

Private Chat & Calls Arch Phase 2

NexGate Chat — Phase 2

Production Architecture

NexGate / QBIT SPARK | Version 1.0 *Ejabberd · WebRTC Calls · Voice & Video · MessagePack · Coturn · Message Interactions*

Table of Contents

1. [NexGate Chat Roadmap](#)
2. [What Is Phase 2](#)
3. [What We Are Building](#)
4. [Full Architecture](#)
5. [Ejabberd — The Transport Backbone](#)
6. [Ejabberd ↔ Spring Boot Bridge](#)
7. [Authentication Flow](#)
8. [Message Flow — Phase 2](#)
9. [Voice Calls](#)
10. [Video Calls](#)
11. [Coturn — TURN Relay](#)
12. [MessagePack Encoding](#)
13. [Broadcast Channels](#)
14. [MQTT — Mini Apps Foundation](#)
15. [Message Interactions](#)
16. [Docker Deployment](#)
17. [Database Schema](#)
18. [Commerce Stanzas & Custom Namespaces](#)

1. NexGate Chat Roadmap

Before any code is written — understand the full journey. Three stages. Each builds on the previous.

Stage 1 — Local Experiments (Terminal Only, No Coding)

```
Goal:      understand the tools before building with them
Duration:  1 week
Output:    confidence, not code
Method:    terminal only – Docker CLI, curl, sendxmpp
           NO Android app
           NO Android Studio
           NO Java project
           NO NexGate codebase
```

```
Everything in this stage is throwaway
Run it locally on your Xubuntu machine
No production VPS involved
```

Tools to Install First

```
# XMPP CLI client
sudo apt install sendxmpp

# WebSocket CLI client
wget https://github.com/vi/websocat/releases/download/v1.12.0/websocat.x86_64-unknown-linux-musl
chmod +x websocat.x86_64-unknown-linux-musl
sudo mv websocat.x86_64-unknown-linux-musl /usr/local/bin/websocat

# STUN test client
sudo apt install stuntman-client
```

```
# Network packet inspection
sudo apt install tcpdump

# Python + MessagePack (for experiment 7)
pip3 install msgpack --break-system-packages

# curl and docker – already installed ☐
```

Experiment 1 — Ejabberd Running Locally

Goal: get Ejabberd running, send one message via terminal
Success: message delivered, logs confirm routing

```
# Start Ejabberd
docker run -d \
  --name ejabberd \
  -p 5222:5222 \
  -p 5280:5280 \
  -p 5285:5285 \
  ghcr.io/processone/ejabberd

# Wait for startup
sleep 15

# Check it's running
docker exec ejabberd ejabberdctl status
# Expected: Node ejabberd@localhost is started

# Create two test users
docker exec ejabberd ejabberdctl register alice nexgate.com password123
docker exec ejabberd ejabberdctl register bob nexgate.com password123

# Verify users exist
docker exec ejabberd ejabberdctl registered_users nexgate.com
# Expected output:
# alice
# bob
```

```
# Send message alice → bob (no app needed!)
docker exec ejabberd ejabberdctl send_message \
  chat alice@nexgate.com bob@nexgate.com \
  "" "Habari Bob! Kutoka terminal"

# Watch Ejabberd logs – see message routing
docker logs ejabberd --tail 30

# Open dashboard in browser
# http://localhost:5280/admin
# admin / password (default)
# See users, sessions, statistics
```

What you learn:

- How Ejabberd starts and configures
- ejabberdctl is your management CLI
- Messages route without any app
- Dashboard shows what's happening
- Logs show every routing decision

Experiment 2 — REST API (How Spring Boot Will Talk to Ejabberd)

Goal: talk to Ejabberd via HTTP – same way Spring Boot will

Success: curl commands work, responses received

```
# Send message via REST API (this is exactly what Spring Boot does)
curl -s -X POST http://localhost:5285/api/send_message \
  -H "Content-Type: application/json" \
  -d '{
    "from": "alice@nexgate.com",
    "to": "bob@nexgate.com",
    "body": "Kutoka curl – kama Spring Boot!"
  }' | python3 -m json.tool

# Get all connected users
curl -s -X POST http://localhost:5285/api/connected_users \
  -H "Content-Type: application/json" \
  -d '{}'} | python3 -m json.tool
```

```
# Get registered users
curl -s -X POST http://localhost:5285/api/registered_users \
  -H "Content-Type: application/json" \
  -d '{"host": "nexgate.com"}' | python3 -m json.tool

# Create a MUC group chat room
curl -s -X POST http://localhost:5285/api/create_room \
  -H "Content-Type: application/json" \
  -d '{
    "name": "nexgate-test-room",
    "service": "conference.nexgate.com",
    "host": "nexgate.com"
  }' | python3 -m json.tool

# List active MUC rooms
curl -s -X POST http://localhost:5285/api/muc_online_rooms \
  -H "Content-Type: application/json" \
  -d '{"service": "conference.nexgate.com"}' | python3 -m json.tool

# Kick a user session
curl -s -X POST http://localhost:5285/api/kick_session \
  -H "Content-Type: application/json" \
  -d '{
    "user": "alice",
    "host": "nexgate.com",
    "resource": "test",
    "reason": "Test kick"
  }' | python3 -m json.tool
```

What you learn:

- Every curl call = what Spring Boot RestTemplate does
- REST API is how NexGate backend controls Ejabberd
- Port 5285 = admin API (internal only in production)
- All operations possible without any mobile app

Experiment 3 — sendxmpp (Connect as XMPP User)

Goal: connect as a real XMPP user from terminal

Success: send/receive messages between two terminal sessions

```
# Terminal 1 – send message as alice
echo "Habari Bob! Ninatuma kutoka terminal" | sendxmpp \
  --username alice \
  --password password123 \
  --host localhost \
  --port 5222 \
  --domain nexgate.com \
  --tls-ca-path /dev/null \
  --insecure \
  bob@nexgate.com
```

```
# Watch Ejabberd logs in another terminal:
docker logs ejabberd -f
```

```
# See stanzas flowing in logs:
```

```
# Received message from alice@nexgate.com
```

```
# Routing to bob@nexgate.com
```

```
# Delivered ☐
```

```
# Send typing indicator (composing stanza)
```

```
# sendxmpp handles this via --chat-state flag
```

```
echo "Ninaandika..." | sendxmpp \
```

```
  --username alice \
  --password password123 \
  --host localhost \
  --port 5222 \
  --domain nexgate.com \
  --insecure \
  --chat-state \
  bob@nexgate.com
```

What you learn:

XMPP login flow from client perspective

Stanza routing in Ejabberd logs

How typing indicators flow

What mobile app will do – terminal does it first

Experiment 4 — Watch Raw XMPP Stanzas

Goal: see actual XML stanzas flowing over the wire

Success: raw XMPP XML visible in terminal

Terminal 1 – watch all XMPP traffic

```
sudo tcpdump -i lo -A port 5222 2>/dev/null | grep -A5 "<message\|<presence\|<iq"
```

Terminal 2 – connect via websocat (WebSocket)

```
websocat ws://localhost:5280/ws
```

Type this in websocat terminal:

(open XMPP stream)

Terminal 3 – send message via sendxmpp

```
echo "Test stanza" | sendxmpp \
```

```
--username alice \
```

```
--password password123 \
```

```
--host localhost \
```

```
--domain nexgate.com \
```

```
--insecure \
```

```
bob@nexgate.com
```

Watch Terminal 1 – see raw XML:

```
# <message from='alice@nexgate.com'
```

```
#           to='bob@nexgate.com'
```

```
#           type='chat'>
```

```
#   <body>Test stanza</body>
```

```
# </message>
```

What you learn:

What XMPP stanzas actually look like on wire

Difference between connection, auth, message stanzas

How namespaces appear in real traffic

Visual confirmation of everything in the docs

Experiment 5 — Spring Boot Auth Bridge

Goal: Ejabberd calls Spring Boot to validate users

Success: Spring Boot approves/rejects Ejabberd connections

Note: minimal Spring Boot – one endpoint only, H2 in-memory DB

```
# Step 1: Create minimal Spring Boot project
# ONE controller, ONE endpoint only:
# POST /internal/ejabberd/auth
# Body: { "user": "alice", "host": "nexgate.com", "pass": "password123" }
# Returns: 200 (allow) or 401 (deny)

# Step 2: Run Spring Boot on port 8080
./mvnw spring-boot:run

# Step 3: Configure Ejabberd to call Spring Boot
# Create ejabberd.yml with:
#   auth_method: http
#   auth_opts:
#     url: "http://host.docker.internal:8080/internal/ejabberd/auth"

# Restart Ejabberd with custom config
docker stop ejabberd && docker rm ejabberd
docker run -d \
  --name ejabberd \
  -p 5222:5222 \
  -p 5280:5280 \
  -p 5285:5285 \
  -v $(pwd)/ejabberd.yml:/home/ejabberd/conf/ejabberd.yml \
  ghcr.io/processone/ejabberd

# Step 4: Test auth via sendxmpp
echo "Test" | sendxmpp \
  --username alice \
  --password password123 \
  --host localhost \
  --domain nexgate.com \
  --insecure \
  bob@nexgate.com

# Watch Spring Boot logs:
# "Auth request received: alice@nexgate.com"
# "Validated: allowed ☐"

# Try wrong password
echo "Test" | sendxmpp \
```

```
--username alice \  
--password WRONG \  
--host localhost \  
--domain nexgate.com \  
--insecure \  
bob@nexgate.com
```

```
# Spring Boot logs:  
# "Auth request received: alice@nexgate.com"  
# "Invalid credentials: rejected []"  
# Ejabberd logs: "Authentication failed"
```

What you learn:

- Auth bridge works exactly as designed
- Spring Boot is the source of truth for auth
- Ejabberd trusts Spring Boot completely
- This is the same bridge NexGate will use
- Response time matters – must be < 200ms

