# **ChatFlow - Real-Time Chat Application Architecture Documentation**

## **Table of Contents**

1. Architecture Overview
2. Tech Stack
3. Data Flow
4. Database Design
5. Scalability Plan
6. Performance Optimization
7. Security Considerations

## **1. Architecture Overview**

ChatFlow uses a **microservices-based architecture** with separation of concerns between real-time communication (WebSocket servers) and traditional REST API operations. The architecture is designed to be horizontally scalable, fault-tolerant, and optimized for low-latency messaging.

### **Key Design Principles**

* **Separation of Concerns**: WebSocket servers handle real-time messaging; REST APIs handle CRUD operations
* **Stateless Services**: All application servers are stateless to enable horizontal scaling
* **Event-Driven Architecture**: Kafka message queues for asynchronous processing
* **Caching Strategy**: Multi-layer caching to reduce database load
* **Database Optimization**: Polyglot persistence (PostgreSQL + MongoDB)

## **2. Tech Stack**

### **Frontend Layer**

* **Web Application**: React 18+ with TypeScript  
  + State Management: Redux Toolkit + RTK Query
  + Real-time: Socket.io-client
  + UI Framework: Tailwind CSS + shadcn/ui
  + Build Tool: Vite
* **Mobile Apps**: React Native  
  + Cross-platform iOS/Android
  + Native push notification support
* **Desktop App**: Electron  
  + System tray integration
  + Native notifications

### **Backend Layer**

* **REST API Servers**: Node.js 20+ with Express.js  
  + Authentication: JWT (JSON Web Tokens)
  + Validation: Joi/Zod
  + Rate Limiting: Redis-based
* **WebSocket Servers**: Node.js with Socket.io or uWebSockets.js  
  + Connection pooling
  + Automatic reconnection handling
  + Room-based broadcasting

### **Database Layer**

* **Primary Database**: PostgreSQL 15+  
  + Stores: Users, Channels, Permissions, Sessions
  + ACID compliance for critical data
  + Read replicas for scaling reads
* **Message Storage**: MongoDB  
  + High-write throughput for messages
  + Flexible schema for message metadata
  + Sharding support for horizontal scaling
* **Search Engine**: Elasticsearch  
  + Full-text search on messages
  + Analytics and aggregations
  + Real-time indexing

### **Caching & Message Queue**

* **Redis Cluster**
  + Session storage
  + User presence data
  + Recent message cache (last 100 messages per channel)
  + Rate limiting counters
* **Redis Pub/Sub**
  + Real-time message broadcasting between WebSocket servers
  + Typing indicators
  + Presence updates
* **Apache Kafka**
  + Event sourcing
  + Asynchronous task processing
  + Analytics events

### **Infrastructure**

* **Load Balancer**: Nginx  
  + SSL/TLS termination
  + WebSocket upgrade support
  + Health checks
* **API Gateway**: Kong or AWS API Gateway  
  + Request routing
  + Authentication/Authorization
  + Rate limiting (100 requests/min per user)
* **CDN**: CloudFlare  
  + Static asset delivery
  + Image optimization
  + DDoS protection

### **Storage & External Services**

* **Object Storage**: AWS S3 or MinIO  
  + File attachments (images, documents, videos)
  + Avatar images
  + Backup storage
* **Push Notifications**
  + FCM (Firebase Cloud Messaging) for Android
  + APNs (Apple Push Notification service) for iOS
* **Email Service**: SendGrid or AWS SES  
  + Account verification
  + Password reset
  + Notification emails

### **Monitoring & Observability**

* **Metrics**: Prometheus + Grafana
* **Logging**: ELK Stack (Elasticsearch, Logstash, Kibana)
* **Tracing**: Jaeger or OpenTelemetry
* **Error Tracking**: Sentry

