ASTERIA AI CONCIERGE - COMPLETE IMPLEMENTATION BLUEPRINT v2.0

ASTERIA AI CONCIERGE - COMPLETE IMPLEMENTATION BLUEPRINT v2.0

Based on your diagnostic analysis and implementation discussions, I'm providing a comprehensive, consolidated plan that resolves all duplicates, leverages your existing advanced Agent Loop architecture, and creates a clear path to full autonomy. This document serves as your single source of truth.

📋 EXECUTIVE SUMMARY

Current State: You have TWO parallel systems - a simple journey-based chat (active) and a sophisticated Agent Loop system (built but unused). This creates confusion and missed opportunities.

Target State: One unified system leveraging the advanced Agent Loop architecture with full Firebase/Firestore integration, Slack orchestration, and RAG knowledge base.

Critical Decision: **USE THE AGENT LOOP SYSTEM** - it's already built and far superior.

🏗️ PHASE 1: SYSTEM CONSOLIDATION (Week 1)

Step 1.1: Fix Immediate Compilation Error

```
// src/components/TestApiButton.tsx
import { useState } from 'react';

export default function TestApiButton() {
  const [response, setResponse] = useState<string>('');
  const [loading, setLoading] = useState(false);

  // Only show in development
  if (process.env.NODE_ENV !== 'development') {
    return null;
  }

  const testApi = async () => {
    setLoading(true);
    try {
      const res = await fetch('/api/test', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ test: true })
      });
      const data = await res.json();
      setResponse(JSON.stringify(data, null, 2));
    } catch (error) {
      setResponse(`Error: ${error}`);
    } finally {
      setLoading(false);
    }
  };

  return (
    <div className="fixed bottom-4 right-4 p-4 bg-gray-800 rounded-lg shadow
      <button
        onClick={testApi}
        disabled={loading}
        className="px-4 py-2 bg-blue-500 text-white rounded hover:bg-blue-600
      >
        {loading ? 'Testing...' : 'Test API'}
      </button>
      {response && (
        <pre className="mt-2 p-2 bg-gray-900 text-xs text-gray-300 rounded ove
          {response}
        </pre>
      )}
    </div>
  );
}
```

Step 1.2: Archive Current Simple System

```
#!/bin/bash
# Archive current simple system before migration to Agent Loop

# Create archive directory with timestamp
TIMESTAMP=$(date +"%Y%m%d_%H%M%S")
ARCHIVE_DIR="src/archive/simple-system-${TIMESTAMP}"

echo "📦 Archiving simple chat system to ${ARCHIVE_DIR}"
mkdir -p "${ARCHIVE_DIR}"

# Archive simple chat route
cp src/app/api/chat/route.ts "${ARCHIVE_DIR}/chat-route-simple.ts"

# Archive journey-based components
cp src/lib/services/journey.js "${ARCHIVE_DIR}/"
cp src/lib/services/classifier.js "${ARCHIVE_DIR}/"
cp src/lib/services/tickets.js "${ARCHIVE_DIR}/"
cp src/lib/services/extractor.js "${ARCHIVE_DIR}/"

# Archive basic notification systems
```

```
cp src/lib/services/slack.js "${ARCHIVE_DIR}/"
cp src/lib/services/sms.js "${ARCHIVE_DIR}/"

# Create migration notes
cat > "${ARCHIVE_DIR}/MIGRATION_NOTES.md" << EOF
# Simple System Archive - ${TIMESTAMP}

## Archived Components:
- Simple chat route with journey detection
- Basic classifier (7 service buckets)
- Journey phase tracking
- Basic Slack/SMS notifications

## Migration Path:
- Replacing with Agent Loop architecture from asteria-deploy/
- Consolidating duplicate services
- Upgrading to advanced notification system with throttling
EOF

echo "✅ Archive complete. Safe to proceed with Agent Loop integration."
```

Step 1.3: Consolidate Agent Loop System

```
// src/lib/agent/index.ts - Unified Agent System Export
export { AsteriaAgentLoop } from './agent_loop';
export { IntentPlanner } from './planner';
export { ServiceExecutor } from './executor';
export { InteractionReflector } from './reflector';
export { GoalChecker } from './goal_checker';

// Tool exports
export * from './tools/fetch_active_services';
export * from './tools/create_ticket';
export * from './tools/notify_concierge';
export * from './tools/escalate_to_human';
```

```
// Type exports
export type {
  AgentContext,
  AgentResponse,
  ServiceBucket,
  ToolResult,
  PlanStep,
  ReflectionInsight,
  GoalValidation
} from './types';

// Configuration
export const AGENT_CONFIG = {
  maxIterations: 10,
  planningModel: 'gpt-4o',
  executionModel: 'gpt-4o-mini',
  enableLearning: true,
  enableThrottling: true,
  notificationChannels: ['slack', 'sms', 'email'],
  serviceTimeouts: {
    aviation: 30000,
    dining: 15000,
    hotel: 20000,
    transportation: 15000,
    experience: 25000,
    shopping: 15000,
    custom: 30000
  }
};
```

Step 1.4: Unified Type System

```
// src/lib/agent/types.ts - Single Source of Truth for Types

export interface ServiceRequest {
  id: string;
  memberId: string;
  rawText: string;
  parsedJson: ParsedRequest;
  status: ServiceStatus;
  priority: Priority;
  assignedTo?: string;
  createdAt: Date;
  updatedAt: Date;
  quotePdfUrl?: string;
  conversationHistory: Message[];
}

export interface ParsedRequest {
  intent: ServiceBucket;
  confidence: number;
  entities: ExtractedEntities;
  preferences: MemberPreferences;
  constraints: ServiceConstraints;
}

export interface ExtractedEntities {
  dates?: DateRange[];
  locations?: Location[];
  people?: Person[];
  services?: string[];
  budget?: Budget;
  special_requests?: string[];
}

export interface ServiceConstraints {
  oneRoofRequired?: boolean;
  preferredHotels?: string[];
  dietaryRestrictions?: string[];
  accessibility?: string[];
}
```

```
export type ServiceBucket =
  | 'aviation'
  | 'dining'
  | 'hotel'
  | 'transportation'
  | 'experience'
  | 'shopping'
  | 'custom';

export type ServiceStatus =
  | 'NEW'
  | 'CLASSIFIED'
  | 'PLANNING'
  | 'EXECUTING'
  | 'QUOTE_SENT'
  | 'CONFIRMED'
  | 'FAILED'
  | 'ESCALATED';

export type Priority = 'LOW' | 'MEDIUM' | 'HIGH' | 'URGENT';

export interface AgentContext {
  request: ServiceRequest;
  member: MemberProfile;
  iteration: number;
  toolsUsed: string[];
  insights: ReflectionInsight[];
}

export interface AgentResponse {
  message: string;
  metadata: {
    intent: ServiceBucket;
    confidence: number;
    needsHuman: boolean;
    nextSteps?: string[];
    ticketId?: string;
    quotePdfUrl?: string;
  };
  suggestions?: ServiceSuggestion[];
  internalNotes?: string;
}

export interface PlanStep {
  step: number;
  action: string;
  tool?: string;
  parameters?: Record<string, any>;
  expectedOutcome: string;
}

export interface ToolResult {
  success: boolean;
  data?: any;
  error?: string;
  duration: number;
}

export interface ReflectionInsight {
  pattern: string;
  frequency: number;
  recommendation: string;
  impact: 'LOW' | 'MEDIUM' | 'HIGH';
}

export interface GoalValidation {
  achieved: boolean;
  score: number;
  missingElements?: string[];
  recommendations?: string[];
}
```

}

```
export interface MemberProfile {
  id: string;
  tier: 'GOLD' | 'PLATINUM' | 'ELITE';
  preferences: MemberPreferences;
  history: ServiceRequest[];
  totalSpend: number;
}

export interface MemberPreferences {
  communication: 'email' | 'sms' | 'whatsapp';
  brands: string[];
  dietary: string[];
  interests: string[];
}

export interface ServiceSuggestion {
  tier: 'good' | 'better' | 'extraordinary';
  service: string;
  price?: string;
  availability?: string;
  reasoning: string;
}

// Firebase/Firestore specific types
export interface FirestoreServiceRequest extends Omit<ServiceRequest, 'createc
  createdAt: FirebaseFirestore.Timestamp;
  updatedAt: FirebaseFirestore.Timestamp;
}

// Slack specific types
export interface SlackNotification {
  srId: string;
  blocks: any[];
  channel: string;
  priority: Priority;
}
```

```
// Message types for conversation tracking
export interface Message {
  role: 'user' | 'assistant' | 'system';
  content: string;
  timestamp: Date;
  metadata?: Record<string, any>;
}
```