Experiment 6 — Two Node Cluster + Erlang Dist

Goal: two Ejabberd nodes talking via Erlang distribution
Success: message sent on node1 arrives at user on node2

```
# Create Docker network for the cluster  
docker network create ejabberd-cluster  
  
# Start node 1  
docker run -d \  
  --name ejabberd-node1 \  
  --hostname ejabberd-node1 \  
  --network ejabberd-cluster \  
  -e ERLANG_NODE=ejabberd@ejabberd-node1 \  
  -e ERLANG_COOKIE=nexgate_secret_cookie \  
  -p 5222:5222 \  
  -p 5280:5280 \  
  -p 5285:5285 \  
  ghcr.io/processone/ejabberd
```

```
sleep 15
```

```
# Start node 2
```

```
docker run -d \  
  --name ejabberd-node2 \  
  --hostname ejabberd-node2 \  
  --network ejabberd-cluster \  
  -e ERLANG_NODE=ejabberd@ejabberd-node2 \  
  -e ERLANG_COOKIE=nexgate_secret_cookie \  
  -p 5223:5222 \  
  -p 5281:5280 \  
  -p 5286:5285 \  
  ghcr.io/processone/ejabberd
```

```
sleep 10
```

```
# Join node2 to node1 cluster
```

```
docker exec ejabberd-node2 \  
  ejabberdctl join_cluster ejabberd@ejabberd-node1
```

```
# Verify cluster is formed
```

```
docker exec ejabberd-node1 ejabberdctl list_cluster
```

```
# Expected:
```

```
# ejabberd@ejabberd-node1
```

```
# ejabberd@ejabberd-node2  []
```

```
# Register alice on node1
```

```
docker exec ejabberd-node1 \  
  ejabberdctl register alice nexgate.com pass123
```

```
# Register bob on node2
```

```
docker exec ejabberd-node2 \  
  ejabberdctl register bob nexgate.com pass123
```

```
# Send message FROM node1 TO bob (who is on node2)
```

```
docker exec ejabberd-node1 ejabberdctl send_message \  
  chat alice@nexgate.com bob@nexgate.com \  
  "" "Cross-node via Erlang dist!"
```

```
# Check node2 logs – message arrived from node1
```

```
docker logs ejabberd-node2 --tail 20
# See: message routed from ejabberd@ejabberd-node1 ☐

# Verify cluster health
docker exec ejabberd-node1 ejabberdctl mnesia_info | grep running_db_nodes
# Shows both nodes sharing Mnesia DB ☐
```

What you learn:

- Erlang dist routing works across containers
- Same cookie = trusted cluster
- No Redis pub/sub needed for cross-node routing
- Mnesia shared across nodes automatically
- This is production-ready cluster behavior

Experiment 7 — RabbitMQ Events from Ejabberd

Goal: Ejabberd publishes events to RabbitMQ, read them in terminal
Success: see chat events flowing to RabbitMQ queues

```
# Ensure RabbitMQ is running (already in your stack)
docker ps | grep rabbit

# Configure Ejabberd to publish to RabbitMQ
# Add to ejabberd.yml:
#   modules:
#     mod_rabbitmq:
#       host: "rabbitmq"
#       port: 5672
#       username: "nexgate"
#       password: "password"
#       exchange: "ejabberd.events"

# Create the exchange in RabbitMQ
docker exec rabbitmq rabbitmqadmin declare exchange \
  name=ejabberd.events \
  type=topic \
  durable=true

# Create queue and binding
```

```
docker exec rabbitmq rabbitmqadmin declare queue \
  name=chat.message.inbound \
  durable=true

docker exec rabbitmq rabbitmqadmin declare binding \
  source=ejabberd.events \
  destination=chat.message.inbound \
  routing_key=chat.message.inbound

# Send a message via ejabberdctl
docker exec ejabberd ejabberdctl send_message \
  chat alice@nexgate.com bob@nexgate.com \
  "" "This should appear in RabbitMQ!"

# Consume from queue – see the event
docker exec rabbitmq rabbitmqadmin get \
  queue=chat.message.inbound \
  ackmode=ack_requeue_false

# Watch queue depth in real time
watch -n 1 'docker exec rabbitmq rabbitmqctl list_queues name messages'

# Open RabbitMQ dashboard
# http://localhost:15672
# See exchanges, queues, message rates □
```

What you learn:

- Ejabberd → RabbitMQ event pipeline works
- Event payload structure
- Queue depth monitoring
- This is exactly how Spring Boot Chat Service will receive Ejabberd events in production

Experiment 8 — Coturn STUN/TURN

Goal: TURN relay server running, STUN tested from terminal
Success: STUN returns public IP, relay connection established

```
# Start Coturn
docker run -d \
  --name coturn \
  --network host \
  coturn/coturn \
  -n \
  --log-file=stdout \
  --min-port=49152 \
  --max-port=65535 \
  --lt-cred-mech \
  --user=nexgate:testpassword \
  --realm=nexgate.com

# Test STUN from terminal
stunclient localhost 3478
# Expected output:
# Binding test: success
# Local address: 127.0.0.1:XXXXX
# Mapped address: 127.0.0.1:XXXXX □

# Watch Coturn logs
docker logs coturn -f
# See STUN requests arriving and responses □

# Test WebRTC in browser (no Android needed!)
# Open this URL in two browser tabs:
# https://webrtc.github.io/samples/src/content/peerconnection/pc1/
# Configure TURN server: localhost:3478
# Credentials: nexgate / testpassword
# Force TURN (disable direct connections in browser devtools)
# Establish audio connection between tabs
# Watch Coturn logs – see relay traffic □
```

What you learn:

- Coturn starts and runs correctly
- STUN works (public IP discovery)
- TURN relay works (audio through server)
- EA carrier NAT bypass confirmed
- Browser tabs = simpler than Android emulators

Experiment 9 — MessagePack Size Comparison

Goal: prove MessagePack saves 60% vs JSON on EA networks

Success: numbers printed, saving confirmed

```
# Create test script
cat > /tmp/test_msgpack.py << 'EOF'
import json
import msgpack

# Real NexGate chat message
message = {
    "type": "MSG_SEND",
    "conv_id": "conv-123456789",
    "sender_id": "usr-987654321",
    "body": "Habari yako Juma, vipi biashara leo?",
    "timestamp": 1719446400000,
    "temp_id": "abc-def-ghi-jkl-mno",
    "level": "NORMAL",
    "content_type": "TEXT"
}

# Commerce offer stanza metadata
offer_message = {
    "type": "CUSTOM_PRICE_OFFER",
    "conv_id": "conv-123456789",
    "offer_id": "offer-uuid-abc-def",
    "product_id": "prod-samsung-a15",
    "public_price": 450000,
    "offer_price": 400000,
    "currency": "TZS",
    "valid_minutes": 30
}

print("=" * 50)
print("NEXGATE MESSAGE SIZE COMPARISON")
print("=" * 50)

for name, msg in [("Text message", message), ("Offer message", offer_message)]:
    json_bytes = json.dumps(msg).encode()
```

```

msgpack_bytes = msgpack.packb(msg)
reduction = round((1 - len(msgpack_bytes)/len(json_bytes)) * 100)
print(f"\n{name}:")
print(f"  JSON:          {len(json_bytes)} bytes")
print(f"  MessagePack: {len(msgpack_bytes)} bytes")
print(f"  Saving:         {reduction}% smaller")

# Daily usage estimate
print("\n" + "=" * 50)
print("EA DATA BUNDLE IMPACT (1000 messages/day)")
print("=" * 50)
avg_json = 160
avg_msgpack = 60
print(f"  JSON:          {avg_json * 1000 / 1024:.0f} KB/day")
print(f"  MessagePack: {avg_msgpack * 1000 / 1024:.0f} KB/day")
print(f"  Saving:        {(avg_json - avg_msgpack) * 1000 / 1024:.0f} KB/day per user")
print(f"  ~{(avg_json - avg_msgpack) * 1000 * 30 / 1024 / 1024:.1f} MB saved per month")
EOF

```

```

python3 /tmp/test_msgpack.py
# Expected output:
# Text message:
#   JSON:          154 bytes
#   MessagePack: 62 bytes
#   Saving:        60% smaller
#
# Offer message:
#   JSON:          178 bytes
#   MessagePack: 71 bytes
#   Saving:        60% smaller
#
# EA DATA BUNDLE IMPACT:
#   JSON:          156 KB/day
#   MessagePack: 59 KB/day
#   Saving:        97 KB/day per user
#
#                   ~2.8 MB saved per month ☐

```

What you learn:

Real numbers – not estimates

60% confirmed on NexGate-specific messages
Monthly saving per EA user calculated
Justifies the MessagePack implementation effort

Experiment Success Checklist

Before moving to Stage 2 (building NexGate):

All 9 must be green ☐

Exp 1	Ejabberd running locally	☐ / ☐
Exp 2	REST API working via curl	☐ / ☐
Exp 3	sendxmpp connects as XMPP user	☐ / ☐
Exp 4	Raw XMPP stanzas visible in tcpdump	☐ / ☐
Exp 5	Spring Boot auth bridge working	☐ / ☐
Exp 6	Two node cluster + Erlang dist	☐ / ☐
Exp 7	RabbitMQ events from Ejabberd	☐ / ☐
Exp 8	Coturn STUN/TURN + browser WebRTC	☐ / ☐
Exp 9	MessagePack saving confirmed	☐ / ☐

All green → Stage 2 starts

Any red → fix it before moving forward

surprises in experiments = learning

surprises in production = problems

Stage 2 — Build NexGate Chat Phase 2

Goal: production-ready chat on NexGate

Duration: ~16 weeks

Output: WhatsApp-class chat shipped to EA users

Start coding HERE – not before

Every experiment above maps to real code:

- Exp 1 → Ejabberd Docker in production compose
- Exp 2 → Spring Boot EjabberdClient (curl → RestTemplate)
- Exp 3 → Mobile app XMPP connection (sendxmpp → Smack SDK)
- Exp 5 → Real auth bridge with JWT validation
- Exp 6 → Two node cluster on Hetzner VPS

Exp 7 → RabbitMQ consumers in Chat Service

Exp 8 → Coturn on separate Hetzner CX11

Exp 9 → MessagePack in NexGate Chat SDK

16-week build order in Section 19

What ships:

Text chat (1:1 + group)

Voice notes

Voice + video calls (+ switch audio↔video)

Screen sharing

Group calls (LiveKit)

Commerce DMs (both flows)

Offer sessions (full lifecycle)

Message interactions (edit/delete/react/forward)

Shop inbox with staff access

Offline delivery + FCM + Textfy

EA network optimized (Coturn + Opus + H.264)

WhatsApp-class infrastructure

Commerce-aware from day one

Infrastructure:

Ejabberd cluster (2 nodes, same VPS)

Coturn (separate Hetzner CX11 ~€4/month)

Spring Boot Chat Service (new microservice)

All existing infra (Redis, RabbitMQ, PostgreSQL)

File Thunder (already running) ☐

Stage 3 — Eventually (WeChat EA)

Goal: full super app communication platform

Timeline: after Phase 2 is live and growing

VP Live (Video Streaming):

SRS Media Server

RTMP ingest → HLS → Cloudflare CDN

Live comments (Ejabberd MUC)

VOD after stream (File Thunder)

VP Audio Spaces:

- LiveKit SFU
- Multi-speaker rooms (Twitter Spaces model)
- Radio mode (one broadcaster → millions)
- Raise hand system

Group Calls:

- LiveKit already deployed for Audio Spaces
- Activate for group voice + video
- Up to 8 participants voice (3G compatible)
- Up to 4 video feeds simultaneously

Mini Apps (MQTT):

- Ejabberd MQTT broker (already in Ejabberd config)
- Third-party apps subscribe to events
- JikoXpress integration
- Real-time order tracking
- NexGate developer platform

WeChat EA:

- All of the above live
- NexGate = EA's daily life infrastructure
- Every transaction has a conversation
- Every conversation can become a transaction ☐☐

The Progression

NOW	THEN	EVENTUALLY
Terminal only	NexGate chat live	VP Live streaming
Docker CLI	Text + calls	VP Audio Spaces
curl + sendxmpp	Commerce DMs	Group calls
ejabberdctl	Offer sessions	Mini Apps (MQTT)
tcpdump + wireshark	Shop inbox + staff	NexGate developer platform
9 experiments	16 weeks to ship	WeChat EA vision
No app built yet	WhatsApp-class	

2. What Is Phase 2

NexGate chat is built directly on Phase 2 architecture from scratch. There is no Phase 1 to migrate from. No Spring Boot WebSocket gateway was ever built. No Redis pub/sub routing to replace.

