Convergo & Salebase - Complete Technical Documentation

Project Owner: Amadi Chibuzor
Location: Cremona, Lombardy, Italy
Date: February 23, 2026
Status: Active Development (Phase 3 - Prototype)

Executive Summary

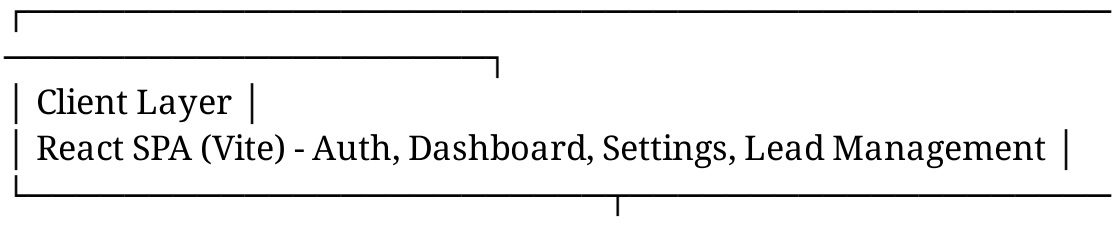
Convergo is a multi-tenant conversational AI platform that enables businesses to automate customer messaging across Instagram and Facebook using Meta's Graph API and large language models (OpenAI + Naija LLM). The platform handles OAuth authentication, webhook-based message routing, AI-powered response generation, and provides analytics dashboards for monitoring conversation metrics.

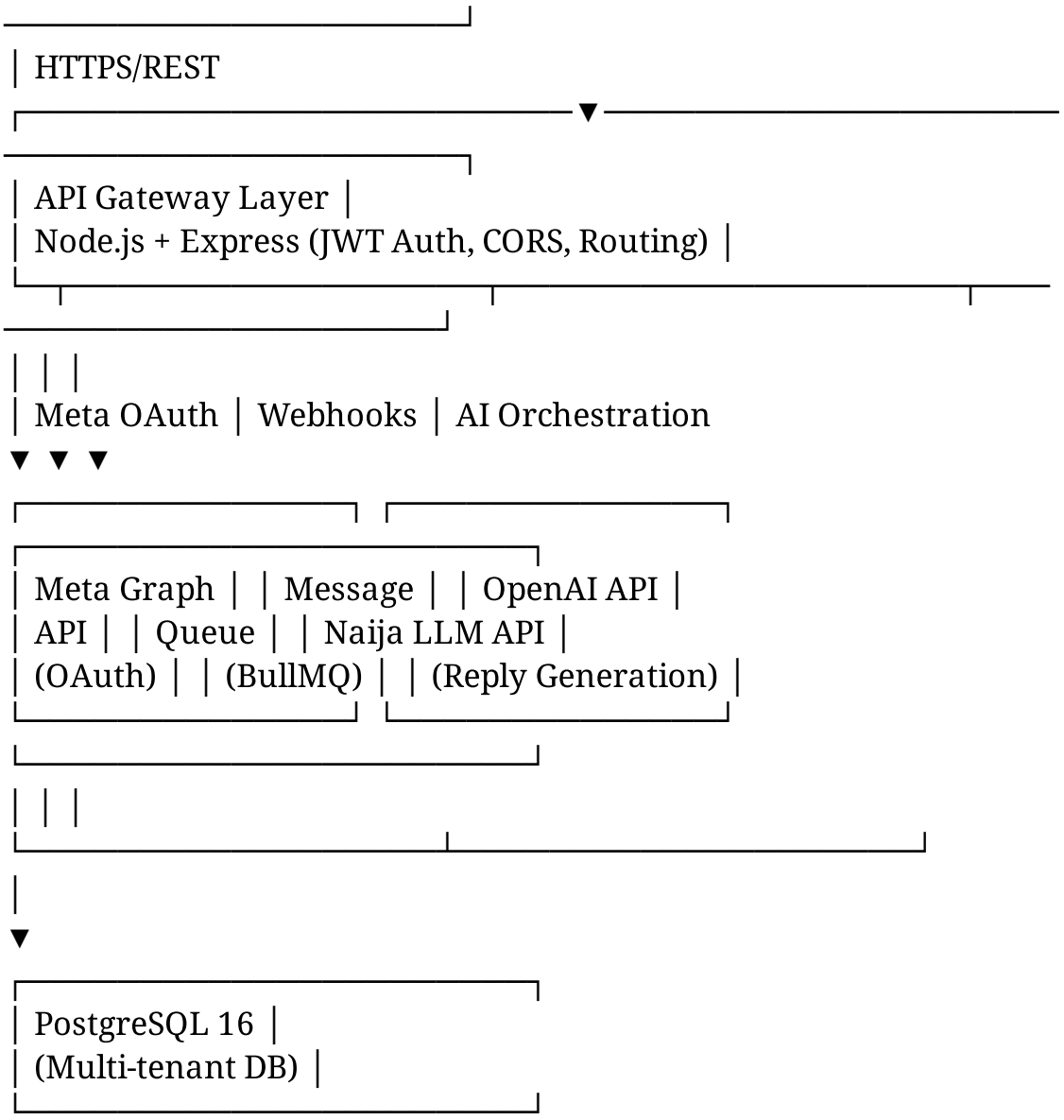
Salebase extends Convergo with sales-focused capabilities including lead capture, AI-driven lead qualification, pipeline management, and automated follow-up generation across email and WhatsApp channels.

Core Value Proposition: Businesses connect their Meta accounts → AI automatically handles customer messages → Sales teams receive qualified leads with scoring and suggested actions.

Architecture Overview

System Architecture





Technology Stack

Backend Infrastructure

- **Runtime:** Node.js 20+ with TypeScript support
- **Framework:** Express 4.x
- **Database:** PostgreSQL 16 with pg-promise driver
- **Authentication:** JSON Web Tokens (JWT) with bcrypt password hashing
- **Queue System:** BullMQ with Redis for async job processing
- **Logging:** Winston for structured logging
- **HTTP Client:** Axios for external API calls

Frontend Infrastructure

- **Framework:** React 18 with Vite build tool
- **Routing:** React Router DOM v6
- **State Management:** Zustand (lightweight alternative to Redux)
- **Forms:** React Hook Form with validation
- **Styling:** Tailwind CSS with custom components
- **API Client:** Axios with interceptors for token management

External Integrations

- **Meta Graph API:** v21.0 (Instagram + Facebook Messaging)
- **OpenAI API:** GPT-4 Mini for response generation
- **Naija LLM:** Custom Nigerian context-aware language model

---

Complete Roadmap Implementation

Phase 1: Setup and Architecture ✅

Goal: Establish foundational infrastructure and empty deployment.