Step 1.5: Consolidated Agent Loop Implementation

```
// src/lib/agent/agent_loop.ts - Main Agent Orchestrator
import { db } from '@/lib/firebase/admin';
import { IntentPlanner } from './planner';
import { ServiceExecutor } from './executor';
import { InteractionReflector } from './reflector';
import { GoalChecker } from './goal_checker';
import {
  AgentContext,
  AgentResponse,
  ServiceRequest,
  PlanStep,
  ToolResult,
  GoalValidation
} from './types';
import { AGENT_CONFIG } from './index';

export class AsteriaAgentLoop {
  private planner: IntentPlanner;
  private executor: ServiceExecutor;
  private reflector: InteractionReflector;
  private goalChecker: GoalChecker;

  constructor() {
    this.planner = new IntentPlanner();
    this.executor = new ServiceExecutor();
    this.reflector = new InteractionReflector();
    this.goalChecker = new GoalChecker();
  }

  async process(
    message: string,
    memberId: string,
    conversationId?: string
  ): Promise<AgentResponse> {
    // Initialize or fetch service request
    const serviceRequest = await this.initializeRequest(
      message,
      memberId,
      conversationId
    );

    // Get member profile for context
    const memberProfile = await this.getMemberProfile(memberId);

    // Initialize agent context
    const context: AgentContext = {
      request: serviceRequest,
      member: memberProfile,
      iteration: 0,
      toolsUsed: [],
      insights: []
    };

    try {
      // Phase 1: Plan
      console.log('🎯 Phase 1: Planning');
      const plan = await this.planner.createPlan(context);
      await this.updateRequestStatus(serviceRequest.id, 'PLANNING');

      // Phase 2: Execute
      console.log('⚡ Phase 2: Executing');
```

```
const results: ToolResult[] = [];

for (const step of plan.steps) {
  if (context.iteration >= AGENT_CONFIG.maxIterations) {
    console.log('⚠️ Max iterations reached, escalating');
    return await this.escalateToHuman(context, 'Max iterations reached');
  }

  const result = await this.executor.executeStep(step, context);
  results.push(result);
  context.toolsUsed.push(step.tool || 'direct_response');
  context.iteration++;

  // Update Firestore with progress
  await this.logProgress(serviceRequest.id, step, result);

  if (!result.success && step.tool) {
    console.log(`⚠️ Tool ${step.tool} failed, adjusting plan`);
    break;
  }
}

await this.updateRequestStatus(serviceRequest.id, 'EXECUTING');

// Phase 3: Reflect
console.log('🔍 Phase 3: Reflecting');
const insights = await this.reflector.analyze(context, results);
context.insights = insights;

// Store insights for future learning
if (AGENT_CONFIG.enableLearning) {
  await this.storeInsights(serviceRequest.id, insights);
}

// Phase 4: Goal Check
console.log('✅ Phase 4: Checking Goals');
```

```
    const validation = await this.goalChecker.validate(context, results);

    if (!validation.achieved) {
      console.log('🔄 Goals not met, retrying with adjustments');
      return await this.retryWithAdjustments(context, validation);
    }

    // Success - prepare response
    const response = await this.prepareResponse(context, results, validation);

    // Update final status
    await this.updateRequestStatus(
      serviceRequest.id,
      response.metadata.needsHuman ? 'ESCALATED' : 'QUOTE_SENT'
    );

    return response;

  } catch (error) {
    console.error('❌ Agent loop error:', error);
    await this.handleError(context, error as Error);
    return await this.escalateToHuman(context, 'System error occurred');
  }
}

private async initializeRequest(
  message: string,
  memberId: string,
  conversationId?: string
): Promise<ServiceRequest> {
  let request: ServiceRequest;

  if (conversationId) {
    // Fetch existing conversation
    const doc = await db.collection('service_requests').doc(conversationId).get()
    if (doc.exists) {
      request = doc.data() as ServiceRequest;
      request.conversationHistory.push({
        role: 'user',
        content: message,
        timestamp: new Date()
      });
      await doc.ref.update({
        conversationHistory: request.conversationHistory,
        updatedAt: new Date()
      });
    } else {
      request = await this.createNewRequest(message, memberId);
    }
  } else {
    request = await this.createNewRequest(message, memberId);
  }

  return request;
}

private async createNewRequest(
  message: string,
  memberId: string
): Promise<ServiceRequest> {
  const srId = `SR-${Date.now().toString().slice(-6)}`;
  const request: ServiceRequest = {
    id: srId,
    memberId,
    rawText: message,
    parsedJson: {
      intent: 'custom',
      confidence: 0,
      entities: {},
      preferences: {},
      constraints: {}
    },
    status: 'NEW',
    priority: 'MEDIUM',
    createdAt: new Date(),
    updatedAt: new Date(),
    conversationHistory: [{
      role: 'user',
      content: message,
      timestamp: new Date()
    }]
  };

  await db.collection('service_requests').doc(srId).set(request);
  return request;
}

private async getMemberProfile(memberId: string): Promise<MemberProfile> {
  const doc = await db.collection('members').doc(memberId).get();
  if (doc.exists) {
    return doc.data() as MemberProfile;
  }

  // Default profile for new members
  return {
    id: memberId,
    tier: 'GOLD',
    preferences: {
      communication: 'email',
      brands: [],
      dietary: [],
      interests: []
    },
    history: [],
    totalSpend: 0
  };
}
```

```
private async updateRequestStatus(
  requestId: string,
  status: ServiceRequest['status']
): Promise<void> {
  await db.collection('service_requests').doc(requestId).update({
    status,
    updatedAt: new Date()
  });
}

private async logProgress(
  requestId: string,
  step: PlanStep,
  result: ToolResult
): Promise<void> {
  await db.collection('sr_actions').add({
    srId: requestId,
    action: step.action,
    tool: step.tool,
    result: result.success ? 'success' : 'failed',
    duration: result.duration,
    timestamp: new Date(),
    data: result.data,
    error: result.error
  });
}

private async storeInsights(
  requestId: string,
  insights: ReflectionInsight[]
): Promise<void> {
  for (const insight of insights) {
    await db.collection('agent_insights').add({
      srId: requestId,
      pattern: insight.pattern,
      frequency: insight.frequency,
      recommendation: insight.recommendation,
      impact: insight.impact,
      timestamp: new Date()
    });
  }
}

private async prepareResponse(
  context: AgentContext,
  results: ToolResult[],
  validation: GoalValidation
): Promise<AgentResponse> {
  // Find ticket creation result if exists
  const ticketResult = results.find(r =>
    r.data?.ticketId || r.data?.ticket?.id
  );

  // Find quote generation result if exists
  const quoteResult = results.find(r =>
    r.data?.quotePdfUrl || r.data?.quote?.url
  );

  // Build response message based on service type
  const message = this.buildResponseMessage(
    context.request.parsedJson.intent,
    validation.score,
    ticketResult?.data,
    quoteResult?.data
  );

  return {
    message,
    metadata: {
      intent: context.request.parsedJson.intent,
      confidence: context.request.parsedJson.confidence,
      needsHuman: validation.score < 0.8,
      ticketId: ticketResult?.data?.ticketId,
      quotePdfUrl: quoteResult?.data?.quotePdfUrl,
      nextSteps: validation.recommendations
    },
    suggestions: await this.generateSuggestions(context),
    internalNotes: this.generateInternalNotes(context, results)
  };
}

private buildResponseMessage(
  intent: ServiceBucket,
  score: number,
  ticketData?: any,
  quoteData?: any
): string {
  if (score >= 0.9 && ticketData?.ticketId) {
    return `Perfect! I've created your ${intent} request (${ticketData.ticketId}). ` +
           `Our concierge team will begin working on this immediately. ` +
           `You'll receive updates via your preferred communication method.`;
  } else if (score >= 0.7) {
    return `I've captured your ${intent} request and our team is reviewing the det
           `We may need some additional information to ensure everything is perfec
  } else {
    return `I understand you need help with ${intent} services. ` +
           `Let me connect you with our specialized concierge team who can better
  }
}

private async generateSuggestions(
  context: AgentContext
): Promise<ServiceSuggestion[]> {
  // This would integrate with your service discovery system
  return [];
}

private generateInternalNotes(
  context: AgentContext,
  results: ToolResult[]
): string {
  const notes = [
    `Intent: ${context.request.parsedJson.intent}`,
    `Confidence: ${context.request.parsedJson.confidence}`,
    `Tools used: ${context.toolsUsed.join(', ')}`,
    `Iterations: ${context.iteration}`,
    `Member tier: ${context.member.tier}`
  ];

  if (context.insights.length > 0) {
    notes.push(`Key insights: ${context.insights[0].pattern}`);
  }

  return notes.join('\\n');
}

private async retryWithAdjustments(
  context: AgentContext,
  validation: GoalValidation
): Promise<AgentResponse> {
  if (context.iteration >= AGENT_CONFIG.maxIterations - 2) {
    return await this.escalateToHuman(context, 'Goals not achieved after retries')
  }

  // Adjust context based on validation feedback
  context.request.parsedJson.entities = {
    ...context.request.parsedJson.entities,
    special_requests: validation.missingElements
  };

  // Recursive call with adjusted context
  return await this.process(
    context.request.rawText,
    context.request.memberId,
    context.request.id
  );
}

private async escalateToHuman(
  context: AgentContext,
  reason: string
): Promise<AgentResponse> {
  // Use escalation tool
  const escalationResult = await this.executor.executeStep(
    {
      step: 999,
      action: 'Escalate to human concierge',
      tool: 'escalate_to_human',
      parameters: {
        requestId: context.request.id,
        reason,
        context: context.request.parsedJson
      },
      expectedOutcome: 'Human concierge notified'
    },
    context
  );

  return {
    message: "I've connected you with our expert concierge team who will provic
    metadata: {
      intent: context.request.parsedJson.intent,
      confidence: context.request.parsedJson.confidence,
      needsHuman: true
    },
    internalNotes: `Escalated: ${reason}`
  };
}

private async handleError(
    context: AgentContext,
    error: Error
  ): Promise<void> {
    await db.collection('agent_errors').add({
      srId: context.request.id,
      error: error.message,
      stack: error.stack,
      context: {
        iteration: context.iteration,
        toolsUsed: context.toolsUsed
      },
      timestamp: new Date()
    });
  }
}
```

🏗️ PHASE 2: UNIFIED SYSTEM INTEGRATION (Week 1-2)