## **3. Data Flow**

### **Message Sending Flow**

1. **Client → Load Balancer**: User sends message via WebSocket connection
2. **Load Balancer → WebSocket Server**: Routes to appropriate WS server
3. **WebSocket Server → Validation**: Validates message, checks permissions
4. **WebSocket Server → MongoDB**: Persists message to database
5. **WebSocket Server → Redis Pub/Sub**: Publishes message to channel
6. **Redis Pub/Sub → All WS Servers**: Broadcasts to all connected servers
7. **All WS Servers → Clients**: Delivers message to all channel members
8. **WebSocket Server → Kafka**: Sends event for async processing
9. **Background Workers → Elasticsearch**: Indexes message for search
10. **Background Workers → Push Services**: Sends notifications to offline users

### **Authentication Flow**

1. **Client → REST API**: POST /auth/login with credentials
2. **REST API → PostgreSQL**: Validates credentials
3. **REST API → Redis**: Creates session, stores JWT
4. **REST API → Client**: Returns JWT token + refresh token
5. **Client → WebSocket**: Establishes WS connection with JWT
6. **WebSocket Server → Redis**: Validates JWT, loads session
7. **WebSocket Server**: Authorizes connection, joins user to rooms

### **File Upload Flow**

1. **Client → REST API**: POST /upload with file
2. **REST API → Validation**: Checks file type, size (max 50MB)
3. **REST API → S3**: Uploads file, generates signed URL
4. **REST API → PostgreSQL**: Creates attachment record
5. **REST API → Client**: Returns file URL and metadata
6. **Client → WebSocket**: Sends message with attachment reference
7. **Background Worker**: Generates thumbnails for images/videos

## **4. Database Design**

### **PostgreSQL Schema (Relational Data)**

#### **Users Table**

CREATE TABLE users (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

email VARCHAR(255) UNIQUE NOT NULL,

username VARCHAR(50) UNIQUE NOT NULL,

password\_hash VARCHAR(255) NOT NULL,

role VARCHAR(20) DEFAULT 'user' CHECK (role IN ('admin', 'moderator', 'user')),

display\_name VARCHAR(100),

avatar\_url TEXT,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

last\_seen TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

is\_active BOOLEAN DEFAULT true,

settings JSONB DEFAULT '{}'::jsonb

);

CREATE INDEX idx\_users\_email ON users(email);

CREATE INDEX idx\_users\_username ON users(username);

CREATE INDEX idx\_users\_last\_seen ON users(last\_seen);

#### **Channels Table**

CREATE TABLE channels (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

name VARCHAR(100) NOT NULL,

description TEXT,

type VARCHAR(20) CHECK (type IN ('public', 'private', 'direct')),

created\_by UUID REFERENCES users(id) ON DELETE SET NULL,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

is\_archived BOOLEAN DEFAULT false,

metadata JSONB DEFAULT '{}'::jsonb

);

CREATE INDEX idx\_channels\_type ON channels(type);

CREATE INDEX idx\_channels\_created\_by ON channels(created\_by);

#### **User\_Channels Table (Many-to-Many)**

CREATE TABLE user\_channels (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

user\_id UUID REFERENCES users(id) ON DELETE CASCADE,

channel\_id UUID REFERENCES channels(id) ON DELETE CASCADE,

role VARCHAR(20) DEFAULT 'member' CHECK (role IN ('owner', 'admin', 'moderator', 'member')),

joined\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

last\_read\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

is\_muted BOOLEAN DEFAULT false,

is\_pinned BOOLEAN DEFAULT false,

UNIQUE(user\_id, channel\_id)

);

CREATE INDEX idx\_user\_channels\_user ON user\_channels(user\_id);

CREATE INDEX idx\_user\_channels\_channel ON user\_channels(channel\_id);

CREATE INDEX idx\_user\_channels\_last\_read ON user\_channels(last\_read\_at);

### **MongoDB Collections (Document Data)**

#### **Messages Collection**

{

\_id: ObjectId,

messageId: "uuid-v4",

channelId: "uuid-v4",

userId: "uuid-v4",

content: "Message text",

type: "text|file|system",

replyToMessageId: "uuid-v4|null",

createdAt: ISODate,

updatedAt: ISODate,

deletedAt: ISODate|null,

isEdited: false,

isPinned: false,

metadata: {

mentions: ["userId1", "userId2"],

links: ["https://example.com"],

hashtags: ["topic1"]