Deliverables

- Repository structure with separate frontend/backend folders
- Environment variable configuration for Meta, OpenAI, and database
- Empty backend deployed to AWS EC2 or Render
- Frontend scaffold deployed to Vercel/Netlify
- Database schema designed and migrations created

Landmark: Skeleton app online with frontend + API connected via health check endpoint.

Phase 2: Authentication and Meta Connection ✅

Goal: Enable user registration and Meta account linking.

Deliverables

- JWT-based authentication system with signup/login endpoints
- Password hashing with bcrypt (10 rounds)
- Meta OAuth 2.0 flow implementation:
  - Authorization redirect to Facebook OAuth dialog
  - Callback handler storing access tokens
  - Token refresh logic for long-lived tokens
- Frontend pages:
  - Login/Signup forms with validation
  - "Connect Instagram and Facebook" button triggering OAuth flow
  - Success/error handling for connection status

Required Meta Permissions

- `pages_messaging` - Facebook page messaging
- `instagram_manage_messages` - Instagram DM access
- `instagram_basic` - Basic Instagram profile info
- `pages_read_engagement` - Read page engagement data
- `pages_manage_metadata` - Manage page settings
- `pages_show_list` - List user's pages
- `public_profile` - Basic user profile info

Landmark: User can sign up, log in, and successfully connect Meta accounts with stored tokens.

Phase 3: Messaging Webhooks & AI Replies

Goal: Core prototype—incoming messages trigger AI-generated replies.

Deliverables

- Meta Webhooks endpoint at `/api/meta/webhook`
- Webhook verification with verify token matching
- Incoming message parsing and storage:
  - Extract sender ID, message text, timestamp
  - Store in `messages` table with `direction='in'`
- Customer profile management:
  - Auto-create entries in `clients` table
  - Link messages to client profiles

- AI reply generation route (`/api/ai/reply`):
  - Fetch conversation history
  - Call OpenAI GPT-4 Mini with context and tone settings
  - Integrate Naija LLM for Nigerian context-aware responses
  - Store confidence scores for quality monitoring
- Reply delivery via `/api/meta/sendMessage`
- Dashboard UI showing:
  - Real-time message feed (incoming + replied)
  - Client profiles with message history
  - Reply status indicators

Landmark: Working prototype—messages arrive via webhook → AI generates reply → reply sent back to customer via Meta API.

Current Status: Phase 3 in progress.

Phase 4: Dashboard & Analytics

Goal: Usable interface for monitoring and controlling AI behavior.

Deliverables

- Enhanced dashboard with:
  - Message list with sorting and pagination
  - Filters: unreplied/replied/all messages
  - Search by client name or message content
  - Conversation threads grouped by client
- Settings page:
  - AI tone configuration (friendly, professional, casual, Nigerian informal)
  - Auto-reply toggle (enable/disable automation)
  - Confidence threshold for manual review
  - Business hours configuration
- Analytics panel:
  - Total messages received/sent (daily, weekly, monthly)
  - Reply rate percentage
  - Average response time
  - Top conversation topics (keyword extraction)
- Optional: Google Analytics integration for usage tracking

Landmark: Usable interface to monitor conversations and control AI reply behavior.

Phase 5: Polish & Testing

Goal: Transform prototype into stable MVP ready for beta testing.

Deliverables

- Comprehensive testing:
  - Test all Meta permissions in production mode
  - Verify webhook signature validation
  - Test token refresh for expired access tokens
- Mock data system for demo mode:
  - Sample conversations
  - Simulated message flow
  - Demo accounts for presentations
- Improved error handling:
  - Graceful degradation when APIs fail
  - User-friendly error messages
  - Retry logic for transient failures
- Queue system implementation (BullMQ):
  - Offload reply generation to worker processes
  - Handle high message volumes without blocking
  - Retry failed jobs with exponential backoff
- UI/UX improvements:
  - Loading states for all async operations
  - Toast notifications for success/error
  - Skeleton loaders for data fetching
  - Mobile-responsive design refinements

Landmark: Stable MVP ready for limited beta with real users.

Phase 6: Deployment & Demo

Goal: Publicly accessible working prototype with documentation.

Deliverables

- Production deployment:
  - Backend: Render or AWS EC2 with PM2 process manager
  - Database: Managed PostgreSQL (Render, AWS RDS, Supabase)
  - Frontend: Vercel or Netlify with environment variables
- SSL certificates and custom domains
- Demo preparation:
  - Scripted demo flow: signup → connect Meta → AI auto-reply
  - Video walkthrough (3-5 minutes)
  - Screenshot documentation
- Complete README with:
  - Setup instructions (local + production)
  - Environment variable reference
  - API documentation
  - Troubleshooting guide

Landmark: Hosted prototype accessible at custom domain, ready for demos and real Meta page testing.

