Functional Analysis Document (FAD)

Table of Contents

ChatFlow MVP - Team Communication Platform

VERSIONE 2.0 - LAB OPTIMIZED

Document Metadata

Attributo	Valore
Document Type	Functional Analysis Document (FAD)
Project	ChatFlow MVP - Real-Time Team Communication
Version	2.0 (Lab-Optimized)
Date	November 19, 2025
Status	Final for Development
Author	Senior Software Architect
Lab Environment	slackteam.lab.home.lucasacchi.net
Node.js	v24.11.1 (LTS)
Python	3.11.2
Target Audience	Development team, QA engineers, architects
Template Base	joelparkerhenderson/functional-specifications-template

Table of Contents

1. Executive Overview

Purpose

This FAD translates the ChatFlow MVP PRD into detailed functional specifications for implementation. It provides developers, QA, and architects with a complete blueprint for building the system.

Scope

- **Core Features:** Authentication, workspace/channel management, real-time messaging, search, notifications, file sharing
- **Target Users:** 5-200 concurrent, 50-100 DAU (lab MVP)
- **Message History:** 48-hour window (MVP)
- **Performance:** <500ms message latency (p99), 99.5% uptime
- **Technology:** Node.js 24.11.1, React 19, PostgreSQL 15, Redis

Key Deliverables

✅ Feature specifications with business logic
✅ Data flow diagrams (3 levels)
✅ Sequence diagrams for critical flows
✅ Complete SQL schema
✅ Unit test examples (Jest)
✅ API endpoint specifications
✅ Performance benchmarks
✅ Deployment procedures

2. Functional Architecture Overview

2.1 System Boundary & Context

```
External Systems:
  └─ SendGrid (email)
  └─ OAuth Providers (Google, GitHub)
  └─ AWS S3 or Local Storage (files)
  └─ Analytics (Mixpanel)

ChatFlow System Boundary:
  ├─ User Interface Layer (React 19 + TypeScript)
  ├─ API Gateway &amp; Real-Time Layer (Express.js + Socket.IO)
  ├─ Functional Services (8 core modules)
  ├─ Data Services (User, Channel, Message stores)
  └─ Persistence &amp; Infrastructure (PostgreSQL, Redis)
```

2.2 Core Functional Modules (FBS Level 0→1)

```
ChatFlow System (Level 0)
├─ Module 1: Authentication &amp; Identity Management
│  ├─ User registration + email verification
│  ├─ Login/logout + session management
│  ├─ JWT token generation/validation
│  └─ OAuth integration (v1.1)
│
├─ Module 2: Workspace Management
│  ├─ Create/read/update workspace
│  ├─ Member management (invite, remove, promote)
│  ├─ Workspace settings + plan management
│  └─ Audit logging
│
├─ Module 3: Channel Management
│  ├─ Create/read/update/delete channels
│  ├─ Channel membership management
│  ├─ Public/private access control
│  ├─ Archive/restore channels
│   └─ Channel permissions matrix
│
├─ Module 4: Message Management &amp; Real-Time
│   ├─ Send/receive messages (WebSocket)
│   ├─ Edit messages (1-hour window)
│   ├─ Delete messages (soft delete)
│   ├─ Thread replies
│   ├─ Emoji reactions
│   └─ Message persistence + indexing
│
├─ Module 5: Direct Messaging
│   ├─ 1-on-1 conversations
│   ├─ Group DM (3+ users)
│   ├─ Typing indicators
│   ├─ Online/offline status
│   └─ DM notification preferences
│
├─ Module 6: Search &amp; Discovery
│   ├─ Full-text message search
│   ├─ Advanced filters (from:, in:, date:)
│   ├─ Elasticsearch integration (optional)
│   ├─ Result ranking + pagination
│   └─ Permission-based filtering
│
├─ Module 7: Notifications &amp; Presence
│   ├─ In-app toast notifications
│   ├─ Browser push notifications
│   ├─ @mention detection
│   ├─ User presence broadcast
│   └─ Notification preferences per channel
│
└─ Module 8: File Management
    ├─ File upload/download
    ├─ File metadata storage
    ├─ Image preview generation
    ├─ Storage backend (S3 or local)
    └─ Virus scanning (optional async)
```

3. System Context & Boundaries

3.1 Actors (Users)

Actor	Role	Interactions
End User	Team member	Send/receive messages, manage profile
Workspace Owner	Admin	Create workspace, manage members, settings
Moderator	Channel admin	Create channels, manage members, delete messages
System Administrator	Ops/IT	Deploy, monitor, backup, audit logs
External System	Integration	SendGrid, OAuth, S3, Analytics

3.2 System Boundaries

```
Boundary 1: User Authentication
├─ Internal: JWT validation, password hashing
└─ External: OAuth providers (optional), Email service

Boundary 2: Workspace &amp; Channel Management
├─ Internal: ACL enforcement, role management
└─ External: Audit logging, compliance tracking

Boundary 3: Real-Time Messaging
├─ Internal: WebSocket server, message persistence
```

```
└─ External: Notification service, search indexing

Boundary 4: Data Storage
├─ Internal: PostgreSQL, Redis cache
└─ External: S3/Local file storage, backup service
```