}

}

// Indexes

db.messages.createIndex({ channelId: 1, createdAt: -1 })

db.messages.createIndex({ userId: 1, createdAt: -1 })

db.messages.createIndex({ "metadata.mentions": 1 })

db.messages.createIndex({ replyToMessageId: 1 })

#### **Message\_Reactions Collection**

{

\_id: ObjectId,

messageId: "uuid-v4",

userId: "uuid-v4",

emoji: "👍",

createdAt: ISODate

}

// Compound index to prevent duplicate reactions

db.message\_reactions.createIndex({ messageId: 1, userId: 1, emoji: 1 }, { unique: true })

### **Elasticsearch Index (Search Data)**

#### **Messages Index**

{

"mappings": {

"properties": {

"messageId": { "type": "keyword" },

"channelId": { "type": "keyword" },

"userId": { "type": "keyword" },

"content": {

"type": "text",

"analyzer": "standard",

"fields": {

"keyword": { "type": "keyword" }

}

},

"createdAt": { "type": "date" },

"mentions": { "type": "keyword" },

"hashtags": { "type": "keyword" }

}

}

}

### **Database Relationships**

* **Users → Channels**: Many-to-Many (via user\_channels)
* **Users → Messages**: One-to-Many
* **Channels → Messages**: One-to-Many
* **Messages → Reactions**: One-to-Many
* **Messages → Attachments**: One-to-Many
* **Messages → Read Receipts**: One-to-Many

## **5. Scalability Plan**

### **Horizontal Scaling Strategy**

#### **Application Layer Scaling**

* **Stateless Services**: All API and WebSocket servers are stateless
* **Auto-Scaling**: Kubernetes HPA (Horizontal Pod Autoscaler)
  + Scale based on CPU (>70%), memory (>80%), or custom metrics
  + Min: 3 replicas, Max: 50 replicas per service
* **WebSocket Server Scaling**:
  + Each server handles 10,000 concurrent connections
  + Redis Pub/Sub ensures message delivery across all servers
  + Sticky sessions based on user ID for connection stability

#### **Database Scaling**

**PostgreSQL Scaling**

* **Read Replicas**: 3-5 read replicas for query distribution
  + User queries → Read replicas
  + Writes → Primary database
  + Replication lag monitoring (<100ms)
* **Connection Pooling**: PgBouncer with 200 connections per pool
* **Vertical Scaling**: Initial deployment on 16 vCPU, 64GB RAM
* **Future**: Citus for horizontal sharding if needed (>10M users)

**MongoDB Scaling**

* **Sharding Strategy**: Shard by channelId
  + Each shard handles messages for a subset of channels
  + 3-node replica sets per shard
* **Initial Setup**: 3 shards, 9 total nodes
* **Scaling**: Add shards as message volume grows (1B+ messages)

**Redis Scaling**

* **Redis Cluster**: 6-node cluster (3 master, 3 replica)
* **Separate Clusters**:
  + Cluster 1: Session & cache data
  + Cluster 2: Pub/Sub messaging
* **Memory Management**: LRU eviction for cache, no eviction for sessions

#### **Message Queue Scaling**

* **Kafka Cluster**: 5-7 brokers initially
* **Partitioning**: 20 partitions per topic
  + Messages topic: Partitioned by channelId
  + Notifications topic: Partitioned by userId
* **Consumer Groups**: Multiple workers per consumer group
* **Retention**: 7 days for messages, 30 days for analytics

### **Geographic Distribution**

* **Multi-Region Deployment**:
  + US East, US West, Europe, Asia regions
  + User routing based on geographic proximity
* **Data Replication**:
  + PostgreSQL: Streaming replication across regions
  + MongoDB: Global clusters with zone-aware sharding
* **CDN**: Multi-region edge locations

