System Design Document: Scalable and Resilient PickSpot API

1. Architecture Overview

System Diagram

graph TD

A[Client Apps (Cranes & Yard Apps)] --> B[Load Balancer (Nginx/ALB)]

B --> C[API Pods (Spring Boot in Kubernetes)]

C --> D[In-Memory Cache (Redis Cluster)]

C --> E[Database (PostgreSQL with Read Replicas)]

C --> F[Monitoring Stack (Prometheus + Grafana)]

C --> G[Message Queue (Optional, Kafka)]

The architecture consists of the following components:

- Nginx: Acts as a reverse proxy for load balancing and TLS termination.
- Spring Boot API Pods: Stateless API servers that handle all /pickSpot requests.
- In-Memory Cache (Redis): Provides low-latency yard-map lookups, refreshed every 10 seconds.
- Database (PostgreSQL): Stores persistent data and serves as a fallback for cache misses.
- Monitoring Stack (Prometheus + Grafana): Tracks system performance metrics and alerts on anomalies.

2. Key Design Features

Load Balancer

- Technology: Nginx or a Managed Load Balancer (AWS ALB, GCP Load Balancer).

Features:

- TLS termination for secure communication.
- Round-robin or least-connections strategy for traffic distribution.
- Rate limiting to prevent abuse and overload.

API Pods

- Technology: Spring Boot applications (stateless, deployed as Docker containers in Kubernetes).

Features:

- Each pod processes requests independently, ensuring scalability and fault tolerance.
- Horizontal scaling supports traffic spikes (e.g., add more pods when traffic increases).
- Readiness probes ensure healthy pods serve traffic.

Cache

- Technology: Redis (clustered for high availability).

Features:

- Stores yard-map data for O(1) lookups, minimizing database queries.
- Refreshes every 10 seconds using a background thread.
- TTL (Time-to-Live) ensures cache entries expire after a specific duration to avoid stale data.

Database

- Technology: PostgreSQL with read replicas.

Features:

- ACID-compliant for reliable data storage.
- Read replicas handle high read traffic, offloading the primary database.
- Connection pooling optimizes concurrent database queries.

Monitoring Stack

- Technology: Prometheus + Grafana.

Features:

- Collects real-time metrics like P95 latency, error rates, and CPU usage.
- Provides alerting based on predefined thresholds (e.g., high latency or error rates).
- Supports distributed tracing with tools like Jaeger or Zipkin for request flow analysis.

3. Traffic Handling and Scalability

Normal Day (100 RPS)

- Setup: 2 API pods handle traffic, with each pod processing ~50 RPS.
- Headroom: Each pod has a capacity of ~100 RPS, ensuring sufficient buffer.

Peak Hour (500 RPS)

- Setup: 5 API pods handle traffic, with each pod processing ~100 RPS.
- Scaling: Kubernetes automatically adds pods when CPU utilization exceeds 70%.

Concurrency Model

- Load Balancer: Distributes traffic evenly across pods using round-robin or least-connections strategy.
- Rate Limiting: Throttles excessive requests to prevent backend overload.
- Back-Pressure: A message queue (e.g., Kafka) buffers requests during traffic spikes for asynchronous processing.

4. Failure Handling

API Pod Failure

- Scenario: An API pod crashes or becomes unresponsive.
- Mitigation: Kubernetes restarts the pod, and Nginx routes traffic to healthy pods.

Redis Failure

- Scenario: Redis becomes unavailable.

- Mitigation: The system falls back to the database for yard-map lookups. Performance may degrade until Redis is restored.

Database Failure

- Scenario: The primary database instance fails.
- Mitigation: A read replica is promoted to primary, ensuring minimal downtime.

Load Balancer Failure

- Scenario: The load balancer becomes unresponsive.
- Mitigation: Use a managed load balancer with built-in redundancy and failover.

5. Monitoring and Observability

Key Metrics

P95 Latency:

- Target: ≤ 300 ms.
- Tracks response time for 95% of requests.

Error Rate:

- Target: ≤ 1%.
- Monitors the percentage of failed requests.

Pod CPU Usage:

- Target: < 70%.
- Ensures pods are not overloaded.

Alerts

High Latency:

- Trigger: P95 latency > 400 ms for 5 minutes.
- Action: Notify the on-call engineer via PagerDuty or Slack.

High Error Rate:

- Trigger: Error rate > 5% for 3 minutes.
- Action: Investigate and mitigate the root cause.

Resource Bottleneck:

- Trigger: Pod CPU usage > 90% for 10 minutes.
- Action: Scale up pods or investigate bottlenecks.

6. Deployment Strategy

Blue-Green Deployment

- Deploy the new API version on a separate set of pods.
- Add the new pods to the load balancer while keeping the old version active.
- Monitor the new version for stability.

- Gradually remove the old version from the load balancer.

Enhancements

- Automate deployments using CI/CD pipelines (e.g., Jenkins, ArgoCD).
- Use canary deployments for gradual traffic rollout.

7. Summary

This modified system design ensures:

Scalability: Horizontal scaling of API pods and Redis supports traffic spikes.

Resilience: Fallback mechanisms and clustered components handle failures gracefully.

Observability: Real-time monitoring and alerts ensure system reliability.

The architecture meets the requirements of 500 RPS with ≤ 300 ms P95 latency, providing a robust and efficient solution for heavy traffic scenarios.