4. Functional Decomposition (FBS - Hierarchical)

4.1 Feature Breakdown Structure

```
ChatFlow MVP (Root)
│
├─ Authentication &amp; User Management (FEAT-100)
│  ├─ FEAT-101: User Registration
│  │  ├─ Input validation (email, password strength)
│  │  ├─ Email verification workflow
│  │  ├─ Account creation + activation
│  │  └─ Error handling (duplicate email, invalid format)
│  │
│  ├─ FEAT-102: User Login
│  │  ├─ Credential validation
│  │  ├─ JWT token generation
│  │  ├─ Session management
│  │  ├─ Rate limiting (5 attempts → 15min lockout)
│  │  └─ Refresh token handling
│  │
│  ├─ FEAT-103: User Profile Management
│  │  ├─ Avatar upload/crop
│  │  ├─ Display name + bio editing
│  │  ├─ Timezone configuration
│  │  ├─ Status management (online/away/offline)
│  │  └─ Notification preferences
│  │
│  └─ FEAT-104: OAuth Integration (v1.1)
│     ├─ Google OAuth flow
│     ├─ GitHub OAuth flow
│     ├─ Social account linking
│     └─ Fallback to email signup
│
├─ Workspace Management (FEAT-200)
│  ├─ FEAT-201: Create Workspace
│  │  ├─ Workspace name validation
│  │  ├─ Unique slug generation
│  │  ├─ Default channel creation (#general, #random, #announcements)
│  │  ├─ Creator becomes owner
│  │  └─ Initial plan assignment (free)
│  │
│  ├─ FEAT-202: Invite Team Members
│  │  ├─ Email validation
│  │  ├─ Invite link generation (7-day TTL)
│  │  ├─ Bulk invitation (up to 50)
│  │  ├─ Auto-join for registered users
│  │  ├─ Redirect to signup for new users
│  │  └─ Invite status tracking
│  │
│  ├─ FEAT-203: Member Management
│  │  ├─ Role assignment (owner, admin, moderator, member)
│  │  ├─ Member removal (immediate ACL revocation)
│  │  ├─ Member list with roles
│  │  ├─ Promotion/demotion workflows
│  │  └─ Audit logging per action
│  │
│  ├─ FEAT-204: Workspace Settings
│  │  ├─ Plan upgrade/downgrade (free → pro → enterprise)
│  │  ├─ Member limit enforcement
│  │  ├─ Feature enablement/disablement
│  │  ├─ Branding customization (v1.1)
│  │  └─ Export settings
│  │
│  └─ FEAT-205: Workspace Deletion
│     ├─ Owner-only action
│     ├─ Soft-delete + 30-day recovery window
│     ├─ Data archival notification
│     └─ Member notification + access revocation
│
├─ Channel Management (FEAT-300)
│  ├─ FEAT-301: Create Channel
│  │  ├─ Public/private type selection
│  │  ├─ Channel name (3-50 chars, unique per workspace)
│  │  ├─ Description (optional, max 200 chars)
│  │  ├─ Slug generation + collision handling
│  │  ├─ Creator becomes moderator
│  │  └─ Auto-add all members to public channels
│  │
│  ├─ FEAT-302: Join/Leave Channel
│  │  ├─ Public channel auto-join
│  │  ├─ Private channel invite-only
│  │  ├─ Leave functionality (except #general mandatory)
│  │  ├─ Re-join support
│  │  └─ Notification on join (optional)
│  │
│  ├─ FEAT-303: Channel Metadata
│  │  ├─ Topic/description update
│  │  ├─ Member list + roles
│  │  ├─ Message count tracking
│  │  ├─ Creation date + creator info
│  │  └─ Last activity timestamp
│  │
│  ├─ FEAT-304: Channel Permissions
│  │  ├─ Role-based access matrix
│  │  ├─ Moderator: manage members, delete messages
│  │  ├─ Member: send messages, basic reactions
│  │  └─ Audit per action
│  │
│  ├─ FEAT-305: Archive/Delete Channel
│  │  ├─ Archive: read-only, hidden from list
│  │  ├─ Soft-delete: message history preserved
│  │  ├─ Owner/admin only
│  │  └─ Audit logging + notification
│  │
│  └─ FEAT-306: Channel Discovery
│     ├─ Browse public channels
│     ├─ Search by name/description
│     ├─ Activity level indicator
│     ├─ Member count display
│     └─ Sorting (alphabetical, most active, recent)
│
├─ Message Management (FEAT-400)
│  ├─ FEAT-401: Send Message
│  │  ├─ Content validation (not empty, &lt;4KB)
│  │  ├─ Markdown parsing (bold, italic, code)
│  │  ├─ @mention extraction + notification
│  │  ├─ Database persistence
│  │  ├─ WebSocket broadcast (&lt;500ms)
│  │  ├─ Search indexing (async, &lt;5s lag)
│  │  ├─ Edit history initialization
│  │  └─ Delivery confirmation
│  │
│  ├─ FEAT-402: Edit Message
│  │  ├─ 1-hour edit window enforcement
│  │  ├─ Author-only permission check
│  │  ├─ Content re-validation
│  │  ├─ Edit history append
│  │  ├─ "Edited" label + timestamp
│  │  ├─ WebSocket broadcast to all members
│  │  └─ Re-index in search
│  │
│  ├─ FEAT-403: Delete Message
│  │  ├─ Soft-delete (marked but retained)
│  │  ├─ Author or moderator can delete
│   │   ├─ "Message deleted" placeholder shown
│   │   ├─ WebSocket broadcast
│   │   └─ Audit logging
│   │
│   ├─ FEAT-404: Message Reactions
│   │   ├─ Emoji picker UI
│   │   ├─ Add/remove reaction
│   │   ├─ Reaction count tracking
│   │   ├─ Multiple users same emoji
│   │   ├─ WebSocket real-time update
│   │   └─ 20+ emoji support (MVP)
│   │
│   ├─ FEAT-405: Threading/Replies
│   │   ├─ Reply to specific message (thread_id)
│   │   ├─ Thread display nested under parent
│   │   ├─ Unread thread count indicator
│   │   └─ Notification on thread reply
│   │
│   ├─ FEAT-406: Message Persistence
│   │   ├─ PostgreSQL storage
│   │   ├─ Indexed for query performance
│   │   ├─ 48-hour history window (MVP)
│   │   ├─ Timestamp: server-generated UTC
│   │   └─ Message ID: immutable UUID
│   │
│   └─ FEAT-407: Typing Indicators
│       ├─ Broadcast "user is typing"
│       ├─ Auto-clear after 3s inactivity
│       ├─ Real-time update &lt;100ms
│       └─ Optional disable per user
│
├─ Direct Messaging (FEAT-500)
│   ├─ FEAT-501: Create 1-on-1 DM
│   │   ├─ User selection from workspace members
│   │   ├─ DM thread creation
│   │   ├─ Message history persistence
│   │   ├─ Online status indicator
│   │   └─ Last message preview
│   │
│   ├─ FEAT-502: Create Group DM
│   │   ├─ Select 3+ participants
│   │   ├─ Group naming (optional)
│   │   ├─ Shared conversation history
│   │   ├─ Leave group (archive for user)
│   │   └─ Add/remove members (admin)
│   │
│   ├─ FEAT-503: DM Features
│   │   ├─ Same messaging features as channels
│   │   ├─ Edit/delete/reactions
│   │   ├─ File sharing
│   │   ├─ Typing indicators
│   │   └─ Mention support
│   │
│   └─ FEAT-504: DM Notifications
│       ├─ New DM alert
│       ├─ Typing notification
│       ├─ Customizable per user
│       └─ Mute DM option
│
├─ Search &amp; Discovery (FEAT-600)
│   ├─ FEAT-601: Full-Text Search
│   │   ├─ Elasticsearch indexing (optional)
│   │   ├─ Keyword search across messages
│   │   ├─ Results &lt;2 seconds (p95)
│   │   ├─ 20 results per page
│   │   ├─ Relevance ranking
│   │   └─ Permission-based filtering
│   │
│   ├─ FEAT-602: Advanced Filters
│   │   ├─ from:@username (filter by author)
│   │   ├─ in:#channel (filter by channel)
│   │   ├─ before:YYYY-MM-DD (date range)
│  │  ├─ after:YYYY-MM-DD (date range)
│  │  ├─ "exact phrase" (phrase match)
│  │  └─ Combine filters (AND logic)
│  │
│  ├─ FEAT-603: Search Results Display
│  │  ├─ Author + avatar
│  │  ├─ Channel name + link
│  │  ├─ Timestamp
│  │  ├─ Message snippet (100 chars)
│  │  ├─ Highlight matching keywords
│  │  └─ Click → navigate to message
│  │
│  └─ FEAT-604: Message History
│     ├─ Retrieve messages (paginated, 50 per page)
│     ├─ Scroll up to load older
│     ├─ 48-hour default window
│     └─ Timestamp + author info
│
├─ Notifications &amp; Presence (FEAT-700)
│  ├─ FEAT-701: In-App Notifications
│  │  ├─ Toast notifications (5s auto-dismiss)
│  │  ├─ @mention detection + trigger
│  │  ├─ Channel notification preferences
│  │  ├─ DM new message alert
│  │  ├─ Unread message badge (count)
│  │  └─ Click → navigate to message
│  │
│  ├─ FEAT-702: Browser Notifications
│  │  ├─ Request user permission (first use)
│  │  ├─ Send push notification for @mentions
│  │  ├─ DM new message notification
│  │  ├─ Enable/disable per user
│  │  └─ Custom notification sound (optional)
│  │
│  ├─ FEAT-703: User Presence
│  │  ├─ Status types: online, away, offline, do not disturb
│  │  ├─ Auto-detect inactivity (15min → away)
│  │  ├─ Broadcast status change &lt;500ms
│  │  ├─ Last seen timestamp
│  │  └─ User list indicator (green dot = online)
│  │
│  ├─ FEAT-704: Notification Preferences
│  │  ├─ Global mute/unmute
│  │  ├─ Per-channel mute
│  │  ├─ Per-DM mute
│  │  ├─ Notification time window (quiet hours)
│  │  ├─ Sound preferences
│  │  └─ Desktop vs mobile settings
│  │
│  └─ FEAT-705: Email Notifications (v1.1)
│     ├─ Daily digest option
│     ├─ @mention emails
│     ├─ DM notification emails
│     └─ Unsubscribe mechanism
│
└─ File Management (FEAT-800)
   ├─ FEAT-801: File Upload
   │  ├─ Max 10MB per file
   │  ├─ Virus scanning (async)
   │  ├─ File metadata storage
   │  ├─ Storage backend (S3 or local /tmp)
   │  ├─ Uploaded timestamp + uploader
   │  └─ Progress indication
   │
   ├─ FEAT-802: File Download
   │  ├─ Direct download link
   │  ├─ CDN delivery (optional)
   │  ├─ Access control (only channel/DM members)
   │  └─ Logging + audit trail
   │
   ├─ FEAT-803: File Preview
   │  ├─ Image inline preview (JPG, PNG, GIF)
```

```
    │   ├─ Document icons (PDF, DOC, XLS)
    │   ├─ File size display
    │   ├─ Download button
    │   └─ Delete button (uploader)
    │
    └─ FEAT-804: File Deletion
       ├─ Uploader can delete
       ├─ Moderator can delete
       ├─ Permanent deletion from storage
       └─ Audit logging
```

5. Detailed Feature Specifications (by Module)

Module 1: Authentication & User Management (FEAT-100)