Step 2.1: New Unified Chat Route

```
// src/app/api/chat/route.ts - Unified Chat Endpoint
import { NextRequest, NextResponse } from 'next/server';
import { AsteriaAgentLoop } from '@/lib/agent';
import { validateRequest, sanitizeInput } from '@/lib/utils/validation';
import { createRateLimiter } from '@/lib/utils/rate-limiter';
import { trackMetrics } from '@/lib/utils/metrics';
import { getServerSession } from 'next-auth';
import { authOptions } from '@/lib/auth';

// Initialize agent and rate limiter
const agent = new AsteriaAgentLoop();
const rateLimiter = createRateLimiter({
  windowMs: 60000, // 1 minute
  maxRequests: 10  // 10 requests per minute per user
```

```
});

export async function POST(request: NextRequest) {
  const startTime = Date.now();

  try {
    // Get session if authenticated
    const session = await getServerSession(authOptions);
    const memberId = session?.user?.id || 'ANON';

    // Parse and validate request
    const body = await request.json();
    const validation = validateRequest(body);

    if (!validation.valid) {
      return NextResponse.json(
        { error: 'Invalid request', details: validation.errors },
        { status: 400 }
      );
    }

    // Rate limiting
    const rateLimitCheck = await rateLimiter.check(memberId);
    if (!rateLimitCheck.allowed) {
      return NextResponse.json(
        {
          error: 'Rate limit exceeded',
          retryAfter: rateLimitCheck.retryAfter
        },
        { status: 429 }
      );
    }

    // Sanitize input
    const message = sanitizeInput(body.message);
    const conversationId = body.conversationId;
```

```
// Process through agent loop
console.log(`📨 Processing request for member: ${memberId}`);
const response = await agent.process(
  message,
  memberId,
  conversationId
);

// Track metrics
await trackMetrics({
  endpoint: '/api/chat',
  memberId,
  duration: Date.now() - startTime,
  intent: response.metadata.intent,
  success: true
});

// Format response for client
const clientResponse = {
  message: response.message,
  conversationId: conversationId || response.metadata.ticketId,
  metadata: {
    intent: response.metadata.intent,
    confidence: response.metadata.confidence,
    suggestions: response.suggestions
  }
};

// Add debug info in development
if (process.env.NODE_ENV === 'development') {
  clientResponse.debug = {
    internalNotes: response.internalNotes,
    processingTime: Date.now() - startTime,
    toolsUsed: response.metadata.toolsUsed
  };
```

```
    }

    return NextResponse.json(clientResponse);

  } catch (error) {
    console.error('❌ Chat endpoint error:', error);

    // Track error metrics
    await trackMetrics({
      endpoint: '/api/chat',
      memberId: 'unknown',
      duration: Date.now() - startTime,
      error: error.message,
      success: false
    });

    // Return user-friendly error
    return NextResponse.json(
      {
        error: 'Service temporarily unavailable',
        message: 'Our concierge team has been notified and will assist you shortly.'
      },
      { status: 500 }
    );
  }
}

// OPTIONS for CORS
export async function OPTIONS(request: NextRequest) {
  return new NextResponse(null, {
    status: 200,
    headers: {
      'Access-Control-Allow-Origin': process.env.NEXT_PUBLIC_APP_URL || '*',
      'Access-Control-Allow-Methods': 'POST, OPTIONS',
      'Access-Control-Allow-Headers': 'Content-Type, Authorization',
      'Access-Control-Max-Age': '86400',
    },
  });
}
```

Step 2.2: Tool Implementations

```
// src/lib/agent/tools/index.ts - Tool Registry
export interface Tool {
  name: string;
  description: string;
  parameters: Record<string, any>;
  execute: (params: any, context: any) => Promise<ToolResult>;
}

export const TOOLS_REGISTRY: Record<string, Tool> = {
  fetch_active_services,
  create_ticket,
  notify_concierge,
  escalate_to_human,
  check_availability,
  generate_quote,
  search_knowledge_base
};

// === Tool 1: Fetch Active Services ===
// src/lib/agent/tools/fetch_active_services.ts
import { db } from '@/lib/firebase/admin';
import { ServiceBucket, ToolResult } from '../types';

export const fetch_active_services: Tool = {
  name: 'fetch_active_services',
  description: 'Fetch available services based on intent and member tier',
  parameters: {
    intent: 'string',
    memberTier: 'string',
    location: 'string?',
    dateRange: 'object?'
  },

  async execute(params: any, context: any): Promise<ToolResult> {
    const startTime = Date.now();

    try {
      // Query active services from Firestore
      let query = db.collection('active_services')
        .where('status', '==', 'active')
        .where('category', '==', params.intent);

      if (params.location) {
        query = query.where('locations', 'array-contains', params.location);
      }

      const snapshot = await query.limit(10).get();
      const services = snapshot.docs.map(doc => ({
        id: doc.id,
        ...doc.data()
      }));

      // Filter by member tier access
      const filteredServices = services.filter(service => {
        const tierRank = { 'GOLD': 1, 'PLATINUM': 2, 'ELITE': 3 };
        const requiredRank = tierRank[service.minimumTier] || 1;
        const memberRank = tierRank[params.memberTier] || 1;
        return memberRank >= requiredRank;
      });

      return {
        success: true,
        data: {
          services: filteredServices,
          count: filteredServices.length,
          filters: params
        },
        duration: Date.now() - startTime
      };

    } catch (error) {
      return {
        success: false,
        error: error.message,
        duration: Date.now() - startTime
      };
    }
  }
 };

 // === Tool 2: Create Ticket ===
 // src/lib/agent/tools/create_ticket.ts
 import { db } from '@/lib/firebase/admin';
 import { ServiceRequest, ToolResult } from '../types';
 import { extractServiceDetails } from '../utils/extractor';
 import { notifySlack } from './slack_integration';

 export const create_ticket: Tool = {
  name: 'create_ticket',
  description: 'Create a service ticket with extracted details',
  parameters: {
    requestId: 'string',
    intent: 'string',
    extractedDetails: 'object',
    priority: 'string'
  },

  async execute(params: any, context: any): Promise<ToolResult> {
    const startTime = Date.now();
```

```
try {
  const ticketId = `TKT-${Date.now().toString().slice(-8)}`;

  // Create ticket document
  const ticket = {
    id: ticketId,
    serviceRequestId: params.requestId,
    memberId: context.member.id,
    intent: params.intent,
    status: 'OPEN',
    priority: params.priority || 'MEDIUM',
    details: params.extractedDetails,
    assignedTo: null,
    createdAt: new Date(),
    updatedAt: new Date()
  };

  await db.collection('tickets').doc(ticketId).set(ticket);

  // Update service request
  await db.collection('service_requests')
    .doc(params.requestId)
    .update({
      ticketId,
      status: 'TICKETED',
      updatedAt: new Date()
    });

  // Notify Slack
  await notifySlack({
    ticketId,
    serviceRequestId: params.requestId,
    intent: params.intent,
    priority: params.priority,
    memberTier: context.member.tier
  });
```

```
      return {
        success: true,
        data: {
          ticketId,
          ticket
        },
        duration: Date.now() - startTime
      };

    } catch (error) {
      return {
        success: false,
        error: error.message,
        duration: Date.now() - startTime
      };
    }
  }
};

// === Tool 3: Notify Concierge ===
// src/lib/agent/tools/notify_concierge.ts
import { ToolResult } from '../types';
import { sendSlackMessage } from '@/lib/services/slack';
import { sendSMS } from '@/lib/services/sms';
import { sendEmail } from '@/lib/services/email';
import { createThrottler } from '@/lib/utils/throttle';

const throttler = createThrottler({
  maxPerHour: 10,
  maxPerDay: 50
});

export const notify_concierge: Tool = {
  name: 'notify_concierge',
  description: 'Notify concierge team through multiple channels',
  parameters: {
    channels: 'array',
    message: 'string',
    priority: 'string',
    attachments: 'object?'
  },

  async execute(params: any, context: any): Promise<ToolResult> {
    const startTime = Date.now();

    try {
      // Check throttling
      const throttleKey = `notify_${context.member.id}`;
      const canNotify = await throttler.check(throttleKey);

      if (!canNotify) {
        return {
          success: false,
          error: 'Notification rate limit exceeded',
          duration: Date.now() - startTime
        };
      }

      const results = [];

      // Send to requested channels
      for (const channel of params.channels) {
        switch (channel) {
          case 'slack':
            const slackResult = await sendSlackMessage({
              channel: params.priority === 'URGENT'
                ? '#urgent-concierge'
                : '#concierge-requests',
              text: params.message,
              blocks: buildSlackBlocks(params, context)
            });
            results.push({ channel: 'slack', success: slackResult.ok });
            break;

          case 'sms':
            if (params.priority === 'URGENT') {
              const smsResult = await sendSMS({
                to: process.env.CONCIERGE_PHONE!,
                body: `URGENT: ${params.message.slice(0, 140)}...`
              });
              results.push({ channel: 'sms', success: smsResult.success });
            }
            break;

          case 'email':
            const emailResult = await sendEmail({
              to: process.env.CONCIERGE_EMAIL!,
              subject: `[${params.priority}] New Concierge Request`,
              html: buildEmailTemplate(params, context)
            });
            results.push({ channel: 'email', success: emailResult.success });
            break;
        }
      }

      return {
        success: results.some(r => r.success),
        data: { results },
        duration: Date.now() - startTime
      };

    } catch (error) {
      return {
        success: false,
        error: error.message,
        duration: Date.now() - startTime
      };
```

```
    }
  }
};

// === Tool 4: Escalate to Human ===
// src/lib/agent/tools/escalate_to_human.ts
export const escalate_to_human: Tool = {
  name: 'escalate_to_human',
  description: 'Escalate complex requests to human concierge',
  parameters: {
    requestId: 'string',
    reason: 'string',
    context: 'object',
    suggestedAgent: 'string?'
  },

  async execute(params: any, context: any): Promise<ToolResult> {
    const startTime = Date.now();

    try {
      // Create escalation record
      const escalation = {
        serviceRequestId: params.requestId,
        memberId: context.member.id,
        reason: params.reason,
        context: params.context,
        suggestedAgent: params.suggestedAgent,
        status: 'PENDING',
        createdAt: new Date()
      };

      const escalationRef = await db.collection('escalations').add(escalation);

      // Update service request
      await db.collection('service_requests')
        .doc(params.requestId)
        .update({
          status: 'ESCALATED',
          escalationId: escalationRef.id,
          updatedAt: new Date()
        });

      // Notify all channels for escalations
      await notify_concierge.execute({
        channels: ['slack', 'sms', 'email'],
        message: `⚠️ Escalation Required\\n\\nRequest: ${params.requestId}\\nRea
        priority: 'HIGH',
        attachments: {
          escalationId: escalationRef.id,
          memberHistory: context.member.history.slice(-5)
        }
      }, context);

      return {
        success: true,
        data: {
          escalationId: escalationRef.id,
          notified: true
        },
        duration: Date.now() - startTime
      };

    } catch (error) {
      return {
        success: false,
        error: error.message,
        duration: Date.now() - startTime
      };
    }
  }
};
```

```
// === Helper Functions ===
function buildSlackBlocks(params: any, context: any): any[] {
  return [
    {
      type: 'header',
      text: {
        type: 'plain_text',
        text: `${params.priority === 'URGENT' ? '🚨' : '📋'} Concierge Notification`
      }
    },
    {
      type: 'section',
      text: {
        type: 'mrkdwn',
        text: params.message
      }
    },
    {
      type: 'context',
      elements: [
        {
          type: 'mrkdwn',
          text: `Member: ${context.member.id} | Tier: ${context.member.tier}`
        }
      ]
    }
  ];
}

function buildEmailTemplate(params: any, context: any): string {
  return `
    <!DOCTYPE html>
    <html>
    <head>
      <style>
        body { font-family: Arial, sans-serif; }
      .container { max-width: 600px; margin: 0 auto; padding: 20px; }
      .header { background: #1a1a1a; color: white; padding: 20px; }
      .content { padding: 20px; background: #f5f5f5; }
      .priority-urgent { color: #ff4444; }
      .priority-high { color: #ff8800; }
      .priority-medium { color: #0088ff; }
     </style>
    </head>
    <body>
     <div class="container">
      <div class="header">
       <h2>Asteria Concierge Notification</h2>
       <p class="priority-${params.priority.toLowerCase()}">${params.priority} F
      </div>
      <div class="content">
       <p>${params.message}</p>
       <hr>
       <p><strong>Member ID:</strong> ${context.member.id}</p>
       <p><strong>Tier:</strong> ${context.member.tier}</p>
       <p><strong>Time:</strong> ${new Date().toISOString()}</p>
      </div>
     </div>
    </body>
    </html>
  `;
}
```