Database Schema

Core Convergo Tables

```
-- User authentication and accounts
CREATE TABLE users (
id BIGSERIAL PRIMARY KEY,
email VARCHAR(255) UNIQUE NOT NULL,
password_hash VARCHAR(255) NOT NULL,
created_at TIMESTAMPTZ DEFAULT now(),
updated_at TIMESTAMPTZ DEFAULT now()
);

-- Meta account connections (Facebook pages + Instagram accounts)
CREATE TABLE meta_accounts (
id BIGSERIAL PRIMARY KEY,
user_id BIGINT NOT NULL REFERENCES users(id) ON DELETE
CASCADE,
access_token TEXT NOT NULL,
```

```
token_type VARCHAR(50) DEFAULT 'Bearer',
expires_in INTEGER,
page_id VARCHAR(255),
page_name VARCHAR(255),
instagram_account_id VARCHAR(255),
instagram_username VARCHAR(255),
created_at TIMESTAMPTZ DEFAULT now(),
updated_at TIMESTAMPTZ DEFAULT now(),
UNIQUE(user_id, page_id)
);

-- Customer/client profiles (message senders)
CREATE TABLE clients (
id BIGSERIAL PRIMARY KEY,
user_id BIGINT NOT NULL REFERENCES users(id) ON DELETE
CASCADE,
meta_psid VARCHAR(255) NOT NULL,
name VARCHAR(255),
email VARCHAR(255),
phone VARCHAR(50),
platform VARCHAR(20) NOT NULL CHECK (platform IN ('instagram',
'facebook')),
meta JSONB DEFAULT '{}',
created_at TIMESTAMPTZ DEFAULT now(),
updated_at TIMESTAMPTZ DEFAULT now(),
UNIQUE(user_id, meta_psid)
);

-- Conversation messages (incoming and outgoing)
CREATE TABLE messages (
id BIGSERIAL PRIMARY KEY,
user_id BIGINT NOT NULL REFERENCES users(id) ON DELETE
CASCADE,
client_id BIGINT NOT NULL REFERENCES clients(id) ON DELETE
CASCADE,
direction VARCHAR(10) NOT NULL CHECK (direction IN ('in', 'out')),
text TEXT NOT NULL,
raw_payload JSONB,
ai_model VARCHAR(50),
confidence NUMERIC(5,2),
```

```
replied BOOLEAN DEFAULT false,
created_at TIMESTAMPTZ DEFAULT now(),
INDEX idx_messages_client (client_id),
INDEX idx_messages_created (created_at DESC),
INDEX idx_messages_replied (replied)
);

-- User settings for AI behavior
CREATE TABLE settings (
user_id BIGINT PRIMARY KEY REFERENCES users(id) ON DELETE
CASCADE,
tone VARCHAR(100) DEFAULT 'friendly and professional',
auto_reply BOOLEAN DEFAULT true,
confidence_threshold NUMERIC(5,2) DEFAULT 0.70,
business_hours_start TIME DEFAULT '09:00',
business_hours_end TIME DEFAULT '18:00',
timezone VARCHAR(50) DEFAULT 'UTC',
updated_at TIMESTAMPTZ DEFAULT now()
);
```

Salebase Extension Tables

```
-- Sales leads (converted from message clients)
CREATE TABLE leads (
id BIGSERIAL PRIMARY KEY,
user_id BIGINT NOT NULL REFERENCES users(id) ON DELETE
CASCADE,
client_id BIGINT REFERENCES clients(id) ON DELETE SET NULL,
source VARCHAR(100) NOT NULL,
name VARCHAR(255),
email VARCHAR(255),
phone VARCHAR(50),
company VARCHAR(255),
status VARCHAR(20) DEFAULT 'new' CHECK (status IN ('new',
'qualified', 'contacted', 'lost', 'won')),
score INTEGER CHECK (score >= 0 AND score <= 100),
meta JSONB DEFAULT '{}',
created_at TIMESTAMPTZ DEFAULT now(),
updated_at TIMESTAMPTZ DEFAULT now(),
INDEX idx_leads_status (status),
```

```
INDEX idx_leads_score (score DESC)
);

-- Sales pipelines
CREATE TABLE pipelines (
id BIGSERIAL PRIMARY KEY,
user_id BIGINT NOT NULL REFERENCES users(id) ON DELETE
CASCADE,
name VARCHAR(255) NOT NULL,
description TEXT,
created_at TIMESTAMPTZ DEFAULT now()
);

-- Pipeline stages
CREATE TABLE stages (
id BIGSERIAL PRIMARY KEY,
pipeline_id BIGINT NOT NULL REFERENCES pipelines(id) ON DELETE
CASCADE,
name VARCHAR(255) NOT NULL,
position INTEGER NOT NULL,
created_at TIMESTAMPTZ DEFAULT now(),
UNIQUE(pipeline_id, position)
);

-- Sales deals
CREATE TABLE deals (
id BIGSERIAL PRIMARY KEY,
user_id BIGINT NOT NULL REFERENCES users(id) ON DELETE
CASCADE,
pipeline_id BIGINT NOT NULL REFERENCES pipelines(id) ON DELETE
CASCADE,
stage_id BIGINT NOT NULL REFERENCES stages(id) ON DELETE
RESTRICT,
lead_id BIGINT NOT NULL REFERENCES leads(id) ON DELETE
CASCADE,
title VARCHAR(255) NOT NULL,
amount NUMERIC(12,2),
currency VARCHAR(10) DEFAULT 'NGN',
expected_close_date DATE,
created_at TIMESTAMPTZ DEFAULT now(),
```

updated_at TIMESTAMPTZ DEFAULT now()
);

Environment Configuration

Required Environment Variables

Server Configuration

PORT=4000
NODE_ENV=production # development | staging | production

Database

DATABASE_URL=postgres://username:password@host:5432/convergo

Alternative format for pg-promise

DB_HOST=localhost
DB_PORT=5432
DB_NAME=convergo
DB_USER=postgres
DB_PASSWORD=secure_password

JWT Authentication

JWT_SECRET=your-super-secret-key-min-32-chars
JWT_EXPIRY=7d # Token expiration time

Meta Graph API (from developers.facebook.com)

META_APP_ID=123456789012345
META_APP_SECRET=abcdef1234567890abcdef1234567890
META_VERIFY_TOKEN=my_custom_verify_token_12345
META_REDIRECT_URI=https://api.yourdomain.com/api/meta/oauth/callback
META_GRAPH_VERSION=v21.0

OpenAI API

OPENAI_API_KEY=sk-proj-xxxxxxxxxxxxxxxxxxxxx
OPENAI_MODEL=gpt-4o-mini
OPENAI_MAX_TOKENS=500
OPENAI_TEMPERATURE=0.4

Naija LLM (custom model endpoint)

NAIJA_API_URL=https://naija-llm.yourdomain.com/api/chat
NAIJA_API_KEY=your_naija_api_key

Redis (for BullMQ queue system)

REDIS_URL=redis://localhost:6379
REDIS_HOST=localhost
REDIS_PORT=6379
REDIS_PASSWORD= # Leave empty if no password

Frontend URLs (for CORS and redirects)

FRONTEND_URL=https://app.yourdomain.com
ALLOWED_ORIGINS=https://app.yourdomain.com,
http://localhost:5173

Logging

LOG_LEVEL=info # error | warn | info | debug

Frontend Environment Variables

API Configuration

VITE_API_URL=https://api.yourdomain.com/api
VITE_WS_URL=wss://api.yourdomain.com # WebSocket for real-time updates

Meta OAuth (for frontend redirect construction)