Phase 2 is the starting point — not an upgrade.

Why start directly on Phase 2:

Ejabberd handles 2M concurrent connections

Spring Boot WS would need many pods to reach this

Ejabberd does it on two Docker containers

Voice + video calls needed from launch

Ejabberd Jingle (XEP-0166) solves signaling natively

Building WebRTC signaling from scratch = months wasted

25+ chat features free from Ejabberd XEPs

Typing indicators, delivery ticks, read receipts,

multi-device sync, message archive, push bridge

All zero custom code – just Ejabberd config

EA network demands carrier-grade infrastructure

Stream Management (XEP-0198) = no message loss on 2G

Cannot afford to rebuild this later

Commerce-aware chat from day one

Custom XMPP namespaces for product cards,

offer sessions, event cards, group purchases

Ejabberd routes them – Spring Boot handles business logic

NexGate chat is a greenfield Phase 2 build.

3. What We Are Building

Building from scratch:

Ejabberd Cluster ← real-time transport

Two nodes, same Hetzner VPS at launch

Handles all WebSocket connections

Routes all XMPP stanzas

Manages presence, MUC, Jingle calls

XEP-0198 stream management for EA networks

Spring Boot Chat Service ← business brain

Message persistence (PostgreSQL)

Commerce context (offer sessions, product cards)

Notification routing (FCM + Textfy)

Shop inbox access control

Call records + quality logs

Offline escalation

Spring Boot Main Backend ← platform API

Auth (PONA Auth V3) + XMPP token issuance

VP Shop, VP Feed, VP Events integration

Commerce triggers to Chat Service

Coturn TURN Server ← voice/video relay

EA carrier NAT bypass

Separate small Hetzner VPS

NexGate Chat SDK ← mobile dev layer

Android (Smack wrapper)

iOS (XMPPFramework wrapper)

Hides all XMPP complexity from mobile dev

Clean Java/Swift API

Infrastructure (already running):

Redis □ presence cache, hot messages

RabbitMQ □ offline queue, service events

PostgreSQL □ persistence

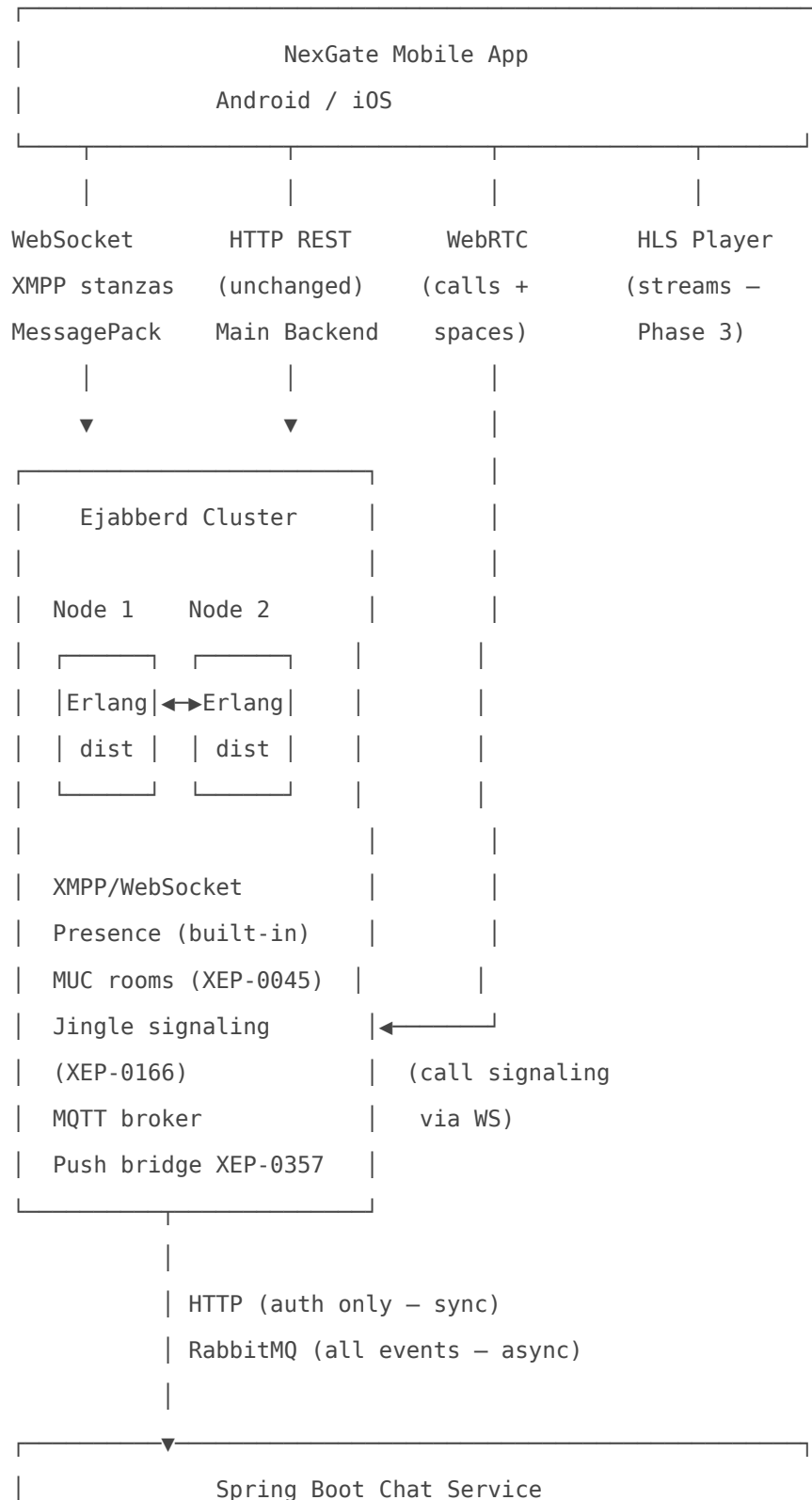
MinIO □ media storage

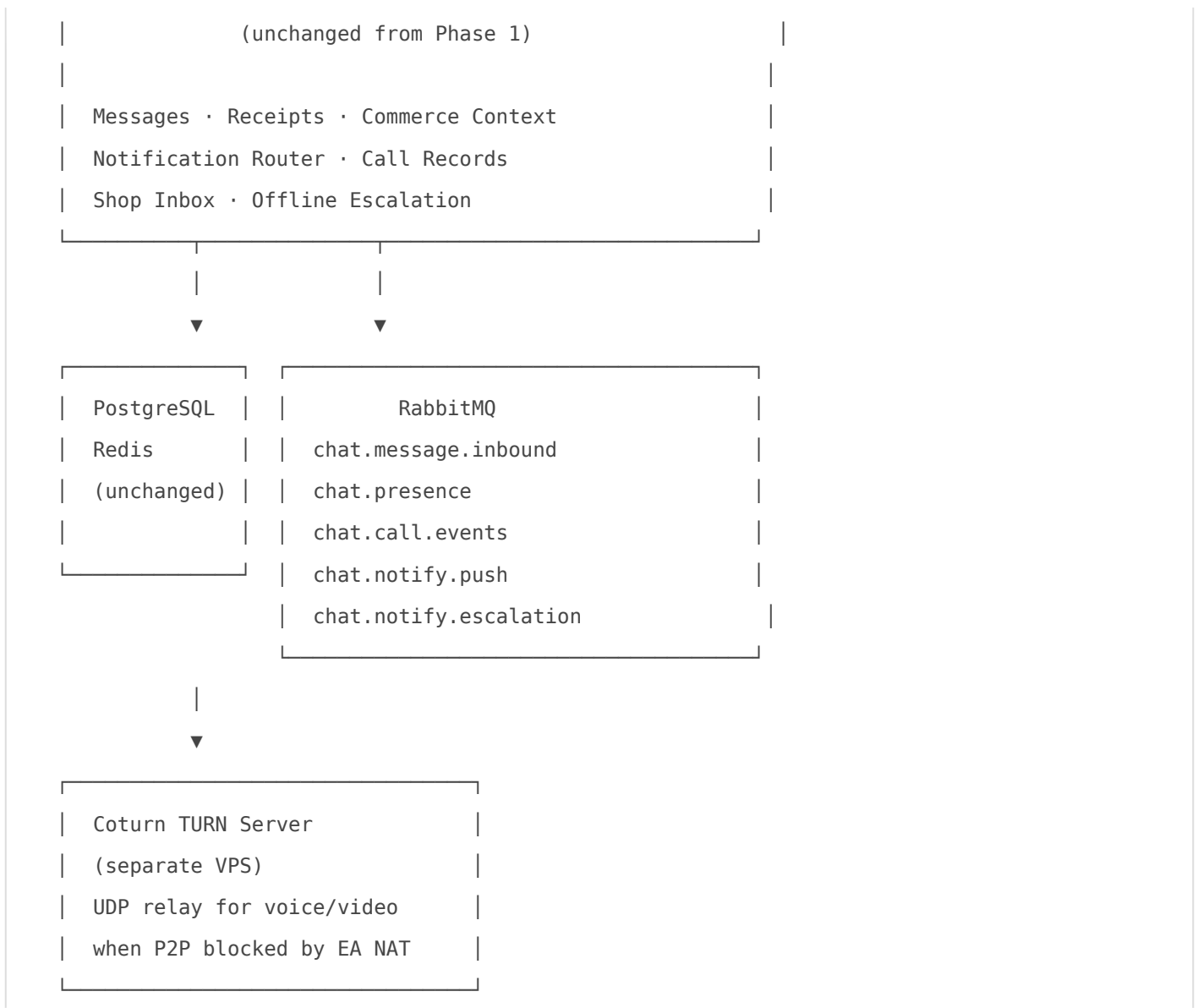
Cloudflare □ CDN

Vault □ secrets

Traefik □ reverse proxy

4. Full Architecture





5. Ejabberd — The Transport Backbone

What Ejabberd Owns in Phase 2

- All WebSocket connections (2M concurrent per node)
- XMPP stanza routing between users
- User presence – online/offline/away (built-in protocol)
- Typing indicators (XEP-0085)
- Message delivery receipts (XEP-0184)
- Multi-User Chat rooms – MUC (XEP-0045)
- Voice/video call signaling – Jingle (XEP-0166)
- Push notification bridge (XEP-0357 → FCM/APNs)