### **Load Balancing Strategy**

* **Layer 4**: TCP load balancing for WebSocket connections
* **Layer 7**: HTTP load balancing for REST APIs
* **Health Checks**: Every 5 seconds, 2 consecutive failures → remove from pool
* **Connection Draining**: 30-second grace period before shutdown

### **Capacity Planning**

| **Metric** | **Small Scale** | **Medium Scale** | **Large Scale** |
| --- | --- | --- | --- |
| Active Users | 10K | 100K | 1M+ |
| Concurrent Connections | 5K | 50K | 500K+ |
| Messages/Second | 100 | 1,000 | 10,000+ |
| API Servers | 3 | 10 | 50+ |
| WebSocket Servers | 3 | 10 | 100+ |
| PostgreSQL | 1 primary + 2 replicas | 1 primary + 5 replicas | Sharded |
| MongoDB | 3-node replica set | 3 shards (9 nodes) | 10+ shards |
| Redis | 3-node cluster | 6-node cluster | 12+ nodes |

## **6. Performance Optimization**

### **Message Delivery Optimization**

#### **1. WebSocket Connection Management**

* **Connection Pooling**: Reuse WebSocket connections
* **Heartbeat Mechanism**: Ping/pong every 30 seconds
* **Automatic Reconnection**: Exponential backoff (1s, 2s, 4s, 8s, 30s max)
* **Connection State Recovery**: Resume from last received message

#### **2. Message Queuing & Batching**

* **Client-Side Queue**: Queue messages during network issues
* **Message Batching**: Batch up to 10 messages or 100ms interval
* **Deduplication**: Message IDs to prevent duplicates
* **Retry Logic**: Max 3 retries with exponential backoff

#### **3. Adaptive Message Loading**

// Progressive loading strategy

- Initial load: Last 50 messages

- Scroll up: Load 50 more (pagination)

- Real-time: Append new messages

- Throttle scroll events: 100ms debounce

### **Caching Strategy**

#### **Multi-Layer Cache Architecture**

**Layer 1: Client-Side Cache**

* IndexedDB for message history (last 500 messages per channel)
* LocalStorage for user preferences
* Service Worker for offline support

**Layer 2: CDN Cache**

* Static assets: 1 year TTL
* User avatars: 7 days TTL
* File attachments: 30 days TTL

**Layer 3: Redis Cache**

// Cache structure

users:{userId} → TTL: 1 hour

channels:{channelId}:members → TTL: 30 minutes

channels:{channelId}:messages:recent → TTL: 5 minutes (last 100 messages)

presence:{userId} → TTL: 5 minutes

typing:{channelId} → TTL: 5 seconds

**Layer 4: Database Query Cache**

* PostgreSQL: Prepared statements
* MongoDB: Query result cache (100MB)

### **Network Optimization**

#### **1. Compression**

* **WebSocket**: Enable permessage-deflate extension
* **HTTP**: Gzip/Brotli compression (80% size reduction)
* **Images**: WebP format with progressive loading

#### **2. Protocol Optimization**

* **HTTP/2**: Multiplexing, header compression
* **WebSocket Binary**: Use binary frames for large payloads
* **Message Format**: Protocol Buffers or MessagePack (vs JSON)

#### **3. Adaptive Quality**

// Network quality detection

if (connectionSpeed === 'slow') {

- Reduce image quality (thumbnail instead of full)

- Disable typing indicators

- Increase message batch size

- Load fewer messages (25 vs 50)

}

### **Database Performance**

#### **1. Query Optimization**

* **Prepared Statements**: Reduce parsing overhead
* **Query Planning**: EXPLAIN ANALYZE for slow queries
* **Index Strategy**:
  + Composite indexes for common queries
  + Partial indexes for filtered queries
  + GIN indexes for JSONB columns

#### **2. Connection Management**

* **Connection Pooling**: Min 10, Max 100 per server
* **Query Timeout**: 5 seconds max
* **Idle Connection Timeout**: 10 minutes

#### **3. Data Archival**

* **Hot Storage**: Last 90 days in primary database
* **Warm Storage**: 90-365 days in compressed format
* **Cold Storage**: >1 year in S3 with metadata only