VITE_META_APP_ID=123456789012345

Feature Flags

VITE_ENABLE_ANALYTICS=true
VITE_ENABLE_MOCK_DATA=false

Repository Structure

```
convergo/
├── api/ # Backend Node.js application
│ ├── src/
│ │ ├── config/
│ │ │ ├── database.ts # PostgreSQL connection
│ │ │ └── env.ts # Environment variable validation
│ │ ├── middleware/
│ │ │ ├── auth.ts # JWT verification middleware
│ │ │ ├── errorHandler.ts # Global error handler
│ │ │ └── metaVerify.ts # Meta webhook verification
│ │ ├── routes/
│ │ │ ├── auth.ts # Signup/login endpoints
│ │ │ ├── metaOauth.ts # Meta OAuth flow
│ │ │ ├── metaWebhook.ts # Incoming message webhooks
│ │ │ ├── messages.ts # Message CRUD operations
│ │ │ ├── clients.ts # Client management
│ │ │ ├── settings.ts # User settings
│ │ │ └── leads.ts # Salebase lead endpoints
│ │ ├── services/
│ │ │ ├── metaApi.ts # Meta Graph API client
│ │ │ ├── aiReply.ts # AI response generation
│ │ │ ├── queue.ts # BullMQ queue setup
│ │ │ ├── userService.ts # User business logic
│ │ │ └── messageService.ts
│ │ ├── jobs/
│ │ │ └── replyWorker.ts # Background job worker
│ │ ├── utils/
│ │ │ ├── logger.ts # Winston logger
│ │ │ └── validators.ts # Input validation schemas
│ │ └── index.ts # Express app entry point
│ ├── migrations/ # Database migrations
│ ├── tests/ # API tests
│ ├── .env.example
│ ├── Dockerfile
│ ├── package.json
│ └── tsconfig.json
│
```

```
├── web/ # Frontend React application
│   ├── src/
│   │   ├── api/
│   │   │   └── client.ts # Axios instance with interceptors
│   │   ├── pages/
│   │   │   ├── Login.tsx
│   │   │   ├── Signup.tsx
│   │   │   ├── Dashboard.tsx
│   │   │   ├── Messages.tsx
│   │   │   ├── Settings.tsx
│   │   │   ├── Leads.tsx # Salebase leads page
│   │   │   └── Pipeline.tsx # Salebase pipeline view
│   │   ├── components/
│   │   │   ├── MessageThread.tsx
│   │   │   ├── ClientCard.tsx
│   │   │   └── StatsCard.tsx
│   │   ├── hooks/
│   │   │   └── useAuth.ts # Authentication hook
│   │   ├── store/
│   │   │   └── authStore.ts # Zustand auth state
│   │   ├── App.tsx
│   │   └── main.tsx
│   ├── public/
│   ├── index.html
│   ├── vite.config.ts
│   ├── tailwind.config.js
│   └── package.json
│
├── docs/ # Documentation
│   ├── API.md # API endpoint documentation
│   ├── DEPLOYMENT.md # Deployment guide
│   └── ARCHITECTURE.md # System architecture details
│
├── .github/
│   └── workflows/
│   ├── backend-ci.yml # Backend CI/CD
│   └── frontend-ci.yml # Frontend CI/CD
│
└── README.md # Project overview
```

Core Code Examples

Backend: Express Server Bootstrap

```
// api/src/index.ts
import express from 'express';
import cors from 'cors';
import helmet from 'helmet';
import 'dotenv/config';
import { authRouter } from './routes/auth';
import { metaOauthRouter } from './routes/metaOauth';
import { metaWebhookRouter } from './routes/metaWebhook';
import { messagesRouter } from './routes/messages';
import { settingsRouter } from './routes/settings';
import { leadsRouter } from './routes/leads';
import { authMiddleware } from './middleware/auth';
import { errorHandler } from './middleware/errorHandler';
import { logger } from './utils/logger';

const app = express();

// Security middleware
app.use(helmet());
app.use(cors({
origin: process.env.ALLOWED_ORIGINS?.split(',') || '*',
credentials: true
}));
app.use(express.json());

// Health check
app.get('/health', (_req, res) => {
res.json({ status: 'ok', timestamp: new Date().toISOString() });
});

// Public routes
app.use('/api/auth', authRouter);
app.use('/api/meta/oauth', metaOauthRouter);
app.use('/api/meta/webhook', metaWebhookRouter);

// Protected routes
app.use('/api/messages', authMiddleware, messagesRouter);
```

app.use('/api/settings', authMiddleware, settingsRouter);
app.use('/api/leads', authMiddleware, leadsRouter);

// Error handling
app.use(errorHandler);

const port = process.env.PORT || 4000;
app.listen(port, () => {
logger.info(Convergo API running on port ${port});
});

Backend: JWT Authentication

// api/src/routes/auth.ts
import { Router } from 'express';
import bcrypt from 'bcrypt';
import jwt from 'jsonwebtoken';
import { db } from '../config/database';
import { logger } from '../utils/logger';

export const authRouter = Router();

authRouter.post('/signup', async (req, res, next) => {
try {
const { email, password } = req.body;

```
// Validate input
if (!email || !password) {
  return res.status(400).json({ error: 'Email and password required' });
}

// Check if user exists
const existing = await db.oneOrNone(
  'SELECT id FROM users WHERE email = $1',
  [email]
);

if (existing) {
  return res.status(409).json({ error: 'Email already registered' });
}
```

```
    // Hash password
    const passwordHash = await bcrypt.hash(password, 10);

    // Create user
    const user = await db.one(
      'INSERT INTO users (email, password_hash) VALUES ($1, $2) RETURNING id
      [email, passwordHash]
    );

    // Generate JWT
    const token = jwt.sign(
      { sub: user.id, email: user.email },
      process.env.JWT_SECRET as string,
      { expiresIn: process.env.JWT_EXPIRY || '7d' }
    );

    logger.info(`New user registered: ${user.email}`);

    res.status(201).json({
      token,
      user: { id: user.id, email: user.email }
    });
```

} catch (error) {
next(error);
}
});

authRouter.post('/login', async (req, res, next) => {
try {
const { email, password } = req.body;

```
    // Find user
    const user = await db.oneOrNone(
      'SELECT id, email, password_hash FROM users WHERE email = $1',
      [email]
    );
```

```
  if (!user) {
    return res.status(401).json({ error: 'Invalid credentials' });
  }

  // Verify password
  const valid = await bcrypt.compare(password, user.password_hash);
  if (!valid) {
    return res.status(401).json({ error: 'Invalid credentials' });
  }

  // Generate JWT
  const token = jwt.sign(
    { sub: user.id, email: user.email },
    process.env.JWT_SECRET as string,
    { expiresIn: process.env.JWT_EXPIRY || '7d' }
  );

  res.json({
    token,
    user: { id: user.id, email: user.email }
  });
```

```
} catch (error) {
next(error);
}
});
```