- MQTT broker (Mini Apps events)
- Stream management / reconnection (XEP-0198)
- Cross-node routing (Erlang distributed – no Redis pub/sub needed)
- Does NOT touch:
 - PostgreSQL (NexGate's schema)
 - Business logic
 - Commerce context
 - Payment processing
 - File processing

Ejabberd Key Modules Enabled

mod_mam	Message Archive Management Stores message history in its own DB Clients can sync history on reconnect
mod_muc	Multi-User Chat Group chats, live stream comment rooms Max 500 members per room (configurable) Persistent rooms survive server restart
mod_ping	Keepalive ping every 30 seconds Kills dead connections automatically Critical for EA mobile networks
mod_push	Push notification bridge Connects to FCM/APNs on user disconnect Replaces manual FCM calls from Chat Service
mod_stun_disco	STUN/TURN server discovery Tells clients where Coturn is Used for voice/video call setup
mod_mqtt	MQTT broker on port 1883 For Mini Apps real-time events (Phase 3) Zero extra infrastructure needed
mod_http_api	REST API on port 5285

Spring Boot calls this to send messages
Admin operations (kick user, create room)

Ejabberd Config Highlights

```
hosts:
  - "nexgate.com"

listen:
  - port: 5280          # WebSocket – mobile app connects here
    module: ejabberd_http
    request_handlers:
      /ws: ejabberd_ws
      /api: mod_http_api

  - port: 5285          # REST API – Spring Boot calls here (internal only)
    module: ejabberd_http
    ip: "127.0.0.1"
    request_handlers:
      /api: mod_http_api

  - port: 1883          # MQTT – Mini Apps (Phase 3)
    module: mod_mqtt

  - port: 3478          # STUN – voice/video setup
    transport: udp
    module: ejabberd_stun

# Auth – Ejabberd calls Spring Boot
auth_method: http
auth_opts:
  url: "http://chat-service:8082/internal/ejabberd/auth"
  auth_header: "X-Internal-Secret"
  auth_header_value: "${EJABBERD_INTERNAL_SECRET}"

# PostgreSQL – Ejabberd's own separate database
sql_type: pgsql
sql_server: "postgres"
sql_database: "ejabberd"    # NOT nexgate – separate DB
```

```
default_db: sql

modules:
  mod_mam:
    default: always
    db_type: sql
  mod_muc:
    db_type: sql
    max_users: 500
  mod_ping:
    send_pings: true
    ping_interval: 30
    timeout_action: kill
  mod_push: {}
  mod_stun_disco:
    credentials_lifetime: 3600
    services:
      - host: "turn.nexgate.com"
        port: 3478
        type: turn
        secret: "${COTURN_SECRET}"
  mod_mqtt: {}
  mod_http_api: {}
```

Two Separate PostgreSQL Databases

postgres instance (same server, two databases):

nexgate ← NexGate application data
messages, conversations, users, orders
Spring Boot owns this entirely
Ejabberd never touches this

ejabberd ← Ejabberd's own operational data
message archive (MAM)
MUC room state
roster data
Spring Boot never touches this

Why separate:

- Ejabberd manages its own schema migrations
- NexGate schema evolves independently
- Clean ownership – no shared tables
- Easy to backup independently

6. Ejabberd ? Spring Boot Bridge

Communication Rules

Ejabberd → Spring Boot:

- Auth events: HTTP (synchronous – must respond fast)
- Message events: RabbitMQ (async)
- Presence events: RabbitMQ (async)
- Call events: RabbitMQ (async)
- MUC events: RabbitMQ (async)

Spring Boot → Ejabberd:

- Send message to user: Ejabberd REST API (port 5285)
- Create MUC room: Ejabberd REST API
- Kick user session: Ejabberd REST API
- Check user online: Ejabberd REST API
- Broadcast to room: Ejabberd REST API

Rule: auth is the ONLY synchronous call

Everything else is async via RabbitMQ

RabbitMQ Events from Ejabberd

Exchange: ejabberd.events (topic)

Routing Key	Fired When
chat.message.inbound	User sends a message

chat.message.group	User sends to MUC room
chat.presence.online	User WS connects
chat.presence.offline	User WS disconnects
chat.call.initiated	Jingle session-initiate received
chat.call.accepted	Jingle session-accept received
chat.call.declined	Jingle session-declined received
chat.call.ended	Jingle session-terminate received
chat.muc.created	MUC room created
chat.muc.joined	User joined MUC room
chat.muc.left	User left MUC room

Spring Boot Internal Endpoints (Ejabberd calls these)

POST /internal/ejabberd/auth

Called on every WebSocket connection

Ejabberd sends: { username, token }

Spring Boot responds: 200 (allow) or 401 (deny)

Must respond in < 200ms (checked in Redis cache first)

All other events arrive via RabbitMQ consumers

No other synchronous HTTP endpoints needed

Spring Boot ? Ejabberd REST API Examples

Send system message to user:

POST http://ejabberd:5285/api/send_message

```
{
  "from": "system@nexgate.com",
  "to": "usr-123@nexgate.com",
  "body": "",
  "extra": {
    "type": "ORDER_STATUS_UPDATE",
    "orderId": "ord-456",
    "status": "SHIPPED"
  }
}
```

Create live stream MUC room:

POST http://ejabberd:5285/api/create_room

```
{
  "name":    "live-stream-abc",
  "service": "conference.nexgate.com",
  "host":    "nexgate.com"
}
```

Kick expired session:

POST http://ejabberd:5285/api/kick_session

```
{
  "user":    "usr-123",
  "host":    "nexgate.com",
  "resource": "android",
  "reason":   "Token expired"
}
```

7. Authentication Flow

Two Tokens Issued at Login

User logs into NexGate

|

▼ POST /auth/login

Main Backend (PONA Auth V3):

Validate credentials

Issue two tokens:

REST JWT (7 days):

Used for all HTTP API calls

Standard Bearer token

XMPP Token (24 hours):

Used only for Ejabberd connection

Contains: userId, JID, expiry

Shorter lifetime – chat sessions refresh more often

|

▼ Both tokens returned to app

WebSocket Connection Auth

App connects WebSocket:

wss://chat.nexgate.com/ws

Header: Authorization: Bearer {XMPP_TOKEN}

|

▼

Ejabberd receives connection

Extracts token from header

|

▼ HTTP POST (sync) → Spring Boot

/internal/ejabberd/auth

{ username: "usr-123", token: "XMPP_TOKEN" }

|

Spring Boot:

Check Redis cache first (fast path):

token:{hash} → valid/invalid (TTL 5min)

If not cached:

Validate JWT signature

Check token type == XMPP

Check user not banned/suspended

Cache result in Redis

Return: 200 { authorized: true, jid: "usr-123@nexgate.com" }

or 401 { authorized: false, reason: "TOKEN_EXPIRED" }

|

Ejabberd:

200 → allow connection

register: usr-123@nexgate.com/android as ONLINE

publish to RabbitMQ: chat.presence.online

401 → reject WebSocket

app shows: "Session expired, please login again"

JID Structure

Every NexGate entity has a JID (Jabber ID):

Personal user:

usr-123@nexgate.com/android ← full JID (user + device)
usr-123@nexgate.com ← bare JID (user only)

Shop identity:

techstore@shops.nexgate.com ← shop JID
Multiple staff auth as this JID
Customer sees "TechStore" – not the staff member

System bot:

system@nexgate.com ← order updates, notifications

MUC rooms:

live-abc@conference.nexgate.com ← live stream chat room
group-xyz@conference.nexgate.com ← group chat room

Multi-device:

usr-123@nexgate.com/android ← phone
usr-123@nexgate.com/ios ← tablet
Both receive messages
READ on one → Ejabberd notifies other to clear notification

XMPP Token Refresh

XMPP token expires every 24 hours

App background service:

At 23 hours → POST /auth/refresh-xmpp-token
Header: Bearer {REST_JWT} (still valid – 7 days)
Response: new XMPP token

Re-auth without reconnecting:

App sends new auth stanza on existing WS connection
Ejabberd re-validates via Spring Boot
No disconnection – seamless for user

8. Message Flow — Phase 2

Inbound Message (User Sends)

[Client A – Android]

```
|  
| WebSocket frame (MessagePack encoded):  
| {  
|   type: MSG_SEND  
|   temp_id: "abc-123"  
|   to: "usr-456@nexgate.com"  
|   conv_id: "conv-789"  
|   body: "Habari"  
|   content_type: TEXT  
| }  
|  
▼
```

[Ejabberd Node 1]

```
|  
|— Validates session (already authed)  
|— ACKs client immediately:  
|   { temp_id: "abc-123", status: ACK }  
|— Routes to usr-456 (if online):  
|   Erlang looks up which node holds usr-456  
|   If Node 1 → delivers directly  
|   If Node 2 → Erlang distributed message (no Redis needed)  
|  
|— Publishes to RabbitMQ: chat.message.inbound  
|   {  
|     from: "usr-123@nexgate.com",  
|     to: "usr-456@nexgate.com",  
|     conv_id: "conv-789",  
|     body: "Habari",  
|     temp_id: "abc-123",  
|     timestamp: 1719446400  
|   }  
|  
|  
|  
| (async)  
|
```

[Spring Boot Chat Service]

```
| consumes chat.message.inbound
```

```
|
|— Authorization check (can A message B?)
|— Resolve message level
|— Write to PostgreSQL (messages table)
|— Write to Redis hot cache (last 50 per conv)
|— Update conversation last_message
|
|— usr-456 online? (check via Ejabberd REST API)
|   YES → DELIVERED receipt after Ejabberd confirms
|   NO  → RabbitMQ offline queue + FCM + escalation timer
|
|— Notify sender: tick update
    REST API → Ejabberd → WS push to Client A
    Client A: ✓✓ (delivered)
```

Cross-Node Routing — No Redis Pub/Sub Needed

Phase 1 (Spring Boot WS):

```
Pod 1 holds Client A connection
Pod 2 holds Client B connection
Redis pub/sub needed to bridge pods
Pod 1 publishes → Redis → Pod 2 delivers
```

Phase 2 (Ejabberd cluster):

```
Node 1 holds Client A connection
Node 2 holds Client B connection
Erlang distributed messaging bridges nodes
Node 1 → Erlang dist → Node 2 delivers
Redis pub/sub no longer needed for routing
(Redis still used by Chat Service for hot cache)
```

This is why Ejabberd can do 2M concurrent:

```
Erlang process per connection (~2KB RAM each)
Native cross-node routing built into the language
No external message bus overhead
```

9. Voice Calls

Components

Signaling:	Ejabberd Jingle (XEP-0166) coordinates call setup via XMPP stanzas
STUN:	Ejabberd built-in (mod_stun_disco) helps devices find their public IP behind NAT
TURN:	Coturn (separate VPS) relay when P2P impossible (EA carrier NAT)
Transport:	WebRTC in mobile app actual audio stream between devices
Codec:	Opus adaptive 6kbps (2G) → 64kbps (WiFi) echo cancellation + noise suppression built in non-negotiable for EA networks

