

IMPLEMENTATION-TECHNICAL-GUIDE.md

REACTIVATEAI: COMPLETE TECHNICAL IMPLEMENTATION

PHASE 1: UNIFIED INBOX (Weeks 1-4)

Database Schema Additions

```
-- Track conversations across channels
CREATE TABLE unified_conversations (
  conversation_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  clinic_id UUID NOT NULL REFERENCES clinics(clinic_id),
  patient_id UUID NOT NULL REFERENCES patients(patient_id),

  -- Metadata
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  last_message_at TIMESTAMP,
  status VARCHAR(50) DEFAULT 'active', -- 'active', 'resolved', 'archived',
  'waiting_response'

  -- Assignment & ownership
  assigned_to UUID REFERENCES clinic_users(user_id),
  assigned_at TIMESTAMP,

  -- Clinical context
  appointment_id UUID REFERENCES appointments(appointment_id),
  reason VARCHAR(255), -- 'appointment_reminder', 'follow_up', 'booking_inquiry',
  etc.

  -- Tags for organization
  tags TEXT[] DEFAULT '{}', -- ['urgent', 'vip', 'post_op']

  INDEX idx_clinic_last_message (clinic_id, last_message_at DESC),
  INDEX idx_patient (patient_id),
  INDEX idx_status (clinic_id, status)
);

-- All messages unified here (SMS, WhatsApp, Email, etc.)
CREATE TABLE unified_messages (
  message_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  conversation_id UUID NOT NULL REFERENCES unified_conversations(conversation_id),

  -- Channel info
  channel VARCHAR(50) NOT NULL, -- 'sms', 'whatsapp', 'email', 'facebook',
  'instagram'
  external_message_id VARCHAR(255), -- Provider's ID (Twilio SID, Meta PSID, etc.)

  -- Direction & sender
  direction VARCHAR(20) NOT NULL, -- 'inbound', 'outbound'
```

```

from_number VARCHAR(50),
to_number VARCHAR(50),

-- Content
message_type VARCHAR(50), -- 'text', 'image', 'file', 'template',
'button_response'
message_text TEXT,
media_url VARCHAR(500), -- For images, video, files

-- Delivery tracking
sent_at TIMESTAMP NOT NULL,
delivered_at TIMESTAMP,
read_at TIMESTAMP,
failed_at TIMESTAMP,
failure_reason VARCHAR(255),

-- Engagement
clicked_at TIMESTAMP,
clicked_url VARCHAR(500),

-- Creator
created_by_user_id UUID REFERENCES clinic_users(user_id),

created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

INDEX idx_conversation (conversation_id, sent_at DESC),
INDEX idx_channel (clinic_id, channel),
INDEX idx_unread (conversation_id, read_at WHERE read_at IS NULL)
);

-- Internal notes (staff-only, not visible to patient)
CREATE TABLE conversation_notes (
  note_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  conversation_id UUID NOT NULL REFERENCES unified_conversations(conversation_id),

  note_text TEXT NOT NULL,
  author_id UUID NOT NULL REFERENCES clinic_users(user_id),

  -- Mentions for team collaboration
  mentioned_users UUID[] DEFAULT '{}', -- @mentions trigger notifications

  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP,

  INDEX idx_conversation (conversation_id, created_at DESC)
);

```

Backend API Endpoints (Node.js/Express)

```

// POST /api/unified-inbox/conversations
// Get paginated list of conversations for clinic