Step 2.3: Core Agent Components

```
// === PLANNER COMPONENT ===
// src/lib/agent/planner.ts
import { OpenAI } from 'openai';
import { AgentContext, PlanStep, ServiceBucket } from './types';
import { AGENT_CONFIG } from './index';
```

```
const openai = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY!
});

export class IntentPlanner {
  async createPlan(context: AgentContext): Promise<{ steps: PlanStep[] }> {
    // First, classify the intent with high accuracy
    const classification = await this.classifyIntent(context.request.rawText);

    // Update context with classification
    context.request.parsedJson = {
      ...context.request.parsedJson,
      intent: classification.intent,
      confidence: classification.confidence,
      entities: classification.entities
    };

    // Generate plan based on intent
    const planPrompt = this.buildPlanPrompt(context, classification);

    const completion = await openai.chat.completions.create({
      model: AGENT_CONFIG.planningModel,
      messages: [
        {
          role: 'system',
          content: `You are an expert luxury concierge planner. Create a step-by-ste
          Available tools: fetch_active_services, create_ticket, notify_concierge, esca
          Return a JSON array of steps with: step, action, tool (optional), parameters
        },
        {
          role: 'user',
          content: planPrompt
        }
      ],
      response_format: { type: 'json_object' },
      temperature: 0.3
    });

    const planData = JSON.parse(completion.choices[0].message.content!);
    return { steps: planData.steps };
  }

  private async classifyIntent(text: string): Promise<{
    intent: ServiceBucket;
    confidence: number;
    entities: any;
  }> {
    const completion = await openai.chat.completions.create({
      model: AGENT_CONFIG.planningModel,
      messages: [
        {
          role: 'system',
          content: `Classify the luxury service request into one of these categories:
          - aviation (private jets, helicopters, charters)
          - dining (restaurants, private chefs, catering)
          - hotel (accommodations, resorts, villas)
          - transportation (cars, yachts, ground transport)
          - experience (events, tours, activities)
          - shopping (personal shopping, rare items)
          - custom (anything else)

          Also extract entities: dates, locations, people count, preferences, budget.
          Return JSON with: intent, confidence (0-1), entities.`
        },
        {
          role: 'user',
          content: text
        }
      ],
      response_format: { type: 'json_object' },
      temperature: 0.1
    });

    return JSON.parse(completion.choices[0].message.content!);
  }

  private buildPlanPrompt(context: AgentContext, classification: any): string {
    const memberInfo = `Member Tier: ${context.member.tier}`;
    const request = context.request.rawText;
    const intent = classification.intent;
    const entities = JSON.stringify(classification.entities, null, 2);

    return `Create a plan for this ${intent} request:

Request: "${request}"
${memberInfo}
Extracted Entities: ${entities}

Previous interactions: ${context.iteration}

Create 3-7 steps that will fulfill this request. Consider:
- Member tier benefits and preferences
- Need for human verification on high-value requests
- Availability checking before confirmation
- Proper notification channels`;
  }
}

// === EXECUTOR COMPONENT ===
// src/lib/agent/executor.ts
import { AgentContext, PlanStep, ToolResult } from './types';
import { TOOLS_REGISTRY } from './tools';

export class ServiceExecutor {
  async executeStep(step: PlanStep, context: AgentContext): Promise<ToolResult
    console.log(`🔧 Executing step ${step.step}: ${step.action}`);
```

```
// If no tool specified, it's a direct action
if (!step.tool) {
  return {
    success: true,
    data: { action: step.action, completed: true },
    duration: 0
  };
}

// Get tool from registry
const tool = TOOLS_REGISTRY[step.tool];
if (!tool) {
  return {
    success: false,
    error: `Tool not found: ${step.tool}`,
    duration: 0
  };
}

// Validate parameters
const params = this.prepareParameters(step.parameters, context);

try {
  // Execute with timeout
  const timeoutMs = AGENT_CONFIG.serviceTimeouts[context.request.parsedJ
  const result = await this.executeWithTimeout(
    tool.execute(params, context),
    timeoutMs
  );

  return result;
} catch (error) {
  return {
    success: false,
    error: error.message,
    duration: 0
```

```
      };
    }
  }

  private prepareParameters(params: any, context: AgentContext): any {
    // Inject context values into parameters
    return {
      ...params,
      memberId: context.member.id,
      memberTier: context.member.tier,
      requestId: context.request.id,
      intent: context.request.parsedJson.intent
    };
  }

  private async executeWithTimeout<T>(
    promise: Promise<T>,
    timeoutMs: number
  ): Promise<T> {
    const timeout = new Promise((_, reject) ⇒
      setTimeout(() ⇒ reject(new Error('Operation timed out')), timeoutMs)
    );

    return Promise.race([promise, timeout]) as Promise<T>;
  }
}

// === REFLECTOR COMPONENT ===
// src/lib/agent/reflector.ts
import { AgentContext, ToolResult, ReflectionInsight } from './types';
import { db } from '@/lib/firebase/admin';

export class InteractionReflector {
  async analyze(
    context: AgentContext,
    results: ToolResult[]
```

```
): Promise<ReflectionInsight[]> {
  const insights: ReflectionInsight[] = [];

  // Analyze tool usage patterns
  const toolUsageInsight = this.analyzeToolUsage(context, results);
  if (toolUsageInsight) insights.push(toolUsageInsight);

  // Analyze member patterns
  const memberInsight = await this.analyzeMemberPatterns(context);
  if (memberInsight) insights.push(memberInsight);

  // Analyze service-specific patterns
  const serviceInsight = this.analyzeServicePatterns(context, results);
  if (serviceInsight) insights.push(serviceInsight);

  // Store insights for learning
  if (AGENT_CONFIG.enableLearning && insights.length > 0) {
    await this.storeInsights(context.request.id, insights);
  }

  return insights;
}

private analyzeToolUsage(
  context: AgentContext,
  results: ToolResult[]
): ReflectionInsight | null {
  const failedTools = results.filter(r => !r.success);

  if (failedTools.length > 0) {
    return {
      pattern: `Tool failures detected: ${failedTools.length}/${results.length}`,
      frequency: failedTools.length,
      recommendation: 'Consider alternative tools or manual escalation',
      impact: 'MEDIUM'
    };
  }

  const avgDuration = results.reduce((acc, r) ⇒ acc + r.duration, 0) / results.leng
  if (avgDuration > 5000) {
    return {
      pattern: 'Slow tool execution detected',
      frequency: 1,
      recommendation: 'Optimize tool parameters or implement caching',
      impact: 'LOW'
    };
  }

  return null;
}

private async analyzeMemberPatterns(
  context: AgentContext
): Promise<ReflectionInsight | null> {
  // Query recent requests from this member
  const recentRequests = await db
    .collection('service_requests')
    .where('memberId', '==', context.member.id)
    .orderBy('createdAt', 'desc')
    .limit(10)
    .get();

  const intents = recentRequests.docs.map(doc ⇒
    doc.data().parsedJson?.intent
  ).filter(Boolean);

  // Find most common intent
  const intentCounts = intents.reduce((acc, intent) ⇒ {
    acc[intent] = (acc[intent] || 0) + 1;
    return acc;
  }, {} as Record<string, number>);
  const topIntent = Object.entries(intentCounts)
    .sort(([,a], [,b]) => b - a)[0];

  if (topIntent && topIntent[1] > 3) {
    return {
      pattern: `Member frequently requests ${topIntent[0]} services`,
      frequency: topIntent[1],
      recommendation: `Pre-populate ${topIntent[0]} preferences for faster servic
      impact: 'MEDIUM'
    };
  }

  return null;
}

private analyzeServicePatterns(
  context: AgentContext,
  results: ToolResult[]
): ReflectionInsight | null {
  const intent = context.request.parsedJson.intent;

  // Service-specific patterns
  switch (intent) {
    case 'aviation':
      if (context.request.rawText.toLowerCase().includes('urgent') ||
          context.request.rawText.toLowerCase().includes('asap')) {
        return {
          pattern: 'Urgent aviation request detected',
          frequency: 1,
          recommendation: 'Prioritize availability checking and enable fast-track bo
          impact: 'HIGH'
        };
      }
      break;

    case 'dining':
      const partySize = context.request.parsedJson.entities?.people?.length;
      if (partySize && partySize > 6) {
        return {
          pattern: 'Large party dining request',
          frequency: 1,
          recommendation: 'Focus on venues with private dining options',
          impact: 'MEDIUM'
        };
      }
      break;

    case 'hotel':
      const hasOneRoof = context.request.rawText.toLowerCase().includes('one r
                        context.request.rawText.toLowerCase().includes('same property'
      if (hasOneRoof) {
        return {
          pattern: 'One-roof accommodation requirement',
          frequency: 1,
          recommendation: 'Filter hotels by full-service capabilities first',
          impact: 'HIGH'
        };
      }
      break;
  }

  return null;
}

private async storeInsights(
  requestId: string,
  insights: ReflectionInsight[]
): Promise<void> {
  const batch = db.batch();

  for (const insight of insights) {
    const ref = db.collection('agent_insights').doc();
```

```
      batch.set(ref, {
        requestId,
        ...insight,
        createdAt: new Date()
      });
    }

    await batch.commit();
  }
}

// === GOAL CHECKER COMPONENT ===
// src/lib/agent/goal_checker.ts
import { AgentContext, ToolResult, GoalValidation } from './types';

export class GoalChecker {
  async validate(
    context: AgentContext,
    results: ToolResult[]
  ): Promise<GoalValidation> {
    const intent = context.request.parsedJson.intent;
    const goals = this.defineGoals(intent, context);

    let achievedCount = 0;
    const missingElements: string[] = [];
    const recommendations: string[] = [];

    // Check each goal
    for (const goal of goals) {
      const achieved = this.checkGoal(goal, context, results);
      if (achieved) {
        achievedCount++;
      } else {
        missingElements.push(goal.description);
        if (goal.recommendation) {
          recommendations.push(goal.recommendation);
        }
      }
    }

    const score = achievedCount / goals.length;

    return {
      achieved: score >= 0.8, // 80% threshold
      score,
      missingElements: missingElements.length > 0 ? missingElements : undefined,
      recommendations: recommendations.length > 0 ? recommendations : undefir
    };
  }

  private defineGoals(intent: ServiceBucket, context: AgentContext): Goal[] {
    const baseGoals: Goal[] = [
      {
        id: 'intent_classified',
        description: 'Service intent clearly identified',
        check: () ⇒ context.request.parsedJson.confidence > 0.7
      },
      {
        id: 'ticket_created',
        description: 'Service ticket created',
        check: (results) ⇒ results.some(r ⇒ r.data?.ticketId),
        recommendation: 'Create service ticket for tracking'
      }
    ];

    // Add intent-specific goals
    switch (intent) {
      case 'aviation':
        baseGoals.push({
          id: 'flight_details',
          description: 'Flight details captured (dates, route, passengers)',
          check: () ⇒ {
```

```
        const entities = context.request.parsedJson.entities;
        return !!(entities.dates && entities.locations && entities.people);
      },
      recommendation: 'Gather flight dates, destinations, and passenger count'
    });
    break;

  case 'dining':
    baseGoals.push({
      id: 'dining_preferences',
      description: 'Dining preferences captured',
      check: () ⇒ {
        const entities = context.request.parsedJson.entities;
        return !!(entities.dates && (entities.preferences || entities.special_requests
      },
      recommendation: 'Confirm date, time, party size, and dietary preferences'
    });
    break;

  case 'hotel':
    baseGoals.push({
      id: 'accommodation_details',
      description: 'Accommodation requirements captured',
      check: () ⇒ {
        const entities = context.request.parsedJson.entities;
        return !!(entities.dates && entities.locations);
      },
      recommendation: 'Specify check-in/out dates and preferred location'
    });

    if (context.request.parsedJson.constraints?.oneRoofRequired) {
      baseGoals.push({
        id: 'one_roof_validation',
        description: 'One-roof capabilities verified',
        check: (results) ⇒ results.some(r ⇒
          r.data?.services?.some((s: any) ⇒ s.hasFullService)
        ),
        recommendation: 'Verify hotel can provide all requested services on-prop
      });
    }
    break;
  }

  return baseGoals;
 }

 private checkGoal(
  goal: Goal,
  context: AgentContext,
  results: ToolResult[]
 ): boolean {
  try {
   return goal.check(results, context);
  } catch {
   return false;
  }
 }
}

interface Goal {
 id: string;
 description: string;
 check: (results?: ToolResult[], context?: AgentContext) ⇒ boolean;
 recommendation?: string;
}
```