Jingle Signaling Stanzas

```
<!-- Step 1: Kibuti initiates call to Juma -->
<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="session-abc-123"
    initiator="kibuti@nexgate.com/android">
    <content name="audio">
      <description xmlns="urn:xmpp:jingle:apps:rtp:1"
        media="audio">
        <payload-type id="111" name="opus" clockrate="48000"/>
      </description>
      <transport xmlns="urn:xmpp:jingle:transports:ice-udp:1"
```

```

        ufrag="someUfrag"
        pwd="somePassword">
        <candidate ... /> <!-- Kibuti's ICE candidates -->
    </transport>
</content>
</jingle>
</iq>

<!-- Step 2: Juma accepts -->
<iq from="juma@nexgate.com/android"
    to="kibuti@nexgate.com/android"
    type="set" id="call-002">
    <jingle action="session-accept"
        sid="session-abc-123">
        <!-- Juma's SDP answer + ICE candidates -->
    </jingle>
</iq>

<!-- Step 3: Call ends -->
<iq type="set">
    <jingle action="session-terminate"
        sid="session-abc-123">
        <reason><success/></reason>
    </jingle>
</iq>

```

Full Voice Call Flow

```

[Kibuti – taps Call]
|
▼ GET /chat/calls/turn-credentials
Spring Boot returns:
{
  iceServers: [
    { urls: "stun:chat.nexgate.com:3478" },
    { urls: "turn:turn.nexgate.com:3478",
      username: "usr-123:1719446400",
      credential: "hmac_token" }
  ],

```

```
ttl: 3600
}
|
▼ Initialize WebRTC PeerConnection
Add audio track (Opus codec)
Gather ICE candidates (STUN discovery)
|
▼ Send Jingle session-initiate via Ejabberd WS
Ejabberd routes to Juma
Ejabberd fires RabbitMQ event: chat.call.initiated
|
Spring Boot:
  Create call record:
    status: RINGING
    started_at: now
  If Juma offline → FCM HIGH priority:
    { type: INCOMING_CALL, callId, callerName, callType: VOICE }
    |
[Juma's phone rings]
Juma taps Answer
|
▼ Juma sends Jingle session-accept via Ejabberd WS
ICE negotiation begins between devices:
|
├─ P2P possible? (good network)
|   Direct connection established ☐
|   No Coturn bandwidth used
|
└─ P2P blocked? (EA carrier NAT)
    Both connect to Coturn relay
    Audio flows: Kibuti → Coturn → Juma
|
Call live ☐☐
RTCP monitors quality every 200ms:
  Good network    → Opus 32-64kbps, clear voice
  3G              → Opus 16kbps, still good
  2G              → Opus 8kbps, slightly robotic but connected
  Very poor       → Opus 6kbps, minimum viable
|
```

```
Kibuti taps End
|
▼ Jingle session-terminate via Ejabberd WS
Ejabberd fires: chat.call.ended
Spring Boot:
  Update call record:
    status: COMPLETED
    ended_at: now
    duration_seconds: calculated
    relay_used: true/false
```

Call State Machine

```
IDLE
| user taps Call
▼
INITIATING
| getting TURN credentials
| creating WebRTC offer
▼
RINGING → MISSED (45s timeout)
| Jume answers
▼
CONNECTING
| ICE negotiation
| finding best path
▼
CONNECTED → RECONNECTING (network drop)
| call live | ICE restart
| | 10s timeout → FAILED
| user ends
▼
ENDING
| Jingle terminate sent
▼
COMPLETED / MISSED / DECLINED / FAILED
```

Codec Ladder — Opus Adaptive

Network	Bitrate	Quality
WiFi / 4G strong	64 kbps	HD voice
4G standard	32 kbps	Clear
3G	16 kbps	Good enough
2G / Edge	8 kbps	Robotic but connected
Barely alive	6 kbps	Minimum viable

Opus switches automatically based on RTCP feedback

No configuration needed – adaptive by design

10. Video Calls

Same Architecture as Voice + Camera

Everything from voice call applies

Additional components:

Video codec: H.264 (primary)
 Hardware accelerated on Tecno, Infinix, Samsung
 Low battery drain – GPU handles encoding
 Fallback: VP8 (software, more CPU)

Camera: Front camera default (switchable)
 Device detects capability at call start

Resolution ladder (adaptive):

WiFi	720p	30fps	1.5 Mbps
4G strong	480p	24fps	800 kbps
3G	360p	15fps	400 kbps
2G	240p	10fps	150 kbps
Very poor	VIDEO OFF – audio only (Opus)		

Degradation order (never drops call):

1. Reduce color depth

2. Reduce resolution (720→480→360→240)
3. Reduce frame rate (30→24→15→10fps)
4. Reduce audio bitrate
5. Disable video entirely → audio only
6. Reduce audio to minimum (6kbps Opus)

Device Tier Detection

App detects device capability at call start:

High-end (Pixel, Samsung S series):

H.264 hardware encoder (GPU)

Start at 720p 30fps

Low battery impact

Mid-range (Samsung A series):

H.264 hardware encoder

Start at 480p 24fps

Medium battery impact

Low-end (Tecno Spark, Infinix Hot):

H.264 software encoder (CPU)

Start at 360p 15fps

High battery impact

Show warning: "Video call may drain battery faster"

Auto-disable video after 10min if battery < 20%

Jingle for Video — Additional Content Block

```
<!-- Video call adds video content block -->
<jingle action="session-initiate" sid="session-xyz">

  <!-- Audio block (same as voice) -->
  <content name="audio">
    <description media="audio">
      <payload-type name="opus" clockrate="48000"/>
    </description>
    <transport .../>
  </content>
```

```
<!-- Video block (added for video calls) -->
<content name="video">
  <description media="video">
    <payload-type id="96" name="H264" clockrate="90000"/>
    <payload-type id="97" name="VP8" clockrate="90000"/>
  </description>
  <transport .../>
</content>

</jingle>
```

11. Coturn — TURN Relay

Why TURN is Mandatory for EA

Direct P2P (ideal):

- Both devices negotiate directly
- Audio/video flows device-to-device
- Ejabberd not involved in media
- No bandwidth cost on your servers

EA reality – P2P often blocked:

- Vodacom, Airtel, Tigo use CGNAT
- Multiple users share one public IP
- P2P connection cannot be established
- Without TURN → call fails

TURN relay (fallback):

- Both devices connect to Coturn
- Coturn relays audio/video between them
- Call works regardless of carrier NAT
- Bandwidth cost on your server (~50KB/min voice)

Coturn Config Highlights

```
listening-port=3478
tls-listening-port=5349
relay-ip=YOUR_COTURN_VPS_IP
realm=nexgate.com
lt-cred-mech          # time-limited credentials
use-auth-secret
static-auth-secret=${COTURN_SECRET}  # from Vault
min-port=49152
max-port=65535
```

TURN Credentials Generation

Credentials are time-limited HMAC tokens
Generated by Spring Boot per call session
Coturn validates them – prevents abuse

Format:

```
username: {userId}:{expiry_timestamp}
credential: HMAC-SHA1(secret, username)
ttl: 3600 seconds (1 hour per call)
```

Only NexGate users can use your TURN server
No credential → Coturn rejects connection

Bandwidth Estimation

Voice call via TURN:

```
Opus 16kbps × 2 directions = ~4KB/min
1 hour call ≈ 240KB per participant
```

Video call via TURN:

```
360p H.264 × 2 directions = ~6MB/min
Force 360p max when on relay to control cost
```

Coturn VPS sizing:

```
Hetzner CX11 (€4/month, 1vCPU/2GB)
20TB bandwidth included
Handles ~500 concurrent voice relay calls
```

12. MessagePack Encoding

Why Switch from JSON

JSON message frame:

```
{"type":"MSG_SEND","conv_id":"conv-123456","sender_id":"usr-789012",  
  "body":"Habari","timestamp":1719446400000,"temp_id":"abc-def-ghi"}
```

Size: ~140 bytes

Every key repeated as string on every message

Numbers encoded as ASCII characters

Parsing: character by character

MessagePack same message:

[binary representation]

Size: ~50 bytes

Keys encoded as integers (schema registered)

Numbers encoded as actual bytes (int32 = 4 bytes)

Parsing: read fixed byte positions

Result:

60-65% smaller on wire

3-5x faster to parse

Critical for users on 2G/3G with limited data bundles

Migration Strategy (No Breaking Change)

Both formats supported simultaneously:

Client sends header:

Content-Type: application/msgpack → MessagePack

Content-Type: application/json → JSON (default)

Ejabberd detects Content-Type
Routes to appropriate deserializer

Migration flow:

Old app version → sends JSON → works fine
New app version → sends MessagePack → works fine
No forced update required
Gradual migration over 30-60 days
Remove JSON support after 90%+ adoption

13. Broadcast Channels

What They Are

Creator → unlimited followers
One-directional: creator posts, followers receive
Like Telegram channels
No replies from followers (unless creator enables Q&A)

Use cases:

Shop announcement channel (\$techstore updates)
Creator content channel (@kibuti posts)
NexGate system channel (platform announcements)

Fan-out Strategy

Small channel (< 10,000 followers):
Write-on-send – Chat Service pushes to each follower
Same as group chat fan-out

Large channel (10,000+ followers):
Lazy fan-out – store message once
Followers fetch on open (read-time delivery)
No per-follower push for casual followers
FCM push only to followers with notifications enabled

Same celebrity bypass pattern as VP Feed:

Hot channels → read-time merge

Normal channels → write-time fan-out

Ejabberd MUC for Channels

Broadcast channel = MUC room with restrictions:

Only owner/admins can send messages

Members are read-only subscribers

mod_muc handles this with role configuration:

Role: moderator → can send

Role: visitor → read only

This means channels are built on the
same MUC infrastructure as group chats
No separate implementation needed

14. MQTT — Mini Apps Foundation

What MQTT Enables

Ejabberd runs MQTT broker on port 1883

No extra infrastructure – already in Ejabberd

Mini Apps subscribe to topics:

orders/{orderId} → real-time order updates

delivery/{trackingId} → GPS delivery tracking

live/{streamId}/viewers → viewer count updates

jiko/{restaurantId} → JikoXpress kitchen events

Spring Boot publishes events:

Order shipped → publish to orders/{orderId}

Mini App receives instantly

No polling needed

MQTT vs XMPP for Mini Apps

XMPP (chat):

- Full protocol, complex stanzas
- Designed for human conversation
- Bidirectional, stateful sessions
- Right tool for chat

MQTT (events):

- Lightweight pub/sub protocol
- Designed for IoT and event streams
- Minimal overhead (2-byte header)
- Right tool for Mini App events
- Works on very limited connections

Both live inside Ejabberd:

- Same server, different protocols
- Mobile app uses XMPP for chat
- Mini Apps use MQTT for events
- Zero additional infrastructure

15. Message Interactions

All message interaction features are handled via standard XMPP XEPs. Ejabberd routes the stanzas automatically — Spring Boot handles persistence and business rules via RabbitMQ events.