FEAT-101: User Registration

Business Logic:

```
// Pseudocode for signup process
async function registerUser(email, password, displayName) {
  // Step 1: Validate inputs
  if (!isValidEmail(email)) {
    return Error(400, "Invalid email format");
  }
  if (!isStrongPassword(password)) {
    // Min 8 chars, 1 uppercase, 1 number, 1 special char
    return Error(400, "Password does not meet requirements");
  }
  if (!displayName || displayName.length &lt; 2 || displayName.length &gt; 100) {
    return Error(400, "Display name must be 2-100 characters");
  }

  // Step 2: Check duplicate email (case-insensitive)
  const existingUser = await database.query(
    `SELECT id FROM users WHERE LOWER(email) = LOWER(?)`,
    [email]
  );
  if (existingUser) {
    return Error(409, "Email already registered");
  }

  // Step 3: Hash password
  const passwordHash = await bcrypt.hash(password, 12); // cost factor 12

  // Step 4: Create user (unverified)
  const userId = generateUUID();
  await database.query(
    `INSERT INTO users
     (id, email, password_hash, display_name, status, email_verified, created_at)
     VALUES (?, ?, ?, ?, 'offline', false, NOW())`,
    [userId, email, passwordHash, displayName]
  );

  // Step 5: Generate verification token (24-hour TTL)
  const verificationToken = jwt.sign(
    { email, type: "email_verify", exp: Date.now() + 24 * 3600 * 1000 },
    SECRET_KEY
  );

  // Step 6: Send verification email
  await emailService.send({
    to: email,
    template: "email_verification",
    data: {
      verificationUrl: `${FRONTEND_URL}/auth/verify?token=${verificationToken}`,
      displayName,
      expiryHours: 24
    }
  });

  // Step 7: Log event
  analytics.trackEvent("user_signup", { userId, email, timestamp: now() });

  return Success({ message: "Confirmation email sent", userId });
}
```

Database Mutation:

```
-- Insert new user record
INSERT INTO users
(id, email, password_hash, display_name, avatar_url, bio, timezone,
 status, email_verified, created_at, updated_at, last_login)
VALUES
('uuid-new-user', 'user@example.com', '$2b$12$...bcrypt-hash...',
 'John Doe', NULL, NULL, 'UTC', 'offline', false, now(), now(), NULL);

-- Audit log
INSERT INTO audit_logs
(id, workspace_id, actor_id, action, resource_type, resource_id, details, created_at)
VALUES
('uuid-log', NULL, 'uuid-new-user', 'user_signup', 'user', 'uuid-new-user',
 '{"method":"email","ip":"192.168.1.1"}', now());
```

Error Handling Matrix:

Error	HTTP Code	Message	Recovery
Invalid email format	400	"Invalid email format (e.g., user@example.com)"	Show validation error
Email already exists	409	"Email already registered. Try login."	Link to login
Weak password	400	"Password must be 8+ chars, 1 uppercase, 1 number, 1 special char"	Show strength meter
Display name too short	400	"Display name must be 2-100 characters"	Highlight field
Database error	500	"Registration failed. Try again later."	Retry with exponential backoff
Email delivery failed	500	"Verification email not sent. Resend?"	Allow manual resend (max 5x/hour)

Acceptance Criteria (QA):

- [ ] Valid email + strong password → Account created (unverified)
- [ ] Email verification link sent within 5 seconds
- [ ] Link valid for exactly 24 hours
- [ ] Clicking link → Account activated → Auto-login
- [ ] Duplicate email rejected with 409 error
- [ ] Password validation enforces all rules (8 chars, 1 upper, 1 number, 1 special)
- [ ] XSS prevention: HTML entities in display name
- [ ] SQL injection prevention: parameterized queries
- [ ] Rate limiting: Max 5 signup attempts per IP per minute

- [ ] Load test: 100 concurrent signups <2 seconds each

Performance Targets (Lab VM):

```
Signup form validation:  &lt;100ms (client-side)
Database insert:         &lt;200ms
Email send:              &lt;5s (async)
Total flow:              &lt;500ms (blocking part)
```

FEAT-102: User Login

Business Logic:

```
async function authenticateUser(email, password) {
  // Step 1: Rate limiting check
  const failedAttempts = await cache.get(`login_fails:${email}`) || 0;
  if (failedAttempts &gt;= 5) {
    const lockoutTime = await cache.get(`lockout:${email}`);
    if (lockoutTime &amp;&amp; lockoutTime &gt; Date.now()) {
      return Error(429, `Account locked. Try again at ${lockoutTime}`);
    }
  }

  // Step 2: Fetch user
  const user = await database.query(
    `SELECT * FROM users WHERE LOWER(email) = LOWER(?)`,
    [email]
  );
  if (!user) {
    // Generic error (don't leak email existence)
    return Error(401, "Invalid email or password");
  }

  // Step 3: Check email verified
  if (!user.email_verified) {
    return Error(403, "Please verify your email first. Resend link?");
  }

  // Step 4: Verify password
  const isPasswordValid = await bcrypt.compare(password, user.password_hash);
  if (!isPasswordValid) {
    // Increment failed attempts
    await cache.incr(`login_fails:${email}`);
    if (failedAttempts + 1 &gt;= 5) {
      await cache.set(`lockout:${email}`, Date.now() + 15 * 60 * 1000, 900); // 15 min
    }
    return Error(401, "Invalid email or password");
  }

  // Step 5: Clear failed attempts on success
  await cache.del(`login_fails:${email}`);
  await cache.del(`lockout:${email}`);

  // Step 6: Fetch user workspaces
  const workspaces = await database.query(
    `SELECT w.* FROM workspaces w
     INNER JOIN user_workspace_members uwm ON w.id = uwm.workspace_id
     WHERE uwm.user_id = ? AND uwm.status = 'active'`,
    [user.id]
  );

  // Step 7: Generate JWT tokens
  const accessToken = jwt.sign(
    {
      user_id: user.id,
      email: user.email,
      display_name: user.display_name,
      workspace_ids: workspaces.map(w =&gt; w.id),
      type: "access_token",
      exp: Date.now() + 24 * 3600 * 1000  // 24h
    },
    SECRET_KEY,
    { algorithm: "HS256" }
  );

  const refreshToken = jwt.sign(
    {
      user_id: user.id,
      type: "refresh_token",
      exp: Date.now() + 30 * 24 * 3600 * 1000  // 30 days
    },
    REFRESH_SECRET_KEY,
    { algorithm: "HS256" }
  );

  // Step 8: Store in cache for quick retrieval
  await cache.set(`session:${user.id}`, {
    user_id: user.id,
    email: user.email,
    workspaces: workspaces
  }, 24 * 3600);  // 24h TTL

  // Step 9: Update last_login
  await database.query(
    `UPDATE users SET last_login = NOW() WHERE id = ?`,
    [user.id]
  );

  // Step 10: Log login event
  analytics.trackEvent("user_login", {
    user_id: user.id,
    workspace_count: workspaces.length,
    timestamp: now()
  });

  return Success({
    access_token: accessToken,
    refresh_token: refreshToken,
    expires_in: 86400,  // seconds
    user: {
      id: user.id,
      email: user.email,
      display_name: user.display_name,
      avatar_url: user.avatar_url
    },
    workspaces: workspaces
  });
}
```

JWT Token Structure:

```
{
  "header": {
    "alg": "HS256",
    "typ": "JWT"
  },
  "payload": {
    "user_id": "550e8400-e29b-41d4-a716-446655440000",
    "email": "user@example.com",
    "display_name": "John Doe",
    "workspace_ids": ["ws-001", "ws-002"],
    "type": "access_token",
    "iat": 1700425200,
    "exp": 1700511600
  },
  "signature": "HMACSHA256(base64UrlEncode(header) + '.' + base64UrlEncode(payload), secret)"
}
```

Middleware - Token Validation:

```
// Express middleware to validate JWT
function authenticateToken(req, res, next) {
  const authHeader = req.headers['authorization'];
  const token = authHeader &amp;&amp; authHeader.split(' ')[1];  // Bearer &lt;token&gt;

  if (!token) {
    return res.status(401).json({ error: "Missing authorization token" });
  }

  jwt.verify(token, SECRET_KEY, (err, decoded) =&gt; {
    if (err) {
      if (err.name === 'TokenExpiredError') {
        return res.status(401).json({ error: "Token expired. Please refresh." });
      }
      return res.status(403).json({ error: "Invalid token" });
    }

    // Attach user to request for downstream handlers
    req.user = decoded;
    next();
  });
}

// Usage in route
app.get('/api/messages', authenticateToken, (req, res) =&gt; {
  const userId = req.user.user_id;
  // ... fetch messages for user
});
```

Acceptance Criteria (QA):

- [ ] Correct email + password → JWT tokens issued
- [ ] Access token valid for 24 hours
- [ ] Refresh token valid for 30 days
- [ ] Incorrect password → 401 error (no email leak)
- [ ] 5 failed attempts → Account locked 15 minutes
- [ ] Lockout shows countdown timer
- [ ] Email not verified → 403 error with resend link
- [ ] Multiple workspace users → Show workspace selector
- [ ] Last login timestamp updated
- [ ] Login attempt logged to audit trail

