CAFFIS CHAT SERVICE

Complete Implementation Plan & Technical Specification

Version: 1.0
Date: August 2025
Project: Enterprise-Grade Real-Time Messaging Microservice
Author: Technical Architecture Team

📋 TABLE OF CONTENTS

---

1. EXECUTIVE SUMMARY

Project Overview

The Caffis Chat Service is a production-grade, enterprise-level real-time messaging microservice designed to integrate seamlessly with the existing Caffis ecosystem. This service will provide WhatsApp/Telegram-level messaging functionality specifically tailored for coffee meetups and social connections.

Business Value

- **Enhanced User Engagement**: Real-time communication increases user retention
- **Seamless Social Experience**: Automatic chat creation from map-based invitations
- **Scalable Architecture**: Enterprise-grade performance for growing user base
- **Revenue Opportunities**: Foundation for premium messaging features

Key Success Factors

- **Perfect Integration**: Maintains existing Apple design system consistency
- **High Performance**: C++ backend handles thousands of concurrent connections
- **Rich Features**: WhatsApp-level functionality with coffee meetup context
- **Clean Architecture**: Microservice design enables independent scaling

---

2. CURRENT SYSTEM ANALYSIS

Existing Architecture

```
📁 Current Project Structure:
├── caffis/              # Main social app
│   ├── client/          # Next.js frontend (Port 3000)
│   └── server/          # Node.js backend (Port 5000)
├── map-service/         # Location-based discovery
│   ├── frontend/        # React frontend (Port 3001)
│   └── backend/         # Node.js backend (Port 5001)
├── docker-compose.unified.yml # Container orchestration
└── PostgreSQL Database     # Shared user system
```

Current Capabilities

- **User Authentication**: JWT-based auth system with email verification
- **Social Invitations**: Users can create and accept coffee meetup invitations
- **Location Services**: Real-time user discovery with Google Places integration
- **Apple Design System**: Consistent glassmorphism UI with custom gradients
- **Dockerized Deployment**: Complete containerization setup

Integration Points Identified

- **Shared User System**: Common PostgreSQL database with user profiles
- **JWT Authentication**: Existing token system can validate chat users

- **Invitation System**: Map service triggers for automatic chat creation
- **Design Consistency**: Established CSS variables and component patterns

---

3. PROJECT OBJECTIVES

Primary Objectives

1. **Seamless Integration**: Chat service appears naturally within existing user flows
2. **Enterprise Performance**: Handle 10,000+ concurrent WebSocket connections
3. **Rich Messaging**: WhatsApp/Telegram-level feature parity
4. **Social Context**: Coffee meetup-aware messaging with location integration
5. **Relationship Management**: "Break up" functionality for clean disconnections

Secondary Objectives

1. **Learning Goals**: Gain production-level C++ experience
2. **Scalability**: Foundation for future messaging enhancements
3. **Revenue Preparation**: Architecture ready for premium features
4. **Mobile Readiness**: WebSocket architecture supports future mobile apps

Success Criteria

- **Performance**: <50ms message delivery latency
- **Reliability**: 99.9% uptime during peak usage
- **User Adoption**: 80% of accepted invitations result in chat usage
- **Integration Quality**: Zero disruption to existing user experiences

---

4. TARGET ARCHITECTURE

Complete Ecosystem Overview

```
┌─────────────────┐  ┌─────────────────┐  ┌─────────────────┐
│   MAIN APP      │  │   MAP SERVICE   │  │  CHAT SERVICE   │
│   Port: 3000    │  │   Port: 3001    │  │   Port: 3003    │
│   Backend: 5000 │  │   Backend: 5001 │  │   Backend: 5002 │
├─────────────────┤  ├─────────────────┤  ├─────────────────┤
│ • Authentication│◄─►│ • User Discovery│◄─►│ • Real-time Chat│
│ • User Profiles │  │ • Location Match│  │ • Message History│
│ • Invitations   │  │ • Place Selection│  │ • File Upload   │
│ • Dashboard     │  │ • Invite System │  │ • Room Management│
└─────────────────┘  └─────────────────┘  └─────────────────┘
         │                    │                    │
         └────────────────────┼────────────────────┘
                              │
                              ▼
                    ┌─────────────────┐
                    │  POSTGRESQL DB  │
                    │ + Redis Cluster │
                    │ Shared & Extended│
                    └─────────────────┘
```

New Components

- **Chat Service Backend**: C++ WebSocket server (Port 5002)
- **Chat Service Frontend**: React interface (Port 3003)
- **Redis Cluster**: Message caching and pub/sub scaling
- **File Storage**: MinIO for images and file attachments
- **Extended Database**: New chat-specific tables

