Private Chat & Calls (DEEP)

Phase 2 Deep Dive

NexGate / QBIT SPARK | Version 1.0 *1:1 DMs · Group Chats · Voice Calls · Video Calls · Ejabberd · WebRTC*

Table of Contents

---

1. Scope

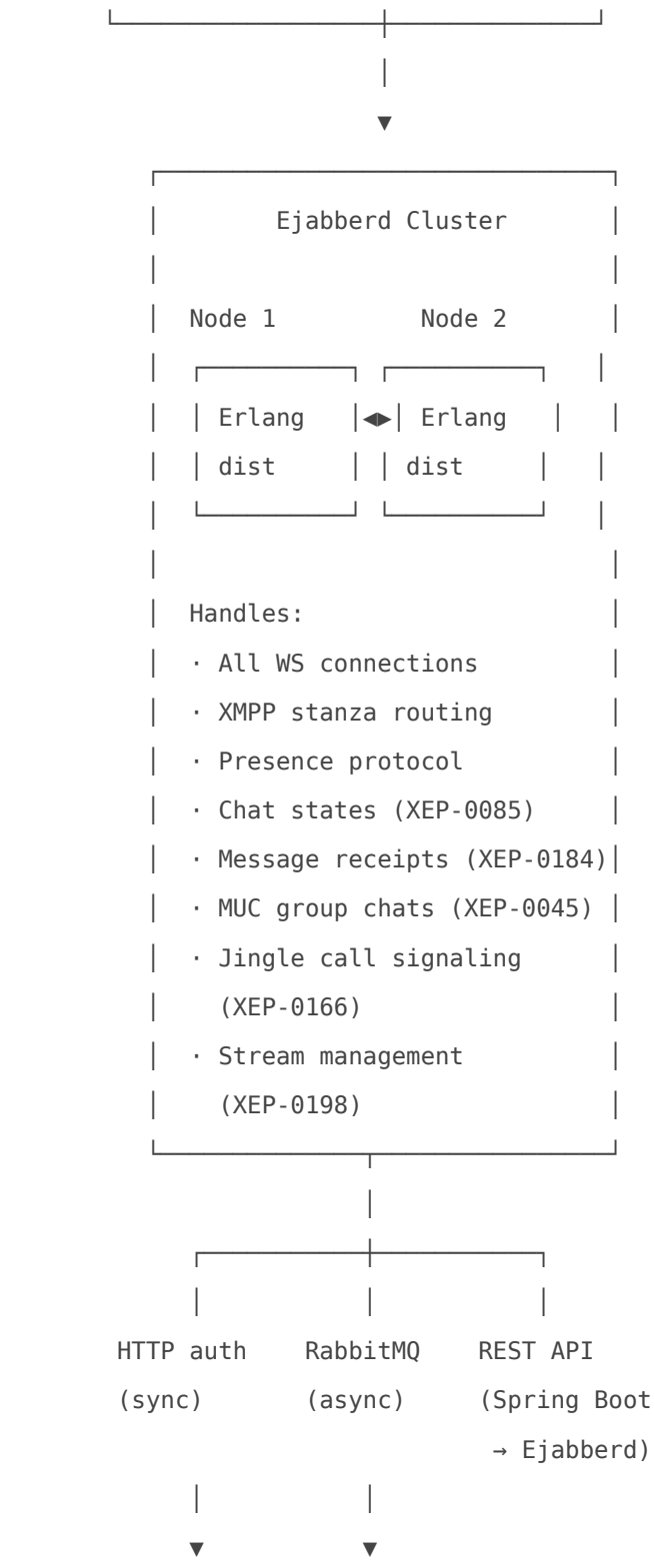
This document covers only private communication features in Phase 2:

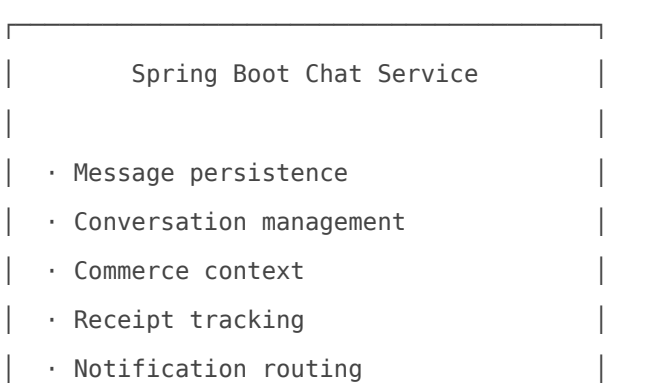
```
IN SCOPE:
  1:1 private DMs (personal + shop commerce DMs)
  Group chats — private + public (up to 500 members)
  Group join model (consent DM + invite link)
  Voice calls (1:1 + group)
  Video calls (1:1 + group)
  Audio ↔ video switching during calls
  Screen sharing
  Chat states (typing, recording voice note)
  Message interactions (edit, delete, react, forward, reply)
  Message receipts (sent, delivered, read)
  Presence (online, offline, last seen)
  Multi-device support
  Offline delivery

OUT OF SCOPE:
  Broadcast channels → NOT built (VP Feed covers this — see Doc 6)
  VP Live streaming  → covered in VP Live doc
  VP Audio Spaces    → covered in VP Live doc
  File Thunder       → covered in File Thunder docs
```

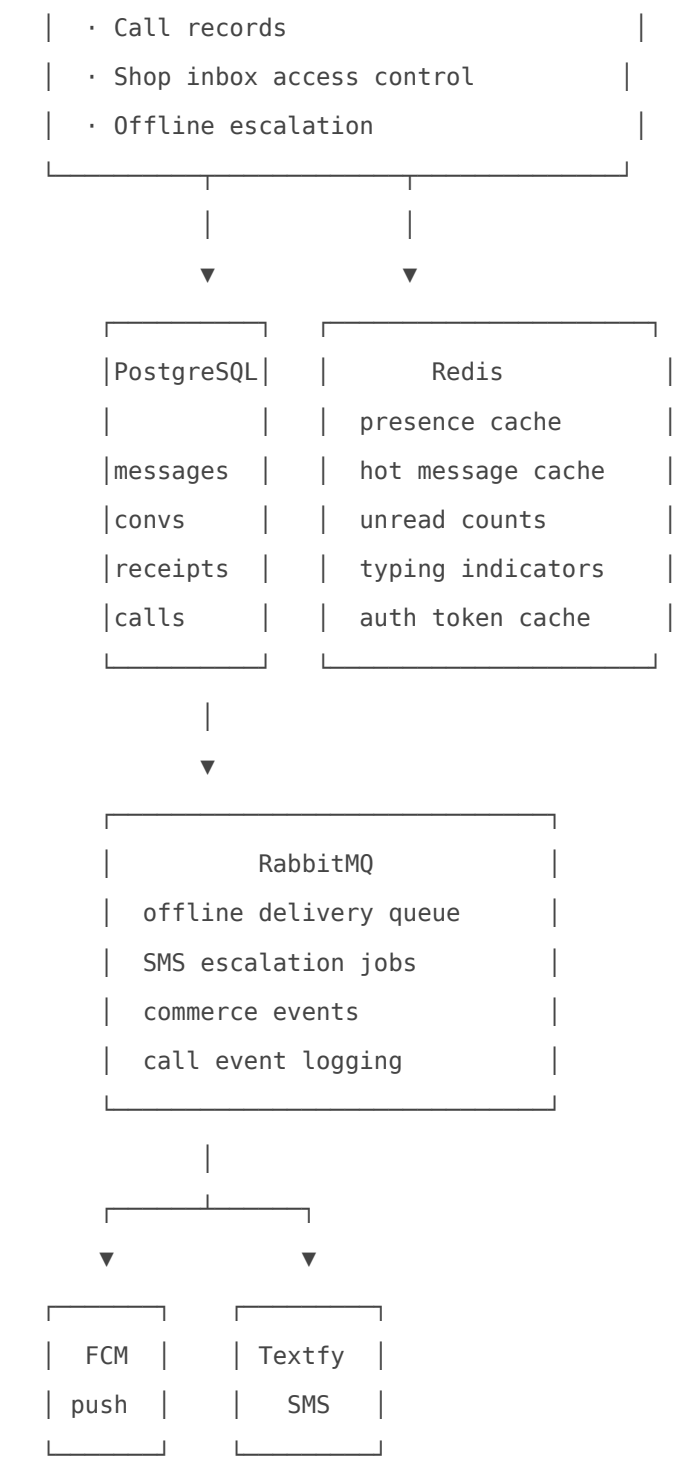
---

2. Architecture Overview

```
┌──────────────────────────────────────────────────────┐
│                NexGate Mobile App                    │
│                                                      │
│   Personal Inbox    Shop Inbox     Call Screen       │
└─────────┬──────────────────┬──────────────┬──────────┘
          │                  │              │
    WebSocket           WebSocket       WebRTC
    XMPP stanzas        XMPP stanzas   (calls only)
    MessagePack         MessagePack
          │                  │              │
```


Ejabberd Cluster
Node 1
Node 2
Erlang dist
Erlang dist
Handles:
· All WS connections
· XMPP stanza routing
· Presence protocol
· Chat states (XEP-0085)
· Message receipts (XEP-0184)
· MUC group chats (XEP-0045)
· Jingle call signaling
(XEP-0166)
· Stream management
(XEP-0198)
HTTP auth
(sync)
RabbitMQ
(async)
REST API
(Spring Boot
→ Ejabberd)


Spring Boot Chat Service
· Message persistence
· Conversation management
· Commerce context
· Receipt tracking
· Notification routing



Also:
  Coturn TURN server (separate VPS)
  → relay for voice/video calls
  → when EA carrier NAT blocks P2P

---

3. XMPP & Ejabberd Fundamentals

JID — Every Entity Has an Address

```
In XMPP every connected entity has a JID (Jabber ID)
Works like an email address for messaging

Personal user (full JID):
  kibuti@nexgate.com/android
  |       |            |
  user    domain       resource (device)

Personal user (bare JID):
  kibuti@nexgate.com
  (without device — used for addressing)

Shop identity:
  techstore@shops.nexgate.com
  (the shop — not the person behind it)

System bot:
  system@nexgate.com
  (order updates, notifications)

Group chat room:
  group-abc@conference.nexgate.com

Multi-device — same user, multiple resources:
  kibuti@nexgate.com/android   ← phone
  kibuti@nexgate.com/tablet    ← tablet
  Both receive messages simultaneously
  READ on one → Ejabberd notifies other to clear notification
```