Backend: Meta OAuth Flow

```
// api/src/routes/metaOauth.ts
import { Router } from 'express';
import axios from 'axios';
import { db } from '../config/database';
import { authMiddleware } from '../middleware/auth';

export const metaOauthRouter = Router();
```

```
// Redirect user to Facebook OAuth dialog
metaOauthRouter.get('/authorize', authMiddleware, (req, res) => {
const baseUrl = 'https://www.facebook.com/v21.0/dialog/oauth';
const params = new URLSearchParams({
client_id: process.env.META_APP_ID!,
redirect_uri: process.env.META_REDIRECT_URI!,
scope: [
'pages_messaging',
'pages_read_engagement',
'pages_manage_metadata',
'pages_show_list',
'instagram_manage_messages',
'instagram_basic',
'public_profile'
].join(','),
state: String(req.user.id),
response_type: 'code'
});

res.redirect(${baseUrl}?${params.toString()});
});

// Handle OAuth callback
metaOauthRouter.get('/callback', async (req, res) => {
try {
const { code, state } = req.query;
const userId = Number(state);
```

```
if (!code) {
  return res.redirect(`${process.env.FRONTEND_URL}/error?msg=oauth_failed
}

// Exchange code for access token
const tokenResponse = await axios.get(
  'https://graph.facebook.com/v21.0/oauth/access_token',
  {
    params: {
      client_id: process.env.META_APP_ID,
      client_secret: process.env.META_APP_SECRET,
```

```
      redirect_uri: process.env.META_REDIRECT_URI,
      code
    }
  }
);

const { access_token, expires_in } = tokenResponse.data;

// Get user's pages
const pagesResponse = await axios.get(
  'https://graph.facebook.com/v21.0/me/accounts',
  {
    params: { access_token }
  }
);

// Store page tokens
for (const page of pagesResponse.data.data) {
  await db.none(
    `INSERT INTO meta_accounts (user_id, access_token, expires_in, page_id, p
     VALUES ($1, $2, $3, $4, $5)
     ON CONFLICT (user_id, page_id)
     DO UPDATE SET access_token = $2, expires_in = $3, updated_at = now()`,
    [userId, page.access_token, expires_in, page.id, page.name]
  );
}

res.redirect(`${process.env.FRONTEND_URL}/connected`);
```

} catch (error) {
console.error('OAuth error:', error);
res.redirect(${process.env.FRONTEND_URL}/error?
msg=oauth_error);
}
});

Backend: Meta Webhook Handler

```
// api/src/routes/metaWebhook.ts
import { Router } from 'express';
import { handleIncomingMessage } from '../services/messageService';
import { logger } from '../utils/logger';

export const metaWebhookRouter = Router();

// Webhook verification (GET request from Meta)
metaWebhookRouter.get('/', (req, res) => {
const mode = req.query['hub.mode'];
const token = req.query['hub.verify_token'];
const challenge = req.query['hub.challenge'];

if (mode === 'subscribe' && token ===
process.env.META_VERIFY_TOKEN) {
logger.info('Webhook verified successfully');
return res.status(200).send(challenge);
}

logger.warn('Webhook verification failed');
return res.sendStatus(403);
});

// Incoming message webhook (POST request from Meta)
metaWebhookRouter.post('/', async (req, res) => {
const body = req.body;

// Always respond quickly to Meta
res.sendStatus(200);

try {
if (body.object === 'page') {
for (const entry of body.entry) {
// Handle messaging events
if (entry.messaging) {
for (const event of entry.messaging) {
await handleIncomingMessage(event);
}
}
```

```
}
}
} catch (error) {
logger.error('Webhook processing error:', error);
}
});
```

Backend: AI Reply Generation

```
// api/src/services/aiReply.ts
import axios from 'axios';
import { db } from '../config/database';
import { logger } from '../utils/logger';

interface ReplyContext {
userId: number;
clientId: number;
messageText: string;
conversationHistory?: Array<{ role: string; content: string }>;
}

export async function generateAiReply(context: ReplyContext) {
const { userId, clientId, messageText, conversationHistory = [] } =
context;

// Get user settings
const settings = await db.oneOrNone(
'SELECT tone, auto_reply FROM settings WHERE user_id = $1',
[userId]
);

const tone = settings?.tone || 'friendly and professional';
const autoReply = settings?.auto_reply !== false;

if (!autoReply) {
logger.info(Auto-reply disabled for user ${userId});
return null;
}

// Build conversation context
const messages = [
```

{
role: 'system',
content: You are a Nigerian business assistant. Reply in a natural, ${tone} tone. Keep responses concise (under 150 words). Use Nigerian English when appropriate. Help customers with their inquiries professionally.
},
...conversationHistory,
{
role: 'user',
content: messageText
}
];

try {
// Call OpenAI API
const response = await axios.post(
'https://api.openai.com/v1/chat/completions',
{
model: process.env.OPENAI_MODEL || 'gpt-4o-mini',
messages,
temperature: Number(process.env.OPENAI_TEMPERATURE) || 0.4,
max_tokens: Number(process.env.OPENAI_MAX_TOKENS) || 500
},
{
headers: {
'Authorization': Bearer ${process.env.OPENAI_API_KEY},
'Content-Type': 'application/json'
}
}
);

```
const replyText = response.data.choices[0].message.content.trim();
const confidence = response.data.choices[0].finish_reason === 'stop' ? 0.85 : 0.

logger.info(`AI reply generated for client ${clientId}`);

return {
  text: replyText,
```

```
    model: process.env.OPENAI_MODEL || 'gpt-4o-mini',
    confidence
  };
```

} catch (error) {
logger.error('AI reply generation failed:', error);
throw error;
}
}