### **API Performance**

#### **1. Rate Limiting**

// Rate limits per user

- Messages: 10/second, 100/minute

- API calls: 100/minute

- File uploads: 5/minute

- Burst allowance: 20 requests

#### **2. Pagination**

* **Cursor-Based**: For message history
* **Offset-Based**: For user/channel lists
* **Page Size**: Default 50, max 100

#### **3. Response Optimization**

* **Field Selection**: Return only requested fields
* **Lazy Loading**: Load attachments/reactions on demand
* **ETags**: HTTP caching for unchanged resources

### **Real-Time Features Optimization**

#### **1. Typing Indicators**

* **Throttling**: Update every 2 seconds max
* **Debouncing**: Stop indicator after 5 seconds of no activity
* **Broadcasting**: Only to channel members, not globally

#### **2. Presence/Status**

* **Batch Updates**: Update presence every 60 seconds
* **Lazy Propagation**: Only update when status changes
* **Staleness Detection**: Mark offline after 5 minutes

#### **3. Read Receipts**

* **Batch Updates**: Send read receipts in batches of 10
* **Debouncing**: Update after 1 second of scrolling stop
* **Selective Tracking**: Only for direct messages and small groups (<10 members)

### **File Handling Optimization**

#### **1. Upload Optimization**

* **Chunked Upload**: 5MB chunks for files >20MB
* **Parallel Upload**: Up to 3 concurrent chunks
* **Resume Support**: Resume interrupted uploads
* **Client-Side Compression**: Images compressed before upload

#### **2. Download Optimization**

* **Signed URLs**: Direct S3 access, bypass API
* **Progressive Download**: Stream large files
* **Thumbnail Generation**: 100x100, 300x300, 600x600
* **Lazy Loading**: Load images in viewport only

### **Monitoring & Alerting**

#### **Key Performance Metrics**

* **P50, P95, P99 Latency**: Message delivery time
* **WebSocket Connection Count**: Active connections per server
* **Message Throughput**: Messages/second
* **Error Rate**: Failed messages/total messages
* **Database Query Time**: P95 < 50ms
* **Cache Hit Rate**: >90%

#### **Alert Thresholds**

* P99 latency > 500ms
* Error rate > 1%
* CPU usage > 80% for 5 minutes
* Memory usage > 85%
* Cache hit rate < 80%
* WebSocket connections > 8,000 per server

## **7. Security Considerations**

### **Authentication & Authorization**

* **Password Security**: Bcrypt with cost factor 12
* **JWT Tokens**: HS256, 15-minute expiry, refresh token rotation
* **Session Management**: Redis-based, 30-day expiry
* **MFA Support**: TOTP for admin/moderator accounts

### **Data Protection**

* **Encryption at Rest**: AES-256 for database and S3
* **Encryption in Transit**: TLS 1.3 for all communications
* **End-to-End Encryption**: Optional for direct messages (future)

### **Input Validation**

* **Message Sanitization**: XSS prevention, HTML entity encoding
* **File Upload Validation**: MIME type checking, virus scanning
* **Rate Limiting**: DDoS protection, abuse prevention

### **Access Control**

* **Role-Based Access Control (RBAC)**: Admin, Moderator, User roles
* **Channel Permissions**: Read, Write, Delete, Manage permissions
* **Audit Logging**: All administrative actions logged

### **Privacy**

* **Data Retention**: Configurable per channel
* **GDPR Compliance**: Data export, right to be forgotten
* **Message Deletion**: Soft delete with 30-day grace period

## **Conclusion**

This architecture is designed to support ChatFlow's growth from thousands to millions of users while maintaining low latency and high reliability. The key strengths are:

1. **Scalability**: Horizontal scaling at every layer
2. **Performance**: Multi-layer caching and optimized protocols
3. **Reliability**: Redundancy and fault tolerance
4. **Flexibility**: Microservices allow independent scaling and updates

The architecture can start simple (3 servers, 1 database) and scale incrementally as user base grows, ensuring cost-effectiveness while maintaining room for growth.