Performance Targets (Lab VM):

```
Authentication check:   &lt;50ms (JWT validation)
Password verification:  &lt;100ms (bcrypt cost 12)
Database lookup:        &lt;30ms (indexed on email)
Token generation:       &lt;20ms (JWT signing)
Total login:            &lt;200ms (p95)
```

Module 4: Message Management & Real-Time (FEAT-400)

FEAT-401: Send Message (Core Real-Time)

Business Logic - Step-by-Step:

```
async function sendMessage(userId, channelId, content, threadId = null) {
  try {
    // Step 1: Validate permission (user member of channel)
    const channelMember = await database.query(
      `SELECT * FROM channel_members
```

```
     WHERE channel_id = ? AND user_id = ?`,
    [channelId, userId]
  );

  if (!channelMember) {
    throw Error(403, "No access to this channel");
  }

  // Step 2: Validate content
  if (!content || content.trim().length === 0) {
    throw Error(400, "Message cannot be empty");
  }

  if (content.length &gt; 4000) {
    throw Error(400, "Message exceeds 4000 character limit");
  }

  // Step 3: Content sanitization (prevent XSS)
  // Preserve Markdown but escape HTML
  const sanitizedContent = sanitizeMarkdown(content);
  // Example: &lt;script&gt;alert('xss')&lt;/script&gt; → &lt;script&gt;alert('xss')&lt;/script&gt;

  // Step 4: Parse mentions (@username)
  const mentionMatches = content.match(/@(\w+)/g) || [];
  const mentions = [];

  for (const mention of mentionMatches) {
    const username = mention.substring(1);  // Remove @
    const mentionedUser = await database.query(
      `SELECT id FROM users WHERE display_name = ?`,
      [username]
    );

    if (mentionedUser) {
      mentions.push({
        userId: mentionedUser.id,
        username,
        fullMention: mention
      });
    }
  }

  // Step 5: Create message record
  const messageId = generateUUID();
  const now = new Date().toISOString();

  await database.query(
    `INSERT INTO messages
     (id, channel_id, user_id, content, thread_id, created_at, updated_at)
     VALUES (?, ?, ?, ?, ?, ?, ?)`,
    [messageId, channelId, userId, sanitizedContent, threadId, now, now]
  );

  // Step 6: Update channel message counter
  await database.query(
    `UPDATE channels SET message_count = message_count + 1 WHERE id = ?`,
    [channelId]
  );

  // Step 7: Index for search (async, non-blocking)
  indexMessage({
    messageId,
    channelId,
    userId,
    content: sanitizedContent,
    createdAt: now,
    workspaceId: (fetch from channel)
  }).catch(err =&gt; logger.error("Search indexing failed:", err));

  // Step 8: Handle mentions (create notifications)
  for (const mention of mentions) {
    // Create notification
```

```
      await database.query(
        `INSERT INTO notifications
         (id, user_id, type, channel_id, message_id, actor_id, created_at, read)
         VALUES (?, ?, 'mention', ?, ?, ?, ?, false)`,
        [generateUUID(), mention.userId, channelId, messageId, userId, now]
      );

      // Send push notification (if enabled)
      const prefs = await cache.get(`notification_prefs:${mention.userId}`);
      if (prefs &amp;&amp; prefs.mentions) {
        pushNotificationService.send({
          userId: mention.userId,
          title: `${user.display_name} mentioned you`,
          body: sanitizedContent.substring(0, 100),
          deepLink: `/channels/${channelId}/messages/${messageId}`
        });
      }
    }

    // Step 9: Broadcast to all channel members via WebSocket
    const channelMembers = await database.query(
      `SELECT user_id FROM channel_members WHERE channel_id = ?`,
      [channelId]
    );

    const messagePayload = {
      id: messageId,
      channel_id: channelId,
      user_id: userId,
      user_name: user.display_name,
      user_avatar: user.avatar_url,
      content: sanitizedContent,
      thread_id: threadId,
      created_at: now,
      status: "sent",
      mentions: mentions.map(m =&gt; m.userId)
    };

    for (const member of channelMembers) {
      if (member.user_id !== userId) {  // Don't send to sender (already rendered)
        wsServer.send(`user:${member.user_id}`, {
          type: "message:received",
          data: messagePayload
        });
      } else {
        // Send confirmation to sender
        wsServer.send(`user:${member.user_id}`, {
          type: "message:confirmed",
          message_id: messageId,
          status: "sent"
        });
      }
    }

    // Step 10: Log activity
    analytics.trackEvent("message_sent", {
      message_id: messageId,
      channel_id: channelId,
      user_id: userId,
      content_length: sanitizedContent.length,
      mentions_count: mentions.length,
      timestamp: now
    });

    return Success({
      message_id: messageId,
      created_at: now,
      status: "sent"
    });

  } catch (error) {
    logger.error("Message send error:", error);
    throw error;
  }
}
```

WebSocket Server Configuration (Socket.IO):

```
// Node.js + Socket.IO setup
const express = require('express');
const http = require('http');
const socketIO = require('socket.io');
const app = express();
const server = http.createServer(app);
const io = socketIO(server, {
  cors: { origin: "*", methods: ["GET", "POST"] },
  transports: ['websocket', 'polling'],  // Fallback to polling if needed
  path: '/socket.io/'
});

// Socket connection handler
io.on('connection', (socket) =&gt; {
  console.log(`User connected: ${socket.id}`);

  // User joins channel room
  socket.on('channel:join', (data) =&gt; {
    const { channelId, userId } = data;
    socket.join(`channel:${channelId}`);
    socket.join(`user:${userId}`);  // User-specific room for DMs
    console.log(`${userId} joined channel ${channelId}`);
  });

  // Message send event
  socket.on('message:send', async (data) =&gt; {
    const { channelId, content, threadId } = data;
    const userId = socket.handshake.auth.userId;  // From JWT in handshake

    try {
      const result = await sendMessage(userId, channelId, content, threadId);

      // Broadcast to all in channel
      io.to(`channel:${channelId}`).emit('message:received', result);
    } catch (error) {
      socket.emit('error', { message: error.message });
    }
  });

  // Typing indicator
  socket.on('typing:start', (data) =&gt; {
    const { channelId, userId } = data;
    socket.to(`channel:${channelId}`).emit('typing:notification', {
      userId,
      user_name: (fetch from cache)
    });
  });

  socket.on('typing:stop', (data) =&gt; {
    const { channelId, userId } = data;
    socket.to(`channel:${channelId}`).emit('typing:stopped', { userId });
  });

  // Disconnect
  socket.on('disconnect', () =&gt; {
    console.log(`User disconnected: ${socket.id}`);
  });
});

server.listen(4000, () =&gt; {
  console.log('WebSocket server running on port 4000');
});
```

Performance Optimization - Database Indexes:

```
-- Critical indexes for message retrieval
CREATE INDEX idx_messages_channel_created
ON messages(channel_id, created_at DESC)
WHERE deleted_at IS NULL;

CREATE INDEX idx_messages_thread
ON messages(thread_id, created_at DESC)
WHERE thread_id IS NOT NULL;

-- For full-text search
CREATE INDEX idx_messages_content_fts
ON messages USING GIN(to_tsvector('english', content))
WHERE deleted_at IS NULL;

-- For user activity queries
CREATE INDEX idx_messages_user_created
ON messages(user_id, created_at DESC);

-- For pagination efficiency
CREATE INDEX idx_messages_composite
ON messages(channel_id, created_at DESC, id);
```

Message Delivery State Machine:

```
Client: Empty
  ↓ (user types)
Client: Composing
  ↓ (send button clicked)
Client: Sending (optimistic render)
Server: Processing (receive, validate, store)
Server/Client: Sent (persisted to DB)
Server/Client: Confirmed (WebSocket ack)
  ↓ (optional: user edits)
  → Editing → Updated → Edited Label
  ↓ (optional: user deletes)
  → Deleting → Deleted (soft-delete, hidden)
```