🏗️ PHASE 3: FIREBASE & INFRASTRUCTURE SETUP (Week 2) 🇲🇦

Step 3.1: Firebase Configuration & Setup

```
// === FIREBASE ADMIN SETUP ===
// src/lib/firebase/admin.ts
import { initializeApp, cert, getApps } from 'firebase-admin/app';
import { getFirestore } from 'firebase-admin/firestore';
import { getAuth } from 'firebase-admin/auth';
import { getStorage } from 'firebase-admin/storage';

// Initialize Firebase Admin
const activeApps = getApps();
const app = activeApps.length === 0 ? initializeApp({
  credential: cert({
    projectId: process.env.FIREBASE_PROJECT_ID,
    privateKey: process.env.FIREBASE_PRIVATE_KEY?.replace(/\\n/g, '\n'),
    clientEmail: process.env.FIREBASE_CLIENT_EMAIL,
  }),
  storageBucket: process.env.FIREBASE_STORAGE_BUCKET,
}) : activeApps[0];

export const db = getFirestore(app);
export const auth = getAuth(app);
export const storage = getStorage(app);

// Configure Firestore settings
db.settings({
  timestampsInSnapshots: true,
  ignoreUndefinedProperties: true,
});

// === FIREBASE CLIENT SETUP ===
// src/lib/firebase/client.ts
import { initializeApp, getApps } from 'firebase/app';
import { getAuth } from 'firebase/auth';
import { getFirestore } from 'firebase/firestore';
import { getStorage } from 'firebase/storage';
```

```
const firebaseConfig = {
  apiKey: process.env.NEXT_PUBLIC_FIREBASE_API_KEY,
  authDomain: process.env.NEXT_PUBLIC_FIREBASE_AUTH_DOMAIN,
  projectId: process.env.NEXT_PUBLIC_FIREBASE_PROJECT_ID,
  storageBucket: process.env.NEXT_PUBLIC_FIREBASE_STORAGE_BUCKET,
  messagingSenderId: process.env.NEXT_PUBLIC_FIREBASE_MESSAGING_SEND
  appId: process.env.NEXT_PUBLIC_FIREBASE_APP_ID,
};

// Initialize Firebase
const app = getApps().length === 0 ? initializeApp(firebaseConfig) : getApps()[0

export const clientAuth = getAuth(app);
export const clientDb = getFirestore(app);
export const clientStorage = getStorage(app);

// === FIRESTORE SCHEMA SETUP ===
// scripts/setup-firestore.ts
import { db } from '../src/lib/firebase/admin';
import { FieldValue } from 'firebase-admin/firestore';

async function setupCollections() {
  console.log('🔧 Setting up Firestore collections...');

  // Create collections with sample documents to establish schema
  const collections = [
    {
      name: 'service_requests',
      sampleDoc: {
        id: 'SR-SAMPLE',
        memberId: 'MEMBER-SAMPLE',
        rawText: 'Sample request',
        parsedJson: {
          intent: 'custom',
          confidence: 0.0,
          entities: {},
          preferences: {},
          constraints: {}
        },
        status: 'NEW',
        priority: 'MEDIUM',
        createdAt: FieldValue.serverTimestamp(),
        updatedAt: FieldValue.serverTimestamp(),
        conversationHistory: []
      }
    },
    {
      name: 'members',
      sampleDoc: {
        id: 'MEMBER-SAMPLE',
        tier: 'GOLD',
        preferences: {
          communication: 'email',
          brands: [],
          dietary: [],
          interests: []
        },
        history: [],
        totalSpend: 0,
        createdAt: FieldValue.serverTimestamp()
      }
    },
    {
      name: 'tickets',
      sampleDoc: {
        id: 'TKT-SAMPLE',
        serviceRequestId: 'SR-SAMPLE',
        memberId: 'MEMBER-SAMPLE',
        intent: 'custom',
        status: 'OPEN',
        priority: 'MEDIUM',
        details: {},
        assignedTo: null,
        createdAt: FieldValue.serverTimestamp(),
        updatedAt: FieldValue.serverTimestamp()
      }
    },
    {
      name: 'active_services',
      sampleDoc: {
        id: 'SERVICE-SAMPLE',
        name: 'Sample Service',
        category: 'custom',
        status: 'active',
        minimumTier: 'GOLD',
        locations: [],
        pricing: {},
        availability: {},
        createdAt: FieldValue.serverTimestamp()
      }
    },
    {
      name: 'agent_insights',
      sampleDoc: {
        requestId: 'SR-SAMPLE',
        pattern: 'Sample pattern',
        frequency: 1,
        recommendation: 'Sample recommendation',
        impact: 'LOW',
        createdAt: FieldValue.serverTimestamp()
      }
    },
    {
      name: 'sr_actions',
      sampleDoc: {
        srId: 'SR-SAMPLE',
        action: 'sample_action',
        tool: 'sample_tool',
        result: 'success',
        duration: 0,
        timestamp: FieldValue.serverTimestamp(),
        data: {},
        error: null
      }
    }
  ];

  for (const collection of collections) {
    try {
      await db.collection(collection.name)
        .doc('_schema')
        .set(collection.sampleDoc);
      console.log(`✅ Created collection: ${collection.name}`);
    } catch (error) {
      console.error(`❌ Error creating ${collection.name}:`, error);
    }
  }

  console.log('🎉 Firestore setup complete!');
}

// === FIRESTORE INDEXES ===
// firestore.indexes.json
const firestoreIndexes = {
  "indexes": [
    {
      "collectionGroup": "service_requests",
      "queryScope": "COLLECTION",
      "fields": [
        { "fieldPath": "memberId", "order": "ASCENDING" },
        { "fieldPath": "createdAt", "order": "DESCENDING" }
      ]
    },
    {
      "collectionGroup": "service_requests",
      "queryScope": "COLLECTION",
      "fields": [
        { "fieldPath": "status", "order": "ASCENDING" },
        { "fieldPath": "priority", "order": "ASCENDING" },
        { "fieldPath": "createdAt", "order": "DESCENDING" }
      ]
    },
    {
      "collectionGroup": "tickets",
      "queryScope": "COLLECTION",
      "fields": [
        { "fieldPath": "status", "order": "ASCENDING" },
        { "fieldPath": "assignedTo", "order": "ASCENDING" },
        { "fieldPath": "createdAt", "order": "DESCENDING" }
      ]
    },
    {
      "collectionGroup": "active_services",
      "queryScope": "COLLECTION",
      "fields": [
        { "fieldPath": "category", "order": "ASCENDING" },
        { "fieldPath": "status", "order": "ASCENDING" },
        { "fieldPath": "minimumTier", "order": "ASCENDING" }
      ]
    },
    {
      "collectionGroup": "agent_insights",
      "queryScope": "COLLECTION",
      "fields": [
        { "fieldPath": "impact", "order": "DESCENDING" },
        { "fieldPath": "frequency", "order": "DESCENDING" },
        { "fieldPath": "createdAt", "order": "DESCENDING" }
      ]
    }
```

```
  ],
  "fieldOverrides": []
};

// === FIRESTORE SECURITY RULES ===
// firestore.rules
const firestoreRules = `
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    // Helper functions
    function isAuthenticated() {
      return request.auth != null;
    }

    function isConcierge() {
      return isAuthenticated() &&
        request.auth.token.role == 'concierge';
    }

    function isMember(memberId) {
      return isAuthenticated() &&
        request.auth.uid == memberId;
    }

    function isAdmin() {
      return isAuthenticated() &&
        request.auth.token.role == 'admin';
    }

    // Service Requests
    match /service_requests/{document} {
      allow create: if true; // Anyone can create a request
      allow read: if isConcierge() ||
        (isAuthenticated() && resource.data.memberId == request.auth.uid);
      allow update: if isConcierge();
      allow delete: if isAdmin();
    }

    // Members
    match /members/{memberId} {
      allow read: if isMember(memberId) || isConcierge();
      allow create: if isAuthenticated();
      allow update: if isMember(memberId) || isConcierge();
      allow delete: if isAdmin();
    }

    // Tickets
    match /tickets/{document} {
      allow read: if isConcierge() ||
        (isAuthenticated() && resource.data.memberId == request.auth.uid);
      allow create, update: if isConcierge();
      allow delete: if isAdmin();
    }

    // Active Services
    match /active_services/{document} {
      allow read: if true; // Public read
      allow write: if isConcierge();
    }

    // Agent Insights
    match /agent_insights/{document} {
      allow read: if isConcierge();
      allow write: if false; // Only server can write
    }

    // SR Actions (Audit Log)
    match /sr_actions/{document} {
      allow read: if isConcierge();
      allow write: if false; // Only server can write
    }
```

```
    // Escalations
    match /escalations/{document} {
      allow read: if isConcierge();
      allow create: if true; // Agent can escalate
      allow update: if isConcierge();
      allow delete: if isAdmin();
    }
  }
}
`;

// === ENVIRONMENT VARIABLES TEMPLATE ===
// .env.local
const envTemplate = `
# Firebase Admin SDK (Server-side)
FIREBASE_PROJECT_ID=thriveachievegrow
FIREBASE_PRIVATE_KEY="-----BEGIN PRIVATE KEY-----\\n...\\n-----END PRIVAT
FIREBASE_CLIENT_EMAIL=firebase-adminsdk-xxxxx@thriveachievegrow.iam.gs

# Firebase Client SDK (Public)
NEXT_PUBLIC_FIREBASE_API_KEY=AIzaSyXXXXXXXXXXXXXXXXXXXXXXXXXXX
NEXT_PUBLIC_FIREBASE_AUTH_DOMAIN=thriveachievegrow.firebaseapp.com
NEXT_PUBLIC_FIREBASE_PROJECT_ID=thriveachievegrow
NEXT_PUBLIC_FIREBASE_STORAGE_BUCKET=thriveachievegrow.appspot.com
NEXT_PUBLIC_FIREBASE_MESSAGING_SENDER_ID=123456789012
NEXT_PUBLIC_FIREBASE_APP_ID=1:123456789012:web:abcdef123456

# OpenAI
OPENAI_API_KEY=sk-XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

# Slack
SLACK_BOT_TOKEN=xoxb-XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
SLACK_SIGNING_SECRET=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
SLACK_CHANNEL_ID=C05XXXXXXXXX
SLACK_URGENT_CHANNEL_ID=C06XXXXXXXXX
```

```
# SendGrid
SENDGRID_API_KEY=SG.XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
SENDGRID_FROM_EMAIL=concierge@thriveachievegrow.com

# Twilio
TWILIO_ACCOUNT_SID=ACXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
TWILIO_AUTH_TOKEN=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
TWILIO_PHONE_NUMBER=+12345678901
CONCIERGE_PHONE=+19876543210

# Concierge Settings
CONCIERGE_EMAIL=concierge@thriveachievegrow.com
NEXT_PUBLIC_APP_URL=https://innercircle.thriveachievegrow.com

# Development
NODE_ENV=development
`;

// Export setup function
export { setupCollections, firestoreIndexes, firestoreRules, envTemplate };
```