Backend: Message Service

// api/src/services/messageService.ts
import { db } from '../config/database';
import { generateAiReply } from './aiReply';
import { sendMetaMessage } from './metaApi';
import { logger } from '../utils/logger';

export async function handleIncomingMessage(event: any) {
const senderId = event.sender.id;
const messageText = event.message?.text;

if (!messageText) {
logger.debug('No text in message, skipping');
return;
}

try {
// Find or create client
let client = await db.oneOrNone(
'SELECT id, user_id FROM clients WHERE meta_psid = $1',
[senderId]
);

```
  if (!client) {
    // Create new client (assign to page owner)
    const pageId = event.recipient.id;
    const metaAccount = await db.oneOrNone(
      'SELECT user_id FROM meta_accounts WHERE page_id = $1',
      [pageId]
```

```
  );

  if (!metaAccount) {
    logger.warn(`No meta account found for page ${pageId}`);
    return;
  }

  client = await db.one(
    `INSERT INTO clients (user_id, meta_psid, platform)
     VALUES ($1, $2, $3)
     RETURNING id, user_id`,
    [metaAccount.user_id, senderId, 'facebook']
  );
}

// Store incoming message
await db.none(
  `INSERT INTO messages (user_id, client_id, direction, text, raw_payload)
   VALUES ($1, $2, 'in', $3, $4)`,
  [client.user_id, client.id, messageText, event]
);

// Generate AI reply
const reply = await generateAiReply({
  userId: client.user_id,
  clientId: client.id,
  messageText
});

if (!reply) return;

// Store outgoing message
await db.none(
  `INSERT INTO messages (user_id, client_id, direction, text, ai_model, confide
   VALUES ($1, $2, 'out', $3, $4, $5)`,
  [client.user_id, client.id, reply.text, reply.model, reply.confidence]
);
```

```
  // Send reply via Meta
  await sendMetaMessage({
    recipientId: senderId,
    text: reply.text,
    userId: client.user_id
  });

  // Mark original message as replied
  await db.none(
    `UPDATE messages SET replied = true
     WHERE client_id = $1 AND direction = 'in' AND replied = false`,
    [client.id]
  );

  logger.info(`Reply sent to client ${client.id}`);
```

```
} catch (error) {
logger.error('Message handling error:', error);
throw error;
}
}
```

Backend: Meta API Client

```
// api/src/services/metaApi.ts
import axios from 'axios';
import { db } from '../config/database';
import { logger } from '../utils/logger';

interface SendMessageParams {
recipientId: string;
text: string;
userId: number;
}

export async function sendMetaMessage(params:
SendMessageParams) {
const { recipientId, text, userId } = params;
```

```
// Get user's access token
const account = await db.one(
'SELECT access_token FROM meta_accounts WHERE user_id = $1
LIMIT 1',
[userId]
);

try {
await axios.post(
https://graph.facebook.com/${process.env.META_GRAPH_VERSION}/
me/messages,
{
recipient: { id: recipientId },
message: { text }
},
{
params: { access_token: account.access_token }
}
);

  logger.info(`Message sent to ${recipientId}`);

} catch (error) {
logger.error('Meta API send error:', error);
throw error;
}
}
```

Frontend: Axios Client with Auth

```
// web/src/api/client.ts
import axios from 'axios';

const api = axios.create({
baseURL: import.meta.env.VITE_API_URL || 'http://localhost:4000/api
',
headers: {
'Content-Type': 'application/json'
}
});
```

```
// Request interceptor - add auth token
api.interceptors.request.use(
(config) => {
const token = localStorage.getItem('convergo_token');
if (token) {
config.headers.Authorization = Bearer ${token};
}
return config;
},
(error) => Promise.reject(error)
);

// Response interceptor - handle auth errors
api.interceptors.response.use(
(response) => response,
(error) => {
if (error.response?.status === 401) {
localStorage.removeItem('convergo_token');
window.location.href = '/login';
}
return Promise.reject(error);
}
);

export default api;
```

Frontend: Dashboard Component

```
// web/src/pages/Dashboard.tsx
import { useEffect, useState } from 'react';
import api from '../api/client';

interface Message {
id: number;
client_id: number;
client_name: string;
direction: 'in' | 'out';
text: string;
created_at: string;
replied: boolean;
}
```

```
interface Stats {
total: number;
unreplied: number;
replyRate: number;
}

export default function Dashboard() {
const [messages, setMessages] = useState<Message[]>([]);
const [stats, setStats] = useState<Stats | null>(null);
const [filter, setFilter] = useState<'all' | 'unreplied' | 'replied'>('all');
const [loading, setLoading] = useState(true);

useEffect(() => {
fetchStats();
}, []);

useEffect(() => {
fetchMessages();
}, [filter]);

const fetchStats = async () => {
try {
const response = await api.get('/messages/stats');
setStats(response.data);
} catch (error) {
console.error('Failed to fetch stats:', error);
}
};

const fetchMessages = async () => {
setLoading(true);
try {
const response = await api.get('/messages', {
params: { filter }
});
setMessages(response.data);
} catch (error) {
console.error('Failed to fetch messages:', error);
} finally {
setLoading(false);
```

}
};

return (

Dashboard

```
{/* Stats Cards */}
{stats && (
  <div className="grid grid-cols-3 gap-4 mb-6">
    <div className="bg-white p-4 rounded-lg shadow">
      <div className="text-gray-600 text-sm">Total Messages</div>
      <div className="text-3xl font-bold">{stats.total}</div>
    </div>
    <div className="bg-white p-4 rounded-lg shadow">
      <div className="text-gray-600 text-sm">Unreplied</div>
      <div className="text-3xl font-bold text-red-600">{stats.unreplied}</div>
    </div>
    <div className="bg-white p-4 rounded-lg shadow">
      <div className="text-gray-600 text-sm">Reply Rate</div>
      <div className="text-3xl font-bold text-green-600">{stats.replyRate}%</d
    </div>
  </div>
)}

{/* Filter Buttons */}
<div className="mb-4 space-x-2">
  <button
    onClick={() => setFilter('all')}
    className={`px-4 py-2 rounded ${
      filter === 'all' ? 'bg-blue-600 text-white' : 'bg-gray-200'
    }`}
  >
    All
  </button>
  <button
    onClick={() => setFilter('unreplied')}
```

```
    className={`px-4 py-2 rounded ${
      filter === 'unreplied' ? 'bg-red-600 text-white' : 'bg-gray-200'
    }`}
  >
    Unreplied
  </button>
  <button
    onClick={() => setFilter('replied')}
    className={`px-4 py-2 rounded ${
      filter === 'replied' ? 'bg-green-600 text-white' : 'bg-gray-200'
    }`}
  >
    Replied
  </button>
</div>

{/* Messages Table */}
{loading ? (
  <div className="text-center py-8">Loading messages...</div>
) : (
  <div className="bg-white rounded-lg shadow overflow-hidden">
    <table className="w-full">
      <thead className="bg-gray-100">
        <tr>
          <th className="px-4 py-3 text-left text-sm font-semibold">Client</th>
          <th className="px-4 py-3 text-left text-sm font-semibold">Direction</
          <th className="px-4 py-3 text-left text-sm font-semibold">Message</th
          <th className="px-4 py-3 text-left text-sm font-semibold">Time</th>
        </tr>
      </thead>
      <tbody className="divide-y">
        {messages.map((msg) => (
          <tr key={msg.id} className="hover:bg-gray-50">
            <td className="px-4 py-3">{msg.client_name || `Client ${msg.client_
            <td className="px-4 py-3">
              <span
                className={`px-2 py-1 rounded text-xs ${
                  msg.direction === 'in'
```

```
                    ? 'bg-blue-100 text-blue-800'
                    : 'bg-green-100 text-green-800'
                }`}
              >
                {msg.direction === 'in' ? 'Incoming' : 'Outgoing'}
              </span>
            </td>
            <td className="px-4 py-3 text-sm">{msg.text}</td>
            <td className="px-4 py-3 text-sm text-gray-600">
              {new Date(msg.created_at).toLocaleString()}
            </td>
          </tr>
        ))}
      </tbody>
    </table>
  </div>
 )}
</div>
```