```

```
router.get('/unified-inbox/conversations', authenticate, async (req, res) => {
  const { clinic_id, status, sort_by } = req.query;

  const conversations = await db.query(`
    SELECT
      c.conversation_id,
      c.status,
      c.last_message_at,
      c.assigned_to,
      p.first_name,
      p.last_name,
      p.phone_primary,
      p.email,
      p.segment,
      COUNT(m.message_id) FILTER (WHERE m.read_at IS NULL) as unread_count
    FROM unified_conversations c
    JOIN patients p ON c.patient_id = p.patient_id
    LEFT JOIN unified_messages m ON c.conversation_id = m.conversation_id
    WHERE c.clinic_id = $1 AND ($2::text IS NULL OR c.status = $2)
    GROUP BY c.conversation_id, p.patient_id
    ORDER BY ${sort_by === 'newest' ? 'c.last_message_at DESC' : 'c.created_at
DESC'}
    LIMIT 50
  `, [clinic_id, status]);

  res.json(conversations);
});

// GET /api/unified-inbox/conversations/:conversation_id
// Get full conversation thread
router.get('/unified-inbox/conversations/:id', authenticate, async (req, res) => {
  const { id } = req.params;

  const conversation = await db.query(`
    SELECT * FROM unified_conversations WHERE conversation_id = $1
  `, [id]);

  const messages = await db.query(`
    SELECT
      message_id,
      channel,
      direction,
      message_text,
      message_type,
      sent_at,
      delivered_at,
      read_at,
      created_by_user_id,
      from_number,
      to_number
    FROM unified_messages
    WHERE conversation_id = $1
    ORDER BY sent_at ASC
  `, [id]);
```

```
const notes = await db.query(`
  SELECT
    note_id,
    note_text,
    author_id,
    created_at
  FROM conversation_notes
  WHERE conversation_id = $1
  ORDER BY created_at DESC
`, [id]);

// Mark messages as read
await db.query(`
  UPDATE unified_messages
  SET read_at = NOW()
  WHERE conversation_id = $1 AND read_at IS NULL
`, [id]);

res.json({
  conversation: conversation[0],
  messages,
  notes
});
});

// POST /api/unified-inbox/messages
// Send message via preferred channel
router.post('/unified-inbox/messages', authenticate, async (req, res) => {
  const { conversation_id, message_text, channel } = req.body;

  // Get conversation + patient
  const { patient_id } = await db.query(
    'SELECT patient_id FROM unified_conversations WHERE conversation_id = $1',
    [conversation_id]
  );

  const { phone_primary, email } = await db.query(
    'SELECT phone_primary, email FROM patients WHERE patient_id = $1',
    [patient_id]
  );

  // Determine channel if not specified (use patient's preference)
  const send_channel = channel || patient.preferred_channel || 'sms';

  let external_message_id;

  if (send_channel === 'sms') {
    const result = await twilioClient.messages.create({
      from: TWILIO_PHONE_NUMBER,
      to: phone_primary,
      body: message_text
    });
    external_message_id = result.sid;
  }
});
```

```
} else if (send_channel === 'whatsapp') {
  const result = await twilioClient.messages.create({
    from: `whatsapp:${TWILIO_WHATSAPP_NUMBER}`,
    to: `whatsapp:${phone_primary}`,
    body: message_text
  });
  external_message_id = result.sid;
} else if (send_channel === 'email') {
  const result = await sendgridClient.mail.send({
    from: CLINIC_EMAIL,
    to: email,
    subject: 'Message from ' + clinic.organization_name,
    text: message_text
  });
  external_message_id = result.headers['x-message-id'];
}

// Store in unified_messages
const message = await db.query(`
  INSERT INTO unified_messages (
    conversation_id, channel, direction, message_text, external_message_id,
    sent_at, created_by_user_id
  ) VALUES ($1, $2, $3, $4, $5, NOW(), $6)
  RETURNING message_id
`, [conversation_id, send_channel, 'outbound', message_text,
external_message_id, req.user.user_id]);

// Update conversation's last_message_at
await db.query(`
  UPDATE unified_conversations SET last_message_at = NOW()
  WHERE conversation_id = $1
`, [conversation_id]);

res.json({ message_id: message[0].message_id, status: 'sent' });
});

// POST /api/unified-inbox/notes
// Add internal note (staff-only)
router.post('/unified-inbox/notes', authenticate, async (req, res) => {
  const { conversation_id, note_text, mentioned_user_ids } = req.body;

  const note = await db.query(`
    INSERT INTO conversation_notes (
      conversation_id, note_text, author_id, mentioned_users
    ) VALUES ($1, $2, $3, $4)
    RETURNING note_id, created_at
  `, [conversation_id, note_text, req.user.user_id, mentioned_user_ids]);

  // Send notifications to mentioned users
  for (const user_id of mentioned_user_ids) {
    await notificationService.sendNotification(user_id, {
      type: 'mention',
      message: `${req.user.full_name} mentioned you in a conversation note`,
      conversation_id
    });
  }
});
```

```

    });
  }

  res.json(note[0]);
});