XEPs — XMPP Extension Protocols

```
XMPP base protocol = just message/presence/iq stanzas
XEPs add specific capabilities on top

XEPs enabled for NexGate private chat:
```

```
XEP-0045   Multi-User Chat (MUC)
           → group chats up to 500 members

XEP-0085   Chat State Notifications
           → typing indicators, recording indicators

XEP-0184   Message Delivery Receipts
           → sent / delivered ticks

XEP-0198   Stream Management
           → reliable delivery on bad networks
           → reconnect without losing messages
           → ACK at stanza level

XEP-0166   Jingle
           → voice and video call signaling

XEP-0357   Push Notifications
           → FCM/APNs bridge when user offline

XEP-0333   Chat Markers
           → read receipts (blue ticks)

XEP-0280   Message Carbons
           → sync messages across multiple devices
```

Three Stanza Types — Everything Is One of These

```
<!-- 1. Message — send content -->
<message from="kibuti@nexgate.com"
         to="juma@nexgate.com"
         type="chat"
         id="msg-001">
  <body>Habari yako!</body>
</message>

<!-- 2. Presence — announce availability -->
<presence from="kibuti@nexgate.com/android">
```

```
  <show>available</show>
  <status>I am here</status>
</presence>

<!-- 3. IQ (Info/Query) — request/response -->
<iq type="get" id="req-001">
  <query xmlns="jabber:iq:roster"/>
  <!-- asking for contact list -->
</iq>
```

---

4. Ejabberd Cluster — Two Nodes

Why Two Nodes Over One

Running a single Ejabberd node works technically. But one node means one point of failure. If that container crashes or the VPS reboots during a deployment — every connected user loses their session, every active call drops, every in-flight message is lost.

Two nodes change the picture completely:

```
Single node:
  Node 1 crashes
  → 100% of users disconnected
  → all active calls dropped
  → messages in-flight lost
  → users notice immediately

Two nodes:
  Node 1 crashes
  → 50% of users reconnect to Node 2 (seconds)
  → Node 2 was already running — no cold start
  → active calls on Node 2 unaffected
  → Ejabberd cluster detects Node 1 gone
  → routes everything to Node 2 automatically
  → most users experience a brief reconnect
    not a full outage
```

Two nodes also doubles the connection capacity:

```
One node  → ~500k-1M concurrent connections
Two nodes → ~1M-2M concurrent connections
Same cost increase as one extra container
```

Launch Plan — Same VPS, Two Containers

For NexGate launch, both nodes run on the same Hetzner VPS. This is the right starting point:

```
 Cheaper — one VPS bill not two
 Simpler — same Docker network, zero latency between nodes
 Enough — two containers on one VPS still gives redundancy
              against container crashes and restarts
 Learning — operate cluster on familiar single VPS first
⚠  VPS hardware failure → both nodes gone
    (acceptable risk at launch stage)
```

When to move to two VPS:

```
NexGate has paying users depending on uptime
VPS hardware failure = real revenue loss
At that point: Option B (two VPS) is worth the cost
```

What is Erlang Dist?

This is the mechanism that makes the two containers feel like one system.

Erlang was designed in 1986 for telecom — specifically for telephone switches that could never go down even when individual machines failed. The solution Ericsson built was **Erlang Distribution**: multiple Erlang nodes connected over a network, sharing a process registry, able to send messages between processes on different machines as if they were local.

```
Normal programming:
  Process on Machine A cannot directly talk to
  process on Machine B
  Need: HTTP, gRPC, message queue, shared DB
  Always an extra hop

Erlang distribution:
  Process on Node 1 sends message to process on Node 2
  Directly — like calling a local function
```

```
  No extra infrastructure
  No Redis, no RabbitMQ for this
  Just: node1_process ! { message_to, node2_process }
  Erlang runtime handles delivery across the network
```

Applied to Ejabberd:

```
Every connected user = one Erlang process (~2KB RAM)
Kibuti connected to Node 1 = process on Node 1
Juma connected to Node 2 = process on Node 2

Kibuti sends "Habari" to Juma:

Node 1 (Erlang):
  "Find Juma's process"
  Check local process registry → not here
  Check Node 2 via Erlang dist → FOUND
  Send message directly to Juma's process on Node 2
  Node 2 delivers to Juma's WebSocket

No Redis pub/sub
No RabbitMQ for this routing
No extra network hops
Microsecond latency between nodes
This is why Ejabberd routes at 2M concurrent
where Spring Boot WS needs Redis pub/sub
```

Erlang Cookie — The Cluster Password

```
Before two Erlang nodes trust each other
they must prove they belong to the same cluster

The shared secret = Erlang Cookie

Node 1 starts → "my cookie is: nexgate_erlang_cookie_xyz"
Node 2 starts → "my cookie is: nexgate_erlang_cookie_xyz"
Same cookie → they trust each other → cluster formed

Unknown node attempts to join:
```

```
  "my cookie is: wrong_cookie"
  → rejected → cannot join cluster

Rules:
  Same cookie on ALL nodes — mandatory
  Long random string — not a simple word
  Stored in HashiCorp Vault → injected as env variable
  Never committed to git
  Rotate periodically like any secret
```

How Nodes Discover and Join Each Other

```
Step 1 — EPMD (Erlang Port Mapper Daemon):
  Each Erlang node registers with EPMD on port 4369
  EPMD is like a local DNS for Erlang nodes
  "I am ejabberd@ejabberd-node1, listening on port X"

Step 2 — Node 2 finds Node 1:
  Node 2 asks EPMD on Node 1's host:
  "Where is ejabberd@ejabberd-node1?"
  EPMD responds with port number
  Node 2 connects directly

Step 3 — Cookie handshake:
  Node 2: "here is my cookie hash"
  Node 1: validates → matches → accept
  Erlang dist connection established

Step 4 — Join cluster:
  ejabberdctl join_cluster ejabberd@ejabberd-node1
  Nodes sync:
    MUC room state
    User session registry
    Mnesia tables (Ejabberd internal DB)
  Cluster ready □

In Docker — hostname is critical:
  Container hostname must match Erlang node name
  ejabberd@ejabberd-node1 → container hostname: ejabberd-node1
```

```
  Mismatch = nodes cannot find each other
```

Docker Compose — Two Nodes on Same VPS

```
ejabberd-node1:
  image: ghcr.io/processone/ejabberd:latest
  container_name: ejabberd-node1
  hostname: ejabberd-node1          # must match ERLANG_NODE
  restart: unless-stopped
  environment:
    - ERLANG_NODE=ejabberd@ejabberd-node1
    - ERLANG_COOKIE=${EJABBERD_ERLANG_COOKIE}  # from Vault
  ports:
    - "5222:5222"    # XMPP TCP
    - "5280:5280"    # WebSocket
    - "5285:5285"    # REST API (internal)
    - "1883:1883"    # MQTT
    - "4369:4369"    # EPMD (Erlang port mapper)
  volumes:
    - ./ejabberd/ejabberd.yml:/home/ejabberd/conf/ejabberd.yml
    - ./ejabberd/node1/data:/home/ejabberd/database
    - ./ejabberd/node1/logs:/home/ejabberd/logs
  networks:
    - nexgate-internal

ejabberd-node2:
  image: ghcr.io/processone/ejabberd:latest
  container_name: ejabberd-node2
  hostname: ejabberd-node2          # different hostname
  restart: unless-stopped
  environment:
    - ERLANG_NODE=ejabberd@ejabberd-node2
    - ERLANG_COOKIE=${EJABBERD_ERLANG_COOKIE}  # same cookie
  ports:
    - "5223:5222"    # different host ports
    - "5281:5280"
    - "5286:5285"
    - "4370:4369"
  volumes:
```

```
    - ./ejabberd/ejabberd.yml:/home/ejabberd/conf/ejabberd.yml
    - ./ejabberd/node2/data:/home/ejabberd/database
    - ./ejabberd/node2/logs:/home/ejabberd/logs
  depends_on:
    - ejabberd-node1
  networks:
    - nexgate-internal
```

