Implementation Plan: Database-Powered Assistant with ChatGPT-like Experience

Here's a comprehensive plan to create a ChatGPT-like assistant that uses your therapy practice database as its knowledge source:

Phase 1: Foundation Enhancement (Week 1)

Day 1-2: Advanced Context Management

Update AgentContext.tsx to maintain conversation history and handle stateful interactions

Implement conversation memory system for follow-up questions

Create context persistence across different questions

Design session-based context management

// Enhancement to AgentContext.tsx

export function AgentProvider({ children }: { children: ReactNode }) {

// Existing state...

const [conversationMemory, setConversationMemory] = useState<{

lastQuery?: string;

lastTopic?: string;

recentEntities?: string[];

activeFilters?: Record<string, any>;

}>({});

// Add context management functions

const updateConversationMemory = (newMemory: Partial<typeof conversationMemory>) => {

setConversationMemory(prev => ({ ...prev, ...newMemory }));

};

// Add context-aware query processing

const processQueryWithContext = async (query: string) => {

// Process query with awareness of conversation history

const response = await queryProcessor.processQuery(query, {

activeClientId,

activeGoalId,

conversationHistory,

conversationMemory

});

// Update conversation memory based on query and response

updateConversationMemory({

lastQuery: query,

lastTopic: detectTopicFromQuery(query),

// Other updates...

});

return response;

};

// Provider value...

}

Day 3-4: Enhanced Query Understanding

Improve the query parser to handle a wider range of questions

Implement concept recognition for therapy domain terms

Add entity extraction for identifying key elements in questions

Create classification for different types of general vs. specific questions

// Enhancement to queryParser.ts

export function parseQueryIntent(query: string, context: QueryContext): QueryIntent {

const lowercaseQuery = query.toLowerCase();

// Extract entities from query

const entities = extractEntities(lowercaseQuery);

// Detect if query is about general concepts or specific client data

const isGeneralQuestion = detectGeneralQuestion(lowercaseQuery, entities);

if (containsAny(lowercaseQuery, BUDGET\_TERMS)) {

if (isGeneralQuestion) {

return {

type: 'BUDGET\_ANALYSIS',

isGeneral: true,

subtopic: detectBudgetSubtopic(lowercaseQuery)

};

} else {

return {

type: 'BUDGET\_ANALYSIS',

clientId: context.activeClientId,

isGeneral: false

};

}

}

// Similar pattern for other question types...

}

function detectGeneralQuestion(query: string, entities: ExtractedEntity[]): boolean {

// Check for general question patterns

const generalPatterns = [

/what (is|are) .\*/i,

/how (do|does|can) .\*/i,

/tell me about .\*/i,

/explain .\*/i

];

// Check if query matches general patterns and doesn't contain specific entities

return generalPatterns.some(pattern => pattern.test(query)) &&

!entities.some(e => e.type === 'ClientName' || e.type === 'ClientID');

}

Day 5: Database Knowledge Service

Create a knowledge retrieval service to access database information

Implement abstract queries for general information

Create specialized data aggregation methods

Build system to retrieve metadata about database structure

// New file: client/src/lib/services/knowledgeService.ts

export const knowledgeService = {

// Get general information about budgets

async getGeneralBudgetInfo(subtopic?: string): Promise<any> {

// Query database for aggregated budget data or metadata

const aggregateData = await apiRequest('GET', '/api/knowledge/budgets', { subtopic });

return aggregateData;

},

// Get general information about client progress

async getGeneralProgressInfo(subtopic?: string): Promise<any> {

// Query database for aggregated progress data

const progressStats = await apiRequest('GET', '/api/knowledge/progress', { subtopic });

return progressStats;

},

// Get metadata about database schema

async getDatabaseMetadata(table?: string): Promise<any> {

const metadata = await apiRequest('GET', '/api/knowledge/metadata', { table });

return metadata;

},

// Get therapy domain concepts

async getTherapyDomainConcepts(concept?: string): Promise<any> {

const conceptsData = await apiRequest('GET', '/api/knowledge/concepts', { concept });

return conceptsData;

