ATLAS System Design

Overview

A highly scalable and fault-tolerant real-time chat application designed to support 100,000 concurrent users. The architecture leverages WebSockets for instant message delivery, a microservices approach for modularity and independent scaling, and distributed data stores for message persistence and user management. Message queues are used to decouple services, enable efficient message broadcasting, and manage presence.

Components

Component	Purpose	Technology
Load Balancer	Distributes incoming HTTP and WebSocket traffic across multiple instances of API Gateway and WebSocket Servers, ensuring high availability and efficient resource utilization.	HAProxy
API Gateway	Acts as the single entry point for all client requests (non-WebSocket). Handles request routing, authentication, rate limiting, and SSL termination.	NGINX
Authentication Service	Manages user registration, login, and issues/validates JSON Web Tokens (JWTs) for secure API access.	Node.js (Express) with JWT
User Service	Manages user profiles, friend lists, contact management, and user metadata.	Node.js (Express)

WebSocket Servers (Chat Service)	Manages persistent WebSocket connections with clients, handles real-time message ingress and egress, and communicates with message brokers for message routing and broadcasting.	Node.js (Socket.IO)
Message Broker (Real-time)	Facilitates real-time message broadcasting between WebSocket servers, ensuring messages are delivered to all relevant connected users across the cluster.	Redis Pub/Sub
Message Broker (Durable)	Provides a high-throughput, fault-tolerant, and durable message streaming platform for inter-service communication, event logging, and processing chat messages before persistence.	Apache Kafka
Message Persistence Service	Stores chat messages for historical retrieval, ensuring high write throughput and distributed storage.	Cassandra
Presence Service	Tracks user online/offline status, last seen timestamps, and potentially user-to-WebSocket server mappings for efficient message delivery.	Redis
Database (User & Metadata)	Stores relational data such as user accounts, chat room configurations, friend relationships, and other structured metadata.	PostgreSQL

Cache Service	Caches frequently accessed data like user profiles, chat room details, and session tokens to reduce database load and improve response times.	Redis

Scalability

- **Horizontal Scaling**: All stateless microservices (Auth, User, WebSocket Servers) can be scaled horizontally by adding more instances behind the Load Balancer.
- **Load Balancing**: HAProxy efficiently distributes TCP/WebSocket connections and HTTP requests across multiple service instances, preventing single points of failure and distributing load.
- **Message Queues**: Apache Kafka provides high-throughput, fault-tolerant message streaming for durable message processing, while Redis Pub/Sub enables efficient real-time message broadcasting across WebSocket server clusters, allowing independent scaling of message producers and consumers.
- **Database Sharding**: Cassandra's distributed architecture inherently handles sharding for chat messages, providing high write availability and linear scalability. PostgreSQL can be scaled with read replicas and logical sharding for user and metadata.
- **Caching**: Redis reduces the load on databases by serving frequently accessed data from in-memory caches, improving read performance and reducing latency.
- **Stateless Services**: Most services are designed to be stateless, simplifying horizontal scaling and resilience to instance failures.
- **Connection Management**: WebSocket servers are designed to handle many concurrent connections, and the distributed Pub/Sub mechanism ensures that any message published by one server can be efficiently delivered to users connected to any other server in the cluster.