How Traefik Load Balances Between Nodes

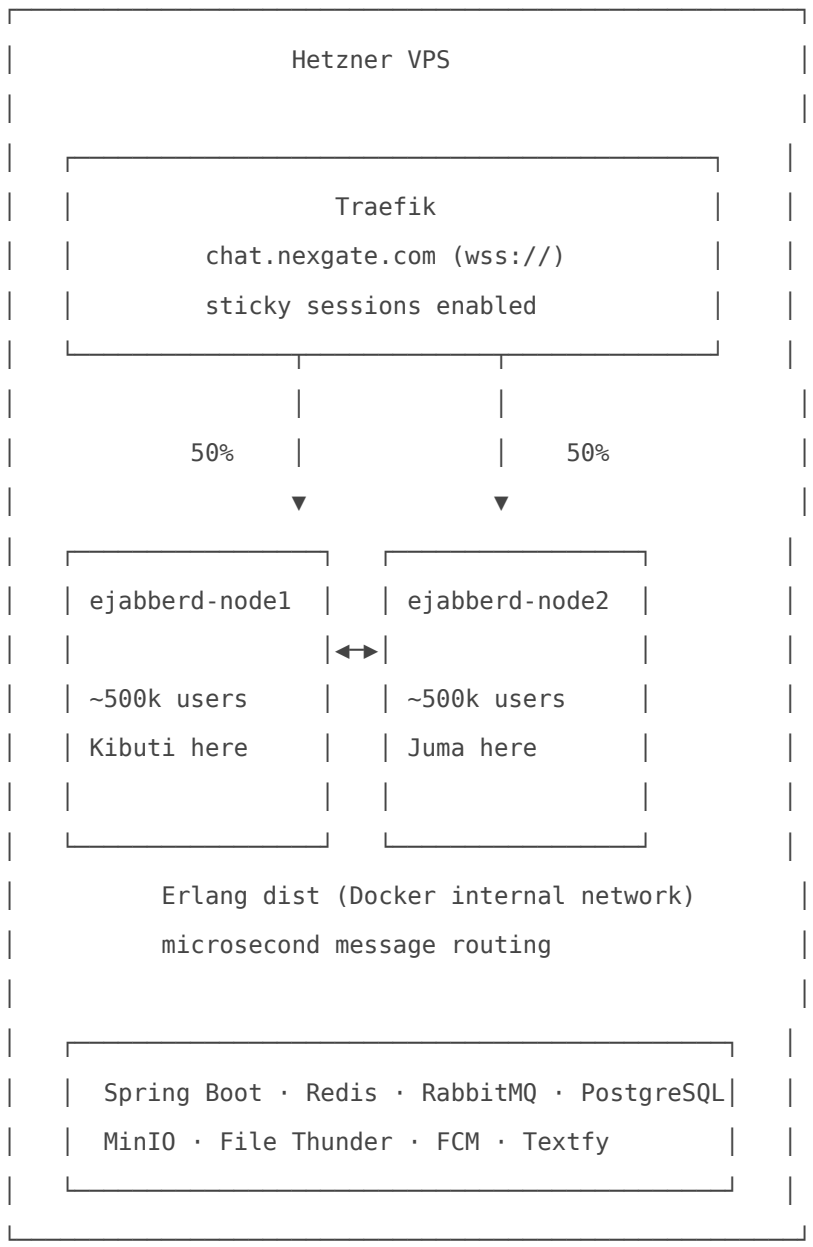
```
Traefik sits in front of both nodes:
chat.nexgate.com → Traefik → Node 1 or Node 2

Critical: WebSocket needs sticky sessions
Once a user connects to Node 1 — they must
always go to Node 1 for that session
(the WS connection lives on that node)

Traefik labels:
  sticky.cookie: true
  sticky.cookie.name: "ejabberd_node"

First connection:
  User hits chat.nexgate.com
  Traefik picks Node 1 (round robin)
  Sets cookie: ejabberd_node=node1
  User connects WebSocket to Node 1

Subsequent requests same session:
  Cookie present: ejabberd_node=node1
  Traefik always routes to Node 1
  WebSocket session stable □

Node 1 crashes:
  Cookie points to dead node
  Traefik detects Node 1 unhealthy
  Routes to Node 2
  User reconnects (brief disconnect)
  Node 2 was already running → fast reconnect
```

What Happens When One Node Goes Down

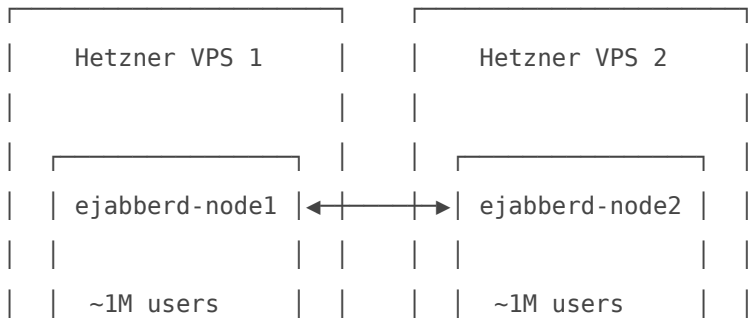
```
Scenario: ejabberd-node1 container crashes

Immediately:
  ~50% of users lose WebSocket connection
  Their apps detect disconnect
  Exponential backoff reconnect starts

Within seconds:
  Apps reconnect to chat.nexgate.com
  Traefik detects Node 1 unhealthy
  Routes all new connections to Node 2
  Users reconnect to Node 2

Stream Management (XEP-0198):
  Short disconnects (< 5 min): session resumable
  Users reconnect → Ejabberd resends missed stanzas
  No messages lost

Longer outage:
  RabbitMQ offline queue holds messages
  FCM push notifications already fired
  When user reconnects → queue drains
  Messages delivered

Calls during crash:
  WebRTC audio/video continues flowing
  (P2P or Coturn — not through Ejabberd)
  Signaling channel dropped
  Active calls: audio continues but
  call management (mute, end) needs reconnect

Node 2 (other 50%):
  Completely unaffected
  No interruption for their users
  Their calls continue perfectly
```

Cluster Architecture — Visual

Same VPS (Launch):



Two VPS (Growth stage):



```
|  └──────────────────┘  |    |  └──────────────────┘  |
|                        |    |                        |
└────────────────────────┘    └────────────────────────┘

        Erlang dist via Hetzner private network
        (free bandwidth between Hetzner VPS)
        low latency — same data center region
```

Mnesia — Ejabberd's Internal Database

```
Ejabberd uses Mnesia (Erlang's built-in DB)
for internal operational data:

What Mnesia stores:
  Active user sessions (who is connected where)
  MUC room membership + state
  Presence subscriptions (roster)
  Offline message buffer (short-term)

In a cluster:
  Mnesia replicates across nodes automatically
  Node 1 and Node 2 share same Mnesia data
  Node 1 updates session table → Node 2 sees it
  This is how Node 2 knows Juma is on Node 2
  and can route Kibuti's message correctly

Mnesia is NOT:
  A replacement for PostgreSQL
  Where NexGate messages are stored
  Where conversation history lives
  (that is all PostgreSQL via Spring Boot)

Mnesia is purely Ejabberd internal
NexGate Spring Boot never touches Mnesia
```

Scale Path Summary

```
Launch:
  1 VPS
  2 Docker containers (node1 + node2)
```

```
  Erlang dist over Docker internal network
  Traefik sticky sessions
  ~1M concurrent capacity
  Zero redundancy against VPS hardware failure
  □ Right for launch

Growth:
  2 VPS (Hetzner private network)
  1 container per VPS
  Erlang dist over private network (low latency)
  True hardware redundancy
  ~2M concurrent capacity
  VPS failure → other VPS serves all users
  □ Right when uptime = revenue

WeChat EA scale:
  3-5 VPS nodes
  Each node handles ~500k-1M users
  Erlang cluster routes everything
  Geographic distribution possible
  □ Right when NexGate is EA infrastructure
```

---

5. Connection Lifecycle

Full Connect ? Reconnect ? Disconnect Flow

```
App launches / user logs in:
     │
     ▼ POST /auth/login (Main Backend)
Receive two tokens:
  REST JWT     → HTTP API calls (7 days)
  XMPP Token   → Ejabberd connection (24 hours)
     │
     ▼ Connect WebSocket
wss://chat.nexgate.com/ws
Header: Authorization: Bearer {XMPP_TOKEN}
     │
```

```
     ▼ XMPP stream opened
<stream:stream to="nexgate.com"
               version="1.0">
     │
     ▼ Ejabberd → Spring Boot (sync HTTP auth)
POST /internal/ejabberd/auth
{ username: "usr-kibuti", token: "XMPP_TOKEN" }
Spring Boot:
  Check Redis cache first (< 5ms if cached)
  Validate JWT signature
  Check user not suspended
  Return 200 or 401
     │
     ▼ Auth success — Ejabberd sends features
<stream:features>
  <sm xmlns="urn:ietf:params:xml:ns:xmpp-session"/>
  <!-- Stream Management XEP-0198 -->
</stream:features>
     │
     ▼ Client enables Stream Management
<enable xmlns="urn:ietf:params:xml:ns:xmpp-sm:3"
        resume="true"/>
Ejabberd: <enabled id="session-abc" resume="true"/>
     │
     ▼ Client sends presence (I am online)
<presence/>
     │
Ejabberd:
  Registers kibuti@nexgate.com/android as ONLINE
  Publishes RabbitMQ: chat.presence.online
  Spring Boot: drain offline queue for kibuti
     │
Connection established □
App shows conversations, unread counts
```

Stream Management — Why It Matters for EA

```
Problem without Stream Management:
  Network drops (very common on EA mobile)
```

```
  TCP connection breaks
  In-flight messages LOST
  User reconnects — no idea what was missed

XEP-0198 Stream Management solution:
  Every stanza gets a sequence number
  Client ACKs received stanzas:
    <a xmlns="..." h="5"/>  ← "I received up to stanza 5"
  Server ACKs received stanzas the same way

  On reconnect:
    Client sends: <resume id="session-abc" h="5"/>
    Ejabberd knows: client got up to stanza 5
    Ejabberd resends: stanzas 6, 7, 8 (unacknowledged)
    Zero message loss

For EA mobile networks:
  Connection drops constantly (3G → 2G → WiFi)
  Stream Management means users never miss messages
  Even on unstable connections
  Critical for NexGate commerce DMs
  (missing an order negotiation message = lost sale)
```

Reconnection Strategy

```
Connection drops detected:
     |
App: exponential backoff reconnect
  Attempt 1: wait 1 second
  Attempt 2: wait 2 seconds
  Attempt 3: wait 4 seconds
  Attempt 4: wait 8 seconds
  Max wait:  30 seconds
     |
On reconnect:
  If session resumable (< 5 minutes offline):
    Resume: <resume id="session-abc" h="last_ack"/>
    Ejabberd resends missed stanzas
    No message loss □
```

```
  If session expired (> 5 minutes offline):
    Full re-auth with XMPP token
    Fetch conversation list from REST API
    Missed messages come from MAM (message archive)
    or from RabbitMQ offline queue drain
```

---

6. 1:1 Private DMs

Sending a Text Message

```
[Kibuti types "Habari" — taps Send]
     |
     ▼ App sends XMPP message stanza:
<message from="kibuti@nexgate.com/android"
         to="juma@nexgate.com"
         type="chat"
         id="msg-abc-123">
  <body>Habari</body>
  <request xmlns="urn:xmpp:receipts"/>
  <!-- Request delivery receipt XEP-0184 -->
  <nexgate xmlns="urn:nexgate:meta">
    <conv_id>conv-789</conv_id>
    <temp_id>local-xyz</temp_id>
    <level>NORMAL</level>
  </nexgate>
  <!-- NexGate custom namespace for app metadata -->
</message>
     |
App shows message as: pending □
     |
     ▼
[Ejabberd Node 1 — Kibuti's node]
  Receives stanza
  ACKs via Stream Management:
    <a h="6"/>   ← "I got your stanza"
  App: pending → sent ✓
```

```
      |
  Is Juma online?
    YES → Juma on Node 2:
            Erlang distributed message → Node 2
            Node 2 delivers to Juma's WS
    NO  → Store in offline queue
            XEP-0160 offline storage
            OR RabbitMQ (NexGate custom)
      |
  Fire RabbitMQ: chat.message.inbound
  (async, does not block delivery)
      |
      ▼
[Spring Boot Chat Service — async]
  Write to PostgreSQL
  Write to Redis hot cache
  Resolve offline escalation if needed
```