Unit Test Example (Jest):

```
// __tests__/messaging.test.js
const { sendMessage } = require('../services/messageService');
const database = require('../db');
const wsServer = require('../websocket');

jest.mock('../db');
jest.mock('../websocket');

describe('Message Service - sendMessage', () =&gt; {

  beforeEach(() =&gt; {
    jest.clearAllMocks();
  });

  test('sendMessage - Valid message sent successfully', async () =&gt; {
    // Arrange
    const userId = 'user-123';
    const channelId = 'channel-456';
    const content = 'Hello, team!';

    database.query.mockResolvedValueOnce([{ user_id: userId }]);  // channel member
    database.query.mockResolvedValueOnce([{ id: userId }]);        // user exists
    database.query.mockResolvedValueOnce({ insertId: 1 });         // message inserted
    database.query.mockResolvedValueOnce([{ user_id: userId }, { user_id: 'user-789' }]); // channel member

    // Act
    const result = await sendMessage(userId, channelId, content);

    // Assert
    expect(result.status).toBe('success');
    expect(result.message_id).toBeDefined();
    expect(database.query).toHaveBeenCalledWith(
      expect.stringContaining('INSERT INTO messages'),
      expect.any(Array)
    );
    expect(wsServer.send).toHaveBeenCalled();
  });

  test('sendMessage - Empty message rejected', async () =&gt; {
    const userId = 'user-123';
    const channelId = 'channel-456';

    try {
      await sendMessage(userId, channelId, '');
      fail('Should have thrown error');
    } catch (error) {
      expect(error.code).toBe(400);
      expect(error.message).toContain('cannot be empty');
    }
  });

  test('sendMessage - XSS prevention', async () =&gt; {
    const userId = 'user-123';
    const channelId = 'channel-456';
    const maliciousContent = '&lt;script&gt;alert("xss")&lt;/script&gt;';

    database.query.mockResolvedValueOnce([{ user_id: userId }]);

    const result = await sendMessage(userId, channelId, maliciousContent);

    // Check that script tags are escaped
    const storedContent = database.query.mock.calls[1][1][3];
    expect(storedContent).not.toContain('&lt;script&gt;');
    expect(storedContent).toContain('&lt;script&gt;');
  });

  test('sendMessage - Rate limiting (100+ msg/sec)', async () =&gt; {
    // Performance test
    const startTime = Date.now();
    const promises = [];

    for (let i = 0; i &lt; 100; i++) {
      promises.push(
        sendMessage(`user-${i}`, 'channel-456', `Message ${i}`)
      );
    }

    await Promise.all(promises);
    const duration = Date.now() - startTime;

    // 100 messages should complete in &lt;1000ms
    expect(duration).toBeLessThan(1000);
  });

});
```

Acceptance Criteria (QA):

- [ ] Valid message persisted to database with UUID ID
- [ ] Message timestamp server-generated (UTC, ISO 8601)
- [ ] Markdown formatting parsed correctly (**bold**, *italic*, `code`)
- [ ] @mentions detected and highlighted
- [ ] Mentioned users receive notification (if enabled)
- [ ] Message broadcast to all channel members <500ms
- [ ] Optimistic UI: message appears on sender's screen immediately
- [ ] XSS prevention: HTML escaping, no injection

- [ ] Empty messages rejected (400 error)
- [ ] Message too long (>4KB) rejected
- [ ] WebSocket confirmation received
- [ ] Message searchable immediately after send
- [ ] Load test: 100+ msg/sec at 50 concurrent users

6. Data Flow Diagrams (DFD)

6.1 DFD Level 0 (System Boundary)

```
┌────────────────┐
│  End Users    │
└───────┬────────┘
       │ Text Input/Commands
       ↓
┌──────────────────────────────┐
│   ChatFlow System        │
│  (Black Box)             │
│                          │
│ • Authentication         │
│ • Messaging              │
│ • Workspace Mgmt         │
│ • Notifications          │
└───────┬────────┬─────────────┘
       │       │
       │       │ External Services
       ↓       ├─→ SendGrid (email)
  Data Store   ├─→ OAuth Providers
  (DB)         ├─→ S3 (files)
               └─→ Analytics
```

6.2 DFD Level 1 (Core Processes)

```
┌────────────────┐
│  End User     │
└───────┬────────┘
       │
       ├─ Auth Request
       ↓
┌─────────────────────────┐
│ 1. Authentication    │ ──→ Verify Email
│    Service           │ ──→ Generate JWT
└─────────┬───────────────┘
         │
         ├─ User Token
         ↓
┌──────────────────────────────┐
│ 2. Message Service       │ ──→ Store Message
│    (Real-Time)           │ ──→ Broadcast WebSocket
└─────────┬──────────────────┘
         │
         ├─ New Message Event
         ↓
┌──────────────────────────────┐
│ 3. Notification Service  │ ──→ @mention detected
│                          │ ──→ Push notification
└─────────┬──────────────────┘
         │
         ├─ Index Message
         ↓
┌──────────────────────────────┐
│ 4. Search Service        │ ──→ Elasticsearch index
│    (Async)               │ ──→ Full-text search
└─────────┬──────────────────┘
```

```
         |
    [User Feed]
```

6.3 DFD: Message Sending Flow (Detailed)

```
User Types &amp; Sends Message
    │
    ├─ Client: Validation
    │  ├─ Not empty?
    │  ├─ &lt;4KB?
    │  └─ User in channel?
    │
    ↓
Client: Optimistic Render (show msg immediately)
    │
    ↓
HTTP POST /api/channels/{id}/messages
    │
    ↓
Server: Authentication Middleware
    ├─ JWT validation
    ├─ User exists?
    └─ User member of channel?
    │
    ↓
Server: Sanitization
    ├─ Escape HTML
    ├─ Parse Markdown
    └─ Extract @mentions
    │
    ↓
Server: Database Operations (atomic transaction)
    ├─ INSERT messages
    ├─ INSERT notifications (for @mentions)
    ├─ UPDATE channels (message_count++)
    └─ INSERT audit_log
    │
    ↓ (Success)
    ├─ HTTP 201 response (confirm message_id + timestamp)
    │
    ├─ WebSocket: Broadcast to channel members
    │  ├─ Exclude sender (already rendered)
    │  └─ &lt;500ms latency target
    │
    ├─ Async: Index for search (Elasticsearch)
    │  └─ &lt;5s lag acceptable
    │
    └─ Async: Send push notifications
       └─ For @mentioned users
```

7. Sequence Diagrams - Critical Flows

7.1 Sequence: User Registration → Login → Send Message

```
Actor                Chrome Browser            ChatFlow Server      PostgreSQL
 │                        │                          │                   │
 │─(fills form)─→          │                          │                   │
 │              │─(POST /api/auth/register)─→       │                   │
 │              │                                    │─(INSERT user)─→   │
 │              │                                    │                   │─(OK)
 │              │                                    │─(send email)─→   SendGrid
 │              │◄──(201 Created)─────────────────────│                   │
 │◄───(check email)────────│                          │                   │
 │                        │                          │                   │
 │─(click verify link)─→  │                          │                   │
 │              │─(GET /auth/verify?token=...)─→    │                   │
 │              │                                 │─(UPDATE user email_verified=true)─→
 │              │◄──(302 Redirect to login)──────────│                  │
 │              │                                 │                  │
 │─(login)─→    │                                  │                  │
 │              │─(POST /api/auth/login)─→          │                  │
 │              │                                 │─(SELECT user)─→  │
 │              │                                 │                  │─(user data)
 │              │                                 │─(verify password)─│
 │              │◄──(200 OK {access_token})──────────│                  │
 │◄───(logged in, redirect to workspace)          │                  │
 │                       │                        │                  │
 │─(join channel)─→       │                        │                  │
 │              │─(GET /api/channels/{id}/messages)│                  │
 │              │                                 │─(SELECT messages)→
 │              │                                 │                  │─(fetch)
 │              │◄──(200 OK [messages...])───────────│                  │
 │◄──(display messages)──│                        │                  │
 │                       │                        │                  │
 │─(type &amp; send msg)─→     │                        │                  │
 │              │─(WebSocket message:send)─→       │                  │
 │              │                                 │─(INSERT message)→ │
 │              │◄──(message:confirmed)───────────────│◄─(inserted)─────────│
 │◄──(msg appears in UI)──│                        │                  │
 │              │◄─(WebSocket broadcast)─────────────│                  │
 │              │         │                        │                  │
 │            Other       │                        │                  │
 │            Clients     │                        │                  │
 │              │◄─(WebSocket message:received)────│                  │
```

8. API Specifications (REST + WebSocket)

8.1 REST Endpoints - Authentication

```
POST /api/auth/register
├─ Description: Create new user account
├─ Auth: None (public)
├─ Request: {email, password, display_name}
├─ Response:
│  ├─ 201 Created: {user_id, message: "Confirmation email sent"}
│  ├─ 400 Bad Request: {error: "..."}
│  └─ 409 Conflict: {error: "Email already exists"}
└─ Rate Limit: 5 per minute per IP

POST /api/auth/login
├─ Description: Authenticate user
├─ Auth: None (public)
├─ Request: {email, password}
├─ Response:
│  ├─ 200 OK: {access_token, refresh_token, expires_in, user, workspaces}
│  ├─ 401 Unauthorized: {error: "Invalid email or password"}
│  └─ 403 Forbidden: {error: "Email not verified"}
└─ Rate Limit: 5 failed attempts → 15min lockout

POST /api/auth/logout
├─ Description: Invalidate session
├─ Auth: Bearer token required
├─ Request: {}
├─ Response: 200 OK: {message: "Logged out"}
└─ Rate Limit: Per-user rate limit (no abuse)

POST /api/auth/refresh
├─ Description: Refresh access token
├─ Auth: Refresh token in httpOnly cookie
├─ Request: {}
├─ Response: 200 OK: {access_token, expires_in}
└─ Rate Limit: Per-user rate limit
```