Overview — All Four Features

Feature	XEP	Status	Ejabberd
Edit message	XEP-0308	Stable ☐	auto routed
Delete for everyone	XEP-0424	Stable ☐	auto routed
Reactions	XEP-0444	Stable ☐	auto routed
Forwarding	XEP-0297	Stable ☐	auto routed
Reply to message	XEP-0461	Experimental	auto routed
Stable stanza IDs	XEP-0359	Stable ☐	auto assigned

All routed by Ejabberd

Spring Boot handles: validation, persistence, rules

XEP-0359 — Stable Stanza IDs (Foundation)

Before the features — this XEP is the foundation all others depend on. Every message gets a stable server-assigned ID used by reactions, edits, retractions, and replies to reference the correct message.

```
<!-- Ejabberd automatically adds stanza-id to every message -->
<message from="kibuti@nexgate.com"
  to="juma@nexgate.com"
  id="client-generated-id">
  <body>Habari</body>
  <stanza-id xmlns="urn:xmpp:sid:0"
    id="server-stable-id-abc123"
    by="nexgate.com"/>
  <!-- server-stable-id-abc123 is what reactions/edits reference -->
</message>
```

Message Editing — XEP-0308

Who can edit: Original sender only
Time window: 15 minutes after send
What: Text body only
Commerce cards: ☐ BLOCKED — financial records are immutable
System messages: ☐ BLOCKED — never editable

```
<!-- Kibuti edits his message -->
<message from="kibuti@nexgate.com"
  to="juma@nexgate.com"
  type="chat"
  id="edit-002">

  <body>Habari yako Juma, vipi biashara?</body>

  <replace xmlns="urn:xmpp:message-correct:0"
    id="server-stable-id-abc123"/>
  <!-- references original message by stanza-id -->

</message>
```

Flow:

Kibuti edits → stanza sent via Ejabberd WS

Ejabberd routes to Juma (if online)

Ejabberd fires RabbitMQ: chat.message.edited

|

Spring Boot:

Is sender original author? ☐

Within 15 minute window? ☐

Not a commerce/system message? ☐

Update messages.body = new text

Update messages.edited_at = now

Increment messages.edit_count

|

Juma's app:

Receives edit stanza

Updates message in place (same position in thread)

Shows "Edited" label under message

Group chats:

Same stanza sent to MUC room JID

Ejabberd MUC broadcasts to all members

All see updated message simultaneously

Delete for Everyone — XEP-0424

Two delete modes:

Delete for me:

Local filter only

No Ejabberd stanza needed

Spring Boot records: message_deletions (scope: SELF)

Recipient unaffected

Delete for everyone:

XEP-0424 retraction stanza

Ejabberd routes to all recipients

Time window: 15 minutes

Commerce cards: ☐ BLOCKED

System messages: ☐ BLOCKED

```
<!-- Delete for everyone – retraction stanza -->
<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-abc123">
    <retract xmlns="urn:xmpp:message-retract:1"/>
  </apply-to>

</message>
```

Flow:

Kibuti retracts → stanza via Ejabberd WS

Ejabberd routes to Juma

Ejabberd fires RabbitMQ: chat.message.retracted

|

Spring Boot:

Is sender original author? ☐

Within 15 minute window? ☐

Not blocked message type? ☐

Soft delete:

messages.deleted_at = now

messages.deleted_by = usr-kibuti

messages.delete_scope = EVERYONE

body NOT removed (audit trail kept)

|

Juma's app:

Receives retraction stanza

Replaces message with:

"This message was deleted"

Same position in thread

Nothing is ever hard deleted from PostgreSQL:

Legal compliance (EA regulations)

Dispute resolution (order/payment disputes)

Admin investigation (fraud cases)

Reactions — XEP-0444

Model:	One reaction per user per message
Emoji set:	Limited set at launch ♥️ 🍌 🍌 🍌 🍌 🍌
Change:	Send new emoji → replaces old
Remove:	Send empty → removes reaction
Commerce cards:	❌ ALLOWED (reactions don't modify content)
System messages:	❌ BLOCKED

```
<!-- Kibuti reacts 🍌to message -->
<message from="kibuti@nexgate.com"
  to="juma@nexgate.com"
  type="chat"
  id="reaction-001">

  <reactions xmlns="urn:xmpp:reactions:0"
    id="server-stable-id-abc123">
    <reaction>🍌/reaction>
  </reactions>

</message>

<!-- Kibuti changes to ♥️ -->
<message ...>
  <reactions xmlns="urn:xmpp:reactions:0"
    id="server-stable-id-abc123">
    <reaction>♥️</reaction>
  </reactions>
</message>

<!-- Kibuti removes reaction -->
<message ...>
  <reactions xmlns="urn:xmpp:reactions:0"
    id="server-stable-id-abc123">
    <!-- empty = removed -->
```

```
</reactions>
</message>
```

Flow:

Kibuti taps 🍌 → reaction stanza via Ejabberd WS

Ejabberd routes to Juma

Ejabberd fires RabbitMQ: chat.message.reaction

|

Spring Boot:

Upsert in message_reactions:

ON CONFLICT (message_id, user_id)

→ update emoji + timestamp

Empty emoji received → delete reaction record

|

Juma's app:

Receives reaction stanza

Updates reaction display below message:

🍌1

Kibuti's own reaction: highlighted

Group chats:

Stanza sent to MUC room

Ejabberd MUC broadcasts to all members

All screens update simultaneously:

🍌3 🍌 2 🍌1

Notification:

Reaction on your message → FCM push

"Juma reacted 🍌to your message"

Level: NORMAL (FCM only – no SMS)

Muted conversations → no reaction notification

Message Forwarding — XEP-0297

What it is:

Client creates NEW message in target conversation

Original message wrapped inside as reference

Server never "moves" anything

Forwarded label shown with original sender name

Forward chain tracking:

chain = 1: "↩ Forwarded from Juma Mwangi"

chain = 2-4: "↩ Forwarded"

chain = 5+: "↩ Forwarded many times" (misinformation warning)

Multi-forward: up to 5 conversations per action

Max chain: no hard limit but UI degrades label

Commerce rules:

Product card: ☐ anyone can forward

Custom price offer: ☐ private deal – blocked

Order confirmation: ☐ private record – blocked

Payment record: ☐ private record – blocked

System messages: ☐ blocked

```
<!-- Kibuti forwards Juma's message to Amina -->
```

```
<message from="kibuti@nexgate.com"
```

```
  to="amina@nexgate.com"
```

```
  type="chat"
```

```
  id="fwd-001">
```

```
<body>Angalia hii</body>
```

```
<forwarded xmlns="urn:xmpp:forward:0">
```

```
  <delay xmlns="urn:xmpp:delay"
```

```
    stamp="2026-07-02T10:32:00Z"/>
```

```
<!-- original send time preserved -->
```

```
  <message from="juma@nexgate.com"
```

```
    to="kibuti@nexgate.com"
```

```
    type="chat"
```

```
    id="msg-original-001">
```

```
      <body>Habari yako rafiki!</body>
```

```
    </message>
```

```
</forwarded>
```

```
<!-- NexGate forward metadata -->
<nexgate-forward xmlns="urn:nexgate:forward">
  <original_sender_name>Juma Mwangi</original_sender_name>
  <forward_chain>1</forward_chain>
</nexgate-forward>
```

```
</message>
```

Flow:

Kibuti taps Forward on Juma's message

Picks Amina's conversation

App creates new message stanza (not routing original)

Sends via Ejabberd WS to Amina

Ejabberd routes normally as new message

Fires RabbitMQ: chat.message.inbound (same as any message)

|

Spring Boot:

Validates forward is allowed (type check)

Creates new messages record:

is_forwarded: true

original_sender_name: "Juma Mwangi"

forward_chain: 1

media_ref: original fileId (no re-upload)

|

Amina's app:

Receives as new message

Renders with forwarded label:

```
[
  | ↩ Forwarded from Juma Mwangi |
  |                               |
  | Habari yako rafiki!         |
  |                               |
  |                               | 10:45 |
]
```

Media forwarding:

References original fileId – no re-upload

10 people forward same image

→ 1 file in MinIO, 10 message records

File Thunder serves same file to all

Message Replies — XEP-0461

Reply to a specific message in thread

Like WhatsApp/Telegram quote-reply

Shows original message above reply

Status: Experimental 

Not yet stable standard

But widely implemented

(Gajim, Monal, many others use it)

Safe to implement – unlikely to change drastically

```
<!-- Juma replies to Kibuti's specific message -->
```

```
<message from="juma@nexgate.com"
```

```
  to="kibuti@nexgate.com"
```

```
  type="chat"
```

```
  id="reply-001">
```

```
<body>Nzuri sana, asante!</body>
```

```
<reply xmlns="urn:xmpp:reply:0"
```

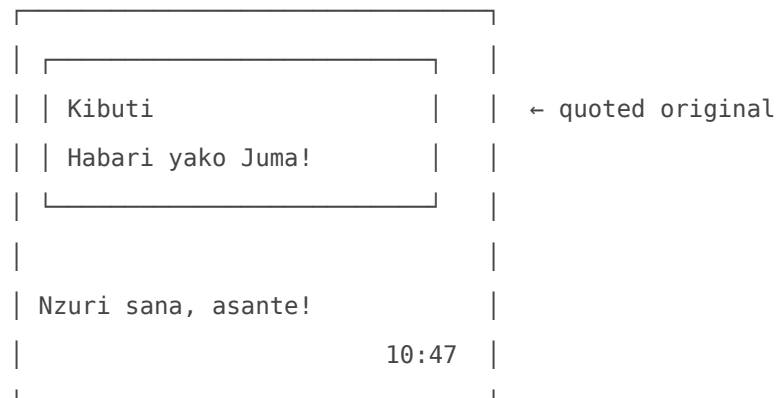
```
  to="kibuti@nexgate.com"
```

```
  id="server-stable-id-abc123"/>
```

```
<!-- id references the message being replied to -->
```

```
</message>
```

UI renders:



Tap on quote → scroll to original message

Commerce Messages — Interaction Rules Summary

Message type	Edit	Delete(all)	React	Forward
Text message	☐ 15m	☐ 15m	☐	☐
Voice note	☐	☐ 15m	☐	☐
Image / Video	☐	☐ 15m	☐	☐
Product card	☐	☐	☐	☐
Custom price offer	☐	☐	☐	☐
Order confirmation	☐	☐	☐	☐
Payment confirmation	☐	☐	☐	☐
System message	☐	☐	☐	☐

Why commerce cards are protected:

- Immutable negotiation record
- Seller cannot change agreed price after the fact
- Buyer cannot claim different price was offered
- Full audit trail in thread – legally important

RabbitMQ Events — New in Phase 2 for Interactions

Exchange: nexgate.chat (topic) – additions:

Routing Key	Fired When
chat.message.edited	XEP-0308 received
chat.message.retracted	XEP-0424 received
chat.message.reaction	XEP-0444 received
chat.message.forwarded	XEP-0297 received
chat.message.delete_self	delete for me (REST call)