Receiving a Message

```
[Juma's app — connected on Node 2]
      |
Ejabberd Node 2 pushes stanza:
<message from="kibuti@nexgate.com"
         to="juma@nexgate.com"
         type="chat"
         id="msg-abc-123">
  <body>Habari</body>
  <request xmlns="urn:xmpp:receipts"/>
</message>
      |
Juma's app:
  Displays message in conversation
  Automatically sends delivery receipt:

<message from="juma@nexgate.com"
         to="kibuti@nexgate.com">
  <received xmlns="urn:xmpp:receipts"
            id="msg-abc-123"/>
```

```
</message>
     |
Ejabberd routes receipt to Kibuti
Kibuti's app: sent ✓ → delivered ✓✓
     |
Juma opens conversation:
<message from="juma@nexgate.com"
         to="kibuti@nexgate.com">
  <displayed xmlns="urn:xmpp:chat-markers:0"
             id="msg-abc-123"/>
</message>
     |
Kibuti's app: delivered ✓✓ → read ✓✓ (blue)
```

Rich Content Cards in DMs

```
NexGate extends XMPP with custom namespaces
for rich content (product cards, events etc)

Product card message:
<message from="kibuti@nexgate.com"
         to="juma@nexgate.com"
         type="chat"
         id="msg-card-001">
  <body>Angalia bidhaa hii</body>
  <nexgate-card xmlns="urn:nexgate:cards">
    <type>PRODUCT</type>
    <ref_id>prod-123</ref_id>
    <snapshot>
      <name>Samsung A15</name>
      <price>450000</price>
      <currency>TZS</currency>
      <image_url>...</image_url>
      <shop_name>TechStore</shop_name>
    </snapshot>
  </nexgate-card>
</message>

Receiving app:
```

```
  Detects nexgate-card element
  Renders rich card UI instead of plain text
  Tappable → deep links to product page
```

---

7. Group Chats

MUC — Multi User Chat

```
Group chats in Ejabberd use XEP-0045 (MUC)
Each group = a MUC room with its own JID:
  group-abc@conference.nexgate.com

Two group types in NexGate:
  PRIVATE:  closed, controlled membership
            not discoverable in search
            default when creating a group

  PUBLIC:   open, anyone can join via link
            discoverable in NexGate search
            explicit choice by creator
```

Creating a Group — Technical Flow

```
User creates group in app:
     |
     ▼ POST /chat/groups/create (Spring Boot)
{
  name: "Business Friends",
  type: "PRIVATE",          ← or PUBLIC
  description: "Dar founders discussion"
}
     |
Spring Boot:
  Create conversation record (type: GROUP)
  Call Ejabberd REST API:
    POST /api/create_room
```

```
    {
      name:    "group-abc",
      service: "conference.nexgate.com",
      options: {
        persistent:             true,
        public:                 false,   ← PRIVATE
        members_only:           true,
        allow_private_messages: false
      }
    }
  Creator auto-joined as OWNER
  Generate invite link token
  Return: { groupId, inviteLink }
```

Group Join Model — Two Mechanisms

```
NexGate principle:
  Nobody ends up in a group
  without actively choosing to join

  No forced direct add
  Two consent-based mechanisms only
```

Mechanism 1 — Consent DM Invitation

```
Admin selects people from contacts/followers/
commerce relationships (NOT strangers)

Each selected person receives a DM:

Spring Boot → Ejabberd REST API:
POST /api/send_message
{
  from: "system@nexgate.com",
  to:   "juma@nexgate.com",
  extra: {
    type:         "GROUP_INVITATION",
    group_id:     "group-abc",
    group_name:   "Business Friends",
```

```
    group_type:   "PRIVATE",
    member_count: 47,
    description:  "Dar founders discussion",
    invited_by:   "Kibuti Mwangi",
    expires_in:   "48h"
  }
}

XMPP stanza (NexGate custom namespace):
<message from="system@nexgate.com"
         to="juma@nexgate.com"
         type="chat">
  <body>You have been invited to join a group</body>
  <nexgate-group-invite xmlns="urn:nexgate:group:1">
    <group_id>group-abc</group_id>
    <group_name>Business Friends</group_name>
    <group_type>PRIVATE</group_type>
    <member_count>47</member_count>
    <invited_by>Kibuti Mwangi</invited_by>
    <expires_at>2026-07-15T10:00:00Z</expires_at>
  </nexgate-group-invite>
</message>

Recipient app renders:
┌──────────────────────────────────────────┐
│ □□Group Invitation                       │
│                                          │
│ Kibuti Mwangi invited you to join:       │
│ □□□Business Friends                      │
│ 47 members · Private Group               │
│ "Dar founders discussion"                │
│                                          │
│ [Accept & Join]      [Decline]           │
└──────────────────────────────────────────┘

If Accept:
  App sends: POST /chat/groups/group-abc/join
  Spring Boot: ejabberdctl add_member group-abc juma
  Juma is now a group member □
```

```
If Decline:
  POST /chat/groups/group-abc/decline
  Not added □
  Kibuti NOT notified (privacy)

If Ignored (48h passes):
  Invitation auto-expired
  Auto-declined silently □
```

Mechanism 2 — Invite Link

```
Admin generates link:
  nexgate.app/join/abc-xyz-def

Person taps link → sees group preview:

PRIVATE GROUP:
  App calls: GET /chat/groups/preview/abc-xyz-def
  Shows preview + [Request to Join] button
  Spring Boot creates join request
  Admin sees request in group management
  Admin approves → member □
  Admin declines → person not notified

PUBLIC GROUP:
  App calls: GET /chat/groups/preview/abc-xyz-def
  Shows preview + [Join Group] button
  Tap → instant member □
  No approval needed

Link settings (stored in DB):
  expires_at:  nullable (never if null)
  max_joins:   nullable (unlimited if null)
  revoked_at:  nullable (active if null)

On revoke:
  Old token deleted
  New token generated
  Old link: "This invite link is no longer valid"
```

Sending a Group Message

```
<!-- Member sends to group -->
<message to="group-abc@conference.nexgate.com"
         type="groupchat"
         id="gmsg-001">
  <body>Hello everyone!</body>
</message>

<!-- Ejabberd MUC reflects back with room JID -->
<message from="group-abc@conference.nexgate.com/Kibuti"
         type="groupchat"
         id="gmsg-001">
  <body>Hello everyone!</body>
</message>
```

```
Ejabberd MUC fan-out:
  Receives from Kibuti
  Broadcasts to ALL room members simultaneously
  Each member's WS gets the stanza
  Erlang handles fan-out natively
  No Redis pub/sub needed
  Spring Boot persists via RabbitMQ event
```

Group Roles — Ejabberd MUC Mapping

```
Ejabberd MUC role   NexGate role    Permissions
──────────────────────────────────────────────────
owner               OWNER           everything
                                    delete group
                                    transfer ownership
admin               ADMIN           manage members
                                    delete any message
                                    pin messages
                                    change group info
moderator           MODERATOR       mute members
                                    remove members
participant         MEMBER          send messages
```

```
                                    delete own messages
visitor             READ_ONLY       view only
                                    (announcement mode)
```

Fan-out Strategy

```
Groups up to 500 members:
  Ejabberd MUC native fan-out
  All members get stanza in real time
  Erlang handles it — no extra logic

Groups approaching 500:
  Recommend switching to PUBLIC group
  with announcement mode (admins only post)
  Better for large audiences
  No broadcast channels needed
  (VP Feed covers mass content distribution)
```

Group Invite Link DB Schema

```
group_invite_links
──────────────────────────────────────────────

link_id           UUID
group_id          UUID
token             TEXT      unique random token
created_by        UUID      admin userId
expires_at        TIMESTAMPTZ  nullable
max_joins         INT          nullable
join_count        INT          default 0
revoked_at        TIMESTAMPTZ  nullable
created_at        TIMESTAMPTZ
```

---

8. Chat States — Typing & Recording

XEP-0085 — Built Into Ejabberd

```
No custom backend code needed
Ejabberd routes chat state stanzas automatically
Spring Boot never sees them (not persisted)
Pure real-time ephemeral signals
```

All States and When App Sends Them

```
State         App sends when              Recipient sees
──────────────────────────────────────────────────────────
composing     user starts typing          "Kibuti is typing..."
paused        user stopped typing         indicator disappears
              (3s no keystroke)
active        user opened conversation    no indicator
              but not typing
inactive      user left conversation      no indicator
              screen (10s elapsed)
gone          user closed conversation    no indicator
recording     user holding mic button     "Kibuti is recording..."
```

Stanzas

```
<!-- User starts typing -->
<message from="kibuti@nexgate.com/android"
         to="juma@nexgate.com"
         type="chat">
  <composing xmlns="http://jabber.org/protocol/chatstates"/>
</message>

<!-- User paused typing -->
<message from="kibuti@nexgate.com/android"
         to="juma@nexgate.com"
         type="chat">
  <paused xmlns="http://jabber.org/protocol/chatstates"/>
</message>

<!-- User is recording voice note -->
<message from="kibuti@nexgate.com/android"
         to="juma@nexgate.com"
```

```
          type="chat">
  <composing xmlns="http://jabber.org/protocol/chatstates"/>
  <recording xmlns="urn:nexgate:states"/>
  <!-- NexGate custom namespace for recording state -->
</message>

<!-- User stopped recording (sent or cancelled) -->
<message from="kibuti@nexgate.com/android"
          to="juma@nexgate.com"
          type="chat">
  <paused xmlns="http://jabber.org/protocol/chatstates"/>
</message>
```

Throttling — Don't Spam the Network

```
Wrong (naive) approach:
  Send composing stanza on every single keystroke
  100 keystrokes = 100 stanzas
  Wastes bandwidth — bad for EA data bundles

Correct approach:
  User starts typing → send composing once
  Keep typing → resend composing every 3 seconds
  User stops → wait 3 seconds → send paused
  Total: ~1 stanza per 3 seconds while typing
  Much more efficient

Mobile dev implements this with a timer:
  startTypingTimer() → fires composing once
  resetTimer() on each keystroke
  onTimerExpire() → send paused
```