```

Frontend Component (React)

```

// components/UnifiedInbox/ConversationThread.tsx
import React, { useState, useEffect } from 'react';
import { format } from 'date-fns';

export function ConversationThread({ conversationId }: { conversationId: string })
{
  const [messages, setMessages] = useState<Message[]>([]);
  const [notes, setNotes] = useState<Note[]>([]);
  const [conversation, setConversation] = useState<Conversation | null>(null);
  const [newMessage, setNewMessage] = useState('');
  const [selectedChannel, setSelectedChannel] = useState<'sms' | 'whatsapp' | 'email'>('whatsapp');
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    // Fetch conversation data
    fetch(`/api/unified-inbox/conversations/${conversationId}`)
      .then(r => r.json())
      .then(data => {
        setConversation(data.conversation);
        setMessages(data.messages);
        setNotes(data.notes);
        setLoading(false);
      });

    // WebSocket for real-time updates
    const ws = new
    WebSocket(`wss://api.reactivateai.com/ws/conversations/${conversationId}`);
    ws.onmessage = (event) => {
      const data = JSON.parse(event.data);
      if (data.type === 'new_message') {
        setMessages(prev => [...prev, data.message]);
      } else if (data.type === 'new_note') {
        setNotes(prev => [...prev, data.note]);
      }
    };

    return () => ws.close();
  }, [conversationId]);

  const handleSendMessage = async () => {
    if (!newMessage.trim()) return;

```

```

const response = await fetch('/api/unified-inbox/messages', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    conversation_id: conversationId,
    message_text: newMessage,
    channel: selectedChannel
  })
});

if (response.ok) {
  setNewMessage('');
  // Message will appear via WebSocket
}
};

if (loading) return <div>Loading...</div>;

return (
  <div className="conversation-thread">
    {/* Patient info header */}
    <div className="conversation-header">
      <div>
        <h2>{conversation?.patient_name}</h2>
        <p>{conversation?.patient_phone}</p>
        <span className={`segment-badge segment-${conversation?.segment}`}>
          {conversation?.segment}
        </span>
      </div>
    </div>

    {/* Messages thread */}
    <div className="messages-container">
      {messages.map((msg) => (
        <div key={msg.message_id} className={`message
message-${msg.direction}`}>
          {/* Channel badge */}
          <span className="channel-badge" title={msg.channel}>
            {msg.channel === 'whatsapp' && '💬'}
            {msg.channel === 'sms' && '📱'}
            {msg.channel === 'email' && '✉️'}
          </span>

          {/* Message content */}
          <div className="message-content">
            <p>{msg.message_text}</p>
          </div>

          {/* Delivery status */}
          <div className="message-status">
            {msg.failed_at && <span className="failed">Failed</span>}
            {msg.read_at && <span className="read">Read</span>}
            {msg.delivered_at && !msg.read_at && <span
className="delivered">Delivered</span>}

```

```

        {!msg.delivered_at && <span className="sending">Sending</span>}
        <time>{format(new Date(msg.sent_at), 'HH:mm')}</time>
      </div>
    </div>
  )})
</div>

{/* Message composer */}
<div className="compose-area">
  {/* Channel selector */}
  <div className="channel-selector">
    <button
      className={selectedChannel === 'whatsapp' ? 'selected' : ''}
      onClick={() => setSelectedChannel('whatsapp')}
    >
      WhatsApp
    </button>
    <button
      className={selectedChannel === 'sms' ? 'selected' : ''}
      onClick={() => setSelectedChannel('sms')}
    >
      SMS
    </button>
    <button
      className={selectedChannel === 'email' ? 'selected' : ''}
      onClick={() => setSelectedChannel('email')}
    >
      Email
    </button>
  </div>

  {/* Message input */}
  <textarea
    value={newMessage}
    onChange={(e) => setNewMessage(e.target.value)}
    placeholder="Type message..."
  />

  {/* Send button */}
  <button onClick={handleSendMessage} className="btn-primary">
    Send
  </button>
</div>
</div>
);
}

```

PHASE 1: 87% NO-SHOW PREDICTION AI (Weeks 5-8)