Step 3.2: Slack Integration & Cloud Functions

```
// === SLACK SERVICE INTEGRATION ===
// src/lib/services/slack.ts
import { WebClient } from '@slack/web-api';
import { db } from '@/lib/firebase/admin';

const slack = new WebClient(process.env.SLACK_BOT_TOKEN);

export interface SlackMessage {
  channel: string;
  text: string;
  blocks?: any[];
  thread_ts?: string;
}

export async function sendSlackMessage(message: SlackMessage) {
  try {
    const result = await slack.chat.postMessage({
      channel: message.channel,
      text: message.text,
      blocks: message.blocks,
      thread_ts: message.thread_ts,
    });

    return { success: true, ts: result.ts };
  } catch (error) {
    console.error('Slack message error:', error);
    return { success: false, error: error.message };
  }
}

export async function notifySlack(params: {
  ticketId: string;
  serviceRequestId: string;
  intent: string;
  priority: string;
  memberTier: string;
}) {
  const blocks = [
    {
      type: 'header',
      text: {
        type: 'plain_text',
        text: `🎫 New ${params.priority} Priority Ticket`
      }
    },
    {
      type: 'section',
      fields: [
        {
          type: 'mrkdwn',
          text: `*Ticket:*\\n${params.ticketId}`
        },
        {
          type: 'mrkdwn',
          text: `*Request:*\\n${params.serviceRequestId}`
        },
        {
          type: 'mrkdwn',
          text: `*Service:*\\n${params.intent}`
        },
        {
          type: 'mrkdwn',
          text: `*Member Tier:*\\n${params.memberTier}`
        }
      ]
    },
    {
      type: 'actions',
      elements: [
        {
          type: 'button',
          text: {
            type: 'plain_text',
            text: 'Claim Ticket'
          },
          style: 'primary',
          action_id: 'claim_ticket',
          value: params.ticketId
        },
        {
          type: 'button',
          text: {
            type: 'plain_text',
            text: 'View Details'
          },
          action_id: 'view_details',
          value: params.serviceRequestId
        },
        {
          type: 'button',
          text: {
            type: 'plain_text',
            text: 'Mark Urgent'
          },
          style: 'danger',
          action_id: 'mark_urgent',
          value: params.ticketId
        }
      ]
    }
  ];

  const channel = params.priority === 'URGENT'
    ? process.env.SLACK_URGENT_CHANNEL_ID!
    : process.env.SLACK_CHANNEL_ID!;

  return sendSlackMessage({
    channel,
    text: `New ${params.priority} ticket: ${params.ticketId}`,
    blocks
  });
}

// === CLOUD FUNCTIONS ===
// functions/src/index.ts
import * as functions from 'firebase-functions/v2';
import * as admin from 'firebase-admin';
import { WebClient } from '@slack/web-api';
import * as sgMail from '@sendgrid/mail';
import * as twilio from 'twilio';

// Initialize services
admin.initializeApp();
const db = admin.firestore();
const slack = new WebClient(process.env.SLACK_BOT_TOKEN);
sgMail.setApiKey(process.env.SENDGRID_API_KEY!);
const twilioClient = twilio(
  process.env.TWILIO_ACCOUNT_SID,
  process.env.TWILIO_AUTH_TOKEN
);

// === CALLABLE FUNCTION: Create Service Request ===
export const createServiceRequest = functions.https.onCall(
  {
    region: 'us-central1',
    memory: '512MiB',
    timeoutSeconds: 60,
  },
  async (data, context) ⇒ {
    // Generate SR ID
    const srId = `SR-${Date.now().toString().slice(-6)}`;

    try {
      // Create service request
      await db.collection('service_requests').doc(srId).set({
        id: srId,
        memberId: data.memberId || 'ANON',
        rawText: data.text,
        parsedJson: {
          intent: 'custom',
          confidence: 0,
          entities: {},
          preferences: {},
          constraints: {}
        },
        status: 'NEW',
        priority: 'MEDIUM',
        createdAt: admin.firestore.FieldValue.serverTimestamp(),
        updatedAt: admin.firestore.FieldValue.serverTimestamp(),
        conversationHistory: [{
          role: 'user',
          content: data.text,
          timestamp: new Date()
        }]
      });

      // Send Slack notification
      await slack.chat.postMessage({
        channel: process.env.SLACK_CHANNEL_ID!,
        text: `New service request: ${srId}`,
        blocks: buildSlackBlocks(srId, data.text, data.memberId)
      });

      return { success: true, srId };

    } catch (error) {
      console.error('Error creating service request:', error);
      throw new functions.https.HttpsError(
        'internal',
        'Failed to create service request'
      );
    }
  }
);

// === HTTP FUNCTION: Slack Interactivity ===
export const slackInteractivity = functions.https.onRequest(
  {
    region: 'us-central1',
    memory: '256MiB',
    timeoutSeconds: 30,
  },
  async (req, res) => {
    // Verify Slack signature
    const signature = req.headers['x-slack-signature'] as string;
    const timestamp = req.headers['x-slack-request-timestamp'] as string;

    if (!verifySlackSignature(signature, timestamp, req.rawBody)) {
      return res.status(401).send('Unauthorized');
    }

    // Parse payload
    const payload = JSON.parse(req.body.payload);
    const actionId = payload.actions[0].action_id;
    const value = payload.actions[0].value;
    const userId = payload.user.id;

    // Handle different actions
    switch (actionId) {
      case 'claim_ticket':
        await handleClaimTicket(value, userId);
        break;
      case 'view_details':
        await handleViewDetails(value, userId, payload.response_url);
        break;
      case 'mark_urgent':
        await handleMarkUrgent(value, userId);
        break;
    }

    // Acknowledge immediately
    res.status(200).send();
  }
);
```

```
// === FIRESTORE TRIGGER: Status Change Notifications ===
export const onServiceRequestUpdate = functions.firestore
  .onDocumentUpdated(
    {
      document: 'service_requests/{srId}',
      region: 'us-central1',
    },
    async (event) => {
      const before = event.data?.before.data();
      const after = event.data?.after.data();

      if (!before || !after) return;

      // Check for status changes
      if (before.status !== after.status) {
        await handleStatusChange(
          event.params.srId,
          before.status,
          after.status,
          after
        );
      }
    }
  );

// === SCHEDULED FUNCTION: Daily Metrics ===
export const dailyMetrics = functions.scheduler
  .onSchedule({
    schedule: 'every day 09:00',
    timeZone: 'America/Los_Angeles',
    region: 'us-central1',
  })
  .onRun(async (context) => {
    const yesterday = new Date();
    yesterday.setDate(yesterday.getDate() - 1);
    yesterday.setHours(0, 0, 0, 0);

    const today = new Date();
    today.setHours(0, 0, 0, 0);

    // Query metrics
    const requests = await db.collection('service_requests')
      .where('createdAt', '>=', yesterday)
      .where('createdAt', '<', today)
      .get();

    const metrics = {
      totalRequests: requests.size,
      byStatus: {},
      byIntent: {},
      avgResponseTime: 0,
    };

    // Calculate metrics
    requests.docs.forEach(doc => {
      const data = doc.data();
      metrics.byStatus[data.status] = (metrics.byStatus[data.status] || 0) + 1;
      metrics.byIntent[data.parsedJson?.intent || 'unknown'] =
        (metrics.byIntent[data.parsedJson?.intent || 'unknown'] || 0) + 1;
    });

    // Send daily report
    await sendDailyReport(metrics);
  });

// === Helper Functions ===
function verifySlackSignature(
  signature: string,
  timestamp: string,
  body: string
): boolean {
  const crypto = require('crypto');
  const signingSecret = process.env.SLACK_SIGNING_SECRET!;

  const baseString = `v0:${timestamp}:${body}`;
  const hmac = crypto
    .createHmac('sha256', signingSecret)
    .update(baseString)
    .digest('hex');
  const computedSignature = `v0=${hmac}`;

  return crypto.timingSafeEqual(
    Buffer.from(signature),
    Buffer.from(computedSignature)
  );
}

async function handleClaimTicket(ticketId: string, userId: string) {
  await db.collection('tickets').doc(ticketId).update({
    assignedTo: userId,
    status: 'IN_PROGRESS',
    updatedAt: admin.firestore.FieldValue.serverTimestamp()
  });

  // Update Slack message
  await slack.chat.postMessage({
    channel: process.env.SLACK_CHANNEL_ID!,
    text: `Ticket ${ticketId} claimed by <@${userId}>`
  });
}

async function handleViewDetails(
  srId: string,
  userId: string,
  responseUrl: string
) {
  const doc = await db.collection('service_requests').doc(srId).get();
  const data = doc.data();

  if (!data) return;

  // Send ephemeral message with details
  await fetch(responseUrl, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      response_type: 'ephemeral',
      text: 'Service Request Details',
      blocks: [
        {
          type: 'section',
          text: {
            type: 'mrkdwn',
            text: `*Request:* ${data.rawText}\\n*Status:* ${data.status}\\n*Member:*
          }
        }
      ]
    })
  });
}

async function handleMarkUrgent(ticketId: string, userId: string) {
  await db.collection('tickets').doc(ticketId).update({
    priority: 'URGENT',
    updatedAt: admin.firestore.FieldValue.serverTimestamp()
  });

  // Move to urgent channel
  await slack.chat.postMessage({
    channel: process.env.SLACK_URGENT_CHANNEL_ID!,
    text: `🚨 Ticket ${ticketId} marked as URGENT by <@${userId}>`
  });
}
```

```
async function handleStatusChange(
  srId: string,
  oldStatus: string,
  newStatus: string,
  data: any
) {
  // Send appropriate notifications based on status change
  if (newStatus === 'CONFIRMED') {
    // Get member details
    const memberDoc = await db.collection('members').doc(data.memberId).get()
    const member = memberDoc.data();

    if (member?.preferences?.communication === 'email') {
      await sgMail.send({
        to: member.email,
        from: process.env.SENDGRID_FROM_EMAIL!,
        subject: 'Your Asteria Request is Confirmed',
        html: buildConfirmationEmail(srId, data)
      });
    } else if (member?.preferences?.communication === 'sms') {
      await twilioClient.messages.create({
        to: member.phone,
        from: process.env.TWILIO_PHONE_NUMBER!,
        body: `Your Asteria request ${srId} has been confirmed! Our concierge tean
      });
    }
  }
}

async function sendDailyReport(metrics: any) {
  const blocks = [
    {
      type: 'header',
      text: {
        type: 'plain_text',
        text: '📊 Daily Asteria Metrics'
      }
    },
    {
      type: 'section',
      text: {
        type: 'mrkdwn',
        text: `*Total Requests:* ${metrics.totalRequests}`
      }
    },
    {
      type: 'section',
      fields: Object.entries(metrics.byStatus).map(([status, count]) => ({
        type: 'mrkdwn',
        text: `*${status}:* ${count}`
      }))
    }
  ];

  await slack.chat.postMessage({
    channel: process.env.SLACK_CHANNEL_ID!,
    text: 'Daily Metrics Report',
    blocks
  });
}

function buildSlackBlocks(srId: string, text: string, memberId: string): any[] {
  return [
    {
      type: 'header',
      text: {
        type: 'plain_text',
        text: `🆕 Service Request ${srId}`
      }
    },
    {
      type: 'section',
      text: {
        type: 'mrkdwn',
        text: `*Request:* ${text}\\n*Member:* ${memberId}`
      }
    },
    {
      type: 'actions',
      elements: [
        {
          type: 'button',
          text: { type: 'plain_text', text: 'Process Request' },
          style: 'primary',
          action_id: 'process_request',
          value: srId
        }
      ]
    }
  ];
}

function buildConfirmationEmail(srId: string, data: any): string {
  return `
    <!DOCTYPE html>
    <html>
    <head>
      <style>
        body { font-family: 'Arial', sans-serif; line-height: 1.6; color: #333; }
        .container { max-width: 600px; margin: 0 auto; padding: 20px; }
        .header { background: #1a1a1a; color: white; padding: 30px; text-align: cente
        .content { padding: 30px; background: #f9f9f9; }
        .footer { text-align: center; padding: 20px; color: #666; }
      </style>
    </head>
    <body>
      <div class="container">
        <div class="header">
          <h1>Your Request is Confirmed</h1>
          <p>Request ID: ${srId}</p>
        </div>
        <div class="content">
          <h2>What's Next?</h2>
          <p>Our expert concierge team is now working on your request. You can ex
          <ul>
            <li>Initial contact within 2 hours</li>
            <li>Detailed proposals within 24 hours</li>
            <li>Continuous updates on your request status</li>
          </ul>
          <p>Your request: "${data.rawText}"</p>
        </div>
        <div class="footer">
          <p>Thank you for choosing Asteria Concierge</p>
        </div>
      </div>
    </body>
    </html>
  `;
}
```