);
}

Frontend: Settings Component

```
// web/src/pages/Settings.tsx
import { useEffect, useState } from 'react';
import api from '../api/client';

export default function Settings() {
const [tone, setTone] = useState('');
const [autoReply, setAutoReply] = useState(true);
const [saving, setSaving] = useState(false);

useEffect(() => {
loadSettings();
}, []);

const loadSettings = async () => {
try {
const response = await api.get('/settings/me');
```

setTone(response.data.tone || '');
setAutoReply(response.data.auto_reply);
} catch (error) {
console.error('Failed to load settings:', error);
}
};

const saveSettings = async () => {
setSaving(true);
try {
await api.post('/settings/me', {
tone,
auto_reply: autoReply
});
alert('Settings saved successfully!');
} catch (error) {
console.error('Failed to save settings:', error);
alert('Failed to save settings');
} finally {
setSaving(false);
}
};

return (

AI Settings

```
<div className="bg-white p-6 rounded-lg shadow space-y-6">
  <div>
    <label className="block text-sm font-semibold mb-2">
      AI Response Tone
    </label>
    <textarea
      value={tone}
      onChange={(e) => setTone(e.target.value)}
      placeholder="e.g., friendly and professional, casual Nigerian style"
      className="w-full border rounded p-3 h-24"
    />
```

```
      <p className="text-sm text-gray-600 mt-1">
        Describe how you want the AI to communicate with customers
      </p>
    </div>

    <div className="flex items-center gap-3">
      <input
        type="checkbox"
        id="autoReply"
        checked={autoReply}
        onChange={(e) => setAutoReply(e.target.checked)}
        className="w-5 h-5"
      />
      <label htmlFor="autoReply" className="text-sm font-semibold">
        Enable automatic AI replies
      </label>
    </div>

    <button
      onClick={saveSettings}
      disabled={saving}
      className="px-6 py-2 bg-blue-600 text-white rounded hover:bg-blue-700
    >
      {saving ? 'Saving...' : 'Save Settings'}
    </button>
  </div>
</div>
```

);
}

Salebase Extension

Lead Qualification Service

```
// api/src/services/leadAi.ts
import axios from 'axios';
import { db } from '../config/database';
import { logger } from '../utils/logger';

interface Lead {
id: number;
user_id: number;
name?: string;
email?: string;
phone?: string;
company?: string;
source: string;
}

export async function qualifyLead(lead: Lead) {
const prompt = `You are a B2B sales assistant. Analyze this lead and
provide:

1. Interest score (0-100)
2. Label: cold, warm, or hot
3. Next action suggestion (brief)

Lead details:
${JSON.stringify(lead, null, 2)}

Respond with JSON only: { "score": number, "label": string,
"next_action": string }`;

try {
const response = await axios.post(
'https://api.openai.com/v1/chat/completions',
{
model: 'gpt-4o-mini',
messages: [{ role: 'user', content: prompt }],
temperature: 0.2
},
{
headers: {
```

'Authorization': Bearer ${process.env.OPENAI_API_KEY},
'Content-Type': 'application/json'
}
}
);

```
const content = response.data.choices[0].message.content;
let result;

try {
  result = JSON.parse(content);
} catch {
  result = {
    score: null,
    label: 'unknown',
    next_action: 'Manual review required'
  };
}

// Update lead with AI insights
await db.none(
  `UPDATE leads
   SET score = $1,
       meta = jsonb_set(
         jsonb_set(
           COALESCE(meta, '{}'::jsonb),
           '{label}', to_jsonb($2::text), true
         ),
         '{next_action}', to_jsonb($3::text), true
       ),
       updated_at = now()
   WHERE id = $4`,
  [result.score, result.label, result.next_action, lead.id]
);

logger.info(`Lead ${lead.id} qualified: score=${result.score}, label=${result.lab
```

```
return result;
```

} catch (error) {
logger.error('Lead qualification error:', error);
throw error;
}
}

Follow-up Generation Service

// api/src/services/followupAi.ts
import axios from 'axios';

interface FollowupParams {
lead: any;
channel: 'email' | 'whatsapp';
context?: string;
}

export async function generateFollowup(params: FollowupParams) {
const { lead, channel, context = 'Standard follow-up' } = params;

const prompt = `You are a professional sales representative.
Write a ${channel} follow-up message to this lead.