Technology Stack

- **Backend**: C++ with Boost.Beast WebSocket library
- **Database**: PostgreSQL with libpqxx C++ driver
- **Caching**: Redis for message queuing and scaling
- **Frontend**: React with TypeScript, matching existing design
- **Real-time**: WebSocket connections with fallback to polling
- **File Storage**: MinIO (S3-compatible) for media uploads

---

5. USER JOURNEY & INTEGRATION

Primary User Flow

1. **Authentication**
    - User logs into main Caffis app
    - JWT token issued, valid across all services
2. **Discovery Phase**
    - User clicks "Trova Cafe" in main dashboard
    - Redirected to Map Service (Port 3001)
    - Discovers nearby coffee lovers
3. **Connection Initiation**
    - User sends location-based meetup invitation
    - Map Service handles invitation logic
    - Invitation includes location and meetup preferences
4. **Chat Activation Trigger**
    - Recipient accepts invitation in Map Service
    - Map Service sends webhook to Chat Service
    - Chat Service automatically creates room
5. **Real-time Messaging**
    - Both users receive chat room notification
    - Chat interface opens (Port 3003) in new tab/popup
    - Full-featured messaging begins immediately
6. **Group Expansion**
    - Users can invite additional people to meetup
    - Chat room expands to group conversation
    - All participants see meetup context
7. **Relationship Management**
    - Users can "disconnect" through main app
    - Chat Service removes all conversation history
    - Clean slate for future interactions

Integration Flow Diagram

```
Main App → "Trova Cafe" → Map Service → Invitation → Acceptance →
Chat Service Webhook → Room Creation → User Notifications →
Chat Interface Launch → Real-time Messaging
```

Cross-Service Communication

- **Main App** ↔ **Chat Service**: User authentication validation
- **Map Service** ↔ **Chat Service**: Invitation acceptance webhooks
- **Chat Service** ↔ **All Services**: User relationship status checks
- **All Services** ↔ **Database**: Shared user profiles and preferences

---

6. FEATURE SPECIFICATIONS

Core Messaging Features

Real-time Communication

- WebSocket-based instant messaging
- Message delivery confirmation (Sent/Delivered/Read)
- Typing indicators with user names
- Online/offline status indicators
- Message timestamps with timezone handling

Message Management

- Send text messages with emoji support
- Edit messages (5-minute time limit)
- Delete messages for self or everyone (if unread)
- Reply to specific messages with threading
- Forward messages between chat rooms
- Message search within conversation history

Rich Media Support

- Image upload and sharing with compression
- File attachments (PDFs, documents) up to 25MB
- Location sharing with map integration

- Voice messages (future enhancement)
- GIF and sticker support
- Preview links with metadata extraction

Advanced Messaging Features

Group Chat Management

- Create group chats from existing 1-on-1 conversations
- Add/remove participants (admin controls)
- Group naming and description
- Admin role management
- Participant list with online status
- Group chat history for new members

Social Features

- Emoji reactions to messages
- Custom coffee-themed stickers
- Message threads for organized discussions
- @mention notifications in groups
- Chat themes matching Caffis design system
- Message scheduling (send later)

Privacy & Security

- End-to-end encryption for private conversations
- Message self-destruction timers
- Block/unblock users
- Report inappropriate content
- Data export for user privacy compliance
- GDPR-compliant data deletion

Coffee Meetup Integration

Contextual Features

- Chat rooms automatically include meetup location

- Shared coffee shop details and photos
- Check-in confirmation when users arrive
- Review prompts after meetup completion
- Rescheduling tools within chat
- Weather updates for outdoor meetups

Event Coordination

- Multiple invitation management
- RSVP tracking within chat
- Shared expense tracking (coffee costs)
- Group photo sharing post-meetup
- Follow-up meeting suggestions
- Feedback collection and ratings

---

7. TECHNICAL ARCHITECTURE

C++ Backend Architecture

WebSocket Server Design

- Boost.Beast for high-performance WebSocket handling
- Asynchronous I/O with thread pool management
- Connection pooling for database operations
- Memory-efficient message queuing
- Horizontal scaling via Redis pub/sub

Database Integration

- libpqxx for native PostgreSQL connectivity
- Connection pooling (50 concurrent connections)
- Prepared statements for security and performance
- Transaction management for data consistency
- Automatic retry logic for connection failures

Performance Optimizations

- Message compression for large payloads

- Binary protocol for internal communication
- Lazy loading for chat history
- Intelligent caching strategies
- Background cleanup processes

