ATLAS System Design

Overview

A real-time chat application architecture designed for 100K concurrent users leverages a microservices approach with WebSockets for persistent, low-latency communication. It incorporates a robust message broker for reliable message delivery and fan-out, scalable databases for message history and user metadata, and a caching layer for performance and presence tracking. The system emphasizes horizontal scalability, fault tolerance, and efficient resource utilization.

Components

Component	Purpose	Technology
Client Applications	User interface for sending and receiving messages, managing chat rooms, and user profiles.	React (Web), Swift (iOS), Kotlin (Android)
CDN	Deliver static assets (JS, CSS, images) with low latency and high availability to clients globally.	AWS CloudFront
DNS	Route user requests to the appropriate load balancer.	AWS Route 53
Load Balancer	Distribute incoming HTTP and WebSocket traffic across multiple instances of backend services, ensuring high availability and efficient resource utilization.	AWS Application Load Balancer (ALB)
API Gateway	Centralized entry point for all API requests, handling authentication, authorization, rate limiting, and request routing to various microservices.	Nginx (with Lua for custom logic) or Kong API Gateway

Authentication Service	Manage user registration, login, session management (JWT token generation and validation), and user profile management.	Node.js (NestJS/Express) with Passport.js/JWT
Chat WebSocket Service	Manage persistent WebSocket connections, handle real-time message sending/receiving, presence updates, and publish messages to the Message Broker.	Go (Gorilla WebSocket library) or Node.js (Socket.IO/ws)
Message Broker	Decouple chat services from message processing, enable high-throughput message ingestion, fan-out to multiple subscribers (e.g., message history, other chat instances), and ensure reliable, durable message delivery.	Apache Kafka
Message History Service	Consume messages from the Message Broker, process them, and persist them into the Message History Database for historical retrieval.	Java (Spring Boot) or Go
Message History Database	Store all chat messages for historical retrieval, optimized for high write throughput and efficient time-series queries.	Apache Cassandra (for its distributed, wide-column capabilities)
User/Room Metadata Database	Store user profiles, chat room configurations, direct message mappings, and user-to-room subscriptions, ensuring strong consistency for critical metadata.	PostgreSQL (with sharding)

Caching & Presence Service	Store frequently accessed data (e.g., user profiles, active chat sessions, recent messages) to reduce database load. Track user online/offline status and facilitate real-time presence updates via Pub/Sub.	Redis Cluster
Notification Service	Send push notifications to mobile devices for offline users when new messages or events occur.	Node.js/Java integrating with APNs (Apple Push Notification Service) and FCM (Firebase Cloud Messaging)
Monitoring & Logging	Collect metrics, logs, and traces from all services for system health, performance monitoring, and debugging.	Prometheus (metrics), Grafana (dashboards), ELK Stack (Elasticsearch, Logstash, Kibana for logs), Jaeger (distributed tracing)

Scalability

- Horizontal Scaling: All stateless services (API Gateway, Authentication Service, Chat WebSocket Service, Message History Service, Notification Service) are designed to be scaled horizontally by adding more instances behind load balancers.
- Database Sharding: PostgreSQL for user/room metadata will be sharded by user_id or room_id to distribute data and query load. Apache Cassandra inherently supports horizontal scaling and data distribution across nodes.
- Message Broker Partitioning: Apache Kafka topics will be partitioned across multiple brokers and consumer groups, enabling high message throughput and parallel processing.
- Caching Layer: Redis Cluster provides sharding and replication for high availability, fault tolerance, and scalable caching/presence management.
- Load Balancing: Using AWS ALB ensures efficient distribution of both HTTP and WebSocket connections across backend instances, supporting high concurrency.
- Stateless Design: Most services are designed to be stateless, simplifying scaling and increasing resilience to failures. WebSocket servers manage connections but offload message processing to the message broker.
- Connection Management: The Chat WebSocket Service is optimized to handle a large number of concurrent, long-lived WebSocket connections efficiently.