Group Chat States

```
In group chats:
  Same stanzas — sent to room JID instead of personal JID
  Ejabberd MUC broadcasts to all room members

UI handling when multiple people type:
```

```
  1 person:   "Juma is typing..."
  2 people:   "Juma and Amina are typing..."
  3+ people:  "3 people are typing..."

App collects composing events from room
Tracks: Set<userId> currentlyTyping
Renders string based on set size
```

---

9. Message Receipts

Three Tick States

```
✓   Sent         Server received + stored stanza
                 Stream Management ACK received

✓✓  Delivered    Recipient device received stanza
                 XEP-0184 receipt returned

✓✓  Read         Recipient opened conversation
(blue)           XEP-0333 chat marker returned
```

XEP-0184 — Delivery Receipt

```
<!-- Kibuti requests receipt in original message -->
<message id="msg-001" ...>
  <body>Habari</body>
  <request xmlns="urn:xmpp:receipts"/>
</message>

<!-- Juma's app automatically responds on delivery -->
<message from="juma@nexgate.com"
         to="kibuti@nexgate.com">
  <received xmlns="urn:xmpp:receipts"
            id="msg-001"/>
  <!-- id references the original message -->
```

```
</message>

Kibuti's app receives this:
  → updates message msg-001 status: DELIVERED
  → shows ✓✓
```

XEP-0333 — Chat Markers (Read Receipt)

```
<!-- Juma opens conversation — app sends displayed marker -->
<message from="juma@nexgate.com"
         to="kibuti@nexgate.com"
         type="chat">
  <displayed xmlns="urn:xmpp:chat-markers:0"
             id="msg-001"/>
  <!-- marks msg-001 and all previous as read -->
</message>

Kibuti's app receives this:
  → updates msg-001 and all before: READ
  → shows ✓✓ blue
```

Group Message Receipts

```
In groups: receipts work per member

Message sent to group of 5:
  Each member's delivery → individual receipt
  All 5 delivered → show ✓✓

Read receipts in groups:
  Show count: "Read by 3"
  Tap to see who read it
  (WhatsApp same pattern)

Spring Boot aggregates:
  Stores each receipt in message_receipts table
  Computes: delivered_count, read_count
  Returns to sender on request
```

11. Presence System

How Presence Works in XMPP

```
Presence is built into XMPP protocol
No custom implementation needed
Ejabberd handles all presence routing

User connects:
  Sends: <presence/>
  Ejabberd broadcasts to all contacts
  who have presence subscription

User disconnects:
  Ejabberd auto-sends: <presence type="unavailable"/>
  All subscribed contacts notified

This is fully automatic
Spring Boot only needs to:
  Listen to RabbitMQ presence events
  Update last_seen_at in PostgreSQL
  Cache presence in Redis (for fast lookup)
```

Presence States

```
<!-- Available (online) -->
<presence from="kibuti@nexgate.com/android"/>

<!-- Away (phone locked / app backgrounded) -->
<presence from="kibuti@nexgate.com/android">
  <show>away</show>
</presence>

<!-- Do Not Disturb -->
<presence from="kibuti@nexgate.com/android">
  <show>dnd</show>
```

```
  <status>In a meeting</status>
</presence>

<!-- Offline -->
<presence from="kibuti@nexgate.com/android"
          type="unavailable"/>
```

Last Seen

```
When user goes offline:
  Ejabberd fires: chat.presence.offline (RabbitMQ)
  Spring Boot:
    Update users.last_seen_at = now
    Remove presence:{userId} from Redis

When contact opens chat with offline user:
  App requests: GET /chat/users/{userId}/presence
  Spring Boot returns:
    { status: "offline", lastSeenAt: "2026-07-02T08:30:00Z" }
  App shows: "Mwisho kuonekana leo saa 2:30"

Privacy settings (Spring Boot enforces):
  EVERYONE   → anyone can see last seen
  CONTACTS   → only conversation partners
  NOBODY     → hide last seen from all
```

Online Indicator in Conversation

```
How app shows "online" in DM header:

Option 1 — Subscribe to presence (XMPP native):
  App sends presence subscription to contact
  Contact auto-notified when they come online
  Ejabberd handles real-time push

Option 2 — Poll on conversation open:
  GET /chat/users/{userId}/presence
  Check Redis: presence:{userId} exists? → online
  Simple, no subscription management
```

```
NexGate recommendation: Option 2
  Simpler to implement
  No subscription state to manage
  Polling on conversation open is fine
  (user only cares when they're IN the conversation)
```

---

12. Voice Calls — Deep Dive

Complete Component Map

```
┌──────────────────────────────────────────────────────┐
│                   Voice Call                         │
│                                                      │
│  Signaling:    Ejabberd Jingle (XEP-0166)            │
│                "who calls who, exchange network info" │
│                                                      │
│  Discovery:    STUN (built into Ejabberd)            │
│                "find your public IP behind NAT"      │
│                                                      │
│  Relay:        Coturn TURN server                    │
│                "relay audio when P2P impossible"     │
│                EA carrier NAT blocks most P2P        │
│                                                      │
│  Transport:    WebRTC PeerConnection                 │
│                "actual audio stream between devices" │
│                                                      │
│  Codec:        Opus                                  │
│                "compress audio for EA networks"      │
│                "adaptive 6kbps (2G) → 64kbps (WiFi)" │
│                                                      │
│  Encryption:   SRTP (built into WebRTC)              │
│                "all audio encrypted end to end"      │
└──────────────────────────────────────────────────────┘
```

ICE — How Devices Find Each Other

```
ICE = Interactive Connectivity Establishment
The algorithm that finds the best path between devices

Step 1 — Gather candidates (both devices do this):
  Local candidate:
    192.168.1.5:54321  ← local network IP
  STUN candidate:
    41.188.xxx.xxx:54321  ← public IP (Vodacom/Airtel IP)
    Found by asking STUN server: "What is my public IP?"
  TURN candidate:
    turn.nexgate.com:3478  ← relay fallback

Step 2 — Exchange candidates:
  Both share their candidate lists
  Via Ejabberd Jingle stanzas

Step 3 — Try connections (priority order):
  1. Direct local network (same WiFi) → fastest
  2. Direct P2P via public IPs → good
  3. TURN relay → always works, higher latency

Step 4 — Use best working path:
  Call starts on winning candidate
  Can switch mid-call if network changes

EA reality:
  Direct local → rarely (different networks)
  Direct P2P → sometimes (depends on carrier)
  TURN relay → most common on Vodacom/Airtel/Tigo
```