Frontend Architecture

React Application Design

- TypeScript for type safety
- Context API for state management
- Custom hooks for WebSocket management
- Component-based architecture matching existing apps
- Responsive design for mobile compatibility

Real-time Integration

- WebSocket client with automatic reconnection
- Optimistic UI updates for better UX
- Message queue for offline/online synchronization
- Push notification integration (future)
- Service worker for background processing

UI/UX Consistency

- Exact replication of existing Apple design system
- CSS variables matching main app and map service
- Framer Motion animations for smooth transitions
- Accessibility compliance (WCAG 2.1)
- Dark mode support preparation

---

8. DATABASE DESIGN

New Chat Tables

chat_rooms

- Primary room entity with metadata

- Links to existing invitation system
- Support for different room types
- Activity tracking for efficient loading
- Soft deletion for "break up" feature

chat_participants

- Many-to-many relationship management
- Role-based permissions (admin/member)
- Join/leave timestamp tracking
- Last read message tracking for unread counts
- Active status for clean user removal

messages

- Core message storage with full history
- Support for different message types
- Edit and deletion tracking
- Reply/thread relationship management
- Partitioning strategy for performance

message_read_status

- Individual read receipt tracking
- Efficient unread count calculations
- Privacy-compliant read confirmation
- Batch update optimization
- Index optimization for performance

user_relationships

- "Break up" functionality implementation
- Block/unblock user management
- Relationship status tracking
- Privacy boundary enforcement
- Clean data removal workflows

Database Performance Strategy

Indexing Strategy

- Composite indexes for common query patterns
- Partial indexes for active records only
- Covering indexes to avoid table lookups
- Regular index maintenance and optimization

Partitioning Plan

- Messages table partitioned by date
- Automatic partition creation and cleanup
- Archive strategy for old conversations
- Performance monitoring and adjustment

Scaling Considerations

- Read replica setup for chat history queries
- Write optimization for real-time messages
- Connection pool sizing and monitoring
- Query performance analysis and optimization

---

9. MICROSERVICE STRUCTURE

Directory Structure

```
chat-service/
├── backend/            # C++ WebSocket Server
│   ├── src/
│   │   ├── main.cpp       # Server entry point
│   │   ├── websocket_server.cpp
│   │   ├── message_handler.cpp
│   │   ├── room_manager.cpp
│   │   ├── database_manager.cpp
│   │   ├── auth_validator.cpp
│   │   ├── file_handler.cpp
│   │   └── redis_client.cpp
│   ├── include/
│   │   ├── websocket_server.h
│   │   ├── message_types.h
│   │   ├── database_models.h
│   │   └── config.h
│   ├── tests/
│   │   ├── unit_tests/
│   │   ├── integration_tests/
│   │   └── performance_tests/
│   ├── docker/
│   │   ├── Dockerfile
│   │   └── entrypoint.sh
│   ├── CMakeLists.txt
│   └── README.md
├── frontend/           # React Chat Interface
│   ├── src/
│   │   ├── components/
│   │   │   ├── ChatRoom/
│   │   │   ├── ChatList/
│   │   │   ├── MessageBubble/
│   │   │   ├── EmojiPicker/
│   │   │   ├── FileUpload/
│   │   │   └── UserStatus/
│   │   ├── hooks/
│   │   │   ├── useWebSocket.ts
│   │   │   ├── useChat.ts
│   │   │   └── useFileUpload.ts
│   │   ├── contexts/
│   │   │   ├── ChatContext.tsx
│   │   │   └── WebSocketContext.tsx
│   │   ├── services/
│   │   │   ├── chatAPI.ts
│   │   │   └── websocketService.ts
│   │   ├── types/
│   │   │   └── chat.types.ts
│   │   └── styles/
│   │       └── chat.css
│   ├── public/
│   ├── package.json
│   └── README.md
├── database/
│   ├── migrations/
│   ├── seeds/
│   └── schema.sql
├── docker-compose.yml
├── .env.example
└── README.md
```

Service Responsibilities

Backend Service (C++)

- WebSocket connection management
- Real-time message processing
- Database operations and caching
- User authentication validation
- File upload and processing
- Room creation and management

Frontend Service (React)

- User interface for chat interactions
- WebSocket client management
- State management and caching
- File upload interface
- Emoji and media handling
- Integration with existing design system

Database Service

- Chat-specific data storage
- Integration with existing user system

- Performance optimization
- Backup and recovery
- Data archiving and cleanup

---

10. IMPLEMENTATION PHASES

Phase 1: Foundation (Weeks 1-4)

Core Infrastructure Setup

- Project structure creation and organization
- Docker containers and development environment
- Basic C++ WebSocket server with Boost.Beast
- PostgreSQL integration with libpqxx
- Redis setup for message queuing
- Basic authentication validation

