	Description
Pros	Top-N is cheap to cache and broadcast to all clients.
Cons	If you want to show "You're ranked #847" to each user, you'd need per-user `ZRANK` , which is expensive at scale.
Why Acceptable	Tradeoff was made in favor of shared leaderboard UX. Per-user rank can be fetched less frequently or delayed.

5. WebSocket delivery only vs. fallback polling

Trade-off	Description
Pros	Keeps client UX real-time and reactive. Low latency push via WS.
Cons	If WS fails or lags, client has no fallback mechanism to poll for state.
Why Acceptable	WS is monitored, and quiz state is ephemeral. For resilience, fallback polling **can** be added later.

Summary Table

Area	Tradeoff Made	Chosen for
Messaging	Redis Pub/Sub vs Kafka	Simplicity, low latency
Leaderboard	Redis vs DB recompute	Real-time responsiveness
Writes	Async via Kafka vs direct DB	Throughput scalability
Ranking model	Top-N vs Per-user rank	Fanout efficiency
Client delivery	WebSocket only vs fallback	Real-time UX

Observability & Monitoring

To ensure the quiz platform operates reliably at scale (up to 1M concurrent users), we apply full-stack observability across services, infrastructure, and real-time flows.

This enables:

- Fast incident detection and diagnosis
- Performance tuning and capacity planning
- Auditability of key events (joins, answers, broadcasts)

Observability Stack

Pillar	Tools
Logs	Loki, ELK (Elasticsearch + Logstash + Kibana), or GCP Cloud Logging
Metrics	Prometheus + Grafana

Tracing	OpenTelemetry + Jaeger or Zipkin
Alerts	Prometheus Alertmanager → Slack, PagerDuty

All services expose `/actuator/prometheus` or native exporters.

Instrumented Components

Quiz Service

- `@Timed` on `joinQuiz()`, `submitAnswer()`
- Counter: `answers_submitted_total{quizId}`
- Timer: `quiz_submission_latency_seconds`
- Kafka producer errors
- Leaderboard update timing

WebSocket Edge

- Active WS connections
- `ws_outgoing_bytes_total`
- `ws_connection_errors_total`
- Redis pub/sub latency (if applicable)

Redis

- `redis_connected_clients`
- `redis_used_memory_bytes`
- `redis_pubsub_channels`
- `redis_commands_duration_seconds`

Kafka

- `kafka_produce_errors_total`
- `kafka_consumergroup_lag`
- `kafka_batch_flush_latency_seconds`

PostgreSQL

- Write QPS
- Replication lag
- Connection pool usage

Dashboards

Category	Panel Example
WebSocket Edge	Active connections per node
Quiz Events	Join rate, submission rate, avg answer latency
Redis	Command rate, memory, slowlogs
Kafka	Per-topic lag, consumer group throughput
PostgreSQL	Insert throughput, slow queries, replication
Errors	4xx/5xx breakdown, Redis failures

Alerting Rules (via Alertmanager)

Alert Condition	Description
`redis_used_memory > 80%`	Redis capacity pressure
`kafka_consumergroup_lag > 10s for 1m`	Async writes falling behind
`quiz_submission_latency_seconds > 500ms P95`	QuizService under load
`ws_connection_errors_total` spike	Broken WS layer or Redis link
`http_5xx_total > 1% over 5m`	Service instability

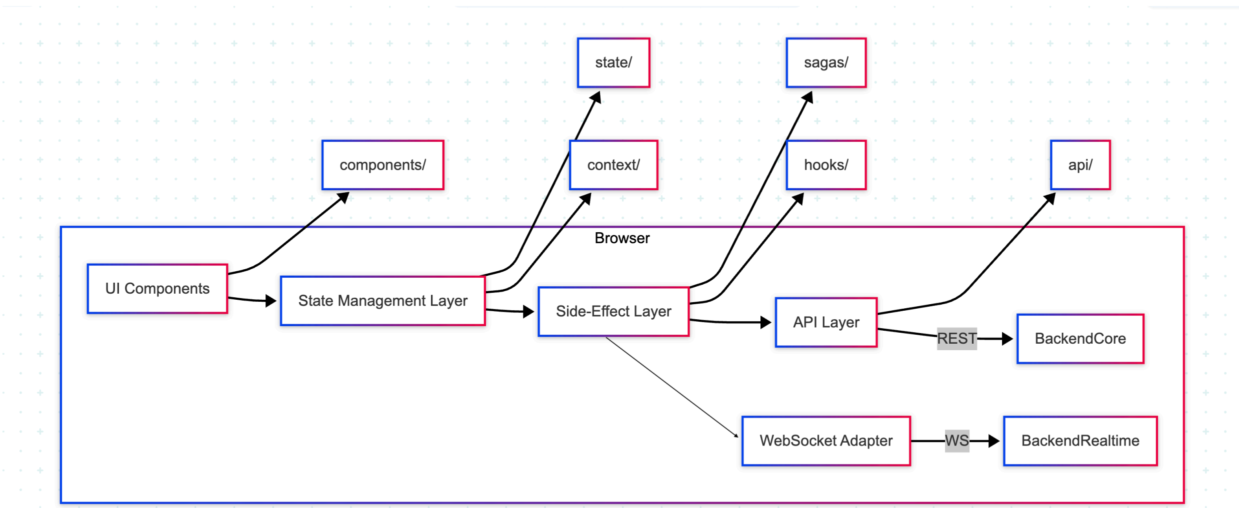
Alerts route to Slack or PagerDuty based on severity.