}

};

Phase 2: Response Generation (Week 2)

Day 1-2: Template System

Create a response template system for generating natural language

Build template library for different question types

Implement variable substitution for database values

Create multi-turn conversation templates

// New file: client/src/lib/agent/responseTemplates.ts

type TemplateData = Record<string, any>;

interface ResponseTemplate {

template: string;

conditions: (data: TemplateData) => boolean;

priority: number;

}

// Budget templates

const budgetTemplates: ResponseTemplate[] = [

{

template: "Based on our therapy practice data, budget plans typically include {{categories}} with an average allocation of {{avgAllocation}} per category. The most common funding source is {{topFundingSource}}.",

conditions: (data) => data.isGeneral && data.subtopic === 'overview',

priority: 1

},

{

template: "Our therapy practice uses {{budgetingApproach}} for budget planning. This involves {{budgetingDescription}}.",

conditions: (data) => data.isGeneral && data.subtopic === 'process',

priority: 1

},

// More templates...

];

// Client-specific templates

// Progress templates

// Strategy templates

// ...

export function selectTemplate(templates: ResponseTemplate[], data: TemplateData): string {

// Find matching template with highest priority

const matchingTemplates = templates

.filter(t => t.conditions(data))

.sort((a, b) => b.priority - a.priority);

if (matchingTemplates.length === 0) {

return "I don't have specific information about that.";

}

return matchingTemplates[0].template;

}

export function renderTemplate(template: string, data: TemplateData): string {

// Replace variables in template with actual data

return template.replace(/\{\{(\w+)\}\}/g, (match, variable) => {

return data[variable] !== undefined ? data[variable] : match;

});

}

Day 3-4: Response Processor

Implement an enhanced response processor

Create adaptive response generation based on query context

Implement fallback strategies for unknown questions

Add supportive explanations and details for complex data

// Enhancement to queryProcessor.ts

async function processGeneralBudgetQuery(intent: QueryIntent, context: QueryContext): Promise<AgentResponse> {

try {

// Get the specific budget subtopic if available

const subtopic = intent.subtopic || 'overview';

// Retrieve general budget information from knowledge service

const budgetInfo = await knowledgeService.getGeneralBudgetInfo(subtopic);

// Select appropriate template based on data and context

const template = selectTemplate(budgetTemplates, {

isGeneral: true,

subtopic,

...budgetInfo

});

// Render template with data

const content = renderTemplate(template, budgetInfo);

return {

content,

confidence: 0.85,

data: budgetInfo,

visualizationHint: budgetInfo.hasVisualization ? 'BUBBLE\_CHART' : 'NONE'

};

} catch (error) {

console.error('Error processing general budget query:', error);

return {

content: "I'm having trouble accessing information about budgets right now.",

confidence: 0.4

};

}

}

// Similar functions for other general query types...

Day 5: Multi-turn Conversation Support

Implement follow-up question handling

Add reference resolution for pronouns ("it", "them", etc.)

Implement context carryover between questions

Create ellipsis resolution (implied subjects from previous queries)

// Enhancement to conversationManager.ts

export function resolveReferences(query: string, context: QueryContext): string {

const { conversationHistory } = context;

if (conversationHistory.length === 0) return query;

const lastMessage = conversationHistory[conversationHistory.length - 1];

// Replace pronouns with their referents

let resolvedQuery = query.replace(/\b(it|this|that|they|them|these|those)\b/gi, (match) => {

// Determine what the pronoun refers to based on context

const referent = findReferent(match, lastMessage.content);

return referent || match;

});

// Handle incomplete questions by carrying over context

if (!containsSubject(resolvedQuery) && context.lastTopic) {

resolvedQuery = `${context.lastTopic} ${resolvedQuery}`;

}

return resolvedQuery;

}

function findReferent(pronoun: string, previousContent: string): string | null {

// Implementation to find what a pronoun refers to in previous content

// This would use NLP techniques to identify likely referents

// ...