Deliverables:

- Working WebSocket server
- Database schema and migrations
- Basic message sending/receiving
- Docker composition integration
- Development environment documentation

Phase 2: Core Messaging (Weeks 5-8)

Essential Chat Features

- Real-time text messaging
- Message persistence and history loading
- User online/offline status
- Basic room creation and management
- Integration with existing authentication
- Message delivery confirmation

Deliverables:

- Functional 1-on-1 messaging
- Chat history with pagination
- Status indicators and typing notifications
- Room creation via Map Service webhooks
- Basic React frontend interface

Phase 3: Rich Features (Weeks 9-12)

Advanced Messaging Capabilities

- Image and file upload system
- Emoji support and reactions
- Message editing and deletion
- Group chat functionality
- Read receipts and message status
- Search within conversation history

Deliverables:

- Complete media sharing system
- Full emoji integration
- Group chat management
- Advanced message features
- Performance optimization

Phase 4: Integration & Polish (Weeks 13-16)

Service Integration and UX

- Perfect design system integration
- Map Service webhook integration
- "Break up" relationship management
- Push notification preparation
- Mobile responsiveness
- Performance testing and optimization

Deliverables:

- Seamless multi-service integration
- Complete Apple design system matching
- Relationship management system
- Production-ready performance
- Comprehensive testing suite

Phase 5: Production & Monitoring (Weeks 17-20)

Deployment and Operations

- Production deployment setup
- Monitoring and logging systems
- Performance analytics
- Security audit and hardening
- Load testing and scaling verification
- Documentation completion

Deliverables:

- Production-ready deployment
- Monitoring and alerting systems
- Performance benchmarks
- Security compliance verification
- Complete documentation package

---

11. TIMELINE & MILESTONES

Development Timeline

Month 1: Foundation (Weeks 1-4)

- Week 1: Project setup, environment configuration
- Week 2: C++ WebSocket server basic implementation
- Week 3: Database integration and schema creation
- Week 4: Redis integration and message queuing

Month 2: Core Features (Weeks 5-8)

- Week 5: Basic messaging functionality
- Week 6: Room management and user authentication
- Week 7: Frontend React application development
- Week 8: Map Service integration and webhooks

Month 3: Advanced Features (Weeks 9-12)

- Week 9: File upload and media sharing
- Week 10: Group chat and advanced messaging
- Week 11: Message management (edit/delete/search)
- Week 12: Emoji system and rich interactions

Month 4: Integration (Weeks 13-16)

- Week 13: Design system perfect matching
- Week 14: Cross-service integration testing
- Week 15: Relationship management features
- Week 16: Performance optimization and testing

Month 5: Production (Weeks 17-20)

- Week 17: Production deployment preparation
- Week 18: Security audit and monitoring setup
- Week 19: Load testing and scaling verification
- Week 20: Documentation and knowledge transfer

Key Milestones

- **Milestone 1 (Week 4)**: Basic WebSocket server operational
- **Milestone 2 (Week 8)**: Core messaging with Map Service integration
- **Milestone 3 (Week 12)**: Full-featured chat with media support
- **Milestone 4 (Week 16)**: Complete service integration
- **Milestone 5 (Week 20)**: Production deployment ready

---

12. RISK ASSESSMENT

Technical Risks

High-Impact Risks

- **C++ Learning Curve**: Mitigation through incremental development and pair programming
- **WebSocket Scaling**: Mitigation through Redis pub/sub and load testing
- **Database Performance**: Mitigation through proper indexing and query optimization
- **Integration Complexity**: Mitigation through API documentation and testing

Medium-Impact Risks

- **File Upload Security**: Mitigation through validation and virus scanning
- **Memory Management**: Mitigation through smart pointers and testing
- **Cross-browser Compatibility**: Mitigation through WebSocket polyfills
- **Mobile Performance**: Mitigation through responsive design testing

Low-Impact Risks

- **Design Consistency**: Mitigation through CSS variable reuse
- **Emoji Licensing**: Mitigation through open-source emoji sets
- **Deployment Complexity**: Mitigation through Docker containerization

Business Risks

User Adoption Risks

- Risk: Users may not engage with chat feature
- Mitigation: Seamless integration with existing user flows
- Mitigation: Contextual chat creation from successful invitations

Performance Risks

- Risk: Poor performance may degrade user experience
- Mitigation: Extensive performance testing and monitoring
- Mitigation: Gradual rollout with performance metrics tracking