Python AI Service

```

# ai_service/models/no_show_predictor.py
import tensorflow as tf
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
import joblib

class NoShowPredictionModel:
    """
    12-feature neural network for no-show prediction
    Expected accuracy: 87%+ on test set
    """

    def __init__(self):
        self.model = None
        self.scaler = None
        self.feature_names = [
            'days_since_visit',
            'historical_no_show_rate',
            'is_early_morning',
            'is_late_evening',
            'is_weekend_appointment',
            'days_until_appointment',
            'cancellation_rate',
            'appointment_type_risk',
            'is_insurance_verified',
            'severe_weather_forecast',
            'reminder_opened',
            'sent_via_preferred_channel'
        ]

    def engineer_features(self, appointments: pd.DataFrame) -> pd.DataFrame:
        """Engineer 12 features for no-show prediction"""
        df = appointments.copy()

        # 1. Days since last visit
        df['days_since_visit'] = (pd.Timestamp.now() -
            df['patient_last_visit_date']).dt.days

        # 2. Historical no-show rate
        historical_rates = df.groupby('patient_id')['was_no_show'].apply(lambda x:
            x.sum() / len(x))
        df['historical_no_show_rate'] =
            df['patient_id'].map(historical_rates).fillna(0.1)

        # 3. Time of day effects
        df['appointment_hour'] = pd.to_datetime(df['appointment_time']).dt.hour
        df['is_early_morning'] = ((df['appointment_hour'] >= 7) &
            (df['appointment_hour'] <= 10)).astype(int)
        df['is_late_evening'] = ((df['appointment_hour'] >= 18) &
            (df['appointment_hour'] <= 21)).astype(int)

        # 4. Day of week effect

```

```

        df['appointment_dow'] =
pd.to_datetime(df['appointment_date']).dt.dayofweek
        df['is_weekend_appointment'] = (df['appointment_dow'] >= 5).astype(int)

# 5. Distance from appointment
        df['days_until_appointment'] = (pd.to_datetime(df['appointment_date']) -
pd.Timestamp.now()).dt.days

# 6. Cancellation history
        cancellation_rates = df.groupby('patient_id')
['was_cancelled'].apply(lambda x: x.sum() / len(x))
        df['cancellation_rate'] =
df['patient_id'].map(cancellation_rates).fillna(0)

# 7. Appointment type risk
        procedure_risk = {'routine': 0.15, 'procedure': 0.25, 'surgery': 0.12,
'followup': 0.20}
        df['appointment_type_risk'] =
df['appointment_type'].map(procedure_risk).fillna(0.15)

# 8. Insurance verification
        df['is_insurance_verified'] = df['insurance_verified'].astype(int)

# 9. Weather risk
        df['severe_weather_forecast'] = 0 # Integrate weather API in production

# 10. Reminder engagement
        df['reminder_opened'] = df['reminder_email_opened'].astype(int)

# 11. Communication channel match
        df['sent_via_preferred_channel'] = (df['reminder_channel'] ==
df['patient_preferred_channel']).astype(int)

        return df[self.feature_names + ['appointment_id', 'was_no_show']]

def train(self, X_train: np.ndarray, y_train: np.ndarray):
    """Train neural network"""
    self.scaler = StandardScaler()
    X_scaled = self.scaler.fit_transform(X_train)

    self.model = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(12,)),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dropout(0.3),
        tf.keras.layers.Dense(32, activation='relu'),
        tf.keras.layers.Dropout(0.3),
        tf.keras.layers.Dense(16, activation='relu'),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(1, activation='sigmoid')
    ])

    self.model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),

```

```

        loss='binary_crossentropy',
        metrics=['accuracy', tf.keras.metrics.AUC(name='auc')]
    )

    history = self.model.fit(
        X_scaled, y_train,
        epochs=100,
        batch_size=32,
        validation_split=0.2,
        class_weight={0: 1.0, 1: 2.0}
    )

    return history

def predict_batch(self, appointments: pd.DataFrame) -> list:
    """Predict no-show risk for batch of appointments"""
    X = self.engineer_features(appointments)
    X_scaled = self.scaler.transform(X[self.feature_names])

    risk_scores = self.model.predict(X_scaled, verbose=0).flatten()

    results = []
    for idx, (app_id, score) in enumerate(zip(X['appointment_id'],
risk_scores)):
        if score >= 0.7:
            risk_level = 'HIGH'
        elif score >= 0.4:
            risk_level = 'MEDIUM'
        else:
            risk_level = 'LOW'

        results.append({
            'appointment_id': app_id,
            'no_show_risk_score': float(score),
            'risk_level': risk_level
        })

    return results

```

FastAPI Endpoint

```

# FastAPI endpoint for inference
from fastapi import FastAPI, HTTPException
import asyncio

app = FastAPI()
predictor = NoShowPredictionModel()

@app.post('/predict/no-show-batch')
async def predict_no_show(appointments_data: list):
    """Endpoint for backend to request no-show predictions"""