Full Voice Call Sequence

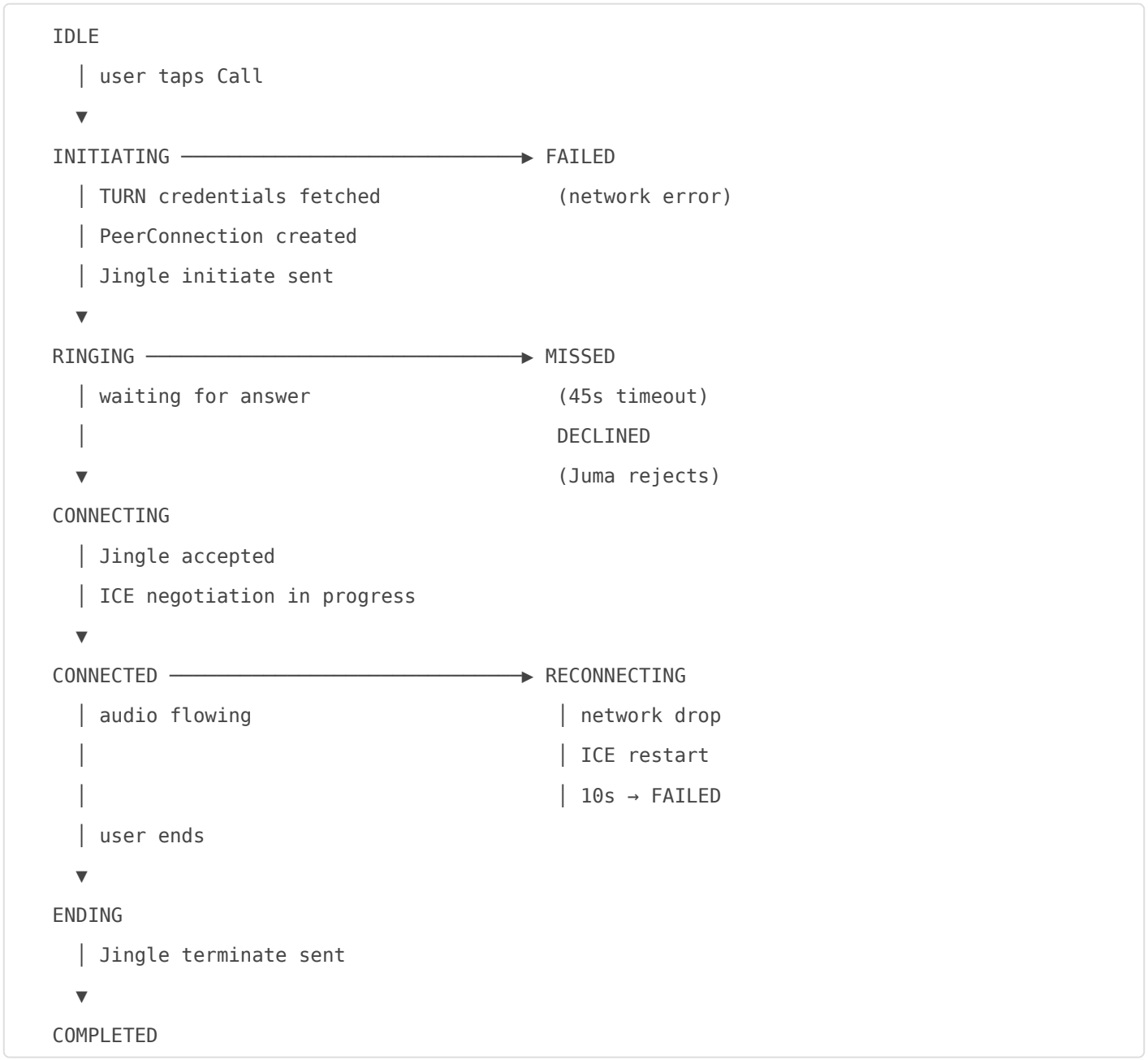
```
[Kibuti taps "Call Juma"]
     │
     ▼ App: GET /chat/calls/turn-credentials
Spring Boot generates HMAC TURN credentials:
{
  iceServers: [
    { urls: "stun:chat.nexgate.com:3478" },
```

```
    { urls: "turn:turn.nexgate.com:3478",
      username: "usr-kibuti:1751500000",
      credential: "hmac_sha1_token" }
  ]
}
     |
     ▼ App initializes WebRTC PeerConnection
Config: iceServers from above
Add audio track:
  Opus codec
  echoCancellation: true
  noiseSuppression: true
  autoGainControl: true
     |
     ▼ App creates SDP offer
WebRTC generates offer describing:
  Codecs supported (Opus preferred)
  Audio capabilities
  Security parameters (DTLS)
     |
     ▼ App sends Jingle session-initiate
<iq from="kibuti@nexgate.com/android"
    to="juma@nexgate.com"
    type="set"
    id="call-001">
  <jingle xmlns="urn:xmpp:jingle:1"
          action="session-initiate"
          sid="sid-abc-123"
          initiator="kibuti@nexgate.com/android">
    <content name="audio" creator="initiator">
      <description xmlns="urn:xmpp:jingle:apps:rtp:1"
                   media="audio">
        <payload-type id="111"
                      name="opus"
                      clockrate="48000"
                      channels="2"/>
      </description>
      <transport xmlns="urn:xmpp:jingle:transports:ice-udp:1"
                 ufrag="abc"
```

```
                pwd="password123">
        <candidate component="1"
                   foundation="1"
                   type="host"
                   ip="192.168.1.5"
                   port="54321"
                   priority="2130706431"/>
        <candidate component="1"
                   foundation="2"
                   type="srflx"
                   ip="41.188.xxx.xxx"
                   port="54321"
                   priority="1694498815"/>
      </transport>
    </content>
  </jingle>
</iq>
     |
     ▼ Ejabberd routes to Juma
Ejabberd fires RabbitMQ: chat.call.initiated
Spring Boot:
  Creates call record (status: RINGING)
  If Juma offline → FCM HIGH priority:
    { type: INCOMING_CALL, callId: "sid-abc-123",
      callerName: "Kibuti", callType: VOICE }
     |
[Juma's phone rings — incoming call screen]
Juma taps Answer
     |
     ▼ Juma: get TURN credentials
Initialize PeerConnection (same config)
Set remote description (Kibuti's SDP)
Create SDP answer
Gather own ICE candidates
     |
     ▼ Juma sends Jingle session-accept
<iq from="juma@nexgate.com/android"
    to="kibuti@nexgate.com/android"
    type="set">
```

```
  <jingle action="session-accept"
          sid="sid-abc-123">
    <!-- Juma's SDP answer + ICE candidates -->
  </jingle>
</iq>
     |
     ▼ Ejabberd routes to Kibuti
Kibuti's app:
  Sets remote description (Juma's SDP)
  ICE negotiation completes
  Best path selected (likely TURN on EA networks)
     |
CALL LIVE □□
Opus audio flowing between devices
     |
RTCP monitors quality every 200ms:
  Reports: packet loss, jitter, RTT, bandwidth
  Opus adapts bitrate automatically:
    64kbps → 32kbps → 16kbps → 8kbps → 6kbps
  Never drops — always degrades gracefully
     |
Kibuti taps End
     |
     ▼ Jingle session-terminate
<iq type="set">
  <jingle action="session-terminate"
          sid="sid-abc-123">
    <reason><success/></reason>
  </jingle>
</iq>
     |
Ejabberd fires RabbitMQ: chat.call.ended
Spring Boot:
  Update call record:
    status: COMPLETED
    ended_at: now
    duration_seconds: 247
    relay_used: true
    end_reason: NORMAL
```

Call State Machine



Opus Codec Ladder

Network	Bitrate	What it sounds like
WiFi / 4G strong	64 kbps	HD voice, crystal clear
4G normal	32 kbps	Clear, natural voice
3G	16 kbps	Good, slight compression
2G / Edge	8 kbps	Robotic but intelligible
Barely alive	6 kbps	Minimum — still connected

```
Opus switches between these automatically
based on RTCP feedback every 200ms
Mobile dev configures nothing — it just works

Key Opus features for EA:
  inbandfec: true    → Forward Error Correction
                       recovers from packet loss
                       without retransmit
  usedtx: true       → Discontinuous Transmission
                       silence = no packets sent
                       saves bandwidth during pauses
  stereo: false      → Mono only for calls
                       half the bitrate vs stereo
```

---

13. Video Calls — Deep Dive

Additional Components vs Voice

```
Voice call +
  Video codec (H.264 primary)
  Camera capture (front/rear switchable)
  Video rendering (remote + local preview)
  Higher bandwidth requirement
  Higher CPU on device
  More Coturn relay bandwidth if P2P fails
```

H.264 — Why for EA

```
H.264 (AVC) chosen because:

Hardware acceleration:
  Every phone since 2013 has H.264 hardware encoder
  Including Tecno Spark, Infinix Hot (dominant in EA)
  Hardware encoder = GPU does the work
  Battery impact: LOW
```

```
  CPU: barely used

Software encoding (VP8, VP9, AV1):
  CPU does all encoding work
  On low-end EA phones: hot, slow, battery drain
  10 minutes of video = significant battery cost
  Users notice and complain

H.264 at low bitrates:
  360p @ 400kbps → works on 3G
  240p @ 150kbps → works on 2G
  Quality acceptable for face-to-face conversation
```

Video Resolution Ladder

```
Device tier + network → resolution selected:

Device      Network      Resolution   FPS    Bitrate
──────────────────────────────────────────────────────
Any         WiFi         720p         30     1.5 Mbps
Any         4G strong    480p         24     800 kbps
Any         3G           360p         15     400 kbps
Any         2G           240p         10     150 kbps
Any         Very poor    AUDIO ONLY   —      Opus only
──────────────────────────────────────────────────────

Degradation order (call never drops):
  1. Reduce color depth
  2. Reduce resolution (720→480→360→240)
  3. Reduce frame rate (30→24→15→10)
  4. Reduce audio bitrate
  5. Disable video completely → audio only
  6. Audio minimum (6kbps Opus)

Upgrade is conservative:
  Wait 5 seconds of stable improved bandwidth
  Then upgrade one step (e.g. 360p → 480p)
  Prevents quality flapping on unstable networks
```

Jingle for Video — Two Content Blocks

```
<jingle action="session-initiate" sid="vid-001">

  <!-- Audio block — always present -->
  <content name="audio" creator="initiator">
    <description media="audio">
      <payload-type id="111" name="opus"
                    clockrate="48000"/>
    </description>
    <transport ...>
      <candidate .../> <!-- ICE candidates -->
    </transport>
  </content>

  <!-- Video block — added for video calls -->
  <content name="video" creator="initiator">
    <description media="video">
      <!-- H.264 preferred -->
      <payload-type id="96" name="H264"
                    clockrate="90000"/>
      <!-- VP8 as fallback -->
      <payload-type id="97" name="VP8"
                    clockrate="90000"/>
    </description>
    <transport ...>
      <candidate .../>
    </transport>
  </content>

</jingle>
```

Camera UI Features

```
Mobile dev implements:

Local preview (Picture-in-Picture):
  Small corner window showing your own camera
  Standard in all video call UIs
```

```
Switch camera:
  Front → Rear → Front toggle
  WebRTC: videoCapturer.switchCamera()

Camera off (privacy):
  videoTrack.setEnabled(false)
  Remote sees: black screen or avatar
  Audio continues

Auto-disable video on battery:
  Monitor: battery < 20% AND on Coturn relay
  Show warning: "Battery low — switching to audio only"
  Disable video track
  Continue audio call
```

---

10. Message Interactions

All message interactions use standard XMPP XEPs. Ejabberd routes stanzas automatically. Spring Boot validates rules and persists via RabbitMQ.

XEP Overview

```
Feature          XEP          Status      Ejabberd
──────────────────────────────────────────────────
Edit message     XEP-0308     Stable □   auto routed
Delete message   XEP-0424     Stable □   auto routed
Reactions        XEP-0444     Stable □   auto routed
Forwarding       XEP-0297     Stable □   auto routed
Reply/Quote      XEP-0461     Exp ⚠□      auto routed
Stable IDs       XEP-0359     Stable □   auto assigned
```

Edit — XEP-0308

```
<!-- Kibuti edits his sent message -->
<message from="kibuti@nexgate.com"
         to="juma@nexgate.com"
```

```
          type="chat"
          id="edit-002">
    <body>Hello Juma, how is business today?</body>
    <replace xmlns="urn:xmpp:message-correct:0"
             id="server-stable-id-abc"/>
    <!-- references original by stanza-id (XEP-0359) -->
  </message>
```

```
  Rules:
    Only original sender can edit □
    Text messages only □
    Within 15 minutes of sending □
    Shows "Edited" label after □
    Commerce cards: NOT editable □
    System messages: NOT editable □

  Spring Boot on receiving edit event (RabbitMQ):
    Validate author + time window
    Update messages.body
    Update messages.edited_at
    Increment messages.edit_count
```

Delete — XEP-0424

```
  <!-- Delete for everyone -->
  <message from="kibuti@nexgate.com"
           to="juma@nexgate.com"
           type="chat"
           id="retract-003">
    <apply-to xmlns="urn:xmpp:fasten:0"
              id="server-stable-id-abc">
      <retract xmlns="urn:xmpp:message-retract:1"/>
    </apply-to>
  </message>
```

```
  Delete for me:
    No stanza needed
    Local REST call only
    POST /chat/messages/{id}/delete { scope: SELF }
```

```
  Recipient unaffected

Delete for everyone:
  XEP-0424 retraction stanza
  Within 15 minutes only
  Commerce cards: NOT deletable □
  System messages: NOT deletable □
  Recipient sees: "This message was deleted"
  Nothing hard-deleted from PostgreSQL (audit trail)
```