return null; // Placeholder

}

Phase 3: Backend API Enhancements (Week 3)

Day 1-2: Knowledge API Endpoints

Create backend API endpoints for knowledge retrieval

Implement aggregation queries for general information

Add metadata endpoints for schema information

Create domain concept lookup endpoints

// New routes in server/routes.ts

// Knowledge API routes

app.get('/api/knowledge/budgets', async (req, res) => {

try {

const subtopic = req.query.subtopic as string;

// Retrieve aggregated budget data

let data: any = {};

if (subtopic === 'overview') {

// Get budget categories

const categories = await db.query(`

SELECT DISTINCT category FROM budget\_items

ORDER BY category

`);

// Get average allocations by category

const allocations = await db.query(`

SELECT category, AVG(unit\_price \* quantity) as avg\_allocation

FROM budget\_items

GROUP BY category

ORDER BY avg\_allocation DESC

`);

// Get top funding sources

const fundingSources = await db.query(`

SELECT funding\_source, COUNT(\*) as count

FROM budget\_settings

GROUP BY funding\_source

ORDER BY count DESC

LIMIT 1

`);

data = {

categories: categories.map(c => c.category).join(', '),

avgAllocation: formatCurrency(allocations[0]?.avg\_allocation || 0),

topFundingSource: fundingSources[0]?.funding\_source || 'NDIS'

};

} else if (subtopic === 'process') {

// Get budgeting approach information

data = {

budgetingApproach: 'client-centered allocation',

budgetingDescription: 'analyzing client needs, prioritizing goals, and allocating resources based on evidence-based practices'

};

}

// Other subtopics...

res.json(data);

} catch (error) {

console.error('Error retrieving budget knowledge:', error);

res.status(500).json({ error: 'Failed to retrieve budget knowledge' });

}

});

// Similar endpoints for progress, strategies, therapy concepts...

Day 3-4: Query Analysis Services

Enhance query understanding capabilities on the backend

Implement domain-specific entity recognition

Add intent classification service

Create advanced query parsing utilities

// New file: server/services/queryAnalysisService.ts

export interface EntityRecognitionResult {

entities: {

text: string;

type: string;

startPos: number;

endPos: number;

}[];

}

export interface IntentClassificationResult {

primaryIntent: string;

confidence: number;

supportingIntents?: string[];

}

export const queryAnalysisService = {

// Recognize therapy domain entities in query

recognizeEntities(query: string): EntityRecognitionResult {

// Implementation using regex, lookup tables, or NLP library

// ...

return { entities: [] }; // Placeholder

},

// Classify the intent of a query

classifyIntent(query: string): IntentClassificationResult {

// Implementation using keyword matching, patterns, or ML-based classifier

// ...

return { primaryIntent: 'unknown', confidence: 0.5 }; // Placeholder

},

// Parse complex queries into structured representation

parseComplexQuery(query: string): any {

// Implementation for breaking down multi-part or complex queries

// ...

return {}; // Placeholder

}

};

Day 5: Database Access Optimization

Optimize database queries for real-time conversation

Create caching layer for frequently accessed information

Implement query optimization for knowledge retrieval

Add load balancing for high-traffic queries

// New file: server/services/cacheService.ts

type CacheEntry = {

data: any;

expiry: number;

};

class KnowledgeCache {

private cache: Map<string, CacheEntry> = new Map();

private defaultTtl = 3600000; // 1 hour in milliseconds

// Get data from cache or execute provider function

async getOrCompute(key: string, provider: () => Promise<any>, ttl?: number): Promise<any> {

const now = Date.now();

const entry = this.cache.get(key);

// Return cached data if valid

if (entry && entry.expiry > now) {

return entry.data;

}

// Execute provider function

const data = await provider();

// Store in cache

this.cache.set(key, {

data,

expiry: now + (ttl || this.defaultTtl)

});

return data;

}

// Invalidate a specific cache entry

invalidate(key: string): void {

this.cache.delete(key);

}

// Clear entire cache

clear(): void {

this.cache.clear();

}

}

export const knowledgeCache = new KnowledgeCache();