```

```
try:
    df = pd.DataFrame(appointments_data)
    predictions = predictor.predict_batch(df)
    return {'predictions': predictions, 'batch_size': len(predictions)}
except Exception as e:
    raise HTTPException(status_code=500, detail=str(e))
```

PHASE 2: FACEBOOK MESSENGER INTEGRATION (Weeks 9-12)

```
// services/facebookService.ts
import axios from 'axios';

class FacebookMessengerService {
  async handleInboundMessage(webhook: MetaWebhook) {
    const message = webhook.entry[0].messaging[0];
    const patientFacebookId = message.sender.id;
    const messageText = message.message.text;

    // Find or create patient
    let patient = await Patient.findOne({ facebook_id: patientFacebookId });

    if (!patient) {
      const facebookProfile = await this.fetchFacebookProfile(patientFacebookId);
      patient = await Patient.create({
        first_name: facebookProfile.first_name,
        last_name: facebookProfile.last_name,
        facebook_id: patientFacebookId,
        clinic_id: webhook.clinic_id
      });
    }

    // Create conversation
    let conversation = await UnifiedConversation.findOne({
      patient_id: patient.patient_id,
      status: 'active'
    });

    if (!conversation) {
      conversation = await UnifiedConversation.create({
        patient_id: patient.patient_id,
        clinic_id: webhook.clinic_id
      });
    }

    // Store message
    await UnifiedMessage.create({
      conversation_id: conversation.conversation_id,
      channel: 'facebook_messenger',
      direction: 'inbound',
      message_text: messageText,
```

```
        external_message_id: message.mid,
        sent_at: new Date(message.timestamp * 1000)
    });
}

async sendMessage(patientFacebookId: string, messageText: string) {
    const response = await axios.post(
        `https://graph.instagram.com/v19.0/me/messages`,
        {
            messaging_product: 'instagram',
            message: { text: messageText },
            recipient: { id: patientFacebookId }
        },
        {
            headers: {
                Authorization: `Bearer ${process.env.META_ACCESS_TOKEN}`
            }
        }
    );

    return response.data;
}
```

PHASE 3: EPIC FHIR INTEGRATION (Weeks 13-16)

```
// services/emr/epicService.ts
import axios from 'axios';

class EpicFHIRService {
    private fhirUrl: string;
    private clientId: string;
    private clientSecret: string;
    private accessToken: string;

    async authenticate() {
        const response = await axios.post(
            `${this.fhirUrl}/oauth2/token`,
            new URLSearchParams({
                grant_type: 'client_credentials',
                client_id: this.clientId,
                client_secret: this.clientSecret
            })
        );

        this.accessToken = response.data.access_token;
    }

    async syncPatients(clinic_id: string) {
        const response = await axios.get(
```

```
`${this.fhirUrl}/Patient`,
{
  params: { 'general-practitioner': clinic_id, _count: 1000 },
  headers: { Authorization: `Bearer ${this.accessToken}` }
}
);

const patients = response.data.entry.map(entry => ({
  external_patient_id: entry.resource.id,
  first_name: entry.resource.name[0]?.given[0],
  last_name: entry.resource.name[0]?.family,
  email: entry.resource.telecom?.find(t => t.system === 'email')?.value,
  phone_primary: entry.resource.telecom?.find(t => t.system ===
'phone')?.value,
  clinic_id
})));

for (const patient of patients) {
  await Patient.upsert(
    { external_patient_id: patient.external_patient_id, clinic_id:
patient.clinic_id },
    patient
  );
}

return patients;
}

async syncAppointments(clinic_id: string) {
  const response = await axios.get(
    `${this.fhirUrl}/Appointment`,
    {
      params: {
        'practitioner:Practitioner.organization': clinic_id,
        status: 'booked',
        date: `>=${new Date().toISOString()}`,
        _count: 500
      },
      headers: { Authorization: `Bearer ${this.accessToken}` }
    }
  );
  );

  const appointments = response.data.entry.map(entry => {
    const appt = entry.resource;
    return {
      external_appointment_id: appt.id,
      appointment_datetime: appt.start,
      clinic_id
    };
  });

  for (const appointment of appointments) {
    await Appointment.upsert(
      { external_appointment_id: appointment.external_appointment_id },
```

```
        appointment
    );
}

return appointments;
}
```

This is the complete implementation guide with production-ready code. All three files should now be available in your workspace.