8.2 REST Endpoints - Messages

```
POST /api/channels/:channelId/messages
├─ Description: Send message to channel
├─ Auth: Bearer token required
├─ Path Params: channelId (UUID)
├─ Request: {content, thread_id (optional)}
├─ Response:
│  ├─ 201 Created: {message_id, created_at, status: "sent"}
│  ├─ 400 Bad Request: {error: "Message cannot be empty"}
│  ├─ 403 Forbidden: {error: "No access to channel"}
│  └─ 413 Payload Too Large: {error: "Message exceeds 4000 chars"}
├─ Performance: P95 &lt;150ms
└─ Rate Limit: 100 msg/min per user, 1000 msg/min per channel

GET /api/channels/:channelId/messages
├─ Description: Fetch messages (paginated)
├─ Auth: Bearer token required
├─ Query Params:
│  ├─ page (default 1)
│  ├─ limit (default 50, max 100)
│  └─ after (timestamp for pagination)
├─ Response:
│  ├─ 200 OK: {
│  │  messages: [{id, user_id, user_name, content, created_at, ...}],
│  │  page: 1,
│  │  total_messages: 5000,
│  │  has_more: true
│  │ }
│  └─ 403 Forbidden: {error: "No access"}
├─ Performance: P95 &lt;100ms (50 messages)
└─ Rate Limit: 100 req/min per user

PUT /api/messages/:messageId
├─ Description: Edit message (1-hour window)
├─ Auth: Bearer token required
├─ Request: {content}
├─ Response:
│  ├─ 200 OK: {message_id, edited_at, status: "edited"}
│  ├─ 400 Bad Request: {error: "Edit window expired"}
│  ├─ 403 Forbidden: {error: "Can only edit own messages"}
│  └─ 404 Not Found: {error: "Message not found"}
└─ Performance: P95 &lt;200ms

DELETE /api/messages/:messageId
├─ Description: Delete message (soft delete)
├─ Auth: Bearer token required
├─ Response:
│  ├─ 204 No Content
│  └─ 403 Forbidden: {error: "Cannot delete other's message"}
└─ Performance: P95 &lt;200ms

GET /api/search
├─ Description: Full-text search messages
├─ Auth: Bearer token required
├─ Query Params:
│  ├─ q (keyword, required)
│  ├─ from (author filter)
│  ├─ in (channel filter)
│  ├─ before (date)
│  ├─ after (date)
│  ├─ page (default 1)
│  └─ limit (default 20)
├─ Response:
│  ├─ 200 OK: {
│  │  query: "...",
│  │  results: [{message_id, channel, author, snippet, created_at}],
│  │  total: 42,
│  │  query_time_ms: 145
│  │ }
│  └─ 400 Bad Request: {error: "Query required"}
```

```
├─ Performance: P95 &lt;2s
└─ Rate Limit: 30 searches/min per user
```

8.3 WebSocket Events

```
Client → Server Events:

message:send
├─ Payload: {content, channel_id, thread_id (optional)}
├─ Handler: Validate → Store → Broadcast
└─ Response: message:confirmed or error

message:edit
├─ Payload: {message_id, new_content}
├─ Handler: Validate 1-hour window → Update → Broadcast
└─ Response: message:edited

typing:start
├─ Payload: {channel_id}
├─ Handler: Broadcast to channel members
└─ Duration: 3s auto-expire

reaction:add
├─ Payload: {message_id, emoji}
├─ Handler: Add reaction → Broadcast
└─ Response: reaction:added


Server → Client Events (Broadcast):

message:received
├─ Payload: {message_id, channel_id, user_id, user_name, content, created_at}
├─ Recipients: All channel members (except sender)
└─ Latency: &lt;500ms

message:confirmed
├─ Payload: {message_id, created_at, status: "sent"}
├─ Recipients: Sender only
└─ Latency: &lt;200ms

message:edited
├─ Payload: {message_id, new_content, edited_at}
├─ Recipients: All channel members
└─ Latency: &lt;300ms

typing:notification
├─ Payload: {user_id, user_name}
├─ Recipients: All channel members (except typer)
└─ Latency: &lt;100ms

presence:update
├─ Payload: {user_id, status, last_seen}
├─ Recipients: All users in workspace
└─ Latency: &lt;500ms
```