Phase 4: Integration & Testing (Week 4)

Day 1-2: UI Enhancements

Improve the agent UI for conversational experience

Implement typing indicators and response animations

Add suggestion chips for follow-up questions

Create UI for displaying different response types

// Enhancement to AgentBubble.tsx

export function AgentBubble() {

const { conversationHistory, isProcessingQuery } = useAgent();

return (

<div className="agent-bubble">

{conversationHistory.map((message, index) => (

<div

key={index}

className={`message ${message.role === 'user' ? 'user-message' : 'agent-message'}`}

>

{message.role === 'assistant' && (

<>

<div className="message-content">{message.content}</div>

{/\* Follow-up suggestions \*/}

{message.suggestedFollowUps && (

<div className="suggestion-chips">

{message.suggestedFollowUps.map((suggestion, i) => (

<button

key={i}

className="suggestion-chip"

onClick={() => handleSuggestionClick(suggestion)}

>

{suggestion}

</button>

))}

</div>

)}

</>

)}

{message.role === 'user' && (

<div className="message-content">{message.content}</div>

)}

</div>

))}

{/\* Typing indicator \*/}

{isProcessingQuery && (

<div className="typing-indicator">

<span></span>

<span></span>

<span></span>

</div>

)}

</div>

);

}

Day 3-4: Integration Testing

Test the integrated assistant with various scenarios

Create test suite for different query types

Test multi-turn conversations

Verify database knowledge retrieval accuracy

// Test scenarios for agent capabilities

const testScenarios = [

{

name: 'General Budget Questions',

queries: [

'What are budget plans?',

'How do budget allocations work?',

'Tell me about NDIS funding'

],

expectedResponsePatterns: [

/budget plans typically include/i,

/allocations work by/i,

/NDIS funding is/i

]

},

{

name: 'Follow-up Questions',

queries: [

'Tell me about goal tracking',

'How is it measured?',

'Which clients are doing well with this?'

],

expectedResponsePatterns: [

/goal tracking helps/i,

/measured using/i,

/select a client first/i

]

},

// More test scenarios...

];

// Manual or automated testing function

async function testAgentCapabilities() {

for (const scenario of testScenarios) {

console.log(`Testing scenario: ${scenario.name}`);

for (let i = 0; i < scenario.queries.length; i++) {

const query = scenario.queries[i];

const expectedPattern = scenario.expectedResponsePatterns[i];

console.log(` Query: "${query}"`);

const response = await agent.processQuery(query);

const matches = expectedPattern.test(response.content);

console.log(` Response: "${response.content.substring(0, 50)}..."`);

console.log(` Match expected pattern: ${matches ? '✓' : '✗'}`);

}

}

}

Day 5: Refinement and Deployment

Final refinements based on testing

Improve response quality where needed

Fix any identified issues

Optimize performance

Prepare for deployment

Document new capabilities

Prepare user guide for conversational assistant

Create admin tools for monitoring and maintenance

Technical Implementation Details

Core Components

Conversation Manager

Maintains state across multiple turns

Resolves references and context

Manages suggestion generation

Knowledge Retrieval System

Connects to database through optimized queries

Aggregates information for general questions

Retrieves specific client data when needed

Response Generation

Uses templates with variable substitution

Constructs natural language responses

Incorporates database facts into coherent answers

Entity Recognition

Identifies therapy domain concepts

Recognizes client names and IDs

Extracts important parameters from queries

Database Integration

The implementation uses the existing database but adds:

New aggregation queries for general knowledge

Metadata tables to understand data structure

Caching layer for performance optimization

Knowledge-oriented API endpoints

Extensions Beyond Initial Plan

This plan extends your original implementation by:

Adding general knowledge capabilities

Creating conversation memory for follow-ups

Implementing reference resolution

Building a template system for natural responses

Adding suggestion chips to guide conversations

Would you like me to begin implementing this plan, focusing first on the conversation management enhancements and general knowledge capabilities? Or would you prefer I adjust any aspects of this implementation approach?