Logging Strategy

- Use **JSON structured logs**
- Add `quizId`, `participantId`, and `questionId` as contextual fields (careful with cardinality)
- All services forward logs to Loki or ELK

- WS and Quiz logs correlated via request IDs or socket session IDs

Client Architecture



Component Descriptions

Layer / Folder	Description
components/	Presentation layer. Includes `QuizPage`, `Leaderboard`, `QuestionCard`, etc. Renders state and emits user interactions (e.g., join, submit answer).
state/	Redux Toolkit slice (`quizSlice`) defines the global state: quiz ID, participant info, current question, leaderboard, and errors.
sagas/	Implements real-time event handling: joins quiz, opens WebSocket, listens to `new_question` and `leaderboard_changed`, and handles answer submission.
api/	Wraps REST API calls such as `joinQuiz(quizId, name)`, abstracting fetch logic from the UI.
BackendCore	Exposes REST endpoints (`/join`) to register participants and submit answers.

BackendRealtime	WebSocket server that pushes messages like `new_question`, `leaderboard_changed`, and receives events like `answer_submission`.

Data Flow: User Joins → Answers → Leaderboard Updates

A. User Joins a Quiz

1. **UI Action:**
    - User submits name and quiz ID via `QuizPage`.
2. **Redux Dispatch:**
    - `joinRequest({ quizId, name })` is dispatched.
3. **Saga Effect:**
    - `handleJoin()` saga calls `joinQuiz()` API (REST).
    - On success, dispatches `joinSuccess({ quizId, participantId, name })`.
4. **WebSocket Connection:**
    - Saga opens a WebSocket:

    ```
    ws://localhost:8090/ws/quiz?quizId=...&participantId=...
    ```

    - Sets up `eventChannel(ws)` to listen for messages.
    - Forks a listener loop:
        - If `msg.type === 'new_question'` → dispatch `receiveQuestion()`
        - If `msg.type === 'leaderboard_changed'` or `leaderboard_data` → dispatch `receiveLeaderboard()`

B. New Question Broadcast

1. **BackendRealtime** pushes `new_question` via WebSocket.
2. **WebSocket Adapter** receives event.

3. **Saga** catches it → dispatches `receiveQuestion(payload)`.
4. **Redux State** is updated → UI (`QuestionCard`) re-renders.

C. User Submits Answer

1. **UI Action:**
    - User selects an option → dispatches `submitAnswer({ questionId, selectedOption })`.
2. **Saga Effect:**
    - `handleSubmitAnswer()` selects `participantId` from Redux.
    - Constructs message:

    ```
    {
      "type": "answer_submission",
      "payload": {
        "participantId": "...",
        "questionId": "...",
        "selectedOption": "...",
        "answeredAt": "..."
      }
    }
    ```

    - Sends via WebSocket.

D. Leaderboard Update

1. **BackendRealtime** emits `leaderboard_changed` via WebSocket (after scoring logic).
2. **Saga WebSocket listener**:
    - Catches the message → dispatches `receiveLeaderboard()`.
3. **Redux State** updates `leaderboard` array.
4. **Leaderboard UI** re-renders instantly.

Reliability – Retry Strategy

To ensure robustness and seamless user experience during network disruptions or transient backend failures, we implement a **retry strategy** for both the initial quiz join request (`joinQuiz` API call) and the WebSocket connection. This helps minimize user-facing errors during periods of instability while providing visual feedback about ongoing reconnection attempts.

Goals

- Handle temporary failures (e.g., network hiccups, backend downtime).
- Provide automatic retries with exponential backoff.
- Visibly inform users of retry progress and eventual success or failure.
- Avoid infinite loops or aggressive reconnect attempts.

Retry Logic for `joinQuiz` API

The `joinQuiz(quizId, name)` call is wrapped with Redux-Saga's built-in `retry` effect:

```
const res: { quizId: string; participantId: string } = yield retry(3, 1000, joinQuiz, quizId, name);
```

- **Retries**: 3
- **Interval**: 1000 ms between attempts
- **Backoff**: Linear
- **Failure Mode**: Falls through to `joinFailure(err.message)` if all retries fail

This ensures a reasonable number of attempts before presenting an error to the user.