Step 3.3: RAG Knowledge Base Implementation

```
// === VECTOR DATABASE SETUP ===
// scripts/setup-vector-db.sql
/*
-- Run this in your PostgreSQL instance with pgvector extension
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE IF NOT EXISTS knowledge_chunks (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  doc_id TEXT NOT NULL,
  chunk_index INTEGER NOT NULL,
  content TEXT NOT NULL,
  embedding vector(1536),
  metadata JSONB,
  source_type TEXT, -- 'hotel_pdf', 'service_doc', 'historical_request', 'policy'
  source_url TEXT,
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW(),
  UNIQUE(doc_id, chunk_index)
);

CREATE INDEX idx_knowledge_chunks_embedding
  ON knowledge_chunks
  USING hnsw (embedding vector_l2_ops);

CREATE INDEX idx_knowledge_chunks_source_type
  ON knowledge_chunks(source_type);

CREATE TABLE IF NOT EXISTS knowledge_documents (
  doc_id TEXT PRIMARY KEY,
  title TEXT,
  source_type TEXT,
  source_url TEXT,
  total_chunks INTEGER,
  metadata JSONB,
  last_indexed TIMESTAMP,
  created_at TIMESTAMP DEFAULT NOW()
);
*/

// === RAG SERVICE ===
// src/lib/rag/service.ts
import { OpenAI } from 'openai';
import { Pool } from 'pg';
import { RecursiveCharacterTextSplitter } from 'langchain/text_splitter';
import * as pdfParse from 'pdf-parse';
import { storage } from '@/lib/firebase/admin';

const openai = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY!
});

const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
  ssl: process.env.NODE_ENV === 'production' ? { rejectUnauthorized: false } : fa
});

export class RAGService {
  private splitter: RecursiveCharacterTextSplitter;

  constructor() {
    this.splitter = new RecursiveCharacterTextSplitter({
      chunkSize: 750,
      chunkOverlap: 100,
      separators: ['\\n\\n', '\\n', '. ', ' ', '']
    });
  }

  // === INGESTION ===
  async ingestDocument(
    docId: string,
    content: Buffer | string,
    metadata: {
      title: string;
      sourceType: 'hotel_pdf' | 'service_doc' | 'historical_request' | 'policy';
      sourceUrl?: string;
      additionalMetadata?: Record<string, any>;
    }
  ): Promise<{ success: boolean; chunksCreated: number }> {
    try {
      // Convert to text if PDF
```

```
let text: string;
if (Buffer.isBuffer(content)) {
  const pdfData = await pdfParse(content);
  text = pdfData.text;
} else {
  text = content;
}

// Split into chunks
const chunks = await this.splitter.splitText(text);

// Generate embeddings
const embeddings = await this.generateEmbeddings(chunks);

// Start transaction
const client = await pool.connect();
await client.query('BEGIN');

try {
  // Insert document record
  await client.query(
    `INSERT INTO knowledge_documents
     (doc_id, title, source_type, source_url, total_chunks, metadata, last_indexe
     VALUES ($1, $2, $3, $4, $5, $6, NOW())
     ON CONFLICT (doc_id) DO UPDATE SET
     title = $2, total_chunks = $5, metadata = $6, last_indexed = NOW()`,
    [
      docId,
      metadata.title,
      metadata.sourceType,
      metadata.sourceUrl,
      chunks.length,
      JSON.stringify(metadata.additionalMetadata || {})
    ]
  );
```

```
      // Insert chunks
      for (let i = 0; i < chunks.length; i++) {
        await client.query(
          `INSERT INTO knowledge_chunks
           (doc_id, chunk_index, content, embedding, metadata, source_type, sourc
           VALUES ($1, $2, $3, $4, $5, $6, $7)
           ON CONFLICT (doc_id, chunk_index) DO UPDATE SET
           content = $3, embedding = $4, updated_at = NOW()`,
          [
            docId,
            i,
            chunks[i],
            `[${embeddings[i].join(',')}]`,
            JSON.stringify({
              ...metadata.additionalMetadata,
              chunkPosition: `${i + 1}/${chunks.length}`
            }),
            metadata.sourceType,
            metadata.sourceUrl
          ]
        );
      }

      await client.query('COMMIT');
      client.release();

      console.log(`✅ Ingested ${chunks.length} chunks for document ${docId}`);
      return { success: true, chunksCreated: chunks.length };

    } catch (error) {
      await client.query('ROLLBACK');
      client.release();
      throw error;
    }

  } catch (error) {
    console.error('❌ Ingestion error:', error);
    return { success: false, chunksCreated: 0 };
  }
}

// === RETRIEVAL ===
async retrieve(
  query: string,
  options: {
    k?: number;
    sourceTypes?: string[];
    threshold?: number;
  } = {}
): Promise<RetrievedChunk[]> {
  const { k = 6, sourceTypes, threshold = 0.7 } = options;

  // Generate query embedding
  const queryEmbedding = await this.generateEmbedding(query);

  // Build query
  let sqlQuery = `
    SELECT
      id,
      doc_id,
      chunk_index,
      content,
      1 - (embedding ⇔ $1::vector) as similarity,
      metadata,
      source_type,
      source_url
    FROM knowledge_chunks
    WHERE 1 - (embedding ⇔ $1::vector) > $2
  `;

  const params: any[] = [`[${queryEmbedding.join(',')}]`, threshold];
  if (sourceTypes && sourceTypes.length > 0) {
    sqlQuery += ` AND source_type = ANY($3)`;
    params.push(sourceTypes);
  }

  sqlQuery += ` ORDER BY similarity DESC LIMIT $${params.length + 1}`;
  params.push(k);

  const result = await pool.query(sqlQuery, params);

  return result.rows.map(row => ({
    id: row.id,
    docId: row.doc_id,
    chunkIndex: row.chunk_index,
    content: row.content,
    similarity: row.similarity,
    metadata: row.metadata,
    sourceType: row.source_type,
    sourceUrl: row.source_url
  }));
}

// === SEARCH KNOWLEDGE BASE TOOL ===
async searchKnowledgeBase(
  query: string,
  context: {
    intent?: string;
    memberTier?: string;
  }
): Promise<{
  answer: string;
  sources: RetrievedChunk[];
  confidence: number;
}> {
  // Enhance query with context
  const enhancedQuery = this.enhanceQuery(query, context);
  // Retrieve relevant chunks
  const chunks = await this.retrieve(enhancedQuery, {
    k: 8,
    sourceTypes: this.getRelevantSourceTypes(context.intent)
  });

  if (chunks.length === 0) {
    return {
      answer: "I couldn't find specific information about that in our knowledge bas
      sources: [],
      confidence: 0
    };
  }

  // Generate answer using retrieved context
  const answer = await this.generateAnswer(query, chunks, context);

  // Calculate confidence based on similarity scores
  const avgSimilarity = chunks.reduce((acc, c) => acc + c.similarity, 0) / chunks.

  return {
    answer: answer.text,
    sources: chunks,
    confidence: avgSimilarity
  };
}

// === SPECIALIZED RETRIEVERS ===
async getHotelCapabilities(hotelName: string): Promise<HotelCapabilities> {
  const chunks = await this.retrieve(
    `${hotelName} meeting rooms capacity catering capabilities`,
    {
      sourceTypes: ['hotel_pdf'],
      k: 10