Security Risks

- Risk: Chat data security and privacy concerns
- Mitigation: End-to-end encryption implementation
- Mitigation: Comprehensive security audit before production

Mitigation Strategies

Development Approach

- Incremental development with frequent testing
- Code reviews and pair programming for C++ components
- Performance benchmarking at each phase
- Security-first design principles

Quality Assurance

- Automated testing suite for all components
- Load testing with realistic user scenarios
- Security penetration testing
- User acceptance testing with real user feedback

---

13. SUCCESS METRICS

Performance Metrics

Technical Performance

- Message delivery latency: Target <50ms average
- WebSocket connection capacity: Target 10,000+ concurrent
- Database query response time: Target <100ms average
- File upload speed: Target 5MB/minute minimum
- System uptime: Target 99.9% availability

User Experience Metrics

- Chat room creation success rate: Target >95%
- Message delivery success rate: Target >99.5%
- Average response time for user actions: Target <200ms
- File upload success rate: Target >98%
- Cross-browser compatibility: Target 100% major browsers

Business Metrics

User Engagement

- Chat activation rate from accepted invitations: Target >80%
- Daily active chat users: Target 60% of total daily users
- Average messages per conversation: Target >20
- Return user rate for chat feature: Target >70%
- Group chat creation rate: Target >30% of conversations

System Utilization

- Server resource utilization: Target <70% during peak
- Database connection efficiency: Target >90%
- Redis cache hit rate: Target >95%
- Storage utilization growth rate: Monitor and plan
- Network bandwidth utilization: Monitor and optimize

Quality Metrics

Code Quality

- Unit test coverage: Target >90%
- Integration test coverage: Target >80%
- Code review completion: Target 100%
- Security vulnerability count: Target 0 high-severity
- Documentation completeness: Target 100% public APIs

User Satisfaction

- User-reported bug rate: Target <1% of interactions
- Feature request fulfillment rate: Target >80%
- User satisfaction score: Target >4.5/5
- Support ticket volume: Target <5% of users
- User retention after chat usage: Target >85%

---

14. DEPLOYMENT STRATEGY

Development Environment

Local Development Setup

- Docker Compose for complete environment
- Hot reload for both frontend and backend
- Database seeding with test data
- Redis local instance for development
- SSL certificates for WebSocket testing

Development Workflow

- Git-based version control with feature branches
- Automated testing on pull requests
- Code review requirements for all changes
- Staging environment for integration testing
- Performance profiling in development

Staging Environment

Pre-Production Testing

- Complete replica of production environment
- Automated deployment from develop branch
- Performance testing with realistic data volumes
- Security testing and vulnerability scanning
- User acceptance testing with beta users

Integration Testing

- Full service integration testing
- Load testing with simulated user traffic
- Database migration testing
- Backup and recovery testing
- Monitoring and alerting verification

Production Deployment

Deployment Strategy

- Blue-green deployment for zero downtime
- Database migration with rollback capability

- Gradual traffic routing to new services
- Real-time monitoring during deployment
- Immediate rollback capability if issues arise

Production Environment

- Container orchestration with health checks
- Load balancing for high availability
- Database clustering for performance
- Redis cluster for message scaling
- CDN integration for file uploads

Monitoring and Operations

- Real-time performance monitoring
- Error tracking and alerting systems
- Log aggregation and analysis
- Automated backup systems
- Security monitoring and threat detection

Scaling Considerations

Horizontal Scaling

- WebSocket server scaling via load balancer
- Database read replicas for chat history
- Redis cluster for message distribution
- CDN scaling for file uploads
- Container auto-scaling based on load

Performance Optimization

- Database query optimization and indexing
- Message compression for large payloads
- Intelligent caching strategies
- Background processing for non-critical tasks
- Regular performance audits and improvements

CONCLUSION

This comprehensive implementation plan outlines the complete development of a production-grade chat service that will seamlessly integrate with the existing Caffis ecosystem. The service will provide enterprise-level performance while maintaining the beautiful Apple design system consistency that defines the Caffis user experience.

The phased approach ensures steady progress with measurable milestones, while the C++ backend provides the performance foundation necessary for real-time messaging at scale. The detailed risk assessment and mitigation strategies address potential challenges proactively.

Upon completion, the chat service will not only enhance user engagement within the coffee meetup context but also provide a solid foundation for future messaging enhancements and premium features.

Next Steps:

1. Review and approve this implementation plan
2. Set up development environment and project structure
3. Begin Phase 1 implementation with foundation setup
4. Establish regular progress reviews and milestone checkpoints

This plan serves as the definitive guide for creating a world-class chat service that will elevate the Caffis platform to new levels of social engagement and user satisfaction.