Retry Logic for WebSocket Connection

WebSocket reconnection is handled manually using an exponential backoff strategy. The logic attempts to connect up to 5 times with increasing delays:

```
function* connectWebSocketWithRetry(url: string, maxRetries = 5)
```

- **Retries**: 5
- **Backoff**: `2^attempts * 100` ms (200ms, 400ms, 800ms, etc.)
- **Failure Mode**: Throws error after final attempt

The implementation ensures that the socket is successfully opened before continuing. On failure, the user is notified through the UI.

Resilience & Test Strategy

To ensure production-grade reliability, we adopt a layered test strategy:

Automated Tests

- Unit tests for QuizService logic (joining, scoring, ranking)
- Integration tests for Redis, Kafka, PostgreSQL flows
- Contract tests for WebSocket message schema (consumer/producer validation)

Real-Time Simulation

- Simulate 10k clients submitting answers via locust.io or k6 with WS support
- Verify leaderboard accuracy under load using fuzzed answer inputs

Chaos & Fault Injection

- Redis latency injection using `toxiproxy`
- Kafka consumer kill-switch scenarios
- WS disconnection / reconnection tests
- Region-specific outages simulated via Kubernetes failure injections

Validation Goals

- Score consistency
- Leaderboard ranking under churn
- Graceful degradation with missing WS events

Rollout Plan

To safely and incrementally roll out the real-time quiz platform to production with confidence. This plan minimizes user impact, validates system behavior under load, and ensures rollback is quick if needed.

Deployment Phases

Phase 0 – Internal QA

Scope	Details
Environment	Internal staging (simulated load)
Users	Internal team only
Scale	100–500 concurrent clients
Monitoring	Full observability stack validated
What to Validate	Leaderboard sync, question timing, scoring

Phase 1 – Beta Group (Shadow Traffic)

Scope	Details
Environment	Production infrastructure
Users	Opt-in beta testers or staff
Scale	1–5k concurrent
What to Validate	Real traffic pattern, Redis scaling, WS fanout
Technique	Feature flag / allowlist
Safety	Logs, alerts, and metrics under scrutiny

Phase 2 – Gradual Rollout

Step	% of Users	Actions
Step 1	10%	Monitor for WS stability, Kafka lag
Step 2	25%	Observe Redis memory usage, DB write spikes
Step 3	50%	Validate leaderboard sync across regions
Step 4	100%	Full rollout with on-call active

Each step is gated by:

- No SLO violations
- Error rate < 0.5%
- Kafka lag < 5s
- Redis memory < 70%

Multi-Region Deployment

To support global audiences with low-latency access:

Strategy

- **Redis Cluster:** Deployed in each region as active-active clusters. Quiz sessions are region-localized via quiz ID prefixing (e.g., `eu-quiz123`).
- **WebSocket Edge:** Deployed per region using Anycast DNS or CDN-based routing (e.g., Cloudflare Spectrum, AWS Global Accelerator).
- **API Gateway:** Multi-region, with global routing based on latency (e.g., GCP Load Balancer or AWS ALB with geo-routing).

Quiz Session Affinity

Users join region-local quiz sessions. Cross-region play is not supported in MVP but can be enabled with a Global Redis cache layer or session replication.

Future Work

- “Quiz Session Handoff” between regions (for future cross-region support)

- “Distributed Consistency Testing” for Redis replication correctness

Latency Budget

To maintain a responsive real-time experience, we define the following latency budgets per operation (P95 targets):

Operation	Latency Target (P95)
REST Join API	< 150 ms
WebSocket connection setup	< 200 ms (TLS + Auth)
Submit Answer → Scoring	< 100 ms (end-to-end)
Score → Leaderboard Update	< 80 ms (PubSub + WS fanout)
Quiz End → Final Leaderboard push	< 200 ms

All metrics are monitored via Prometheus histograms and visualized on Grafana dashboards. Alerts are configured to detect spikes beyond budget.

Rollback Plan

Trigger Condition	Rollback Action
Redis saturation (>90%)	Scale Redis cluster / purge keys
Kafka consumer lag > 30s	Pause traffic → investigate
PostgreSQL slow inserts / timeouts	Fallback to write-buffer only mode
WS nodes crashloop / mem spike	Reduce connection cap per node
P99 latency > 500ms across endpoints	Roll back to previous deployment

Rollback is immediate via:

- Blue/green or canary deploy with toggles
- Helm rollback or CI/CD revert
- Feature flag switch-off

Readiness Checklist

Area	Ready?	Notes

Redis scaling	✅	Clustered, key TTL, hot quiz sharding
Kafka buffering	✅	Write path async, consumers monitored
DB write safety	✅	Batched or buffered via stream
WS connection	✅	Load balanced, fanout from pub/sub
Logging	✅	JSON structured, central aggregation
Metrics	✅	Prometheus + Grafana alerts configured
Alerting	✅	PagerDuty / Slack routing in place