    }
  );

  // Extract structured data from chunks
  const capabilities: HotelCapabilities = {
    hotelName,
    meetingRooms: [],
    cateringOptions: [],
    amenities: [],
    restrictions: []
  };

  // Use GPT to extract structured information
  const extraction = await openai.chat.completions.create({
    model: 'gpt-4o-mini',
    messages: [
      {
        role: 'system',
        content: `Extract hotel capabilities from the provided text chunks.
        Return JSON with: meetingRooms (array of {name, capacity, features}),
        cateringOptions (array of {type, description, dietary}),
        amenities (array), restrictions (array).`
      },
      {
        role: 'user',
        content: chunks.map(c ⇒ c.content).join('\\n\\n')
      }
    ],
    response_format: { type: 'json_object' },
    temperature: 0
  });

  const extracted = JSON.parse(extraction.choices[0].message.content!);
  return { ...capabilities, ...extracted };
}

// === HELPER METHODS ===
private async generateEmbedding(text: string): Promise<number[]> {
  const response = await openai.embeddings.create({
    model: 'text-embedding-3-small',
    input: text
  });
  return response.data[0].embedding;
}

private async generateEmbeddings(texts: string[]): Promise<number[][]> {
  const response = await openai.embeddings.create({
    model: 'text-embedding-3-small',
    input: texts
  });
  return response.data.map(d ⇒ d.embedding);
}

private enhanceQuery(query: string, context: any): string {
  let enhanced = query;

  if (context.intent) {
    enhanced = `${context.intent} service: ${enhanced}`;
  }

  if (context.memberTier === 'ELITE') {
    enhanced += ' premium luxury exclusive VIP';
  }

  return enhanced;
}

private getRelevantSourceTypes(intent?: string): string[] | undefined {
  if (!intent) return undefined;

  const typeMap: Record<string, string[]> = {
    'hotel': ['hotel_pdf', 'policy'],
    'dining': ['service_doc', 'policy'],
      'aviation': ['service_doc', 'policy'],
      'transportation': ['service_doc', 'policy']
    };

    return typeMap[intent] || undefined;
  }

  private async generateAnswer(
    query: string,
    chunks: RetrievedChunk[],
    context: any
  ): Promise<{ text: string; citations: string[] }> {
    const systemPrompt = `You are Asteria, a luxury concierge AI.
    Answer the query using ONLY information from the provided context chunks.
    Cite sources using [1], [2], etc. Be specific and accurate.
    If information is not in the context, say so.`;

    const contextText = chunks.map((chunk, i) =>
      `[${i + 1}] ${chunk.content}`
    ).join('\\n\\n');

    const completion = await openai.chat.completions.create({
      model: 'gpt-4o',
      messages: [
        { role: 'system', content: systemPrompt },
        { role: 'user', content: `Context:\\n${contextText}\\n\\nQuery: ${query}` }
      ],
      temperature: 0.2
    });

    return {
      text: completion.choices[0].message.content!,
      citations: chunks.map(c => c.sourceUrl || c.docId)
    };
  }
}
```

```
// === INGESTION WORKERS ===
// src/lib/rag/workers/pdf-ingester.ts
import { RAGService } from '../service';
import { storage } from '@/lib/firebase/admin';

export class PDFIngestionWorker {
  private ragService: RAGService;

  constructor() {
    this.ragService = new RAGService();
  }

  async ingestHotelPDF(
    bucketPath: string,
    hotelName: string
  ): Promise<void> {
    try {
      // Download from Firebase Storage
      const bucket = storage.bucket();
      const file = bucket.file(bucketPath);
      const [buffer] = await file.download();

      // Generate document ID
      const docId = `hotel_${hotelName.toLowerCase().replace(/\\s+/g, '_')}_${Dat

      // Ingest
      const result = await this.ragService.ingestDocument(
        docId,
        buffer,
        {
          title: `${hotelName} Meeting & Events Guide`,
          sourceType: 'hotel_pdf',
          sourceUrl: `gs://${bucket.name}/${bucketPath}`,
          additionalMetadata: {
            hotelName,
            documentType: 'meeting_guide',
            indexedAt: new Date().toISOString()
          }
        }
      );

      console.log(`✅ Ingested ${hotelName} PDF:`, result);

    } catch (error) {
      console.error(`❌ Failed to ingest ${hotelName} PDF:`, error);
      throw error;
    }
  }

  async ingestHistoricalRequests(): Promise<void> {
    // Ingest confirmed service requests for learning
    const confirmedRequests = await db
      .collection('service_requests')
      .where('status', '==', 'CONFIRMED')
      .orderBy('createdAt', 'desc')
      .limit(100)
      .get();

    for (const doc of confirmedRequests.docs) {
      const data = doc.data();
      const content = `
Service Type: ${data.parsedJson.intent}
Request: ${data.rawText}
Member Tier: ${data.memberTier}
Resolution: ${data.resolution || 'Fulfilled successfully'}
      `.trim();

      await this.ragService.ingestDocument(
        `historical_${doc.id}`,
        content,
        {
          title: `Historical Request - ${data.parsedJson.intent}`,
          sourceType: 'historical_request',
          additionalMetadata: {
            intent: data.parsedJson.intent,
            memberTier: data.memberTier,
            successfulResolution: true
          }
        }
      );
    }
  }
}

// === TYPES ===
interface RetrievedChunk {
  id: string;
  docId: string;
  chunkIndex: number;
  content: string;
  similarity: number;
  metadata: any;
  sourceType: string;
  sourceUrl?: string;
}

interface HotelCapabilities {
  hotelName: string;
  meetingRooms: Array<{
      name: string;
      capacity: number;
      features: string[];
  }>;
  cateringOptions: Array<{
      type: string;
      description: string;
      dietary: string[];
  }>;
  amenities: string[];
  restrictions: string[];
}

// === INTEGRATION WITH AGENT ===
// Add this as a tool in the agent tools registry
export const search_knowledge_base = {
  name: 'search_knowledge_base',
  description: 'Search the RAG knowledge base for service information',
  parameters: {
    query: 'string',
    intent: 'string?',
    memberTier: 'string?'
  },
  async execute(params: any, context: any): Promise<ToolResult> {
    const ragService = new RAGService();
    const startTime = Date.now();

    try {
      const result = await ragService.searchKnowledgeBase(
        params.query,
        {
          intent: params.intent || context.request.parsedJson.intent,
          memberTier: params.memberTier || context.member.tier
        }
      );

      return {
        success: true,
        data: result,
        duration: Date.now() - startTime
      };
    } catch (error) {
      return {
        success: false,
        error: error.message,
        duration: Date.now() - startTime
      };
    }
  }
};
```