Goal: Schedule a discovery call or demo
Tone: Friendly but direct
Length: Maximum 120 words
Context: ${context}

Lead information:
${JSON.stringify(lead, null, 2)}

Generate the message:`;

const response = await axios.post(
'https://api.openai.com/v1/chat/completions',
{
model: 'gpt-4o-mini',
messages: [{ role: 'user', content: prompt }],

```
temperature: 0.5
},
{
headers: {
'Authorization': Bearer ${process.env.OPENAI_API_KEY},
'Content-Type': 'application/json'
}
}
);

return response.data.choices[0].message.content.trim();
}
```

Deployment Guide

Backend Deployment (Render)

1. **Create account at render.com**
2. **Connect GitHub repository**
3. **Create new Web Service:**
   - Build Command: `npm install`
   - Start Command: `npm start`
   - Environment: Node
   - Region: Choose closest to your users
4. **Add environment variables** (all from `.env.example`)
5. **Deploy**
6. **Configure Meta webhook URL:** `https://your-app.onrender.com/api/meta/webhook`

Backend Deployment (AWS EC2)

Launch Ubuntu 22.04 instance

SSH into instance

ssh -i key.pem ubuntu@ec2-xx-xx-xx-xx.compute.amazonaws.com

Install Node.js 20

curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -
sudo apt install -y nodejs

Install PostgreSQL

sudo apt install -y postgresql postgresql-contrib

Install PM2 for process management

sudo npm install -g pm2

Clone repository

git clone https://github.com/yourusername/convergo.git
cd convergo/api
npm install

Set environment variables

nano .env # Add all required vars

Start with PM2

pm2 start src/index.js --name convergo-api
pm2 save
pm2 startup

Install Nginx as reverse proxy

sudo apt install -y nginx

Configure Nginx

sudo nano /etc/nginx/sites-available/convergo

Nginx configuration:

server {
listen 80;
server_name api.yourdomain.com;

```
location / {
    proxy_pass http://localhost:4000;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection 'upgrade';
    proxy_set_header Host $host;
    proxy_cache_bypass $http_upgrade;
}
```

}

Enable site

sudo ln -s /etc/nginx/sites-available/convergo /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl restart nginx

Install SSL with Let's Encrypt

sudo apt install -y certbot python3-certbot-nginx
sudo certbot --nginx -d api.yourdomain.com

Frontend Deployment (Vercel)

Install Vercel CLI

npm install -g vercel

Navigate to frontend folder

cd web

Deploy

vercel --prod

Set environment variables in Vercel dashboard:

VITE_API_URL=https://api.yourdomain.com/api

Database Setup (Render PostgreSQL)

1. **Create PostgreSQL instance** on Render
2. **Copy connection string**
3. **Run migrations:**

Connect to database

psql $DATABASE_URL

Run schema from documentation

\i migrations/001_initial_schema.sql

Meta App Configuration

App Creation

1. Go to **developers.facebook.com**
2. Create new app → **Business** type
3. Add products:
   - **Messenger**
   - **Webhooks**

Permissions

Request these permissions for app review:

- `pages_messaging` - Send/receive messages to Facebook pages
- `instagram_manage_messages` - Send/receive Instagram DMs
- `instagram_basic` - Access basic Instagram profile info
- `pages_read_engagement` - Read page engagement metrics
- `pages_manage_metadata` - Manage page settings

Webhook Setup

1. Go to **Webhooks** product
2. Click **Add Callback URL**
3. Enter:
   - Callback URL: `https://your-api.com/api/meta/webhook`
   - Verify Token: (match `META_VERIFY_TOKEN` in your `.env`)
4. Subscribe to fields:
   - `messages`

- messaging_postbacks
- messaging_optins

Testing

1. Add test user to app
2. Connect test Facebook page
3. Send message to page
4. Verify webhook receives event
5. Check AI reply is sent back

Troubleshooting

Common Issues

Issue	Cause	Solution
Webhook returns 403	Verify token mismatch	Check META_VERIFY_TOKEN matches exactly
No AI replies	OpenAI key invalid or auto-reply disabled	Verify OPENAI_API_KEY and check settings
Token expired error	Access token expired	Implement token refresh logic
Messages not storing	Database connection failed	Check DATABASE_URL and connection
CORS errors	Frontend origin not allowed	Add frontend URL to ALLOWED_ORIGINS

Debug Checklist

Check backend health

curl https://your-api.com/health

Test webhook verification

curl "https://your-api.com/api/meta/webhook?hub.mode=subscribe&hub.verify_token=YOUR_TOKEN&hub.challenge=test"

Check database connection

psql $DATABASE_URL -c "SELECT count(*) FROM users;"

View backend logs (PM2)

pm2 logs convergo-api

View backend logs (Render)

Check Logs tab in Render dashboard

Next Immediate Steps

Phase 3 Completion Checklist

- [ ] Deploy webhook endpoint to production
- [ ] Configure Meta app webhook subscription
- [ ] Test end-to-end flow: DM → webhook → AI → reply
- [ ] Build basic dashboard showing message history
- [ ] Verify messages stored in database correctly
- [ ] Test AI reply generation with different tones
- [ ] Monitor error logs for any failures

Phase 4 Preparation

- [ ] Design message filtering UI
- [ ] Plan settings page layout
- [ ] Define analytics metrics to track
- [ ] Set up monitoring (Sentry, LogRocket, etc.)

Demo Script

5-Minute Demo Flow

1. **Show landing page** (30s)
    - Explain value proposition
    - Click "Get Started"
2. **Sign up** (30s)
    - Enter email and password
    - Show successful registration
3. **Connect Meta account** (1m)
    - Click "Connect Instagram/Facebook"
    - OAuth flow with Facebook
    - Show successful connection confirmation
4. **Dashboard overview** (1m)
    - Tour of interface
    - Stats cards
    - Message filters
5. **Live demo** (2m)
    - Send DM to connected page (from phone)
    - Watch message appear in dashboard
    - Show AI-generated reply
    - Verify reply received on phone
6. **Settings** (30s)
    - Show tone configuration
    - Toggle auto-reply
    - Save settings

Key Talking Points

- **Multi-tenant:** Each business gets isolated data and settings
- **Real-time:** Messages processed instantly via webhooks
- **AI-powered:** Intelligent responses using GPT-4 + Nigerian context
- **Scalable:** Built on modern stack (Node, React, PostgreSQL)
- **Extensible:** Salebase layer adds sales automation

Future Roadmap (Post-MVP)

Q2 2026

- Multi-language support (Yoruba, Igbo, Hausa)
- WhatsApp Business API integration
- Conversation templates and saved replies
- Team collaboration features

Q3 2026

- Advanced analytics and reporting
- A/B testing for AI responses
- Integration marketplace (CRMs, helpdesks)
- Mobile apps (iOS + Android)

Q4 2026

- Voice message transcription
- Image recognition for product inquiries
- Automated appointment booking
- Payment collection integration

Contact & Support

Project Owner: Amadi Chibuzor
Location: Cremona, Lombardy, Italy
Documentation Version: 1.0
Last Updated: February 23, 2026

For technical questions or contributions, please refer to the GitHub repository README.

This documentation is living and should be updated as the project evolves. All code examples are representative and may need adjustment for your specific implementation.