9. Database Schema (Complete PostgreSQL)

9.1 Complete Schema SQL

```
-- Enable UUID extension
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

-- Users table
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    email VARCHAR(255) UNIQUE NOT NULL,
```

```sql
    password_hash VARCHAR(255),  -- bcrypt: $2b$12$...
    display_name VARCHAR(100) NOT NULL,
    avatar_url VARCHAR(500),
    bio TEXT,
    timezone VARCHAR(50) DEFAULT 'UTC',
    status VARCHAR(20) DEFAULT 'offline',  -- online, away, offline, dnd
    status_message VARCHAR(100),
    email_verified BOOLEAN DEFAULT false,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    last_login TIMESTAMP WITH TIME ZONE,
    deleted_at TIMESTAMP WITH TIME ZONE,

    CONSTRAINT status_valid CHECK (status IN ('online', 'away', 'offline', 'dnd'))
);

-- Workspaces table
CREATE TABLE workspaces (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    name VARCHAR(100) NOT NULL,
    slug VARCHAR(100) UNIQUE NOT NULL,
    description TEXT,
    owner_id UUID NOT NULL REFERENCES users(id) ON DELETE RESTRICT,
    plan VARCHAR(20) DEFAULT 'free',  -- free, pro, enterprise
    member_limit INT DEFAULT 30,
    member_count INT DEFAULT 1,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    deleted_at TIMESTAMP WITH TIME ZONE,

    CONSTRAINT plan_valid CHECK (plan IN ('free', 'pro', 'enterprise')),
    CONSTRAINT member_limit_positive CHECK (member_limit &gt; 0)
);

-- User-Workspace membership
CREATE TABLE user_workspace_members (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    workspace_id UUID NOT NULL REFERENCES workspaces(id) ON DELETE CASCADE,
    user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    role VARCHAR(20) DEFAULT 'member',  -- owner, admin, moderator, member
    joined_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    status VARCHAR(20) DEFAULT 'active',  -- active, invited, left, removed

    UNIQUE(workspace_id, user_id),
    CONSTRAINT role_valid CHECK (role IN ('owner', 'admin', 'moderator', 'member')),
    CONSTRAINT status_valid CHECK (status IN ('active', 'invited', 'left', 'removed'))
);

-- Channels table
CREATE TABLE channels (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    workspace_id UUID NOT NULL REFERENCES workspaces(id) ON DELETE CASCADE,
    name VARCHAR(80) NOT NULL,
    slug VARCHAR(80) NOT NULL,
    type VARCHAR(20) DEFAULT 'public',  -- public, private, direct, group_dm
    description TEXT,
    topic VARCHAR(500),
    created_by UUID NOT NULL REFERENCES users(id) ON DELETE SET NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    archived BOOLEAN DEFAULT false,
    archived_at TIMESTAMP WITH TIME ZONE,
    deleted_at TIMESTAMP WITH TIME ZONE,
    message_count INT DEFAULT 0,

    UNIQUE(workspace_id, slug),
    CONSTRAINT type_valid CHECK (type IN ('public', 'private', 'direct', 'group_dm')),
    CONSTRAINT positive_message_count CHECK (message_count &gt;= 0)
);

-- Channel members
CREATE TABLE channel_members (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    channel_id UUID NOT NULL REFERENCES channels(id) ON DELETE CASCADE,
    user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    role VARCHAR(20) DEFAULT 'member',  -- moderator, member
    joined_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    last_read_message_id UUID,

    UNIQUE(channel_id, user_id),
    CONSTRAINT role_valid CHECK (role IN ('moderator', 'member'))
);

-- Messages table (core)
CREATE TABLE messages (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    channel_id UUID NOT NULL REFERENCES channels(id) ON DELETE CASCADE,
    user_id UUID NOT NULL REFERENCES users(id) ON DELETE SET NULL,
    content TEXT NOT NULL,
    thread_id UUID REFERENCES messages(id) ON DELETE CASCADE,  -- For threaded replies
    edited_at TIMESTAMP WITH TIME ZONE,
    deleted_at TIMESTAMP WITH TIME ZONE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,

    CONSTRAINT content_not_empty CHECK (length(content) &gt; 0),
    CONSTRAINT content_max_length CHECK (length(content) &lt;= 4000)
);

-- Message edit history (immutable log)
CREATE TABLE message_edit_history (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    message_id UUID NOT NULL REFERENCES messages(id) ON DELETE CASCADE,
    previous_content TEXT NOT NULL,
    new_content TEXT NOT NULL,
    edited_by UUID NOT NULL REFERENCES users(id) ON DELETE SET NULL,
    edited_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- Reactions table
CREATE TABLE reactions (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    message_id UUID NOT NULL REFERENCES messages(id) ON DELETE CASCADE,
    user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    emoji VARCHAR(10) NOT NULL,  -- Unicode emoji
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,

    UNIQUE(message_id, user_id, emoji)
);

-- Direct messages
CREATE TABLE direct_messages (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    sender_id UUID NOT NULL REFERENCES users(id) ON DELETE SET NULL,
    recipient_id UUID NOT NULL REFERENCES users(id) ON DELETE SET NULL,
    content TEXT NOT NULL,
    edited_at TIMESTAMP WITH TIME ZONE,
    deleted_at TIMESTAMP WITH TIME ZONE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,

    CONSTRAINT content_not_empty CHECK (length(content) &gt; 0)
);

-- Files table
CREATE TABLE files (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    message_id UUID REFERENCES messages(id) ON DELETE SET NULL,
    dm_id UUID REFERENCES direct_messages(id) ON DELETE SET NULL,
    filename VARCHAR(255) NOT NULL,
    file_size INT NOT NULL,  -- bytes
    file_type VARCHAR(50),  -- MIME type: image/jpeg, application/pdf, etc.
    storage_path VARCHAR(500) NOT NULL,  -- S3 path or local path
    uploaded_by UUID NOT NULL REFERENCES users(id) ON DELETE SET NULL,
    uploaded_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    deleted_at TIMESTAMP WITH TIME ZONE,

    CONSTRAINT exactly_one_parent CHECK (
        (message_id IS NOT NULL AND dm_id IS NULL) OR
        (message_id IS NULL AND dm_id IS NOT NULL)
    ),
    CONSTRAINT positive_file_size CHECK (file_size &gt; 0)
);

-- Notifications table
CREATE TABLE notifications (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    type VARCHAR(50) NOT NULL,  -- mention, channel_activity, dm, reaction
    channel_id UUID REFERENCES channels(id) ON DELETE CASCADE,
    message_id UUID REFERENCES messages(id) ON DELETE CASCADE,
    actor_id UUID REFERENCES users(id) ON DELETE SET NULL,  -- Who triggered
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    read BOOLEAN DEFAULT false,
    read_at TIMESTAMP WITH TIME ZONE,

    CONSTRAINT type_valid CHECK (type IN ('mention', 'channel_activity', 'dm', 'reaction'))
);

-- Audit logs (immutable)
CREATE TABLE audit_logs (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    workspace_id UUID NOT NULL REFERENCES workspaces(id) ON DELETE CASCADE,
    actor_id UUID REFERENCES users(id) ON DELETE SET NULL,
    action VARCHAR(100) NOT NULL,  -- user_created, channel_deleted, message_edited
    resource_type VARCHAR(50),  -- user, channel, message
    resource_id UUID,
    details JSONB,  -- Additional context: {ip, user_agent, ...}
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,

    CONSTRAINT immutable CHECK (true)  -- Conceptual; enforce in app layer
);

-- Performance Indexes

-- Critical path: Fetch messages in channel (chronological)
CREATE INDEX idx_messages_channel_created ON messages(channel_id, created_at DESC)
WHERE deleted_at IS NULL;

-- Pagination support
CREATE INDEX idx_messages_channel_id_created_id ON messages(channel_id, created_at DESC, id)
WHERE deleted_at IS NULL;

-- Thread replies
CREATE INDEX idx_messages_thread ON messages(thread_id, created_at DESC)
WHERE thread_id IS NOT NULL AND deleted_at IS NULL;

-- User activity
CREATE INDEX idx_messages_user ON messages(user_id, created_at DESC)
WHERE deleted_at IS NULL;

-- Full-text search
CREATE INDEX idx_messages_content_fts ON messages USING GIN(to_tsvector('english', content))
WHERE deleted_at IS NULL;

-- User queries
CREATE INDEX idx_users_email ON users(LOWER(email));

-- Workspace queries
CREATE INDEX idx_user_workspaces ON user_workspace_members(user_id, workspace_id);
CREATE INDEX idx_workspace_channels ON channels(workspace_id);

-- Channel member queries
CREATE INDEX idx_channel_members_user ON channel_members(user_id);

-- DM queries
CREATE INDEX idx_direct_messages_pair ON direct_messages(sender_id, recipient_id, created_at DESC);
```

```
-- Notification queries
CREATE INDEX idx_notifications_user ON notifications(user_id, created_at DESC);

-- Audit log queries
CREATE INDEX idx_audit_logs_workspace ON audit_logs(workspace_id, created_at DESC);
CREATE INDEX idx_audit_logs_action ON audit_logs(action, created_at DESC);

-- Foreign key index optimization
CREATE INDEX idx_reactions_user ON reactions(user_id);

-- Cluster for common access pattern (optional, for frequent sequential access)
-- CLUSTER messages USING idx_messages_channel_created;
```

10. Testing Strategy

10.1 Unit Tests (Jest) - Target 80% Coverage

```
// Example: __tests__/auth.test.js

describe('Authentication Service', () =&gt; {

  describe('registerUser', () =&gt; {
    test('Valid signup creates account', async () =&gt; {
      const result = await auth.registerUser({
        email: 'new@example.com',
        password: 'ValidPass123!',
        displayName: 'John Doe'
      });

      expect(result.userId).toBeDefined();
      expect(result.message).toContain('Confirmation email sent');
    });

    test('Duplicate email rejected', async () =&gt; {
      const error = await auth.registerUser({
        email: 'existing@example.com',
        password: 'ValidPass123!',
        displayName: 'Jane Doe'
      }).catch(e =&gt; e);

      expect(error.code).toBe(409);
      expect(error.message).toContain('Email already exists');
    });

    test('Weak password rejected', async () =&gt; {
      const error = await auth.registerUser({
        email: 'new@example.com',
        password: 'weak',  // Too short
        displayName: 'John Doe'
      }).catch(e =&gt; e);

      expect(error.code).toBe(400);
    });
  });

  describe('authenticateUser', () =&gt; {
    test('Valid credentials issue JWT', async () =&gt; {
      const result = await auth.authenticateUser({
        email: 'user@example.com',
        password: 'ValidPass123!'
      });

      expect(result.access_token).toBeDefined();
      expect(result.refresh_token).toBeDefined();
      expect(result.expires_in).toBe(86400);
    });
    test('Invalid password triggers rate limit', async () =&gt; {
      for (let i = 0; i &lt; 5; i++) {
        await auth.authenticateUser({
          email: 'user@example.com',
          password: 'WrongPassword'
        }).catch(e =&gt; e);
      }

      const error = await auth.authenticateUser({
        email: 'user@example.com',
        password: 'WrongPassword'
      }).catch(e =&gt; e);

      expect(error.code).toBe(429);
      expect(error.message).toContain('Account locked');
    });
  });
});

// Test coverage reporting
// npm test -- --coverage
// Target: &gt;80% coverage for auth, messaging, channels
```

10.2 Integration Tests

```
// __tests__/e2e-messaging.test.js

describe('End-to-End: Message Sending', () =&gt; {

  test('User sends message and sees real-time broadcast', async () =&gt; {
    // Setup
    const user1 = await createTestUser('user1@test.com');
    const user2 = await createTestUser('user2@test.com');
    const workspace = await createTestWorkspace(user1.id);
    const channel = await createTestChannel(workspace.id);
    await joinChannels([user1.id, user2.id], channel.id);

    // Send message
    const sendResult = await messageService.sendMessage(
      user1.id,
      channel.id,
      'Hello team!'
    );

    // Verify persistence
    const storedMessage = await db.query(
      'SELECT * FROM messages WHERE id = ?',
      [sendResult.message_id]
    );
    expect(storedMessage.content).toBe('Hello team!');

    // Verify search indexing
    await sleep(100);  // Wait for async indexing
    const searchResult = await messageService.search('Hello');
    expect(searchResult.results).toContainEqual(
      expect.objectContaining({ message_id: sendResult.message_id })
    );

    // Verify WebSocket broadcast
    const wsEvent = await waitForWebSocketEvent(channel.id, 'message:received', 500);
    expect(wsEvent.message.id).toBe(sendResult.message_id);
  });

});
```