Reactions — XEP-0444

```
<!-- Add reaction -->
<message from="kibuti@nexgate.com"
         to="juma@nexgate.com"
         type="chat">
  <reactions xmlns="urn:xmpp:reactions:0"
             id="server-stable-id-abc">
    <reaction>□□/reaction>
  </reactions>
</message>

<!-- Remove reaction (empty = removed) -->
<message from="kibuti@nexgate.com"
         to="juma@nexgate.com"
         type="chat">
  <reactions xmlns="urn:xmpp:reactions:0"
             id="server-stable-id-abc">
  </reactions>
</message>
```

```
Rules:
  One reaction per user per message □
  Change: send new emoji (replaces) □
  Remove: send empty reactions element □
  Commerce cards: reactions ALLOWED □
  System messages: reactions NOT allowed □
  Launch emoji set: ❤□ □□□□□□□□□□
```

Forwarding — XEP-0297

```
<!-- Kibuti forwards Juma's message to Alice -->
<message from="kibuti@nexgate.com"
         to="alice@nexgate.com"
         type="chat"
         id="fwd-001">
  <body>Check this out</body>
  <forwarded xmlns="urn:xmpp:forward:0">
    <delay xmlns="urn:xmpp:delay"
           stamp="2026-07-13T10:32:00Z"/>
    <message from="juma@nexgate.com"
             to="kibuti@nexgate.com"
             type="chat">
      <body>Hello everyone!</body>
    </message>
  </forwarded>
  <nexgate-forward xmlns="urn:nexgate:forward">
    <original_sender_name>Juma Mwangi</original_sender_name>
    <forward_chain>1</forward_chain>
  </nexgate-forward>
</message>
```

```
Rules:
  Max 5 conversations per forward action □
  Chain 1:   "Forwarded from Juma Mwangi"
  Chain 2-4: "Forwarded"
  Chain 5+:  "Forwarded many times" (warning)
  Media: references original fileId — no re-upload □
  Custom price offers: NOT forwardable □
  Order/payment records: NOT forwardable □
```

Reply — XEP-0461

```
<!-- Reply to a specific message -->
<message from="juma@nexgate.com"
         to="kibuti@nexgate.com"
         type="chat"
         id="reply-001">
```

```
  <body>Thanks, appreciate it!</body>
  <reply xmlns="urn:xmpp:reply:0"
         to="kibuti@nexgate.com"
         id="server-stable-id-abc"/>
</message>
```

```
Renders as:

┌──────────────────────────────────┐
│ ┌──────────────────────────┐     │
│ │ Kibuti                   │     │  ← quoted
│ │ Hello Juma!              │     │
│ └──────────────────────────┘     │
│ Thanks, appreciate it!           │
└──────────────────────────────────┘


Tap quote → scrolls to original message
```

---

14. Audio ? Video Switching & Screen Share

Switch Audio ? Video During Call

```
Call starts as voice only
User taps camera button during call
No hang up needed — same WebRTC session

Kibuti enables camera:
  Creates video track (H.264)
  Adds to existing PeerConnection
  Sends Jingle content-add stanza:
```

```
<iq from="kibuti@nexgate.com/android"
    to="juma@nexgate.com"
    type="set">
  <jingle xmlns="urn:xmpp:jingle:1"
          action="content-add"
```

```
          sid="sid-abc-123">
    <!-- sid = SAME session as voice call -->
    <content name="video" creator="initiator">
      <description media="video">
        <payload-type id="96" name="H264"
                      clockrate="90000"/>
      </description>
      <transport .../>
    </content>
  </jingle>
</iq>
```

```
Juma accepts:
  Jingle action="content-accept"
  Video starts flowing — same TURN relay □
  Audio uninterrupted during upgrade □

Switch back (video → audio):
  Jingle action="content-remove"
  Removes video content block
  Audio continues

Auto-downgrade (network-triggered):
  RTCP detects bandwidth too low
  App sends content-remove automatically
  Banner: "Video disabled — poor network"
  Resumes when network improves
```

Screen Sharing

```
Screen share = special video track
Instead of camera → captures device screen
Same H.264 encoding
Lower frame rate (5-15fps — screen changes slowly)

Android:   MediaProjection API
iOS:       ReplayKit broadcast extension

Start screen share:
```

```
  User taps screen share icon during call
  System permission dialog appears:
    "Allow NexGate to capture your screen?"
  User accepts
  Screen capture starts

Jingle stanza (adds screen content block):
```

```
<jingle action="content-add"
        sid="sid-abc-123">
  <content name="screen" creator="initiator">
    <description media="video">
      <payload-type id="96" name="H264"
                    clockrate="90000"/>
    </description>
    <transport .../>
  </content>
</jingle>
```

```
During screen share:
  Remote side sees: screen (large) + face (PiP)
  Local side sees: "Sharing screen" banner
  Camera optional: can keep or disable

Stop screen share:
  Jingle content-remove (screen)
  Returns to normal video/audio call

EA network consideration:
  Screen content is mostly static
  H.264 compresses static content very well
  720p screen at ~300kbps (vs 720p camera at 1.5Mbps)
  Works on 3G for text/document sharing □
```

---

15. Group Calls

Why LiveKit for Group Calls

```
1:1 call:
  P2P or Coturn relay
  Two devices, one path
  No server media processing

Group call (3+ people):
  Cannot P2P to everyone simultaneously
  Kibuti uploads 1 stream to LiveKit
  LiveKit forwards to all other participants
  Each participant uploads once → downloads N-1
  SFU = Selective Forwarding Unit

LiveKit already deployed for Audio Spaces
Same Docker container
Same Coturn relay reused
Zero new infrastructure □
```

Group Call Flow

```
Kibuti starts group call from group chat:
     │
     ▼ POST /chat/calls/group/start
Spring Boot:
  Create LiveKit room: group-call-{callId}
  Generate token per participant:
    canPublish: true
    canSubscribe: true
  Return tokens + LiveKit WS URL
     │
Jingle session-initiate sent to all group members:
```

```
<message from="system@nexgate.com"
         to="juma@nexgate.com"
         type="chat">
  <nexgate-call xmlns="urn:nexgate:call:1">
    <type>GROUP_CALL_JOIN_INFO</type>
    <call_id>call-xyz</call_id>
    <call_type>VIDEO</call_type>
    <livekit_url>wss://livekit.nexgate.com</livekit_url>
```

```
    <livekit_token>eyJ...</livekit_token>
    <room_id>group-call-xyz</room_id>
    <expires_in>300</expires_in>
  </nexgate-call>
</message>
```

```
Each member receives → phone rings
Members who join → connect WebRTC to LiveKit
LiveKit SFU forwards all streams □

[LiveKit SFU]
  Kibuti stream ──► forwarded to Juma + Alice
  Juma stream   ──► forwarded to Kibuti + Alice
  Alice stream  ──► forwarded to Kibuti + Juma
```

EA Network Limits for Group Calls

```
Group voice (audio only, Opus):
  3 people: each downloads 64kbps → works on 3G □
  5 people: each downloads 128kbps → works on 3G □
  8 people: each downloads 224kbps → needs 4G ⚠□

Group video (H.264 + Opus):
  3 people: each downloads 800kbps → needs 4G ⚠□
  4 people: each downloads 1.2Mbps → needs strong 4G ⚠□
  5+ people: reduce to active speaker only □

Max participants shown:
  Voice: up to 8 (3G compatible)
  Video: up to 4 feeds simultaneously
  5th+ person: audio tile only (no video feed)
  Active speaker highlighted (larger tile)
```

Simulcast — EA Network Diversity

```
Each participant uploads 3 quality versions:
  Low:    180p + Opus 16kbps
  Medium: 360p + Opus 32kbps
```

```
  High:   720p + Opus 64kbps

LiveKit delivers appropriate quality per receiver:
  Receiver on 2G → low quality streams
  Receiver on WiFi → high quality streams
  Each receiver gets quality their network allows
  Independently per stream

Result:
  Good network user sees HD video
  Poor network user sees low quality
  Everyone stays in the call 
  No one's bad network drops everyone else
```

---

16. Offline Handling

Three Layers of Offline Delivery

```
Layer 1 — Ejabberd XEP-0160 (offline storage):
  User disconnects mid-session
  Ejabberd stores pending stanzas
  On reconnect: delivers immediately
  Covers: short disconnections (seconds to minutes)

Layer 2 — RabbitMQ queue:
  User has been offline longer
  Spring Boot queues messages
  On reconnect: Chat Service drains queue
  Priority order: CRITICAL → IMPORTANT → NORMAL
  Covers: hours to days offline

Layer 3 — FCM + Textfy (notifications):
  Wakes device even when completely offline
  User sees notification → opens app
  Triggers Layer 1 + 2 delivery
  Covers: device asleep, app killed
```

FCM for Calls (Special Case)

```
If Juma is offline when Kibuti calls:

Spring Boot sends FCM HIGH priority:
{
  type: "INCOMING_CALL",
  callId: "sid-abc-123",
  callerName: "Kibuti Mwangi",
  callerAvatar: "https://...",
  callType: "VOICE",
  turnCredentials: { ... }  ← included for fast answer
}
     |
FCM wakes Juma's phone
App shows full-screen incoming call UI
(even if app was completely killed)
     |
Juma taps Answer:
  App already has TURN credentials
  Immediately creates PeerConnection
  Sends Jingle session-accept
  No extra round trip to get credentials
  Faster answer time □

Call ringing timeout: 45 seconds
After 45s → Spring Boot marks: MISSED
          → Juma sees missed call notification
```

Catch-Up on Reconnect

```
User was offline — comes back online:
     |
     ▼ WS connects → Ejabberd → auth success
Spring Boot receives: chat.presence.online (RabbitMQ)
     |
Spring Boot:
  Drain RabbitMQ offline queue for user
  Check MAM (Message Archive) for any gaps
```

```
  Build catch-up summary
     |
App receives catch-up payload:
  Missed messages pushed via WS
  App shows banner:
    "Umekosa ujumbe 12, maagizo 2"
    [Angalia] button

Message Archive (MAM — XEP-0313):
  Ejabberd stores last N days of messages
  Client can query: "give me messages since X"
  Covers edge cases where queue was lost
```

---

17. Multi Device

How Multiple Devices Work

```
Kibuti logged into:
  kibuti@nexgate.com/android  ← phone
  kibuti@nexgate.com/tablet   ← tablet

Message arrives:
  Ejabberd delivers to BOTH devices
  Both show the message
  Both show notification

Kibuti reads on phone:
  Phone sends: <displayed id="msg-001"/>
  Ejabberd: sees kibuti read the message

XEP-0280 Message Carbons:
  Tablet automatically receives the read marker
  Tablet clears notification and marks read
  Without user doing anything on tablet

This is how WhatsApp multi-device works
Ejabberd handles it natively via XEP-0280
```