16. Docker Deployment

docker-compose additions for Phase 2

```
ejabberd:
  image: ghcr.io/processone/ejabberd:latest
  container_name: ejabberd
  restart: unless-stopped
  ports:
    - "5222:5222"      # XMPP TCP
    - "5280:5280"      # WebSocket + HTTP
    - "1883:1883"      # MQTT
    - "3478:3478/udp"  # STUN
  volumes:
    - ./ejabberd/ejabberd.yml:/home/ejabberd/conf/ejabberd.yml
    - ./ejabberd/data:/home/ejabberd/database
    - ./ejabberd/logs:/home/ejabberd/logs
  environment:
    - EJABBERD_BYPASS_WARNINGS=true
  depends_on:
    - postgres
    - rabbitmq
  networks:
    - nexgate-internal

# Coturn on separate VPS – not in same compose
# Deployed independently on Hetzner CX11
# Connects back to NexGate via internal network
```

Traefik — WebSocket Routing

```
# Ejabberd service labels for Traefik

labels:
  - "traefik.enable=true"

# App connects here for chat
  - "traefik.http.routers.chat.rule=Host(`chat.nexgate.com`)"
```

```
- "traefik.http.routers.chat.tls=true"
- "traefik.http.routers.chat.tls.certresolver=letsencrypt"
- "traefik.http.services.chat.loadbalancer.server.port=5280"

# Sticky sessions – CRITICAL for WebSocket
# Same user must always hit same Ejabberd node
- "traefik.http.services.chat.loadbalancer.sticky.cookie=true"
- "traefik.http.services.chat.loadbalancer.sticky.cookie.name=ejabberd_node"
```

Why sticky sessions:

User connected to Ejabberd Node 1

Next request hits Node 2

→ connection context lost → disconnected

Sticky cookie ensures:

usr-123 always → Node 1

usr-456 always → Node 2

WS sessions stable across load balancer

Ejabberd Cluster Config

```
# Second node joins cluster
# On node 2's ejabberd.yml:

hosts:
  - "nexgate.com"

# Erlang cookie must match on all nodes
# Set via environment variable
# Both nodes discover each other automatically
# Erlang distributed handles the rest

# Result:
# Message to usr-456 arrives on Node 1
# usr-456 connected to Node 2
# Erlang routes internally – transparent
```

17. Database Schema

calls (new in Phase 2)

calls

call_id	UUID	PK
caller_id	UUID	
receiver_id	UUID	
conversation_id	UUID	FK → conversations
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 (new in Phase 2)

call_quality_logs

log_id	UUID	PK
call_id	UUID	FK → calls
timestamp	TIMESTAMPTZ	
direction	ENUM	OUTBOUND / INBOUND
bitrate_kbps	INT	
packet_loss_pct	DECIMAL	
jitter_ms	INT	
rtt_ms	INT	
resolution	TEXT	"360p" "480p" "720p" or null
codec_audio	TEXT	"opus"
codec_video	TEXT	"h264" "vp8" or null

broadcast_channels (new in Phase 2)

broadcast_channels

channel_id	UUID	PK
owner_id	UUID	userId or shopId
owner_type	ENUM	USER / SHOP
name	TEXT	
description	TEXT	
avatar_file_id	UUID	
subscriber_count	INT	
type	ENUM	PERSONAL / SHOP / SYSTEM
created_at	TIMESTAMPTZ	

message_reactions (new in Phase 2)

message_reactions

id	UUID	PK
message_id	UUID	FK → messages
conversation_id	UUID	FK → conversations
user_id	UUID	
emoji	TEXT	"👍" "❤️" "👍" etc
reacted_at	TIMESTAMPTZ	

Unique constraint: (message_id, user_id)

→ one reaction per user per message

→ upsert on conflict replaces emoji

message_deletions (new in Phase 2)

message_deletions

id	UUID	PK
message_id	UUID	FK → messages
deleted_by	UUID	userId
scope	ENUM	SELF / EVERYONE
deleted_at	TIMESTAMPTZ	

messages table additions (Phase 2)

New columns added to existing messages table:

edited_at	TIMESTAMPTZ	when last edited
edit_count	INT	how many times edited
original_body	TEXT	body before first edit (audit)
deleted_at	TIMESTAMPTZ	soft delete timestamp
deleted_by	UUID	who deleted
delete_scope	ENUM	SELF / EVERYONE
is_forwarded	BOOLEAN	was this forwarded
forward_chain	INT	forwarding depth (1,2,3...)
original_sender_name	TEXT	display name at forward time
original_message_id	UUID	source message if forwarded
reply_to_id	UUID	FK → messages (for replies)
stanza_id	TEXT	Ejabberd XEP-0359 stable ID

18. Commerce Stanzas & Custom Namespaces

The Extensible Part of XMPP

XMPP was designed to be extended by anyone for anything. The "X" in XMPP = Extensible.

Any application can add custom XML elements inside standard XMPP stanzas using their own namespace. Ejabberd routes the entire stanza as-is — it never parses, validates, or modifies custom elements. Spring Boot reads them on the other side.

Standard stanza:

```
<message from="a@nexgate.com" to="b@nexgate.com">
  <body>Habari</body>
</message>
```

With NexGate custom element:

```
<message from="a@nexgate.com" to="b@nexgate.com">
  <body>Habari</body>
  <nexgate-offer xmlns="urn:nexgate:offer:1">
    ... your custom data here ...
```

```
</nexgate-offer>
</message>
```

Ejabberd:

```
Routes whole stanza as-is []
Never touches nexgate-offer element []
Never validates it []
Just delivers it []
```

NexGate Namespace Registry

All custom namespaces NexGate defines:

urn:nexgate:commerce:1	product cards
urn:nexgate:offer:1	price offer sessions
urn:nexgate:groupbuy:1	Bei ya pamoja cards
urn:nexgate:event:1	event cards
urn:nexgate:feed:1	VP Feed post cards
urn:nexgate:live:1	live stream cards
urn:nexgate:audio:1	audio space cards
urn:nexgate:system:1	system messages
urn:nexgate:forward	forwarding metadata
urn:nexgate:states	recording voice note state
urn:nexgate:meta	message metadata

Versioning (:1, :2):

```
Allows schema evolution
Old app sees :1 → renders fine
New app sees :2 → renders richer UI
Old clients fall back to <body> text
No breaking changes
```

Product Card Stanza

Sent by: Spring Boot via Ejabberd REST API

When: Buyer taps "Chat with Seller" on product page

```
<message from="system@nexgate.com"
  to="techstore@shops.nexgate.com"
  type="chat"
  id="card-001">

<!-- Fallback for basic clients -->
<body>Mteja anaomba habari: Samsung A15</body>

<nexgate-commerce xmlns="urn:nexgate:commerce:1">
  <type>PRODUCT_CARD</type>
  <initiated_by>usr-kibuti</initiated_by>
  <conv_id>conv-789</conv_id>

  <product>
    <id>prod-123</id>
    <name>Samsung A15</name>
    <public_price>450000</public_price>
    <currency>TZS</currency>
    <image_url>https://cdn.nexgate.com/img.jpg</image_url>
    <stock>12</stock>
    <shop_name>TechStore</shop_name>
    <shop_id>shop-456</shop_id>
    <snapshot_at>2026-07-13T08:30:00Z</snapshot_at>
    <!-- price frozen at this moment – never changes -->
  </product>
</nexgate-commerce>

</message>
```

Seller's app renders:

📱 Samsung A15
💰 TZS 450,000
📍 Inapatikana: Vipande 12
🏪 TechStore
👤 [Jibu] [Angalia Bidhaa]

Custom Price Offer Stanza

Sent by: Seller's app via Ejabberd WebSocket

When: Seller attaches price offer from shop
(both Flow 1 post-negotiation and Flow 2 direct attach)

```
<message from="techstore@shops.nexgate.com/amina"
  to="kibuti@nexgate.com"
  type="chat"
  id="offer-002">

<body>Bei yako maalum: TZS 400,000</body>

<nexgate-offer xmlns="urn:nexgate:offer:1">
  <offer_id>offer-uuid-abc</offer_id>
  <conv_id>conv-789</conv_id>
  <valid_minutes>30</valid_minutes>
  <initiated_by>SELLER</initiated_by>

  <product>
    <id>prod-123</id>
    <name>Samsung A15</name>
    <image_url>https://cdn.nexgate.com/img.jpg</image_url>
    <shop_name>TechStore</shop_name>
    <shop_id>shop-456</shop_id>
  </product>

  <pricing>
    <public_price>450000</public_price>
    <offer_price>400000</offer_price>
    <currency>TZS</currency>
    <discount_amount>50000</discount_amount>
    <discount_pct>11</discount_pct>
  </pricing>

  <!-- Staff who sent offer – not visible to buyer -->
  <!-- Buyer always sees "TechStore" not "Amina" -->
  <sent_by_staff>usr-amina</sent_by_staff>
```

```
</nexgate-offer>
```

```
</message>
```

Buyer's app renders:

	Bei Maalum Kwako	
	Samsung A15	
	~~TZS 450,000~~	
	TZS 400,000 (umepunguziwa 50,000)	
	Inaisha: dakika 30	
	Idadi: [- 1 +]	
	[Kataa] [Endelea Kulipa →]	

Offer Response Stanzas

```
<!-- Buyer declines offer -->
```

```
<message from="kibuti@nexgate.com"
  to="techstore@shops.nexgate.com"
  type="chat"
  id="resp-003">
```

```
<body>Nimekataa bei hii</body>
```

```
<nexgate-offer xmlns="urn:nexgate:offer:1">
```

```
<offer_id>offer-uuid-abc</offer_id>
```

```
<response>DECLINED</response>
```

```
</nexgate-offer>
```

```
</message>
```

```
<!-- System sends expiry notification -->
```

```
<message from="system@nexgate.com"
  to="conv-789-participants"
  type="chat"
  id="expire-004">
```

```
<nexgate-offer xmlns="urn:nexgate:offer:1">
  <offer_id>offer-uuid-abc</offer_id>
  <response>EXPIRED</response>
  <message_id>offer-002</message_id>
  <!-- references offer card message to update its UI -->
</nexgate-offer>

</message>
```

Order Confirmation Stanza

Sent by: Spring Boot via Ejabberd REST API
When: Buyer completes checkout successfully

```
<message from="system@nexgate.com"
  to="conv-789-participants"
  type="chat"
  id="confirm-005">

  <body>Agizo limefanikiwa!</body>

  <nexgate-system xmlns="urn:nexgate:system:1">
    <type>ORDER_CONFIRMATION</type>
    <order_id>ord-xyz-789</order_id>
    <conv_id>conv-789</conv_id>
    <offer_id>offer-uuid-abc</offer_id>

    <summary>
      <product_name>Samsung A15</product_name>
      <quantity>1</quantity>
      <amount_paid>400000</amount_paid>
      <currency>TZS</currency>
      <status>CONFIRMED</status>
    </summary>

  </nexgate-system>

</message>
```