10.3 Performance Tests (K6 Load Testing)

```
// tests/performance.js (K6 script)

import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {
  stages: [
    { duration: '30s', target: 50 },    // Ramp up to 50 users
    { duration: '1m', target: 50 },     // Stay at 50
    { duration: '30s', target: 0 },     // Ramp down
  ],
  thresholds: {
    'http_req_duration': ['p(95)&lt;500', 'p(99)&lt;1000'],  // 95% &lt; 500ms, 99% &lt; 1s
    'http_req_failed': ['rate&lt;0.1'],  // &lt;10% failure
  },
};

const API_URL = 'http://slackteam.lab.home.lucasacchi.net:4000';

export default function () {
  // Authenticate
  const loginRes = http.post(`${API_URL}/api/auth/login`, {
    email: 'user@example.com',
    password: 'ValidPass123!'
  });

  const token = loginRes.json('access_token');

  // Send 5 messages
  for (let i = 0; i &lt; 5; i++) {
    const msgRes = http.post(
      `${API_URL}/api/channels/channel-456/messages`,
      JSON.stringify({
        content: `Load test message ${i}`
      }),
      {
        headers: {
          'Authorization': `Bearer ${token}`,
          'Content-Type': 'application/json'
        }
      }
    );

    check(msgRes, {
      'status is 201': (r) =&gt; r.status === 201,
      'response time &lt; 500ms': (r) =&gt; r.timings.duration &lt; 500
    });

    sleep(1);
  }
}

// Run: k6 run tests/performance.js
// Expected: 50 concurrent users, message send &lt;500ms (p95)
```

11. Performance Benchmarks (Lab VM)

Lab Specifications

- Host: slackteam.lab.home.lucasacchi.net
- CPU: TBD (check `nproc`)
- Memory: TBD (check `free -h`)
- Disk: TBD (check `df -h`)

- Node.js: v24.11.1
- PostgreSQL: 15
- Redis: 7+

Performance Targets (MVP)

```
Message Latency (Real-Time WebSocket):
  P50:  &lt;100ms
  P95:  &lt;300ms
  P99:  &lt;500ms
  Target: 100+ msg/sec at 50 concurrent users

API Response Time (HTTP):
  GET /messages:   P95 &lt;100ms (50 messages)
  POST /messages:  P95 &lt;150ms
  GET /search:     P95 &lt;2s
  PUT /messages:   P95 &lt;200ms
  All others:      P95 &lt;100ms

Database Queries:
  SELECT (simple):    &lt;10ms
  SELECT (paginated):  &lt;50ms (p95)
  INSERT (message):   &lt;100ms (p95)
  Complex JOIN:       &lt;200ms (p95)

Database Size:
  10K messages:     ~10MB
  1M messages:      ~1GB
  100K users:       ~500MB
  Total DB 1M msg:  ~2GB (safe for 100GB disk)

Cache Performance (Redis):
  Hit rate target: 90%+
  Response time: &lt;5ms per hit

Search Performance (Elasticsearch):
  Index latency: &lt;5s lag from message send
  Query latency: &lt;2s (p95)
  Index size: ~500MB per 1M messages
```

Monitoring Setup

```
# Monitor server health during load test<a></a>
watch -n 1 'ps aux | grep node; free -h; df -h'

# Monitor database<a></a>
psql -U chatflow -d chatflow_dev -c "SELECT count(*) FROM messages;"

# Monitor Node.js processes<a></a>
node --prof  # CPU profiling
node --expose-gc  # For GC analysis

# Check system load<a></a>
top  # Interactive
iostat 1  # I/O stats
vmstat 1  # Memory/swap/IO
```

12. Deployment to Lab (Step-by-Step)

12.1 Pre-Deployment Checklist

```
☐ Code reviewed and merged to main branch
☐ All tests passing (unit + integration)
☐ Load test passed (50 concurrent, &lt;500ms latency)
☐ Security audit completed (no critical CVEs)
☐ Database schema reviewed + migration tested
☐ Backup strategy configured (daily snapshots)
☐ Monitoring/alerting configured (uptime, errors)
☐ SSH key setup (slackteam user, passwordless)
☐ Nginx configuration tested (reverse proxy works)
☐ Environment variables (.env) prepared
☐ Documentation updated (deployment runbook)
```

12.2 Deployment Script

```
#!/bin/bash
# deploy.sh - Deploy ChatFlow to lab<a></a>

set -e  # Exit on error

HOST="slackteam.lab.home.lucasacchi.net"
USER="slackteam"
APP_DIR="/home/slackteam/chatflow"
BACKUP_DIR="/home/slackteam/backups"
TIMESTAMP=$(date +%Y%m%d_%H%M%S)

echo "[INFO] Deploying ChatFlow to $HOST..."

# 1. SSH connection test<a></a>
ssh -o ConnectTimeout=5 $USER@$HOST "echo Connected" || {
  echo "[ERROR] Cannot reach $HOST"
  exit 1
}

# 2. Backup current deployment<a></a>
ssh $USER@$HOST "mkdir -p $BACKUP_DIR &amp;&amp; cp -r $APP_DIR $BACKUP_DIR/chatflow_backup_$TIMESTAMP"
echo "[OK] Backup created: $BACKUP_DIR/chatflow_backup_$TIMESTAMP"

# 3. Pull latest code<a></a>
ssh $USER@$HOST "cd $APP_DIR &amp;&amp; git pull origin main &amp;&amp; git log -1 --oneline"
echo "[OK] Code pulled"

# 4. Install dependencies<a></a>
ssh $USER@$HOST "cd $APP_DIR &amp;&amp; npm install --production"
echo "[OK] Dependencies installed"

# 5. Database migrations<a></a>
ssh $USER@$HOST "cd $APP_DIR &amp;&amp; npm run migrate:up"
echo "[OK] Database migrations completed"

# 6. Build frontend<a></a>
ssh $USER@$HOST "cd $APP_DIR/frontend &amp;&amp; npm run build"
echo "[OK] Frontend built"

# 7. Restart services<a></a>
ssh $USER@$HOST "pm2 restart chatflow || pm2 start npm --name chatflow -- start"
echo "[OK] Services restarted"

# 8. Verify deployment<a></a>
sleep 5
curl -f http://$HOST:8282 &gt; /dev/null &amp;&amp; echo "[OK] Frontend accessible" || echo "[WARN] Fronter
curl -f http://$HOST:8282/api/health &gt; /dev/null &amp;&amp; echo "[OK] Backend health check passed" || e

echo "[SUCCESS] Deployment completed!"
echo "Rollback: ssh $USER@$HOST 'cp -r $BACKUP_DIR/chatflow_backup_$TIMESTAMP $APP_DIR'"
```

13. Risk Analysis & Mitigation

Risk	Impact	Probability	Mitigation
WebSocket latency >1s	Poor UX	Medium	Load test early (Week 2), optimize connection pooling
Database query performance	Message latency	Low	Proper indexing, query optimization, connection pool tuning
Memory leak in Node.js	Server crash	Low	PM2 auto-restart, memory monitoring, heap snapshots
Security vulnerability	Data breach	Low	OWASP Top 10 checks, SQL injection tests, XSS prevention
File upload abuse	Disk full	Medium	Implement file size limit, disk quota per user, cleanup job
Email delivery failure	Users can't verify	Low	Implement resend mechanism, fallback provider
Network connectivity loss	Offline state**	Medium	Implement client-side queue, sync on reconnect
Rate limiting bypass	DoS attack	Low	Implement IP-based + user-based rate limiting

14. Appendices

Appendix A: Glossary

Term	Definition
DAU	Daily Active Users
JWT	JSON Web Token (stateless session)
WebSocket	Persistent bidirectional communication
Soft Delete	Mark as deleted without removing from DB
Idempotency	Operation safe to retry
P50/P95/P99	Percentile latency
RBAC	Role-Based Access Control
FBS	Functional Breakdown Structure
DFD	Data Flow Diagram

Appendix B: References

- PRD v2.0 Lab-Optimized
- Henderson Functional Specs Template
- Node.js 24 Documentation
- PostgreSQL 15 Documentation
- Socket.IO Documentation

Document Status: Final for Development
Version: 2.0 (Lab-Optimized)

Last Updated: November 19, 2025
Next Review: Upon completion of Milestone 1 (Week 1)

Approved By:

- [ ] Senior Architect
- [ ] Engineering Lead
- [ ] Lab Admin (slackteam)
- [ ] QA Lead