Device Priority

```
If Kibuti active on phone + tablet:
  Both receive messages (carbons)

If only one device active:
  That device receives normally

Presence priority:
  Each resource has a priority number
  Higher priority = preferred delivery target
  Phone: priority 10 (main device)
  Tablet: priority 5 (secondary)
  When both online: phone gets delivery first
  Tablet gets carbon copy

Set in presence stanza:
<presence>
  <priority>10</priority>
</presence>
```

---

18. Shop Inbox in Phase 2

Shop JID — The Shop as XMPP Entity

```
Each NexGate shop has its own JID:
  techstore@shops.nexgate.com

This is NOT Kibuti's personal JID
This is the SHOP's identity

When customer messages TechStore:
  Customer sends to: techstore@shops.nexgate.com
  Any authorized staff member sees it
  All staff respond AS techstore@shops.nexgate.com
  Customer sees "TechStore" — not individual names
```

```
Staff authentication to shop JID:
  Staff logs in with own account
  Switches to shop context in app
  Spring Boot issues shop XMPP sub-token:
    { jid: "techstore@shops.nexgate.com",
      staffId: "usr-amina",
      role: "SUPPORT_AGENT" }
  Ejabberd allows staff to auth as shop JID
  All messages from staff appear as TechStore
```

Multiple Staff — Shared Inbox

```
TechStore has 3 staff:
  Kibuti  (owner — Manager role)
  Amina   (Support Agent)
  John    (Support Agent)

Customer sends message to TechStore:
  Message arrives at techstore@shops.nexgate.com
  Ejabberd delivers to ALL connected TechStore staff
  (All three see the incoming message simultaneously)

Amina responds:
  Response appears as "TechStore" to customer
  Spring Boot audit log:
    { messageId, respondedBy: "usr-amina",
      shopId: "shop-techstore", timestamp }
  Kibuti and John see Amina's response in their inbox too
  (full shared inbox — everyone sees everything)

Benefits:
  No missed customer messages
  Any staff can pick up any conversation
  Owner can monitor all conversations
  Customer always talks to "TechStore"
```

19. Security

Transport Security

```
WebSocket:
  wss:// (WebSocket Secure)
  TLS 1.3 termination at Traefik
  All chat traffic encrypted in transit

TURN relay:
  SRTP (Secure Real-time Transport Protocol)
  Voice/video encrypted even through Coturn
  Coturn relays encrypted packets
  Coturn cannot decrypt audio/video

XMPP tokens:
  Short-lived (24 hours)
  Signed with RS256 (asymmetric)
  Separate from REST JWT
  Stored in Vault
```

Internal Service Security

```
Ejabberd → Spring Boot:
  X-Internal-Secret header
  Secret stored in Vault
  Only Ejabberd knows this secret
  Spring Boot rejects any request without it

Spring Boot → Ejabberd:
  Admin token (Ejabberd API key)
  Stored in Vault
  Port 5285 bound to 127.0.0.1 only
  Not exposed to public internet

All inter-service secrets:
  Stored in HashiCorp Vault 
```

```
  Rotatable without restart
  Never in environment files
  Never in Docker Compose plain text
```

Message Privacy

```
Server-side:
  Messages stored in PostgreSQL (encrypted at rest)
  Media stored in MinIO (server-side encryption)
  Shop conversations isolated from personal inbox
  Staff cannot access personal DMs of owner

In transit:
  WSS for all WebSocket traffic
  SRTP for all call media

Future (E2E encryption):
  Signal Protocol integration possible
  Would use OMEMO (XEP-0384) on top of XMPP
  Ejabberd supports OMEMO natively
  Messages encrypted on device
  Server stores ciphertext only
  Not in Phase 2 scope — plan for Phase 3
```

---

20. Database Schema

conversations

```
conversations
──────────────────────────────────────────────

id                UUID
type              ENUM    DM / GROUP / COMMERCE
owner_type        ENUM    USER / SHOP
owner_id          UUID    userId or shopId
title             TEXT    groups only
avatar_file_id    UUID
```

```
status            ENUM    ACTIVE / ARCHIVED / BLOCKED
created_by        UUID
created_at        TIMESTAMPTZ
last_message_at   TIMESTAMPTZ
last_message_preview TEXT
```

conversation_members

```
conversation_members
_____________________________________________

conversation_id   UUID
user_id           UUID
role              ENUM    OWNER / ADMIN / MODERATOR / MEMBER
joined_at         TIMESTAMPTZ
last_read_at      TIMESTAMPTZ
last_read_seq     BIGINT
is_muted          BOOLEAN
muted_until       TIMESTAMPTZ
notifications     ENUM    ALL / MENTIONS / NONE
```

messages

```
messages
_____________________________________________

id                UUID
conversation_id   UUID
sender_id         UUID
seq               BIGINT    monotonic per conversation
type              ENUM      TEXT / IMAGE / VIDEO /
                            VOICE_NOTE / FILE /
                            PRODUCT_CARD / CUSTOM_PRICE_OFFER /
                            EVENT_CARD / GROUP_PURCHASE_CARD /
                            POST_CARD / ORDER_CONFIRMATION /
                            ORDER_STATUS_UPDATE /
                            PAYMENT_CONFIRMATION / SYSTEM
body              TEXT
media_ref         UUID      File Thunder fileId
context_type      ENUM      PRODUCT / ORDER / PAYMENT /
```

```
                          EVENT / GROUP_PURCHASE
context_ref_id    UUID
snapshot_json     JSONB     frozen context at send time
reply_to_id       UUID
status            ENUM      SENT / DELIVERED / READ / FAILED
level             ENUM      NORMAL / IMPORTANT / CRITICAL
edited_at         TIMESTAMPTZ
deleted_at        TIMESTAMPTZ
created_at        TIMESTAMPTZ
```

message_receipts

```
message_receipts
_____________________________________________

message_id        UUID
user_id           UUID
status            ENUM      DELIVERED / READ
device_id         TEXT
timestamp         TIMESTAMPTZ
```

calls

```
calls
_____________________________________________

call_id           UUID
caller_id         UUID
receiver_id       UUID
conversation_id   UUID
type              ENUM      VOICE / VIDEO
status            ENUM      RINGING / CONNECTED /
                            COMPLETED / MISSED /
                            DECLINED / FAILED
started_at        TIMESTAMPTZ
answered_at       TIMESTAMPTZ
ended_at          TIMESTAMPTZ
duration_seconds  INT
relay_used        BOOLEAN
end_reason        ENUM      NORMAL / NETWORK /
```

```
                                TIMEOUT / DECLINED
```

call_quality_logs

```
call_quality_logs
──────────────────────────────────────────────
log_id            UUID
call_id           UUID
timestamp         TIMESTAMPTZ
bitrate_kbps      INT
packet_loss_pct   DECIMAL
jitter_ms         INT
rtt_ms            INT
resolution        TEXT      null for voice calls
codec_audio       TEXT      "opus"
codec_video       TEXT      "h264" "vp8" null
```

shop_conversation_access

```
shop_conversation_access
──────────────────────────────────────────────
shop_id           UUID
user_id           UUID
role              ENUM      MANAGER / SUPPORT_AGENT / READ_ONLY
granted_by        UUID
granted_at        TIMESTAMPTZ
revoked_at        TIMESTAMPTZ
```

notification_log

```
notification_log
──────────────────────────────────────────────
id                UUID
user_id           UUID
message_id        UUID
level             ENUM      NORMAL / IMPORTANT / CRITICAL
fcm_status        ENUM      SENT / DELIVERED / FAILED
sms_status        ENUM      SENT / DELIVERED / FAILED / SKIPPED
```

```
sms_provider      TEXT
sent_at           TIMESTAMPTZ
delivered_at      TIMESTAMPTZ
opened_at         TIMESTAMPTZ
```

Summary

Private chat and calls in NexGate Phase 2 are built on four pillars:

Ejabberd Cluster runs as two Docker containers on the same Hetzner VPS at launch. Erlang Distribution connects them directly — messages between nodes route in microseconds without Redis pub/sub. Traefik sticky sessions keep each user's WebSocket on one node. If one node crashes the other keeps serving. At growth stage two separate Hetzner VPS give true hardware redundancy.

Ejabberd handles everything real-time — WebSocket connections, XMPP stanza routing, presence, chat states, message receipts, MUC group chats, and Jingle call signaling. All message interactions (edit XEP-0308, delete XEP-0424, reactions XEP-0444, forwarding XEP-0297, replies XEP-0461) are routed automatically — Spring Boot only handles persistence and rule validation.

Group chats use a consent-based join model. Nobody enters a group without actively choosing. Two mechanisms: consent DM invitation (admin handpicks from their network, each person accepts or declines) and invite link (private groups require admin approval, public groups allow instant join). Both private and public group types supported. No forced adding — better than WhatsApp.

WebRTC handles all calls. 1:1 calls use P2P or Coturn relay via Jingle signaling. Group calls use LiveKit SFU (already deployed for Audio Spaces) — zero new infrastructure. Audio↔video switching uses Jingle content-add/remove without ending the session. Screen sharing uses MediaProjection (Android) and ReplayKit (iOS) as a special video track. Opus adapts from 64kbps to 6kbps. H.264 hardware acceleration keeps battery impact low on EA phones.

Spring Boot Chat Service handles all business logic — message persistence, commerce context, offer sessions, shop inbox access control, notification routing, and call records. Auth with Ejabberd is synchronous HTTP (needs immediate allow/deny). Everything else is async via RabbitMQ.

The shop inbox is isolated from personal DMs at the JID level — the shop has its own Ejabberd identity, multiple staff share it, and customers always see the shop brand, never individual staff names.

NexGate Private Chat & Calls — Phase 2 Deep Dive v1.0 QBIT SPARK | XMPP · Ejabberd · WebRTC · Jingle · Coturn · Opus · H.264 · Group Calls · Screen Share

---

Revision #1
Created 14 July 2026 07:09:28 by Admin Qbit

Updated 14 July 2026 07:19:32 by Admin Qbit