Both buyer and seller see:

👤 Agizo Limethibitishwa	
Ord #ORD-XYZ-789	
Samsung A15 × 1	
TZS 400,000 imelipwa	
[Fuatilia Agizo]	

Bei ya Pamoja Card Stanza

```
<message from="kibuti@nexgate.com"
  to="juma@nexgate.com"
  type="chat"
  id="gb-006">

<body>Jiunge na group buy hii!</body>

<nexgate-groupbuy xmlns="urn:nexgate:groupbuy:1">
  <group_buy_id>gb-xyz</group_buy_id>
  <product_id>prod-123</product_id>
  <product_name>Samsung A15</product_name>
  <product_image>https://cdn.nexgate.com/img.jpg</product_image>
  <group_price>350000</group_price>
  <public_price>450000</public_price>
  <currency>TZS</currency>
  <current_participants>7</current_participants>
  <target_participants>10</target_participants>
  <expires_at>2026-07-13T18:00:00Z</expires_at>
</nexgate-groupbuy>

</message>
```

Event Card Stanza

```
<message from="kibuti@nexgate.com"
  to="juma@nexgate.com"
```

```
type="chat"
id="evt-007">
```

```
<body>Jiunge na event hii!</body>
```

```
<nexgate-event xmlns="urn:nexgate:event:1">
  <event_id>evt-456</event_id>
  <title>Dar Tech Summit 2026</title>
  <date>2026-08-15T09:00:00Z</date>
  <venue>Julius Nyerere ICC, Dar es Salaam</venue>
  <ticket_price>25000</ticket_price>
  <currency>TZS</currency>
  <cover_image>https://cdn.nexgate.com/evt.jpg</cover_image>
  <available_tickets>150</available_tickets>
</nexgate-event>
```

```
</message>
```

VP Feed Post Card Stanza

```
<message from="kibuti@nexgate.com"
  to="juma@nexgate.com"
  type="chat"
  id="post-008">
```

```
<body>Angalia post hii</body>
```

```
<nexgate-feed xmlns="urn:nexgate:feed:1">
  <post_id>post-789</post_id>
  <author_name>Kibuti Mwangi</author_name>
  <author_avatar>https://cdn.nexgate.com/av.jpg</author_avatar>
  <caption>Bidhaa mpya zimefika! 📦</caption>
  <media_url>https://cdn.nexgate.com/post.jpg</media_url>
  <media_type>IMAGE</media_type>
  <like_count>245</like_count>
</nexgate-feed>
```

```
</message>
```

Spring Boot — How It Handles Custom Stanzas

All stanzas arrive via RabbitMQ: `chat.message.inbound`

Spring Boot parses XML and routes by namespace:

Namespace detected	Handler
<code>urn:nexgate:commerce:1</code>	<code>handleProductCard()</code>
<code>urn:nexgate:offer:1</code>	<code>handleOfferSession()</code>
<code>urn:nexgate:groupbuy:1</code>	<code>handleGroupBuy()</code>
<code>urn:nexgate:event:1</code>	<code>handleEventCard()</code>
<code>urn:nexgate:feed:1</code>	<code>handlePostCard()</code>
<code>urn:nexgate:system:1</code>	<code>handleSystemMessage()</code>
none of the above	<code>handleTextMessage()</code>

Offer Session — Spring Boot Processing

`CUSTOM_PRICE_OFFER` received:

Spring Boot:

Create message record (type: `CUSTOM_PRICE_OFFER`)

Create `commerce_offer_sessions` record:

`offer_id:` from stanza
`buyer_id:` conversation partner
`shop_id:` sender shop JID
`product snapshot:` from stanza
`offer_price:` from stanza (server authoritative)
`expires_at:` now + `valid_minutes`
`status:` PENDING

Schedule RabbitMQ delayed job:

`delay:` `valid_minutes`
`payload:` { `offerId`, `action:` EXPIRE }

Send FCM to buyer:

"TechStore amekutumia bei maalum"
Level: IMPORTANT

Buyer taps "Endelea Kulipa":

POST `/checkout/initiate` { `offerId`, `quantity` }

Spring Boot:

Validate: status=PENDING, not expired, buyer matches

Update status: CHECKOUT

Price from DB – never from client []

Return: { checkoutUrl, checkoutToken }

Order completes:

Update status: COMPLETED

order_id: linked

Send ORDER_CONFIRMATION stanza to conversation

Expiry fires (RabbitMQ delayed job):

Status still PENDING? → mark EXPIRED

Status already changed? → do nothing

Send OFFER_EXPIRED stanza to conversation

commerce_offer_sessions Table

commerce_offer_sessions

offer_id	UUID	PK
conv_id	UUID	FK → conversations
message_id	UUID	FK → messages
shop_id	UUID	
buyer_id	UUID	
sent_by_staff	UUID	staff who sent (audit only)
product_id	UUID	
product_name	TEXT	
product_image_url	TEXT	
snapshot_json	JSONB	full product at offer time
public_price	BIGINT	TZS
offer_price	BIGINT	TZS (custom – server auth)
currency	TEXT	TZS
quantity_min	INT	
quantity_max	INT	
discount_amount	BIGINT	
discount_pct	DECIMAL	
status	ENUM	PENDING / ACCEPTED /

DECLINED / EXPIRED /
CHECKOUT / COMPLETED /
CANCELLED / ABANDONED

valid_minutes	INT	
expires_at	TIMESTAMPTZ	
initiated_by	ENUM	BUYER / SELLER
notes	TEXT	
created_at	TIMESTAMPTZ	
responded_at	TIMESTAMPTZ	
checkout_at	TIMESTAMPTZ	
completed_at	TIMESTAMPTZ	
order_id	UUID	FK → orders (after completion)

19. Build Order

NexGate chat is built from scratch — no migration, no Phase 1 to carry forward. This is the recommended sequence:

Week 1-2 – Local Experiments

- Ejabberd running in Docker locally
- Two containers (node1 + node2) clustered
- Auth bridge: Spring Boot validates XMPP tokens
- Send first message between two test JIDs
- Confirm Erlang dist working between nodes
- Confirm RabbitMQ events firing to Spring Boot

Week 3-4 – PostgreSQL Schema + Chat Service

- All tables created (messages, conversations, receipts, calls, offer sessions, reactions etc)
- Spring Boot Chat Service:
 - RabbitMQ consumers for all Ejabberd events
 - Message persistence
 - Receipt tracking
 - Notification routing (FCM + Textfy)

Week 5 – Ejabberd Staging Deployment

- Deploy to Hetzner staging VPS
- Two node cluster live

Traefik sticky sessions configured
Auth bridge connected to Chat Service
Send first real message through staging Ejabberd

Week 6-7 – Mobile SDK + Basic Chat

NexGate Chat SDK (Android + iOS)
Smack / XMPPFramework wrapper
Clean send/receive API
Auto-reconnect + Stream Management
Text messages working end-to-end
Typing indicators
Delivery + read ticks
Presence (online/offline)

Week 8 – Message Interactions

Reactions (XEP-0444)
Edit messages (XEP-0308)
Delete for everyone (XEP-0424)
Forwarding (XEP-0297)
Replies (XEP-0461)

Week 9 – Voice Calls

TURN credentials endpoint in Spring Boot
Coturn deployed (Hetzner CX11)
Jingle signaling through Ejabberd
WebRTC on Android/iOS
Opus audio confirmed on 2G test
Coturn relay confirmed on EA network

Week 10 – Video Calls

H.264 video track added
Adaptive resolution ladder
Resolution ladder tested on 3G

Week 11 – Commerce DMs

Custom namespace stanzas:
PRODUCT_CARD
CUSTOM_PRICE_OFFER
OFFER_DECLINED / OFFER_EXPIRED
ORDER_CONFIRMATION

Offer session lifecycle

Both commerce flows (buyer initiates + seller attaches)

Checkout redirect flow

Shop inbox isolation + access control

Week 12 – Group Chats + Broadcast

MUC rooms (Ejabberd XEP-0045)

Group message reactions

Group typing indicators

Broadcast channels (read-only MUC)

Week 13 – Offline + Notifications

RabbitMQ offline queue

FCM HIGH priority integration

Textfy SMS escalation

Notification levels (NORMAL/IMPORTANT/CRITICAL)

Catch-up banner on reconnect

Week 14 – MessagePack

MessagePack encoding in SDK

Content-Type header detection in Ejabberd

Both JSON + MessagePack supported simultaneously

EA bandwidth savings confirmed

Week 15 – Testing + EA Network Testing

Test on actual Vodacom/Airtel SIM cards

Test on Tecno/Infinix devices

Test on 2G/3G networks

Call quality on Coturn relay confirmed

Commerce flow end-to-end confirmed

Week 16 – Ship ☐☐

Production deployment

Two Ejabberd nodes live

All features confirmed

NexGate chat is live

Summary

NexGate chat is built directly on Phase 2 architecture from scratch. No migration. No legacy code. Greenfield build on carrier-grade infrastructure from day one.

Ejabberd Cluster (two Docker containers, same Hetzner VPS at launch) handles all WebSocket connections, XMPP stanza routing, presence, MUC group chats, Jingle voice/video signaling, and 25+ chat features via standard XEPs — all at zero custom code cost. Erlang Distribution connects the two nodes directly, routing messages between them in microseconds without Redis pub/sub.

Spring Boot Chat Service owns all business logic — message persistence, commerce context, offer session lifecycle, shop inbox access control, notification routing, and call records. It communicates with Ejabberd asynchronously via RabbitMQ for all events except auth, which is synchronous HTTP because Ejabberd needs an immediate allow/deny decision.

Custom XMPP Namespaces extend the protocol for NexGate's commerce features. Product cards, custom price offers, offer session responses, Bei ya pamoja cards, event cards, and post cards all travel as custom XML elements inside standard XMPP stanzas. Ejabberd routes them as-is — Spring Boot parses and handles them. Commerce messages are server-authoritative and immutable: offer prices come from the database, not the client. Public product prices are never touched.

WebRTC + Coturn handles voice and video calls. Jingle stanzas through Ejabberd coordinate setup. Opus adapts audio from 64kbps on WiFi to 6kbps on 2G. H.264 hardware acceleration keeps video calls battery-friendly on EA phones. Coturn relay ensures calls work behind EA carrier NAT on Vodacom, Airtel, and Tigo.

MessagePack reduces message frame size 60-65% — real saving for EA users on limited data bundles. Both JSON and MessagePack supported simultaneously during SDK rollout.

The build is 16 weeks from local experiments to production. WhatsApp-class infrastructure. Commerce-aware from day one. EA network optimized throughout.

NexGate Chat Platform — Phase 2: Production Architecture v1.0 QBIT SPARK | Ejabberd · Coturn · WebRTC · Commerce Stanzas · Edit · Delete · React · Forward

Revision #11

Created 13 July 2026 06:34:02 by Admin Qbit

Updated 14 July 2026 06:57:18 by Admin Qbit